Trademark notices

The author has taken every precaution to list the below trademarks mentionned within this book at the time of writing. However the author assumes no responsibility for errors or omissions or for damages that may result from use of information. The author will correct/amend in the next printing any errors found by the readers.

TEX™ is a trademark of the American Mathematical Society.
Microsoft® is a registered trademark of Microsoft Corporation.
UNIX® is a registered trademark in the United States and other countries.
VAX™ and VMS™ are trademarks of Hewlett-Packard.
Xcode, OS x are registered trademarks of Apple Computer Inc.
Adobe Reader® is a registered trademark and PDF™ a trademark of Adobe Systems Incorporated.
SOLARIS® is a registered trademark of Sun Microsystems, Inc. now owned by Oracle Corporation.

License

2nd Preface

Today is the 28th of March 2012 as I'm writing this 2nd preface. Many internal thoughts have been debated but I finally am at peace with the following objectives of this compiler/compiler project named Yac$_2o_2$. It's basicly 12 years of my on / off composing and applied use of this tool. This means that it should be reasonably robust and not throw too many tantrums.

I am releasing Yac$_2o_2$ into the public domain so people can learn from it, extend / maintain it, and study/apply some of its ideas to other software domains. Outside of Yac$_2o_2$ I hope the following principles used will merit your own integration into your own software problem space:

- How to apply non-determinism for optimum outcome evaluations
- Better writing of software similar to Literate programming with multi-media outputs.

My hopes are teaching institutes will pick up on Yac$_2o_2$ and integrate it into their Computer Department curriculum. Institutions writing language specifications like Internet protocols might give this tool a whirl to refine their RFC proposals. To language creators who will create new languages in attempts to simplify, solve, and explore very difficult problems requiring parallel computing of very high order.

In closing this is a draft of ideas applied to language recognition. At the moment it is not Unicode sensitive. Please don't consider this a reason to not use it or to dismiss its effectiveness. This was a design decision back 12 years ago due to the immaturity of operating systems using Unicode and my inexperience in this domain. As Yac$_2o_2$ grows up it should take on this sensitivity. I would be delighted if Yac$_2o_2$ has a very short life compared to the longevity of *Yacc*. Progress in the making... I welcome your suggestions, ommissions and errors found to improve its content.

Dave Bone

Preface

Hello and let me introduce myself. I come from a software development background having many eclectic software interests some of which are real-time embedded systems, device drivers, virtual distributed databases, and applied language theory. My language theory skills started from an ad hoc writing of translators back in the early 1970ies, matured to using recursive descent techniques with regular expressions, finally to writing of my own compiler/compiler. Yac$_2o_2$ was born from practical reasons as there was not a compiler/compiler available to me at the time on the system used, and from a curiosity to learn and adapt language theory to my interests; from this point forward in the text I'll use its nickname O_2.

Here are some of my projects using language theory techniques:

- translator to remap disk allocation of a Job Control Language (JCL) spanning 3 generations of different job controls
- some home grown language preprocessors for Basic, Cobol, and Pascal
- a screen/report language along with its runtime support system independent of device type
- translators for O_2's grammars, syntax directed code, and other internal languages like its linker control language
- a simple database query language on a network type database
- re-target a Pascal source code variant into another source compliant to another Pascal compiler
- network database schema translator to define and to create equivalent relational databases, and to generate a Pascal program per database that exports its tables's data into the relatonal database's import utility format
- data-mining a database using a Levenshtein minimum distance algorithm

Enough parading of medals Dave! I'm not trying to toot my horn but only to mention some broad range of systems that I touched using language theory. Of course the Internet's plethora of protocols invisibly allows u to roam around by use of language theory and ditto to program runtime load linkers, analysis of high energy particle diagrams, defect analysis of printed circuit boards, vision recognition, data mining for patterns, and those deep space explorers relaying their discoveries back to Mother Earth. This should give you the reader a feel of the unexpected software domains open to your own language theory applications.

For the novice programmer or computer science student, you should find O_2 fun to use and learn, makes you more skilled in applied language theory to your future projects with improvization. For those readers experienced with compiler/compilers like Yacc [10], Bison [4], Antlr [2], you might be interested in how I use multi-threading to implement nondeterminism applied to the basic lr(1) algorithm. With "in parallel" activities, this should free you to explore projects with multiple outcomes: as you will be now one half of O_2's critical regions.

Now to the point, this is a handbook about using O_2 compiler/compiler generator. It's been 10 years in the making with its support library. The genesis came from my master's thesis back in the mid 1990ies in writing a LR(1) algorithm to emit object-oriented code for a compiler/compiler. In tasting of my own "grammar dog food" as applied to the C++ language, I concluded in heated terms with myself that language theory as currently implemented in a compiler/compiler was in my opinion needing a rethink overhaul. Please do not consider this as an ego statement. From reading the various news groups at the time, I felt that language theory had not advanced very much since the late 1980ies. This is not to be taken as a criticism towards the newer crop of compiler/compilers removing Yacc's limitations by applying lr(k) ambiguity resolution as these newer systems were not available when I arrived at this conclusion. I felt hard-earned software wisdoms had not been applied to grammars. For example look at a LALR grammar for Java [8]. It is large, and one

piece. Software wisdom number 1 is "why is it not composed of smaller grammars similar to external library routines to make up a whole?" where each part could be recycled. Software wisdom number 2 is: Should not these grammars snippets be self documenting as in "Literate Programming"? I know this last point is contentious but I lean towards the small, reusable, and the documented. Just imagine what a typesetting system can do when applied to grammars.

So please bear with me as I will be describing how to use O_2 with the ethereal goal to minimize your learning investment to be productive. The only skills I ask of you the reader is a bit of programming background in C or C++, a sense of humour, and an openness to some e-media patois. Having some formal learning in "language theory" would help but is not mandidtory. My approach will be pragmatic, develop each idea from the small with lots of examples. There will be a bit of top/down, bottom-up, and for the impatient "let me try it now" to learn by experience. For the more experienced reader in compiling, there will be references to papers and text books that I found extremely helpful along my journey to build O_2. Sprinkled throughout will be Occam-like expressions and some light entertainment from my development experiences. As in the "La Tomatina Festival" in Bunyol Spain, I leave you to judge whether it's only good to pelt, or not too bad in the tasting.

Dave Bone
Dec. 2008

Contents

List of Tables

List of Figures

Chapter 1

I hear your dancing shoes a tapping

1.1 Stock taking

So you made it to here though you are still probably considering "should i invest my cerebral energy to learning and using Yac$_2o_2$?", or more fundamentally "what is the setup cost to get this thing up and running onto my laptop?". From now on I'll use Yac$_2o_2$'s nickname O_2[1]. Let's take stock of what software you should have on your computer.

1. TEX the typesetting system [15]
2. "Literate programming" [13], [20] — the *Cweb* system [17]
3. *mpost* a graphics drawing program [6]
4. O_2 compiler/compiler environment
5. a C++ compiler tool chain and a "Pthread" library [21] for multi-threading support
6. any "pdf" reader — like "xpdf"
7. an UNIX operating system of your choice, HP's OpenVMS on their Alpha platform, or Microsoft's XP operating system run as a 32 bit console application compiled from their Visual Studio 8 [2]. So far the Unix ports have been successful on Ubuntu, Sun's Solaris 10, and Apple's Intel and Power PC OS X systems.

Take heart as it looks like a lot of work to prepare before using O_2 but there is a white knight in lion's clothing to help you. Though points 1–3 are not mandatory, they can be obtained from TUG [19] by downloading their software. I enjoyed the installation experience as I'm a toe-tapping-clink-clack-ouch type of guy with too much raring-to-go octane in my blood. Give it a try and please consider joining TUG. [3]

[1] Is it a breath of fresh air or just hot?

[2] It has not been ported using the C# ".net" platform as my multi-threading is currently based on "pthreads" and so wrapper functions were used to map to non-unix systems calling to their each threading API which in Microsoft case came from their "process.h" header. Due to Microsoft's libraries using their "C runtime (CRT)" with multi-threading support, it runs in "debug mode" caused by problems in their release DLL libraries. Down the road their multi-threaded support under their ".net' / CLR environment will be tried. But for now it is deprecated and has not been ported to their newer Operating systems: aka Windows 7/8.

[3] An apology due to this minor inconvenience: the "cwebmac.tex" file must be modified so that file listings can be included into the grammar generated documents. To do this, u must find where the .../eplain/eplain.tex file is installed on your computer from the TEX installation. It should be in folder "/usr/local/texlive/2011/texmf-dist/tex/eplain/eplain.tex" where the year of the distribution in this example is 2011. Then u must add/edit ../cweb/cwebmac.tex" file from the *Cweb* installation. The following line is added at the beginning of this file:

input "/usr/local/texlive/2011/texmf-dist/tex/eplain/eplain.tex"

The balance of items 5–7 are your choice but the C++ tool chain and *Pthreads* library are mandatory. Your computer laptop environment should have them pre-installed. Depending on the computer supplier, they have on their website "open source" links to various projects for the downloading. The "Gnu" compiler tool suite [5] has been used on Apple's OS X Power PC and Intel laptops, Ubuntu, Sun's *sunstudio* IDE on their Solaris Operating System, and HP's compiler suite for VMS [4]. Any open source "pdf" reader can be used to view the generated documents.

Though the listed platforms running O_2 are few, they are germane to the various flavours of UNIX and should port easily to the other UNIX platforms. Porting to these platforms have brought out various weaknesses in each compiler tool chain. I use the word weakness in the sense that the conformance with the C++ standards by various vendors is still a "work-in-progress". For now I nod towards Gnu's porting standards that recommend staying as close to *C* as possible. So I am living in the C++ zone of "template thin ice". My template coding borders on the trivial with extensive use of token containers providing access to the parse stream. These accessors vary from streaming terminal tokens to an assortment of tree walkers with filter mechanisms on qualifying tokens supplied to the parser. It is the inheritance/template facility that gives uniformity to each token container type for the parsers. Parsers is used in the plural as parsing is done in stages and each parse stage can be a separate parser having its own token container types where their grammars act as logic automaton. More on this later but for the impatient have a look at Token containers in chapter 8.

The *Pthreads* library raised one issue with the memory allocated per thread stack at thread creation within O_2's runtime library. Thread stack size is a parameter tuneable when compiling the library to the ported operating system.

1.1.1 How much longer will this take?

From my experience, installing points 1–4 should be in the order of a couple of hours. Downloading time of the packages is where most of your time will be spent. Most "Open source" UNIX platforms already have some packages already pre-installed. In the /usr/local/yacco2 folder, you should first read the "README.pdf" document before running the Bash script of your operating platform. It describes various issues u could run across installing O_2. As O_2 takes on more Operating platforms, this document will be a running account of work-arounds and possible gotchas. The different Bash install scripts also have comments describing their options open to your fine-tunning.

1.2 The game and those names: yet another...

What is this name Yac_2o_2 anyway? In keeping with UNIX tradition it is a play on words: as in 2 thirds of a pun — pu?. And your retort could be: upu? The letter contortions were chemical in nomenclature with a twist of "Bollywood". So what's this name Yac_2o_2 and "how does one pronounce it?". It's a tongue twister that possibly a cold pronounces very well? I'll leave it to you the reader to indulge and come up with a name. Some suggestions have been like double "oh 2" yack or yuck or some molecular contortions that only the chemist-in-u could pronounce, or leave it to...? I only ask that it passes the movie rating for "general public 13+" viewers.

1.3 O_2's file directories

Table 1.1 gives a bird-eye's view of O_2's directories. The principle directory for development is "library". It houses the parser runtime library, the canned Terminal definitions, and various C++ header files that are used in compiling a grammar's emitted C++ files by O_2. The other directories are how O_2 evolved. There are files in there that might possibly enlighten the curious reader. It assumes that Yac_2o_2 was installed at: "/usr/local/yacco2".

[4] Though some of these references are aged, their newer flavours have been used. I did not change them to circa 2014 material as i wanted to re-emphasis that the UNIX software continuum works well!

Name	Directory Comments
bin	O_2's executables
bld_bash_xxx	Unix install scripts: where xxx is APPLE, GNU, SOLARIS
compiler	O_2's development subdirectories
diagrams	TEX and *mpost* related files
docs	generated documents for Yac$_2o_2$ system
externals	some common O_2 *Cweb* files
grammar-testsuite	Published grammars checking lrness of O_2
library	runtime library and include sources
o2linker	O_2linker *Cweb* development
o2testdriver	some dieting and exercising tests
qa	quality assuance test scripts

Table 1.1: Yac$_2o_2$'s directories

Name	Comments
∗.cpp	C++ header files of canned Terminal vocabularies
	8 bit ASCII characters
	lr constants, and threading support for *CC* digestion
∗.h	header files and ditto to their support
∗.w	*Cweb* files that O_2's library is written in
lib	O_2's runtime library subdirectories
	Release, Debug
	libyacco2.a is the runtime library to link with
grammars	canned Terminal definitions
	yacco2_k_constants.T — definition of parsing conditions
	yacco2_characters.T — 8 bit ASCII character definitions

Table 1.2: library contents — runtime library and canned vocabularies

1.3.1 Sub directory "library" — the oracle to creating your own compiler

This is your anchor point to developing your own language and parser. It provides the canned Terminal definitions that must be included in your grammars, C++ code supporting the grammars as threads, and finally the O_2's runtime library for the native operating system's linker. Table 1.2 details its contents: some C++ header files defining the canned "raw characters" and "Lr constant" terminals, other header files for the emitted grammars as threads, and the subdirectories: "lib" supplies O_2's runtime library depending on your flavour of Debug or Release for linking. "grammars" contains the canned source Terminal definitions for the "raw ASCII characters" and the "Lr constants". These 2 files must be included into your grammars.

The detailing of the canned terminals will be discussed in the Terminal chapter 3. It explores their anatomy, and questions like "why is the Terminal vocabulary divided into 4 classes?" and "how is the grammar's overall vocabulary segregated: Non terminals and Terminals?" answered. It will show how easy it is to create a Terminal definition within the "User or Error" vocabulary classes, going from simple to your own customized definition with C++ data elements and methods.

1.3.2 What are those file extensions?

When developing O_2 I tried to give clues about "what the files were" by their extensions. Some of the extensions are default extensions used by other packages. Here is a list of extensions and their meanings:

- w — *Cweb* files of written programs
- lex — lexical files namely the grammars

- T — the files making up the Terminal Vocabulary
- fsc — files for O_2linker to assemble
- mp — *mpost* files to for graphic diagrams
- tex — TEX files containing typesetting macros
- h — C++ header files
- cpp — C++ programs

The "w" extension files are *Cweb* programs. They have dual personalities to generate "pdf" program documentation and C++ code. Coincidently the "lex" extension is used by the *Yacc* compiler/compiler system. As I never used *Yacc* it's interesting to see how similar thoughts arrive at the same usage. There is no restriction to file naming as each file name is explicitly inputted to all O_2's programs.

There are 2 hardwired Terminal files that use the "T" extension. "raw characters" defines the "ASCII" character set, and "Lr constants" defining special conditions within the parsing library: for example the "eog" terminal indicates the "end-of-grammar file" condition. I normally define my own "error" and "user" terminals using this extension. It adds a bit of highlights in your directory.

The "fsc" is short for "first set control". These files get emitted for each grammar by O_2 that are then digested by O_2linker. There is a handcrafted file referenced as "yyy.fsc" in Figure 1.3 that brings in all these generated grammar linker files for O_2linker to munch on. You are free to use your own file naming convention for the "master linker" file and the grammars. If you choose to use another extension for the grammar file, be warned to adjust the *Bash* script file to find that extension.

The balance of menagerie files relate to *mpost*, C++, and TEX/Cweb activities. You could see variations on these extensions dependent on the software platform: most particular in the C++ area.

1.4 Odds and ends

Regarding your Integrated Development Environment(IDE), this is your choice. For example, you could use Eclipse, Sun's *sunstudio*, Apple's Xcode, or a plain old text editor. You need a C++ compiler along with the"Pthread" runtime library [21] to run the grammars as threads.

Any "pdf" reader will do if you want to use a GUI to view your document on your screen and to use its print menu to output the document to a printer. If you prefer to spool the document directly to a laser printer, a "pdf-to-postscript" converter like "pdftops" generates the "postscript" equivalent document and one spools it to the printer using "lp" program[5]. Here are the command line interface (*CLI*) commands to do this assuming a default printer has been set up:

```
pdftops xxx.pdf xxx.ps
lp xxx.ps
```

where xxx is the name of your document having the "ps" "postscript" extension.

1.5 O_2's tool kit to construct compilers

Cweb is the literate *C*/C++ programming environment that O_2 was written with. One of *Cweb*'s programs *cweave* is used by O_2 to generate its grammar documents for typesetting by TEX's tool chain[6]. O_2 subscribes to the same philosophy as "literate programming" in that a grammar's code contains both its grammar logic and its document logic. Both logics are woven within its grammar file. Its typesetting directives are *cweave*'s directives, and TEX's macro facilities. To spice up the grammar's generated document (a grammar's logic

[5]For information about a UNIX xxx program, run the "man xxx" command substituting the proper program name for xxx. "man man" gives details about UNIX's manual utility.

[6]O_2 uses *pdftex* program in the tool chain.

by pictures minus the 1000 words), *mpost* draws the grammar's individual productions as railroad diagrams [9]. *cweave* is then run against this file to generate a TEX file for typesetting by the TEX tool chain.

Though TEX and *Cweb* programs are not mandatory to use O_2, they are the jeweller's tools to self-documenting that in my opinion is a must to maximize your O_2 experience. Don't lament as this document manufacturing tool chain is automated by a *Bash* script[7] that comes with O_2's system. Figure 1.1 and Figure 1.3 gives you maps to O_2's manufacturing environment: compiling the grammars, linking the grammars together, and creating the various grammar documents. All examples in this book use *Bash* scripts of simple iteration constructs that are easily adaptable to other command line interpreters.

Figure 1.1 looks a bit overwhelming but it is the overview of compiling all your grammars. There are 2 distinct parts:

- compile each grammar by O_2 into C++ code snippets along with its "used thread" summary file for O_2linker (in)digestion, or to generate a grammar's documentation file controlled by an input option. The "pdf" document assembly is shown in Figure 1.3.
- *CC* then compiles all the outputted files from O_2 and O_2linker.

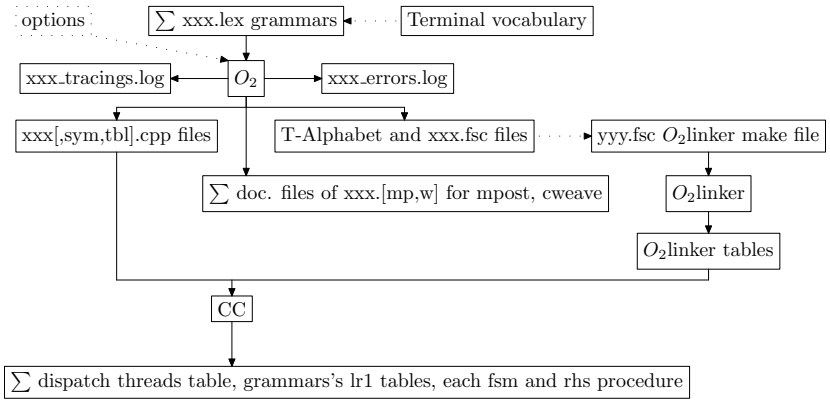


Figure 1.1: O_2's run environment

The compiling of the grammars by Yac$_2o_2$ uses its alias O_2 and "CC" to alias the C++ compiler in Figure 1.2. It shows the generated output that becomes digestion material for the programs in Figure 1.3. There are 2 log files out-putted: "xxx_errors.log" and "xxx_tracings.log" where "xxx" is the grammar file name without its extension that is currently being compiled. In quick passing they are used to debug a grammar: more to the point the entourage of grammars in a multi-core/multi-threaded environment where many activities are running simultaneously requires extensive debugging facilities. This subject "how to debug a grammar" will be developed fully in chapter 9 dealing with errors.

1.5.1 Options

There are 3 optional options that can be inputted in any order to O_2. Two of them deal with the Terminal Vocabulary to be dealt with in Chapter 3. The remaining option generates the grammar's documentation. O_2 uses the UNIX tradition of having the options follow the name of the program being run before the inputted grammar file. The minus sign "-" prefixes each option.

1. "-p" — generate the grammar's documents for printing.
2. "-t" — generate the "User terminal class" Terminals.
3. "-err" — generate the "Error class" Terminals.

[7]Bash is one of the command line interfaces(*CLI*) that comes with UNIX/LINUX.

For the astute reader who looked-ahead, "why aren't the other Terminal classes open to compiling?". Both the "raw characters" and "Lr constants' terminal classes are canned. They are fixed in cement and so never never change[8].

1.6 Run run run run away

There are 2 documents that are generated from a grammar: commented grammar code, and the emitted Lr(1) tables. "-p" print option uses the grammar's embedded *cweave* directives along with O_2's generated *cweave*'s directives to direct how the document is composed. The commented document's content includes *mpost* generated railroad diagrams per grammar rule, C++ code, and any grammar writer comments using the "literate programming" genre. If you try to compile the C++ code, **you will get compiler errors** caused by the "literate programming" directives. To generate a grammar's documents, the following command would be run assuming that you are in the directory housing the "xxx.lex" grammar file:

```
/usr/local/yacco2/bin/o2 -p xxx.lex
```
[9]

"xxx.w" and "xxx_idx.w" are the output files that are run through *mpost*, *cweave*, and *pdftex* programs to create the "pdf" documents. Figure 1.2 shows how "xxx.w" is transformed into the grammar's commented code document. The same process is done with "xxx_idx.w" to create the emitted Lr(1) tables document.

1.6.1 Eliminating finger blisters

To make it easier, you can edit one of *Bash*'s start up "profile" files so that an unqualified O_2 program can be found using its "PATH" variable. To do this, add the following line to your chosen start up file:

```
PATH=$PATH:/usr/local/yacco2/bin/o2
```
[10]

Assuming this modification, here are a couple of commands to generate the grammar's C++ files; the first one also generates the "error" and "t" terminal class C++ files:

```
o2 -err -t xxx.lex
o2 xxx.lex
```

"-err" or "-t" options are only used when their terminal's class has been modified or a new terminal defined.

1.6.2 All-in-one

A canned script comes with O_2 to compile all your grammars:

```
/usr/local/yacco2/compiler/grammars/o2grammars.sh
```

You can copy this into your development directory with your own additional commands. All these options can be inputted as parameters when evoking this script. When no option is inputted, the script compiles all the grammar files without generating terminals. For example, to generate all of O_2's grammar documents from *Bash*, its *CLI* "." operator evokes this inputted script with its "-p" parameter:

```
. /usr/local/yacco2/compiler/grammars/o2grammars.sh -p
```

[8] "That's what u think Dave but so far time and my hacking have lived with this untruth?"

[9] In this example, O_2 is explicitly qualified.

[10] Example assumes O_2's installation was placed at /usr/local/yacco2. Once upon a time, Yac$_2o_2$ was placed at the disk root level: /yacco2.

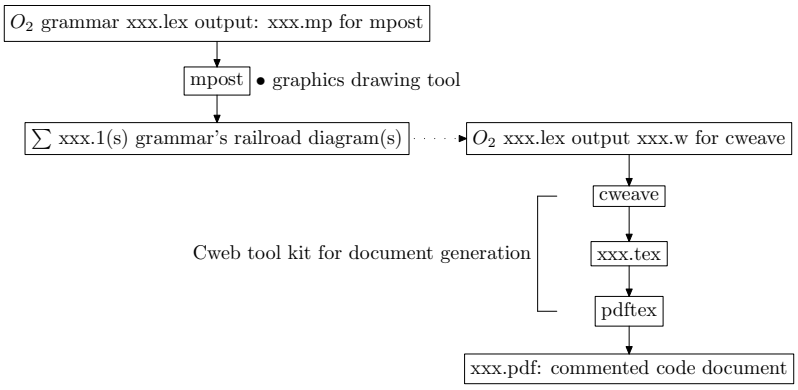


Figure 1.2: One of "xxx.lex" grammar documents manufactured: commented code

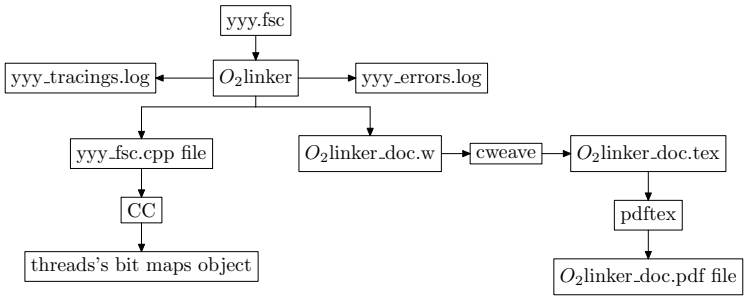


Figure 1.3: O_2's linker and document manufacture

These documents are generated in the "/tmp/o2/"[11] directory so that the temporary work files are removed from the system when the computer is shutdown. The gened documents are moved to the folder housing the grammars. There are 2 flavours of document files created with these extensions "pdf" and "ps" for postscript. The postscript file can be directly spooled to the printer instead of having to go through a "pdf reader" program to print it. Please give "o2grammars.sh" a read, clone it to your own directory, and edit to your own tastes.

1.7 O_2linker trying to control those ...

To glue all the grammars together, figure 1.3 illustrates O_2's companion "the linker". The role of O_2linker is its manufacture of the "grammar threads" table and ancillary "threads bit maps" used in thread dispatching. "O_2linker_doc.pdf" is a summary document that acts as a "table of content" for all the grammars. This is a free lunch that is served to you from your grammars. When your mired in the debugging of multi-threaded grammars, it sure is handy to see who calls whom; it's a tell tale of what and where[12]. A script example compiling the thread table for O_2:

```
o2linker /usr/local/yacco2/compiler/grammars/yacco2.fsc
```

Chapter 11 confesses all about O_2linker.

[11] If needed, this subdirectory is created automatically from the script.
[12] O_2's own file for the linker is "yacco2.fsc"

1.8 To tongue wag?

The code snippets presented in the book will be written in the "literate programming" genre. When required each source line will be prefixed by a line number acting only as a label for my tongue wagging. Reference [20] details the typesetting directives that can be used. *mpost* [6] diagrams can be created and included into your grammar for document output. TEX macros can also be written and used. You the author have many liberties to typesetting your grammar's document. My intention is to introduce you to my tastes in software development with the possibility that you might just try "literate programming" for your own software projects[13].

1.9 Summary

This chapter discussed the preliminaries required to have O_2 running: its dependencies on the TEX typesetting system, and the use of "literate programming" provided by *Cweb* system to manufacture documents and to write software. Maps to O_2's facilities and how to use them were described.

The next chapter will explore "What is a grammar?". It will draw from O_2's own grammars to parse its command line input. The two types of grammars: monolithic and thread are introduced, their interaction shown, and the minimum *Cweb* program written to start building your own compiler.

[13] Now be-off with your wagging tongue Dave or heads will roll.

Chapter 2

What are Grammars?

This book describes O_2's implementation of deterministic context free lr(1) grammars as defined in Knuth's paper [12]. Threaded grammars extends this definition and will be explored later. "language theory" behind grammars will not be developed. I point you to a marvellous book "Formal Languages and their Relation to Automata" by Hopcroft and Ullman [7]. This subject is well treated with excellent references, the expressive powers per class of grammar, and the equivalence between automata and grammars. Though it was published circa 1969, its graphics and typesetting material are a treat to the eye particularly when there was limited typesetting software to do this[1]. I still enjoy reading it today and believe you will too with its later edition.

Before exploring the chapter's question, here is another one: So what can u do with grammars? Some question? Let's see. They act as recognizers on input sequences: some form of constraints. Not a very good start. These sequences can be a simple as ensuring that the patterns are kosher. Again so what! Well at various boundaries, special programmed code can be run with it: called syntax directed code. The name comes from associating various grammar's sequence points aka the grammar's syntax with your own programming logic. This logic could be to build an abstract tree from the expression of the subrule, giving a name to the sequence seen and represented by its own meta terminal (terminals representing other terminals). This aliasing allows one to build up other sequences for other grammars to deal with. Is this a recursive statement? Yes it is with its reducing self snapshots. So we can conclude that grammars can be an assembly line process: dealing with input, digesting and regurgitating the input to be re-digested, and possibly provide a result[2]. Your own grammar Rorschach images are what matters with your own absorption rates of thought.

Like "Alice in Wonderland" [3], "where should I begin?". Let's see what a grammar text file looks like on page 227. Humm not too intuitive and it's rather ugly as in duckling [1]. Before I break up its anatomy lets informally describe a grammar. It is like a cooking recipe composed of four parts:

- Terminals are the raw materials making up a taboule. This is called the Terminal vocabulary.
- Non-terminals or rules are names given to various assembly/cooking stages. This is called the Non-Terminal or Rule vocabulary.
- Productions relate non-terminals with other non-terminals and terminals. They are the cooking instructions.
- Start rule. This is the starting point to "let's start cooking".

2.1 Where's the beef? — Vocabulary

Terminals and Rules together make up a grammar's vocabulary. Each one is independent of the other. They are not related. Some examples of Terminals are the parts making up a meat flipper, taboule's ingredients — onions, salt, vinegar, parsley, maybe some garlic, and oliver oil, the alphabet used to write this book, a

[1] TEX was being worked on approximately 10 years later.
[2] Lets hope its not bowdlerizing?

set of numbers, gene sequencing, pixels on screen monitoring your MRI scan. They are the raw materials that will be combined somehow into something.

Rules are names given to instructions. The component sequences or instructions are given by the Production list relating the rule names to their various sequences.

2.2 Slice and dice — Line dancing

Each instructional step(s) is provided by the list of Productions. A Production is composed of a left-hand side and a right-hand side that provides the instructional sequences with a special symbol separating the 2 sides. The left-hand part is a rule name drawn from the Non-terminal vocabulary. The right-hand part aka subrule is a mixed sequence of rules and terminals drawn from the grammar's vocabulary. A string or line of symbols is another way to describe this sequence. The sequence can also be empty or void of symbols: the epsilon symbol ϵ is sometimes used to denote this condition though there are no symbols present[3].

Here is a production list rendered in Table 2.1 from the grammar's raw text file. I felt rephrasing the components in a different form reenforced their duality: rules and their symbols.

Left-hand side Rule Name	Separator symbol	Right-hand side Subrule
Reol	→	Rdelimiters
Rdelimiters	→	"x0a"
	→	"x0d" |.|
	→	"x0d" "x0a"

Table 2.1: Sample Production list taken from page 227

|.|[4], "x0a", "x0d"[5] are symbols from the terminal vocabulary. |.| comes from the "Lr constant" class of terminals while the other 2 are examples of raw character symbols. The 2 rules are "Reol", and "Rdelimiters". "Reol" has just one subrule where as "Rdelimiters" has 3. Licence was taken to not duplicate the rule's part of "Rdelimiters" in the table for its 2 other subrules with the implicit meaning that they are associated with the previous listed rule. O_2 grammars use this liberty to express its subrules. There can be many possible assembly instructions per rule. Within the deterministic context free grammars, you cannot have two or more instructions being completed at-the-same time. Only one subrule can be completed under deterministic constraints. Breaking of the lr constraints leads to that dreaded condition ambiguity[6]. With the extension of threaded grammars, this restriction is removed. This is nondeterminism — parallel tasks being done simultaneously. I hear your critical moaning on efficiency but with the magic of lookahead sets and arbitration... More to come later[7].

2.3 And a one and a 2 and a...

For the overall assembly to succeed there must be only one starting point which is the "Start rule". From the given example "Reol" is the "Start rule" as it is the first rule defined. One restriction exists, a "Start rule" can never appear in a subrule. A "Start rule" can have many variations (subrules).

2.4 Bones, sinew, and stuffings

Figure 2.1 overviews the anatomy of an O_2 grammar. What I see is not what was described making up a formal grammar: a tupple of 4 components <S,T,R,P> start rule, grammar's vocabulary of terminals and

[3] Oxymoron and a closed loopback definition?
[4] invisible symbol that does magic from the lr constant class
[5] symbolic hexadecimal forms for "line feed" and "carriage return"
[6] We"ll see later how this can be the taming-of-the-shrew?
[7] The street hawker's future promises.

rules, and the productions. Fortunately there are no vowels in this tupple. But what are these extra things: fsm and thread definition? So lets see the liberties taken with a O_2 grammar definition:

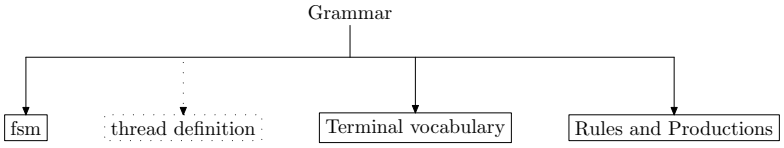


Figure 2.1: Skeleton of an O_2's grammar

- *fsm* is a packaging agent for emitting C++, to contain associated "syntax directed code", and documentation comments.
- What is this "thread" component? It is a grammar extension that defines a grammar to be run as a thread[8]. It is optional indicated by the dashed lines around it. Without it presence in the grammar definition, it is equivalent to other compiler/compiler implementations that I baptized as a "monolithic grammar".
- The "Terminal vocabulary" matches the tupple. It has an extension to enumerate the terminals that is developed later.
- "Rules and Productions" are combined as one entity. It is more concise but equivalent to the formal definition. I'm a bit germanic in over describing things. This is a form of apology to the *C* pundits of minimalization[9].

The following subsections are hors d'oeuvres before their main courses are developed. Each dish is cut from the example at page 227. This saves your book and your patience from tattering.

2.4.1 Fsm component

```
fsm (fsm-id "eol.lex",fsm-filename eol
 ,fsm-namespace NS_eol,fsm-class Ceol,fsm-version "1.0"
 ,fsm-date "6/2003",fsm-debug "false"
 ,fsm-comments "end-of-line matcher."
 )
```

The fsm component has a design similar in form to a C++ function call where the call signature is a list of paired keyword/value. It is an acronym for "finite state machine"[10]. Each keyword attribute starts with the prefix "fsm-" followed by its name. Below describes their meanings:

- id – the grammar's raw text file name: "eol.lex""
- filename – the base file name for the C++ emitted files: eol
- namespace – same as C++ namespace facility and functionality: NS_eol
- class – emitted C++ class that defines the grammar: Ceol
- version – just a literal comment used in the documents cover sheets: "1.0"
- date – create date used in the documents cover sheets: "6/2003"
- debug – "true" or "false" value to turn on or off grammar tracing: "false"

[8] Missing rib?
[9] But I'd rather not overload words with contextual misunderstandings?
[10] U can beat me up on this but at the time i could not think of anything better.

- comments – short description of grammar: used by O_2linker's document generation: "end-of-line matcher"

fsm will be developed fully in Chapter 4 as it has many "syntax directed code" directives for the grammar writer to cover error handling, building of abstract trees, symbol table management, ambiguity arbitration, to grammar-to-grammar communication[11].

2.4.2 Threads and silk worms

```
parallel-parser (
  parallel-thread-function
    TH_eol
  ***
  parallel-la-boundary
    eolr
  ***
)
```

parallel-parser defines the grammar as a thread. Its contents are wrapped by the "(" ")" pair. There are just 2 attributes — *parallel-thread-function* and *parallel-la-boundary* whose contents are each closed off by "***"[12]. It is the demarcation for each syntax directed code directive. Thread directives explained:

- *parallel-thread-function* provides the thread's called name for the emitted C++ code. Its fully C++ qualified name is the grammar's namespace plus the thread name: NS_eol::TH_eol.
- *parallel-la-boundary* allows the grammar writer to fine tune the lookahead boundary for the grammar[13]. It is similar to an arithmetic expression using rules and terminals as terms to define the thread's "end-of-tokens" set. This is a follow set for the start rule. Each rule in the expression uses its "first set" calculation to provide its contributing terminals. Though sketchy you have the notion. The above example uses the "eolr" terminal. It comes from the "Lr constant class" of terminals representing all the terminals in the vocabulary. This context is grammar writer sensitive. It saves your fingers and diets your computer memory[14].

2.4.3 Terminal Vocabulary

```
@"/usr/local/yacco2/compiler/grammars/yacco2_T_includes.T"
```

This is O_2's "@" file inclusion facility used to bring in the terminal vocabulary. This file contains the following include files and C++ comment.

```
/*
File: yacco2_T_includes.T
Purpose: Terminal alphabet definitions used by yacco2's grammars.
Author: Dave Bone
Date: 28 May 2003
*/
@"/usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.T"
@"/usr/local/yacco2/library/grammars/yacco2_k_symbols.T"
@"/usr/local/yacco2/library/grammars/yacco2_characters.T"
@"/usr/local/yacco2/compiler/grammars/yacco2_terminals.T"
@"/usr/local/yacco2/compiler/grammars/yacco2_err_symbols.T"
```

[11] Your imagination is your Kodachrome

[12] This is ugly Dave! I know it's not elegant but this is how it evolved. What hindsight would u like to use against your own creations? Why this character combination? I experimented with other sequences and i felt this was the least unelegant. I wanted something that was gentle to one's view but still had enough signaling power of intent.

[13] Follow set used in determining when a subrule is to be reduced.

[14] Is this a set within a set within a ...?

The "T" file extension is per your own tastes. There are 5 file types.

- "yacco2_T_enumeration.T" provides the enumeration scheme for the terminals. Though not part of a formal grammar, it is required to contain the mapping of the terminals for set membership.
- yacco2_k_symbols.T defines the canned lr constant terminals.
- yacco2_characters.T defines the canned raw character terminals.
- yacco2_terminals.T defines your own terminals within your own file account.
- yacco2_err_symbols.T defines your own error terminals housed in your account. The term errors covers all degrees of messages from the gentle to the abrupt.

Their appearance order is important due to the implicit enumeration scheme used by O_2. For the moment, a fully qualified directory path must be provided. The grammar writer can use any name for the first and last 2 files.

2.4.4 Rules and Productions

The Rules vocabulary has a simple containment structure provided by the rules keyword:

rules { $\sum$ each rule definition }

Each individual rule definition contributes to its vocabulary where each rule's name must begin with "R"[15]. Inside each rule definition are its subrules that implicitly contribute to its production list. Though gauche in this excerpt, the line numbers are for my directed comments.

```
20 rules{
21  Reol (){
22    -> Rdelimiters {
23      /@ Return the |eol| token back to the caller. @/
24    op
25     CAbs_lr1_sym* sym = NS_o2_terms::PTR_eol__;
26     sym->set_rc(*parser()->start_token__
27     ,rule_info__.parser__,__FILE__,__LINE__);
28     sym->set_line_no_and_pos_in_line
29     (*parser()->start_token__);
30     RSVP(sym);
31    ***
32    }
33 }
34
35  /@ @** |Rdelimiters| rule.\fbreak @/
36  Rdelimiters
37  /@Eol Variants. \invisibleshift escapes from shift/reduce
38   conflict caused by x0d common prefix. As it's not in the
39   token stream, it deals well with this conflict type.
40  @/
41    ()
42    {
43     ->
44      /@ lf: Unix\fbreak@/
45       "x0a"
46     ->
47      /@ cr: Mac.
48       Note \invisibleshift removes shift/reduce conflict
```

[15] A mnemonic for my Golden years à la Bowie

```
49        caused by the lookahead boundary of |eolr|.\fbreak
50      @/
51      "x0d" |.|
52     -> /@ cr:lf Windows\fbreak@/ "x0d" "x0a"
53    }
54 }// end of rules
```

The individual rules are sequentially defined without any importance to order. The first rule is deemed the "Start rule". Line 21 starts "Reol" definition while line 36 starts "Rdelimiter" definition. Lines having the `->` symbol introduces a subrule sequence that are the individual productions. For "Reol" line 22 begins its its only subrule. Lines 43, 46, 52 start the 3 subrule definitions for "Rdelimiter".

2.4.5 Those comments

Two types of comments exist: C++ and typesetting. Both the C++ comments are supported:

// single line comment

/* ... */ multiple lines of comments

They can be sprinkled anywhere within the text file. If they are part of a "syntax directed code" directive they will be emitted in the C++ code as they act as guide posts for the grammar writer when paper reviewing C++ code[16]. Here is a comment that identifies the raw grammar text file.

```
/* FILE:    eol.lex  Date: 17 Juin 2003
   Purpose: end-of-line recognizer
*/
```

Typesetting comments have this form:

/@ ... @/[17]

Anything goes between these typesetting brackets. Lines 4–7, 19, 23, 35, 47–50 below are examples of typesetting comments. They are dormant until O_2's "-p" option is inputted. Normally *cweave* directives are added along with your prose, graphics, and other media particularly using *mpost*'s drawing facilities. Mix this with *Cweb* and TEX's directives and u have quite a media cocktail. Though C++ comments in the plain grammar file have some value, give typesetting comments a try in my opinion merits your learning efforts.

```
 4 /@
 5   @** Eol Thread.\fbreak
 6   Fame: end-of-line matching for Unix, Mac, or Windows.
 7 @/
19 /@ @** Rules.\fbreak @/
23     /@ Return the |eol| token back to the caller. @/
35     /@ @** |Rdelimiters| rule.\fbreak @/
47 /@ cr: Mac.
48    Note \invisibleshift removes shift/reduce conflict
49     caused by the lookahead boundary of |eolr|.\fbreak
50 @/
```

Some of the above typesetting comments include *cweave*'s directives while some of the text contains TEX macros. If you look closely, these comments are sprinkled at various syntax points within the grammar. That is, before the grammar's *fsm* and throughout its *rules* block. Though the Terminal vocabulary was not described, typesetting directives are allowed.

2.5 Tweedle Dee Twitter Dum – Break time

I leave you to attend to my kitchen chores. Take a break while i prepare the basic C++ program to parse O_2's command line.

[16] God-forbid if u are using those IDE's code debuggers... i do and good stuff.

[17] It's a play on comments between C++and *cweave*. A bit corny but do i speak the truth?

U're back so fast? I am flattered or you're closing the book on me. I've finished my patter as you are probably getting bored with my staleness. So what would u suggest to improve a rather dry subject? I hear the chorus: Bring on a parsing program.

2.5.1 O_2's frontend parser

The basic C++ program is edited for aesthetics. I told a white fib as this was supposed to be a *cweave* program as stated in chapter one. Why not? I just didn't want to put you now through yet another indirect explanation on *Cweb* programming before seeing a parse program. So here are the basic rudiments needed to parse:

- O_2 library interface — *Parser* with supporting facilities
- The terminals vocabulary
- Monolithic grammar
- O_2's tracing variables
- Terminals containers

The C++ code is simple and its comments should be adequent for the programmer to follow [18]. I could have given a simple parsing example but i felt that u are a mature audience and a real example would be most appropriate. So here is the killing of one stone with 2 birds: phase 1 fetches the command line input and parses it, and phase 2 does lexical parsing on the grammar file. It is a neat start as it demonstrates independent monolithic grammars for different parsing stages. Stage one is a temporary parse that gets thrown away while the 2nd part becomes a parse scaffold where one's output becomes the next stage's input. As will be shown, each parse stage has the same coding pattern: set up the parse, parse it, and deal with the potential errors.

[18] I don't want to be too intestinal in my banter.

I hear your hoots, so here is the program:

```
// o2_example.txt
#include "o2.h" // o2 api, vocabulary, grammars, and externals
YACCO2_define_trace_variables(); // macro: o2 tracing variables
// input options: -t,-err, -p
 yacco2::CHAR T_SW('n');
 yacco2::CHAR ERR_SW('n');
 yacco2::CHAR PRT_SW('n');

// terminal containers
 yacco2::TOKEN_GAGGLE JUNK_tokens;
 yacco2::TOKEN_GAGGLE P3_tokens;
 yacco2::TOKEN_GAGGLE Error_queue;

std::string o2_file_to_compile;// cli extracted grammar file
int main(int argc,char*argv[])
{
 cout<<yacco2::Lr1_VERSION<<std::endl;
 // Fetch and parse the command line
 GET_CMD_LINE(argc,argv,Yacco2_holding_file,Error_queue);
  if(Error_queue.empty()!=true){
   DUMP_ERROR_QUEUE(Error_queue);
   return 1;
  }

 YACCO2_PARSE_CMD_LINE(T_SW,ERR_SW,PRT_SW
                       ,o2_file_to_compile,Error_queue);
  if(Error_queue.empty()!=true){
   DUMP_ERROR_QUEUE(Error_queue);
   return 1;
  }

 // Lexing the grammar
 using namespace NS_pass3;
 tok_can<std::ifstream> cmd_line(o2_file_to_compile.c_str());
 Cpass3 p3_fsm;
 Parser pass3(p3_fsm,&cmd_line,&P3_tokens
             ,0,&Error_queue,&JUNK_tokens,0);
 pass3.parse();

 if(Error_queue.empty()!=true){
  DUMP_ERROR_QUEUE(Error_queue);
  return 1;
 }
exit:
 lrclog<<"Exiting O2"<<std::endl;
 return 0;
}
```

Sample program: parse command line input, and lexical phase

Let's review what is needed by the parser itself. O_2's generic *Parser* requires 3 arguments:

- The associated grammar to parse with
- An input token container called a supplier
- An output token container called a producer

All other arguments to *Parser* are optional. *Parser* also needs the surrounding vocabulary context: terminals and its associated rule vocabulary. The *Parser's* trace variables must be present. The macro **YACCO2_define_trace_variables()** supplies them. That is it.

The commented code should give you the feel of what's going on[19]. The example shows 2 of the 5 requirements: O_2's tracing macro, and the token containers. Now the indirections explained.

[19]Like weeds, there's that indirection creep. This will be a bit dragged out like a "Henry James" story?: anticipation, dissatisfaction, and anti-climatic. Let's taste and do the taste buds guessing before the chef corrects us.

2.5.2 Indirections take one — o2.h

"o2.h" file provides the various definitions: O_2's library interface, the grammars, external routines and variables, and the Terminal vocabulary. Their C++ comments indicate their intent. The majority of files are grammar definitions for parsing stages after parsing the command line data:

```
#ifndef o2__
 #define o2__ 1
 #include "globals.h"
 #include "extndefs.h"            // constant definitions
 #include "yacco2_stbl.h"         // symbol table
 using namespace yacco2_stbl;

 // --------grammar definitions ----
   #include "o2_lcl_opts.h"  // cmd line parser
   #include "o2_lcl_opt.h"   // threaded grammar: cmd line options
   #include "pass3.h"        // lexical grammar
   #include "la_expr_lexical.h"
   #include "la_expr.h"
   #include "enumerate_T_alphabet.h"
   #include "epsilon_rules.h"
   #include "mpost_output.h"
   #include "prt_xrefs_docs.h"
   #include "eval_phrases.h"
   #include "enumerate_grammar.h"
   #include "rules_use_cnt.h"
 // --------- end of grammar definitions ---
  extern void COMMONIZE_LA_SETS();
  extern int NO_LR1_STATES;
  extern STATES_SET_type VISITED_MERGE_STATES_IN_LA_CALC;
  extern LR1_STATES_type LR1_COMMON_STATES;
  extern CYCLIC_USE_TBL_type CYCLIC_USE_TABLE;
  extern void Print_dump_state(state* State);
#endif
```

Include file "o2.h": run context for sample program

If you look closely you'll see grammars used as logic automata: "epsilon_rules.h", "enumerate_grammar.h", and "rules_use_cnt.h". They are not parsing data but are using their token stream as logic points. The logic sequencing of the grammar is tied to syntax points within its structure. These automata detect epsilon rules, enumerate the terminal and rules vocabulary, and optimize how the grammar's rules get called and recycled. I'll gum stick this thought into your craw: Let's see where u take it particularly when nondeterminism is thrown in as logic potentials?

2.5.3 Indirections take two — globals.h

"globals.h" file defines O_2's library interface and the Terminal vocabulary. There is little magic to these definition files. They are generated by the options given to O_2. File paths can be relative or explicitly expressed. The below code gives you a twist of both. Going the relative file path route depends on your C++ compiler include directives to resolve their where-abouts. Here is its extracted contents:

```
#ifndef globals_h__
#define globals_h__ 1
#include <stdarg.h>
#include <stdlib.h>
#include "yacco2.h"                    // o2's api

// terminal vocabulary: enumeration and 4 terminal classes
#include "yacco2_T_enumeration.h"
#include "yacco2_err_symbols.h"    // gened by -err option
#include "/usr/local/yacco2/library/yacco2_characters.h"
#include "/usr/local/yacco2/library/yacco2_k_symbols.h"
#include "yacco2_terminals.h"      // gened by -t option

using namespace std;
using namespace NS_yacco2_T_enum;
using namespace NS_yacco2_k_symbols;
using namespace NS_yacco2_terminals;
using namespace yacco2;
#endif
```

Include file "globals.h": O_2's API and terminal vocabulary

yacco2.h defines O_2 interface using the relative file path facility. It includes all the C++ template types needed for containers. All languages including O_2 must use the canned *yacco2_characters.h* and *yacco2_k_symbols.h* definitions. The 2 remainders *yacco2_err_symbols* and *yacco2_terminals* are your own creation (both in file name and content). The C++ *using namespace* directive is for my convenience: your liberties and tastes prevail.

2.5.4 Indirections take three — token containers

O_2 supports 4 container types:

- File container whose input comes from a text file.
- String stream. Text placed in a string that gets tokenized.
- Terminal token stream. These are Terminals created from grammars and added to a container using syntax directed code points.
- Tree based container with various tree walkers: depth first, breath only, and forest walkers. Yes when trees are built, you can walk them like a token stream. There is a very nice facility that allows one to walk trees in different ways with a filtering mechanism to allow or disallow the type of terminals to be issued in the token stream. Let's rephrase this, trees are conditionally branched structures that now are flattened. Trees normally get morphed according to various post processing stages: O_2 has some delightful fruits for your sampling.

The first 2 container types are input only. Their sources are text to be tokenized. The 3rd container type, starts as an output container in one parsing stage where its contents are entered by a grammar then it becomes an input container for another parse stage. The tree container is also read only. Its difference to the others is the tree node's content is a terminal. There is an indirection to its contents accorded by its link structure and how it is walked. The starting example has 2 container types in use: the input container of file source, and 2 output containers of terminal streams: regular terminals, and error terminals. The input container automatically handles the opening of the text file, turns each text character read into its terminal equivalence, and tags each terminal produced with its text file location: line number and character position within the line. This benefits error reporting and debugging token streams. The output container can be a mix of below terminals:

- Lr contant's end-of-token-stream terminal indicator.
- Individual inputted text characters equivalences.
- Summarized terminal: eg, a sequence of characters that becomes a specific keyword like *fsm*.

- Meta-terminal built upon other terminals created from n - 1 previous parse stages.
- Error terminals.

So enough detours on containers. The exampled containers are *JUNK_tokens*, *P3_tokens*, and *Error_queue*. Though *JUNK_tokens* is just a holding bin of tokens that are created and are to be recycled back to heap heaven, it has been rarely used as winding down of a process tears down the process's memory. My comments are for your own memory debris. *Error_queue* holds error tokens for error processing or messaging digested by the *DUMP_ERROR_QUEUE* procedure. *P3_tokens* is an output token stream that becomes an input token stream later in the following parse stage.

Containers can act as temporary variables; they are holding token stages of some parsing activity. Put another way, the containers are a collage of terminal cubism. Here's my take on dilly-dally-ism:

CLI ⇒ holding file ⇒ file container ⇒ CLI parser ⇒ switches

2.5.5 GET_CMD_LINE: Fetching input

I felt life would be a lot easier to use a parser to fetch its own fodder[20]. So here is a simple process to fill a holding file with the program's runtime arguments.

```
extern
 void GET_CMD_LINE(int argc,char*argv[],const char*Holding
                  ,yacco2::TOKEN_GAGGLE&Errors)
 {
  using namespace std;
  using namespace NS_yacco2_err_symbols;
  using namespace yacco2;
  ofstream ofile;
  ofile.open(Holding,ios_base::out|ios::trunc);
  if(!ofile){
   CAbs_lr1_sym*sym= new Err_bad_filename(Holding);
   sym->set_external_file_id(1);
   sym->set_line_no(1);
   sym->set_pos_in_line(1);
   Errors.push_back(*sym);
   return;
  }
  if(argc==1){
   char cmd_line[Max_buf_size];
   cout<<"Please enter Command line to process: ";
   cin.get(cmd_line,Max_buf_size,'\n');
   ofile<<cmd_line;
   ofile.close();
   return;
  }
  for(int x= 1;x<argc;++x){
   ofile<<argv[x]<<' ';
  }
  ofile.close();
 }
```

Fetch program's command line data and store in a holding file

O_2's holding file is "yacco2cmd.tmp" provided by *Yacco2_holding_file* variable. Due to a terminal's source file positioning mechanism, a holding file is required to start things off. All terminals should be source file located. Error reporting uses the error terminal's co-ordinates to display the source file's line and character position. The above error checking is for my own programming hygiene for a hardwired holding file name that is improper to the operating system that also hardwires its error co-ordinates to a non-existant file. The code illustrates how u can build your own front-end command line parser by putting its contents into a holding file that is food for your parser's input container.

[20] Is this narcissistic?

2.5.6 YACCO2_PARSE_CMD_LINE: Parsing da input

Finally. The nut is cracked and here's its pulp.

```
extern void YACCO2_PARSE_CMD_LINE
   (yacco2::CHAR& T_sw,yacco2::CHAR& ERR_sw,yacco2::CHAR& PRT_sw
   ,std::string& Grammar_to_compile
   ,yacco2::TOKEN_GAGGLE& Error_queue)
{
 using namespace NS_yacco2_err_symbols;
 using namespace yacco2;
 tok_can<std::ifstream>             // tmp input container
         Cmd1_tokens(Yacco2_holding_file);
   if(Cmd1_tokens.file_ok()==NO){
    yacco2::Delete_tokens(Cmd1_tokens.container());
    CAbs_lr1_sym*sym= new Err_bad_filename(Yacco2_holding_file);
    sym->set_external_file_id(1);
    sym->set_line_no(1);
    sym->set_pos_in_line(1);
    Error_queue.push_back(*sym);
    return;
   }

 TOKEN_GAGGLE lcl_options_tokens;   // tmp output container

 using namespace NS_o2_lcl_opts;
 Co2_lcl_opts opts_fsm;             // grammar to parse with
 Parser options(opts_fsm,&Cmd1_tokens,&lcl_options_tokens
               ,0,&Error_queue,0,0);// initialize parser

 options.parse();                   // Just do it!

 yacco2::Delete_tokens(Cmd1_tokens.container());// tidy up
 if(Error_queue.empty()!=YES) return;

 /*
  * Pass back results thru the called arguments drawn from
  * opts_fsm grammar's variables.
  * One way to pass parsed results between stages.
  */
 T_sw   = opts_fsm.t_sw_;
 ERR_sw = opts_fsm.err_sw_;
 PRT_sw = opts_fsm.prt_sw_;
 Grammar_to_compile += opts_fsm.file_to_compile_;
}
```

Parse command line data

This code shows the use of temporary parsing containers, *opts_fsm* grammar holding the parsed results, and how these results are passed back through the passed-in arguments. O_2 is open to your own disciplines. I wanted to show another twist in how you can pass parsing results between stages without an output container. Now how are errors reported, as I don't see anything but the main program tests for their presence? Well the argument *Error_queue* is passed into the *Parser* to receive error terminals. Not to overload you, *Parser* has a method *ADD_TOKEN_TO_ERROR_QUEUE* to deposit errors into this queue from a grammar's syntax directed code. The main program reports errors when this container has booty. Though not seen, jeez more indirection, I use another grammar to deal with errors: the *DUMP_ERROR_QUEUE* routine is similar to *YACCO2_PARSE_CMD_LINE* to deal with error token streams. Well here's its clutter:

```
extern void DUMP_ERROR_QUEUE(yacco2::TOKEN_GAGGLE&Errors)
{
 using namespace NS_yacco2_k_symbols;
 using namespace yacco2;
 Errors.push_back(*yacco2::PTR_LR1_eog__);// follow set for
 Errors.push_back(*yacco2::PTR_LR1_eog__);// start rule to accept
 using namespace NS_o2_err_hdlr;
 Co2_err_hdlr fsm;
 Parser pass_errors(fsm,&Errors,0);
 pass_errors.parse();
 yacco2::Parallel_threads_shutdown(pass_errors);
}
```

To dump error(s): grammar used to do this

I'd like to entice u by showing the out-of-the-box error reporting facility of O_2. Here are 2 samples: a non-existent grammar file, and a bad option.

```
bone $: /yacco2/bin/o2 exl.lex
O2 version: .669 Date: July/2009
Error in file#: 1 "yacco2cmd.tmp"
exl.lex
^
 fpos: 0 line#: 1 cpos: 1
 who thru it: /yacco2/compiler/grammars/o2_lcl_opts.cpp line#: 208
 bad filename filename: "exl.lex" does not exist

bone $: /yacco2/bin/o2 -t -er eol.lex
O2 version: .669 Date: July/2009
Error in file#: 1 "yacco2cmd.tmp"
-t -er eol.lex
   ^
 fpos: 3 line#: 1 cpos: 4
 who thru it: /yacco2/compiler/grammars/o2_lcl_opt.cpp line#: 44
 bad cmd-opt
```

Two examples of error reporting

The 2 interesting parts are where and who threw the error. The where indicates the file containing the error, and displays the line-in-error contents. The error's character position in the line is underscored by the "^" character followed by an appropriate message. The message can be custom made or comes from the error terminal's literal value. Both types are shown. To aid the developer or person reporting the error, the source grammar that threw the error is identified along with its C++ source line location. This is bi-directional reporting: for the user of O_2, and for the grammar developer: me. It also applies to your own language development using O_2. This is standard out-of-the-box stuff.

Back to *DUMP_ERROR_QUEUE*, there is more to come later on error processing as there are some subtleties in this example needing explanation[21].

2.5.7 More comments on *Parser*

The *Parser* type has 7 inputs[22]. The 7 inputs are for discussion sake. Here are their details related to the example:

1. The associated grammar that parses the input: *opts_fsm*.
2. The input file token container aka supplier: *Cmd1_tokens*. It is supplied the canned holding file name whose contents are the extracted program's arguments: *Yacco2_holding_file*. Containers use C++ template facility so that you can create your own: `tok_can` < `std::ifstream` > is your file container.

[21] Again those false promises of deliverables? End-of-the-terminal-stream condition, no output container, shutting down of threads? When will these other considerations stop popping up like a Jack-in-a-box? There are more neat facilities to catch errors, deal with them, and report on them. A free-lunch will be served to all.

[22] My apologies for the slow start but I feel the intro should be semi-detailed. Though a bit sketchy, I felt a vacation brochure approach semi-prepares you for your voyage into O_2 land.

3. The output token container *lcl_options_tokens* is a temporary container as it is not used downstream. It deals with the options and the file name to parse.
4. Input token position to start reading from the supplier container. The token position is relative to zero. Hmm, threaded grammars use this feature as their token stream starts at a point only known by its calling grammar. Of course in the world of parallelism exclusivity reigns?
5. *Error_queue* is the container to hold error terminals. Normally you would provide a globally defined error container to the root parser. This container becomes visible across threaded grammars to add their own booty.
6. Recycling container: *0*. Allows one to add "newed" Terminals from the heap to be deleted aka recycled some time: garbage collection. Though the thought is there, the practicality has been limited in my experience.
7. Symbol table facility: 0. Seems to be adequate as used in O_2 and a Pascal translator that I wrote. Why isn't it used in the example? It could have been to demonstrate it but then it raises the question: what utility does it have in a command-line parsing? Nada or a very contrived optional symbol table lookup.

U can abbreviate *Parser*'s input. All arguments from 4 onwards can be omitted as the *Parser* has defaulted values.

2.5.8 Lexing the grammar

I'll save u going back to check the program example, here is the lexing code.

```
// Lexing the grammar
using namespace NS_pass3;
tok_can<std::ifstream> cmd_line(o2_file_to_compile.c_str());
Cpass3 p3_fsm;
Parser pass3(p3_fsm,&cmd_line,&P3_tokens
             ,0,&Error_queue,&JUNK_tokens,0);
pass3.parse();

if(Error_queue.empty()!=true){
 DUMP_ERROR_QUEUE(Error_queue);
 return 1;
}
```

As u can see, it uses the same parsing pattern described for the command line data:

- Set up the *Parser* for parsing: *p3_fsm* grammar to parse with, input container *cmd_line*, and the output container *P3_tokens*.
- Parse it.
- Check for errors.

Though not seen, *P3_tokens* container is used down stream for the semantic parse stage[23].

2.6 Who's snoring now?

This has been a bit dragged out. Have a look in **/usr/local/yacco2/docs** to see the 2 *Cweb* generated grammars: o2_lcl_opts and o2_lcl_opt along with the other grammars of O_2. They are in "PDF" and postscript formats. Though I've haven't shown how a threaded grammar is referenced and called within a grammar, this will be addressed in the Threads chapter 7.

I hope you can conclude from this introduction to grammars that structuring a parser is quite simple. Enticements and promises have been made to u that the following chapters hope to deliver on. So off with your head into slumberland and see u in terminals: does anyone real know what time it is? — You're tired and so am i sayranara.

[23] Your semantics Dave better be interesting. Or here comes those clowns — soft shuffle do-da do-dah, i gotta keep the white cane from hooking me off the stage.

Chapter 3

tʒ:min(ə)ls

Here's the terminal anatomy.

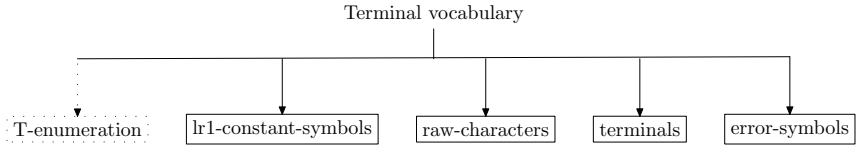


Figure 3.1: Skeleton of O_2's terminal vocabulary

The *T-enumeration* is drawn with dashed lines to indicate that it is not formally part of a grammar's vocabulary but an artifact to O_2's implementation. The order of their appearance is left-to-right and processed top/down and bottom-up. The only reason for this order is due to an optimization of assigning the symbol's enumeration number at the time of its appearance instead of post assigning it after all the vocabulary types have been seen. From the above diagram, why is the Terminal vocabulary divided into 4 classes? There is nothing profound. It just worked out that terminals divided logically into these 4 groups: meta-terminals to describe various parsing situations, the raw characters, your own take, and finally error reporting. From O_2's development and experience this segregation seems right. Sorry for the letdown to chapter one's "promises and expectations" as one can see the hype didn't match the deliverables.

Each terminal box has the same defining pattern similar to the "c" language procedure definition: the name of the procedure, its formal arguments enclosed in "(" and ")" parentheses followed by its defining code body contained by the open / close brace pairing. Here's an example of an Error terminals vocabulary without its content:

error-symbols (*file-name* `yacco2_err_syms`,*name-space* `NS_yacco2_err_syms`)
{ $\sum$ each error terminal definition }

The 2 arguments are for code generation purposes. *file-name* provides the file-name prefix used in generating the C++ files of "h" and "cpp" file extensions. *name-space* supplies the C++ namespace enclosing this terminal definition. Peeking back to *Indirections take two — globals.h* on page 17, u'll see the use of C++ namespace directives to some of these vocabularies: `NS_yacco2_T_enum`, `NS_yacco2_k_symbols`, and `NS_yacco2_terminals`.

I'll start the discussion by generalizing symbol definitions. Then i'll describe the canned definitions leaving *T-enumeration* scheme for the last comments. It merits some verbiage as it does have a logic directive, though rarely used completes your understanding.

3.1 k² on symbols

The basic terminal definition is a 2 part tupple: the literal name and its associated C++ component. All terminals use this pattern. Most of your error terminals use this pattern without adding any additional code.

Here's an example:

```
"not a lhs kw" (sym-class Err_not_a_lhs_kw)
```

The literal `"not a lhs kw"` is the symbol referenced in a grammar's rhs. It is contained within the paired double quotes `"`. The *sym-class* is the keyword to introduce the C++ class name for its code implementation that is *Err_not_a_lhs_kw*. Grammars reference the literal part while "syntax directed code" reference the C++ part.

The bracketing open/close parentheses is an enclosing structure that allows u to customize your own code injection points for the terminal definition. When u go this route, all C++ code must be coded by the grammar writer. The liberties come with some effort through code directives. These custom directives get placed after the class-name value. Here's a simple customized terminal from the *lr1-constant-symbols* class.

```
 eog
/@
Used to indicate an end-of-grammar or an end-of-file condition.
When the end of the container is reached, calls for another terminal
will always return the |eog|.
It's your door bouncer before hell.
@/
(sym-class LR1_eog
 { // start of customization
   user-declaration    // code directive
    LR1_eog(); // ctor
   ***                 // end of code directive

   user-implementation // code directive
    LR1_eog::LR1_eog()
    T_CTOR("eog",T_LR1_eog_,0,false,false){}
    LR1_eog LR1_eog__;
    yacco2::CAbs_lr1_sym* yacco2::PTR_LR1_eog__ = &LR1_eog__;
   ***                 // end of code directive
 } // end of customization
)
```

How come the *eog* literal name is not enclosed with double quote `"`? It could be but if the literal does not contain spaces, it can be unquoted. Now what are those /@ ... @/ comments? Well u can graffiti your own subconscious using *Cweb* comment directives. Let's sit back a bit and reflect — here is a terminal being commented on, does this mean that there are generated "pdf" documents per terminal class? Yep. All parts of a grammar have *Cweb* styled documents.

Following the *sym-class* name is the optional open/close braces to bracket these customized directives and their C++ code. The 2 code directives are *user-declaration* and *user-implementation*. As noted before, each code directive is terminated by its "`***`". Not to belabour u with comment stuttering, the above example should sigh post u. Now on to the more interesting part: your own creativity.

3.1.1 Recycling those Tes

I raise the question: When do Tes get put back to heap heaven? This is very open question to the point of being stupid. I'll try to explain it by a prepositional context: it depends on the life-span of each T. Man that foot in your mouth is becoming a fixture. A T is birthed by C++'s `new` verb. Tes can hang around until your process stops running and it is left to the operating system to deal with its memory footprint. Or there are 2 delete attributes per symbol: AB and AD. AB deals with the abort situation while AD deals with popping the symbol off the parse stack. They are both optional and are indicated by the "[" and "]" brackets in the below T definition.

```
"not a lhs kw" [AB] [AD] (sym-class Err_not_a_lhs_kw)
```

As they were not present in the 1st example, this means that `"not a lhs kw"` T is not depending on these delete attributes for its recycling. I'll try to give each an example when these attributes might come in handy.

The aborting of a grammar:
This can take place when more than 1 threaded grammar is competing and one or more grammars might not complete successfully. There could possibly be remnants of Tes laying on their parse stacks and u want to ensure that your house is in order. I know it's a marginal example; it allows your partially built structure to be dismantled. It is the generic Parser that does the cleaning.

The ridding of a T when popped from the parse stack:
Possibly the T is a carrier of another T. Its contents are extracted and so it can be recycled to the memory heap. Before the `|+|` symbol was created, a called thread returning more than one type of T used this approach to lower the calling grammar's number of subrules fielding each specific returned T. This carrier pigeon approach was originally done by the "identifier" threaded grammar that contained the symbol table lookup to resolve the general identifier into its possible keyword T. It returned a T of "identifier", and its discovered O_2's keyword as content. Again stretched as a good example but u have some control over the T's life cycle. Using syntax directed code, u can dynamically turn "on" or "off" the stacked symbols's AB, or AD delete attributes by the following methods:

AB method `set_affected_by_abort(boolean Value)` — true or false
AD method `set_auto_delete(boolean Value)` — true or false

U can query the symbol's attributes by the following methods:

AB `boolean affect_by_abort()` — returned true or false
AD `boolean auto_delete()` — returned true or false

One warning: be careful that u haven't containered the deleted symbol and that container's contents are read again.

3.2 Roll-your-own Directives

All directives are expressions of where to put its content within the C++'s structure that gets built. I purposely did not mention *class* as O_2 evolved it became apparent that C++ *struct* was more appropriate. There is no order to their appearance except for impersonators that are caught. There are 2 parts to a terminal, the struct definition or prototype, and the structure's implementation. I suffixed the directive's intent by either *declaration* or *implementation* where applicable. Here is a list of directives:

- *user-declaration*: definitions of variables or methods.
- *user-implementation*: implementation details of the declaration.
- *destructor*: code for a simulated C++ dtor method.
- *constructor*: code that gets injected into the C++ struct's ctor method. Not used in a terminal definition. Its utility is for *rule* definitions and the *fsm* construct.
- *op*: a method that is executed each time a grammar definition is used. This occurs when it is first created and whenever it is reused. A terminal definition does not support it. Its utility is for *rule* definitions and the *fsm* construct as each one supports reuse.

Other grammar constructs like *fsm* or *rule* use this same directives pattern. I'm mantraing for later remembrance recall.

3.2.1 *user-declaration* directive

It allows u to add methods and variables to the definition. C++'s class privacy considerations are allowed. It's up to the grammar writer taste or efficiency constraints as to going public. In building C++ definitions, i was very much concerned about speed. Having a very safe but sluggish compiler/compiler would have defeated its usefulness. "My notes to myself" within O_2's library details these saw-offs: virtualized classes and inheritance were minimized as hoards of terminals can be overwhelming to your cpu and your patience.

Here's an Error terminal example *Err_nested* definition and its C++ progeny with 2 private variables and public methods to access them. It gets created when "recursive file" includes exceeds some predetermined threshold.

```
// grammar definition
"nested files exceeded"
(sym-class Err_nested
 {
   user-declaration
    public:
     Err_nested
         (yacco2::INT Nested_file_cnt,std::string& File_name);
     yacco2::INT nested_cnt(){return nested_cnt_;};
     std::string* file_exceeded(){return &file_exceeded_;};
    private:
     yacco2::INT nested_cnt_;
     std::string file_exceeded_;
   ***
   user-implementation
    Err_nested::
    Err_nested(yacco2::INT Nested_cnt,std::string& File_name)
    T_CTOR("nested files exceeded",T_Enum::T_Err_nested_,0,false,0)
     {nested_cnt_ = Nested_cnt;
      file_exceeded_ += File_name.c_str();}
   ***
 }
)

 // c++ emitted definition
  struct Err_nested_files_exceeded:public yacco2::CAbs_lr1_sym{
   public:
    Err_nested
         (yacco2::INT Nested_file_cnt,std::string& File_name);
    yacco2::INT nested_cnt(){return nested_cnt_;};
    std::string* file_exceeded(){return &file_exceeded_;};
   private:
    yacco2::INT nested_cnt_;
    std::string file_exceeded_;
  };
```

The example explicitly declares the privacy used for learning purposes. Its variables are `nested_cnt_` and `file_exceeded_`. `yacco2::INT nested_cnt()` and `std::string* file_exceeded()` are access methods. The *Err_nested* terminal is the C++ type and constructor name with arguments `Nested_file_cnt` and `File_name` to be supplied when this type is created as an object. From the example, the mapping is very obvious between the grammar definition and its C++ counterpart. The reason for the inheritance to `CAbs_lr1_sym` symbol is to make all symbols uniform in their identity attributes: "who am i" in literal terms and in C++ enumeration terms. The GPS co-ordinates are built into the abstract symbol.

The class construction is the *sym-class* value: *Err_nested*. Again due to efficiency reasons, the symbol is defined as a *struct* instead of a *class*. I did not want all the implicitly generated class *ctor* and *dtor* baggage that can come with a C++ compiler. If u want it then explicitly write it yourself.

3.2.2 *user-implementation* directive

Here's the C++ code from the above *user-declaration* example:

```
Err_nested::
Err_nested(yacco2::INT Nested_cnt,std::string& File_name)
 T_CTOR("nested files exceeded",T_Enum::T_Err_nested_,0,false,0)
 {
  nested_cnt_ = Nested_cnt;
  file_exceeded_ += File_name.c_str();
 }
```

I included some subtlety as the 2 access methods are defined in the *user-declaration*. This is another way that C++ allows one to define and implement within the same definition though exposing its implementation at the header file level. It eliminates the need for a secondary implementation file. The mapping comments between the grammar and its C++ implementation is 1:1 but what is T_CTOR in the *Err_nested* constructor? It is a macro that initializes the inheritance symbol `CAbs_lr1_sym`.

3.2.3 The `T_CTOR` macro — cross those Tes

Here are its passed arguments explained:

1. Terminal's literal value. Comes in handy for tracing and messaging.
2. Enumerated value used in lookahead and follow sets. It is the most efficient way to deal with items and set membership.
3. Address of its destructor method. Why is this not as a class *dtor* which should handle this? The overhead was too high in minimal use by defaulting to *class*'s implicit behaviour. This is why *struct* is used to define a terminal.
4. Boolean value whether to automatically delete it when popped off the parse stack.
5. Boolean value whether to delete it when an abort action occurs within a parse. It's a "clean up your own mess" attitude when parsing. It takes on a role when parallel parsing happens using threads. Remember, this is nondeterminism at work.

Details, details, and more details. That mad hatter and rabbit sure have reason to mumble. At least u the grammar writer have control over what u want to create. It just comes with a bit-of-assembly and your own mumblings.

3.2.4 *destructor* directive

When needed for special cleanup duties, this is your directive. Here is an example of use as its utility is rare. It includes the grammar definition, and C++'s definition and implementation.

```
"basic-char"
/@
Basic source character set indicator.
It can be used to assess the good, the bad, and the ulgy
character traits.
For example, prescan the characters against the
|bad_char_set| thread and return one of the 2 terminals.
It is left here but not used.
@/
 (sym-class  T_bchar{
  user-declaration
    public:
     T_bchar(CAbs_lr1_sym* Basic_char);
     CAbs_lr1_sym* basic_char()const;
     void zero_out_basic_char();
    private: CAbs_lr1_sym* basic_char_;
  ***
  destructor
   if(R->basic_char_!=0) delete R->basic_char();
  ***
  user-implementation
    T_bchar::
    T_bchar(CAbs_lr1_sym* Basic_char)
     T_CTOR("basic-char",T_Enum::T_T_bchar_,&dtor_T_bchar,false,0)
     {basic_char_ = Basic_char;}
```

```
    void T_bchar::zero_out_basic_char(){basic_char_ = 0;}
    CAbs_lr1_sym*
    T_bchar::basic_char()const{return basic_char_;}
  ***
  }
)

// c++ definition
  struct T_basic_char:public yacco2::CAbs_lr1_sym{
   public:
    T_bchar(CAbs_lr1_sym* Basic_char);
    CAbs_lr1_sym* basic_char()const;
    void zero_out_basic_char();
    private:CAbs_lr1_sym* basic_char_;
    static
    void dtor_T_bchar(yacco2::VOIDP This,yacco2::VOIDP P);
  };

// c++ implementation
  T_bchar::T_bchar(CAbs_lr1_sym* Basic_char)
   T_CTOR("basic-char",T_Enum::T_T_bchar_,&dtor_T_bchar,false,0)
   {basic_char_ = Basic_char;}
  void T_bchar::zero_out_basic_char(){basic_char_ = 0;}

  CAbs_lr1_sym* T_bchar::basic_char()const{return basic_char_;}

  void
  T_bchar::dtor_T_bchar(yacco2::VOIDP This,yacco2::VOIDP P){
   bool ABORT_STATUS =
       ((yacco2::Parser*)P)->top_stack_record()->aborted__;
   T_bchar* R = (T_bchar*)(This);
   if(R->basic_char_!=0)
    delete R->basic_char();
  }
```

`T_bchar::dtor_T_bchar(yacco2::VOIDP This,yacco2::VOIDP P)` destructor's implicit additional code is due to the wind-down of C++ delete operation. The destructor is not part of the terminal *struct* definition but defined globally static. Why? u got it efficiency reasons [1]. So the memory chunk is going to recycled heaven and the called destructor knowns only that it's a thunk-of-something, and some *Parser* initiated the problem. So O_2 must do resuscitation: aka by casting of its known type. The second parameter provides the *Parser* run environment and it too needs CPR. Casting its context activates the parse stack for evaluation.

3.3 Canned terminals: *lr1-constant-symbols*

Though there is a special directive in *lr1-constant-symbols*, it is not meaningfull to your understanding though for the curious have a look at `yacco2_k_symbols.T` file. Now we'll explore those constant terminals. Some of their comments are advanced for your understanding. They are there to flavour your thoughts for future discussion in chapters to come. There are eight terminals:

1. `"|?|"` represents a questionable grammar situation. It pinpoints programmed error points within the grammar. The subrule using this symbol has a lr(0) reduction as the lookahead is not kosher and so would probably not reduce in the lr(1) context. It can be used both in the following grammar expressions:
   1) → `|?|`
   2) → `|||  |?|` NULL // thread call expression

[1] That diary of yours Dave better be juicy with interesting tidbits.

Point 1 covers the state where the current token being parsed is improper. Point 2 is more interesting as it captures a returned terminal that the thread passes back as an error.

The `|?|` was not one of the original "k" terminals. It replaced the "eof" terminal which was marginal in intent. I felt the `|?|` symbol drew the reader's eye within the grammar where "faulty" points where captured and to force lr(0) context processing to reduce its subrule. Why lr(0) context? Glad u asked, the lookahead terminal — the current terminal being parsed, is in error and so "how is the subrule with the `|?|` to reduce after its shifted T?". It must be divorced of any lookahead and just acted upon. Now another question arises: "how is this condition detected in a parsing state of mixed conditions — threading, shifting, reducing"? There is a pecking order on the conditions tried by the parser:

- threading
  if tried and unsuccessful the balance of conditions are attempted
- shifts pecking order by their presence in current parse state:
  can the current token be shifted?
  `|?|` — error condition
  `|.|` — explicit ϵ
  `|+|` — any terminal from the all T classes
- reduce
  note shifting is favoured over reducing

2. `eog` indicates an end-of-grammar or an end-of-file condition. When the token container is reached, calls for another terminal will always return the `eog`. It's your door bouncer before purgatory.

3. `eolr` indicates all-terminals of the terminal vocabulary including itself. It saves finger blisters by not having to be explicit in the thread's lookahead expression. Dieting hasn't been this effective to code bloat.

4. `"|||"` presence within the individual state of the "fsm" table dictates potential threads to run. You see it sprinkled throughout my grammars to call threads. This is part of O_2's raison d'être.

5. `"|r|"` presence within the individual state of the "fsm" table is to force a reduce operation. Why? it's a back-to-back situation within the state table whereby a thread should reduce while its reducing lookahead is the `|||` indicating to run a thread. U never program with it or u'll receive the wrath of O_2. It is reserved for outputted state tables.

6. `"|.|"` is a nice way to program out of an ambiguous grammar. It can also lower the code bloat of a thread's first set.

7. `"|+|"` represents the wild token situation. Lowers the specific shifts of the finite-state-table and allows the grammar writer to field the unexpected from returned threads. Good stuff.

   Caveat: One should use the `|?|` to field unknown Tes if they are to be interpreted as errors.

8. `"|t|"` has split personalities: used in O_2linker to process the transient first sets generated by threads, and used within a grammar's "chained call procedure" expression to lower thread overhead by calling a procedure with explicit intent on double use of its "first set" token. I'll give an example of a "chained procedure call" expression drawn from the "pass3.lex" grammar handling the grammar's file include expression:

   → "`@`" Rprefile_inc_dispatcher

   The "Rprefile_inc_dispatcher" grammar rule has the following subrule:

   → `|t|` "file-inclusion" NS_prefile_include::PROC_TH_prefile_include

   The "chained" part is in the duplicating of "`@`"; that is, the parsing mechanism does not get a new terminal when shifted but passes this T to the called procedure. The called PROC_TH_prefile_include procedure / thread has its start rule's subrule as:

→"@" Rpossible_ws Rfile_string Reof

The repeated use of "@" is to reenforce the idea that the procedure called cuz of "@": there's that "first set" again. Well time will pass its comments on this thought process.

All these terminals are not part of the token stream. They indicate various situations within the parse tables of the grammar. eog is a slight exception as it indicates the end-of-the-token-stream regardless of container source. Its canned object is placed twice into the container for parsing reasons [2]. Now all user terminals are brought to life through the new verb. They come from a memory heap and are transformed into the requested object. As part of the carbon footprint on the memory heap, eog is globally registered as a pointer: `extern yacco2::CAbs_lr1_sym* yacco2::PTR_LR1_eog__;` It is this reference that all containers use as their token indicating the end-of-something. So what is its GPS that u've been toting? Well there ain't any due the recycle-referencing of it in all containers. Reality is it's just a stop-as-i-want-to-get-off indicator. The other global pointers are:

```
extern yacco2::CAbs_lr1_sym* PTR_LR1_parallel_operator__;
extern yacco2::CAbs_lr1_sym* PTR_LR1_fset_transience_operator__;
extern yacco2::CAbs_lr1_sym* PTR_LR1_invisible_shift_operator__;
extern yacco2::CAbs_lr1_sym* PTR_LR1_questionable_shift_operator__;
extern yacco2::CAbs_lr1_sym* PTR_LR1_all_shift_operator__;
extern yacco2::CAbs_lr1_sym* PTR_LR1_eolr__;
```

Their value comes not in their object use but as global names in referencing their enumerated ids for syntax-directed code purposes. This will come out in later chapters when "variations on syntax directed code" are played[3].

3.4 Canned terminals take two: *raw-characters*

They are plain mundane. Their only sparkle comes in their object implementation efficiency. Its an array of 256 fixed elements that are hardwired for delivery speed. I did not want to trash the memory heap. The external file number and raw character position of the GPS co-ordinates get overridden at the access time a new repeat text character is asked for: ie a container knows whether the token request is already inside itself or whether it must fetch the next external character. This is a "just-in-time" approach to character symbol delivery. It is not perfect as the character symbol is the same for all references to it and only the last physical text GPS is valid. So how does one maintain a raw character original gps? This is addressed later in the book. I'll tease u with this clue — morphing: chapter 6 Parser describes the "start_token" method and its use towards this end. The other claim-to-plain is how their grammar literal name encodes across the 2^8 bit spectrum. It uses an improvised variant of hexadecimal encoding for bytes drawn from the "c" language that are not displayable: `"xyz'` where the x indicates that the expression is hexadecimal encoding followed by the 2 hexadecimal digits represented by yz drawn from 0 – 9, a – f. Other displayable characters in the 32 – 126 decimal range use their ASCII representation. Not to belabour this, here are some examples showing the 2 literal types:

```
"x00" (sym-class raw_nul{
  user-declaration
    public:
    raw_nul(INT Ext_file,size_t Pos);
  ***
  user-implementation
    raw_nul::raw_nul(INT Ext_file,size_t Pos){
      T_CTOR_RW("x00",T_raw_nul_,false,false,Ext_file,Pos)}
  ***
  }
)
```

[2] What is this another guessing game? Think reduction as in cooking?

[3] Can u guess why eog is not part of the other pointer definitions though contained within the same namespace? Circular dependencies: O_2 program versus O_2 library and its eog use.

```
"\"" (sym-class raw_dbl_quote{
  user-declaration
    public:
    raw_dbl_quote(INT Ext_file,size_t Pos);
  ***
  user-implementation
    raw_dbl_quote::raw_dbl_quote(INT Ext_file,size_t Pos){
      T_CTOR_RW("\"",T_raw_dbl_quote_,false,false,Ext_file,Pos)}
  ***
})

"T" (sym-class raw_T{
  user-declaration
    public:
    raw_T(INT Ext_file,size_t Pos);
  ***
  user-implementation
    raw_T::raw_T(INT Ext_file,size_t Pos){
      T_CTOR_RW("T",T_raw_T_,false,false,Ext_file,Pos)}
  ***
})

"xfe" (sym-class raw_xfe{
user-declaration
  public:
  raw_xfe(INT Ext_file,size_t Pos);
***
user-implementation
  raw_xfe::raw_xfe(INT Ext_file,size_t Pos){
      T_CTOR_RW("xfe",T_raw_xfe_,false,false,Ext_file,Pos)}
***
})
```

Appendix B page 253 provides a complete list of the "lr k" and raw characters grammar symbols with their literal name, C++ structure, and enumerate label.

3.5 *T-enumeration* and counting

Before the specifics are detailed, let's look at what is to be achieved. All grammar symbols have unique name tags that cover a grammar's vocabulary composed of Terminals and Rules (rules are more relaxed as they are isolated within their grammar). These name tags are used in 2 different settings: the grammar, and its C++ code implementation. From an efficiency perspective the literal name tag is great for reporting purposes but doesn't make the grade in runtime performance. So a symbol carries 2 tags: tongue wagging, and runtime. I felt carrying both were reasonable in space and performance. This enumeration scheme maps the symbols into the positive whole numbers of 2^{16} bit size. Here is the numeric mapping across the 4 Terminal classes:

0 <= lr k <= 7 < raw characters < 264 <= user Tes < Errors

Numbers for efficiency are fine but how should one reference them out of syntax directed code? Before i answer this why should one need to specifically probe the identity of a symbol? Later in the book u'll see how wild tokens are programmed, referenced, and ranked. Just to entice u there are no blisters on my fingers due to this economy. Back to the first question, this is where the enumeration construct comes in. It is a symbol dictionary where each grammar symbol represents its numeric value.

3.5.1 2 2 2 in one enumerates

When writing the O_2 library, the token containers referenced the "lr constant" symbols. These references covered both their enumerate value and literal names like `eog`. At the beginning of development, the symbols order were different, experimented with, and there was a hardwired base of error terminals to deal with token

containers tantrums and error reporting. This was not acceptable. I did not want the user of O_2 to carry extra mental baggage caused by my implementation shortcomings. Now the order is acceptable with the 2 canned terminal definitions and liberties given to the grammar creator. There is one caveat though: As i use namespaces to segregate the vocabularies, when writing your own vocabulaties there are dual namespaces dealing with T-enumeration: the base enumeration that i bootstrapped with for the O_2 library and your own T-enumeration. O_2 library uses the first 2 terminal classes: LR constants and Raw characters. Lets stop messing around. Get to the point Dave. Here's a partial copy of O_2's emitted enumeration scheme.

```
//
// file: yacco2_T_enumeration.h
// Gened Date and Time: Fri Sep 30 12:19:52 2005
//
#ifndef __yacco2_T_enumeration_h__
#define __yacco2_T_enumeration_h__ 1
namespace NS_yacco2_T_enum{
struct T_Enum{
  enum enumerated_terminals {
   sum_total_T = 543
   ,no_of_terminals = 114
   ,no_of_raw_chars = 256
   ,no_of_lr1_constants = 8
   ,no_of_error_terminals = 165
,start_LRK=0,end_LRK=7
,start_RC=8,end_RC=263
,start_T=264,end_T=377
,start_ERR=378,end_ERR=542
,start_R=543
  ,T_LR1_questionable_shift_operator_ = 0
  ,T_LR1_eog_ = 1
  ,T_LR1_eolr_ = 2
  ,T_LR1_parallel_operator_ = 3
  ,T_LR1_reduce_operator_ = 4
  ,T_LR1_procedure_call_operator_ = 7
  ,T_LR1_invisible_shift_operator_ = 5
  ,T_LR1_all_shift_operator_ = 6
  ,T_LR1_fset_transience_operator_ = 7
  ,T_raw_nul_ = 8
  ,T_raw_soh_ = 9
  ...
  ,T_raw_us_ = 39
  ,T_raw_sp_ = 40
  ,T_raw_exclam_ = 41
  ,T_raw_dbl_quote_ = 42
  ,T_raw_no_sign_ = 43
  ...
  ,T_raw_xfe_ = 262
  ,T_raw_xff_ = 263
  };//close enum
};// close template
}// namespace
#endif
```

I'll go backwards to describe the scheme. It maps to a "c" enum type: `enum enumerated_terminals`. Each enumerate symbol is assigned an increasing number. Also included in the enum list are summary symbols about the grammar vocabulary: number of symbols per class, their total number, each class's beginning and ending values. They are emitted per grammar environment and available for your C++ code. I included other symbols to give u a glimpse into the complete symbol mapping scheme. `start_T=264` is the starting point for the user terminal symbols due the number of canned "lr constant" and raw character symbols. `end_T=???` end point depends on the number of terminals defined within your grammar. All

the following symbols will take on their own specific values according to your terminal definitions; the bootstrapped example has these assigned values:

- `end_T` = 377
- `start_ERR` = 378
- `end_ERR` = 542
- `start_R` = 543

Though we're discussing Terminals, Rules builds on the enumeration scheme following Terminal ordering. The `start_R` symbol is used as an anchor point to the *fsm* generation and its local rule definitions.

Namespace `NS_yacco2_T_enum` with a struct `T_Enum` encloses the enumeration type. They are the qualifiers to access these symbols. Your own implementation will use your own namespace drawn from its emitted file. The below example uses my *T-enumeration* definition, u would substitute your own *namespace* label for your own specifics in place of `NS_yacco2_T_enum`:

```
NS_yacco2_T_enum::T_enum::start_T
```

You can use C++ `using namespace xxx;` directive but be warned that different C++ compiler implementations might jump u. I've found this is the safest way as i was ambushed in different ports due to ambiguity between the 2 definitions: my *T-enumeration* definition and O_2's library included definition. Why? This is due to each C++'s compiler's symbol scope lookup. Compiler/compiler O_2 is like your own compiler, it also includes Yac$_2o_2$'s support library.

3.5.2 U call it, i call it: the symbol's enumerate name

From the list above u can deduce that the label uses its C++ *sym-class* name and prefixes it with `T_` and appends a _. The `eog` symbol enumerate label is `T_LR1_eog_`.

I wasn't too creative at the time of implementation. It is adequate. For the "c" programmer in u, u'll curse me for lack of brevity but truth is told.

3.5.3 *T-enumerate* `constant-defs` directive

As *T-enumerate* is the first definition processed and emitted, it becomes a perfect place to define constants and to globalize definitions across all the emitted files. Can u give me an example of this? This is drawn from a Pascal translator where different grammar threads were processing expressions of different base number types.

```
T-enumeration
(file-name pas_T_enumeration
,name-space NS_pas_T_enum) {
constant-defs
enum RADIX_TYPE{dec_radix,hex_radix,octal_radix};
***
}
```

Well not too enlighening Dave! You're now aware and guide yourself accordingly. The only advice i can give is draw somehow that dependency graph of definitions to see if it is required.

3.6 Abstracting the abstract: `CAbs_lr1_sym`

All symbols of a grammar inherit its genes. Rephrased, a grammar's symbols are its Tes and Res: rules. C++'s inheritance facility is a nice way to pass on green eyes, black hair, and other traits that a compiler/compiler expects from symbols. What is this primordial thing?

3.6.1 Traits revealed

Here is a document extract from O_2's library. `CAbs_lr1_sym` is your base structure from which all grammar symbols of terminal and rule alphabets are derived. Two symbol identities are maintained: description and enumeration. The descriptive form is its name defining itself in its vocabulary and referenced in the grammar's subrules. While the enumeration id depends on how Yacco2 has iterated across the Terminal alphabet. This iteration is described elsewhere.

To save space, an union structure is used between the co-ordinate of a terminal and the rule's associated number of right-handside elements (subrule) and parser context. At one time there was a distinction of generated symbols for the rule and its subrules. Now a subrule is a method within the rule's class. The utility for separate symbols for rules and their subrules was evaluated. The cost of the extra subrule symbols was too heavy in the little utility that they gave but rarely exercised!

A rule and the lrk constants terminals have no association with the token source stream, only terminals do in their various forms — error, raw characters, and user defined. The source file co-ordinates are expressed in terms of a line number and a character position within the line. A file number index is kept as a key into the global table of copied files that holds their file names.

The balance of the variables are grammatical attributes: 'auto delete', 'auto abort', and its destructor function if present. Why is there a dtor function instead of a class destructor. Efficiency! Virtual tables can be expensive in space and time. In this case, it is not needed very often and it is controlled by Yacco2's output code. Remember there are hoards of symbols: at least one per character.

I've added the terminal's compressed set key to speed things up for the lookahead set operations. Some parsing operations use the raw enumerate value as it is a 1:1 in content. Lookahead sets are composed of sorted tuples composed of a partition number and its elements members derived from the terminal's enumerated value. This eliminates the calculation of a terminal's enumerate value to its set equivalent every time it is checked for set membership.

```
struct CAbs_lr1_sym {
  CAbs_lr1_sym(          // ctor 1 : raw characters
     yacco2::KCHARP      Id
     ,yacco2::FN_DTOR    Dtor
     ,yacco2::USINT      Enum_id
     ,bool               Auto_delete
     ,bool               Affected_by_abort
     ,yacco2::USINT      Ext_file_no
     ,yacco2::UINT       Rc_pos);

  CAbs_lr1_sym(          //ctor 2: Tes --- lrk, error, and regular T
     yacco2::KCHARP      Id
     ,yacco2::FN_DTOR    Dtor
     ,yacco2::USINT      Enum_id
     ,bool               Auto_delete
     ,bool               Affected_by_abort);

  CAbs_lr1_sym(          // ctor 3: grammar rule
     yacco2::KCHARP      Id
     ,yacco2::FN_DTOR    Dtor
     ,yacco2::USINT      Enum_id
     ,yacco2::Parser*    P
     ,bool               Auto_delete=false
     ,bool               Affected_by_abort=false);

  yacco2::KCHARP  id()const;                          // literal id
  yacco2::USINT   enumerated_id()const;               // number id
  void            set_enumerated_id(yacco2::USINT Id);
  void            set_auto_delete(bool X);
  bool            auto_delete()const;
  void            set_affected_by_abort(bool X);
  bool            affected_by_abort()const;
```

```
  yacco2::UINT    rc_pos();
  void            set_rc_pos(yacco2::UINT Pos);
  yacco2::UINT    external_file_id();
  void            set_external_file_id(yacco2::UINT File);
  void            set_rc(yacco2::CAbs_lr1_sym& Rc
                        ,yacco2::KCHARP GPS_FILE=__FILE__
                        ,yacco2::UINT GPS_LINE=__LINE__);
  yacco2::UINT    line_no();
  void            set_line_no(yacco2::UINT Line_no);
  yacco2::UINT    pos_in_line();
  void            set_pos_in_line(yacco2::UINT Pos_in_line);
  void            set_line_no_and_pos_in_line(yacco2::CAbs_lr1_sym& Rc);
  void            set_line_no_and_pos_in_line(yacco2::UINT Line_no
                                             ,yacco2::UINT Pos_in_line);
  void            set_who_created(yacco2::KCHARP File
                                 ,yacco2::UINT Line_no);
  yacco2::UINT    who_line_no();
  yacco2::KCHARP  who_file();
  yacco2::Parser* parser();
  yacco2::FN_DTOR dtor();
  yacco2::USINT   rhs_no_of_parms();
  yacco2::KCHARP  id__;
  yacco2::FN_DTOR dtor__;
  yacco2::USINT   enumerated_id__;
  bool            auto_delete__;
  bool            affected_by_abort__;
  UCHAR           enum_id_set_partition_no()const;
  UCHAR           enum_id_set_member()const;

  struct tok_co_ordinates{
   yacco2::KCHARP who_file__;
   yacco2::UINT   who_line_no__;
   yacco2::UINT   rc_pos__;
   yacco2::UINT   line_no__;
   yacco2::USINT  external_file_id__;
   yacco2::USINT  pos_in_line__;
   Set_entry      set_entry__;
  };
  struct rule_info{
    yacco2::Parser*  parser__;
    yacco2::USINT    rhs_no_of_parms__;
  };
  union {
    tok_co_ordinates tok_co_ords__;
    rule_info        rule_info__;
  };
};
```

I'm mantraing again on constructors for the abstraction. There are 2 types of symbols: Tes and rules. Tes are distinguished by raw characters due to their immediate affiliation to a source file, and the 3 other T classes: error, and meta terminals must depend on the raw characters for their source file coordinates while the lrk class indicates only parsing conditions as they are not part of the token stream. Errors and meta terminals get GPS associated by a code directive like `op` and use of the above `rc` procedural variants. The variants allow u to use another T's co-ordinates, to explicitly associate its where-abouts by line and character position within an external source file. Replayed, only Tes are related to a physical location in some source file as generically associating a rule to its subrule token stream just didn't make sense. If this is required, the rule's definition would be enhanced by declaring these extra variables.

`CAbs_lr1_sym`'s variables can be directly accessed if u want speed or indirectly thru their named access

methods. U'll see mixed variants throughout the book's examples. All symbols are distinguished by their enumerated ids: `enumerated_id__`. `id()` supplies a symbol's literal name for tracing purposes. To note, rules/subrules enumerates are locally defined in each grammar's `fsm` definition. O_2's library keeps an opened files dictionary by open order. It provides the file's literal name for tracing and error reporting purposes where the GPS of the T identifies its source character number within the line.

3.6.2 Quirks on symbols, rules, and consistency

Why the symbol's overlay of information between the Tes and rules when they are not related? Good question. I tried to give a unified interface to all the symbols within a grammar by `CAbs_lr1_sym` using C++ inheritance. As the grammar's vocabulary are independent by definition, it made sense to minimize the space used per symbol but maximize their consistency. A rule's main contribution is its `parser__` variable or like method `parser()` while a T's contribution is its source file coordinates. An example please.

```
Rangled_string  () {// grammar rule definition
  -> Ropen_angle |.|{
  op
    Cangled_string* fsm =
                  (Cangled_string*) rule_info__.parser__->fsm_tbl__;
    CAbs_lr1_sym* sym = new T_angled_string((const char*)&fsm->ddd_);
    sym->set_rc(*rule_info__.parser__->start_token__
               ,__FILE__,__LINE__);
    RSVP(sym);
  ***
  }
}
// c++ header declaration
struct Rangled_string:public yacco2::CAbs_lr1_sym {
                      ^^^^^^^^^^^^^^^^^^^^^^^^^^
                      // use of inheritance
  Rangled_string(yacco2::Parser* P);
  public:
  void sr1();
};

// c++ implementation showing CAbs_lr1_sym ctor for a rule
Rangled_string::Rangled_string(yacco2::Parser* P)
 :CAbs_lr1_sym
  ("Rangled_string",0,Cangled_string::R_Rangled_string_,P,false,false)
  // applying inheritance by ctor for a rule                                    ^
  //  where P is the rule's soul mate
  // gets placed in rule_info__.parser__ field of symbol
{}
```

Here `Rangled_string` rule is in 3 forms: grammar definition, and the 2 C++ variants. The above example shows use of inheritance giving symbol consistency by `CAbs_lr1_sym`'s methods/variables.

3.7 Summary

The vocabulary framework was developed from both perspectives: grammar and C++ implementation. 3 out of four Terminal classes were elaborated. *terminals* was intentionally left out as their creation is the same as Error terminals described. U have more than a notion on how to declare terminal classes and to define them symbols. So off with your fingers as u know how to count and see u in *fsm*.

Chapter 4

fsm — finite state machine

Ahh the envelope that holds the grammar. It doesn't hold the Terminals. They are self contained in their own namespace cocoons. So what is this statement: grammar holder? And what an acronym? And *rules* haven't been covered yet either! So Jack be nimble Jack better be quick and all the kings men couldn't put Dave together again. Enough musical theatrics. *fsm* is the dumping ground for rules and productions of the grammar. It supplies the emitted C++ class name and enveloping namespace of the grammar type. It has many directives for u the grammar writer to place code triggers and scribbles from their actions. So it is the registrar of your grammar thoughts.

Now to scribble: Here are the C++ code placement directives with their brief summaries.

- *failed* last chance to capture a failed parse of a grammar **when the grammar aborted**. It allows the grammar writer to test various conditions and to post an error. The error posting can be returned back to the calling grammar or placed within the error container for post evaluation.

- *user-prefix-declaration* code placed before the *fsm* definition. Usually include statements and global variables.

- *user-suffix-declaration* code placed after the *fsm* definition.

- *constructor* code that gets executed when the *fsm* constructor initializes.

- *destructor* wind-down code that is executed when the *fsm* is being recycled somewhere. Currently not deployed. Due to rule use optimization, there is a table of **newed rules** created per grammar instance according to their parsing situation that should be returned back to memory stock when the grammars are recycled. At the moment, i default to process wind-down hygiene to recycle back heap items. So its only value is as a last point of execution to register something with something. It will be left out in the following explanations. Towards the end of the book, "Pieces of thought" in chapter 13 will discuss things like destructors.

- *op* is a method that is always executed when the *fsm* object is called. It's a pre-parse checkpoint where one can initialize, check and abort the parse with appropriate returned error token before the parse takes place, proxy the token for the parse, to exercising plain software hygiene practices. Its importance is when a threaded grammar object is reused and appropriate duties are required before parsing activities begin.

- *user-imp-tbl* code is included in the grammar's tables code section. One uses it whenever there is some form of circularity taking place on thread headers: aka recursion — a thread calls a thread that calls a previous called thread.

- *user-imp-sym* code is included in the grammar's tables symbols section. Ditto on circularity in the sister directive above.

4.1 Where are those directives placed?

All directives can be inputted in any order but please hold the repeats. I'll hold the comments on *failed* directive as this is treated in "Errors" in chapter 9. Here is a quick review of *fsm*'s anatomy introduced on page 11. It's a fancy parameterized procedure. The directives pattern follows the *fsm-class* parameter enclosed by braces:

```
fsm
(fsm-id "err_symbols_ph_th.lex"
,fsm-filename err_symbols_ph_th
,fsm-namespace NS_err_symbols_ph_th
,fsm-class Cerr_symbols_ph_th
      { ∑ each directive }
,fsm-version "1.0",fsm-date "22 mar 2004",fsm-debug "false"
,fsm-comments "Parse Error vocabulary.")
```

The best way to feel comfortable is for me to give examples with some running comments as to what is being achieved per directive.

4.1.1 *user-prefix-declaration*

This example provides the basic pattern used by most of your grammars.

```
fsm
(fsm-id "err_symbols\_ph_th.lex"
,fsm-filename err_symbols_ph_th
,fsm-namespace NS_err_symbols_ph_th
,fsm-class Cerr_symbols_ph_th{
  user-prefix-declaration
using namespace NS_yacco2_terminals;
#include "lint_balls.h"
#include "identifier.h"
#include "term_def_ph.h"
#include "c_string.h"
  ***
  user-declaration
    public:
    T_error_symbols_phrase* error_symbols_phrase_;
  ***
  constructor
    error_symbols_phrase_ = 0;
  ***
  op
    if(error_symbols_phrase_ != 0){
      delete error_symbols_phrase_;
      error_symbols_phrase_ = 0;
    }
    error_symbols_phrase_ = new T_error_symbols_phrase;
    error_symbols_phrase_->set_rc(*parser__->start_token__
                                  ,__FILE__,__LINE__);
    AST* t = new AST(*error_symbols_phrase_);
    error_symbols_phrase_->phrase_tree(t);
  ***
}
,fsm-version "1.0",fsm-date "22 mar 2004",fsm-debug "false"
,fsm-comments "Parse Error vocabulary.")
```

Here the grammar will be calling other grammars and so it needs to include their header files. Generally

the axiom is: any definition that's defined outside the grammar and referenced gets placed here. These C++ statements are placed at the beginning of this grammar's header file. Here is its C++ extract for the given example.

```
/*
 File: err_symbols_ph_th.h
 Date and Time: Tue Nov  3 13:22:06 2009
*/
#ifndef __err_symbols_ph_th_h__
#define __err_symbols_ph_th_h__ 1
// grammar's vocabulary headers automatically added by O2
#include "yacco2.h"
#include "yacco2_T_enumeration.h"
#include "yacco2_k_symbols.h"
#include "yacco2_err_symbols.h"
#include "yacco2_terminals.h"
#include "yacco2_characters.h"

// user-prefix-declaration code
using namespace NS_yacco2_terminals;
#include "lint_balls.h"
#include "identifier.h"
#include "term_def_ph.h"
#include "c_string.h"
```

The preliminary code defines O_2's library: `"yacco2.h"` and your grammar's terminal definition files: enumeration scheme, and the 4 terminal classes. The emitted code for the *user-prefix-declaration* is one to one: it just copies what is inputted to this area following the grammar terminal introduction.

4.1.2 *user-declaration*

As the directive's name suggests, it declares variables and methods within the grammar's *fsm* definition. All the access privacy restrictions are allowed. U can use the complete repertoire of C++'s declaration capabilities. In this example, *error_symbols_phrase_* variable is declared public for efficiency reasons and gets initialized in either *constructor* or *op* directives. Though not exampled, its implementation counterpart is the *user-implementation* directive.

4.1.3 *user-implementation*

Not too creative Dave as the directive's name suggests. It usually implements the *fsm*'s methods. This example is drawn from O_2's parsing of terminals definition classes:

```
fsm
(fsm-id "term_def_ph.lex"
,fsm-filename term_def_ph
,fsm-namespace NS_term_def_ph
,fsm-class Cterm_def_ph{
  user-prefix-declaration
#include "lint_balls.h"
#include "identifier.h"
#include "c_string.h"
#include "t_def_delabort_tags.h"
#include "terminal_def_symclass.h"
#include "o2_sdc.h"
#include "yacco2_stbl.h"
#include "o2_externs.h"
#include "cweb_or_c_k.h"
  ***
```

```
  user-declaration
    public:
    void add_sdc_to_directive(CAbs_lr1_sym* Dir,T_syntax_code* Sdc);
    T_terminal_def* term_def_;
  ***
  user-implementation
    void Cterm_def_ph::
    add_sdc_to_directive(CAbs_lr1_sym* Dir,T_syntax_code* Sdc){
      using namespace NS_yacco2_T_enum;
      using namespace NS_yacco2_terminals;
      yacco2::INT eid = Dir->enumerated_id__;
      switch (eid){
      case T_Enum::T_T_user_declaration_: {
         T_user_declaration* k = (T_user_declaration*)Dir;
         k->syntax_code(Sdc);
         break;
        }
      ... // constructor and destructor left out to save space
      ... // same pattern as others: cast to specific T and save code
      case T_Enum::T_T_op_: {
        T_op* k = (T_op*)Dir;
        k->syntax_code(Sdc);
        break;
       }
      case T_Enum::T_T_user_implementation_: {
        T_user_implementation* k = (T_user_implementation*)Dir;
        k->syntax_code(Sdc);
        break;
       }
      default: {
        CAbs_lr1_sym* sym = new Err_improper_directive;
        sym->set_rc(*Dir,__FILE__,__LINE__);
        RSVP_FSM(sym);
        parser__->set_stop_parse(true);
       }
      }
    }
  ***
  op
    if(term_def_ != 0){
     delete term_def_;
     term_def_ = 0;
    }
    term_def_ = new T_terminal_def;
    term_def_->set_rc(*parser__->start_token__,__FILE__,__LINE__);
  ***
  constructor
    term_def_ = 0;
  ***
}
,fsm-version "1.0",fsm-date "24 mar 2004",fsm-debug "false"
,fsm-comments "Parse a terminal symbol.")
```

I included more directives to flavour your thoughts. *add_sdc_to_directive* is a method to store the directive's code. The implementation is the C++ code to implement it. No more no less. It demonstrates how the token's enumeration id is used to filter out improper directives within its context, register the code for code emission, how an error is created, GPS tagged, and one of the ways to return it to the calling thread[1].

[1]Damm u astute reader, what about a proper parse and what does it return to the calling grammar? I'll keep u in suspense

4.1.4 *constructor*

Preliminary code that is executed **one-time-only** when the grammar object is being born. In this case from the 1st example given it ensures the *error_symbols_phrase_* variable is initialized. U might question why it was not placed in the grammar initialization list. It was easier to include the code within the constructor's body. This opened up all the possibilities to your imagination rather than just variable initialization. Here's its C++ code:

```
Cerr_symbols_ph_th::
Cerr_symbols_ph_th()
  :yacco2::CAbs_fsm
    ("err_symbols_ph_th.lex"
    ,"1.0"
    ,"22 mar 2004"
    ,false
    ,"Parse Error vocabulary."
    ,"Tue Nov  3 13:22:06 2009 "
    ,S1_Cerr_symbols_ph_th){
  error_symbols_phrase_ = 0;
}
```

As *op* is always executed, unless u need the "1-time-only" condition, u can omit the *constructor* directive.

4.1.5 The other directive: *user-suffix-declaration*

Its utility is marginable but it allows u to tack on at the end of the grammar's definition this code. I cannot give u an example as i've not had to use it. This completes the where-to-place-code for your creative activities: before, into, and after *fsm*.

4.2 *fsm*'s header

Sounds like a football/soccer match. Not to be too technical, the grammar's header provides the following definitions:

- Grammar defined both as a thread entity and as a procedure: an optimization.
- Grammar's start state.
- Grammar's rules recycling table for reuse: another optimization. **new / delete** reincarnations kills the performance. See "Mumblings to myself" in O_2's code document regarding speed.
- Grammar's namespace and its class definition.
- Localized enumerated rules and subrules.
- Your defining directives.
- Fixed name methods for the generic *Parser*: *op*, *failed*, *reduce_rhs_of_rule*.
- Grammar's rule definitions. Contained within each rule's definition are its subrules aka productions.
- If used, the *user-suffix-declaration* code.

Here is a threaded grammar's header definition:

```
/* File: angled_string.h     Date and Time: Tue Nov  3 13:22:15 2009 */
#ifndef __angled_string_h__
#define __angled_string_h__ 1
// automatically gened headers by O2
```

but it's equivalent to what u've seen here.

```
 #include "yacco2.h"
 #include "yacco2_T_enumeration.h"
 #include "yacco2_k_symbols.h"
 #include "yacco2_err_symbols.h"
 #include "yacco2_terminals.h"
 #include "yacco2_characters.h"
// end of automatically gened headers by O2

// user-prefix-declarations
#include "esc_seq.h"
extern yacco2::Thread_entry ITH_angled_string; // Global thread version
extern yacco2::State S1_Cangled_string;        //

// Start of fsm definition
namespace NS_angled_string {
 yacco2::THR _YACCO2_CALL_TYPE
 TH_angled_string(yacco2::Parser* Caller);// grammar called as a thread
 yacco2::THR_result _YACCO2_CALL_TYPE           // thread's clone
  PROC_TH_angled_string(yacco2::Parser* Caller);// called as a procedure
 using namespace NS_yacco2_T_enum;// enumerate
 using namespace yacco2;

 struct fsm_rules_reuse_table_type{// optimized grammar rules recycling
  fsm_rules_reuse_table_type();
  int no_rules_entries_;
  Per_rule_s_reuse_table* per_rule_s_table_[2];
 };

 class Cangled_string: public yacco2::CAbs_fsm { // fsm definition
  public:
  enum rules_and_subrules{ // local enumeration scheme for rules
   start_of_rule_list = NS_yacco2_T_enum::T_Enum::sum_total_T
   ,R_Rangled_string_ = 569//start_of_rule_list + 0
    ,rhs1_Rangled_string_ = 1
   ,R_Ropen_angle_ = 570//start_of_rule_list + 1
    ,rhs1_Ropen_angle_ = 2
  };
  // no of la sets = 2, no of states = 4
  Cangled_string();
  ~Cangled_string();

  //* canned methods
   void op();
   bool failed();
   void reduce_rhs_of_rule
      (yacco2::UINT Sub_rule_no
      ,yacco2::Rule_s_reuse_entry** Recycled_rule);
   fsm_rules_reuse_table_type fsm_rules_reuse_table;
   static int rhs_to_rules_mapping_[3];

   // user-declaration directive
    public:
    char ddd_[1024*32];
    int ddd_idx_;
    void copy_str_into_buffer(std::string* Str);
    void copy_kstr_into_buffer(const char* Str);
 };

 // grammar's rules
```

```
 struct Rangled_string;
 struct Ropen_angle;
 struct Rangled_string:public yacco2::CAbs_lr1_sym {
  Rangled_string(yacco2::Parser* P);
  public:
  void sr1();
 };

 struct Ropen_angle:public yacco2::CAbs_lr1_sym {
  Ropen_angle(yacco2::Parser* P);
  public:
  void sr1();
 };
} // end of namespace
#endif
```

The C++ comments should be adequate to orient u through the reading of the definition. Before optimization set in, all my definitions were C++ classes. I left it as a class as the overhead per grammar was small compared to symbols like Terminals and Rules whose life cycles are frenzied.

4.2.1 Counting those rules

Lets sit back and reflect on rules. They are local to the grammar. They also need an enumeration scheme for the generated tables. Given that the tables deal with Terminals and Rules, each one's enumeration scheme cannot overlap. The simplest way to do this was just build after the Terminals enumeration scheme's endpoint. In the above example drawn from O_2's Terminals, `rules_and_subrules` variable defines the labels enumerated. The Terminals's enumeration header file provides the beginning point for the labels: `NS_yacco2_T_enum::T_Enum::sum_total_T` symbol to count from. A rule's label uses its name, prefixed with `R_` and ended with _. Each of its subrules labels has the same pattern whereby the prefix `R_` is replaced with `rhs%_` where `%` is the subrule's position in the rule relative to 1 [2]. Your own enumeration header file would have a different namespace attached to it.

Why are u explaining this when rules are not tested for their presence? From what u think is happening, all this is implicitly handled by the *Parser*: true. Though rare in having to probe the identity of an unknown item on the parse stack, u can still test stacked items that are out of bounds of the rule's right-hand-side of stacked symbols, or from a context like *fsm* that turns its cyclop's eye against the digester. This is covered in detail in the "Rules" chapter 5.

Now the dependency on symbol `sum_total_T` requires "neighbourhood watch" vigilance. This is why if u add or subtract terminal symbols from the Terminal vocabulary, it is very important to **regen all the grammars with the -t option** and to rerun O_2linker as all parse tables will be out-of-wack caused by the new number of terminals. U've been warned cuz if u don't, the parsing using its tables are now out-of-date. The result is a frustrated grammar writer that knows that the grammars written should be parsing properly but aren't: Tales from the bitten.

4.3 Nice but how to access *fsm*'s booty?

I'll jump ahead a touch in how this entity is accessed from various contexts. My comments are low-cal for the moment and should clue u into the potential open to u the grammar writer. The generic *Parser* receives the *fsm* object's address at constructor time. The *fsm* object also has a cross reference to the *Parser* object using it to parse with. An active rule receives its *Parser* object's address when called: U see it in its implementation:`(Parser* P)`. There are some hardwired methods that a rule structure inherits: `CAbs_lr1_sym`. One of them is `parser` method. For speed, one can directly access the rule's variables. As the *Parser* is generic, it knows nothing about the individual grammars that will be passed to it. This is why the defining grammar's *fsm* inherits from the base class *yacco2::CAbs_fsm*. So within the syntax directed code of the rule, u must cast the specific *fsm* type. Why did i not use C++'s runtime type information facilities?

[2] Why aren't each rule's enumeration value not using this symbol with an incrementing expression: and a 0 and a 1 etc? I felt both the explicit value and comment `// NS_yacco2_T_enum::T_Enum::sum_total_T + x` expression cross checked my code.

Speed. From there u can access its contents and methods. Here is a taste from grammar `subrule_def`'s start rule `Rsubrule_def`:

```
Rsubrule_def (){
 -> Rvector |.| Rcommon_comments Rrhs_expr {
   op
    Csubrule_def* fsm = (Csubrule_def*)rule_info__.parser__->fsm_tbl__;
    fsm->subrule_def_->bld_its_tree();
    RSVP(fsm->subrule_def_);
    fsm->subrule_def_ = 0;
   ***
   }
}
```

Well well well, seems the *op* is a directive for rules: Yep. And rules begin with `R` and there's a "lr constant" symbol `|.|` used in the production expression: again yep. So re-baptizing happens against the specific fsm whose name comes from its *fsm-class* parameter, followed by building its abstract tree. `RSVP` is a macro that handles the returned terminal back to the calling grammar's accept queue. And some cleanup done for the next round.

4.3.1 Glimpses of amnesia: `Parser` and rule tag team

Why the long way to get at `Parser*` with the exampled rule above when its implementation directly passes it as parameter `P`? I wanted to reenforce that the rule symbol inherits from the CAbs_lr1_sym. Well here's my amnesia as i didn't introduce it before discussing now. The truth is telling: i never used the `P` argument in O_2's grammars. I did though in other translators thereafter. The rule always gets its `Parser` freshly squeezed regardless whether its a first time appearance or a standing ovation whereas the Terminal symbols get theirs when firstly minted only. Thereafter, one needs to relie on *fsm* or the current rule as the terminal's past remembrance is presently whiffs of the past and out-of-date. It's your call.

4.4 Canned methods: CAbs_fsm

All the parameters of *fsm* are accessible in 2 forms: by method and by variable. It is supplied by `CAbs_fsm` that the concrete grammar inherits. No berating my take on the breaking of object-orient principles until my reasons are told: Speed and inline weaknesses were reasons enough. This is a closed system and i felt it warranted the violation. In my experience, there is little reason to access these arguments except for the *debug* argument. This will be detailed in debugging of a grammar. Here are the methods and their internal variable names.

- id(), id__: the grammar name in "c" literal.
- version(), version__: version in "c" literal.
- date(), date__: version in "c" literal.
- comments(), comments__: version in "c" literal.
- debug(), debug__: debug switch in boolean.

Other attributes available:

- gened_date, gened_date__: when compiled in "c" literal.
- start_state(), start_state__: the start state of the grammar.
- parser(), parser__: *Parser* address allowing the grammar writer to manipulate parsing contexts.

The `start_state__` variable is used by O_2 library in setting up of the parsing environment. At runtime the *Parser* sets the grammar object's `parser__` variable with its image thus giving the capability from within the *fsm* object access to its parsing environment. More to come later in the book on parsing.

4.5 Canned methods: take two in the grammar concrete

I'll leave out the "rules use" table as this is an internal implementation detail of O_2.

- `reduce_rhs_of_rule` handles the reduction of a grammar rule's when one of its subrules has been recognized. It collapses the parse stack with the subrule's symbols and replaces them with their parent rule. You the grammar writer have no need to deal with it apart from curiosity.
- directives that override `fsm`'s virtual methods: `op` and `failed`.
  - `op` a method that is always executed when the parse is started. It allows the grammar writer to hang special setup code.
  - `failed` a method that is a last chance to field a parse error and to deal with it **when the grammar aborted**.

The last 2 directive methods use C++'s *virtual* facility/inheritance for generic visibility and override capabilities.

4.5.1 `op` directive — the `fsm`'s first chance to do something.

Let's reflect on what it means "first chance to do something" before the parsing begins: for example initializing the `fsm` local variables, possibly notifying a global service, logging events. If the grammar is a thread, u can even play with the passed T before its parsing starts, return back to the calling grammar without parsing: a trampoline effect with consequences. U can consider this method as a pre-conditional checkpoint before the parsing begins and do whatever u want. Here's a sample taken from "enumerate_T_alphabet.lex". I've included more preliminary code than needed to show how i have programmed some of my grammars:

```
/*
FILE:   enumerate_T_alphabet.lex
Dates:   9 Aug. 2005
Purpose:   enumerate terminal symbols.
How: Each T phase contains its mapped symbols and create order list.
*/
/@
@** |enumerate_T_alphabet| grammar.\fbreak
Enumerate the Terminal symbols starting at
zero and going positively outward.
When the enumeration is finished,
the global |START_OF_RULES_ENUM| becomes
the starting enumerate of the grammar's rules.
This is the cut off boundary for shift / reduce LR1 compatibilty.
Why?
As a state is composed of shift vectors using their
enumerates, only terminals up to but not including
the start rule's enumerate are needed.
The exception to this is the ''eosubrule'' terminal that
represents the reducing subrule situation is not included in the
 shift set. So it is easy to iterate thru the state's vectors
in building up the shift set for the lr1 compatibility
check: as the state's vector map is sorted
by enumerate value, just stop when the current
vector's enumerate is part of the rule's domain.

In this grammar, each rule is a logic sequencer fetching
its phase's batch of symbols
for the baptism.
This grammar demonstrates zero token consumption.\fbreak
\fbreak
How:\fbreak
```

```
Each terminal phase contains its mapped
 symbols and create order list. Global Phase table:\fbreak
|O2_xxx| are the individual phases.\fbreak
Grammar Phases:\fbreak
\ptindent{0 - O2\_FSM\_PHASE : T\_fsm\_phrase}
\ptindent{1 - O2\_PP\_PHASE : T\_parallel\_parser\_phrase}
\ptindent{2 - O2\_T\_ENUM\_PHASE : T\_enum\_phrase}
\ptindent{3 - O2\_ERROR\_PHASE : T\_error\_symbols\_phrase}
\ptindent{4 - O2\_RC\_PHASE : T\_rc\_phrase}
\ptindent{5 - O2\_LRK\_PHASE : T\_lr1\_k\_phrase}
\ptindent{6 - O2\_T\_PHASE : T\_terminals\_phrase}
\ptindent{7 - O2\_RULES\_PHASE : T\_rules\_phrase}
Within each rule is the symbol iteration.\fbreak
The enumeration starts with the |lr k|
symbols followed by
raw characters, Terminals, and Errors.
Why is zero the start point?
Glad u asked as i use modulo 32 on the terminal enumerate
to arrive at its set number and bit position  within the set.
See set discussion in \O2's library documentation as to the
reasons.\fbreak
Caveat:\fbreak
As i used a keyword descent trigger parse phases, i must
check here if all my phrases are present
before the enumeration can take place.
Why here? This grammar frontends the triggered rule
phase parse due to its related grammars using symbol
 enumeration as edit checks. So dot the Ts and Is
before the ruling!
@/
fsm
(fsm-id "enumerate_T_alphabet.lex"
,fsm-filename enumerate_T_alphabet
,fsm-namespace NS_enumerate_T_alphabet
,fsm-class Cenumerate_T_alphabet{
  user-prefix-declaration
#include "o2_externs.h"
  ***
  user-declaration
    public:
     NS_yacco2_terminals::T_enum_phrase* enum_phrase_;
  ***
  op
    START_OF_RULES_ENUM = 0;
    CAbs_lr1_sym* gps = O2_FSM_PHASE;
    CAbs_lr1_sym* esym(0);
    CAbs_lr1_sym* ph = O2_T_ENUM_PHASE;

    if(ph == 0){
      esym = new ERR_no_T_enum_phrase;
      goto error_fnd;
    }
    gps = ph;
    ph = O2_ERROR_PHASE;
    if(ph == 0){
      esym = new ERR_no_errors_phrase;
      goto error_fnd;
    }
    gps = ph;
```

```
    ph = O2_RC_PHASE;
    if(ph == 0){
      esym = new ERR_no_rc_phrase;
      goto error_fnd;
    }
    gps = ph;
    ph = O2_LRK_PHASE;
    if(ph == 0){
      esym = new ERR_no_lrk_phrase;
      goto error_fnd;
    }
    gps = ph;
    ph = O2_T_PHASE;
    if(ph == 0){
      esym = new ERR_no_terminals_phrase;
      goto error_fnd;
    }
    all_phases_ok:
      enum_phrase_ = O2_T_ENUM_PHASE;
      return;
    error_fnd:
      parser__->add_token_to_error_queue(*esym);
      if(gps != 0) // anchor error against previously good phase
       esym->set_rc(*gps,__FILE__,__LINE__);
      parser__->set_abort_parse(true);
      return;
  ***
  constructor
    START_OF_RULES_ENUM = 0;
    enum_phrase_ = 0;
  ***
}
,fsm-version "1.0",fsm-date "9 Aug. 2005",fsm-debug "false"
,fsm-comments "Enumerate T symbols: the oracle for parsing lookups.")
```

The above example shows how it is checking for double dipping calls and how it can abort the parse. It also describes that i use recursive descent parsing by threaded grammars within a bottom-up parsing context.

4.5.2 `failed` directive — the last gasp

The best way to describe it is by a concrete example taken from O_2's frontend command line option grammar "o2_lcl_opt.lex". Remember it only gets called when the grammar has **aborted** and **not stopped**. Here is its code:

```
fsm
(fsm-id "o2_lcl_opt.lex",fsm-filename o2_lcl_opt
,fsm-namespace NS_o2_lcl_opt,fsm-class Co2_lcl_opt{
/@
Trap the failed option and return a bad command.
This covers errors like the premature prefix -e where it should
be -err. i could have been less specific to trap
non first set options (-z) by defaulting  to this
facility but i'm teaching myself...
As this thread is executed according to its first set ``-'',
any failed attempt is a bad option.
Please note the use of the |RSVP_FSM| macro.
Its context is different than the normal Rule's
use of |RSVP| macro.
@/
```

```
  failed
    CAbs_lr1_sym* sym = new Err_bad_cmd_lne_opt;
    sym->set_rc(*parser__->start_token__,__FILE__,__LINE__);
    RSVP_FSM(sym);
    return true;
  ***
}
,fsm-version "1.1",fsm-date "18 Oct. 2003",fsm-debug "false"
,fsm-comments "\\O2's individual command line option recognizer.")
parallel-parser
(
  parallel-thread-function
    TH_o2_lcl_opt
  ***
  parallel-la-boundary
    eolr
  ***
)
@"/yacco2/compiler/grammars/yacco2_T_includes.T"

rules{
Ro2_lcl_opt (
){
  -> Rminus Rspec_parm Rmust_lint{
  op
    sf->p2__->rtn_parm_->set_rc(*rule_info__.parser__->start_token__
                                ,__FILE__,__LINE__);
    RSVP(sf->p2__->rtn_parm_);
  ***
  }
}

Rminus (){
  -> "-"
}

Rspec_parm (
lhs {
  user-declaration
    public:
    CAbs_lr1_sym* rtn_parm_;
  ***
  }
){
  ->
  /@
Generate Terminals vocabulary.
  @/
"t"
  {
    op
      rtn_parm_ = new T_option_t;
    ***
    }
  ->
  /@
Generate Terminals vocabulary.
  @/
"p"
```

```
  {
    op
      rtn_parm_ = new T_option_p;
    ***
    }
  ->
/@
Generate Raw characters vocabulary.
Watch out as this is only used at bootstrap time of \O2.
Users of \O2 should not use it unless they are experimenting.
@/
"r" "c"
{
    op
      rtn_parm_ = new Err_bad_cmd_lne_opt;
      rtn_parm_->set_rc(*rule_info__.parser__->start_token__
                        ,__FILE__,__LINE__);
    ***
    }
  ->
/@
Generate Error vocabulary.
@/
"e" "r" "r"
{
    op
      rtn_parm_ = new T_option_err;
    ***
    }
  ->
/@
Generate LR K vocabulary.
Watch out as this is only used at bootstrap time of \O2.
Users of \O2 should not use it unless they are experimenting.
Currently not supported.
@/
"l" "r" "k" {
    op
      rtn_parm_ = new Err_bad_cmd_lne_opt;
      rtn_parm_->set_rc(*rule_info__.parser__->start_token__
                        ,__FILE__,__LINE__);
    ***
    }
  -> |?|  //catch bad attempts at options
  {
    op
      rtn_parm_ = new Err_bad_cmd_lne_opt;
      rtn_parm_->set_rc(*rule_info__.parser__->start_token__
                        ,__FILE__,__LINE__);
    ***
  }
}
Rmust_lint (){
  -> "x09" // tab
  -> " "
  -> |+|  //catch suffixed extras on option
  {
    op
      parser()->set_abort_parse(true);
```

```
    ***
  }
}
```

The Cweb comments above tells it all. The parse can abort from a faulty non prefixed option like `-x`, a partially prefixed parameter like `-et`, or a superfulous suffix like `-errr`. The faulty input can be explicitly caught within the grammar by use of the special k terminals: `|?|` or `|+|` symbols described later in the book. The `failed` facility allows the grammar writer to catch the unsuccessful parse when the `Parser` is winding down doing cleanup duties. For now `failed` is only supported in threaded grammars as it allows the grammar writer a way to return a T back to the calling grammar and to indicate that the parse was successful. Here u the grammar writer have the freedom to "last chance" field the aborted parse and work within this context to do something.

4.6 Closing remarks

I hope this whets your appetite to continue: Interweaves of future images for the coming.

Chapter 5

Rules give me liberties?

Well here's the heart of the grammar. A rule whose pre-established sequences of symbols are its etudes from the T and rules vocabulary. Let's re-look at a rule's individual architecture. In chapter 2, section "Rules and Productions" 2.4.4 outlined its basic structure and sequencing. It left out the directives open to hang your logic points per rule, and variations on those special "k" terminals. Lets overview a rule's definition.

rule name [AB AD] ([directives] [arbitration])
{
$\sum$ each subrule definition
}

A rule's definition starts with its name that begins with a letter: upper or lower case possibly followed by any number of underscore: "_", upper or lower case letters, or numbers. Well my golden years are waning so i relaxed the big "R" requirement. "AB" and "AD" are delete attributes for the Parser to act upon; they are optional indicated by the surrounding "[" and "]" brackets. They are the same as the delete attributes used in Terminal definitions. When the grammar aborted and the parse stack is being cleansed, `Parser` will recycle the "on parse stack" rule back to the memory heap when the "AB" attribute is on. During a reduce operation when it pops the items from the parse stack, each item is checked whether its "AD" attribute is "on" and should be deleted[1].

Due to a rule optimization, these **attributes are presently dormant**. U can provide them but they are not acted upon. Rules are brought to life by the `Parser` with C++'s `new` verb. There's that optimization theme again, but using a recycled approach gave approximately a 25% speed improvement. Rules per grammar have their own recycle table. Remember the delete attributes of Tes are always acted upon.

A rule's directives are optional indicated by the enclosing square brackets: "[" and "]". Its directives are exactly the same as described in Chapter 3 Section 3.2 but applied to the rule within its own context. Why are directives optional and what purpose do they serve? 2 questions are asked. The first question opens the answer that its subrules provide symbol sequencing without any interaction with its parent rule. The syntax directed code of the subrule could be interacting only with the grammar's `fsm` construct. `fsm` is the global root container across all the subrules and rules. The 2nd question allows one to tailor variable definitions and methods for the specific rule. That is, there is interaction between its subrules and self. From rule recursion (left or right), the right-hand-side's rule within the subrule could copy its contents into its left-hand-side rule when the subrule is being reduced to. I know u are questioning this but think about the dynamic building of trees from the bottom-up where each subrule's symbols on the parse stack might contribute to the reduced parent rule's node. This will be explained later with examples.

What is this optional arbitration: "[" arbitration "]"? Here comes the judge. When 2 or more threaded grammars are competing and both return a T, how does one decide which T to use? Arbitration. In the calling grammar's current parse state, it launched the threads and has associated arbitration code of the grammar writer in place[2]. Arbitration is state sensitive giving complete control by the grammar writer

[1] What are these acronyms AD and AB? A is attribute where AB builds on it: abort. D is delete. Not too creative Dave but your query dear reader is answered.

[2] There are optimizations; O_2 is open to no customized code and bypasses the arbitration process when there is only 1 returned T from the called threads.

to select the winning T. What does this mean? U can have multiple places throughout your grammar where different arbitration code takes place. What happens to the other rejected Tes? They are destroyed. Arbitration is done before the Parser has to select the subrule to reduce using the winning T. I'm a bit head of myself but u'll appreciate its detailing later in this chapter and the power to deal with nondeterminism in a deterministic way.

5.1 Stop the fluff and let the subrules begin

So how do u express a subrule?

$\sum$ each subrule definition

The above certainly doesn't cut it. Chapter 2 in "Rules and Productions" explored some rules with their subrules. A recap: an individual subrule has the following syntax:

`->` $\sum_0^n$ symbols drawn from T or Rules vocabulary [op directive]

The `->` introduces the line of symbols that are drawn from the grammar's vocabulary: Terminals and Rules. I used the summation symbol showing that the right-hand-side of symbols could have a lower bound of 0 symbols, or an upper bound of n finite number of Tes and grammar rules. This is the customary way to express a subrule before threading was added.

5.1.1 Just show me how to call a thread a thread

Threads can be thought of as an equivalent to procedure calls in recursive descent parsing. A subrule thread expression uses the same symbol sequencing with this restriction — it contains just 3 symbols of pre-determined order:

1) A thread call operator: `|||` or `|t|`[3]
2) The returned T from the called thread
3) The thread name to call[4]

You cannot mix any other symbols within this expression or it will be declared illegal. The potential `op` directive can follow this thread expression. I'll give a real example of subrules with thread expressions within a rule:

```
-> ||| "ws" NS_ws::TH_ws
-> ||| "comment" NS_c_comments::TH_c_comments
-> ||| "eol" NS_eol::TH_eol
-> ||| |?| NULL
```

The `->` begins each subrule followed by the parallel thread operator: `|||`. Following the parallel operator is the returned T from the called thread. In the first subrule the returned T is `"ws"`. The called thread has to be fully qualified in C++ terms: `NS_ws::TH_ws` where `NS_ws` is the name space of the grammar amd `TH_ws` the name of the thread[5]. Now what is this thread named `NULL`? This is a keyword that indicates that no thread is called but that one of the other threads could be returning its associated T: `|?|`. `|?|` is the questionable T that is used to catch errors. Use of the "lr k" class terminals like `|+|` will be explained later. The thread call expression is simple but very powerful. It can be mixed with other competing non-thread subrule expressions. There is a pecking order of how things are called and accepted:

1. thread calls. If they are present in the parse state and successfully completed. Successfully completed means that the thread has returned a T regardless in class type: Error or meta T. No returned T is an implicit statement that the thread was unsuccessful allowing the following peaking order to continue.

2. shift. It has its own pecking order whereby the first successful shift stops any additional parsing attempts. The order starts at the specific T and works its way to the general. Can the current parse token be shifted first? The error condition is tested for by the `|?|` presence in the parse state. Explicit

[3]This thread call variant will be addressed later.
[4]The thread name is not officially part of the T vocabulary: call it a quasi meta terminal.
[5]How it is declared in a grammar will be addressed later in the book along with the thread's first set, and lookahead set.

epsilon condition is then checked for by the `|.|` presence. At last the wild token symbol is checked: `|+|`.

3. reduce. It is tried only if none of the above shift types where successful and a possible reduction is present within the current parse state[6]. Note shifting is favoured over reducing.

With these pecking orders, the lr(1) `Parser` has been extended in a backtracking way except that it toe taps in the same parse state. Just remember that shifting is tried first in its 2 variants: thread calls and Tes. If present in the current parse state, threading is attempted first. The attempted call depends on whether the current parse T is in the threads first sets(if there are competing threads). `Parser` considers it successful when there is at least a returned T and will not try any other shift type. The next attempt uses the current T to parse the regular bottom-up way. The special Tes: `|+|`, `|.|`, and `|?|` add another extension to the parsing context that is checked after the current T cannot be shifted. The last parse operation attempted is the reduction using the current T against the state's potential follow sets. This raises a question: What happens if none of these operations can be performed? `Parser` will quietly complain with an appropriate message and shut down its activities. The message is directed to the grammar writer with enough information to fix the faulty grammar.

Here are 2 mixed examples of called threads with shifts. The first example drawn from "prefile_include.lex" shows the potential "white space" consumption by a thread. If it is not successful meaning there is no white space present then the explicit epsilon condition continues the parse.

```
Rpossible_ws  (){
  -> |.|    // explicit epsilon: shift out conflict
  -> ||| "ws" NS_ws::TH_ws
}
```

The 2nd example shows error catching and `op` use from O_2's "T_enum_phrase.lex" grammar:

```
Rphrase (){
  ->  ||| "T-enum-phrase" NS_T_enum_phrase_th::TH_T_enum_phrase_th {
  op
    ADD_TOKEN_TO_PRODUCER_QUEUE(*sf->p2__);
  ***
  }
  ->  ||| |?| NULL { // catch thread returned error
  op
    ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__);
  ***
  }
  ->  |?| { // error due to no first set to start thread
  op
    CAbs_lr1_sym* sym = new Err_no_open_parenthesis;
    sym->set_rc(*rule_info__.parser__->current_token()
                ,__FILE__,__LINE__);
    ADD_TOKEN_TO_ERROR_QUEUE(*sym);
  ***
  }
}
```

It demonstrates how errors are caught by the `|?|` symbol from "returned Tes" by the calling subrule in subrule 2. And the 3rd subrule shows how error pinpointing within a grammar is detected and reported on due to the faulty current T.

5.1.2 Only the lonely and the subrule directive `op`

There is only 1 directive that a subrule can have: `op`. Here is the rule from O_2's options grammar using the "called thread" expression with their associated directives:

[6]The ϵ subrule is a reduce.

```
Rspec_parm  (){
  -> ||| "option-t" NS_o2_lcl_opt::TH_o2_lcl_opt {
    op
      Co2_lcl_opts* fsm = (Co2_lcl_opts*)rule_info__.parser__->fsm_tbl__;
      ++fsm->parm_cnt_;
      fsm->t_sw_ = 'y';
      sf->p2__->set_auto_delete(true);
    ***
    }
  -> ||| "option-err" NULL {
    op
      Co2_lcl_opts* fsm = (Co2_lcl_opts*)rule_info__.parser__->fsm_tbl__;
      ++fsm->parm_cnt_;
      fsm->err_sw_ = 'y';
      sf->p2__->set_auto_delete(true);
    ***
    }
  -> ||| "option-p" NULL {
    op
      Co2_lcl_opts* fsm = (Co2_lcl_opts*)rule_info__.parser__->fsm_tbl__;
      ++fsm->parm_cnt_;
      fsm->prt_sw_ = 'y';
      sf->p2__->set_auto_delete(true);
    ***
    }
  -> ||| |?| NULL {
  /@
A false option so report it.
  @/
    op
      rule_info__.parser__->set_use_all_shift_off();
      ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
}
```

Some subtleties explained:
Only one thread is called: `NS_o2_lcl_opt::TH_o2_lcl_opt` that explicitly receives the `"option-t"` T. `"option-err"`, `"option-p"` are also returned from this thread along with errors caught by the `|?|` symbol. There is no attached importance to the associated returned T in that it is an explicit statement dealing with this returned T within its `op` directive. Any of the other returned Tes could have been associated with this first subrule and dealt with accordingly. For the valid Tes, each `op` code sets a parameter in the grammar's `fsm` construct. The catcher-in-the-? puts the error T into the error queue, and indicates to the Parser to stop parsing. It also has some comments for *Cweb* to deal with.

5.1.3 Parse stack casting of characters?

Well i'm a bit mired in my information vortex just created so i might as well spill the truth. Some of the above examples referenced a variable like `sf->p2__`. What and where did this `sf` variable come from. `sf` is short for stack frame. It is a structure that spans across the parse stack containing the subrule's symbols. It gets custom made per state and emitted by O_2. Each parse state will contain its own specific stack frame structure if there is associated syntax directed code referring to the stacked items.

Each symbol and its relative position within the subrules line of symbols is cast. A simple axiom holds: `px__` where p indicates parameter and x is its symbol position within the symbol string relative to 1. In a "called thread" expression, its returned T is the 2nd symbol position as the 1st symbol is `|||`. Here is sample C++ code for the `Rphrase` subrule 1.

```
void Rphrase::sr1(){
```

```
  struct SF{
   LR1_parallel_operator* p1__;
   State* s1__;
   bool abort1__;
   Rule_s_reuse_entry* rule_s_reuse_entry1__;
   T_enum_phrase* p2__;
   State* s2__;
   bool abort2__;
   Rule_s_reuse_entry* rule_s_reuse_entry2__;
  };
  SF* sf = (SF*)rule_info__.parser__->parse_stack__.sf_by_top(2);
  ADD_TOKEN_TO_PRODUCER_QUEUE(*sf->p2__);
}
```

`p1__` and `p2__` are cast to their related symbols as expressed in the grammar's subrule. Later the stack frame and its own subtleties and access routines will be explained. As for now i'll tease u: what do u think the 2nd subrule maps to for the `|?|` symbol? And what about the amorphous `|+|` symbol? And what is its contents? [7]

5.2 Reworking the rule's directives

All this time was spent talking about the right-hand-side of a rule: aka subrule or productions whatever is your verbiage comfort zone. Now a rule can itself be costumed and we will refer to it as the left-hand-side: lhs of a production re rule. If we look at the activity of a bottom-up parser: Tes are consumed until it hits some boundary. At this point, part of stacked symbols will get associated with its lhs rule. The rule's subrule that matched the stacked symbols gets taken off the parse stack and the rule's symbol replaces it on the parse stack. This keeps happening between Tes and rules until the goal symbol: "Start rule" gets stacked.

Why the review? Well rules are consolidators of subrules: excuse the pun but they are reducers. Their life expectancy is short. That is some subrule containing this symbol will eventually be reduced to another rule whose existence is contained in some other subrule. Only the "Start rule" lives forever. All this round-about-chatter on rules who are really divisional action points.

So if rules are action points why should they have their own methods and variable definitions? This question asks if the contents of its recognized subrule is not used then should not the rule itself be barren of attributes. Sometimes yes and sometimes no: jees the straddle of indecision. There could still be actions outside of the parser that the rule does. So rules are digesters of their subrule symbols in reversed hierarchies. Some examples are needed: though the keyword character sequence is recognized, the individual characters are not needed; just create the specific keyword T. Building of abstract trees from leaves to parent nodes becomes a good reason for their own costume jewellery: attributes as they reference their appropriate subrule contents. So where and how does a rule hang its definitions? A rule's definition was described to be like a procedure call with parameters whose body were the individual list of subrules. The 2 parameters were optional and enclosed in "(" and ")": directives, and arbitration.

rule name [AB AD] ([directives] [arbitration])

I designed the parameters to be keyword identified, and i also thought their order was important. This last point was to give consistency to a grammar's writing style.

5.2.1 The rule's parameters: take 1 lhs and its dwarfs

A rule's directives are enclosed in the `lhs { directives }` construct. `lhs` introduces the block of directives. I know u'll question my stutters but here are the rule's directives that are the same as Terminals. It refills my short term memory.

- *user-declaration*: definitions of variables or methods.

[7] What are those trumpet toots? Ahh the running of the bulls and my red flags.

- *user-implementation*: implementation details of the declaration.
- *destructor*: code for a simulated C++ dtor method. Currently not acted upon due to u got it: evil optimization?
- *constructor*: code that gets injected into the C++ struct's ctor method. It's a **one-time-only** occurrence. It is a C++'s ctor.
- *op*: a method that is executed each time a rule definition is used. This occurs when it is first created and whenever it is reused. I repeat rules are recycled and so when they are brought back into service, `op` will get executed before the `Parser` executes a subrule directive. It's a good house keeping method.

I will give some examples below to flavour your imagination. Here is a partial building of a tree. Though tree building is for future chapters, the rule's directive and associated subrules activities gives u the reader a feeling for how one accesses the subrule symbols. `AST` is the tree node type. Each rule `Rse` and `Roperators` have the same variable definition as `Rexp` defined by the `user-declaration` directive. Notice there are 2 occurrences of `Rse` in some of the below subrules. This demonstrates multiple rule use at the same time. Left recursive subrules where its rule starts a subrule expression is another example of this.

```
Rexp  (
lhs {
  user-declaration
    public:
    AST* type_;
  ***
  }
){
  ->  Rse Roperators Rse {
    op
      AST* p1 = sf->p1__->type_; // 1st parm: rule Rse
      AST* p2 = sf->p2__->type_; // 2nd parm: rule Roperators
      AST* p3 = sf->p3__->type_; // 3rd parm: rule Rse
      sf->p1__->type_ = 0;       // cleanup leave no remnants
      sf->p2__->type_ = 0;
      sf->p3__->type_ = 0;
      type_ = p1;
      AST::join_sts(*p1,*p2);    // horizontal joining of leaves
      AST::join_sts(*p2,*p3);
    ***
    }
  -> Rse |.| {
    op
      AST* p1 = sf->p1__->type_;
      sf->p1__->type_ = 0;
      type_ = p1;
    ***
    }
  ->  Rse Roperators Rse Rformat_phrase {
    op
      AST* p1 = sf->p1__->type_;
      AST* p2 = sf->p2__->type_;
      AST* p3 = sf->p3__->type_;
      AST* p4 = sf->p4__->type_;
      sf->p1__->type_ = 0;
      sf->p2__->type_ = 0;
      sf->p3__->type_ = 0;
      sf->p4__->type_ = 0;
      AST* eost = // see if adding to end of p3 is required
         AST::get_youngest_sibling(*p3);
      if(eost == 0) eost = p3; // p3 has no younger siblings
```

```
      type_ = p2;
      AST::join_sts(*eost,*p4); // brothers joining hands
      AST::crt_tree_of_2sons(*p2,*p1,*p3); // voila a tree is born
    ***
    }
  -> Rse Rformat_phrase |.| {
    op
      AST* p1 = sf->p1__->type_;
      AST* p2 = sf->p2__->type_;
      sf->p1__->type_ = 0;
      sf->p2__->type_ = 0;
      AST* eost = AST::get_youngest_sibling(*p1);
      if(eost == 0) eost = p1;
      AST::join_sts(*eost,*p2);
      type_ = p1;
    ***
    }
}
```

Promises to come:
Start thinking on the boundary points for parse actions per symbol type. Their interactive contexts will be discussed later in the book. Lay back and do your own musings against these entities: rules: lhs, subrules: rhs, fsm: grammar to grammar communications, syntax directed interactions with the `Parser`. Think pre, within, and post interactions within a hierarchical context.

This next example shows how to consolidate "syntax directed code" using `op` within the rule's definition to extract the subrule's symbol from the parse stack. It saves doing this per subrule and shows how it uses the grammar's `fsm` to build up the hexadecimal string. The complete grammar uses left recursion to loop through the hexadecimal character sequences and concatenates their contents within the grammar's `fsm`.

```
Rhex_no_letter  (
lhs{
   op
     Cesc_seq* fsm = (Cesc_seq*) rule_info__.parser__->fsm_tbl__;
     size_t pos = rule_info__.parser__->parse_stack__.top_sub__ - 1;
     CAbs_lr1_sym* sym = rule_info__.parser__->get_spec_stack_token(pos);
     fsm->hex_data_ += sym->id__;// bld up hex char seq by concatenation
   ***
}
) {
  -> a -> b -> c -> d -> e -> f
  -> A -> B -> C -> D -> E -> F
}
```

The last example shows how to define a local method within a rule and how it is used. It gives u a peek at how error handling is done, how u can control the stopping of a parse, and return a good or error T back to the calling grammar. This is a threaded grammar extract from "o2_linker_opts.lex".

```
Rfile  (
lhs{
  user-declaration
    public:
      void edit_file(CAbs_lr1_sym* Sym){
        using namespace NS_yacco2_err_symbols;
        using namespace NS_yacco2_k_symbols;
        Co2_linker_opts* fsm =
           (Co2_linker_opts*)rule_info__.parser__->fsm_tbl__;
        std::ifstream ifile;
        ifile.open(fsm->file_to_compile_.c_str());
        if (ifile.good()){
```

```
          ifile.close();
          return;
        }
        CAbs_lr1_sym* sym = new Err_bad_filename(fsm->file_to_compile_);
        sym->set_rc(*Sym,__FILE__,__LINE__);
        ADD_TOKEN_TO_ERROR_QUEUE(*sym);
        rule_info__.parser__->set_stop_parse(true);
      };
  ***
}
){
  -> ||| "unquoted-string" NS_unq_str::TH_unq_str {
    op
      T_unquoted_string* uqstr = sf->p2__;
      Co2_linker_opts* fsm =
         (Co2_linker_opts*)rule_info__.parser__->fsm_tbl__;
      fsm->file_to_compile_ += uqstr->unquoted_string()->c_str();
      edit_file(uqstr);
      sf->p2__->set_auto_delete(true);
    ***
    }
  -> ||| |?| NULL {
    op
      rule_info__.parser__->set_use_all_shift_off();
      ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__);
    ***
    }
  -> ||| "xc-str" NS_xc_str::TH_xc_str {
    op
      T_xc_str* xcstr = sf->p2__;
      Co2_linker_opts* fsm =
         (Co2_linker_opts*)rule_info__.parser__->fsm_tbl__;
      fsm->file_to_compile_ += xcstr->c_string()->c_str();
      edit_file(xcstr);
    ***
    }
  -> |?| { // no file given
    op
      rule_info__.parser__->set_use_all_shift_off();
      CAbs_lr1_sym* sym = new Err_no_filename();
      sym->set_rc(*rule_info__.parser__->current_token()
                 ,__FILE__,__LINE__);
      ADD_TOKEN_TO_ERROR_QUEUE(*sym);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
}
```

The `edit_file` method is declared and defined within the `user-declaration` directive. Why the combo declaration and definition inside the declaration directive? I was lazy. I could have only declared it using this directive and defined it using the `user-implementation` directive. Just remember an implementation requires full C++ qualification using the rule's name: `Rfile::`. The implementation would have been written:

```
  user-implementation
    void Rfile::edit_file(CAbs_lr1_sym* Sym)
         {
            code ...
         }
  ***
```

5.2.2 The rule's parameters: arbitration's many takes

Before i layout what the arbitration parameter looks like, let's figure out what it is all about. Does it make sense to arbitrate within a monolithic grammar? How about a threaded grammar? This is a pink herring! Arbitration must occur when 2 or more threads are called from a rule and 2 or more Tes can be returned. Let's rephrase this, it does not matter what T type is returned as long as more than 1 T can be returned at the same time. So if there is no threads being called from a rule, arbitration is not needed. Even if there are multiple thread calls from a rule, if there is only 1 T returned, there is no need to arbitrate.

The arbitration parameter can be introduced by itself as the `lhs` keyword is optional indicated by the square brackets [8].

```
[ lhs [{ directives }] ,] // potential directives code
[parallel-control-monitor { arbitration code }]
```

Remember tandomed **lhs** and **parallel-control-monitor** keywords are separated by a ",". Like a "c" procedure call the `parallel-control-monitor` keyword introduces it code block that is contained within the `arbitrator-code` directive:

```
arbitrator-code code ...  ***
```

Here is an example.

```
Rsym_def1  (
lhs, // lhs, presence shows original design requirement. Now out damn spot!
parallel-control-monitor
{
/@
    Note how the arbitration is done: list check.
    In this grammar, at most 2 terminals can be returned.
    When 2 terminals are returned, one is in error and the other is
    a result from the terminals-sufx. This is either the
    syntax-code or an error: eg no code present.
@/
  arbitrator-code // directive
    using namespace NS_yacco2_T_enum;
    // i, and ie are wired into the emitted code along
    // with arbitrated_parameter label
    for(i=1;i<=ie;++i){
      if(Caller_pp->pp_accept_queue__[i].accept_token__->enumerated_id__
          != NS_yacco2_T_enum::T_Enum::T_Err_no_terminal_key_present_){
        goto arbitrated_parameter;// hardwired label to break out of loop
      }
    }
  *** // ending the directive
}
){
  ->  ||| "terminal-def" NS_term_def_ph::TH_term_def_ph {
    op
      Cterminals_phrase_th* fsm =
        (Cterminals_phrase_th*)rule_info__.parser__->fsm_tbl__;
      sf->p2__->classification(T_terminal_def::t);
      CAbs_lr1_sym* r = fsm->terminals_phrase_->
            add_t_to_alphabet(sf->p2__,rule_info__.parser__);
        RSVP(r);
        rule_info__.parser__->set_stop_parse(true);
     ***
    }
  ->  ||| "no key-value present in definition" NULL {
```

[8] Originally **lhs** presence was mandidtory when arbitration was to take place. This became a pain-in-the-ass requirement for me and was down graded to optional.

```
      op
        Cterminals_phrase_th* fsm =
          (Cterminals_phrase_th*)rule_info__.parser__->fsm_tbl__;
        RSVP(fsm->terminals_phrase_);
        fsm->terminals_phrase_ = 0;
        rule_info__.parser__->set_stop_parse(true);
      ***
      }
  ->  ||| |?| NULL {
      op
       using namespace NS_yacco2_T_enum;
       int id = sf->p2__->enumerated_id__;
       if(id >= T_Enum::start_ERR && id <= T_Enum::end_ERR){
         RSVP(sf->p2__);
         rule_info__.parser__->set_stop_parse(true);
       }else{
        CAbs_lr1_sym* sym = new Err_not_a_terminal_definition;
        sym->set_rc(*sf->p2__,__FILE__,__LINE__);
        RSVP(sym);
        rule_info__.parser__->set_stop_parse(true);
       }
      ***
      }
}
```

The *Cweb* comments and subrules's code directives are left in to allow u the reader to become accustomed to their interplays. Their comments are pertinent to this example only. U can have as many potential returned Tes as called threads. Well the arbitration block sure is ugly. But it is sufficient. To summarize, a grammar can have many arbitrations spread through its rules. If there are more than 1 result returned, nondeterminism is a happening and so u must go through the courts to resolve. Chapter 7 on "Threads" subsection 7.4 expands on how the arbitration code is assigned to the parsing state when multiple threading subrules are taking place from different rules.

I leave u to stew on this thought:
Subrules make up a rule. What happens when the rule has subrules that start with other rules that have nondeterminism occuring? ie there can be multiple arbitrations occurring due to each rule against that specific parser state. How do u resolve this? O_2 currently does not try to resolve the code generation. This is a weakness to be explored in its next release. Remember arbitration code is only required when more than 1 T is returned into the calling thread's accept queue at the same time and a choosing is needed. Not arbitrating on multiple outcomes will throw an error by O_2. To correct this weakness try to contain all the arbitration code within the 1 rule instead of distributing it throughout the many rules.

5.2.3 Sequestered arbitrator-code

To comment i must introduce how a grammar receives it returned Tes. There is an "accept queue" of terminals per parser that has a grammar assigned to it. Regardless whether the grammar is monolithic or a thread, there is an assigned `Parser` to each one. Grammars can call other grammars in a nested fashion without any care about the other activities around them. Their own `Parser` cocoons them from the others. The only important thing is a called grammar must know its caller so that its findings can be reported. Rephrased, multiple nondeterminism can take place simultaneously where eventually the calling `Parser` will get reactivated. Recursive thread calls are allowed as each activated thread has its own activity and its rules are local to itself even when multiple occurrences of a cloned self is happening. Each called thread has access to its caller's accept queue.

An "accept queue" is protected against multiple parse writers that could plop their booty into it at the same time. So when all the called threads are finished executing, the calling parser/grammar gets wakened. Before it tries to continue its parsing activity: i.e. determine which called thread's subrule should be continued along, arbitration must take place on the returned contents. If only one thread was called, or

one T was returned the threat of arbitration is extinguished. Now u have the notion that the "accept queue" of returned Tes must be traversed. It is an array of unknown Tes where it starts counting from 1. The order of how the Tes are placed is random. This depends on the number of cored cpus simultaneously running and when the called thread got clearance to deposit its contents into the accept queue. Now the grammar writer must rule over these nondeterministic results.

To enlighten u on what an arbitrator piece of code looks like, here is its emitted C++brethren for the above rule. This is where u see how the returned Tes are identified by use of a symbol's enumerate id.

```
yacco2::THR _YACCO2_CALL_TYPE
NS_terminals_phrase_th::AR_Rsym_def1(yacco2::Parser* Caller_pp)
{
 yacco2::KCHARP ar_name = "AR_Rsym_def1";
 #include "war_begin_code.h"
   using namespace NS_yacco2_T_enum;
   for(i=1;i<=ie;++i){
     if(Caller_pp->pp_accept_queue__[i].accept_token__->enumerated_id__
       != NS_yacco2_T_enum::T_Enum::T_Err_no_terminal_key_present_){
       goto arbitrated_parameter;
     }
  }
 #include "war_end_code.h"
}
```

It is a function with a naming protocol of `AR_` with its rule name `Rsym_def1` within the grammar's namespace `NS_terminals_phrase_th`. The included files bounding the arbitration code provides the common logic to set things up for queue traversal, and its queue cleanup once the selected T has been chosen before returning from the thread. The passed in parameter is the parser acting on this grammar: `Caller_pp`. The `Parser` is detailed in chapter 6.

5.3 Playing the odds

Lines of symbols being recognized by a `Parser` are not too interesting. Let me raise some issues when developing a parser that might pique your curiosity:

- Ambiguity
- Error catching
- Relaxed token recognition

5.3.1 Doh? ambiguity

Like anyone i blow explicatives when this occurs: of course the mild variety — jees, gosh and al. The hard part is where and why it's happening. Ambiguity occurs when 2 or more parse activities are happening at the same time: do i shift or do i reduce. The lr(1) algorithm detects this situation. How the ambiguous state was arrived at, and reported on for the grammar writer to fix will be discussed later in the book.

Some quick fixes: avoiding the reduction
If only shifts were occurring this will never give an ambiguous context. Only when reduces are happening with other competing shifts or other reduces might it occur. So what is this quick fix? There are 2 ways to get out of this brine: shift and shift some more.

First shift: `|.|` is your Pied Piper.
It is the third T type tested for by the parser in the shift pecking order: the current T is tried first, then the possibility of catching a rogue by the `|?|` presence in the parse state. As it is not in the token stream, it becomes a shift out of the current parse state regardless of the wild token abandonment. The `|.|` symbol can be strung together as many times within a subrule: no restriction holds. Its value is get away from the

reduction that is causing the conflict. Well Dave this is nice but please give a real example where a conflict needed resolution. Let's go back to the shortest grammar given: end-of-line detector. A bit of explanation is required before i fully explain the value of the `parallel-la-boundary` expression introduced in Section 2.4.2. Threaded grammars use this expression to fine-tune the grammar's end-of-token-stream: call this the thread's lookahead. A thread can be called from anywhere within a segment of the token stream; what is its end-of-token-stream? Well the answer is it depends on the context. For the end-of-line detector, its end-of-token-stream could be all the terminals that are not linefeeds, or carriage returns. Well if the number of Tes in a grammar's vocabulary is big, this set will be large. So how can one reduce this set size? Da da to the rescue: the `eolr` symbol. It represents all the symbols in the T vocabulary including itself. So this shrinks the end-of-token-stream set to 1 T entry. But it could be too general for a context sensitive boundary. Well there are 2 ways to handle this:

1. Have a way to remove the offending T(s) from the lookahead set of `eolr`. This is done by allowing a T addition/subtraction type expressions in `parallel-la-boundary` construct. This will be explored later in the book.

2. Just shift out of the conflict using the `|.|` symbol.

Back to the end-of-line detector in the *eol.lex* grammar. The ambiguous situation is caused by using only the `eolr` in its `parallel-la-boundary` expression as i wanted to lower the number of items in its lookahead set. The conflicting 2 subrules are its last 2 subrules having the common character prefix `"x0d"`. The offending conflict is the 2nd subrule wants to reduce against all the Tes given by the thread's lookahead set that represents all-the-terminals while the 3rd subrule wants to shift its 2nd character `"x0a"` which is included in the lookahead set. I felt the use of `|.|` in the 2nd subrule to shift out of the conflict merited the saving of approximately .6k Tes within its end-of-token-stream set and it was much more efficient in its reduction's table symbol lookup. Here i made a planned decision to use a quasi ambiguous situation for space/run efficiency reasons. Now u might raise the question: but what happens to the calling grammar that receives this T but has an invalid lookahead T? Does this not lower the sensitivity to reporting the error back to the programmer(spelling mistake intended). Well not really. It just delays the error detection back to the calling grammar.

Second shift type: Parallelism the splitter of contexts be it rosy or blue.
Use of threads allows one to shift the ambiguity into 2 competing contexts. Each thread can deal with its own situation and return its findings. Again give me an example. What about a subset/superset situation? A bit ambiguous? Try this: without a proper symbol table, 2 competing threads could be: 1 recognizes an identifier while the other just keywords. Here the identifier is a superset of the keywords and will recognize and report the keywords as an identifier. If u try this within one grammar, u'll get that euphemism: "not a lr(1) grammar". Your example is a bit forced as the identifier should handle both situations using a symbol table lookup[9], this gives u the power of what threads can do and how arbitration selects the keyword over the identifier. Nice but i have say 40 keywords in my language, do u expect me to explicitly test for each one in the arbitrator code? Well the arbitrator loops through the "accept queue" looking for the subset item. This can be done in 2 ways. Can u figure out how?

Clue 1: The enumerate id of the item is tested.

Clue 2: The loop is only entered if the subset condition is present.

Clue 3: Think complement in your test[10]

.

5.3.2 Pinpointing dhem errors

Dhem bone dhem bones and dry bones... U the grammar writer don't make mistakes only the programmers of your language do. So lets look at errors and what u'd ideally like to do. Early detection would be the

[9] This was how it was done originally in O_2. Later in the book, it will be described with a more efficient method.
[10] `==` or `!=` against the superset's enumeration id. It's your taste.

ideal. Across the parse spectrum errors are unexpected breaks in the token symbols delivered that the parser expects when walking the symbol lines of subrules. So my take was to allow a programmable way of catching error conditions within any type of grammar. To do this i needed to create a new symbol `|?|`: represents the questionable situation. I wanted the symbol to be eye catching with intent when reading the grammar so now u know why the use of the question mark. It is not part of the token stream being read but a parse condition just like the `eog` symbol. Now explicitly placing this in a subrule allows the grammar writer to act on the situation thru syntax directed code. It also requires a way to stop parsing gracefully as the subrule having the `|?|` symbol is a catcher only. The subrule is a dead-end and the parse should not continue after this point though it could be reduced. Again there is a way to indicate to the `Parser` to stop parsing. 2 ways of stopping a parse there are: stop gracefully and stop abruptly. The abrupt stop is an abort statement in a programmed way that can still return an error T by use of the `failed` facility. The graceful way will consider the parse as successful. In a called thread, it has 2 ways to return an error thru 2 queues: Accept and Error. The accept queue's content is discriminated against and emptied with the winning T used by the parser to continue along a subrule. In this case the shift symbol is the `|?|` symbol whose content is the returned error T. I'll rephrase this, though the symbol is `|?|` in the state's table, the parsed stacked frame symbol is the nondescript `CAbs_lr1_sym` symbol. It is an abstract symbol that gives commonality to all the inherited symbols of T and Rules stature.

Please reflect on this: what happens to the subrule that contains this symbol and its reducing lookahead set is out of wack? How can one reduce the subrule? The Parsing chapter 6 details its facilities while Chapter 9 "Error: how to catch a rogue" examples dhem strategies.

Now u raised the issue of errors placed in an Error queue. Again this will be detailed later. To close it off the Error queue's contents are forever. Typically its contents are handled by an error grammar in an error procedure that does what it wants and typically shuts down the parse process.

5.3.3 `|+|` relaxing your fingers

I'm a bit lazy and if i can reduce my efforts with a more powerful expression i will. Take the wild symbol: `|+|`. Without it u are a slave to explicitly state all your intentions. If the intentions are small then that's okay. But if the recognized expression has many variations, then this just becomes laborious in coding all them subrules which is unacceptable. It bloats the emitted lr1 tables, bloats the grammar's readability, and bloats your impatience. This sounds great but are there any caveats?

Bird's view 1: compulsive eating
So regardless of the T seen the parser just keeps consuming. How do u get out of this situation and how can u determine when the border has been reached? Let's look at the 2nd question. As the current token is nondescript; it is wild without a name. Use of the enumerate id within the syntax directed code is the only way to identify it explicitly. The identity can be within a class of Tes by their start/stop enumerate values. Call this range finding. The 1st question asked is: the `Parser` is a consumer that could overrun its token stream. The generic `Parser` guards against this condition with its own abruptness.

Bird's view 2: how to diet?
But how do u stop these hunger pangs? Not a fair question as i had not exposéd the `Parser`'s facilities. Well i'm reeling in some futures. But how would u deal with it? There is a parser facility to turn "on" and "off" this craving: `set_use_all_shift_on` or `set_use_all_shift_off`. To stop it requires the grammar writer to explicitly turn it off when the threshold has been reached or u'll overrun. This allows the `Parser` to continue parsing up to the "Start rule" instead of tail chasing the tokens in this spin pit till obesity sets in.

The explicit versus implicit token condition.
O_2 allows u to mix specific Tes with the open-ended T within a rule's subrules. The `Parser` attempts to shift from the explicit token and works back to the general. So u are open to your own coding style.

5.4 Concluding remarks on generality?

The |+| facility within a grammar is heaven. It's a wild card in the token stream where the covered token gets tested. It saves your grammar's tables from bloat and allows u to snap your fingers to your favourite tune while grammar coding. It deals with the good side of parsing. |?| catches the bad side of parsing: general errors. Each symbol can appear in the thread call expression and in local subrule expressions. This gives 2 conditional contexts that can be dealt with: called threads, or the local subrule. The parse priority is: thread calls are tried first. If they are not present or the called threads do not return a T, then the local subrules are tried. Here are 2 examples of dual uses:

```
Rphrase (){
  ->  ||| "T-enum-phrase"
      NS_T_enum_phrase_th::TH_T_enum_phrase_th {
  op
    ADD_TOKEN_TO_PRODUCER_QUEUE(*sf->p2__);
  ***
  }
  ->  ||| |?| NULL {
  op
    ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__);
  ***
  }
  ->  |?| { // error due to no first set to start thread
  op
    CAbs_lr1_sym* sym = new Err_no_open_parenthesis;
    sym->set_rc(*rule_info__.parser__->current_token(),__FILE__,__LINE__);
    ADD_TOKEN_TO_ERROR_QUEUE(*sym);
  ***
  }
}
```

The above example shows that it expects a specific T: "T-enum-phrase" to be returned from the called thread. Use of the |?| within the called thread allows one to trap the other returning Tes from the called thread and to report an error. The comment against the subrule -> |?| gives u a glimpse into the mechanics of how a thread is called. It is O_2linker that determines the first set of terminals that the thread can start parsing with. This is equivalent to your recursive descent parsers.

The following 2nd example is a kaleidoscope of thread calls and a local wild catcher taken from the grammar dealing with the lookahead expression of parallel-la-boundary. I sampled this to get u the reader reved up for future explorations into these facilities. Their code and comments are an introduction to the type of syntax directed code that one normally uses.

```
Rtok  (){
  -> ||| "c-string" NS_c_string::TH_c_string{
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p2__);
    ***
    }
  -> ||| "c-literal" NS_c_literal::TH_c_literal{
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p2__);
    ***
    }
  -> ||| "#***" NS_o2_code_end::TH_o2_code_end{
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      T_eocode* k = sf->p2__;
      k->set_auto_delete(true);
      rule_info__.parser__->set_use_all_shift_off();
```

```
      fsm->exp_->la_tok_can()->push_back(*yacco2::PTR_LR1_eog__);
      fsm->exp_->la_tok_can()->push_back(*yacco2::PTR_LR1_eog__);
      RSVP(fsm->exp_);
      fsm->exp_=0;
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
  -> ||| identifier NS_identifier::TH_identifier{
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p2__);
    ***
    }
  -> |+| {
/@
Watch out for overrun.
@/
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      CAbs_lr1_sym* la_sym = sf->p1__;
      using namespace NS_yacco2_T_enum;
      using namespace NS_yacco2_terminals;
      int id = la_sym->enumerated_id__;
      if(id == T_Enum::T_LR1_eog_){
        CAbs_lr1_sym* sym = new Err_no_end_of_code;
        sym->set_rc(*la_sym,__FILE__,__LINE__);
        RSVP(sym);
        rule_info__.parser__->set_stop_parse(true);
        return;
      }
      fsm->exp_->la_tok_can()->push_back(*la_sym);
    ***
    }
  -> ||| "bad eos" NULL {// errors
    op
      RSVP(sf->p2__);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
  -> ||| "bad esc" NULL {// errors
    op
      RSVP(sf->p2__);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
  -> ||| "comment-overrun" NULL {// errors
    op
      RSVP(sf->p2__);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
  -> ||| |+| NS_la_lrk_T::TH_la_lrk_T {
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p2__);
   ***
    }
 -> "+" {
    op
```

```
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p1__);
    ***
    }
 -> "-" {
    op
      Cla_expr_src* fsm = (Cla_expr_src*)rule_info__.parser__->fsm_tbl__;
      fsm->exp_->la_tok_can()->push_back(*sf->p1__);
    ***
    }
 -> Rlint
}

Rlint  (){
  -> ||| lint NS_lint_balls::TH_lint_balls
  -> |.|
}
```

Notice the rule `Rlint` is referenced in one of `Rtok`'s subrules. It is part of the possible thread calls for the parse state where `Rtok` is closured. Why was it not added like the others instead of going thru another indirection by referencing the `Rlint` rule? Elsewhere in the grammar, this rule is also used. It's a way to duplicate its subrules thru closure[11]. The first set of each thread determines whether it gets called depending on the current parse token[12].

An ending thought: what happens when both `|+|` and `|?|` are competing in the same parse state? Is there a conflict of generalizations? ie a generalized ambiguity?

5.5 Where is epsilon?

Seems u missed commenting on the ϵ condition: no symbols present in a subrule. I did this intentionally as `|.|` is an explicit way of dealing with it and it serves more situations. Yes a subrule with no symbols is allowed. Yes it is more efficient in that there is no shifting onto the parse stack with a representational symbol `|.|`. From experience i found through convenience it did the same thing as an implicit ϵ . What i like about `|.|`: it is an explicit statement of a quasi epsilonal situation. It gets crossed referenced in the generated documents and when reading the grammar, it brings home its intent. It's your call in taste or custom. At least try it and I think u'll lean my way.

5.6 Unearthing a rule: anatomy secrets

Here's a sample rule in C++patois:

```
struct Rangled_string:public yacco2::CAbs_lr1_sym {
  Rangled_string(yacco2::Parser* P);
  public:
  void sr1();
};
```

Consistency comes from C++'s inheritance against `CAbs_lr1_sym` symbol. All symbols from a grammar's vocabulary inherit from this abstract symbol: rules and Tes. Have a reread on the terminals describing `CAbs_lr1_sym`'s attributes. From the above, a rule contains its subrules. In this example there is only 1 subrule present. If there is no "syntax directed code" associated with this subrule, it will not be present.

[11] My apologies for fast forwarding of these expressions; they will be developed later.

[12] Don't fret. Optimizations are abounding here. Each T has a list of what threads it can be called. My journey to get me to this stage mandated these optimizations for O_2. Without them, O_2 would be a curiosity and not a serious contender: As saying goes "You're benched when u don't perform".

5.7 Putting to sleep Rules

U are now versant on Rule coding. How to embellish their content by coding directives, fielding error situations, chronicling their woes, and an inkling on controlling parsing activity in wild token abandonment. Experiment experiment and improvisation is the best way to test your coding sensitivity in parser control and error reporting. Read other grammars to see what can be done and u'll be soon a grammar sommelier. Now on to the **Parser**.

Chapter 6

Parser

In chapter "What are grammars?" sections "O_2's frontend parser" 2.5.1 and "More comments on *Parser*" 2.5.7 outlines its parametric requirements and some examples of its use in the O_2's command line parsing and lexing of its input. Please have a reread of these sections to refresh your memory on `Parser` class as i will start discussing its internal methods.

6.1 How do u activate parsing?

A simple question asked but subtleties to follow. The `Parser` class object receives its content when defined. A parser requires a monolith grammar (not threaded) to parse with. Its input and output containers of terminals can be non-existant. Hemm this sounds rather bizzare and strange: no input to parse with or any outputted results. In the customary way, both Tes containers (input: supplier and output: producer) are present. In the unconventional sense, the grammar could be a logic automaton using epsilon action points. These bizzare ways will be highlighted later. I wanted to startle u the reader from my drone.

Here are the 2 parsing methods to deal with some form of input:

1. `parse_result parse();`
   The returned result is one of the enumerates: erred, accepted, reduced, paralleled, no_thds_to_run. The 2 enumerates of import are `erred` and `accepted`. All others are internal hat racks of parsing actions.

2. `bool spawn_thread_manually(yacco2::USINT Thread_id);`
   A false returned value indicates that it was called as a procedure and true as a thread[1].

And u thought there was only 1 way to "cook a parse". In my combinatorial way, why not allow manual calling of parallel threads out of your own syntax directed code. The `Thread_id` parameter is manufactured by O_2linker. All threaded grammars have their own global thread record. Just show me! Here's an example dealing with escape sequences where i wanted to parse inside `op` as a finite state automata.

```
Ropen_angle (){
  -> "<" {
  op
    Cangled_string* fsm = (Cangled_string*)
                rule_info__.parser__->fsm_tbl__;
loop:
    switch (rule_info__.parser__->current_token()->enumerated_id__){
     case T_Enum::T_raw_lf_: goto overrun;
     case T_Enum::T_raw_cr_: goto overrun;
     case T_Enum::T_LR1_eog_: goto overrun;
     case T_Enum::T_raw_gt_than_: goto closestr;
     case T_Enum::T_raw_back_slash_: goto escseq;
     default: goto other;
```

[1] U got it! Optimization.

```
    }
  closestr:{ // end of string
     rule_info__.parser__->get_next_token();
     return;// end of angled string
  }
  overrun:{
     CAbs_lr1_sym* sym = new Err_bad_eos;
     sym->set_rc(*rule_info__.parser__->start_token__
                ,__FILE__,__LINE__);
     RSVP(sym);
     rule_info__.parser__->set_stop_parse(true);
     return;
  }
  escseq:{ // what type of escape
     using namespace NS_esc_seq;
     Parser::parse_result result =
       rule_info__.parser__->
         start_manually_parallel_parsing(ITH_esc_seq.thd_id__);
                                        ^^^^^^^^^^^^^^^^^^^^
     if(result == Parser::erred){
       // in this case, it will not happen: here for education
       rule_info__.parser__->set_abort_parse(true);
       return;
     }
     // process returned token
     Caccept_parse& accept_parm =
                  *rule_info__.parser__->arbitrated_token__;
     CAbs_lr1_sym* rtn_tok = accept_parm.accept_token__;
     int id = rtn_tok->enumerated_id__;
     accept_parm.accept_token__ = 0;
     if(id != T_Enum::T_T_esc_seq_) {
       RSVP(rtn_tok);
       return;
     }
     T_esc_seq* finc = (T_esc_seq*)(rtn_tok);
     fsm->copy_str_into_buffer(finc->esc_data());
     rule_info__.parser__->
         override_current_token
              (*accept_parm.la_token__,accept_parm.la_token_pos__);
     delete finc;
     goto loop;
  }
  other:{
    fsm->copy_kstr_into_buffer
              (rule_info__.parser__->current_token()->id__);
    rule_info__.parser__->get_next_token();
    goto loop;
    }
 ***
  }
}
```

How is this thread id `ITH_esc_seq.thd_id__` constructed? The thread name comes from the "esc_seq.lex" grammar: TH_esc_seq and prefixes it with "I". The thread dispatch record definition is:

```
struct Thread_entry {
  yacco2::KCHARP           thread_fnct_name__;
  yacco2::Type_pp_fnct_ptr thread_fnct_ptr__;     // ptrs addresses
  yacco2::USINT            thd_id__;
  yacco2::Type_pc_fnct_ptr proc_thread_fnct_ptr__;// resolved by linker
```

```
};
```

Here is O_2linker's emitted code for one of its thread table records.

```
yacco2::Thread_entry ITH_esc_seq =
   {"TH_esc_seq",NS_esc_seq::TH_esc_seq,9,NS_esc_seq::PROC_TH_esc_seq};
```

The first field is for tracing purposes. The 2 pointers deal with the thread and its optimized clone the procedure. `thd_id__` is O_2linker's assigned thread id: `9` created in lexicographical order by thread name. This should jolt u like a caffeine spike? More later, just trying to keep u awake.

6.2 The brown parse() bag

There is no magic to it. When the parse object is initialized, the starting parse point in the input container `supplier__` is given, and the T fetched. The terminal starting point defaults to the beginning: 0. U can start a parse anywhere within the token stream. Give me an example please. Threaded grammars. Another one svp. What about a form of recursive decent parsing. O_2 used this strategy in parsing its own phases: for example fsm, rules, Tes classes where each parsing phase starts consuming the Tes after its keyword T[2]. All the dispatched procedures return their parsing result. Either the parse is successful or dang its got problems and u have to deal with them.

6.2.1 Recursing the descent

Here's an example taken from the lexical parse "pass3.lex". The subrule extract demonstrates how one can test the current T and call a procedure from its syntax directed code. The following code samples are in the Cweb dialect. I'm trying to get u acclimatized to its use but don't place too much importance now on the Cweb directives having some combination of `@`. Mixed inside these comments are TEX's macros starting with a `\`. Their content are also directives to the generated documents. Please give the comments an eyeballing while skipping over these directives.

```
-> ||| |+| NULL{// keywords emitted from identifier
/@
@*2 Dispatch keyword to process its construct phrase.\fbreak
Again
neat stuff with its co-operation of top/down and bottom-up
parsing paradigms. Notice that i use the catch all ``|+|''
to show case it where as i could have referenced``keyword''.
This is how the parser works:\fbreak
\ptindent{1)   thread call}
\ptindent{1.1) check state's table for specificly returned T}
\ptindent{1.2) check for ``catch all'' returned presence}
\ptindent{2) try current token T to shift if thread call did't work}
\ptindent{     or is not present in state}
\ptindent{3) check for ``catch all'' presence in the state to shift}
Note: there are 2 types of ``catch all'': one for returned T from
thread calls and the other for regular parsing.
@/
  op
    CAbs_lr1_sym* key = sf->p2__;// extract specific keyword
    yacco2::INT cont_pos = rule_info__.parser__->current_token_pos__;
    CAbs_lr1_sym* cont_tok = rule_info__.parser__->current_token();
    bool result = // dispatch on phase keywords: eg. fsm, rules
     PROCESS_KEYWORD_FOR_SYNTAX_CODE(*rule_info__.parser__
                                    ,key,&cont_tok,&cont_pos);
    if(result == false){
```

[2]Here is an explanation of why. The parser recognizes the subrule of the returned keyword token from the called thread, and also the lookahead T and position to continue parsing from is returned back for the parser to realign itself back into the token stream.

```
      rule_info__.parser__->set_abort_parse(true);
      return;
    }
    ADD_TOKEN_TO_PRODUCER_QUEUE(*key);
    // adv. to phase's LA pos
    rule_info__.parser__->override_current_token(*cont_tok,cont_pos);
  ***
  }
```

Notice PROCESS_KEYWORD_FOR_SYNTAX_CODE receives as parameters the calling parser and the keyword token to dispatch on. The remaining 2 parameters will be explained later. So lets look at this procedure.

6.2.2 A song: Re cursing Re

Please read its content. They're my comments to myself when i wrote it. Though ugly in its free form, u should see them in the grammar's generated documents[3].

```
@*2 Process syntax code: |PROCESS_KEYWORD_FOR_SYNTAX_CODE|.\fbreak
This is the dispatch routine called
from the |pass3| grammar per phase to parse.
What you will see within the |pass3.lex|
grammar is a subrule with the appropriate keyword.
Originally the grammar thread |identifier| returned a
carrier terminal called |keyword|.
Inside it was the individual keyword recognized that had to be extracted
by the syntax directed code.
This was done to lower the amount of subrules within |pass3|.
This was the past truth; evolution in the wild card facility now
lowers the explicit programming of subrules without need for the carrier T keyword:
just return the specific T.
The specific keyword is now passed to this routine for processing.
If a success is returned, the extracted keyword is put into
 the output container
while an error stops the parse with the error placed into
the 'error queue'.
The carrier terminal |keyword| has the 'AD' auto delete
attribute defined
so |keyword| is deleted when it is popped from the parse stack.

This is a simple way to deal with individual components.
Within each of the to-be-parsed phases, their syntax is
grammatically verified.
As it comes out of a lexical grammar, down the road, the
syntax part must check for the proper sequencing of these phases.
For example, the production component cannot come before
the LRk phase etc.
Due to a fixed ordering of T vocabulary components enumeration,
T's components must be in lrk,rc,user, and error terminals parse order
as i add their forests to the grammar tree while they are being built!
Out of order will violate the T vocabulary's enumeration
as i walk the tree as created when enumerating.
@<accrue source for emit@>+=
extern
bool
PROCESS_KEYWORD_FOR_SYNTAX_CODE@/
    (yacco2::Parser& Parser@/
    ,yacco2::CAbs_lr1_sym*  Keyword@/
    ,yacco2::CAbs_lr1_sym** Cont_tok@/
```

[3]Promises promises and ... U'll be pleased when i review them later. I think?

```
    ,yacco2::INT*           Cont_pos)@/
{
  using namespace NS_yacco2_T_enum;
  using namespace NS_yacco2_terminals;
  using namespace yacco2;
  switch (Keyword->enumerated_id__){
   case T_Enum::T_T_fsm_ :{
     @=T_fsm* fsm = (T_fsm*)Keyword@>;
     return
       process_fsm_phrase(Parser,*fsm,Cont_tok,Cont_pos);}
   case T_Enum::T_T_enumeration_ :{
     @=T_enumeration* enumer = (T_enumeration*)Keyword@>;
     return
       process_T_enum_phrase(Parser,*enumer,Cont_tok,Cont_pos);}
   case T_Enum::T_T_error_symbols_ :{
     @=T_error_symbols* err = (T_error_symbols*)Keyword@>;
     return
       process_error_symbols_phrase(Parser,*err,Cont_tok,Cont_pos);}
  ...
   case T_Enum::T_T_rules_ :{
     @=T_rules* rule = (T_rules*)Keyword@>;
     return
       process_rules_phrase(Parser,*rule,Cont_tok,Cont_pos);}
   default: {
     CAbs_lr1_sym* sym = new Err_not_kw_defining_grammar_construct;
     sym->set_rc(*Keyword);
     Parser.add_token_to_error_queue(*sym);
     return Failure;}
  }
  return Success;
}
```

I think this is neat as the grammar has jumped out of the throes of an lr(1) parser and allowed another context to do its own thing which so happens to be parsing along some token road. What do u think? 2 non fingers up or lunch for the lions?

6.2.3 Bottoming out the recursion

Well each phase has its own parsing procedure that contains a monolithic grammar to parse the phase. It is determined by the keyword's enumerate id. I felt the recursive descent model expresses well the overview approach to each phase, and i wanted to show to myself that the mixed genre of parsing fits well. Here's a sample process: `process_rules_phrase`.

```
@*3 Driver of |process_rules_phrase|.
@<accrue source for emit@>+=
bool
process_rules_phrase@/
      (yacco2::Parser&  Calling_parser@/
      ,NS_yacco2_terminals::T_rules&  Phrase_to_parse@/
      ,yacco2::CAbs_lr1_sym** Cont_tok@/
      ,yacco2::INT*  Cont_pos)@/
{
  @<enumerate Terminal alphabet@>;
  using namespace NS_yacco2_terminals;
  using namespace yacco2;
  using namespace NS_rules_phrase;
  TOKEN_GAGGLE op1;
  yacco2::INT start_pos = Calling_parser.current_token_pos__;
  token_container_type* ip1 = Calling_parser.token_supplier();
```

```
  TOKEN_GAGGLE* er1 = Calling_parser.error_queue();
  Crules_phrase rule_ph;
  Parser p(rule_ph,ip1,&op1,start_pos,er1); // setup containers, T
  p.parse(); // go parse the rules construct

  if(er1->empty() == YES){// cheeck those booboos
    *Cont_pos = p.current_token_pos__; // <== lookahead reset
    *Cont_tok = p.current_token();     // <== material
    TOKEN_GAGGLE_ITER i = op1.begin();
    CAbs_lr1_sym* sym = *i;
    @=T_rules_phrase* t = (T_rules_phrase*)sym@>;
    Phrase_to_parse.rules_phrase(t);
    if(O2_RULES_PHASE != 0){
     CAbs_lr1_sym* sym = new Err_already_processed_rule_phase;
     sym->set_rc(Phrase_to_parse);
     p.add_token_to_error_queue(*sym);
     goto error_exit;
    }
    O2_RULES_PHASE= t;
    AST* gt = t->phrase_tree();
    BUILD_GRAMMAR_TREE(*gt);
    @<enumerate Rule alphabet@>;
    return Success;
  }
  error_exit:
   lrclog << "error in rules-phrase" <<  std::endl;
   return Failure;// error placed in |error_queue| by called thread
}
```

Sweet. The called procedure sets up its own parser and associated grammar using the passed in parser `Calling_parser`. And away it goes a consuming. Eventually it stops parsing and returns how it did: `true` for successful and `false` for any number of reasons.

6.2.4 Stop! I have a mental block.

Here is a bottom-up parser dispatching on the keyword and letting someone else continue parsing along its token stream. How does the calling parser know where to continue within its token stream when the called procedure has eaten some of its tokens? After the parse, the following excerpt checks for any errors by their presence in the "Error queue". No errors then it sets the last 2 parameters passed to the procedure indicating the repositioning in the token stream. I call this the future lookahead: the next token to continue with, and where it sits within the token stream.

```
  if(er1->empty() == YES){
   *Cont_pos = p.current_token_pos__; // <== lookahead reset
   *Cont_tok = p.current_token();     // <== material
    ...
```

Whooa i see above `p.current_token_pos__` use. What is this? I understand `p.current_token()` is an internal method of `Parser`. Okay i withheld some information. Variables can be played with by either their methods or directly by name. They are publicly available for speed. All of `Parser`'s variables names are suffixed by 2 _.

6.2.5 The resultant

In the dispatcher a returned success sets its parser's token stream parse point and future token as current. The `override_current_token` method below does this.

```
  // adv. to phase's LA pos
  rule_info__.parser__->override_current_token(*cont_tok,cont_pos);
```

So what do u think? U the grammar writer have both worlds to play with: of course actioned out of some bottom-up perspective.

6.3 Tweedle de dum and those thing-me-bobs of Parser

Let's look at what can be queried and played with inside the parser. The 1st thing that comes to mind is the token stream: its token containers. When a parsing begins, there is its "starting token" and position within the input container. This start marker keeps itself around throughout the parse for various situations but for only 1 reason: GPSing. When a new T is created be it for the good or for the bad, this marker can be used to associate the T with a source file position. Then there is the current parse token and its position within the token stream. Well what about the parsing stack itself? Yep. And the parsing conditions, results, and actions? Yep again. All these entities can be queried and overridden. So let's systematically look at each one.

6.3.1 Parser's input token stream: content and position

- `CAbs_lr1_sym* get_next_token()`
  Get next token and set it as parser's current token. It automatically increments the current token pos and fetches it. It will return: T or eog signalling the end-of-the-token-stream reached if the input container exists, or a `null` value if there is no container present.

- `CAbs_lr1_sym* get_spec_token(yacco2::UINT Pos)`
  Get any token from the input container. The token number is relative to 0. This method allows one to iterate thru the input token stream without hurting the parser. Only returns Tes that are in the container. If the end-of-container is reached, a `null` value is returned which is the same for the non-existent container.

- `CAbs_lr1_sym* current_token()`
  Returns the current token that parser is/will be working with. The future tense indicates the current parse action being done has consumed it. For example a shift operation has already pushed its current token onto the stack and fetched the next token as the current T for the parser to deal with. Querying this inside the subrule being shifted gives u the future T. Of course u remembered that the `sf` stack frame provides u with its shifted T.

- `CAbs_lr1_sym* start_token()`
  The start token beginning the parse be it a monolithic or threaded grammar. It gets assigned at the initialization of a parse. When threaded grammars are reused, each time they are called the start token is changed to the point in the token stream segment from where it starts parsing from. Normally used to supply the source file coordinates when creating a new T.

- `CAbs_lr1_sym* set_start_token(CAbs_lr1_sym& T)`
  Set the start token marker for the parser. Each time a thread grammar is called, its start token and current parse token will be re-aligned to its current token stream segment.

- `yacco2::UINT start_token_pos()`
  Where the starting T of the parse resides in the input container.

- `void set_start_token_pos(UINT Pos)`
  Declare its position within a called thread.

- `void reset_current_token(yacco2::UINT Pos)`
  Realign the parser's T and its input position. After a return from a successful thread call, this method realigns the parser's T position in the token stream to continue parsing.

- `void override_current_token(CAbs_lr1_sym& T,UINT Pos)`
  Re-align the parser's current token and position in the input token stream. It does double duty.

- `void override_current_token_pos(UINT Pos)`
  Re-align the token stream position for the parser to continue parsing from.
- `yacco2::UINT current_token_pos()`
  Token stream position of current T being parsed within the input container.

6.3.2 Parse stack: hygiene and content

Normally the grammar writer does not need access to it unless the parse stack items are being probed or the `op` directive for the rule has to get at its content. Later in the chapter the stack frame will be developed with examples of probing.

- `void cleanup_stack_due_to_abort()`
  Parsing hygiene for another round of parsing. It off loads its contents reacting to the "AD" and "AB" attributes of each popped item. It resets the stack with its start state as an optimization.
- `yacco2::lr1_stk* parse_stack()`
  Get the parser's internal parse stack. Normally not used but there for u the grammar writer to possibly iterate thru its items. Automatic tracing also does this for free.
- `yacco2::INT no_items_on_stack()`
  Tales of the stack for iterating.
- `Cparse_record * get_stack_record(UINT Pos)`: specifically access the stack frame per pushed symbol. The stack frame subscript is relative to 0.
- `yacco2::Cparse_record * top_stack_record()`
- `void remove_from_stack(INT No_to_remove)`
  Pop the stack up to the number passed. The method is forgiving when the requested number is outside the stack's number of items.
- `void add_to_stack(yacco2::State & State_no)`
  Parse stack mechanics.
- `yacco2::INT current_stack_pos()`
  Details on where the top item is on the stack.
- `void clear_parse_stack()`
  Hygiene when the grammar/parse winds down for another round of cycling.
- `CAbs_lr1_sym* get_spec_stack_token(yacco2::UINT Pos)`
  Returns the pushed symbol in the specific position.

6.3.3 Accept queue: the effects of outer-parses

The following macros `RSVP(P1 Token)`, `RSVP_WLA(P1 Token, P2 la token, P3 la pos)`, and `RSVP_FSM(P1 Token)` add a T symbol into the accept queue. Each macro is a collection of statements that adds the potential token to be considered for acceptance, and the future token stream restart point into the calling grammar's "accept queue". Let me repeat this in another way, the called thread receives its parsing requestor and it is its responsibility to at least wake up the caller and possibly return some form of result into the requestor's accept queue. Note i said possibly as the grammar may wish to not return anything but just abort. It still has the responsibility to clean itself up for another round of parse jigging, and if it is the last dispatched thread to end wake up the snoring ogre.

`RSVP` is used within a grammar's rules or subrules and defaults using the current token of the parser for it future lookahead restart point. Whereas `RSVP_WLA` allows u to return a different start point to continue parsing from. `RSVP_FSM` is used within the `fsm` context. Give me some contexts: its internal methods like `failed` or your own creations. The reason for the differences between the first 2 macros and the third is due to getting at the calling parser's context.

6.3.4 Other containers: Output, Errors, and Recycling

Tokens can be fetched outside of the parser's consumption regimen. These methods are available:

- `token_container_type* token_supplier();`
  Get the parser's input container.

- `void set_token_supplier(token_container_type& T_container );`
  Set or change the parser's input container. Not used very often but gives u the freedom to play around with. Same ditto remarks to the other containers.

- `token_container_type* token_producer();`
  Get the parser's output container.

- `void set_token_producer(token_container_type& T_container );`
  Set or change the output container.

- `ADD_TOKEN_PRODUCER_QUEUE(P1 Token)` macro
  Add to assigned parser's output container where its parameter is a T pointer. Called from a rule/subrule context. This allows one to output from a parser a terminal stream that becomes a supplier token stream for another grammar to parse from. `add_token_to_queue(CAbs_lr1_sym* T);` is the parser's internal method. Generated code example from a rule's context:
  ```
  rule_info__.parser__->add_token_to_producer(*Token);
  ```
  The Token is dereferenced and therefore must exist; the pointer cannot be nil. The same comment holds for the other to follow. Remember the rule's subrules are part of the rule's definition and have the same access to the parser's internals.

- `TOKEN_GAGGLE* error_queue();`
  Get the parser's error container.

- `void set_error_queue(TOKEN_GAGGLE& E_container );`
  Set or change the error container. It is restricted or limited to one type of container to override.

- `ADD_TOKEN_ERROR_QUEUE(P1 Token)` macro
  Add T to assigned parser's error container where its parameter is a T pointer. This becomes a holding queue of error class terminals that can be processed by a grammar dealing with the Errors Tes. To do this, this container becomes the supplier container to the parser. The parser's internal method is `add_token_to_error_queue(CAbs_lr1_sym* T);`. Here's its generated code out of a rule's context:
  ```
  rule_info__.parser__->add_token_to_error_queue(*Token);
  ```

- `ADD_TOKEN_ERROR_QUEUE_FSM(P1 Token)` macro
  Used inside a method of "fsm" to get at the "parser's" context. It adds a T to assigned parser's "Error container" where its parameter is a T pointer produced or contained in a `fsm` context. Here's its generated code from the `fsm` context:
  ```
  parser__->add_token_to_error_queue(*Token);
  ```

- `token_container_type* recycle_bin();`
  Get the parser's recycle bin.

- `void set_recycle_bin(token_container_type& T_container );`
  Set or change the Recycle container.

- `add_token_to_recycle_bin(CAbs_lr1_sym& T)` macro
  Add T to Recycle bin. It's more of a thought that probably has not too much merit but it's here for your taking. This becomes a holding container of possible Tes to eventually recycle back into amorphous memory.

Basically u can do anything u want to these containers. For example input files having nested include statements can be done as follows. The interesting thing is how the current parser passes a reference to itself to the procedure PROCESS_INCLUDE_FILE which provides all the calling parser's contents including its containers its input/output, and error containers. Here the called procedure lexes its output into the calling output's container, and watches out to not prematurely add an eog Tes due to the end-of-file condition as this is a token stream within a token stream where only the first file that is being lexed deals with the end-of-token condition.

Why are u also passing the output container as it is contained in the referenced parser? Just to reenforce my own understanding. Here is the excerpt for O_2's lexer dealing with nested include files. The comments should be enough for u to navigate through your own learning/understanding guideposts. There are 2 parts:

1. The Rtoken rule snippet where its subrule calls the PROCESS_INCLUDE_FILE procedure and the rule Rprefile_inc_dispatcher. It shows a thread call variation using the |t| symbol which is explained in "Threads" chapter 7.

2. The procedure PROCESS_INCLUDE_FILE shows how thru recursion the include files are processed using the same monolithic grammar "pass3.lex" that the calling parser uses.

```
Rtoken (){
... eluded subrules
  ->
  /@
  \Yacco2's pre-processor include directive.\fbreak
 \fbreak
 This demonstrates a nested environment
 where the grammar uses recursion by
 calling a function which contains the |pass3| grammar sequence.
 In this example, grammar |pass3|
 manually calls a thread via
 |start_manually_parallel_parsing|
 to get its file name to process.
 With the returned ``file-inclusion'' terminal,
 |PROCESS_INCLUDE_FILE| is called to parse
 the include file: a bom-de-bom-bom bump-and-grind sequence.
 The |use_cnt_| is a global variable that protects
 against the file include recursion of calling self
 until a stack overflow occurs.
 @/

"@" Rprefile_inc_dispatcher
... eluded subrules
}
Rprefile_inc_dispatcher  (){
  -> |t| "file-inclusion" NS_prefile_include::PROC_TH_prefile_include {
    op
      CAbs_lr1_sym* err = sf->p2__->error_sym();
      if(err != 0) {
        rule_info__.parser__->set_abort_parse(true);
        ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__);
        ADD_TOKEN_TO_ERROR_QUEUE(*sf->p2__->error_sym());
        return;
      }
      bool result =
           PROCESS_INCLUDE_FILE
                  (*rule_info__.parser__
                  ,*sf->p2__
                  ,*rule_info__.parser__->token_producer__);
```

```
      if(result == false){ // exceeded nested file limit
        rule_info__.parser__->set_abort_parse(true);
        return;
      }
      ADD_TOKEN_TO_RECYCLE_BIN(*sf->p2__);//file name inside
      return;
    ***
    }
  -> |t| |?| NULL {
    op
      sf->p2__->set_auto_delete(true);
      CAbs_lr1_sym* sym = new Err_bad_directive;
      sym->set_rc(*sf->p2__,__FILE__,__LINE__);
      RSVP(sym);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
  -> |?| {
    op
      CAbs_lr1_sym* sym = new Err_no_directive_present;
      sym->set_rc(*rule_info__.parser__->current_token()
                 ,__FILE__,__LINE__);
      RSVP(sym);
      rule_info__.parser__->set_stop_parse(true);
    ***
    }
}
```

The external procedure: PROCESS_INCLUDE_FILE handles the nested files. What's is interesting is its use of the "pass3.lex" grammar that is itself the same grammar from the calling parser. It demonstrates nested grammars calling nested grammars. O_2 handles this thru its thread dispatch table by dynamically activating and maintaining threads that can run, and procedural recursion where a monolithic grammar is enclosed within the recursing procedure as a local variable maintained on its procedural scope stack.

```
extern bool PROCESS_INCLUDE_FILE@/
      (yacco2::Parser& Calling_parser@/
      ,NS_yacco2_terminals::T_file_inclusion& File_include@/
      ,yacco2::token_container_type&  T2);
@*2 Process Nested include files: |PROCESS_INCLUDE_FILE|.\fbreak
This routine gets called from the |pass3| grammar
when Yacco2's include file expression has been parsed.
It uses the hardwired definition |Nested_file_cnt_limit|.
The one important point is how it appends
the newly processed include tokens to the calling parser's
producer container. |pass3| has a global variable to ensure that
only 2 |PTR_LR1_eog__| tokens are emitted to the producer container
from the first |pass3| invocation. All other recursively invoked
|pass3| guard against these extra end-of-grammar tokens
when they finish parsing.

The global |yacco2::FILE_CNT__| is incremented within the
|tok_can<std::ifstream>| container. This global is just a file counter
 of files opened by the template container.
To cope with recursion, there is the global |yacco2::STK_FILE_NOS__|.
This ‘‘call stack’’ global maintains the currently opened files
 so that tokens generated are associated with the top file.
Error reporting then GPSs to the source
line and character position within this tagged file
when an error token is created using this source file coordinates
taken from the faulty T token.
```

```
It's depth is tested for file recursion overrun.

Constraints:\fbreak
 \ptindent{ip1:  Calling parser environment}
 \ptindent{ip2:  include token to process}
 \ptindent{ip3:  Token container to append to}
Note: The filename to process has already had its existence check
done by the |T_file_inclusion| token.

Why the passing of the |Calling_parser|? There are 2 reasons all
related to error processing. Firstly, the error queue is needed and
secondly to abort the |Calling_parser|. Though the returned
result also indicates success or failure, I thought that
it was better to indicate by action rather than intent
what should be done. This is a little draconian but...

Error processing:\fbreak
 \ptindent{op1: nested file limit is exceeded}
 \ptindent{op2: bad file name}

Note: if an error occurs, the calling parser is aborted.
No error correction is done.
It stops the processing immediately so that the error
can be reported back to the grammar writer.

@<accrue source for emit@>+=
extern
bool
PROCESS_INCLUDE_FILE@/
      (yacco2::Parser& Calling_parser@/
      ,NS_yacco2_terminals::T_file_inclusion& File_include@/
      ,yacco2::token_container_type&  T2)@/
{
  using namespace NS_yacco2_terminals;
  using namespace NS_yacco2_err_symbols;
  using namespace yacco2;
  std::string* ps_fn = File_include.file_name()->c_string();
   lrclog << "Pre-process file: " << ps_fn->c_str() <<  std::endl;
  if(yacco2::STK_FILE_NOS__.size() >= Nested_file_cnt_limit){
    CAbs_lr1_sym* s =
        new Err_nested_files_exceeded(Nested_file_cnt_limit,*ps_fn);
    s->set_line_no_and_pos_in_line(File_include);
    s->set_external_file_id(File_include.tok_co_ords__.external_file_id__);
    Calling_parser.add_token_to_error_queue(*s);
    Calling_parser.abort_parse__;
    return Failure;
  }

  tok_can<std::ifstream> p1_tokens(ps_fn->c_str());
  if(p1_tokens.file_ok() == NO){
    yacco2::Delete_tokens(p1_tokens.container());
    CAbs_lr1_sym* sym = new NS_yacco2_err_symbols::Err_bad_filename(*ps_fn);
    sym->set_external_file_id(File_include.tok_co_ords__.external_file_id__);
    sym->set_line_no_and_pos_in_line(File_include);
    Calling_parser.add_token_to_error_queue(*sym);
    Calling_parser.abort_parse__;
    return Failure;
  }
  using namespace NS_pass3;
```

```
  Cpass3 p3_fsm;
  Parser pass3(p3_fsm // assigning the containers from its calling parser
              ,&p1_tokens
              ,&T2
              ,0
              ,Calling_parser.error_queue()
              ,Calling_parser.recycle_bin__);
  pass3.parse();
  yacco2::Delete_tokens(p1_tokens.container());

  if(pass3.error_queue()->empty() != YES){
    Calling_parser.abort_parse__;
    return Failure;
  }

  yacco2::STK_FILE_NOS__.pop_back();
  return Success;
}
```

6.4 Pavlovian parse behaviors?

Is it the tail wagging the dog or the fleas teasing the parser? I'm commenting to myself: parser's behaviour or my grammar's dictates? What is available to the grammar writer? The parser supplies the parsing context. Boy is that a stupid statement. But u only have 2 contexts to grab at: syntax directed code and syntax directed code. Again Dave your showing your intelligence quota. So bear with me, how does the grammar dictate it wishes within various contexts? This is done through the productions aka subrules or their parent rule. From there the context of the parse is declared and captured by the `op` directive and its behavioural modification. So the catching contexts are: the good and the bad and sometimes the ugly. The wild meta terminals of `|+|` and `|?|` generalize catching some point within the token stream. Specifics are explicitly programmed.

6.4.1 Querying and modifying behaviour

- `void set_use_all_shift_on();`
  This allows one to control the all-u-can-eat operator: `|+|`. If it is turned off the parser will not respect it in the parse state. Normally if the grammar turns its off then it has to be reset when it is recycled using `fsm`'s `op` directive.

- `void set_use_all_shift_off();`
  Stops the voracious appetite of the parser. This allows the parser to get out of bingeing. This is programmed by the grammar writer so that the parser will work its way up to the start state.

- `bool use_all_shift();`
  Checks the presence of this parsing option.

- `bool abort_parse();`
  Checks whether the parser aborted somehow be it explicitly programmed or due to the input T stream. Normally used in the `failed` last-chance method of the grammar.

- `void set_abort_parse(bool Yes_or_no);`
  A "yes" is bool `true`. Normally called to abort by the grammar writer as it stops the parser in its tracks. Use of the `failed` directive in the `fsm` allows one to do something. If nothing is done then the threaded grammar returns unsuccessful. The calling parser of the threaded grammar can still continue parsing. In a monolithic grammar, it is lights out.

- `bool stop_parse();`
  A way to test whether a stop parsing happened so that it does not work its way towards the start state: yao[4].

- `bool set_stop_parse(bool On_or_off);`
  A cleanup reset or a declaration to the parser to stop its parsing. It's a gentle way to stop parsing when turned on rather than the more abrupt abort. It is still considered a successful parse within a threaded grammar. Twined with this has to be a returned T back to the calling grammar be it an error T or a resulting T thru the "accept queue". It is the calling grammar who fields thru the thread call expressions and reacts accordingly by possible arbitration.

6.4.2 Partial summary

So the wildness of a parser can be trained, and on cue performances dance to your tune. In this context it was nice that u explained recursive descent requiring all this alignment but i'm a bottom-up threaded type of grammar writer using your compiler/compiler. Why put me thru all of this? Well the threaded model requires the same token stream alignment back at the calling parser but comes for free with the `RSVP` macro variants. Unfortunately there are no cookies for reenforcement though i hope this explanation does give u the knowledge to play around with O_2. U are open to your own improvisations.

6.5 Parse stack: probing and gropping

Ahh the parse stack. Normally invisible, and quietly huffing and puffing with it symbols and the oracle of their contexts. Let's look at it makeup and then I'll describe the coordination between the subrule and its boundary, and the parent rule when a reduce operation occurs. The goal is to feel familiar with its activities, how it registers parsing state contexts with the grammar's vocabulary.

Background the anatomy of the stack:
The stack is just an array of stack frames. The stack frame is composed of 2 parts:

1. the parse state
2. the shifted symbol

Figure 6.1 depicts this. It also has some additional attributes dealing with abort and rule reuse that are not meaningful for this discussion.

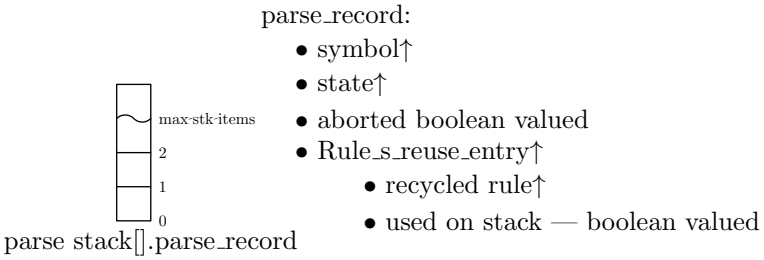


Figure 6.1: Parse stack overview

What happens in bottom-up parsing is the parser starts with State 1 without a shifted symbol. It is nonexistent until the parser fetches its current T and determines what action to do: shift, reduce, accept, or an error. Let's assume the current parse token allows a shift operation to occur. When it shifts the current T onto the stack, 2 things happen:

1. the current parse token being shifted gets registered inside the top parse stack frame and then a new T is fetched as the current parse token.

[4]Acronym for yet another optimization or a scream-in-the-nite

2. Now a new parse stack state frame is created and pushed onto the stack with its shifted into state set while its shift symbol zeroed out.

The 2 stack frames are straddled between the past state and its symbol that is shifted, and the present "shifted into" state[5]. This was recorded this way for tracing purposes. All the information about the parse table and how far it got is stored onto the parse stack. This goes on until the current parse state and current T says "u need to reduce". Now the straddle line between the previous stack frame containing the shifted symbol and the current parse stack just containing the parse state that says reduce me. To get at the shifted symbol on the parse stack, **u must go back to the previous stack frame**.

Here's an out-of-the-box tracing sample of just described parsing the `-t` option within the command line by grammar " o2_lcl_opt.lex".

```
o2 -t /usr/local/yacco2/compiler/grammar/eol.lex
```

The state number and the shifted symbol are reported in literal terms. Threads are identified by the operating system given thread-id number and by their name as more than one thread of same name could be running at the same time: nested recursion. Each stack item starts with its state number. If the state is shifting into another state the `--`, item shifting, and `->` indicates this. The parse stack is drawn in bottom to top order showing the various activities per trace line. Prefixing each line is the grammar's context that is reporting: thread or monolithic. The other stuff will be detailed in a later chapter but left there to whet your curiosity.

```
..1::o2_lcl_opt.lex::1----> 2
                        ^
  stack frame 1: '-' is pushed onto the stack and associated with State 1
  stack frame 2: 'shifted into' State 2 with no associated symbol
YACCO2_TH__::1::o2_lcl_opt.lex:: Popping items from stack # to pop: 1
..1::o2_lcl_opt.lex::1----> 2
   YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 2
.1::o2_lcl_opt.lex::1
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: -
YACCO2_TH__::1::o2_lcl_opt.lex:: Finished Popping items from stack
..1::o2_lcl_opt.lex::1--Rminus-> 3
                       ^^^^^^
  grammar's rule replacing its popped subrule's item '-'
...1::o2_lcl_opt.lex::1--Rminus-> 3--t-> 19
YACCO2_TH__::1::o2_lcl_opt.lex:: Popping items from stack # to pop: 1
...1::o2_lcl_opt.lex::1--Rminus-> 3--t-> 19
   YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 19
..1::o2_lcl_opt.lex::1--Rminus-> 3
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: t
YACCO2_TH__::1::o2_lcl_opt.lex:: Finished Popping items from stack
...1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4
                                  ^^^^^^^^^^
  grammar's rule replacing its popped subrule's item 't'
....1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4-- -> 7
                                                    ^
                         a space being pushed
YACCO2_TH__::1::o2_lcl_opt.lex:: Popping items from stack # to pop: 1
....1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4-- -> 7
   YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 7
...1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal::
YACCO2_TH__::1::o2_lcl_opt.lex:: Finished Popping items from stack
....1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4--Rmust_lint-> 8
                                                    ^^^^^^^^^^
  grammar's rule replacing its popped subrule's item: space
```

[5]Exception to this is the start of the parse. What is it past shifted into state? nada or we're in trouble.

```
YACCO2_TH__::1::o2_lcl_opt.lex:: Popping items from stack # to pop: 3
....1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4--Rmust_lint-> 8
   YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 8
...1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: Rmust_lint
...1::o2_lcl_opt.lex::1--Rminus-> 3--Rspec_parm-> 4
   YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 4
..1::o2_lcl_opt.lex::1--Rminus-> 3
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: Rspec_parm
..1::o2_lcl_opt.lex::1--Rminus-> 3
YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 3
.1::o2_lcl_opt.lex::1
   YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: Rminus
YACCO2_TH__::1::o2_lcl_opt.lex:: Finished Popping items from stack
..1::o2_lcl_opt.lex::1--Ro2_lcl_opt-> 1
                        ^^^^^^^^^^^
  Accept grammar's start rule replacing its popped subrule's 3 items
..1::o2_lcl_opt.lex::1--Ro2_lcl_opt-> 1
      successful accepted parse
YACCO2_TH__::1::o2_lcl_opt.lex::accept-parallel-parse
                     current token "/" pos: 3
YACCO2_TH__:: accept tok: option-t tok pos: 0 la tok: / la tok pos: 3
                       1) ^^^^^^^^       2) ^      3) ^          4) ^
     1) returned T in accept queue
                                             ^                        ^
     2 + 4) token stream position relative to 0: gps used start_token,
     3) where restart T is / which is the start of the grammar
              file name:       /yacco2/compiler/grammar/eol.lex
 YACCO2_TH__::1::GPS ACCEPT FILE: yacco2cmd.tmp
                 GPS ACCEPT LINE: 1 CHR POS: 1
 YACCO2_TH__::1::GPS ACCEPT LA FILE: yacco2cmd.tmp
                 GPS LA LINE: 1 CHR POS: 4
 ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  report back the findings in the accept queue by calling thread
  The gps is reported in human terms relative to 1.
YACCO2_TH__::1::o2_lcl_opt.lex:: Popping items from stack # to pop: 1
..1::o2_lcl_opt.lex::1--Ro2_lcl_opt-> 1
YACCO2_TH__::1::o2_lcl_opt.lex::popped state:: 1
.1::o2_lcl_opt.lex::1
YACCO2_TH__::1::o2_lcl_opt.lex::exposed rule/terminal:: Ro2_lcl_opt
YACCO2_TH__::1::o2_lcl_opt.lex:: Finished Popping items from stack
```

Enticements. Are u revved up for other snippets on tracing: 9.4.1? The Error chapter continues exploring this subject by discussing the dynamic tracing variables open to u.

6.5.1 An example would be appreciated and more worth than your k^2 mutters

Here's a grammar extract dealing with hexadecimal digits.

```
Rhex_no_digit  (
lhs{
  op
   Cesc_seq* fsm = (Cesc_seq*) rule_info__.parser__->fsm_tbl__;
   size_t pos = rule_info__.parser__->parse_stack__.top_sub__ - 1;
   CAbs_lr1_sym* s = rule_info__.parser__->get_spec_stack_token(pos);
   fsm->hex_data_ += s->id__;
  ***
}
) {
```

```
  -> 0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9
  -> a -> b -> c -> d -> e -> f
  -> A -> B -> C -> D -> E -> F
}
```

Before i comment on its op code let us look at the schematic of the stack frame for the hexadecimal example from the subrule perspective and its replacing rule perspective before the popping of the stacked symbols take place. The schematic shows what the rule would see.

```
..1::hex_example.lex::1--a-> 2
                         ^
  stack frame 1: pushed 'a' digit on stack associated with State 1
  stack frame 2: ''shifted into'' State 2 with no associated symbol
  In state 2, a reduce operation is detected for this subrule and so the
  reduce_rhs_of_rule() method would create the Rhex_no_digit rule
  containing the reducing subrule.
  At this point the subrule op directive would fire if it exists.
  In this case there is no such subrule directive present.
  Now the the Rhex_no_digit's op directive would fire.
YACCO2_TH__::1::hex_example.lex:: Popping items from stack # to pop: 1
..1::hex_example.lex::1--a-> 2
   YACCO2_TH__::1::hex_example.lex::popped state:: 2
.1::hex_example.lex::1
   YACCO2_TH__::1::hex_example.lex::exposed rule/terminal:: a
YACCO2_TH__::1::hex_example.lex:: Finished Popping items from stack
..1::o2_lcl_opt.lex::1--Rhex_no_digit-> 3
                      ^^^^^^^^^^^^^
  grammar's rule replacing its popped subrule's item 'a'
```

As u can see to get at the pushed digit in the first stacked line above, stack frame 1 has to be retrieved. This is why 1 is subtracted from the current stack frame:

```
   size_t pos = rule_info__.parser__->parse_stack__.top_sub__ - 1;
```

Now the stack frame's symbol can be fetched and played with.

```
   CAbs_lr1_sym* s = rule_info__.parser__->get_spec_stack_token(pos);
   fsm->hex_data_ += s->id__;
```

The stack is the same for both the subrule and rule except that O_2 makes life easier for the subrule by aligning properly all its subrule's stacked symbols by generating the sf variable ready made for u the grammar writer and your subrule's op.

The rule's op directive has full access to the stack frames for its reducing subrule and all its symbols. This is why they are the same. The caveat is the stacked symbols are anonymous and in reverse order: ie the 1st subrule's symbol is deeper in the stack than its last symbol. So the formula to calculate where your item is: stack size minus 1 gives u the last item shifted, stack size minus 2 gives u the 2nd to last shifted item ... [6] If a specific action is required then it is better to program it within the subrule's op directive where the stacked frame contains the cast symbols provided by the sf variable. The rule's op directive is general. All it knows is one of its subrules is being reduced and its rhs symbols is on the stack.

In this example each subrule had 1 symbol shifted: a hex digit. So the shifted symbol's specifics was not important to the rule. If specifics are important, 2 things must be done: have the specific subrule identify itself to the rule that is reducing and secondly using this subrule's identity, fetch and cast the specific stacked symbol needed. From the enumeration scheme, each T class has a range of ids and each T it specific id. Subrules and rules have identities. The subrules enumerate ids are in show order and start from 1 per rule. The rules though are created in show order and built after the T enumeration scheme. Man all this to say that using the stack methods to determine the number on the stack and to walk the stack in top to bottom order allows u to fetch and play with the stack frames.

[6] What about a real epsilon situation? I'll let u figure it out with this clue there is no symbol shifted. Is this a go fish command? Yep just testing your understanding.

6.5.2 The axiom expressed: specificity vs generality

Specifics are more easily dealt with at the subrule level while generality occurs in the rule's `op` part. Once the rule's `op` is done, then the reduce operation removes all the reducing subrule's symbols from the parse stack and sets the shifted rule onto the stack frame and creates another stack frame for the shifted into state[7].

Except for the start stack frame and an epsilonable frame, all other stack frames are twined: state/shift symbol, and "shift into" state. Previously in the book i raised the possibility that a rule could look outside its own context. With access to the parse stack's items, and both T and rule symbol identifications u are now open to grope the parse tables be it for error recovery purposes or some form of context triggers. I have not done much exploration in this regard to error recovery in the sense of backtracking or dynamic parse table modification. This is raised for your own experiments.

6.5.3 Quirks from Parser's point-of-view on symbols and rules

Rules have associated subrules and when being parsed, a specific subrule will have its items on the parse stack. When the subrule has been recognized for a reduce operation, it might need the assigned `Parser` to the grammar. So how does the subrule get at its `Parser`? Well a subrule rule method is defined within its rule definition. In a `fsm` object, the `reduce_rhs_of_rule` method has the subrule's number of symbols to reduce and determines its lhs rule which has the associated `Parser` assigned to it. Here is sample code with comments to explain its actions.

```
void
Cdbl_colon::reduce_rhs_of_rule
    (yacco2::UINT Sub_rule_no
    ,yacco2::Rule_s_reuse_entry** Recycled_rule){
   int reducing_rule = rhs_to_rules_mapping_[Sub_rule_no];
   Per_rule_s_reuse_table* rule_reuse_tbl_ptr =
        fsm_rules_reuse_table.per_rule_s_table_[reducing_rule];
   Rule_s_reuse_entry* re(0);
   find_a_recycled_rule(rule_reuse_tbl_ptr,&re);
   (*Recycled_rule) = re;
   fnd_re: switch (Sub_rule_no){
    case rhs1_Rdbl_colon_:{
      Rdbl_colon* sym;
      if(re->rule_ == 0){
       sym = new Rdbl_colon(parser__); // <== grammar's current parser
       re->rule_ = sym;
      }else{
       sym = (Rdbl_colon*)re->rule_;
       //if ctor present on code sym->reset_ctor();
      }
      (*Recycled_rule)->rule_ = sym;
      sym->rule_info__.rhs_no_of_parms__ = 2; // pop # items for parser
      sym->sr1(); // <=== subrule with op directive
      sym->op();  // <=== its rule also has the op directive
                  // if subrule or rule don't have an op directive
                  // then that method will not be present
      return;
     }
    default: return;
   }
}
```

The above grammar's C++ code has only 1 rule `Rdbl_colon` having 1 subrule. All grammars having more rules and subrules have the same coding pattern. When the rule is created, `reduce_rhs_of_rule` draws from

[7] This is done very efficiently without any new/delete taking place. Can u guess how?. Of course u can the stack array is fixed in size of 256 items. To increase it u must adjust its symbolic size: `MAX_LR_STK_ITEMS` in O_2's library and regenerate it.

its associated `Parser` and sets it in the rule to be reduced to. A recycled rule is local to each grammar instance and already has its associated parser assigned as it is already been created. Being part of the rule's object, the executed subrule's code has access to all of the rule's `CAbs_lr1_sym`'s variables and methods. This is seen in the commented line "pop # items for parser". `sym->rule_info__` prefixes its generic specifics: `parser__` and `rhs_no_of_parms__` To get at its token stream GPS specifics, u cannot **always rely on the stacked items**. Stacked rules don't have these specifics and neither does the meta lrk symbols whose contexts are changing dynamically per run context. So how do u get at these coordinates? U can test or probe the stacked items for T entities or better yet the `Parser`'s maintains contexts like `start_token` which is the 1st T to start the parsing. So why is `Parser` important? Syntax directed code between the 2 contexts `Parser` and grammar demands this telling. From a grammar's perspective it only knows about its Tes, Rules, and `fsm` but it can influence the `Parser` behaviour thru its own `op` directives. When a parser fires up, it receives the grammar to parse with and tells the grammar about itself. Previously described in this chapter was the parser's interplay with the grammar's subrule, rule, and `fsm op` syntax directed code directive.

As i didn't comment much on `rhs_no_of_parms` what utility does it have to u the grammar writer? It is used internally by the parser to remove subrule's items off the parse stack. So far its utility to me has been silent but maybe there are tell tales of own soundings.

Chapter 7

Threads

Man what's all this build up? Sargent Pepper is preening himself, and "the band" tooting while the crowd is uneasy chanting "threads threads show us dhem threads". Before we explore the physiology of threads, let us look at what is really being done by them. Problems can be broken down into smaller problems that eventually get assigned as tasks, scheduled, worked on, and results combined. Now where's the pert? Stop the fluff and blow dry and get on with it. So here are some other terms to focus on: divide and conquer, "top down", multi-tasking, parallelism of activities.

Basically a thread is a basic parsing unit. It itself can divide-and-conquer by calling other parsing units. Here the token stream is being divided into parsing segments that get digested or conquered so to speak by work units. The token stream segments are deterministically calculated and placed inside a context. These contexts are the bottom-up parse tables for each grammar. Now lets look at enhancing this a bit. Why not add competition to the mix where these basic parsing units can be competing at the same time. Ahh non-deterministic competition. That is nice but doesn't it lead to confusion: too many cooks do spoil the broth? You are right of course to raise this potential problem. And this is where arbitration comes into play. Whether its fair or not all drones have their day and some benevolent ruler rules over the outcomes.

So the stick-it is: parsing units can be context sensitive to where and when they express themselves. Another take on rephrasing, ambiguity is "divided and conquered" by separately competing parsing units. As oring gives choice to logic, so to parse streams does subrules and threads give choice.

7.1 Thread's "bi and tri" cepts

Apart from how they get run, is there any differences between monolithic and threaded grammars? Not really. Both have Tes, and Rules, and productions that are exactly the same. They use the same parser to nibble on their input. Their makeup is expressed the same with an added refinement that a threaded grammar can refine its end point in its token segment. What a let down. Man the tooting was revving up me up but this. Come on there's got to be more to it than this.

Before i give u back your money, let's look at the dynamic contexts that this works in. The traditional "lalr" type grammars/compiler/compilers all were monolithic. The grammar is one jumbo clump of symbols housed in different rules and the subrules. I ask u where are the parsing divisions? Silly question mate: THE RULES Dave, the rules. And i continue the probing: how can i recycle these grammatical units in this age of aqua marine?

These leading questions basically end in a conclusion that there is no recycling of grammars as basic work units. It is an all-or-nothing or a rewrite approach to grammars as expressed by compiler/compilers like Yacc. Yes u can extract their rule snippets but be careful of their contextual use and possibly the ambiguity extensions by the compiler/compiler. These 2 x notches should raise your blood pressure and frustrations.

Again why aren't grammars publishable for reuse like algorithms? Grammar snippets if packaged in threaded grammars are recyclable, publishable, and can be tested/used by many. They are basic parsing snippets to some expression. Take C++ type comments. Why not grab a published threaded grammar ready-made and ready-tested for your own use? So do i have your "pro or con" cepts to this thought?

7.2 Refining the boundaries

In section 2.4.2, the `parallel-parser` grammar construct described how a monolithic grammar becomes a threaded grammar. In review, 2 things were required:

1. The thread name supplied by the `parallel-thread-function` directive.

2. The lookahead boundary ending the thread's token segment is supplied by the `parallel-la-boundary directive!thread's end lookahead statement` directive.

Before i get technical, how can u declare these boundaries, and what are they? Let's look at this from a reduce perspective of 1 subrule within the traditional monolithic grammar. In bottom-up parsing the lr(1) algorithm determines when a subrule(s) end of it symbol trail has been reached. All parsing action potentials are stored in the parse table as states. Now the parse has arrived at a state having the potential to reduce. What determines this potential to reduce over a shift? These state actions are divided by their own action sets of Tes. Each member of a set has no commonality within any of the other sets established. With the current token the parser determines what type of parse action might take place: shift, accept, reduce, or error. So the current T discriminates against the three potential actions: shift, reduce or accept dependent on these sets within the current parse state. Outside of these sets is an error.

The lr(1) algorithm builds up the follow set of Tes for the reduce operation to take place. All the following symbol string contexts after the last subrule's symbol are noted and their Tes determined using the first set calculation[1]. This is your typical bottom-up compiler/compiler activity: to deterministically work out the boundary points aka sets of Tes for the various operations and to detect ambiguous situations on the same current T: shift verses reduce, or 2 or more reduces using the same T.

For the threaded grammar what is its follow sets? What Tes end its token segment? Threaded grammars can be activated anywhere along the token stream. Repeat anywhere. Its token segment to be recognized is deterministic but what about its ending boundary? Unless restrictions are placed on it, it is open-ended and contains all the Tes. This is why the `parallel-la-boundary` directive was created. It is a declaration of restrictions. U the grammar writer now become the human lr(1) algorithm to the threads. By being too general, O_2 could nag u with "not a lr(1) grammar" message until u refine the boundary. Later O_2's nattering will be described in their clues to correct the discovered ambiguities.

7.2.1 `parallel-la-boundary`: pieces of eight

A quick review of past descriptions. This directive contains arithmetic type Tes expressions declaring what Tes are included or excluded in the thread's lookahead boundary. The "eol.lex" grammar example gave its reasons why it declared only all-the-Tes-allowed by using only the `eolr` symbol. Use of this symbol can become the starting point to which Tes can be excluded from this boundary by use of the minus sign `-`. Other ways to declare the boundary is to explicitly add Tes to it without `eolr` use. So the arithmetic add sign `+` does this. Here's a simple example: `"/" + "'"`. The forward slash and the single quote are declared as the la bound. Now how would u declare a `+` in the expression as this will interfere with the plus operator? **Just double quote it**. So here is an example removing the 2 operators from the all-symbol: `eolr - "-" - "+"`. This la boundary contains all the Tes except the plus and minus signs.

So this is nice but what if the expression has many terms. How can it be simplified and possibly made more readable? Presto the rule. Egh? Lets look at a rule. What is the "first set" of a rule? It is all the Tes that each of its subrules start with[2]. So why not define a rule whose only raison d'être is to supply its terminal "first set" to the lookahead expression. Let's define such a rule:

```
Rla_minus(){
 -> -
 -> +
}
```

[1] Sure is lite in calories and content. I do not want to be too technical in my explanation. So i ask some pardon from the real connoisseurs of cc.

[2] I intentionally left out epsilonal rules that force the calculation to proceed thru the subrule's rhs. It does not enhance the explanation.

Now we can rewrite the above expression as `eolr - Rla_minus`. Also rules written strictly for la boundary calculation do not have to participate in any parsing contexts. They can be independent of the grammar's defining token stream. That is, they are not part of any rule's subrules symbols that are defining the token stream to be parsed.

Closing remarks about the la expression.
U can plus or minus as much as u want. The result is evaluated to ensure that the lookahead set is not empty. What happens if u did this `eolr - eolr`? U get the picture. The lookahead expression evaluator is forgiving if u did something like this: `eolr + "l" + a`. Here the expression is adding `l` and `a` to a set already containing these letters. U can mix as many Tes and Rules as u like in the expression. The generated document contains this expression and its resultant members within the la set.

7.3 Calling threads Olé olé oé

All threaded grammars are represented in 2 ways: as a thread and as a procedure. This duality is due to an optimization. The thread mechanism can be expensive in allocation set up: address space, registers etc. Also mutual exclusion mechanisms against the calling thread's critical region are taxing. My simple experiment gained me 25% speed improvement with this duality. At thread launch time, how a threaded grammar is called is assessed. If there is only 1 thread that can be called from the current parse state, then it is called as a procedure [3].

Let's look at the different ways a grammar thread can be called from a grammar. There are 3 ways:

- By a grammar's thread call operators.
  - `|||`. Regular way to call a grammar thread.
  - `|t|`. Explicitly called as a procedure and not as a thread.
- Manually called from syntax directed code.
  ```
  Parser::parse_result
  Parser::start_manually_parallel_parsing(UINT Thread_id);
  ```

7.3.1 Thread wheeler dealers: `|||` and `|t|`

Let's review a subrule's "thread call" grammar statement:

- `->` "thread call operator" returned T "called thread"
  The thread call operator can be `|||` or `|t|`. The "returned T" can be specific or one of the following wild T symbols: `|+|`, or `|?|` indicates wildness or an error capturing. The "called thread" is specific or `NULL` used to capture its returned T from a called thread of some brethern subrule.

The `|t|` call has possibly 1 perceived advantage and time will settle this. When used it is an explicit declaration that only one thread is being called. It comes with a precondition though: the thread/procedure's first set must be repeated before the `|t|` call of the calling grammar. I wanted a top/down procedural approach to thread calling with its efficiencies. This was developed before I optimized thread versus procedure call assessment. An example should clarify this: O_2 has a file inclusion operator "@". So the lexical grammar seeing this operator then used the `|t|` operator to digest the following filename. To make it explicit, this called procedure's first set must begin with the "@" character: its grammar's start rule begins with the "@" symbol. When a procedure call is explicit, the `Parser` passes the previous token stream position pointing to the "@" token to the procedure; this position is one back of the current token of the calling grammar. `|||` thread call and possibly an implicit procedure call are called with the current token. Please see section 6.3.4 where this grammar example is given dealing with grammar include file statements.

[3] The current T is checked against each potential thread's first set as to whether it can run. When the result is zero no threads are run and the Parser drops to the traditional state action check of shift, reduce, or accept.

The ||| operator now optimizes the potential to call grammar units according to the number of threads to launch. Using the current token the launch number is dynamicly evaluated against each thread's first set. If the launch number is 1, the procedure call is issued with its efficiencies.

7.3.2 Capturing a returned T

U can specifically program all the returned Tes by individual subrules. If the number of Tes is small this is an appropriate way to do things. But if the number of returned Tes is large and there are unknown Tes like errors, then it would be more appropriate to use a mixed strategy of specifically stated Tes per each subrule and also possibly a subrule using a wild T for the remaining returned Tes. There is no importance placed on what T is assigned to what subrule apart from possibly the syntax direced code gets mapped to that specific T within the stack frame variable `sf`. Outside of this the wild T captured by the subrule can be probed against its identity and a specific programmed action taken. `|+|` captures all returned Tes that are not specifically expressed in other brethren subrule thread calls. Depending on what u are going to do with the returned T it could be legitimate or possibly an error. The `Parser` determines what subrule to reduce after arbitration has selected the returned T. If the specific T is within one of the subrules, then the `Parser` selects it for reduction. If this is not the case then `Parser` tries to select the appropriate subrule to reduce against the wild T. The difference between `|?|` and `|+|` wildness is in how the subrule gets reduced. The `|?|` indicates an error condition and also not to depend on the lookahead to reduce against. U cannot use both wild Tes within called thread expressions within a rule. This truely is an ambigous situation: What wild T takes precedence as they both field the same Tes, and what subrule should it reduce to? O_2 blurts out this condition in that dreaded message "not a lr(1) grammar". The same comment holds for regular subrule expressions within a rule: choose your designated wildness. U can mix the 2 wild Tes within a rule as long as their subrule contexts are different. For example, `|+|` use within thread call expressions and a `|?|` within regular subrule expression used to capture the errant last attempt shift.

If the threaded call was not successful and no T returned, then `Parser` does the same thing against normal grammar expressions: try to shift specifically followed by a wild type shift. The last parse attempted is a reduce operation before a parse error is raised.

7.3.3 Manually u come a calling

Here's an example calling the "eol.lex" thread.

```
Parser::parse_result rtn_code =
      parser__->start_manually_parallel_thread(ITH_eol.thd_id__);
```

Each thread has a globally defined thread record generated by O_2linker using the following naming scheme: The capital `I` is prefixed to the thread name. Thread ids are assigned in lexicographical name order by O_2linker. The thread entry structure is defined as:

```
struct Thread_entry {
  yacco2::KCHARP           thread_fnct_name__;
  yacco2::Type_pp_fnct_ptr thread_fnct_ptr__;
  yacco2::USINT            thd_id__;
  yacco2::Type_pc_fnct_ptr proc_thread_fnct_ptr__;
};
```

The `thd_id__` variable is the lexigraphical value assigned to the thread. Each threaded grammar is maintained in an internal launch thread table where each launched thread is optimized for reuse. Threads are never killed but recycled into a "thread availability" pool.

The returned code `rtn_code` above indicates whether the call was successful or not. The `parse_result` enumerate is defined within `Parser` as:

```
  enum parse_result{erred
                   ,accepted
                   ,reduced
                   ,paralleled
                   ,no_thds_to_run
                   };
```

`Parser::erred` and `Parser::accepted` are thread call outcomes. The other items are used internally by `Parser` dealing with specific parsing operations.

Here is a real example taken from the `angled_string.lex` grammar. Rule `Ropen_angle` contains its own finite state automaton for speed sake. I'm repeating this example from the `Parser` chapter to minimize page flipping and its generated impatience. The code is very simple acting on specific Tes to build up the escaped character sequence returned from the `TH_esc_seq` thread. Notice this is the same thread that is called automatically by `Parser` when thread calls are detected.

```
Ropen_angle  (){
  -> "<" {
    op
      Cangled_string* fsm =
         (Cangled_string*) rule_info__.parser__->fsm_tbl__;
loop:
      switch (rule_info__.parser__->current_token()->enumerated_id__){
       case T_Enum::T_raw_lf_: goto overrun;
       case T_Enum::T_raw_cr_: goto overrun;
       case T_Enum::T_LR1_eog_: goto overrun;
       case T_Enum::T_raw_gt_than_: goto closestr;
       case T_Enum::T_raw_back_slash_: goto escseq;
       default: goto other;
      }
    closestr:{ // end of string
     rule_info__.parser__->get_next_token();
     return;// end of angled string
    }
    overrun:{
      CAbs_lr1_sym* sym = new Err_bad_eos;
      sym->set_rc(*rule_info__.parser__->start_token__
                  ,__FILE__
                  ,__LINE__);
      RSVP(sym);
      rule_info__.parser__->set_stop_parse(true);
      return;
    }
    escseq:{ // what type of escape
      using namespace NS_esc_seq;
      Parser::parse_result result = // MANUALLY CALL THE THREAD
         rule_info__.parser__
            ->start_manually_parallel_parsing(ITH_esc_seq.thd_id__);
      if(result == Parser::erred){  // check for error
          // in this case, it will not happen: here to educate
          rule_info__.parser__->set_abort_parse(true);
          return;
      }
      // Process returned token
      Caccept_parse& accept_parm =
          *rule_info__.parser__->arbitrated_token__;// extract rtned T
      CAbs_lr1_sym* rtn_tok = accept_parm.accept_token__;
      int id = rtn_tok->enumerated_id__;
      accept_parm.accept_token__ = 0;
      if(id != T_Enum::T_T_esc_seq_) {
        RSVP(rtn_tok); // pass back error token and finished fsa parsing
        return;
      }
      T_esc_seq* finc = (T_esc_seq*)(rtn_tok);
      fsm->copy_str_into_buffer(finc->esc_data());
      rule_info__.parser__->//re-align token stream and continue parsing
            override_current_token(*accept_parm.la_token__
```

```
                                ,accept_parm.la_token_pos__);
      delete finc; // recycle back the returned T
      goto loop;
    };
    other:{
      fsm->copy_kstr_into_buffer(rule_info__.parser__->
                                 current_token()->id__);
      rule_info__.parser__->get_next_token();
      goto loop;
    }
    ***
    }
}
```

What is important is how the syntax directed code extracts the returned `Caccept_parse` parameter `arbitrated_token__`, and realignment of the token stream for self to continue parsing by using the `override_current_token` function. This alignment must take place as the called thread has consumed some of the current grammar's token stream. Here is the `Caccept_parse` definition giving its called assessment: returned T and where it began within the parsed token stream, and the lookahead boundary: current token to continue parsing with and where it is within the token stream. It is not important as to how `Caccept_parse` gets created but how to re-alignment the token stream with its returned data. In my experience manually calling a grammar thread is rare but now u know how to.

```
struct Caccept_parse{
   Caccept_parse
           (yacco2::Parser&       Th_reporting_success@/
           ,yacco2::CAbs_lr1_sym& Accept_token@/
           ,yacco2::UINT          Accept_token_pos@/
           ,yacco2::CAbs_lr1_sym& La_token@/
           ,yacco2::UINT          La_token_pos);
   Caccept_parse();
   void initialize_it();
   void fill_it(Caccept_parse& Accept_parse);
   void fill_it(yacco2::Parser&   Th_reporting_success@/
           ,yacco2::CAbs_lr1_sym& Accept_token@/
           ,yacco2::UINT          Accept_token_pos@/
           ,yacco2::CAbs_lr1_sym& La_token@/
           ,yacco2::UINT          La_token_pos);

   ~Caccept_parse();
   yacco2::Parser*       th_reporting_success__;
   /**
   * Re-alignment variables
   **/
   yacco2::CAbs_lr1_sym* accept_token__;
   yacco2::UINT          accept_token_pos__;
   yacco2::CAbs_lr1_sym* la_token__;
   yacco2::UINT          la_token_pos__;
};
```

7.3.4 Manually producing an error

The above example shows where and how to detect an error from the called thread but did not generate and return an error T as it was there for teaching purposes. Here is an example of such code that receives a manually called thread error where its error T is passed as an accept token. It then posts it back to its calling grammar inside its error queue, and then stops its parsing activity:

```
  if(result == Parser::erred){
    Caccept_parse& accept_parm = *parser()->arbitrated_token__;
    CAbs_lr1_sym* rtn_tok = accept_parm.accept_token__;
```

```
    accept_parm.accept_token__ = 0;
    ADD_TOKEN_TO_ERROR_QUEUE(*rtn_tok);
    parser()->set_abort_parse(true);
    return;
  }
```

U are also open to thrown away the returned error and generate your own error T. What is important is what u want to do: abort the parse signaling back to a caller grammar something, continue with your own error corrections etc. Your liberties are your own inventiveness.

7.4 Parallel parsing and arbitration

What does this section's title mean — parallel parsing? From what we've just seen, multiple thread expressions are allowed within a rule's subrules or spread across multiple rules brought together by the lr(1)'s closure operation. If the current terminal to parse is in one or more of the potential called threads's first sets, then only those specific threads are run. Now consider when more then 1 thread can be launched. This is the parallel parsing context: more then 1 thread running at the same time while the launching grammar waits for the answer(s). It is the act of determining potential outcomes. U probably are asking why such a nondeterministic situation would occur and of what value does it have to u the grammar writer? In C++ how can u determine whether the expression is a variable expression or a function call? Within a normal grammar expression u can't express this without having ambiguity[4].

Thread expressions divide the token stream into separate parsing context / environments via the thread model. It delays the boundary determination of what type of expression will be recognized by allowing each parse thread to run within its own token boundary expression. This is the power of the bottomup parsing approach extended by parallel potential outcomes: It delays the determination of what can be recognized without the ambiguous contraints of a singlular grammar context. Now mix this thought into parallel parsing: Each launched thread can launch their own parsing threads: aka nested parallel parsing. The launching grammar waits for its result while its threading children are dancing the parse jig. From ahigh, the first grammar cascades its threads and becomes the last grammar to receive its threaded results. Topdown parsing uses the recursive decent procedure calls while O_2's uses parallel parsing by threads to give more contextual expressivness and grammar writer freedoms. There is no interference between the potentially competing threads. Each thread is its own firewall to parsing mayhem. This model allows the grammar writer to write smaller self contained expressions. Do not think threads only as a ambiguity breakout but also more freedom to explore different parsing terrains.

7.4.1 A Mathematical perspective on Parallel parsing

From a mathematical perspective threading allows one to run parallel subset-superset environments without any barriors. Here is a made-up example: one thread recognizes keywords only while the other thread recognizes the general tokens — variables of which could be function names, keywords, etc. Just to note this situation is not the most efficient way to do things but illustrates the power of the threading model used to parse this ambiguous situation. The beauty of this approach is threads break up the token stream into smaller grammatical parts, simplifyies parsing expressions, and keeps conflicting contexts contained within each thread. Thus they allow u to experiment and to explore new language paradyns that monolithic grammars labour under.

7.4.2 Ar-bi-tra-shion

Now to address the last part of this section's title: Arbitration. Why are u reviewing this as section 5.2.2 "The rules parameters: arbitrations many takes" already described the directive. That section described the Arbitration framework with examples. Please have a reread to refamiliarize yourself with Arbitration's fit

[4]The way some grammars do it is express a superset of what to recognize and post-process the parsed expression for kosherness.

within the grammar Rule. Here I'll be refining its usefulness, commenting on its syntax-directed code, and post parsing continuence after a thread call.

The majority of "thread use" launches just 1 grammar unit aka a thread to parse some expression. So arbitration is not required though the arbitration function is still present in the grammar's emitted tables but optimized for singularness. O_2 emits arbitration functions with unique names whenever a thread expression in a subrule is present. An individual grammar can have more than 1 arbitration function which are context sensitive to the specific lr(1) state deploying a thread expression. Collectively each grammar's arbitration units are self contained within their own grammar namespaces.

The following comments are on arbitrating idiosyncracies. Arbitration comes about when there are multiple competing threads that have returned results back to the calling grammar thru its "Accept queue". Now the calling grammar regulates the potential of multiple outcomes. Results come in T form from either the Error or normal T vocabulary. Yes Errors are allowed as returned Tes. Now the grammar determines what result to keep while throwing out the balance of unwanted potential results. After this the grammar realigns its token stream position and forwards the arbitrated result back to the parser to continue. Arbitration is a post-assessment by its own syntax directed code before it relinqishes controll back to the parser that finds the subrule matching the returned T to shift on that could execute its own **op** directive if present.

So how to arbitrate on returned terminals in the calling grammar's "Accept queue"? The first thing to note is items within the "Accept queue" are random in order due to thread management dictates as to when it allows each called thread access to plop their booty into it. For the moment the "Accept queue" is a queue without any other enhancements like a hash table. Due to this and input randomness, it could require multiple read passes on the queue to determine what Tes are competing for existence. Your arbitration code is injected between the precanned "Accept queue" traversal loop of the manufactured arbitration function. It could be blocked code, conditional blocked code containing your own traversal loops, to functional calls. This loop sandwich comes with an exit label "arbitrated_parameter" for your code to exit to when the arbitrated result has been selected. The precanned code following the exit label sets up the selected T to continue parsing on, empties the "Accept queue" of all Tes while destroying the unsuccessfull Tes in the queue. In the "library" folder u will find the Arbitration traversal code "war_begin_code.h" and "war_end_code.h" files that are the pieces of bread surrounding your injected code [5].

Arbiration code placement?

Whose rule's arbritration code is used for the arbitration? For now the code must be associated with a rule having a direct threading subrule statement. What? Rules are brought into a state by the closure operation. Currently O_2 searches the state's productions having a **direct thread operation** to potentially perform. If arbitration code is associated with one of the **direct thread operation** production's parent rule, O_2 generates the arbitration procedure name with the prefix "AR_xxx" where xxx is the found rule name. This procedure's address is associated within the state number's lr table.

The error reporting for the arbitration code emittor is zero. What happens when there are multilpe arbitration code snippets from different closured rules? It takes the last found rule's code to generate the arbitration procedure. If a rule has arbitration code that indirectly spawns threads, does its arbritration code get used? Nada.

For now, there is no code generated when the rule's productions do not directly spawn threads even though it uses the closure operation to indirectly include threads by these closured rules. So be warned dear reader[6]. The arbitration procedure name(s) has been added to lr states network of the grammar's document. If a state is threading as `|||` is its "vectored into symbol", arbitration is flagged by either the procedure name used in form of **AR_xxx** where xxx is the rule name where the abritration code was defined, or ϵ indicating no arbitration defined or found.

A mild warning, read your grammar's gened document and check / verify the "Lr1 State Network" section

[5] Oh no says u the reader this is not too efficient using arbitration! When there are less then 10 items in a queue, list processing is efficient and it leads to the question whether "Should u have that many sucessfully competing threads launched?". Depends on what u are doing. Context sensitivity with potential outlooks definitely will lead u into a different wonderland! Experiment and enjoy the voyage.

[6] Yes a minor inconvenience until the next set of O_2 improvements occur. False promises?

and its threading states for O_2's arbitration code handling [7].

Please read **/usr/local/yacco2/docs/test_components.pdf** grammar giving a "arbitration-code" example receiving different lexical Tes tested by the "/usr/local/yacco2/qa/*.dat" files. Testdriver's document **/usr/local/yacco2/docs/testdriver.pdf** exercises/verifyies the lexical grammars. Pay attention to **YACCO2_AR__** use in verifying arbitration assessment. **/usr/local/yacco2/qa/o2testdriver_tests.sh** is the bash script exercising *testdriver*.

7.4.3 Now for an explained example

Let's take the subset / superset situation where the variables are made up of 1 or more letters a–z represented as variable-T, and the keywords are "a" and "b" represented as "a-kw" and "b-kw" Tes. We'll give the thread names to be called variable and keyword. So calling these 2 threads would have a potential outcome of: only a variable-T returned, or the returned variable-T along with a keyword: "a-kw" or "b-kw". So the number of items in the queue are either 1 or 2. When there are 2 items, their returned order is unknown: variable-T, (a-kw or b-kw) or the reverse order. Knowing there are 2 items, the arbitration code would search for the T returned that is not a variable-T. Before the Arbitration function is called, the Parser optimizes whether to execute it by checking if there are more than 1 item in the "Accept queue". With only one item the **arbitration function call is bypassed** and the 1 item extracted from the "Accept queue" becomes the current T to continue parsing with. Use of the T vocabulary's enumeration definition NS_yacco2_T_enum::T_Enum::xxx in the example below allows one to distinguish the Tes in the "Accept queue". Section 3.5 "*T-enumeration* and counting" explains the enumeration scheme of the Terminal vocabulary for your own project. The namespace will be different than NS_yacco2_T_enum. Here is the madeup calling grammar's arbitration code whereby the grammar containing it is itself a threaded grammar called by another grammar.

```
Rsym_def1  (
parallel-control-monitor
{
  arbitrator-code // directive
    using namespace NS_yacco2_T_enum;
    // i, and ie are wired into the emitted code along
    // with arbitrated_parameter label
    for(i=1;i<=ie;++i){
      if(Caller_pp->pp_accept_queue__[i].accept_token__->enumerated_id__
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             // accessing the current accept queues's returned T
          != NS_yacco2_T_enum::T_Enum::T_variable_T_){
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
             //testing against enumeration vocabulary
        goto arbitrated_parameter;// hardwired label to break out of loop
      }
    }
  *** // ending the directive
}
){
  ->  ||| "variable-T" NS_variable::TH_variable {
    op
        RSVP(r);
        rule_info__.parser__->set_stop_parse(true);
     ***
    }
  ->  ||| "a-kw" NS_keyword::TH_keyword {
    op
        RSVP(r);
```

[7] Due to some C++ statements like "type casting", *cweave* could emit some unstructured code sections. From the website: `http://www-cs-faculty.stanford.edu/~uno/cweb.html`, here is the extracted comments on this situation: *"The authors do not intend to change CWEB henceforth unless some devastating new bug is discovered. Non-catastrophic infelicities should therefore be considered permanent features of CWEB."*. I raise a challenge to the "open-source community and computer science departments, why not help in improving such an excellent product!

```
        rule_info__.parser__->set_stop_parse(true);
     ***
    }
  ->  ||| "b-kw" NULL {
    op
        RSVP(r);
        rule_info__.parser__->set_stop_parse(true);
     ***
    }
  ->  ||| "|?|" NULL { // capture possible Error tes
    op
        RSVP(r);
        rule_info__.parser__->set_stop_parse(true);
     ***
    }
}
```

Once the Arbitration has taken place, the Parser continues to run selecting the approprate subrule having the arbitrated T[8]. The Parser then will execute its associated syntax-directed code. In the above example u can see the Parser executing the grammar is terminated by the last 2 statements: RSVP places a T inside its calling grammar's "Accept queue" and then indicates to the Parser to stop parsing by calling the set_stop_parse method. U can do whatever u want. This is not a generic way of parallel parsing but just an illustration. The above example could have built a tree, add to a symbol table while continuing to parse within the grammar. Eventually though if your grammar is a thread it must return some result back to its calling grammar or just abort. If the Parser of the calling grammar receives that the attempted threading failed, it continues by falling through to normal bottomup parsing. It tries its hand to continue by using the current T against one of the parse operations: a shift, reduce, accept, or an error.

U can be critical towards threading and its inefficencies measured against a procedural call. But time will build better computers to improve threading and messaging approaches to processing. My goal with O_2 is to simplify parsing and extend it using nondeterminism to explore other language constructs. Is this an attempt to tame a shrew or just shrewing?

7.5 Introduction into O_2 ugliness: Please check...

U just received that unexpected ugly message on your console from O_2. "Not a lr(1) grammar!". It came about when trying to generate the parse states for the threaded grammar "eol.lex". From what we have discussed on ambiguity, somewhere within your grammar there are 2 competing subrules where one of them is reducing and there is a common T that both want to use. I'll lightly discuss this situation before the chapter on "Tracing" explores it fully. I see your perplexed eyebrows wondering why this intro. Mine too when i reread this part. It is here cuz of the thread's `parallel-la-boundary` expression.

This common T is in a reduce set while the other subrule could be shifting or it could also be reducing. Let's look at this shift/reduce problem from the threaded grammar perspective and its lookahead expression. The following is a dump for the errant "eol.lex" grammar without the `|.|` symbol used to shift out of the ambiguity. Basicly the dump contains all the states gened so far and their anatomy. It directed us to "Not LR1 — check state conflict list of state: 1 for details". Here is the output described:

```
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_includes.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_k_symbols.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_characters.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_terminals.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_err_symbols.T
```

[8] If both threads could return Error Tes, what do u do to capture them without using double subrules using the wild Tes? If U require to discriminate between the Errors from each thread, there is not too much programming leeway but to specifically define subrule(s) from the choosen thread's Errors while using a wild T to capture the other thread's Errors. U could also have a subrule per Error T. When going the "list all Error subrules" route, when u refine the grammar remember that specifically listing the Errors leaves u open to missed new Error Tes. This is one of the benefits of wildness to capture the future returns.

```
Enumerate T alphabet
Total lrk symbols: 8
Start lrk symbol: 0 Stop lrk symbol: 7
Total rc symbols: 256
Start rc symbol: 8 Stop rc symbol: 263
Total T symbols: 114
Start T symbol: 264 Stop T symbol: 377
Total error symbols: 191
Start error symbol: 378 Stop error symbol: 568
Total symbols: 569
Enumerate Rule alphabet
Dump of P3 tokens
1:: #fsm file no: 2 line no: 32 pos: 1
2:: #parallel-parser file no: 2 line no: 37 pos: 1
3:: #T-enumeration file no: 4 line no: 32 pos: 1
4:: #lr1-constant-symbols file no: 5 line no: 48 pos: 1
5:: #raw-characters file no: 6 line no: 13 pos: 1
6:: #terminals file no: 7 line no: 15 pos: 1
7:: #error-symbols file no: 8 line no: 39 pos: 1
8:: #rules file no: 2 line no: 51 pos: 1
O2 version: .669 Date: July/2009
===>Please check Grammar dump: eol_tracings.log for Not LR1 details
Not LR1 --- check state conflict list of state: 1 for details
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

State dump
->State: 1 Entry: -1 Symbol:  No symbol
 Follow set:
  Rule no: 569 Name: Reol
  follow set:
    eolr
  Rule no: 570 Name: Rdelimiters
  transitions
    S1xReol
 Vectors:
  Symbol no: 18 Symbol: x0a
   RxSRxPos: 2.1.1 x0a Closure S1 CS-gen S1 goto S2 reduced S2
  Symbol no: 21 Symbol: x0d
   RxSRxPos: 2.2.1 x0d Closure S1 CS-gen S1 goto S3 reduced S3
   RxSRxPos: 2.3.1 x0d Closure S1 CS-gen S1 goto S3 reduced S4
  Symbol no: 570 Symbol: Rdelimiters
   RxSRxPos: 1.1.1 Rdelimiters Closure S1 CS-gen S1 goto S5 reduced S5
 ===>Conflict states:
    State no: 3
    ^^^^^^^^^^^
->State: 2 Entry: 18 Symbol: x0a Birthing closure state: 1
 Follow set:
 Vectors:
  Symbol no: -369 Symbol: eosubrule
   reduce set #: Empty see following transition
   RxSRxPos: 2.1.2 Eos of Rdelimiters Closure S1 CS-gen S1 reduced S2
->State: 3 Entry: 21 Symbol: x0d Birthing closure state: 1
 Follow set:
 Vectors:
  Symbol no: -369 Symbol: eosubrule
   reduce set #: -1
     eolr
   RxSRxPos: 2.2.2 Eos of Rdelimiters Closure S1 CS-gen S1 reduced S3
  Symbol no: 18 Symbol: x0a
```

```
   RxSRxPos: 2.3.2 x0a Closure S1 CS-gen S1 goto S4 reduced S4
->State: 4 Entry: 18 Symbol: x0a Birthing closure state: 1
 Follow set:
 Vectors:
  Symbol no: -369 Symbol: eosubrule
   reduce set #: Empty see following transition
   RxSRxPos: 2.3.3 Eos of Rdelimiters Closure S1 CS-gen S1 reduced S4
->State: 5 Entry: 570 Symbol: Rdelimiters Birthing closure state: 1
 Follow set:
 Vectors:
  Symbol no: -369 Symbol: eosubrule
   reduce set #: Empty see following transition
   RxSRxPos: 1.1.2 Eos of Reol Closure S1 CS-gen S1 reduced S5

Common States dump
1::Common State: -1 Entry Symbol: No symbol
state no: 1
2::Common State: 18 Entry Symbol: x0a
state no: 2
state no: 4
3::Common State: 21 Entry Symbol: x0d
state no: 3
4::Common State: 570 Entry Symbol: Rdelimiters
state no: 5
```

State 1's `===>Conflict states:` lists State 3 that has the problem. Looking at State 3 in the dump, its makeup comes under its heading "Vectors:". Here it sequentially lists all its members that its dealing with. The first entry has Symbol no: -369 Symbol: eosubrule. Anything with a negative number is a special condition where in this case it means "end-of-subrule-reached" indicated by its literal name "eosubrule". All the other vectors like T or Rule are positive and their literal identifier expressed. Common members are sequentially listed together. As it is a reducing subrule, following it is its calculated reduce set with its full contents displayed: `eolr`. It has for the moment a negative number. This is not important for this discussion just to say that there is an optimization on reduce sets to commonize them where their set numbers will be assigned from 1 when the grammar is lr(1) to the good. Its contents is the all-symbol `eolr` which was expressed in the `parallel-la-boundary` directive. Lets continue looking at the other subrules in the state.

```
Symbol no:  18 Symbol:  x0a
```

This is a shift operation; so there is a reduce/shift conflict. But why? Remember the discussion using `eolr` to minimize the number of symbols in the lookahead set. Here the all-symbol represents itself and all other symbols. So the cause of the conflict has been determined. But how to correct it? These "out damn spots" can be played with by adding the `|.|` symbol to your conflicting subrules or to break the ambiguous situation into competing threads. But this could be a fast fix solution allowing u to at least compile the grammar and run it. To really view the parsing contexts in real-time-slow-motion by trace-tracking, you will have to go to the "Errors" chapter 9.4.1. This allows u to view parsing contexts logs that can be studied, and to refine your understanding on the conflicting contexts. With this post assessment the proper way to correct the ambiguity can be programmed.

Details on how a reducing subrule's followset (lookahead set)

```
reduce set #:  -1
eolr
```

is arrived at are given thru potential transitive lists of form: SxR where S is state and R the rule. It is a transitive chain of contexts that give their contents to the originating rule's follow set. Each reducing subrule identifies where its rule's follow set is within its closure state (where the rule and its subrules were born). S3's reducing subrule above has this declaration:

```
RxSRxPos:  2.2.2 Eos Rdelimiters Closure S1 CS-gen S1 reduced S3
```

The `RxSRxPos 2.2.2` expresses that the 2nd rule `Rdelimiters`'s 2nd subrule whose symbol position 2 which

is its end-of-the-subrule's string of symbols has been reached indicated by `Eos`. This means it is reducing and it was born in closure state `Closure S1`. In state `S1` under "Follow set:" is an entry for rule `Rdelimiters` whose follow set is empty. But it inherits (transitions) to `S1xReol` whose follow set contains `eolr`. There are other details that for now are not important. Just to say a follow set is made from the union of all other contributing follow sets.

7.6 Summary

Well u can now write your own grammars. Both monolithic and threads are familiar entities to u. How u divide and conquer your T stream is of your own making. Acquitting yourself from error situations detected and passed along, parsing stoppage, and roll-your-own internal parsing reviewed. I've tried to give u a view on grammatical freedoms of course this is all open to your own takes.

Glimpses of frustrations and grammatical lr(1) squawking was litely discussed to pique your curiosity to venture into those future chapters to refine your techniques. Now its on to the promised containerland where u Dave have bravoed their qualities. Well lets hope its worth your patience and experiences dear reader.

Chapter 8

Containers

As i'm the only one talking, i hope u are enjoying my attempt(s) at describing how to use O_2 with humour. So far grammars have been described: `terminals` Tes for short, `rules`, `subrules` aka productions and `fsm`. The thread extension developed and its control over its lookahead boundary explained.

Now the containers descriptions will round out your basic understanding of O_2. Yes that's in the plural. There is more to it then just the parsing contexts of input, output, error, and recycling with a fixed type of container. The basic container starts with the raw characters that get digested and assembled into terminals of your own making. My wish-list grew as i was developing O_2. Just having one container type did not cut-the-candy. There were different parsing contexts that demanded more flexibility like rule/subsule structuring. I'll hold this thought to tease u.

So what is needed for terminal containers? The minimum is a list or supply of abstract terminals. Why in the abstract? O_2 is open-ended to Tes but it has no clue on their futures nor their content. Though containers are all sequential in nature, when my threading thoughts coalesced direct access became a requirement. From different run contexts this provided more liberty across ribbons of terminal segments. For example, potential future lookaheads, to past querying for tracing, logging, started showing up in my development. Then dynamic conditional parsing contexts came along. What do u mean by conditional parsing contexts? Let some dynamically created text [1] within a grammar's subrule to be parsed. This could be a string that was already parsed as a literal but whose contents takes on another parsing context. So the conditional could be within the syntax directed code of the grammar that requires another parsing to take place. And then there are trees. What about them? Wouldn't it be nice to have a container that traverses a tree in some manner delivering its nodal content? And what about applying some form of filtering mechanism on the terminals wanted or to be excluded such that the branching structure becomes flattened dishing up sequential Tes belonging to a subset of the tree's content?

The problem i was faced with was to make all these containers act the same: to support sequential and random access without disturbing the `Parser`. The `Parser` is generic and it should not know how its input is being delivered. The container definitions should be open-ended for other future variants without any interference to the `Parser`. I've only expressed the parse perspective but containers should be accessible in their own right from outside this context. One should not be concerned with the delivered Tes within a restrictive context like the grammar and its `Parser`. Just iterate thru the container and get me them Tes as the marketers say: just do it.

8.1 RC jetsome

Raw characters normally come from a file. So there is a standard container that handles opening the file to dishing out its contents. For the first cut, ASCII 8 bit only characters are supported [2]. Raw characters have a short life as they are the ingredients of other terminals. That is segments of raw characters are snippets to

[1] How and where the text comes from could be internally constructed from your own compiler. Mumblings or conversations with oneself with responses?

[2] The reason was to flesh out my ideas on multi-threaded parsing first before venturing into the Unicode world with different operating systems having their different takes on Unicode support and their C++ compilers.

terminals being built. One decision i made was to not dynamically "new" a RC terminal from the memory heap as it was from initial experimentation too expensive in heap allotment and in the parser's running time. Remember there are lots and lots of raw characters to be parsed.

O_2 stores the raw character terminals in a fixed memory pool. Each symbolic raw character has a reservation in the memory pool of 1 to 1 mapping. This saves heap trashing at the expense of what? At the expense of memorizing its GPS: associated file, line number and character position. The raw character GPS is a just-in-time assignment. When the raw character gets assigned to its rc terminal, its GPS is dynamically associated within the file. Reuse of the rc terminal overrides its past GPS association with its present marking. Normally raw characters get requested by a parser under some grammar's supervision where fetched characters get built up into other entities like reserved words, strings, numbers, or variable definitions. When the rc terminal's GPS is needed, this requires immediate reference to the raw character's GPS before it gets remapped to another position within the file. As described before, the GPS is used in tracing, error reporting, and other mundanities. Usually the associated GPS with the newly created terminal being built from raw characters uses its starting raw character. The GPSs of the other characters making up the terminal can be ignored. Meta terminals (terminals built from other terminals) do not have this GPS-first-use-copy requirement as they are newed from the memory heap and have permanent GPS memory. The parsing grammar automatically stores the starting terminal's GPS for such copying. This is most evident when grammar threads are used and their created terminals need to tag its GPS. [3]

This approach has not hurt the parsers written in GPS reporting so far and it certainly improves O_2's performance by 25%. So u've been cautioned.

8.2 2 solitudes: doing the walk

This first example gives u the taste of using a tree container and directly walking it in prefix order. It is taken from the "la_expr.lex" grammar that parses the thread's lookahead expression. The parent node is the rule and its subrules children are fetched (breadth-only). The filter mechanism works against the T's id which in this case is `NS_yacco2_T_enum::T_Enum::T_T_subrule_def_`. From there the subrule's elements are walked looking for specific T types to add to another container.

```
void Cla_expr::add_rule_to_set
         (NS_yacco2_terminals::rule_def* Rule
         ,T_IN_STBL_SET_type* Set_to_add_to
         ,std::set<int>* Rules_already_processed){
  using namespace NS_yacco2_terminals;
  int r_id = Rule->enum_id();
  if(Rules_already_processed->find(r_id)
           != Rules_already_processed->end()) return;
  Rules_already_processed->insert(r_id);
  AST* r_t = Rule->rule_s_tree(); // rule tree node to traverse
  set<yacco2::INT> elem_filter;   // Filter: set of Tes ids
  /*
   * load up filter with T ids
   */
  elem_filter.insert(NS_yacco2_T_enum::T_Enum::T_T_subrule_def_);

  tok_can_ast_functor ast_functor;// generic accept all Tes given
  /*
   * Walker receives its tree node to walk, functor, T filter and type
   */
  ast_prefix_wbreadth_only       // tree walker
         pwbo(*r_t
         ,&ast_functor
```

[3] I know u remembered this. It is noted for my memory jogging: When the Parser starts up, it stores the first T fetched with its GPS. It is accessible thru its *start_token()* function. Use of this returned T allows u to get at its GPS for various uses like creating new Tes of Error or Meta-T class. If u need the GPS of a T that follows the **start_token()** then u must immediately fetch its GPS within the syntax-directed-code of the grammar. I leave this question: what GPS is returned when a specific fetching of a T from a container is returned? Experiment!

```
        ,&elem_filter
        ,ACCEPT_FILTER);
  tok_can<AST*> tok_can(pwbo);   // container of tree Tes

  /*
   * iterate container using its random access operator[]: 0..n
   * The eog symbol is end-of-container reached which is
   * end-of-tree.
   */
  CAbs_lr1_sym* s(0); // the abstract sym supplied by container
  int x(0); // T number to fetch starting from zero
  for (;(s=tok_can.operator[](x))!= yacco2::PTR_LR1_eog__;++x){
    T_subrule_def* sr_def = (T_subrule_def*)s;// cast
    vector<CAbs_lr1_sym*>* elems_list = sr_def->subrule_elems();
    CAbs_lr1_sym* sym = (*elems_list)[0];
    int id = sym->enumerated_id__;
    switch(id){ // discriminate on T's id
      case NS_yacco2_T_enum::T_Enum::T_T_eosubrule_: break;
      case NS_yacco2_T_enum::T_Enum::T_refered_T_: {
        refered_T* rt =
             (refered_T*)sym; // abracadabra: a referenced T
        add_element_to_set(rt->t_in_stbl(),Set_to_add_to);
        break;
        }
      case NS_yacco2_T_enum::T_Enum::T_refered_rule_: {
        refered_rule* rt =
             (refered_rule*)sym; // abracadabra: a referenced rule
        rule_def* r = rt->Rule_in_stbl()->r_def();
        add_rule_to_set(r,Set_to_add_to,Rules_already_processed);
        break;
        }
      default:{
        break;
        }
    }
  }
}
```

The container's details will be described later. The C++comments should orientate u to its doings.

8.2.1 A grammar's stumblings

The 2nd example below shows a parse inside a grammar's syntax directed code using a tree to be walked. I'm reenforcing tree use as u'll see that they are very versatile and become more prominent in post lex/syntax semantics. It uses the monolithic "prt_sr_elements.lex" grammar as a logic automata whereby each rule/subrules relationships have appropriate syntax directed code directing the output to the `prt_elems_fsm.ow_index_file_ = &fsm->ow_index_file_;`.

8.2.2 Contaminates

A filter is supplied to accept a subset of the Terminal vocabulary by enumerated id. It is declared as `set<int> prt_sr_elems_filter_;` within the `fsm` construct of the grammar. This is a set of integers filled by the grammar's `op` directive as follows:

```
op
  prt_sr_elems_filter_.insert(T_Enum::T_T_subrule_def_);
  prt_sr_elems_filter_.insert(T_Enum::T_refered_T_);
  prt_sr_elems_filter_.insert(T_Enum::T_T_eosubrule_);
  prt_sr_elems_filter_.insert(T_Enum::T_refered_rule_);
```

```
    prt_sr_elems_filter_.insert(T_Enum::T_T_called_thread_eosubrule_);
    prt_sr_elems_filter_.insert(T_Enum::T_T_null_call_thread_eosubrule_);
                ... elided code
  ***
```

It allows one to accept or bypass the Tes when a tree's node is read. What type of filter it is: **accept** or **bypass** is supplied to the tree walker by O_2's constants **ACCEPT_FILTER** which is "boolean true" or **BYPASS_FILTER** which is false.

8.2.3 Patting the head while rubbing the stomach and limboing

Here's the parse inside a grammar's syntax directed code using the above tree filter:

```
Rprt_xrefs_docs(){
  -> Rrules eog {
    Cprt_xrefs_docs* fsm =
          (Cprt_xrefs_docs*)rule_info__.parser__->fsm_tbl__;
                              ... elided code
    // print its rhs elements
    AST* sr_t = srd->subrule_s_tree(); // rule's subrules tree

    /*
     * tree walker where it receives the tree node to walk.
     * In this example a generic functor ``tok_can_ast_functor''
     * accepts any T given to it once passed thru the filter.
     */
    tok_can_ast_functor sr_elems_walk_functr;// functor: accept all Tes
    ast_prefix_1forest                                  // tree walker
          prt_sr_elems_walk(*sr_t                       // tree node
                            ,&sr_elems_walk_functr
                            ,&fsm->prt_sr_elems_filter_
                            ,ACCEPT_FILTER);
    tok_can<AST*> prt_sr_elems_can(prt_sr_elems_walk);// tree container
    using namespace NS_prt_sr_elements;
    Cprt_sr_elements prt_elems_fsm;
    prt_elems_fsm.ow_index_file_ =
          &fsm->ow_index_file_; // output file from current grammar
    Parser prt_elems(prt_elems_fsm     // grammar to parse with
                    ,&prt_sr_elems_can // input container
                    ,0);               // no output container
    prt_elems.parse();                 // just parse it!

    std::list<state_element*>& dlist = xi->second;
    std::list<state_element*>::iterator yi = dlist.begin();
    std::list<state_element*>::iterator yie = dlist.end();
    KCHARP w_subrule_derived_states =
           "\\Subrulederivedstatesindent ";
    KCHARP w_merged = "{\\Mergedstate{%i}}";
    std::set<int> chk_merge;
    chk_merge.clear();
    for(;yi!=yie;++yi){// derive state list per closured production
      fsm->ow_index_file_ << w_subrule_derived_states << endl;
      state_element* se = *yi;
      state_element* dse = se;
      for(;dse!=0;dse = dse->next_state_element_){// walk the plank
        fsm->ow_index_file_ << dse->self_state_->state_no_;
        if(dse->next_state_element_ != 0){
          std::set<int>::iterator si =
          chk_merge.find(dse->next_state_element_
```

```
                        ->self_state_->state_no_);
          if(si != chk_merge.end()){
            sprintf(fsm->big_buf_
                   ,w_merged
                   ,dse->goto_state_->state_no_);
            fsm->ow_index_file_ << fsm->big_buf_ << endl;
            break;
          }
        }
        chk_merge.insert(dse->self_state_->state_no_);
        fsm->ow_index_file_ << "\\ \\ " << endl;
      }
    }
                               ... elided code
  ***
  }
}
```

I leave this sketchy example as an enticement as later in this chapter trees are fully developed.

8.3 More requirements and cracked thoughts

Consistency of behaviours demanded an interface whereby these underlining mechanics were hidden. Importance is the what and not how it gets done. To do this C++ has 2 powerful facilities: inheritance, and templates. Mixing templates within the inheritance context makes for a very flexible abstraction facility.

I admit upfront that i'm not an expert on template use. U'll probably see this in my attempts to this interface. There are probably much more efficient ways in using templates. At the time of developing containers my goal was to rough out the accessing and just get it going. There were weaknesses in template deployments by the various compilers i used. Witness the excellent book by Nicolai M. Josuttis "The C++ Standard Library - A Tutorial and Reference" [11]. With it came cautions on the current state of template deployments. This is not an excuse for my deployment efforts nor their weaknesses. It is my compromise to the past to get O_2 into a working portable product allowing me to learn and play with it within various development contexts. So on with the show Dave, on with the show.

8.4 Abstracting the containers

Here are the out-of-the-box containers:

1. List of terminals.
2. Source file to be tokenized.
3. String of raw characters that is similar to the source file container in that it produces from a character string its raw character Tes from memory.
4. Tree. A tree node that gets walked somehow and its nodal content delivered. Each node can come from the grammar's vocabulary: Rules and Terminals.

The "list of terminals" uses the generic C++ template facility while the balance are specialized templates to optimize for speed. The inheritance and base template now follow.

8.4.1 Basing the base: `tok_base` and `tok_can`

I will not detail the template form for creating new container types but review what is required.

- `UINT pos();`
  Current T number that container is working on relative to 0.

- `UINT size();`
  The number of items in the container. Zero indicates empty.
- `void clear();`
  Empty the container of its contents. It does not do any recycling on its entries like giving it back to the memory heap.
- `push_back(CAbs_lr1_sym*);`
  Place the terminal at the back or end of the container.
- `bool empty();`
  Is the container empty of terminals?
- `CAbs_lr1_sym* operator[](UINT Pos);`
  Fetch a specific T by token number relative to 0. If the request is out of bounds, it will return the `eog` terminal. This method does double duty: random and sequential accessing. Sequential delivery depends on incrementing the requested terminal number by 1 per request. All containers determine whether the requested T is already inside it, whether the end-of-container has been reached, or keep fetching until either the end-of-container has been reached or the terminal found.

Each container has these above methods. This is dictated by use of C++'s *virtual* keyword against these methods. Depending on the container type, they could have additional variables/methods inside them. This is particular in the tree container where it maintains two internal containers: the tree stack of walked nodes, and its found Tes. Containers have a write-once read-many times attitude. This means u cannot override the contents of the container once an entry has be made.

From a `Parser` perspective these 2 methods access the associated input container's contents:

```
CAbs_lr1_sym* get_next_token();
CAbs_lr1_sym* get_spec_token(UINT Pos);
```

The `Parser` also maintains its own access references into its input container:

```
CAbs_lr1_sym* current_token();
UINT current_token_pos();
```

Other container memories are kept like:

```
CAbs_lr1_sym* start_token();
UINT start_token_pos();
```

As discussed in the `Parser` chapter, u can query and adjust these attributes.

8.5 TOKEN_GAGGLE Generic container

Do u like duck à la orange? I do and so does O_2. Just imagine terminals walking like domesticated geese. Try keeping them in line: a little here and a little there we go. Typically this is your starting outputted container (producer) for a parser that started its parsing with a "source file" container. I mention typically as this usually is the starting point for lexing and syntax parse stages. Later parse stages would use the tree container. In fact all the other output containers for `Parser` like error, and recycling use this container type.

To keep the Tes in line, u have the type `GAGGLE_TOKEN_ITER` allowing u to iterate thru the container. It uses the same pattern as C++'s Standard Template Library (STL) containers with their `begin()` and `end()` methods. But why go thru this route when the container already has an access operator? Just start from 0 and away u go. Here are the 2 ways to do it. Have a gander back in the grammar chapter 2.5.1 on how the `P3_tokens` container is defined and outputted to. Here's the C++ template way of iterating through a container.

```
yacco2::TOKEN_GAGGLE_ITER i= P3_tokens.begin();
yacco2::TOKEN_GAGGLE_ITER ie= P3_tokens.end();
int y(1);
cout<<"Dump of P3 tokens"<<endl;
for(;i!=ie;++i){
 CAbs_lr1_sym* sym= *i;
 if(sym == yacco2::PTR_LR1_eog__) continue;
```

```
 cout<<y<<":: "<<sym->id__
  <<" file no: "<<sym->tok_co_ords__.external_file_id__
  <<" line no: "<<sym->tok_co_ords__.line_no__
  <<" pos: "<<sym->tok_co_ords__.pos_in_line__
  <<endl;
  ++y;
}
```

Here is O_2's way to use the container's `CAbs_lr1_sym* operator[](Pos)`. All following containers will use this access pattern.

```
cout<<"Dump of P3 tokens"<<endl;
int y(1);
int x(0);
CAbs_lr1_sym* sym(0);
for(;(sym = P3_tokens.operator[](x)) != yacco2::PTR_LR1_eog__;++x){
 cout<<y<<":: "<<sym->id__
  <<" file no: "<<sym->tok_co_ords__.external_file_id__
  <<" line no: "<<sym->tok_co_ords__.line_no__
  <<" pos: " << sym->tok_co_ords__.pos_in_line__
  <<endl;
  ++y;
}
```

Note that `sym` is an abstract terminal. To make it into a specific type, one must test its enumerated id and then cast it into it concrete type.

8.6 tok_can<std::ifstream> Source file container

U might question why is this and the following defined containers use this template genre instead of defining it with a *typedef*? This was a conscience decision by me to reenforce to myself what template container type it was. Rather a weak decision but leaking my thoughts. I leave it to u to define your own `typedef` if u find it too close to detail. May i suggest to u Dave that `typedef tok_can_file_source` might have been more appropriate? Enough already.

Its claim is turn the raw characters into raw Tes from a source file. It has 2 ctors:

```
tok_can(const char* File_name);
tok_can();
```

One provides the "file name" to open with immediately, and the other with no arguments will receive its "file name" later. The delay allows u to do your own "file name" concatenating or verifying that it exists before u commit the container against a specific file. The container does not support going through a potential list of base directories looking for the file. It deals only in explicitly pathed files. The other extra methods are:

- Is the file legitimate? `bool file_ok();`
- Literal file name used. `std::string file_name();`
- File name to be used. `void set_file_name(const char* File_name);`
- Open a delayed file. `void open_file();`

Apart from automatically fetching the individual text from the file, other partial benefits are its automatic GPSing of the terminal to its raw source file/character position. This takes the burden off the grammar writer when tracing or error reporting as the raw character terminals become the GPS sourcing to these error terminals that in turn are reported back to u in source file terms. An example taken from O_2 command line parse:

```
// Lexing the grammar
using namespace NS_pass3;
tok_can<std::ifstream> // source file container with grammar to parse
      cmd_line(o2_file_to_compile.c_str());
Cpass3 p3_fsm;
Parser pass3(p3_fsm,&cmd_line,&P3_tokens
            ,0,&Error_queue,&JUNK_tokens,0);
pass3.parse();
```

The pass3 parser now digests cmd_line's contents and leaves its output in the output container P3_tokens or worse in the Error_queue container.

8.7 tok_can<std::string> Memory container

I admit i haven't used this container very much. It is more of an after-thought on dynamically created statements where the content is resident in a memory buffer to be parsed. It bypasses the need to write the buffered data to an output file before parsing its contents. Messaged based systems like SOAP/XML protocols, or dynamically generated SQL programs would use this container extensively. Like the source file container, there are 2 ctors: one with the string to parse with, and the other delayed having no argument.

```
tok_can(const char* String,CAbs_lr1_sym* GPS=0);
tok_can();
```

Now u might raise the issue of how can one relate the GPS to text where there is a non-existent source file? U can't unless u use a proxy from some other T source relating it to an external file or reporting of the error from an error reporting grammar simulates the lines read with their line number and character displacement.

So let's look at giving the container a string to be parsed immediately. All it knows is there is no file name associated with it but it could potentially associate a proxy T with it by using its GPS as the starting point for the inputted string. The proxy T can be null thus removing the potential GPS outsourcing. So it would have a line number 1 and the start character position 0. Dave this is half baked. I know it is as it raises the thought that the memory text to be parsed was already in some external source file so why use the memory route to parse it? My example will tell why.

Delaying the string to be parsed allows u to declare the container before it gets used in the parse:

```
void set_string(const char* String);
```

sets the string to be parsed in the container.

What about recycling of this container for multiple string reuse? What i mean is: can the container's content be cleared and reloaded with new text to be tokenized? It was not built for this. I designed it as a "quick and dirty" way to dynamically create and parse text to be thrown away after use.

I'll give u a real example from a Pascal translator that had to parse a file declaration whereby its attributes had to be retargeted into a different set of file attributes for another Pascal compiler. It dealt with Pascal's reset and rewrite verbs. The attributes were declared within a Pascal's literal. So the string itself was properly accepted by the translator as a literal. But its contents needed to be reparsed to extract the file attributes inside it. So relating the literal raw characters to the contents allowed me to report invalid content back to the source program's literal string.

```
p3:;// extract default rsx, rms options
  tok_can<string> default_str_can(str_to_parse.c_str(),dft_str);
  using namespace NS_reset_rewrite_dflt_opts;
  Creset_rewrite_dflt_opts dflt_opts_fsm;//extract rsx/rms file opts
  Parser dflt_opts(dflt_opts_fsm
                  ,&default_str_can
                  ,parser()->error_queue());
  dflt_opts.parse();// go pull some teeth
  if(dflt_opts_fsm.error_ != 0){ // fsm holds potential error posting
   parser()->add_token_to_error_queue(*dflt_opts_fsm.error_);
   return;
  }
  (*Access_method) += dflt_opts_fsm.access_method_.c_str();
```

```
    (*Sharing) += dflt_opts_fsm.sharing_.c_str();
    if (dflt_opts_fsm.deflt_extension_.empty() != true){
     (*Default_ext) += ",DEFAULT := '";
     (*Default_ext) += dflt_opts_fsm.deflt_extension_.c_str();
     (*Default_ext) += "'";
    }
```

Here is a buffer resident input example where the Error reporting grammar has to deal with its line number and character displacement to allow the errant buffered line to be outputted with the uparrow underlining where the source line problem is.

```
Rerror (
lhs{
  user-declaration
  void error_where(CAbs_lr1_sym* E_sym){
   Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
   std::string line_of_data;
   yacco2::UINT lno(1);
   yacco2::UINT dlno(E_sym->tok_co_ords__.line_no__);
   const char* chr_ptr = fsm->mailbuffer_;
   for(;lno<=dlno;++lno){
     for(;*chr_ptr !=0;++chr_ptr){
       line_of_data += *chr_ptr;
       if(*chr_ptr == '\n'){
         ++chr_ptr;
         break;
       }
     }
     if(lno == dlno) break;
     line_of_data.clear();
   }

   std::string space(" ");
   std::string::size_type f = line_of_data.find_first_of('\t');
   for(;f != std::string::npos;){
     line_of_data.replace(f,1,space);
     f = line_of_data.find_first_of('\t');
   }

   (*fsm->log_file_) << "Error in mail buffer: " << std::endl;

   (*fsm->log_file_) << line_of_data.c_str() << std::endl;
   for(int pos = 1;pos < E_sym->tok_co_ords__.pos_in_line__;++pos){
     (*fsm->log_file_) << ' ';
   }
   (*fsm->log_file_) << '^' << std::endl;
   (*fsm->log_file_) << "\tfpos: " << E_sym->tok_co_ords__.rc_pos__
     << " line#: " << E_sym->tok_co_ords__.line_no__
     << " cpos: " << E_sym->tok_co_ords__.pos_in_line__
     << std::endl;
   if(E_sym->tok_co_ords__.who_file__ != 0){
     (*fsm->log_file_) << "\twho thru it: " << E_sym->tok_co_ords__.who_file__
       << " line#: " << E_sym->tok_co_ords__.who_line_no__
       << std::endl;
   }
  };
  ***
}
){
  -> "nested files exceeded" {
```

```
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
  error_where(sf->p1__);
  (*fsm->log_file_) << "\t" << sf->p1__->id__
    << " nested number: " << sf->p1__->nested_cnt() << std::endl;
  DELETE_T_SYM(sf->p1__);
***
}
-> "no filename" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "bad filename" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__
 << " filename: \"" << sf->p1__->file_name()->c_str()
 << "\" does not exist" << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "bad int-no" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "bad int-no range" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "bad eos" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "bad esc" {
op
  Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
 error_where(sf->p1__);
 (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
 DELETE_T_SYM(sf->p1__);
***
}
-> "comment-overrun" {
```

```
  op
    Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
   error_where(sf->p1__);
   (*fsm->log_file_) << "\t" << sf->p1__->id__<< std::endl;
   DELETE_T_SYM(sf->p1__);
  ***
  }
  -> "bad char" {
  op
    Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
   error_where(sf->p1__);
   (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
   DELETE_T_SYM(sf->p1__);
  ***
  }
  -> |+| {// catch balance of errors
  op
   Cerror_hdlr* fsm = (Cerror_hdlr*) rule_info__.parser__->fsm_tbl__;
   error_where(sf->p1__);
   (*fsm->log_file_) << "\t" << sf->p1__->id__ << std::endl;
   DELETE_T_SYM(sf->p1__);
  ***
  }
}
```

The only importance here is: create the container, set up the grammar, and parse with it. I know i'm boring but this is your 3 chord song with variation.

8.8 tok_can<yacco2::AST*> Trees wonderful trees

Your wait will definitely be worth it. Before i open up this wonderful Pandora's box[4], let's sit back and look at what i was trying to do. Trees are not terminals. They are branching structures linked somehow thru pointers. Their contents can be anything. My design had to be open-ended on a tree's contents and canonical in the tree node structure. The only way i could do this was have the tree's contents be a pointer to something. This something was the grammar's T vocabulary. So lets digress a bit and look at this canonical tree structure.

8.8.1 The AST

Well u know trees: there is a left branching link, and a right branching link, and its contents. As O_2 was built using the double O principle: object oriented, i wanted tree behaviours ready made. For example the building of trees, walking them, cut, paste, cloning of trees, and fetching specific nodes. Here is AST definition. This should flavour your palate drawn from O_2's library documentation.

```
@*2 Tree node definition |AST|.\fbreak
Note on linkages:\fbreak
\ptindent{1) lt parent to son relationship: dominant order}
\ptindent{2) rt older to younger relationship: equivalence order}
\ptindent{3) pr points to previous older brother or parent}
The ‘‘pr’’ relationship provides a backward link in the tree.
It’s just a pointer to an older node in the tree:
a younger brother linking to its older brother or
the 1st son linking to its parent.
A dink node (double income no kids) would have lt null: no kids.
Within its surrounding, A dink node could still be a son or a forest.
```

[4] I say this as potential ills to be solved by manageable problems?

```
@<Struct...@>+=
struct AST{
AST(yacco2::CAbs_lr1_sym& Obj);
AST();
~AST();
static AST*  restructure_2trees_into_1tree(AST& S1,AST& S2);
static void  crt_tree_of_1son(AST& Parent,AST& S1);
static void  crt_tree_of_2sons(AST& Parent,AST& S1,AST& S2);
static void  crt_tree_of_3sons(AST& Parent,AST& S1,AST& S2,AST& S3);
static void  crt_tree_of_4sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4);
static void  crt_tree_of_5sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4,AST& S5);
static void  crt_tree_of_6sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4,AST& S5,AST& S6);
static void  crt_tree_of_7sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4,AST& S5,AST& S6,AST& S7);
static void  crt_tree_of_8sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4,AST& S5,AST& S6,AST& S7,AST& S8);
static void  crt_tree_of_9sons(AST& Parent,AST& S1,AST& S2,AST& S3
                               ,AST& S4,AST& S5,AST& S6,AST& S7
                               ,AST& S8,AST& S9);
static void  join_pts(AST& Parent,AST& Sibling);
static void  join_sts(AST& Elder_sibling,AST& Younger_sibling);
static void  ast_delete(AST& Node,bool Due_to_abort=false);
static AST*  find_depth(AST& Node,yacco2::INT Enum);
static AST*  find_breadth(AST& Node,yacco2::INT Enum);
static yacco2::CAbs_lr1_sym*  content(AST& Node);
static AST*  get_1st_son(AST& Node);
static AST*  get_2nd_son(AST& Node);
static AST*  get_3rd_son(AST& Node);
static AST*  get_4th_son(AST& Node);
static AST*  get_5th_son(AST& Node);
static AST*  get_6th_son(AST& Node);
static AST*  get_7th_son(AST& Node);
static AST*  get_8th_son(AST& Node);
static AST*  get_9th_son(AST& Node);
static AST*  get_spec_child(AST& Tree,yacco2::INT Cnt);
static AST*  get_child_at_end(AST& Tree);
static AST*  add_child_at_end(AST& Tree,AST& Child);
static AST*  get_younger_sibling(AST& Child,yacco2::INT Pos);
static AST*  get_older_sibling(AST& Child,yacco2::INT Pos);
static AST*  get_youngest_sibling(AST& Child);
static AST*  get_parent(AST& Child);
static AST*  common_ancestor@/
  (Type_AST_ancestor_list& ListA,Type_AST_ancestor_list& ListB);

static AST*  brother(AST& Node);
static AST*  previous(AST& Node);
static void  zero_1st_son(AST& Node);
static void  zero_2nd_son(AST& Node);
static void  zero_brother(AST& Node);
static void  zero_previous(AST& Node);
static void  zero_content(AST& Node);
static void  set_content(AST& Node,yacco2::CAbs_lr1_sym& Sym);
static void  set_content_wdelete(AST& Node,yacco2::CAbs_lr1_sym& Sym);
static void  set_previous(AST& Node,AST& Previous_node);
static void  wdelete(AST& Node,bool Wdelete);
```

```
static bool  wdelete(AST& Node);
// chop and dice them
static void  replace_node(AST& Old_to,AST& New_to);
static void  relink(AST& Previous,AST& Old_to,AST& New_to);
static void  relink_between(AST& Previous,AST& Old_to,AST& New_to);
static void  relink_after(AST& Previous,AST& New_to);
static void  relink_before(AST& Previous,AST& New_to);
static void  add_son_to_tree(AST& Parent,AST& Son);
static AST*  divorce_node_from_tree(AST& Node);
static AST*  clone_tree
             (AST& Node_to_copy,AST* Calling_node
             ,ast_base_stack::n_action Relation=ast_base_stack::init);
 /*
  * Public internal variables
  */
  AST* lt_;
  AST* rt_;
  AST* pr_;// caller who links to it
  yacco2::CAbs_lr1_sym* obj_;
  bool wdelete_;
};
```

From the above u have lots of activities that u can do againt a tree structure. Their methods should be reasonably self describing. How do u relate a T with a tree node? There are 2 ways that this can be done: non-intrusively and intrusively.

In the non-intrusive way, a T does not contain any tree structures. The T is created like the raw characters from an external file source. At some junction point in the grammar like a reducing subrule, it will create a tree of related T items drawn from the parse stack. The `AST*` variable will contain as its contents the T. The T has no knowledge that this is happening. It is the grammar that keeps a running tab on the building of tree nodes, their contents, and their interrelationships. Then it records somewhere the tree's pointer for future reference and relationships.

In an intrusive way the `AST*` tree appendages are part of the T definition. Here the T is created and as a secondary step in its creation, it takes on its own tree node pointer: a reflection of self within its contents. The grammar still relates this T's node with other tree nodes by subrules/rules interaction. In bottom-up parsing, the trees get built in that order dealing with their stacked contents and recording their results as variable tree pointers within the rule.

8.8.2 Some tree building examples

Here is the `bld_its_tree()` method drawn from the `rule_def` of O_2's T vocabulary. This is an intrusive example as the `rule_def` T contains a reference to its own tree. The rule's tree can only get built after one of its subrules has been discovered. Each subrule definition in turn has an intrusive tree reference as content. Their T contents themselves could be of either non/intrusive type. This examples illustrates building of a tree outside of the parse stack contents.

```
 void rule_def::bld_its_tree(){
   its_tree_ = new AST(*this);
   T_subrules_phrase* subrules_ph = subrules();
   std::vector<T_subrule_def*>* subrule_list = subrules_ph->subrules();
   std::vector<T_subrule_def*>::iterator sri = subrule_list->begin();
   std::vector<T_subrule_def*>::iterator srie = subrule_list->end();
   AST* subrule_lvl = 0;
   for(;sri!=srie;++sri){       // walk subrules getting their own tree
     T_subrule_def* srdef = *sri;
     AST* srt = srdef->subrule_s_tree();
     if(subrule_lvl == 0){
      AST::crt_tree_of_1son(*its_tree_,*srt);
      subrule_lvl = srt;
```

```
     }else{                    // join subrules as brothers
      AST::join_sts(*subrule_lvl,*srt);
      subrule_lvl = srt;
     }
   }
 }
```

Now an example taken from the Pascal translator where the tree is dynamicly being built in bottom-up fashion from the parse stack and rule `Rexp`'s own variable `type_` registers its tree assembly for the rule `Rpas_var_def` to use in its own tree building.

```
Rpas_var_def  (){
  -> "var" Rexp |.| {
   op
     AST* p1 = new AST(*sf->p1__);// create node and associate content
     AST* p2 = sf->p2__->type_;   // fetch from stack
     sf->p2__->type_ = 0;

     AST* var_defst = new AST();
     T_var_defs* var_defs = new T_var_defs(var_defst);
     var_defs->set_rc(*sf->p1__,__FILE__,__LINE__);
     AST::set_content(*var_defst,*var_defs);
     AST::join_pts(*var_defst,*p1); // parent / child relation
     AST::join_sts(*p1,*p2);        // older / younger sibling relation
     /*
      * Could have built tree this way also
      *        AST::crt_tree_of_2sons(*var_defst,*p1,*p2);
      * Done to show how u can piece meal a tree's assembly
      * over time.
      */
     RSVP(var_defs);
    ***
    }
}

/*
 * Illustrates how rule definition registers tree results against
 * its internal variables for future reference by rule Rpas_var_def
 * and left recursion on rule Rexp in the 2nd subrule
 * and how the subrules are being built in bottom-up tree snippets
 * using their parse stack contents.
 */
Rexp  (
lhs {
  user-declaration
    public:
    AST* type_;
    AST* eoc_;
  ***
  }
){
  -> Rid_list
       R_TON_scope // Rule dealing with symbol table scope
            Rneeded_colon Rtype_denoter
       R_TOFF_scope
     Rneeded_semi_colon {
     op
       AST* p1 = sf->p1__->type_; // rule Rid_list's tree node
       AST* p3t = sf->p3__->type_;// rule Rneeded_colon tree node
       AST* p4 = sf->p4__->type_; // rule Rtype_denoter tree node
```

```
      AST* p6t = sf->p6__->type_;// rule Rneeded_semi_colon tree node
      sf->p1__->type_ = 0;
      sf->p4__->type_ = 0;

      AST* vdt = crt_var_def_ast_from_id_list(p1,p3t,p4);
      type_ = vdt;
      // find the last in chain
      AST* lic = vdt;
      for(;lic != 0;lic=AST::brother(*lic)){
        if(AST::brother(*lic) == 0) break;
      }
      AST::join_sts(*p4,*p6t);
      eoc_ = lic;
    ***
    }
  -> Rexp Rid_list
       R_TON_scope
            Rneeded_colon Rtype_denoter
       R_TOFF_scope
     Rneeded_semi_colon {
    op
      AST* p1 = sf->p1__->type_;// build on left recursion: rule Rexp
      AST* p1e = sf->p1__->eoc_;
      AST* p2 = sf->p2__->type_;
      AST* p4t = sf->p4__->type_;

      AST* p5 = sf->p5__->type_;
      AST* p7t = sf->p7__->type_;
      sf->p1__->type_ = 0;
      sf->p2__->type_ = 0;
      sf->p5__->type_ = 0;

      AST* vdt = crt_var_def_ast_from_id_list(p2,p4t,p5);
      type_ = p1;
      AST::join_sts(*p1e,*vdt);
      // find the last in chain
      AST* lic = vdt;
      for(;lic != 0;lic=AST::brother(*lic)){
        if(AST::brother(*lic) == 0) break;
      }
      AST::join_sts(*p5,*p7t);
      eoc_ = lic;
    ***
    }
}
            ... elided grammar code
```

These examples should energize your imagination to fuel your own possibilities. I wanted minimum programming effort to maintain and manipulate trees. So u can easily build them, determine their lineage but what about walking them? Next on the agenda.

8.9 Ents, Trees, and Walking?

I point u to a marvellous set of books acronym TAOCP — "The Art of Computer Programming" by Donald E. Knuth. All his expositions are a treat to the mind and eye. Please have a go if u haven't done so particularly in Volume 1: "Fundamental Algorithms" where trees are discussed [16].

Here's O_2's library extract explaining tree containers, functors, and walkers. The `AST` structure allows one to build tree structures where each node enrobes a terminal symbol's address. Each node contains a

left link representing dominance: parent to child relation, a right link representing equivalence: siblings or brothers — your preference of terminology, and a previous link representing an older node; this can be nil as the node is the root, an older brother, or a parent as the node is the oldest child. The previous link depends on where within the tree the node sits. The left and right links can be nil indicating no children, or no younger brothers.

To support the creation and walking of the trees, various static procedures are available. There are 2 tree walkers: prefix and postfix. The way the tree is built, there is no infix walker! The balance of the walkers are variants on these 2 that have restrictions on how much of the tree is to be read.

- Restriction 1: the node is a forest where pre and post fix walks are done — though the node can be linked with brothers, as a forest it stays within its bounds.
- Restriction 2: breadth only walk — walk self and younger brothers.
- Restriction 3: prefix with breadth only — the node is considered a parent; walk itself and its immediate children.

The container has 3 parts: the container of tokens that match the filtering mechanism, the parts needed to walk the tree, and a token access mechanism. As an optimization, the token access determines whether the requested token-by-number is in the container. This allows one to iterate randomly a tree structure. The tree walker linearizes the token stream. It uses a finite automaton with 5 elements in its alphabet: init, left, right, visit, eoc. These represent how the node has been processed. The left and right elements indicate that the dominance or equivalence link is being followed. The init, visit, and eoc are states on how the node was processed. Originally, the initial access of the node represented by 'init', and the end of the node access before it is popped from the stack represented by 'eoc' allowed the user to fine tune the walker's behaviour but this was overkill. The 'visit' state breaks out of the tree traversal and allows one to deal with the situation. Each tree walker implements these states in their 'exec' and 'advance' methods. To control the tree traversal, a stack is used due to the type of control needed to break out of the traversal. Recursion does not allow one to do this due to its implicit call stack and continuous behaviour as opposed to discrete stepwise logic. The only difference to iterating the tree container versus the other token containers is a tree container can only be accessed by token-number. There is no STL type iterator. One accesses the container by its 'operator[]' method iterating by the numbers started by 0. To break out of the iteration, the returned terminal is tested against the `LR1_eog` terminal indicating end-of-tree met.

A functor mechanism is available to capture info at time of the visited node. It can be a stand alone behaviour or it could be used in conjunction with a grammar. For example if a tree's node is being printed by use of a grammar, the recursion level count must be maintained by the functor and used by the grammar's subrule. Why not process the recursion count at the time of the grammar's subrule reduction? Remember: the lookahead terminal to reduce the subrule is the current stack configuration that is one ahead of what's needed. Hence the need for the functor and its registering of recursion level.

As a tree structure is very large and diverse, to deal with specific node types, a set mechanism of inclusion or exclusion of symbols is supported. With these walkers and companions — filters and functor, a tree is walked in linear fashion just like a normal token stream. This allows one to write grammars to consume tree structures in the same spirit as a to-be-parsed language. Typically these phases are the down stream stages of the semantic side to compilation. Really good stuff!

8.9.1 Tree filters and Functors

A filter is basically a set of enumerated ids taken from the grammar's T vocabulary. It uses the C++'s set facility: `std::set<int>`. If a filter is not present it is assumed that all the Tes are acceptable. By stating what type of filter it is: "accept" or "bypass", the tree's contents can be skipped or available for acceptance. Depending on the set membership, choosing whether its an accept or bypass set is a finger blister or a "carpal tunnel" constraint. It's your call on how u want to declare the items in the filter.

To add more flexibility to this mix, in "The C++ Programming Language Special Edition" by Bjarne Stroustrap [18] talks about functors. In the chapter "Overview of Standard Library Algorithm" function objects are discussed. They provide a way to trigger a user supplied function to be called when iterating thru each item within a container. Please have a read as he expresses functors very well. So why not allow

the same functionality to happen when iterating thru a tree and when the tree's contents is accepted by the filter to then trigger this function call.

I will not describe the "how to build" a functor but list what is open to u from out-of-the-box of O_2's library. Then i'll give an example of a custom-made functor. The best way to learn is to read the documentation of O_2's library.

- `prt_ast_functor` Print the tree's node visited.
- `str_ast_functor` Build a literal string from the visited tree node.
- `remove_unwanted_ast_functor` Remove a node from a tree.
- `tok_can_ast_functor` Basic functor that accepts the filter's T given to it. It tells the tree walker to stop looping so that the container can dish up the T for the container's requestor which is normally a parser.
- `insert_back_recycled_items_functor`

Here are extracted comments from O_2's library about some of these functors. These functors are examples of how to create your own functor. `prt_ast_functor` prints out a tree in indented format. `str_ast_functor` claim to fame is in its use of the `BYPASS_FILTER` given the many abstract meta-terminals that parent each subtree: for example the Pascal railroad diagrams with expression, simple expression, term, factor, etc. Depending on how abstract u make the tree, there are still parent nodes that u might not want to see. `str_ast_functor` builds a source string from a tree used in a Pascal translator from Oregon to HP Pascal source code retargeting.

An improvement: the address of the functor is passed to the call-back function so that is can also act as a container. The reason behind this is the `str_ast_functor`. It originally had a global string for the function to fill. As the functor is the driver of the call-back, it is the one that knows when the source string should be cleared for reuse.

To bring this all together, it is the tree walker that receives the filter, and functor. The walker applies the filter and passes the accepted T to the functor which in turn could reject the "potential accepted" T, or do other activities with it. From one of the code examples above here are their declarations:

```
tok_can_ast_functor sr_elems_walk_functr;// accept all Tes of filter
ast_prefix_1forest             // tree walker with accept filter
prt_sr_elems_walk(*sr_t        // tree node to traverse
                 ,&sr_elems_walk_functr
                 ,&fsm->prt_sr_elems_filter_
                 ,ACCEPT_FILTER);
```

8.9.2 A customized functor

I'll just give u the code with comments and let u study it. Here is the problem needing to be solved. The Pascal translator needed to remap the `reset` and `rewrite` verbs due to their file attributes being declared differently from the old Pascal code into a newer Pascal. It required finding somewhere in the source tree the closest Pascal `close` statement associated with the specific open statement.

`functor_result_type operator()(yacco2::ast_base_stack* Stk_env)`
is the function operator that must be written. Use of inheritance against the `Type_AST_functor` demands this. `nearest_close_ast_functor` is a C++ class open to all coding constraints. What is important is the dictate back from the functor to the tree walker on what it should do: `bypass_node`, `accept_node`, or `stop_walking`. I included a 2nd example used in tandem with this one to give u a feel on how to chain together functors. The coding gendre is *Cweb* and C++.

```
/@
@*4 |nearest_close_ast_functor|.\fbreak
Try to find the ‘‘close’’ statement(s) having the
common parent to the ‘‘delete’’ statement.
That is, the tree is walked as breadth-only
along the younger brothers of the ‘‘delete’’ node.
The ‘‘call-stmt’’ filter is used along with the delete's file
descriptor variable to discriminate against the found nodes.
@/
 struct nearest_close_ast_functor :public Type_AST_functor{
   yacco2::functor_result_type
   operator()(yacco2::ast_base_stack* Stk_env){
    using namespace NS_pas_terminals;
    if(fnd_close_ast_ != 0)         // fnd node so fush out
      return yacco2::stop_walking;  // balance of nodes to read
    cnode_ = Stk_env->cur_stk_rec_->node_;
    AST* plist_t = AST::get_2nd_son(*cnode_);
    T_call_stmt* cs = (T_call_stmt*)AST::content(*cnode_);
    T_identifier* pid = cs->proc_id()->identifier();
    string lcid(pid->identifier()->c_str());
    transform(lcid.begin(),lcid.end(),lcid.begin(),tolower);
    string close("close");
    if(close != lcid){
      return yacco2::bypass_node;// not a close procedure call
    }
    get_p1_fd:
    ast_prefix_wbreadth_only
            pwb(*plist_t,ast_functr_,parms_filter_,ACCEPT_FILTER);
    tok_can<AST*> parms_tok_can(pwb);
    using namespace NS_reset_rewrite_parms;// parse passed parms
    Creset_rewrite_parms parms_fsm;
    Parser reset_rewrite(parms_fsm,&parms_tok_can,0);
    reset_rewrite.parse();

    T_variable* p1_var = (T_variable*)parms_fsm.p_[1];

    string p1_cstr;
    string fq_p1_cstr;
    rule_->get_p1(parms_fsm.pt_[1],p1_var,&p1_cstr,&fq_p1_cstr);
    string to_fnd(close_fd_);
    string potential_fd(p1_cstr.c_str());
    close_node:
     if(to_fnd != potential_fd){ // not the closest to fd
      return yacco2::bypass_node;// continue looping
     }
     fnd_close_ast_ = cnode_;
     return yacco2::accept_node;
 };

 nearest_close_ast_functor
    (const char* FD
    ,tok_can_ast_functor* AST_functr
    ,std::set<yacco2::INT>* PARMS_filter
    ,Rcall_st* Rule
    ,bool Stop_search_if_fnd=YES
    )
 :fnd_close_ast_(0),close_fd_(FD)
 ,ast_functr_(AST_functr),parms_filter_(PARMS_filter)
 ,rule_(Rule)
```

```
  ,stop_search_if_fnd_(Stop_search_if_fnd){};

  AST* fnd_close_ast_;
  private:
  const char* close_fd_;
  yacco2::ast_base_stack* stk_env_;
  yacco2::ast_base_stack::s_rec* srec_;
  yacco2::AST* cnode_;
  tok_can_ast_functor* ast_functr_;
  std::set<yacco2::INT>* parms_filter_;
  Rcall_st* rule_;
  bool stop_search_if_fnd_;
};

/@
@*4 |ranked_nearest_close_ast_functor|.\fbreak
Find the ``close'' statement having the
common parent to the ``delete'' statement
and ranked after the current delete node being matched.
@/
struct ranked_nearest_close_ast_functor :public Type_AST_functor{
  yacco2::functor_result_type
  operator()(yacco2::ast_base_stack* Stk_env){
   if(fnd_close_ast_ != 0){ // stop the walking
      return yacco2::stop_walking;
    }
    using namespace NS_pas_terminals;
    AST* cnode = Stk_env->cur_stk_rec_->node_;
    if(pnode_fnd_ == YES) goto srch_for_close_in_potential_list;
    if(pnode_ == cnode){         // from here on all visited nodes
      pnode_fnd_ =YES;
      return yacco2::accept_node;// will be tested
    }
   srch_for_close_in_potential_list:
    int pot_no = potential_close_list_->size();
    for(int x=0;x< pot_no;++x){
      AST* t = (*potential_close_list_)[x];
      if(t == cnode){
        fnd_close_ast_ =cnode;
        return yacco2::accept_node;
      }
    }
    return yacco2::bypass_node;//not closest to fd continue looping
 };

ranked_nearest_close_ast_functor
     (AST& Delete,Type_AST_ancestor_list& Potential_close_list)
     :fnd_close_ast_(0)
     ,pnode_(&Delete)
     ,pnode_fnd_(NO)
     ,potential_close_list_(&Potential_close_list)
  {};
  AST* fnd_close_ast_;
  private:
   yacco2::AST* pnode_;
   bool pnode_fnd_;
   Type_AST_ancestor_list* potential_close_list_;
};
```

8.9.3 Walkers or otherwise

Remember there are only pre and postfix walkers. The variants come in isolating the scope of the tree walk which means there is no infix walker of any sort. There is one interesting tree walker exception — moonwalk looking for ancestors. Music tells its tale. Each walker has the same parameters:

```
(AST& Tree node,functor*,Filter* Filter,Filter type);
```

The best way to describe them is list them. Please note that walkers having the **1forest** as part of their name considers the node as a root even though it could be a node within another tree. This means that the walk contains itself only to its own node and its children. Related brethren are not walked.

- Postfix walks. The tree's leaves are visited before their parent.

  ```
  ast_postfix(...);
  ast_postfix_1forest(...);
  ```

  Even if the tree node has links to older or younger brothers, the walk contains itself to its given start node and its children.

- Prefix walks. The parent is visited before its children.

  ```
  ast_prefix(...);
  ast_prefix_1forest(...);
  ```

  Stay within the confines of this node acting as a tree visiting only itself and children.

  ```
  ast_prefix_wbreadth_only(...);
  ```

  Visits self and only its immediate children.

- Breath-only. Walk self and any younger brothers.

  ```
  ast_breadth_only(...);
  ```

- Moonwalk. Walk tree backwards looking for some specific ancestor.

  ```
  ast_moonwalk_looking_for_ancestors(...);
  ```

From a bird's perch this is how the tree container delivers the token request. It has a "visited nodes walk stack" and an internal container of accepted Tes. If the token number requested is in its internal container, it immediately returns the T back to the requestor. Now the fun begins when the requested T has to be continued by walking thru the tree. The walker uses its internal walk stack to discretely control the visited nodes, filter their content, and to fire off its functor. Given the result from the functor, it could continue walking until one of the following happens, accept the T to be returned back to the requestor, or stop/abort the walk due the tree-end reached or user functor dictates. There is always a T returned back from the container be it `eog` or specific.

As the tree container returns Tes under some constraint, what if u need its associated tree node? Well the tree container has such a method:

- `AST* ast(UINT Pos);`

Here's an example using the `parms_tok_can` container fetching its fourth token's tree node:

```
AST* expt_t = parms_tok_can.ast(3);
```

Remember the token number starts from 0. U might question why the need for the tree node when its T content has been delivered? Depending on the delivered T, quite possibly u might need to walk its underlining tree with other filtering constraints. Or possibly the delivered T has no intrusive tree definition inside it and u want to follow its older relations. U get the notion and are now open to splitting these tree triggers into other T streams.

8.10 Summary

In closing I will not develop the specifics of the tree functor nor the tree container's internal components. This is left in O_2's library documentation. It should entice u to read its documentation and learn from O_2's own implementation. Now on to "errors", debugging, and traces of frustrations. Of course this is for the curious as u don't make mistakes. It's only the tool being used fault. Sure Dave sure.

Chapter 9

Errors: how to catch a rogue

Before we set off on our safari, let's look at the adversary. It's the shadow that is elusive and shy; the unexpected ambush; the out-of-the-blue appearance of an unknown symbol. So we're stalking this unknown down the symbol tracks leading into unknown territory. Instead of trees, canyons, plains, it is herds of symbols linearized into shapes. It's the unexpected shape that becomes our quarry. Grammars define expected symbol shapes. When the shape distorts is when the quarry announces itself.

So how to detect the unexpected particularly when the unexpected is not explicit? The only grammar detector available is the wild symbol. It itself is amorphous in shape. Fortunately for us the "wild symbol" has an identity that represents at parse time a dynamic unknown symbol that also has an explicit identity by probing. These unknowns basically are all the Tes that are not on the guest list to the grammar's state's party. So at various points within the grammar's symbol sequence, traps can be set to catch the elusive prey. These vigilance points must be programmed by u the grammar writer.

9.1 Wild one i think i love u

The 2 wild symbols are: `|?|` and `|+|`. `|?|` indicates the unknown symbols that are in error while `|+|` leaves the wildness open to interpretation. The interpretation could be that the arrived symbol is kosher or that it's errant.

The use of these symbols depends on how u want to deal with the errant arrival. The `|?|` wild symbol states to the `Parser` to not depend on the lookahead boundaries if a reduce operation is to take place. It is a declaration that states to the `Parser` do-not-use the lookahead symbol for your subrule reduce. This doesn't make sense Dave. Okay, a symbol arrives 2 ways to the `Parser`: directly by the `get_next_token()` procedure and indirectly thru a returned thread call statement. For the direct T access, the `Parser` should find it in its state's shift table and deal with it accordingly. This second return type also states what/where the token stream restart position will be by its companion lookahead boundary data; for the calling grammar this is where it will continue parsing. It is not the called thread's returned T that causes the problem as this is caught explicitly or wildly by the calling grammar. It is the lookahead boundary used to reduce the called thread statement that is errant. As it is erroneous, the reduce of the thread call statement will probably not take place as the lookahead T could possibly not be in the reduce's follow set for that subrule. This is why the called thread reduce for the wild symbol `|?|` is divorced from its follow set; this is called an lr(0) reduction where the 0 indicated no follow set dependency.

9.2 Trappings

This section will explain by examples how to set various traps. There are 2 error settings: the no thread call, and the thread call. Let's look at the regular grammar error detection. Basically a grammar rule's subrule states what the order Tes should be delivered by the input token container. This expression can be explicitly stated for each T. For example the following subrule describes the T sequence to accept the word "cat".

```
Rcat_in_the_hat {
```

```
  -> c a t
}
```

Now how would u capture an errant sequence like "cab" or "cite"? If our T vocabulary is very small u could explicitly program all errant sequences but this leads to a false sense of doing it right. What happens when the T vocabulary is expanded? This leads to the combinatorial explosion to deal with all the erroneous sequences. This is error prone to possible omission and also explodes the parse tables in size. The better way is to set traps at points within the sequences.

Now how would u inject these traps within the above subrule? One way is the following:

```
Rcat_in_the_hat {
  -> c a t
  -> c a |?|
  -> c |?|
}
```

This partially captures in a semi open-way errors but it itself is too explicit. It also goes against the above warning of too fat in table size.

9.2.1 General entrapment and the `failed` directive

So let's re-look at this from a trapper's perspective. Where should the traps be placed? If the caught error is to return "a general error" T as in a bad O_2's command line option, the `failed` directive would be appropriate. This directive applies both to threaded and monolithic grammars. The difference is a threaded grammar returns an error thru the "Accept queue" of the calling grammar who uses arbritration to deal with it whereas the monolithic grammar places the error directly into the "Error queue". For example, threaded grammar "o2_lcl_opt.lex" programmed it this way [1]:

```
fsm
(fsm-id "o2_lcl_opt.lex",fsm-filename o2_lcl_opt
,fsm-namespace NS_o2_lcl_opt
,fsm-class Co2_lcl_opt{
/@
Trap the failed option and return a bad command.
This covers errors like the premature prefix -e where it should
be -err. i could have been less specific to trap
non first set options like (-z) by defaulting  to this
facility but i'm teaching myself...
As this thread is executed according to its first set ``-'',
any failed attempt is a bad option.
Please note the use of the |RSVP_FSM| macro.
Its context is different than the normal Rule's
use of |RSVP| macro.
@/
  failed
       CAbs_lr1_sym* sym = new Err_bad_cmd_lne_opt;
       sym->set_rc(*parser__->start_token__,__FILE__,__LINE__);
       RSVP_FSM(sym);
       return true;
  ***
}
```

The `failed` directive gets called when the parse is unsuccessful. Normally an unsuccessful parse is due to the parsing lr1 tables not being able to field the rogue T. But u can explicitly declare an unsuccessful parse by using the `set_abort_parse(true)` procedure from your syntax directed code which was fielded by a recognized subrule. Under this parse condition, the parser always always calls the `failed` directive. If `failed` returns a true back to the parser, the parse is considered successful: a T will have been placed into the accept queue of the calling grammar from within the `failed` code for the calling grammar to field. A returned false

[1] See pager_1.lex in ./grammar-testsuite for a monolithic grammar example.

code means the parse really has failed and aborted to which the calling grammar will receive an unsuccessful parse status. This returned parse condition back to the calling grammar depends on the number of threaded grammars called. If all the called grammars aborted then the threaded call expression within the calling grammar is considered unsuccessful. Remember this is important in the conditional parsing attempts by the `Parser`: "threaded grammar calls are attempted first" followed by the normal parsing operations: shift, accept, reduce. If u stop the parse by calling the `set_stop_parse(true)` procedure, the `failed` directive is not called. This allows one to return a T into the accept queue of the calling grammar and to not complete the parse activity back to the "Start rule". The parse is considered successful. `set_stop_parse` makes the parse a little more efficient in that the parser's run activity stops after the recognized subrule and does not continue pushing/popping grammar symbols to eventually do an "accept" operation.

In the above example, the general error T is setting its `GPS` coordinates to the thread's start token. As the "failed" procedure is part of the `fsm`, it uses the RSVP_FSM macro to get at the parser so that it can return the error T thru the accept queue of the calling grammar. Why not post it to the "error queue"? **"accept queue" posting is the normal way that threaded grammars** report their results back to the calling grammar. It is left to the calling grammar to decide what/how to deal with it. This is transitive in nature to threaded grammars: as a threaded grammar could call a threaded grammar and so the protocol is return any T back thru the "accept queue". The threaded grammar receiving an error could just repost it back in the "accept queue" of its calling grammar. And in music terms "the beat goes on". Finally a monolithic grammar will receive it calling error T and post it to the error queue and quietly quit.

Though the `failed` function is always there within each grammar's `fsm`, it is only executed when the parse is unsuccessful. If the grammar does not explicitly program for it by using the `failed` directive, it quietly does nothing even when u've taken other measures to deal with the errant parse.

9.2.2 Differences in registering errors

Where do u eventually register an error? In the Error container Dave! But what if the called grammars chain is long, how does it get placed into the error container and what about letting your caller grammar know about it? Macros **ADD_TOKEN_TO_ERROR_QUEUE** or its suffixed **_FSM** brother place the error T into the error container. But they do not tell the calling grammar that this was done. The caller grammar is told only what happened to the called grammar be it good or aborted; that is all. The easiest way is to pass the error T back to the caller is via the macros **RSVP** or **RSVP_FSM**. In the returning grammars called chain, using this route leaves the error handling responsibility to the monolithic grammar that started the parsing call chain. It is the monolithic grammar that places the error T into the error container and possibly shuts down for the following post error evaluation code to cut in. Only one error T gets selected from a grammar call chain. The accept queue can have many entries placed into it but it is the calling grammar's "arbitration code" dealing with the results returned from the called grammars that selects the appropriate T to continue parsing with. Normally the potential of multiple Tes getting placed into the accept queue is determined on the number of competing threads launched at the same time. What the arbitration code does with the other Tes could in theory be extracted and shuffled elsewhere. I have not tried this but it looks plausible. If the error cascades more than one error T, what should u do? Well u can still place the other error Tes into the error container but u must i said must return an error T back to the caller.

Why this puffy route nose mashing? Returning an error back to the calling grammar lets the calling grammar do what it wants. It could act as a proxy and refine the error into another more comprehensive error T or the error could be taken less seriously and continued parsing rather that coming to a sudden parse stop. Eventually the monolithic grammar will be given an error T to deal with which is to place it into the error container. It could use the variant macros **ADD_TOKEN_TO_ERROR_QUEUE** to place it into the error container or use the parser's method **add_token_to_error_queue()** to do it. Why the variants to these error place macros? It is the context from where they are used: **_FSM** from within the grammar's **fsm** methods or from a grammar's rules. For now error passing is not too sophisticated: just simple and minimalist in effectiveness. Efficient? Flexible? Time tells its own tale.

9.3 Detailing errors

Generality is nice but possibly generic error messages leave the targeted user with a head scratch and a guessing game as to what it meant. U the grammar writer want to be more sensitive and return a more appropriate message to your audience depending on the context fielded. Let's look at a real threaded grammar where there are different sensitivities to errors. "T_enum_phrase_th.lex" grammar handles the enumeration parse of a grammar. I'll provide an unabridged version as u are more mature in reading grammars. Don't look at this as landfill but a small lesson on how i did things. U can review the enumeration construct described in `fsm` chapter 4 but it should not be necessary as the grammar is quite simple and the rules names should clue u into their intent.

It calls 6 other threads. `user-prefix-declaration` directive includes their definitions. This threaded grammar is called out by the monolithic grammar "T_enum_phrase.lex". The "o2_sdc.h" stands for "syntax directed code" and the other include names should be adequate to figure out their intent. As a memory jog each grammar's construct phase was parsed by a topdown procedure containing a monolithic grammar that called its companion threaded grammar. I cant this to reenforce the idea that a parse can be broken up into individual monolithic grammars where the overall parsing spectrum is divided into smaller parsing spectrums all drawing their input from 1 container. Except for O_2's last rules phase having the `eog` as its ending boundary, all other phases have T segment snippets ending at the following phase start segment. If u don't remember the enumeration construct syntax please reread the Terminals chapter 3.

```
/@
@** |T_enum_phrase_th| Thread.\fbreak
parse T-enumeration phrase.
@/
fsm
(fsm-id "T_enum_phrase_th.lex"
,fsm-filename T_enum_phrase_th
,fsm-namespace NS_T_enum_phrase_th
,fsm-class CT_enum_phrase_th{
  user-prefix-declaration
#include "lint_balls.h"
#include "eol.h"
#include "c_comments.h"
#include "identifier.h"
#include "c_string.h"
#include "o2_sdc.h"
    using namespace NS_yacco2_terminals;
  ***
  user-declaration
    public:
    T_enum_phrase* enum_phrase_;
  ***
  op
    if(enum_phrase_ != 0){
      delete enum_phrase_;
      enum_phrase_ = 0;
    }
    enum_phrase_ = new T_enum_phrase;
    enum_phrase_->set_rc(*parser__->start_token__,__FILE__,__LINE__);
    AST* t = new AST(*enum_phrase_);
    enum_phrase_->phrase_tree(t);
  ***
  constructor
      enum_phrase_ = 0;
  ***
}
,fsm-version "1.0",fsm-date "22 mar 2004",fsm-debug "false"
,fsm-comments
    "Parse ``T-enumeration'' construct: Time out smell the tullips.")
```

```
parallel-parser
(
  parallel-thread-function
    TH_T_enum_phrase_th
  ***
  parallel-la-boundary
    eolr
  ***
)
@"/yacco2/compiler/grammars/yacco2_T_includes.T"

rules{
RT_enum_phrase (){
  -> Rlint
     Ropen_par
       Rparameters
     Rclose_par
     Rlint
     Rk_defs_phrase
     Rlint  {
    op
      CT_enum_phrase_th* fsm =
           (CT_enum_phrase_th*)rule_info__.parser__->fsm_tbl__;
      RSVP(fsm->enum_phrase_);
      fsm->enum_phrase_ = 0;
    ***
    }
}
Ropen_par (){
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_open_parenthesis;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  "("
}
Rclose_par (){
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_close_parenthesis;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  ")"
}
Rparameters (){
  -> Rlint
     Rfilename_phrase Rlint
     Rnamespace_phrase Rlint
}
Rfilename_phrase (){
  -> Rfilename Rlint Rfilename_id
}
Rfilename (){
```

```
  ->  ||| "#file-name" NS_identifier::TH_identifier
  ->  ||| |?| NULL {
  op
    sf->p2__->set_auto_delete(true);
    CAbs_lr1_sym* sym = new Err_no_filename_present;
    sym->set_rc(*sf->p2__,__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_filename_present;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
}
Rfilename_id (){
  ->  ||| identifier NS_identifier::TH_identifier {
  op
    CT_enum_phrase_th* fsm =
        (CT_enum_phrase_th*)rule_info__.parser__->fsm_tbl__;
    fsm->enum_phrase_->filename_id(sf->p2__);
  ***
  }
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_filename_id_present;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
}
Rnamespace_phrase  (){
  -> "," Rlint Rnamespace Rlint Rnamespace_id
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_comma_present;
    sym->set_rc(*parser__->start_token__,__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
}
Rnamespace  (){
  ->  ||| "#name-space" NS_identifier::TH_identifier
  ->  ||| |?| NULL {
  op
    sf->p2__->set_auto_delete(true);
    CAbs_lr1_sym* sym = new Err_no_namespace_present;
    sym->set_rc(*sf->p2__,__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  |?| {
```

```
  op
    CAbs_lr1_sym* sym = new Err_no_namespace_present;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
}
Rnamespace_id  (){
  ->  ||| identifier NS_identifier::TH_identifier {
  op
    CT_enum_phrase_th* fsm =
        (CT_enum_phrase_th*)parser__->fsm_tbl__;
    fsm->enum_phrase_->namespace_id(sf->p2__);
  ***
  }
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_namespace_id_present;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    rule_info__.parser__->set_stop_parse(true);
  ***
  }
}
Rk_defs_phrase  (){
  -> "{" Rlint Rconstant_epi_defs Rlint Rclose_brace
}

Rconstant_epi_defs  (){
  ->  Rconstant_defs Rconstant_defs_code
  ->
}
Rconstant_defs  (){
  ->  ||| "#constant-defs" NS_identifier::TH_identifier {
  op
    sf->p2__->set_auto_delete(true);
  ***
  }
  ->  ||| |?| NULL {
  op
    RSVP(sf->p2__);
    parser__->set_stop_parse(true);
  ***
  }
}
Rconstant_defs_code  (){
  ->  ||| "syntax-code" NS_o2_sdc::TH_o2_sdc {
  op
    CT_enum_phrase_th* fsm =
        (CT_enum_phrase_th*)rule_info__.parser__->fsm_tbl__;
    fsm->enum_phrase_->kdefs(sf->p2__);
  ***
  }
  ->  ||| |?| NULL {// error from syntax code thread
  op
    CAbs_lr1_sym* sym = new Err_no_kdefs_code_present;
    sym->set_rc(*sf->p2__,__FILE__,__LINE__);
    sf->p2__->set_auto_delete(true);
```

```
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  |?| { // thread not called due to current T not in its first set
  op
    CAbs_lr1_sym* sym = new Err_no_kdefs_code_present;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
}
Rclose_brace  (){
  ->  |?| {
  op
    CAbs_lr1_sym* sym = new Err_no_close_brace;
    sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
    RSVP(sym);
    parser__->set_stop_parse(true);
  ***
  }
  ->  "}"
}
Rlint  (){ // remove the white stuff from the grammar's clothing
  ->  ||| lint NS_lint_balls::TH_lint_balls
  ->  // implicit epsilon rule ie handles no white space
      // the explicit |.| could have been used
}
}// end of rules
```

The first thing u notice is the use of rules to package specific Tes and the error traps set by `|?|`. Each error T is specific to the rule's subrule context. Take particular note that it uses the `current_token()` as it GPS reference point to the source T and associated file:

```
sym->set_rc(*parser__->current_token(),__FILE__,__LINE__);
```

It places the error into the the "accept queue" by `RSVP` macro and then directs the `Parser` to stop parsing:

```
parser__->set_stop_parse(true);
```

In rule `Rconstant_defs_code` there is also double `|?|` use. There are 2 parsing contexts going on: errors from the thread call statement, and regular parsing whereby the thread call wasn't called or did not return a T. The C++ comments against each subrule explain what each error trap is doing. U will also notice that sometimes the returned error T's "auto delete" attribute is turned on so that as it is popped from the parse stack it is retired to heap heaven and another more specific error is created in its place and returned back to its calling grammar.

```
sf->p2__->set_auto_delete(true);
```

9.4 |+| and possible problem solving entourage

`|+|` use must ensure that the follow set is legitimate **i.e. the Tes in it are all there for reducing** or else O_2 will mutter:

```
Error Can't find parallel sym reduce in
      FSM id:  aaa state:  bbb token:  ccc subs:  ddd
```

Whether u used the `|+|` to capture errors or forgot to program for it, **error capturing is done by `|?|`** and not `|+|`. Your grinding my teeth Dave as i'm getting impatient with your misleading section title.

Without the `|+|`, u explicitly need to program for all, i said all returned legitimate Tes. Not using it

can incur an error due to omission. So use it to capture all acceptable Tes knowing that there will not be an error. On the other hand u could program against errors using a finite state automata approach within your syntax directed code. This was demonstrated in the "angled_string" example on page 93. Okay but what about a mixed bag of wild legits and illegitimate Tes? That was previously answered: **this is an ambiguous situation** and unfortunately **it requires an explicit declaration on all the legitimate Tes** while using the `|?|`.

Back to the above error message, the details are the `Parser`'s grammar state table snapshot for u to correct. The clues point within the troubled parsing sequence. The message is a leadin to how would we go about figuring out where to set the error trap(s)?

9.4.1 Dribs and Drabs or Dribbles for short

Let's look at the problem from the chickadee's perspective. A monolithic grammar starts parsing and it could call some threaded grammars that themselves call other threaded grammars. Here we have a nested parsing set of activities that could be running in parallel. Somewhere within this setting the parser's sensors go red and billboards its own "blue screen" of death. Now we'll review what is open to us to help solve this problem.

One could go back to the grammar's documents that details the overall lr state tables and grammar's rules and subrule sequencing pictorials. Though not covered until later in the book (u can have a peek if u like right now in "The jewels of O_2" chapter 12). It is one part in bringing together all the materials for error detection/correction assessment. But sometimes the level of grammar calls can be deeply nested wrinkling your brow with perplexity on "how did the parse arrive here?". To this end the O_2linker's document contains the social networking of grammars [2]. It's the who-called-whom bible with an indexed summary per grammar that focuses u to a smaller search space of grammars to deal with. Documents are static; they provide the framework of what's currently been programmed but they are not dynamic in the telling of what just happened. U have to play "what if parse games" to arrive at the troubled spot reported. So onto dynamic tracings. What is dynamic tracing? It's the potential to output various information streams using sensitivities in realtime. These potentials are built into O_2 and are continuously checked throughout its code base. They are normally dormant until a specific information stream is turn on. These potentials can be turned on/off anywhere by your code. I'll rephrase this; u can do it in your mainline, in called functions, monolithic and threaded grammars by syntax directed code as many times as u want. Dynamic tracing is open-ended to its when/what/how turn-ons. To do this, there are globally defined variables supplied by the `YACCO2_define_trace_variables();` macro. U must define these variables when using O_2's library or u'll get linker errors of unresolved symbols.

O_2 globally defines the file object `yacco2::lrclog` for tracing. It initially starts tracing to **1lrtracings.log** file to deal with the command line input errors: bad options or undefined grammar file to parse. After this it traces to **xxx_tracings.log** where `xxx` represents the grammar file name without its extension: ".lex". The trace file object **yacco2::lrclog** is available for your own scribbles and is equivalent to `std::cout` in use with the same output operations. In a parallel parsing environment, the only caveat is a mutex must be used to eliminate scrambled text intermixed by possibly 2 or more grammars logging to the file **at-the-same-time**. Logging to an opened trace file is equivalent to using C++'s `cout`. The lock and unlock mutex functions are globally defined in Yac$_2o_2$. Here's an example from a grammar subrule:

```
-> Rexample {
op
  using namespace yacco2;
  LOCK_MUTEX(yacco2::TRACE_MU); // rsvp
   lrclog << "An example outputted to the trace file" << std::endl;
  UNLOCK_MUTEX(yacco2::TRACE_MU); // let the others howl
***
}
```

Here's O_2's list of globally defined mutexes.

- yacco2::TOKEN_MU — token dispensor access.

[2] Another wait-and-see.

- yacco2::TRACE_MU — used to log tracing.
- yacco2::TH_TBL_MU — access thread dispatch table.
- yacco2::SYM_TBL_MU — symbol table access.

Only TRACE_MU and SYM_TBL_MU mutexes are open for your use while the others are internal to O_2's implementation. The symbol table mutex will be developed in functors. Yes there is a trigger mechanism to hook into the fetching of Tes to be remapped by your own symbol table. It is not too sophisticated but served a Pascal translator handling Pascal's scoping rules at the grammar level where the grammar rules controlled the search scope allowed by the symbol table. This will be detailed in "Pieces of thought" chapter 13 later. To recap, the error sleuth has the following forensic tools:

- Grammar documents —grammar and O_2linker documents reviewed later in the book.
- Log files: the tell tale diagnostics.
- Global variables control what O_2 traces dynamically.
- Post process the "Trace file" as it is text and not binary data. Text editor, *bash scripting*, *grep*, *awk*, or *split* are tools open to viewing and filtering its data. The traced records are canonical in their formats for customized filter scripts.

9.5 Head scratching when your parser abruptly ends

So `Parser` just gave u the torro sign:

```
''Error - Can't find symbol to shift in...''
```

Variations on the same error message lead-in

```
''Error - Can't find...''
```

indicates the current T is not within the grammar's current parse state table. Depending on the message a flood of parse state details accompany it like the grammar name, current parse state, the current T and its position. So what to do? Let's now look at the tracing capabilities to see what led to this state-of-affairs.

Bootstrapping O_2 by using itself to define itself brought out (exposed my learnings/weaknesses) the necessity of different tracing capabilities to debug and to verify itself. The tracing design uses globally defined variables where each trace type is independent of the other types and each trace type could be turned "on" or "off" anywhere by your code. I felt the speed bump was very minor compared to the openness to where and when these traces could be used.

The `YACCO2_define_trace_variables();` macro globally defines these variables for u. U include it in your program's global scope; see sample program in section 2.5.1 as an example of its use. Here are the global variables and their associated trace types:

- `yacco2::YACCO2_T__` — trace Tes when fetched.
- `yacco2::YACCO2_MSG__` — trace communication messages exchanged between each parsing work unit.
- `yacco2::YACCO2_TH__` — trace grammar's parse stack activity.
- `yacco2::YACCO2_AR__` — trace potential "accept queue" contents when arbitration is required.
- Trace mutexes acquire/release activities controlling various critical regions. When there's paranoia, this helps settle the illusions across the various critical regions.
  - `yacco2::YACCO2_MU_TRACING__` deals with the trace log file mutex activity. U must acquire the rights to its mutex `yacco2::TRACE_MU` before u can scribble to the log file. Then release it when finished or u'll be waiting for a long time.
  - `yacco2::YACCO2_MU_TH_TBL__` controls the thread table. This is strictly internal to O_2.
  - `yacco2::YACCO2_MU_GRAMMAR__` traces each grammar's mutex acquire and release. Again strictly internal to O_2.

For your knowledge, grammars exchanging messages have an agreed to set of responsibilities that require mutual exclusion control over their critical regions and wakeup messages to the thread library.

- `yacco2::YACCO2_THP__` — trace thread performance meanderings.
- `yacco2::YACCO2_TLEX__` — trace customized macros. Quips to yourself when things ain't going well.

Now the quantity of traced data can be overwhelming and so tracing output needed some fine-tuning against some trace classes and to allow specific grammars to trace themselves while leaving the other grammars to be trace dormant. Each trace class has a variable to turn on. regarding grammars to trace, the specific grammar's `fsm`'s `fsm-debug` variable must be turned on and also the thread trace class variable; yacco2::YACCO2::TH__ variable. For example to trace class Tes, set its variable to on:

yacco2::YACCO2_T__ = 1;

Yac$_2o_2$ has defined types of `ON` and `OFF` which are boolean equivalents to 1 and 0.

yacco2::YACCO2_T__ = yacco2::OFF;

To identify the type of trace being dumped, each trace class is prefixed to its outputted messages.

Again i repeat myself to remove any confusion when first trying out tracing. There are 2 files that contain the tracings: the initial command line parse that gets outputted to a hardwired **1lrtracings.log** file [3] and the grammar being compiled: **xxx_tracings.log** where xxx is the name of the grammar. Why the 2 files? Yes the run's output could all be placed into the first file covering both a bad grammar name inputted and its good tracings. But i wanted the grammar's tracing to be contained to its own setting making it easier for u to postprocess it rather than making a copy of the **1lrclog.log** file as it would get destroyed when O_2 was run again.

Well this is nice Dave for O_2 but what about my own parser product using Yac$_2o_2$'s library? These tracing facilities come from the Yac$_2o_2$'s library and so are germain to your own product: A freebie to u. The trace file object **yacco2::lrclog** is open for your own dealings. It keeps the 1st file open until u explicitly close it and reopen it with another file name. It is defined as an object of *std::ofstream*.

9.5.1 Fetching those Tes

```
yacco2::YACCO2_T__=1;// to trace or not to?
```

Every time a T is fetched regardless of work unit, it is identified. Here is a sample of its traced output from O_2's lexical stage:

```
YACCO2_T__::1::pass3.lex:: get_spec_token pos: 0
YACCO2_T__::tok_can token: / *: b3f6f0 pos: 0 enum: 55"/"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 1
YACCO2_T__::1::pass3.lex:: get_spec_token: returned token /
  pos: 0 enum: 55"/"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 1
YACCO2_T__::1::pass3.lex:: enum: 55 "/" pos: 0
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 1
YACCO2_T__::1::cweb_or_c_k.lex token*: b3f6f0 enum: 55 pos: 0 "/"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 1
YACCO2_T__::1::cweb_or_c_k.lex get_next_token:: pos to fetch: 1
YACCO2_T__::tok_can token: * *: b3f728 pos: 1 enum: 50"*"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 2
YACCO2_T__::1::cweb_or_c_k.lex get_next_token:: pos: 1  enum: 50 "*"
  token fetched*: b3f728
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 2
YACCO2_T__::1::cweb_or_c_k.lex get_next_token:: pos to fetch: 2
YACCO2_T__::tok_can token:   *: b3f760 pos: 2 enum: 17" "
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 1 GPS CHR POS: 3
                    .... elided trace
```

[3] What an idiotic file name. Well you're seeing my penchant for naming but 1 comes before "a" in the listing of a file directory.

```
YACCO2_T__::1::cweb_or_c_k.lex get_next_token:: pos: 1555  enum: 21 "
" token fetched*: b54b18
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 31 GPS CHR POS: 3
YACCO2_T__::1::eol.lex token*: b54b18 enum: 21 pos: 1555 "
"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 31 GPS CHR POS: 3
YACCO2_T__::1::eol.lex get_next_token:: pos to fetch: 1556
YACCO2_T__::tok_can token:
 *: b54b50 pos: 1556 enum: 18"
"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 31 GPS CHR POS: 4
                    .... elided trace
YACCO2_T__::1::eol.lex get_next_token:: pos: 1557  enum: 110 "f"
 token fetched*: b54b88
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 32 GPS CHR POS: 1
YACCO2_T__::1::identifier.lex token*: b54b88 enum: 110 pos: 1557 "f"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 32 GPS CHR POS: 1
YACCO2_T__::1::identifier.lex get_next_token:: pos to fetch: 1558
YACCO2_T__::tok_can token: s *: b54bc0 pos: 1558 enum: 123"s"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 32 GPS CHR POS: 2
YACCO2_T__::1::identifier.lex get_next_token:: pos: 1558  enum: 123 "s"
 token fetched*: b54bc0
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 32 GPS CHR POS: 2
YACCO2_T__::1::identifier.lex get_next_token:: pos to fetch: 1559
YACCO2_T__::tok_can token: m *: b54bf8 pos: 1559 enum: 117"m"
 ::GPS FILE: /usr/local/yacco2/compiler/grammars/eol.lex GPS LINE: 32 GPS CHR POS: 3
                    ... elided trace
```

All its traces are 2 line entries apart when the `get_next_token` physically fetches the next T where it identifies itself. This is shown in the first line above. Each canonical trace output identifies itself by its assigned thread id number and the grammar requester, position requested within the token stream (remember token sequences start from 0) and its returned T in memory terms as a memory usage check, enumeration id and literal content. The source file coordinates are also displayed by line number and character position allowing u to use a text editor to go directly to the troubled spot. O_2 reports all files sourced which includes nested include files. Note the "command line" trace in **1lrtracings.log** has no associated source file and so `GPS` coordinates are reported as 0.

U might question why each grammar requester has a unique thread id assigned to it. Like recursive descent parsing, threaded grammars can be nested and recurse on itself or have parallel activities where equivalent grammars could be working for a different requester at-the-same time. This identity distinguishes their clones.

This trace type can be very verbose if it is left turned on for the entire parse stages. It can knock out your favourite text editor by blowing up on memory consumption. When this happens use the *split* utility in UNIX to partition the trace file into smaller file chunks. It might be your only way to orient yourself to fine-tune future bouts of tracings by initially observing all fetched Tes starting from the very beginning of your lexical parsing. The use of a trace variable could be conditionally coded within your program to start and stop tracing anywhere as many times u deem fits the situation. For example:

```
if(parser__.get_current_pos() == 32000)
  yacco2::YACCO2_T__ = 1;
                    ...
```

I used the explicit namespace in the above example to get at the variable. U are open to use C++'s namespace state:

```
using namespace yacco2;
 ...
if(parser__.get_current_pos() == 32000)
  YACCO2_T__ = 1;
                    ...
```

Or u can focus on a specific stage in parsing like semantics when this trace type is used. Each trace type is independent of the other types. So choose your strategies without any concern on the other types.

9.5.2 Communication messages exchanged between grammars

```
yacco2::YACCO2_MSG__ = 1;
```

This tells the tale between caller and the called. The what and where of the token stream between the requester and its servant(s), and the work progress: starts and stops. This trace is neat as it logs every communique between the coordinate sets of parsing units: requester/requested. Here is a sample of what gets traced.

```
YACCO2_MSG__::1::o2_lcl_opts.lex --> start threads
YACCO2_MSG__::1::o2_lcl_opts.lex calling PROC:: --> before procedure call
YACCO2_MSG__::PROC::1::PROC_TH_o2_lcl_opt start parsing
YACCO2_MSG__::1::o2_lcl_opt.lex requesting parser*: fffffd7fffdcec60
YACCO2_MSG__::PROC::1::o2_lcl_opt.lex Caller's # threads to run:: 1
 Caller's # active threads: 1 Self # competing threads: 1
YACCO2_MSG__::PROC::1::o2_lcl_opt.lex passed token*: b3ef80"-" pos: 0
 ::GPS FILE: yacco2cmd.tmp GPS LINE: 1 GPS CHR POS: 1
YACCO2_MSG__::1::o2_lcl_opt.lex:: called thread reducing thread
 active count of caller thread 1::o2_lcl_opts.lex active thread count::1
YACCO2_MSG__::1::o2_lcl_opt.lex:: called thread after reducing thread
 active count of caller thread 1::o2_lcl_opts.lex active thread count::0
YACCO2_MSG__::PROC::1::PROC_TH_o2_lcl_opt ==>procedure finished working
YACCO2_MSG__::1::o2_lcl_opts.lex returned from PROC:: result: 1
YACCO2_MSG__::1::o2_lcl_opts.lex --> start threads
YACCO2_MSG__::1::o2_lcl_opts.lex calling PROC:: --> before procedure call
YACCO2_MSG__::PROC::1::PROC_TH_c_comments start parsing
YACCO2_MSG__::1::c_comments.lex requesting parser*: fffffd7fffdcec60
YACCO2_MSG__::PROC::1::c_comments.lex Caller's # threads to run:: 1
 Caller's # active threads: 1 Self # competing threads: 1
YACCO2_MSG__::PROC::1::c_comments.lex passed token*: b3f028"/" pos: 3
 ::GPS FILE: yacco2cmd.tmp GPS LINE: 1 GPS CHR POS: 4
YACCO2_MSG__::1::c_comments.lex:: called thread reducing thread
 active count of caller thread 1::o2_lcl_opts.lex active thread count::1
YACCO2_MSG__::1::c_comments.lex:: called thread after reducing thread
 active count of caller thread 1::o2_lcl_opts.lex active thread count::0
YACCO2_MSG__::PROC::1::PROC_TH_c_comments ==>procedure finished working
YACCO2_MSG__::1::o2_lcl_opts.lex returned from PROC:: result: 0
                    .... elided trace code
```

Please note this trace type is not local to a grammar but across all parsing spectrums: monolithic, threaded, optimized grammar procedure calls be it automatic or manually called by some syntax directed code. Yac$_2$o$_2$ documents the message protocol. Lightly the interchange of messages begins with the requester grammar broadcasting message detailing on how many grammar threads being requested to work. Part of the message protocol details when each requested thread reduces the active thread count before a wakeup message is returned to the requesting grammar. This message happens when the thread is finished parsing.

In the above sample, there was a optimized procedure call rather than calling a thread and going to sleep until the prince does his magic; given more than 1 thread launched, it is the last thread completed be it successful or not to awaken the snoring ogre. Upon startup, each requested grammar details this startup message in its own terms. Following this are the requested grammars progress messages back to its caller. Mixed into this is the possible messages exchanged by the requested grammar to its own requested grammars. Eventually the original requester will be re-activated.

This trace type came about due to the 5 computer scientists dining problem. When all activities go into a wait state it certainly revs up your ire. At least this trace documents the work assignments and progress. So far the "after u syndrome within a circle" seems to behave itself.

9.5.3 Dump of a grammar's parse stack

```
yacco2::YACCO2_TH__ = 1;
```

U get all the gory details each time the `Parser`'s parse stack changes: states and stacked symbols. This is the slow motion play of each parse activity. So u thought the T tracing was noisy. U should see this trace across a few actively tracing grammars. It will definitely command u to refine your post evaluation skills on the output log file. This is why a grammar is traced only if its `fsm-debug` variable is on. U can turn on a grammar by retranslating it after changing its `fsm-debug` variable to **true**. Or change the grammar's emitted "fsm cpp" code contructor to **true** and recompile. Here's `angled_string.lex`'s *fsm* constructor's `Cangled_string` turned on to trace:

```
Cangled_string::
Cangled_string()
  :yacco2::CAbs_fsm
    ("angled_string.lex"
    ,"1.0"
    ,"17 Juin 2003"
    ,true //<=== turn on tracing: original value was false
    ,"Angled string lexer: < ... > with c type escape sequences."
    ,"Wed Mar 24 11:29:14 2010 "
    ,S1_Cangled_string){

  ddd_idx_ = 0;
  ddd_[ddd_idx_] = 0;
}
```

Section 6.5 gives u a sample of its output.

9.5.4 Arbitrator, u think i...

```
yacco2::YACCO2_AR__ = 1;
```

When a grammar's arbitration procedure is called, it dumps the grammar's potential "accept queue" contents. Remember a grammar can have many arbitration points with its grammar. Multiply that against all the other grammars that could be parallel dancing. It allows u to see what was arbitrated on and why the Arbitrator missed his calling and delivered a wrong T. Or should i say against your jurisprudence?

9.5.5 Semi-random walks in thread performance

```
yacco2::YACCO2_THP__ = 1;
```

A simple way to observe how threads get called with start/stop call costs. I used this to measure single cpu performance against a multi-cpu chipped system. It provides the meandering traces of how multi-threading is dispatched and who is executing while other threads are waiting for their turn to execute. U shouldn't need to use this trace type but it's there for your calling/curiosity.

9.5.6 Acquire/Release of mutexes of various trace classes

These variables are used to ensure no mutex lockups by tracing their mutex activities — a smougasborg of the five computer-scientists dining problem.

- Variable yacco2::YACCO2_MU_TRACING__ deals with the tracing output. It is an observer on how the operating system deals with inter-activities of parallel threads tracing their output to the same file. U wouldn't want a mixed college of one thread's output concatenated with another's parts: atomic divisions of info is a must.

- Thread launching and reuse demands a mutex to access the thread dispatching table. The thread table is yacco2::YACCO2_TH_TBL__. As parallel activities are happening with multi-core cpus, a thread's launching requirements of other threads requires this exclusitivity.
- yacco2::YACCO2_MU_GRAMMAR__ allows one to observe the actual acquire/release mutex per individual grammar.

Tell tales on my paranoia into mutual exclusion implementing O_2 and threads.

9.5.7 Customized user emergencies macro traces

```
yacco2::YACCO2_TLEX__ = 1;
```

The automatic trace depends on whether the grammar's `fsm-debug` variable is turned on. This limits the verbosity. Here is the extract from Yac$_2o_2$'s library document regarding roll-your-own tracing. They are always traced when the TLEX variable is on regardless of context:

These are "roll your own" macros for when the going gets rough and tough, and you don't have a bloody clue. At least you can leave some tracks of your own originality. Good luck and this is said with sincerity as I needed them once.

The grammar writer can put them inside the grammar's syntax directed code directives. They basically give the parallel details on the thread, critical region etc. The passed in parameter is what the grammar writer wants to display. Basic, crude but may be helpful. Before going this route though, the other macro traces should be adequate. Other forms of tracings using **yacco2::lrclog** or **std::cout** are rudimentary but also effective. There are 2 macros:

```
sstrace_parallel_supportss(ssPROC_NAME)
sstrace_parallel_support_envss(ssPROC_NAME)
```

Each macro's parameter is usually a literal message to be traced within the parse units run context. As it is a text substitution, C++ code could be given. The only warning is the parameter gets placed between surrounding `<<` operators. `sstrace_parallel_support_envss` prints the interplay between the requesting grammar and the launched thread. They basicly save your fingers and head to figure out how to trace the runtime details per grammar like its name, thread id, etc.

9.6 Yin/Yang to parsing stoppage

Covered was the unexpected aborts and how to solve them: The Yin. But parsing stoppage can be under your own programming dictates. The Yang to stoppage. Why would one want to stop a parse? Previously described by programming example was parsing by a finite state automata within a subrule's syntax directed code. It was done for speed: the pop corn effect by a bottom-up lr(1) deterministic parser was too expensive when applied to lexical entities. Still it had to deal with errors discovered and had to stop parsing. So let's look at what can be done. There are 2 programmed ways to stop a parse:

- Stop. `set_stop_parse(true);`
- Abort. `set_abort_parse(true);`

Here are some extracted "notes to myself" from Yac$_2o_2$'s documentation.

9.6.1 Some comments on stopping a parse by syntax directed code

Apart from the don't do anything approach, the grammar writer can talk to the parser and dictate his intentions. The 2 methods open are abort-the-parse or stop-parsing. The abort-the-parse action allows the thread to stop without any T returned to the caller grammar or use the `failed` directive to last-chance return an error T back to the caller. The stop-parsing approach returns a T thru the "accept queue" back to its caller but does not want to continue the complete parse through to its "start rule". It just short-circuits the overall grammar's parsing action. Remember that if the parse has been successful "why complete the

parsing thru to the start-rule?". Depending on your local grammar logic this might be the most expedient way to program. Here are the 2 methods to do this:

```
set_stop_parse(true);
set_abort_parse(true);
```

What about the reducing of this subrule? Well it occurs, as entry into the syntax directed code that contains the grammar writer's code to execute these statements are kosher reducing conditions. So why the "abort-parse" versus "stop-parse" difference. "stop-parse" should contain the `RSVP` macro that enters the returned T into the calling grammar's "accept queue". The "abort-parse" normally does not contain this action. The odd case is when the grammar is monolithic and wants to register the error in the "error queue" for post evaluation after the grammar halts. The `ADD_TOKEN_TO_ERROR_QUEUE(error T);` macro uses the assigned error queue container within a rule/subrule's context while macro `ADD_TOKEN_TO_ERROR_QUEUE_FSM(error T);` is done within the grammar's `fsm` context: for example, `failed` directive or possibly one of your own `fsm`'s methods. The harder way is fetch the error container's address and use its `push_back(T);` method. U might need to go this route if u have multiple error containers[4] and u want to place it outside the error container given or not given to the grammar.

```
CAbs_lr1_sym* er = new Err_bogus_symbol;// create the error T
GBL_error_que->push_back(*er); // place it in the global error container
```

9.7 More extracted notes on "Error detection and handling"

I include these ramblings as these give another perspective on this subject: my mire waddling. There is some repetition but i feel its mutterings should strengthen your understanding or your utters tarnishing my thoughts. Consider it a bloated summary with improvisations.

9.7.1 Error detection and handling

Let's review how this can be done. Within a grammar's production there are points where an invalid symbol could arrive. If one does not program for it, the parser will go kapout. So what are the options open to a grammar writer? First there is a "`failed`" directive in the "fsm" construct that will field aborted parses. It is the last chance to deal with errors in a rather insensitive way. If there are many contexts within the grammar that could go wrong then this approach is too insensitive to be specific about the context's error point. Though the errant current token is available to report on, what was the inappropriate context that threw it? Well u could try to figure it out from the remnants on the parse stack.

To deal with specific error points, the `|?|`, `|+|`, and `|.|` symbols can catch errant tokens, or one can be very specific in specifying the errant T to catch. This last option can be very daunting when one has 500+ Tes to deal with and lets be honest not really appropriate. This was why i introduced the meta-terminals `|?|` and `|+|`. To catch a rogue and associate syntax directed code to handle the situation, these symbols MUST be within prefix subrules where they are the last symbol in the subrule's symbol string. What does this mean? Having a string of symbols where these catch T symbols are buried within a larger symbol string means the subrule's containing these symbols will not be executed as its sentence has not been completely recognized. For example:

→ a `|?|` b — will not handle the error at the `|?|` point

→ a Rqueshift b — will catch the problem

Cuz rule `Rqueshift` → `|?|` will reduce but remember to accompany it with the `op` directive's dictates to the `Parser`.

9.7.2 Caution: meta-terminal shifts ranking: 1 and a 2 and a 3 — the welshing of `|?|`, `|.|`, `|+|`

Following the current T, the `|?|` symbol is checked first for its presence within the current parse state followed by the `|.|` symbol as it is normally used to get out of a quasi-ambiguous parse. The `|+|` aka wild shifter is the last to be checked in the parse state. It is their presence within the parse state that activates

[4] I'm huffing and puffing but u might have discriminated between error types? warnings versus severe?

their use. The `|?|` is an error statement and was my reason to put it at the head of the conditional shifts. So watch your shifts as this could catch u like me. Remove 1 of the 2 competing shift symbols against `|?|`: `|+|` or `|.|`. For the moment i have not issued an error message on this situation.
Post comment on this: Dave it is not too clear. Ambiguity between `|?|` and `|+|` within competing subrules will be reported as "not a LR(1) grammar". As `|?|` is tested first for its presence the `|.|` **will never get executed** if u're using it to shift out of an error situation.

9.7.3 Dictate no 1: Last symbol in subrule's symbol string must be the "catcher in the Error"

Make sure your error catch point has `|?|` or `|+|` as its last symbol within the symbol string and let your syntax directed code decree the error escape route to be taken. Yeah that's fine but what if the symbol string to be recognized contains many catch points? Just make each symbol string segment a separate rule with the error code catch point being the last symbol in the string competing with its legitimate accepted T symbols and use these rules within another rule's subrule as part of its symbol string to be recognized! The lr algorithm is a collection of various symbol string configurations per state in various accepted T points along their parsing. So by transitive closure these prefix rules get included in the state to be recognized along with the other similar prefix symbols. When the prefix rule's "rhs" boundary is recognized, depending on the "error catcher" used, the reduce will fire either in good form or as an error.

9.7.4 What to do when an error is detected?

For now i have not thought out error correction strategies though i am marginally aware of the backtracking techniques. I will now discuss current programming options open to the grammar writer. Depending on the context, the thread could abort which is the most drastic. This takes place when no error catching is programmed and O_2 issues a runtime message on the aborted grammar with its run stack goodies. This might be okay to get things going but isn't too appropriate within a production environment. Well the catch points have 2 programming options available:

1. Return an error token back to the calling grammar and stop the active grammar parsing.

2. Abort the parse and field it using the "`failed`" directive to return an error T.

Point 1 should be your main course of action. That is both macros `RSVP` and `RSVP_FSM` return a T back to the calling grammar through the accept queue facility as if the parse was successful. This is what point 2 does using the `RSVP_FSM` macro as its execution is within the "fsm" context of the grammar and not the reducing rule. The calling grammar can then field this returned T specifically or use the two meta-terminal `|?|` or `|+|` to deal with them. They are allowed in any subrule symbol string context: thread calls where its returned T can be one of these symbols, and the regular subrule symbol string.

9.7.5 Source file pinpointing where the error occurred

Built into O_2 is the facility to tag each T with its appropriate source file's GPS — filename, line number, and character position. These coordinates are used to print out the errant source line with an arrow underlining the errant source token. So when an error T is created, use of the `set_rc` and variants allows one to pinpoint the error T against the GPS's source file T. Have a read on "Abstract symbol class for all symbols" — `CAbs_lr1_sym` section 3.6.

9.7.6 Some subtleties on making the errant T fire off the "error catching syntax directed code"

Let me pose a question: What happens when the errant T is not in the lookahead set to reduce that subrule? Well it will not get executed! Ugh. This is just not acceptable Dave. Well to the rescue is the `|?|` symbol. It is not in the token stream but represents an errant situation. So where is this errant T placed? When one enters the subrule's syntax directed code segment, all its subrule's elements have been shifted onto the parse

stack where this last errant symbol is represented by `|?|`. But the `|?|` symbol **does not advance past the errant T** as in regular parsing. So what does this mean? The current errant T is also the lookahead symbol for the reduction. But wait what if this T is not in the lookahead set to reduce this subrule. Well i made this type of reduce a lr(0) context: no lookahead symbol required to reduce the subrule.

To get at the current elements on the normal parse stack, O_2 emits within each subrule's syntax directed code a "stack frame" mapping each subrule's symbol to a variable "`sf->pxx__`" where xx is the symbol's string position. The difference between `|?|` and `|+|` is `|+|` depends on the lookahead set to reduce. If u are not depending on the lookahead set, then `|+|` should be as in the finite state example for the good and the bad. If u are relying on the `Parser`'s actions to deal with an error rather than your own code `|?|` is the only option to error catching.

Now what then is the advantage to using `|+|`? As it's a wild symbol shifter, it really lowers the grammar's parse tables sizes and eases the grammar writer's typing. One can test its under-its-hood symbol's T's enumerate value and do your own programming actions to stop consuming wildly allowing the grammar to continue parsing up to the "start rule", or stop parsing completely by returning a T back to the calling grammar not continuing to parse to the start rule.

9.7.7 Dictate no 2: Games on returning the new lookahead T back to the calling grammar

U can play games with resetting the new lookahead T that is passed back with its `RSVP` T companion within the accept queue. This is what happens when just 1 T is returned: the lookahead T is the parse stream continue point and also its contents to set the calling parser's current token to continue with. As an aside why use the returned lookahead's T contents instead of just resetting the continue T from the token stream's container using the lookahead token position? Well u could also remap the current token into another T type due to say a symbol table remapping — like Pascal and its "const-id", "function-id" as described in the railroad diagrams of "The Pascal Reference Manual". The remapping facility is open for use via the "Table lookup functor" facility. The following methods adjust the parser's token stream:

```
override_current_token_pos(symbol,position);
override_current_token(symbol);
reset_current_token(position);
```

In a dual competing threads situation where each grammar have accepted their parse and are returning their booty to the calling grammar, the calling grammar must use arbitration to select the T gift and sets its parse stream accordingly and the balance in the "accept queue" are so-to-speak thrown away. Of course the **arbitration** facility is programmed by the compiler writer when 2 or more successful threads are returning their booty back to the calling grammar. Normally this does not occur as there is just one thread that will report its findings but this city is built on rock and nondeterminism. So in a subset/superset competition or an accept and error combo it is quite acceptable and open to the arbitrator's choosing. Forgotten arbitration code will be regurgitated by the O_2 library in message form for your fixing.

The one caveat to watch for is: What is the current token and its position in the parse stream when it enters the subrule's syntax directed code? `|?|` still has the errant T as its current T and *to reset back to the previous T* u only subtract 1 from the current token position while `|+|` demands 2 be subtracted as the current T is the new lookahead T and subtracting 1 only gets the last T which is the T covered by the `|+|`. So u've been warned.

9.7.8 Warning no 3: if `|+|` being used, don't forget to turn it off

This symbol is voracious: eats and eats everything in its path. So u can arrive at trying to eat the "end-of-the-parse-stream" "eog" symbol forever... O_2 guards against this but is rather abrupt in its message to the grammar writer and immediate stoppage of the parse. So u'll see in some of the suggested below grammars the `set_use_all_shift_off` method being called to get out of this perpetual bingeing and possibly continue up the parse chain to the "start rule". Here is a list of some O_2 grammars having error handling and premature parse stoppage to learn from.

1. Command line parsing: monolithic grammar `o2_lcl_opts.lex` calls the options assessment thread grammar `o2_lcl_opt.lex`.

2. Parsing of a threaded grammar's lookahead expression. Look at grammar `la_express.lex` use of `set_abort_parse(true);` to abort a parse.

3. `c_string.lex` shows how within a subrule to implement a finite state automata by calling its input token container, and how to stop the grammar from parsing after the finite state automaton has completed. It is a delayed reduction as the subrule is being reduced but the finite automaton within it continues consuming the grammar's input Tes before it eventually stops.

Point 1 gives an example of how the `failed` directive in the called thread `o2_lcl_opt.lex` is programmed and `set_stop_parse(true)` use in the calling monolithic grammar `o2_lcl_opts.lex`. Grammar `pass3.lex` and point 2 give more examples on monolithic use to aborting. Point 3 also shows programming use of the "`set_abort_parse(true)`". For the really curious, why not use the `ls -1 *lex | xargs grep` combo to settle your appetite against O_2's grammars.

9.7.9 The last word, amen and happy parsing

Remember that the normal flow of errors should finally be placed into the "error queue" by a monolithic grammar and then the "error queue" post-processed to report its findings. Macros `ADD_TOKEN_TO_ERROR_QUEUE` and its variant `FSM_ADD_TOKEN_TO_ERROR_QUEUE` allow u to do this. `pass3.lex` gives lots of examples and O_2's program shows its way of post-verbing the troubles. And with all this error stutter, each grammar does a post-execution grammar cleanup on current parsing for the next round of their calling. Again what does this mean? A semi-abort was done just to stop its execution leaving the grammar's parse stack dirty. But each grammar does a cleansing for its next round of calling either by "procedure call" if no nesting calls of itself is occurring or by the heavy thread call. Hygiene is important so the cat washes itself for the next eating [5].

9.8 Post evaluation on errors: Truely the last word

Though this is not pleasant at least providing semi-intelligent clues to the stoppage is civil. In my parsing bouts after a parse stage has finished, a check is done on the "error container". Having content, how should u go about addressing your audience and the telling of bad news? Well why not use a grammar to process its contents? Though the example does not support multilingualism, I have thought about it by using a message table or maybe threaded parallelism? It will get addressed after the book is completed along with the not so slim bodied characters: some variant form of unicode.

The following subsections discuss an example taken from O_2 using a grammar/parser combo to walk its contents and to use the subrule's `op` directive to output the bad news. This is just one way to do it. The other is to explicitly iterate thru the container and report it by some subroutine. Your taste prevails. Detection, reporting, and the reporter are the 3 parts to the problem explored.

9.8.1 Detection

Detection is whether the "Error container" has content. This starts at the 1st monolithic grammar level and thereafter. What does this mean? Remember that the very first monolithic gammar is the command line process then followed by the parsing of the language inputted.

```
if(Error_queue.empty()!=true){
 DUMP_ERROR_QUEUE(Error_queue);// Error_queue is global container
 return 1;
}
```

[5] Is this a Cheshireism?

Please read **/usr/local/yacco2/docs/testout.pdf** program where it discusses the kosher way to test for error posting. One should check the Parser's returned code as to whether error reporting should take place. It's not 100% correct as it indicates that the Parser erred but possibly no Error T was placed into the error queue/container! A programmer ommission or miss use of what queue the error was being placed into? And so the error detection process is two level: Parser error detection at the monolithic grammar level followed by Error container content check: if empty, an ommission error message should be outputted for correcting your compiler code. Here is the **testout** program code snippet with literate programming.

```
using namespace NS_pager_1;
tok_can<std::ifstream> cmd_line(To_parse_file.c_str());
Cpager_1 pager_1_fsm;
Parser testout_parse(pager_1_fsm,&cmd_line,&P3_tokens,0,&Error_queue,&JUNK_tokens,0);
if (testout_parse.parse() == Parser::erred){
std::cout << "===============>ERROR OCCURRED" << std::endl;
  @<if error queue not empty then deal with posted errors@>;
}
```

The first example above works but can be open to a subtle error due the compiler writer's inappropriate error posting to the **Accept** queue. This confusion can happen as Error posting is passed back to the calling grammar thru the "Accept queue" and not the "Error queue" by a threaded grammar. It is the one and only **monolithic** grammar that posts the error into the Error container and **not to the Accept** queue. At least the returned code test indicates a Parser error occured clueing u to your ommittance in error posting. One should still test the Error container for posted errors and throw a derogatory compiler writer ommission message when empty. So be warned. O_2's code uses the "Error container" test as the Error posting was properly done.

9.8.2 Reporting

I think it is neat that a parsing activity is being used to report on erroneous parsing activities. It's a nice way to deal with it without too much cost to get it going. Use of the `|+|` will always capture and report on the forgotten Tes until u specifically add a subrule to handle it.

Also notice that some errors have more than 1 symbol used to report on within its subrule. For example the subrules handling `"file-inclusion"` errors. U are open to build up streams of error Tes per error message. Each error token stream can be viewed as a statement within your error language. Yes i said language. Most people program or think of the error stream equivalency to lexical parsing: a one dimensional spectrum/point or as a finite automata. Why not evolve to the next stage in error handling as in one of the other 3 grammar classifications? Of course O_2 is a subset of deterministic grammars but with threads/nondeterminism where does it now fit within this context?

```
extern void DUMP_ERROR_QUEUE(yacco2::TOKEN_GAGGLE& Errors)
{
 using namespace NS_yacco2_k_symbols;
 using namespace yacco2;
 /**
 * two eog added to container: this is the proper way
 *  1st for the shift operation in the 2 Start rule's subrules
 *  2nd acts as the lookhead for the accept operation
 *
 *  If eog not added, the container will still issue eog
 *  due to end-of-container reached.
 **/
 Errors.push_back(*yacco2::PTR_LR1_eog__);
 Errors.push_back(*yacco2::PTR_LR1_eog__);

 using namespace NS_o2_err_hdlr;
 Co2_err_hdlr fsm;
 Parser pass_errors(fsm,&Errors,0);
 pass_errors.parse();
}
```

9.8.3 Don't shoot the messenger o2_err_hdlr.lex

It outputs the errors into lrclog log file and to the console via cout. Sophisticated it is not. Notice how easy it is to identify the specific error by its literal. The |+| captures the unspecified Tes balance to report on. I included only the first cout statement that duplicates its lrclog statement. Thereafter for space it has been removed.

```
/*
FILE: o2_err_hdlr.lex
Date:    8 Jul 2003
Purpose: grammar to process errors from yacco2
Conduit: none
Change: 11 Mar 2004 - replace tab with space so that
                          displaced error lines up properly
*/
/@
@** |o2_err_hdlr| grammar.\fbreak
Grammar that processes and prints the posted errors from \Yacco2.
@/
fsm
(fsm-id "o2_err_hdlr.lex",fsm-filename o2_err_hdlr
,fsm-namespace NS_o2_err_hdlr
,fsm-class Co2_err_hdlr
,fsm-version "1.0",fsm-date "8 Jul 2003",fsm-debug "false"
,fsm-comments "Logic sequencer: Print out errors from \\O2.")
@"/yacco2/compiler/grammars/yacco2_T_includes.T"

rules{
Ro2_err_hdlr (){
  -> Rerrors eog
  -> eog
}

Rerrors (){
  -> Rerror
  -> Rerrors Rerror
}

Rerror (
lhs{
  user-declaration
  void error_where(CAbs_lr1_sym* E_sym){
    std::string& ext_fle =
       yacco2::FILE_TBL__[E_sym->tok_co_ords__.external_file_id__];
    std::string line_of_data;
    std::ifstream ifile;
    ifile.open(ext_fle.c_str());
    if(ifile.good()){
      yacco2::UINT lno(1);
      yacco2::UINT dlno(E_sym->tok_co_ords__.line_no__);
      for(;lno<=dlno;++lno){
       getline(ifile,line_of_data);
       if(lno == dlno) break;
       line_of_data.clear();
      }
    }

    std::string space(" ");
    std::string::size_type f = line_of_data.find_first_of('\t');
    for(;f != std::string::npos;){
```

```
     line_of_data.replace(f,1,space);
     f = line_of_data.find_first_of('\t');
    }

    yacco2::lrclog << "Error in file#: "
           << E_sym->tok_co_ords__.external_file_id__
           << " \"" << ext_fle.c_str() << "\"" << std::endl;
    yacco2::lrclog << line_of_data.c_str() << std::endl;
    std::cout << "Error in file#: "
           << E_sym->tok_co_ords__.external_file_id__
           << " \"" << ext_fle.c_str() << "\"" << std::endl;
    std::cout << line_of_data.c_str() << std::endl;
    for(int pos = 1;pos < E_sym->tok_co_ords__.pos_in_line__;++pos){
     yacco2::lrclog << ' ';
     std::cout << ' ';
    }
    yacco2::lrclog << '^' << std::endl;
    yacco2::lrclog << "\tfpos: "
           << E_sym->tok_co_ords__.rc_pos__
           << " line#: " << E_sym->tok_co_ords__.line_no__
           << " cpos: " << E_sym->tok_co_ords__.pos_in_line__
           << std::endl;
    if(E_sym->tok_co_ords__.who_file__ != 0){
      yacco2::lrclog << "\twho thru it: "
           << E_sym->tok_co_ords__.who_file__
           << " line#: " << E_sym->tok_co_ords__.who_line_no__
           << std::endl;
    }
    ifile.close();
  };
  ***
}
){
  -> "nested files exceeded" {
  op
    error_where(sf->p1__);
    yacco2::lrclog
      << "\t" << sf->p1__->id__
      << " nested number exceeded: "
      << sf->p1__->nested_cnt() << std::endl;
  ***
  }
        ... elided code
  -> "file-inclusion" "bad filename" {
  op
   error_where(sf->p1__);
   yacco2::lrclog << "\t" << sf->p1__->id__;
   yacco2::lrclog << " " << sf->p2__->id__ << std::endl;
  ***
  }
  -> "file-inclusion" "no filename" {
  op
   error_where(sf->p1__);
   yacco2::lrclog << "\t" << sf->p1__->id__;
   yacco2::lrclog << " " << sf->p2__->id__ << std::endl;
  ***
  }
        ... elided code
  -> |+| {// catch balance of errors + report
```

```
  op
   error_where(sf->p1__);
   yacco2::lrclog << "\t" << sf->p1__->id__ << std::endl;
  ***
  }
}
}// end of rules
```

U could use the T filter mechanism by dynamically generating its filter set members depending on say command line parameters. Using the container/filter combo with parsing makes your task that much easier in reporting a subset of error findings.

9.9 Views from your IDE debugger

Let's now talk interactivity. I don't have too much to say on debugger techniques but views on my own self observances: self-watching-self without the recursion? The previous exposé said in the french sense as i'm Quebeçois? had tracing to a log file with post evaluation required. I'm an advocate on using any type of tool to solve a problem. Linear problems can be daunting but add parallelism and u've 2-in-oned your problem space. All emitted grammar code is C++. The lr(1) tables are standard "c" structures. U should not have to deal with the tables. Their generated documents should be adequate unless u feel they are buggy or u're going to remap them dynamically.

So what type of source code debugging can u do? This depends on your compiler/hardware platform. I've used local debuggers on all the platforms ported to. Setting the break point at a source line is wonderful particularly when dealing in parallel activities. It allows u to view the local variables of the thread and possibly global variables values. For myself just stopping at the breaks points is assuring. All the debuggers displayed local variables for my kosher eyeballing. Sometimes pointers had to be chased like trees or passed in parameter references or the `fsm` grammar object. I try not to spend too much time debugging thru-a-looking-glass due to what i call constricted scopes: u gain localness at the expense of generalness: dhem trees but what forest? The local point-of-scope can be too interactive/debugging in attempts. I find reading a hardcopy of a program particularly in the "Literate Programming" genre provides global/local scoping interplay while wrestling with the problem. It allows me to pencil out new rearming strategies to solve the problem be it in combination with interactivity and tracing. To each one's own tastes, or upbringing said as i started with punched cards rather than a 21 inch multi-coloured set of screens having panoramic windows opened in various contexts.

The one annoyance is "what is an abstract object's real structure and how to get at its augmented types?". This can be a bit trying unless u can dynamically cast the object into a inherited type by your debugger or u need to add debug code by casting the abstract object into its real self and do the compile/link/run cycle to view its augmented values.

Now on to the various contexts.

9.9.1 Setting your territorial boundaries

Here are the typical tracing contexts:

- Main program.
- Global functions.
- Grammars.
  - `xxx.cpp` grammar's `fsm` definition, and the grammar's rules/subrules definitions with their syntax directed code where xxx is the grammar's name.
  - `xxxtbl.cpp` grammar's lr(1) tables.
  - `xxxsym.cpp` grammar's thread and procedure definition, and the grammar's `reduce_rhs_of_rule` definition that deals with rule/subrule reduce operations.

- Yac$_2o_2$'s library. If the debug flavour is available on your system, u must link to it to be able to break point on it source lines. It would be found in `/usr/local/yacco2/library/lib/Debug`. This is platform/compiler dependent on its generated debug symbols table. U shouldn't need to break on its source code. The dynamic tracing should be adequate like T fetching and a grammar's parse stack.

9.10 Toilet training your Tes

So what is this talk about hygiene and the Tes? Let's look at this question from a transactional perspective. A grammar calls grammars that calls grammars. Along the way the chain of activity is broken but partially tree build activity has taken place. Give me an example Dave as this is too general. U are parsing a Soap[6]/Xml tagged language and one of the elements is bad: could be the paired tagged label or its data. But the previous elements have been recognized and meta Tes generated for future reference: the tree is being built as the xml structure is being recognized. If these partially created Tes are not attended to, u could have memory leaks. So how can u recycle back these Tes to their heap haven? Let us now review what can be done. From a grammar's rule's perspective, these symbols are always kept around for reuse and so will not produce a memory leak. But what about the Tes?

9.10.1 Recycling of Tes

Tes are created 2 ways: raw characters come from a fixed pool of memory. U do not need to attend to them. It is the meta Tes generated by your grammars that have to be tracked. Where are they stored? This depends on how u deal with them. From a parsing perspective they get created by some grammar and placed into its producer container to be digested by down stream grammars where this container is now a supplier. So the parser container used is one registry. U could also in parallel (nested called grammars) build your own tree of Tes and keep them registered outside the parse container by global variables. Remember there now are 2 references to them but only one to be used in recycling or double dipping will snap u. Having a global reference to the root node now becomes another registry. Depending on the activity, recycling of Tes could be placed in the "recycle container" for it to be walked and deleted later back to heap haven. As u can see there is no generic way of dealing with the recycling of Tes. It is left to u the language designer to deal with it. Being completely paranoid, u could keep your own T memory reference count registry. U are open to how u must maintain the memory integrity.

Now for the review on how Tes are deleted. I suggest u reread Chapter 3 on Tes to refresh your memory on the subtleties brought about by `destructor`, why C++'s *dtor* aka `~T name()` was not used, and the additional **R** variable definition, and possibly **ABORT_STATUS**/parse stack frame reference injected into the code body. Here is a T definition excerpt from an Xml language and a grammar rule deleting it. `T_mb_mess` terminal symbol contains references to its sub elements that also need deleting: date_, frm_, and query_.

```
"mb.mess"
/@
Message broker mess element containing its
subelements: Date, Frm, and Query.
@/
(sym-class T_mb_mess{
  user-declaration
    public:
    T_mb_mess(T_mb_mess_date* Date
              ,T_mb_mess_frm* Frm
              ,T_mb_mess_query* Query);
    T_mb_mess_date* date();
    T_mb_mess_frm* frm();
    T_mb_mess_query* query();
    AST* tree_node(){return tree_node_;};
    void tree_node(AST* Node){tree_node_ = Node;};
    AST* tree_node_;
```

[6] Expletive cleanings?

```
    private:
    T_mb_mess_date* date_;
    T_mb_mess_frm* frm_;
    T_mb_mess_query* query_;
   ***
  user-implementation
    T_mb_mess::
    T_mb_mess(T_mb_mess_date* Date,T_mb_mess_frm* Frm,T_mb_mess_query* Query)
    T_CTOR("mb.mess",T_Enum::T_T_mb_mess_\
          ,&T_mb_mess::dtor_T_mb_mess,false,false) // <- dtor ref: REQUIRED
      {date_ = Date;frm_ = Frm;query_ = Query;
       tree_node_ = new AST(*this);
       AST::crt_tree_of_3sons
                    (*tree_node_
                    ,*date_->tree_node()
                    ,*frm_->tree_node()
                    ,*query_->tree_node());
yacco2::lrclog << "newing T_mb_mess: " << this << std::endl;
    }
    T_mb_mess_date* T_mb_mess::date(){return date_;};
    T_mb_mess_frm* T_mb_mess::frm(){return frm_;};
    T_mb_mess_query* T_mb_mess::query(){return query_;};
  ***
  destructor
yacco2::lrclog << "deleting T_mb_mess: " << R << std::endl;
    DELETE_T_SYM(R->date_);// macro: delete T from inside a dtor
    DELETE_T_SYM(R->frm_);
    DELETE_T_SYM(R->query_);
    delete R->tree_node_;
  ***
  }
)

/*
* failed directive taken from the mess.lex grammar.
* Note how it explicitly tests and calls the DELETE_T_SYM macro
*  for each of its referenced T symbols.
* Their Tes's dtor functions are called and
* could call their own T dependencies.
*/
 failed
    CAbs_lr1_sym* sym = new ERR_improper_xml_tag_mess;
    sym->set_rc(*parser__->current_token__,__FILE__,__LINE__);
    if(date_ != 0){
      DELETE_T_SYM(date_);
      date_ = 0;
    }
    if(frm_ != 0){
      DELETE_T_SYM(frm_);
      frm_ = 0;
    }
    if(query_ != 0){
      DELETE_T_SYM(query_);
      query_ = 0;
    }
    RSVP_FSM(sym);// macro: rtn error to calling thread thru
                  // its accept queue from within fsm code
    return true;
 ***
```

T "mb.mess" is explicitly defined by the grammar writer with its implementation details. What is important is the use of the destructor directive. Here inside it is one of the delete lines:

```
DELETE_T_SYM(R->date_);
```

DELETE_T_SYM is a macro that recycles the T symbol **T_mb_mess_date** referenced by variable R->date_ back as a memory blob. I know u remembered that the R variable is automatically C++ generated by O_2 for each dtor procedure. Here is its C++ code:

```
@d DELETE_T_SYM(T)
  if(T != 0){
    if(T->enumerated_id__ > END_OF_RC_ENUMERATE){
      if(T->dtor__ != 0){
       (*T->dtor__)(T,0);// stack frame 0
      }
      delete T;
    }
  }
```

IMPORTANT note: u must include the global static dtor function reference within the CTOR macro of the T definition for it to activate: Warning u have been! Its C++ generated function name above is:

```
T_mb_mess::dtor_T_mb_mess
```

Please note its destructor's code also recycles the tree's AST node:

```
delete R->tree_node_;
```

Why is there 2 used contexts of DELETE_T_SYM macro in the above example? If the grammar failed, it washes out the partially created Tes by its failed directive. The 2nd context is the deleting action of T_mb_mess itself by its own destructor. This indicates that the T was successfully built and somewhere down the road there is a cleanup being done as the xml tree is being destroyed or euphemistically being recycled.

How is this example's T delete chain started?
This example had built the xml tree and globally deleted its nodes' contents by deleting their referenced objects within the parent node. These referenced objects in turn deleted their referenced objects. Anything that has been newed must be returned to it rightful owner. The new verb borrows from the memory heap. It is the delete verb that returns it back to the lender. The above example uses the failed directive in the "mess.lex" grammar to deal with any premature failings and partially created Tes. Other called threads within this example uses the same delete pattern on their own failings. U the grammar writer are open season to your own foibles on how u'd washed your dirty linen: walking the tree in postfix order is another way of removing those stains when the xml tree is fully built.

9.11 Summary

Be as bohemian as u like. I tried to give u as much license/inventiveness to your debug travels while Yac$_2o_2$ can leave tracks of its own findings. Hopefully there will not be too much frustration on your part regarding tantrum fits by O_2 or its companion the Yac$_2o_2$ library. Like golf where the score does not tell the obstacles overcome, your solution will have some interesting tales to tell. Tattlers are your logs. So slice and dice and flavour your source code with debug condiments.

Yes it is an annoyance when forced to debug but please mix fun into your investigations. "bonne chance".

Chapter 10

Tracings: not lr1 ;{

So far writing grammars has been a breeze for you until that blinking message "Not a lr(1) grammar!" arrived on your terminal. Well in chapter 6 on parsing it discussed the output to your "xxx_tracings.log" file where xxx is the prefix name of your grammr being compiled. When rewriting the lr1 state driver, i felt that the parsing contexts and their decisions should be logged. At least i could judge if the new implementation was correct. Upon a post evaluation of this logging, it seemed to me these decisions gave value and understanding to how the states were gened upto the "non lr(1)" condition. And so the traced decisions are part of O_2 instead of removing the temporary scaffolding. They complement the lr states being generated with their conflicting state(s). How to correct it is a different story! At least the gening lr states contexts are described and one should be able to figure out where and why the ambiguity occured within your grammar.

Now contexts are where it's at.[1] Have a read on O_2**.pdf** document for definitions like closure(d), transition, productions, subrules, state anatomy, follow set contribution, and for the curious on implementation details and my diatribes to getting it right.

10.1 A light overview of the Lr states generator

It is a top down algorithm generating states for each closured rule's productions symbols starting from the closured only state aka start state. The 1 exception is the state rule's productions are also considered implicitely closured as the start rule never appears in any production symbol strings. After all the closured state's closured rules productions have been state gened, the next closured part state within the growing lr state network becomes the next state to generate its closured rules's productions. This goes on until all the gened states have been evaluated for closured rules generation. The reducing set boundaries for a rule are the string of symbols to the right of a rule in the production yielding their terminals. If a closuring rule is the last symbol in the string, then it transitions to using the follow set of its parent rule. Epsilonal rules can make interior rules reducing set right bounded as the symbols string for its follow set could all be epsilonable. At state creation time, the closured rules's follow sets are calculated.

So a state containing a closured part gens its productions within a context: closure state / vector(s). Each different vector is a separate gening context of the productions within it. The reason for the context is each closured state / vector could gen some common states leading to conflict states out of the same closured part state produced by different vectors. Without the separated contexts, one might think that the common states could be merged but the follow sets from the merge now receive their content from the different contexts possibly making a legitimate lr(1) grammar non lr(1) [2].

[1] Son plastics is the future to invest in?

[2] Dave: 2 thumbs down on such a simplistic explanation without the specifics: right bounded rules, where/how follow sets get their booty. Please read O_2's documentation for a more elaborate explanation. Time will drop its comments on how well this document explains it. At least all the actors are there and discussed with their roles, and O_2's plot outlined.

10.2 Onto examples using this new tracings

The 3 grammars cementing this new approach are "lalr1_dp1.lex", "knu1_sick.lex", and "lr1_sp6.lex". Grammars 2 and 3 kept gening states until an "out of memory" condition: recursion overflow! Their fix created another problem: stopping the looping declared the 1st grammar as not lr(1) which is wrong. The rewrite now handles the situation properly[3]. Forgive me on the names as grammar "knu1_sick.lex" is a modified Knuth grammar taken from his paper [12] to test out state looping along with "lr1_sp6.lex" modified from a David Spector paper. These grammars and others are documented within the **../qa** folder (qa for quality assurance). There are other bash scripts to test out improper and kosher grammars[4] [5]. I'll discuss each grammar's tracings to help u in understanding a trace log and with the 2 grammars having issues.

The decisions identify by function name entered with a recursive count '.' indicator showing who is calling whom. Their names declare their behaviours which should orient u to their objectives. Each gening context list its state number and vector using its vocabulary enumeration. The gening context is compared against each potential subrule candidate's gen context to include or not in their continuing effort toward lr state generation.

10.2.1 "lalr1_dp1.lex" grammar traced log: lalr1_dp1_tracings.log

Here is its traced log output for a lr(1) accepted grammar.

```
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_includes.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_k_symbols.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_characters.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_terminals.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_err_symbols.T
Enumerate T alphabet
Total lrk symbols: 8
Start lrk symbol: 0 Stop lrk symbol: 7
Total rc symbols: 256
Start rc symbol: 8 Stop rc symbol: 263
Total T symbols: 114
Start T symbol: 264 Stop T symbol: 377
Total error symbols: 191
Start error symbol: 378 Stop error symbol: 568
Total symbols: 569
Enumerate Rule alphabet
Dump of P3 tokons
1:: #fsm file no: 2 line no: 21 pos: 1
2:: #T-enumeration file no: 4 line no: 36 pos: 1
3:: #lr1-constant-symbols file no: 5 line no: 54 pos: 1
4:: #raw-characters file no: 6 line no: 21 pos: 1
5:: #terminals file no: 7 line no: 20 pos: 1
6:: #error-symbols file no: 8 line no: 45 pos: 1
7:: #rules file no: 2 line no: 27 pos: 1
O2 version: .776 Distribution Date: Oct 13 2014

lr state driver considered state: 1 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 1
. for vector: 105
..gen_a_state for <1,105> requesting state: 1
..gen_a_state for <1,105> requesting state: 1 NEW STATE CREATED: 2
...gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 2
....gen_a_state for <1,105> requesting state: 2
....gen_a_state for <1,105> requesting state: 2 NEW STATE CREATED: 3
```

[3] Father time and his scythe could change this statement :{.

[4] **ls /usr/local/yacco2/qa/*sh** #to see their variants.

[5] **ls /usr/local/yacco2/qa/*pdf** #lists various grammar documents for your insatiable curiosity.

```
.....gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 3
......gen_a_state for <1,105> requesting state: 3
......gen_a_state for <1,105> requesting state: 3 NEW STATE CREATED: 4
.......gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 4
....gen_a_state for <1,105> requesting state: 2
....gen_a_state for <1,105> requesting state: 2 NEW STATE CREATED: 5
.....gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 5
......gen_a_state for <1,105> requesting state: 5
......gen_a_state for <1,105> requesting state: 5 NEW STATE CREATED: 6
.......gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 6
....gen_a_state for <1,105> requesting state: 2
....gen_a_state Bypass subrule as its gening <2,572> different then gening <1,105>
. for vector: 106
..gen_a_state for <1,106> requesting state: 1
..gen_a_state for <1,106> requesting state: 1 NEW STATE CREATED: 7
...gen_transitive_states_balance_for_closure_vector for < 1,106> goto state: 7
....gen_a_state for <1,106> requesting state: 7
....gen_a_state Bypass subrule as its gening <7,111> different then gening <1,106>
....gen_a_state for <1,106> requesting state: 7 NEW STATE CREATED: 8
.....gen_transitive_states_balance_for_closure_vector for < 1,106> goto state: 8
......gen_a_state for <1,106> requesting state: 8
......gen_a_state for <1,106> requesting state: 8 NEW STATE CREATED: 9
.......gen_transitive_states_balance_for_closure_vector for < 1,106> goto state: 9
....gen_a_state for <1,106> requesting state: 7
....gen_a_state for <1,106> requesting state: 7 NEW STATE CREATED: 10
.....gen_transitive_states_balance_for_closure_vector for < 1,106> goto state: 10
......gen_a_state for <1,106> requesting state: 10
......gen_a_state for <1,106> requesting state: 10 NEW STATE CREATED: 11
.......gen_transitive_states_balance_for_closure_vector for < 1,106> goto state: 11
....gen_a_state for <1,106> requesting state: 7
....gen_a_state Bypass subrule as its gening <7,572> different then gening <1,106>
. for vector: 570
..gen_a_state for <1,570> requesting state: 1
..gen_a_state for <1,570> requesting state: 1 NEW STATE CREATED: 12
...gen_transitive_states_balance_for_closure_vector for < 1,570> goto state: 12
....gen_a_state for <1,570> requesting state: 12
....gen_a_state for <1,570> requesting state: 12 NEW STATE CREATED: 13
.....gen_transitive_states_balance_for_closure_vector for < 1,570> goto state: 13
lr state driver considered state: 2 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 2
. for vector: 111
..gen_a_state for <2,111> requesting state: 2
..gen_a_state Bypass subrule as its gening <1,105> different then gening <2,111>
..gen_a_state subrule COMMON PREFIX state gened by a different gening context.  <2,111> goto state: 3
...gen_transitive_states_balance_for_closure_vector for < 2,111> goto state: 3
....gen_a_state for <2,111> requesting state: 3
....gen_a_state Bypass subrule as its gening <1,105> different then gening <2,111>
. for vector: 571
..gen_a_state for <2,571> requesting state: 2
..gen_a_state Bypass subrule as its gening <1,105> different then gening <2,571>
. for vector: 572
..gen_a_state for <2,572> requesting state: 2
..gen_a_state for <2,572> requesting state: 2 NEW STATE CREATED: 14
...gen_transitive_states_balance_for_closure_vector for < 2,572> goto state: 14
lr state driver considered state: 3 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 3
. for vector: 108
..gen_a_state for <3,108> requesting state: 3
..gen_a_state Bypass subrule as its gening <1,105> different then gening <3,108>
```

```
lr state driver considered state: 4 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 4
lr state driver considered state: 5 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 5
. for vector: 107
..gen_a_state for <5,107> requesting state: 5
..gen_a_state Bypass subrule as its gening <1,105> different then gening <5,107>
lr state driver considered state: 6 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 6
lr state driver considered state: 7 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 7
. for vector: 111
..gen_a_state for <7,111> requesting state: 7
..gen_a_state subrule COMMON PREFIX state gened by a different gening context.  <7,111> goto state: 8
...gen_transitive_states_balance_for_closure_vector for < 7,111> goto state: 8
....gen_a_state for <7,111> requesting state: 8
....gen_a_state Bypass subrule as its gening <1,106> different then gening <7,111>
. for vector: 571
..gen_a_state for <7,571> requesting state: 7
..gen_a_state Bypass subrule as its gening <1,106> different then gening <7,571>
. for vector: 572
..gen_a_state for <7,572> requesting state: 7
..gen_a_state for <7,572> requesting state: 7 NEW STATE CREATED: 15
...gen_transitive_states_balance_for_closure_vector for < 7,572> goto state: 15
lr state driver considered state: 8 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 8
. for vector: 107
..gen_a_state for <8,107> requesting state: 8
..gen_a_state Bypass subrule as its gening <1,106> different then gening <8,107>
lr state driver considered state: 9 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 9
lr state driver considered state: 10 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 10
. for vector: 108
..gen_a_state for <10,108> requesting state: 10
..gen_a_state Bypass subrule as its gening <1,106> different then gening <10,108>
lr state driver considered state: 11 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 11
lr state driver considered state: 12 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 12
. for vector: 1
..gen_a_state for <12,1> requesting state: 12
..gen_a_state Bypass subrule as its gening <1,570> different then gening <12,1>
lr state driver considered state: 13 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 13
lr state driver considered state: 14 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 14
lr state driver considered state: 15 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 15
Evaluate rules count
Output enumeration header file
Output grammar header file
Output grammar cpp file
Output grammar sym file
Output grammar tbl file
Exiting O2
```

The state generation trace starts with the statement: **lr state driver considered state:**. Each potential state being considered for generation has this statement. This is the algorithm doing its strut along the newly added states looking for "to be gened" closured contexts. Following the statement are the behaviors taken by the generator. The function names should clue u into their behaviors and the recursive levels showing the calling relationships within its gening context.

From the above log, "lr state driver considered state: 1 for vector: -1" starts the generation. Why the -1 vector? As the dynamicly generated states are walked for potential closured part generation, the state's vectors are not known until the its vectors are read: hence the -1 vector. Now onto the closured state assessment.

The function **gen_transitive_states_for_closure_context** considers the vectors within the state to gen their productions. **gen_transitive_states_balance_for_closure_vector** gens the states for these productions. Appropriate subrule productions already gened under a different closure state/vector context are bypassed. **gen_a_state** is the work horse to creating new states or merging the potentally created state into another closure state/vector state network.

Here are the generated lr states by their closured contexts: state 1 gens states 2 thru to 13, state 2 gens state 14, and state 7 gens state 15 which is the last closure state context completing the lr state network for the grammar. States like 9, are evaluated but no subrules found to gen or bypass. So what's up? This is a reducing state without any other shift items and so onto the next network's state for evaluation. "lr state driver considered state: 3" had closured shift items but were already gened so onto the next state to evaluate. Please read its generated documents: **/usr/local/yacco2/qa/lalr1_dp1.pdf** and **lalr1_dp1_idx.pdf** showing its generated states. The 2nd document gives the lookahead sets used in the reducing productions, merged into states graphs, and its reducing states list. So now dear reader, u are armed to deal with O_2 or your own grammar's foot stompings. The following 2 subsections deal with grammars not meeting the lr(1) constraint and some comments on how u can solve O_2's inconveniences.

10.2.2 knu1_sick.lex grammar's trace log: knu1_sick_tracings.log

Here is its dump stating that it is not lr(1).

```
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_includes.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_k_symbols.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_characters.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_terminals.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_err_symbols.T
Enumerate T alphabet
Total lrk symbols: 8
Start lrk symbol: 0 Stop lrk symbol: 7
Total rc symbols: 256
Start rc symbol: 8 Stop rc symbol: 263
Total T symbols: 114
Start T symbol: 264 Stop T symbol: 377
Total error symbols: 191
Start error symbol: 378 Stop error symbol: 568
Total symbols: 569
Enumerate Rule alphabet
Dump of P3 tokons
1:: #fsm file no: 2 line no: 29 pos: 1
2:: #T-enumeration file no: 4 line no: 36 pos: 1
3:: #lr1-constant-symbols file no: 5 line no: 54 pos: 1
4:: #raw-characters file no: 6 line no: 21 pos: 1
5:: #terminals file no: 7 line no: 20 pos: 1
6:: #error-symbols file no: 8 line no: 45 pos: 1
7:: #rules file no: 2 line no: 36 pos: 1
O2 version: .776 Distribution Date: Oct 13 2014
```

```
lr state driver considered state: 1 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 1
. for vector: 105
..gen_a_state for <1,105> requesting state: 1
..gen_a_state for <1,105> requesting state: 1 NEW STATE CREATED: 2
...gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 2
....gen_a_state for <1,105> requesting state: 2
....gen_a_state Bypass subrule as its gening <2,105> different then gening <1,105>
....gen_a_state for <1,105> requesting state: 2
....gen_a_state for <1,105> requesting state: 2 NEW STATE CREATED: 3
.....gen_transitive_states_balance_for_closure_vector for <1,105> goto state: 3
......gen_a_state for <1,105> requesting state: 3
......gen_a_state for <1,105> requesting state: 3 NEW STATE CREATED: 4
.......gen_transitive_states_balance_for_closure_vector for < 1,105> goto state: 4
........gen_a_state for <1,105> requesting state: 4
........gen_a_state for <1,105> requesting state: 4
........gen_a_state for <1,105> requesting state: 4 NEW STATE CREATED: 5
===>Please check Grammar dump file: /usr/local/yacco2/qa/knu1_sick_tracings.log for Not LR1 details
Not LR1 --- check state conflict list of state: 1 for details

State dump

->State: 1 Entry: -1 Symbol:  No symbol
   Follow set:
    Rule no: 569 Name: Rstart
    follow set:
      eolr
    Rule no: 570 Name: RS
    follow set:
      eog     merges
      S4
   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: Empty see following transition
      RxSRxPos: 2.1.1 Eos of RS Closured S1 CS-gening-it S1 reduced S1
    Symbol no: 105 Symbol: a
      RxSRxPos: 2.2.1 a Closured S1 CS-gening-it S1 goto S2 reduced S????
    Symbol no: 106 Symbol: b
      RxSRxPos: 2.3.1 b Closured S1 CS-gening-it S1 reduced S????
    Symbol no: 570 Symbol: RS
      RxSRxPos: 1.1.1 RS Closured S1 CS-gening-it S1 reduced S????
  ===>Conflict states:
      State no: 2
      State no: 4
      State no: 5
End of state

->State: 2 Entry: 105 Symbol: a Birthing closure state: 1
   Follow set:
    Rule no: 571 Name: RA
    follow set:
      b
   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: -1
        b
      RxSRxPos: 3.1.1 Eos of RA Closured S2 CS-gening-it S2 reduced S2
    Symbol no: 105 Symbol: a
```

```
      RxSRxPos: 3.2.1 a Closured S2 CS-gening-it S2 reduced S????
    Symbol no: 571 Symbol: RA
      RxSRxPos: 2.2.2 RA Closured S1 CS-gening-it S1 goto S3 reduced S????
End of state

->State: 3 Entry: 571 Symbol: RA Birthing closure state: 1
   Follow set:
   Vectors:
    Symbol no: 106 Symbol: b
      RxSRxPos: 2.2.3 b Closured S1 CS-gening-it S1 goto S4 reduced S????
End of state

->State: 4 Entry: 106 Symbol: b Birthing closure state: 1
   Follow set:
    Rule no: 570 Name: RS
    transitions
      S1xRS

   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: -1
        eog
      RxSRxPos: 2.1.1 Eos of RS Closured S4 CS-gening-it S4 reduced S4
    Symbol no: 105 Symbol: a
      RxSRxPos: 2.2.1 a Closured S4 CS-gening-it S4 goto S2 reduced S????
    Symbol no: 106 Symbol: b
      RxSRxPos: 2.3.1 b Closured S4 CS-gening-it S4 goto S5 reduced S????
    Symbol no: 570 Symbol: RS
      RxSRxPos: 2.2.4 RS Closured S1 CS-gening-it S1 reduced S????
End of state

->State: 5 Entry: 106 Symbol: b Birthing closure state: 1
   Follow set:
    Rule no: 571 Name: RA
    follow set:
      a
   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: -1
        a
      RxSRxPos: 3.1.1 Eos of RA Closured S5 CS-gening-it S5 reduced S5
    Symbol no: 105 Symbol: a
      RxSRxPos: 3.2.1 a Closured S5 CS-gening-it S5 reduced S????
    Symbol no: 571 Symbol: RA
      RxSRxPos: 2.3.2 RA Closured S4 CS-gening-it S1 due to rt bnd  reduced S????
End of state
Common States dump
1::Common State: -1 Entry Symbol: No symbol
state no: 1

2::Common State: 105 Entry Symbol: a
state no: 2

3::Common State: 106 Entry Symbol: b
state no: 4
state no: 5

4::Common State: 571 Entry Symbol: RA
state no: 3
```

From the above trace, state 5 is being gened under <1,105> context and is the conflicting non lr(1) state. With the recursion level expanding, state 1 gens 2 gens 3 gens 4 then 5. The conflict is between its shift "a" and reducing production follow set which also has an "a". Its dumped anatomy details this. But why is it causing a problem? It is the epsilon production of the **Ra** rule that is reducing. So where/who are the contributors to its follow set? Again looking at state 5, you'll see Ra's follow set is from 2.3.2 — rule 2 subrule 3 position 2. Have a look at the grammar lex file. There u'll see "a" is in 2.3.3 position. There is a comment left in the grammar as to why the conflict.

Hey I've got a mental block and headache Dave from your explanation! Rule 2.2.1 is being closured gened but why is Rule 2.3.1 also being gened within this context giving the conflict? It is due to a right bounded rule **Rs** from Rule 2.2.3. in state 4. And demands gening all its subrules as part of the gening context.

Well reason why the conflict found but how to correct it! My comments in this grammar explans why but does this give u any leads to your own inconsistencies? I guess not but my mumblings might give some insight to play within your own mumbling grammars. Possibly using threaded grammars with arbitration might help u. This requires splitting the inconsistent grammar into threaded pieces but allows u to then to view the generated grammars's documents to refine your grammar thoughts to re-consolidate the threaded pieces back into 1 grammar. As threading isn't too expensive, why not leave the threaded pieces in place?

10.2.3 lr1_sp6.lex grammar's trace log: lr1_sp6_tracings.log

Here is its dump stating that it is not lr(1).

```
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_includes.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_k_symbols.T
Pre-process file: /usr/local/yacco2/library/grammars/yacco2_characters.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_terminals.T
Pre-process file: /usr/local/yacco2/compiler/grammars/yacco2_err_symbols.T
Enumerate T alphabet
Total lrk symbols: 8
Start lrk symbol: 0 Stop lrk symbol: 7
Total rc symbols: 256
Start rc symbol: 8 Stop rc symbol: 263
Total T symbols: 114
Start T symbol: 264 Stop T symbol: 377
Total error symbols: 191
Start error symbol: 378 Stop error symbol: 568
Total symbols: 569
Enumerate Rule alphabet
Dump of P3 tokons
1:: #fsm file no: 2 line no: 18 pos: 1
2:: #T-enumeration file no: 4 line no: 36 pos: 1
3:: #lr1-constant-symbols file no: 5 line no: 54 pos: 1
4:: #raw-characters file no: 6 line no: 21 pos: 1
5:: #terminals file no: 7 line no: 20 pos: 1
6:: #error-symbols file no: 8 line no: 45 pos: 1
7:: #rules file no: 2 line no: 23 pos: 1
O2 version: .776 Distribution Date: Oct 13 2014

lr state driver considered state: 1 for vector: -1
.gen_transitive_states_for_closure_context for closure state: 1
. for vector: 125
..gen_a_state for <1,125> requesting state: 1
..gen_a_state for <1,125> requesting state: 1 NEW STATE CREATED: 2
...gen_transitive_states_balance_for_closure_vector for < 1,125> goto state: 2
....gen_a_state for <1,125> requesting state: 2
....gen_a_state for <1,125> requesting state: 2 NEW STATE CREATED: 3
```

```
.....gen_transitive_states_balance_for_closure_vector for < 1,125> goto state: 3
......gen_a_state for <1,125> requesting state: 3
......gen_a_state for <1,125> requesting state: 3 NEW STATE CREATED: 4
.......gen_transitive_states_balance_for_closure_vector for < 1,125> goto state: 4
........gen_a_state for <1,125> requesting state: 4
........gen_a_state for <1,125> requesting state: 4 NEW STATE CREATED: 5
.........gen_transitive_states_balance_for_closure_vector for < 1,125> goto state: 5
........gen_a_state for <1,125> requesting state: 4
........gen_a_state for <1,125> requesting state: 4 NEW STATE CREATED: 6
===>Please check Grammar dump file: /usr/local/yacco2/qa/lr1_sp6_tracings.log for Not LR1 details
Not LR1 --- check state conflict list of state: 1 for details

State dump

->State: 1 Entry: -1 Symbol:  No symbol
   Follow set:
    Rule no: 569 Name: Repi_test
    follow set:
      eolr
   Vectors:
    Symbol no: 125 Symbol: u
      RxSRxPos: 1.1.1 u Closured S1 CS-gening-it S1 goto S2 reduced S????
  ===>Conflict states:
      State no: 5
      State no: 6
End of state

->State: 2 Entry: 125 Symbol: u Birthing closure state: 1
   Follow set:
    Rule no: 570 Name: Repi_rt_bnd_tst
    follow set:
      eog     transitions
      S1xRepi_test

    Rule no: 573 Name: Rlr1_sp5
    follow set:
      m     transitions
      S2xRepi_rt_bnd_tst

    Rule no: 574 Name: RS
    transitions
      S2xRlr1_sp5

    Rule no: 575 Name: RA
    follow set:
      z
    Rule no: 576 Name: RB
    follow set:
      x
   Vectors:
    Symbol no: 105 Symbol: a
      RxSRxPos: 6.1.1 a Closured S2 CS-gening-it S2 goto S3 reduced S????
      RxSRxPos: 6.3.1 a Closured S2 CS-gening-it S2 goto S3 reduced S????
      RxSRxPos: 6.2.1 a Closured S2 CS-gening-it S2 goto S3 reduced S????
    Symbol no: 108 Symbol: d
      RxSRxPos: 7.1.1 d Closured S2 CS-gening-it S2 reduced S????
      RxSRxPos: 8.1.1 d Closured S2 CS-gening-it S2 reduced S????
    Symbol no: 570 Symbol: Repi_rt_bnd_tst
      RxSRxPos: 1.1.2 Repi_rt_bnd_tst Closured S1 CS-gening-it S1 reduced S????
```

```
    Symbol no: 573 Symbol: Rlr1_sp5
      RxSRxPos: 2.1.1 Rlr1_sp5 Closured S2 CS-gening-it S2 reduced S????
    Symbol no: 574 Symbol: RS
      RxSRxPos: 5.1.1 RS Closured S2 CS-gening-it S2 reduced S????
    Symbol no: 575 Symbol: RA
      RxSRxPos: 6.4.1 RA Closured S2 CS-gening-it S2 reduced S????
    Symbol no: 576 Symbol: RB
      RxSRxPos: 6.5.1 RB Closured S2 CS-gening-it S2 reduced S????
End of state

->State: 3 Entry: 105 Symbol: a Birthing closure state: 1
   Follow set:
    Rule no: 577 Name: RD
    transitions
      S2xRS

   Vectors:
    Symbol no: 106 Symbol: b
      RxSRxPos: 6.1.2 b Closured S2 CS-gening-it S1 due to rt bnd  goto S4 reduced S????
      RxSRxPos: 6.2.2 b Closured S2 CS-gening-it S1 due to rt bnd  goto S4 reduced S????
    Symbol no: 110 Symbol: f
      RxSRxPos: 9.1.1 f Closured S3 CS-gening-it S3 reduced S????
      RxSRxPos: 9.2.1 f Closured S3 CS-gening-it S3 reduced S????
      RxSRxPos: 9.3.1 f Closured S3 CS-gening-it S3 reduced S????
    Symbol no: 577 Symbol: RD
      RxSRxPos: 6.3.2 RD Closured S2 CS-gening-it S2 reduced S????
End of state

->State: 4 Entry: 106 Symbol: b Birthing closure state: 1
   Follow set:
    Rule no: 575 Name: RA
    follow set:
      z     transitions
      S2xRS

    Rule no: 576 Name: RB
    follow set:
      x
   Vectors:
    Symbol no: 108 Symbol: d
      RxSRxPos: 7.1.1 d Closured S4 CS-gening-it S4 goto S5 reduced S5
      RxSRxPos: 8.1.1 d Closured S4 CS-gening-it S4 goto S5 reduced S5
    Symbol no: 575 Symbol: RA
      RxSRxPos: 6.1.3 RA Closured S2 CS-gening-it S1 due to rt bnd  goto S6 reduced S????
    Symbol no: 576 Symbol: RB
      RxSRxPos: 6.2.3 RB Closured S2 CS-gening-it S1 due to rt bnd  reduced S????
End of state

->State: 5 Entry: 108 Symbol: d Birthing closure state: 1
   Follow set:
   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: Empty see following transition
      RxSRxPos: 7.1.2 Eos of RA Closured S4 CS-gening-it S1 due to rt bnd  reduced S5
      reduce set #: Empty see following transition
      RxSRxPos: 8.1.2 Eos of RB Closured S4 CS-gening-it S1 due to rt bnd  reduced S5
End of state

->State: 6 Entry: 575 Symbol: RA Birthing closure state: 1
```

```
   Follow set:
    Rule no: 578 Name: RE
    transitions
      S2xRS

   Vectors:
    Symbol no: -369 Symbol: eosubrule
      reduce set #: -1
        eog eolr m
      RxSRxPos: 10.1.1 Eos of RE Closured S6 CS-gening-it S6 reduced S6
    Symbol no: 130 Symbol: z
      RxSRxPos: 10.2.1 z Closured S6 CS-gening-it S6 reduced S????
    Symbol no: 578 Symbol: RE
      RxSRxPos: 6.1.4 RE Closured S2 CS-gening-it S1 due to rt bnd  reduced S????
End of state
Common States dump
1::Common State: -1 Entry Symbol: No symbol
state no: 1

2::Common State: 105 Entry Symbol: a
state no: 3

3::Common State: 106 Entry Symbol: b
state no: 4

4::Common State: 108 Entry Symbol: d
state no: 5

5::Common State: 125 Entry Symbol: u
state no: 2

6::Common State: 575 Entry Symbol: RA
state no: 6
```

The previous grammar log discussed "how to" read the conflicting state. Looking at state's 6 dump, the state is a reduce/shift type.

The shift part is **z** and the reducing set doesn't have a **z** entry but terminals: eog eolr m.

So what gives?
As "eolr" is part of the set, this represents the "all terminals" in the T vocabulary including self. So conflict found: shift **z** and reduce **eolr** which includes **z**.

Now where did **eolr** come from?
Go look at **/usr/local/yacco2/qa/lr1_sp6.lex** grammar. The rule **Repi_eog** is right bounded and epsilonable. So what is the end of a grammar. Well O_2 represents the out-of-bounds by "eolr". Now the solution could be to remove the epsilon production from rule **Repi_eog**.

10.3 Summary.

Well u could call this section a bit thin in content and substance. For the moment this is O_2 offerings. As time clicks away, I'm sure this will be better constructed and thought out. Users like yourself will have suggestion of many flavours to improve this. So onto the Jewels of O_2[6].

[6]What are those cracking noises? Just peanuts being shelled ... from what gallery?

Chapter 11

The social network of grammars: O_2linker

"mirror mirror on the wall" or "facebook" ain't got nothin on dis. It's the crawler of potential threads extracting their callings and brokering it into calling relationships. Okay Dave your 2 micro seconds of staging is up so on with the show. Please have a re-look at Figure 1.3 overviewing O_2linker's runtime.

Basically the problem at hand is when should a thread get called? This is easy when it is present within the grammar's parse state. Well this is generally right but thread thumping can be heavy — please turn down the volume. So as an optimization, the thread's "first set" is calculated which is the starting Tes to its potential token stream segment. Along with this is an efficient way to declare the "first set" per thread. This is done globally whereby each T has a bit pattern of threads where this T is in their "first set". Using the current token, the parser can quickly determine whether its own parse state's threads list has the potential to run. The current potential total number of threads supported by O_2linker is 640: 20 integers * 32 bits. The table structures used by O_2 are open-ended if this limit is extended and space optimized according to the number of defined threads against the number of 32 bit words needed. In the example below, there are approximately 50 defined threads whose thread bit patterns take up two 32 bit words. [1]

Another problem is how should a thread be addressed within the lr(1) tables? Just call me Sir. Well this requires an ordering against the threads names: call this the thread's call-id. Now how does Yac$_2o_2$'s library know whom these threads are? Yac$_2o_2$'s library is open-ended to the future having a big question mark against this future. To get to the future threads, Yac$_2o_2$ references global thread-calling variables manufactured by O_2linker. This output module gets compiled and the native static linker resolves these references for your own translator. So there u have it. Not quite. The threads themselves have to be memory addressed, and these addresses kept in a runtime table for dispatching and reuse. Here is an excerpt of O_2's thread module from O_2linker. The capitalized variable names are global symbols used by Yac$_2o_2$ library.

```
// BIT MAPS
#define TOTAL_NO_BIT_WORDS 2*1024*50
int                   yacco2::TOTAL_NO_BIT_WORDS__(TOTAL_NO_BIT_WORDS);
yacco2::ULINT         bit_maps[TOTAL_NO_BIT_WORDS];
void*                 yacco2::BIT_MAPS_FOR_SALE__ = (void*)&bit_maps;
int                   yacco2::BIT_MAP_IDX__(0);
void*                 yacco2::T_ARRAY_HAVING_THD_IDS__ = (void*)&t_array;

// THREAD STABLE of O2.
yacco2::Thread_entry ITH_angled_string =
     {"TH_angled_string"                 // thread name for tracing
     ,NS_angled_string::TH_angled_string // thread reference
     ,0                                  // thread call-id
```

[1] A watch out: one local `pthread` library implementation demanded linker/thread stack allocation experimentation to get O_2 to run properly. The problem was in balancing what the process allocated stack space needed to what a thread needed in stack allocation due to the `pthread` implementation.

```
     ,NS_angled_string::PROC_TH_angled_string // equivalent function ref
     };
yacco2::Thread_entry ITH_bad_char_set =
     {"TH_bad_char_set"
     ,NS_bad_char_set::TH_bad_char_set
     ,1
     ,NS_bad_char_set::PROC_TH_bad_char_set};
yacco2::Thread_entry ITH_cweb_or_c_k =
     {"TH_cweb_or_c_k",NS_cweb_or_c_k::TH_cweb_or_c_k,2
     ,NS_cweb_or_c_k::PROC_TH_cweb_or_c_k};
yacco2::Thread_entry ITH_c_comments =
     {"TH_c_comments",NS_c_comments::TH_c_comments,3
     ,NS_c_comments::PROC_TH_c_comments};
              ... elided code
yacco2::Thread_entry ITH_ws =
     {"TH_ws",NS_ws::TH_ws,43
     ,NS_ws::PROC_TH_ws};
yacco2::Thread_entry ITH_xc_str =
     {"TH_xc_str",NS_xc_str::TH_xc_str,44
     ,NS_xc_str::PROC_TH_xc_str};

/**
* Internal table of thread structures
**/
struct thd_array_type {
  yacco2::USINT         no_entries__;
  yacco2::Thread_entry* first_entry__[45];};
thd_array_type thd_array = {
 45
 ,
  {
    &ITH_angled_string // address to each above thread's record
   ,&ITH_bad_char_set
   ,&ITH_cweb_or_c_k
   ,&ITH_c_comments
              ... elided code
   ,&ITH_ws
   ,&ITH_xc_str
  }
};

void* yacco2::THDS_STABLE__ // global reference to thread table
      = (void*)&thd_array;

/**
*  Terminal thread sets
**/
struct T_0_type{
 yacco2::ULINT first_entry__[2];
};

T_0_type T_0 = {// for T: LR1_questionable_shift_operator
//8: TH_err_symbols_ph_th
//10: TH_fsm_class_phrase_th
//11: TH_fsm_phrase_th
//14: TH_la_expr_src
//17: TH_linker_preamble_code
//19: TH_lr1_k_phrase_th
//22: TH_o2_sdc
```

```
//25: TH_parallel_parser_ph_th
//27: TH_rc_phrase_th
//29: TH_rhs_component
//30: TH_rtn_component
//31: TH_rules_phrase_th
//32: TH_rule_def_phrase
//34: TH_subrules_phrase
//37: TH_terminals_phrase_th
//41: TH_T_enum_phrase_th
//42: TH_unq_str
 {3930737920 // thread call-id bit pattern
  ,1573      // across two 32 bits integers
             // space calculated against number of defined threads
 }
};
struct T_1_type{
 yacco2::ULINT first_entry__[2];
};

T_1_type T_1 = {// for T: LR1_eog
//14: TH_la_expr_src
//17: TH_linker_preamble_code
//22: TH_o2_sdc
//28: TH_rhs_bnd
//29: TH_rhs_component
//30: TH_rtn_component
//42: TH_unq_str
 {1883389952
  ,1024
 }
};
struct T_6_type{
 yacco2::ULINT first_entry__[2];
};

T_6_type T_6 = {// for T: LR1_all_shift_operator
//14: TH_la_expr_src
//17: TH_linker_preamble_code
//22: TH_o2_sdc
//29: TH_rhs_component
//30: TH_rtn_component
//42: TH_unq_str
 {1614954496
  ,1024
 }
};
    ... elided code
struct T_568_type{
 yacco2::ULINT first_entry__[2];
};

T_568_type T_568 = {// for T: Err_not_lhs_pcnrl_mntr
//14: TH_la_expr_src
//17: TH_linker_preamble_code
//22: TH_o2_sdc
//29: TH_rhs_component
//30: TH_rtn_component
//42: TH_unq_str
 {1614954496
```

```
  ,1024
 }
};
struct t_array_type {
  yacco2::USINT no_entries__;
  yacco2::thd_ids_having_T* first_entry__[569];
};
t_array_type t_array = {
  569
  ,{
(yacco2::thd_ids_having_T*)&T_0 // LR1_questionable_shift_operator
    ,(yacco2::thd_ids_having_T*)&T_1 // LR1_eog
    ,0// LR1_eolr
    ,0// LR1_parallel_operator
    ,0// LR1_reduce_operator
    ,0// LR1_invisible_shift_operator
    ,(yacco2::thd_ids_having_T*)&T_6 // LR1_all_shift_operator
    ,(yacco2::thd_ids_having_T*)&T_7 // LR1_fset_transience_operator
    ,(yacco2::thd_ids_having_T*)&T_8 // raw_nul
    ,(yacco2::thd_ids_having_T*)&T_9 // raw_soh
    ,(yacco2::thd_ids_having_T*)&T_10 // raw_stx
    ,(yacco2::thd_ids_having_T*)&T_11 // raw_etx
                ... elided code
    ,(yacco2::thd_ids_having_T*)&T_563 // Err_not_kw_defining_grammar_construct
    ,(yacco2::thd_ids_having_T*)&T_564 // Err_use_of_N_outside_Rules_construct
    ,(yacco2::thd_ids_having_T*)&T_565 // Err_misplaced_or_misspelt_Rule_or_T
    ,(yacco2::thd_ids_having_T*)&T_566 // Err_not_a_Rule
    ,(yacco2::thd_ids_having_T*)&T_567 // Err_empty_file
    ,(yacco2::thd_ids_having_T*)&T_568 // Err_not_lhs_pcnrl_mntr
   }
};
void* yacco2::T_ARRAY_HAVING_THD_IDS__ = (void*)&t_array;
```

11.1 A grammar's declaration to O_2linker

As part of emitting the lr(1) tables for a grammar, O_2 emits its grammar inter-relationships for O_2linker. This is another language that also gets parsed by O_2linker assessment. Here is grammar "pass3.lex" output: "pass3.fsc" file for O_2linker consumption:

```
/*
 File: pass3.fsc
 Date and Time: Wed Mar 24 11:29:12 2010
*/
transitive   y
grammar-name "pass3"
name-space   "NS_pass3"
thread-name  "Cpass3"
monolithic   y
file-name    "pass3.fsc"
no-of-T      569
list-of-native-first-set-terminals 3
   LR1_questionable_shift_operator
   LR1_eog
   raw_at_sign
end-list-of-native-first-set-terminals
list-of-transitive-threads 5
  NS_ws::TH_ws
```

```
  NS_identifier::TH_identifier
  NS_cweb_or_c_k::TH_cweb_or_c_k
  NS_eol::TH_eol
  NS_bad_char_set::TH_bad_char_set
end-list-of-transitive-threads
list-of-used-threads 5
  NS_bad_char_set::TH_bad_char_set
  NS_cweb_or_c_k::TH_cweb_or_c_k
  NS_eol::TH_eol
  NS_identifier::TH_identifier
  NS_ws::TH_ws
end-list-of-used-threads
fsm-comments
"\\O2's lexer constructing tokens for syntax parser stage."
```

The outputted file's extension is ".fsc" acronym for "first set control'. I wasn't too creative as u can see. There are 5 parts to this language making up the file:

1. The grammar generating it and its type.

2. The native "first set" of Tes for the grammar.
   Native means these are direct Tes beginning the "Start rule".

3. List of called threads from its "Start rule".
   This is the reason for it being called "list-of-transitive-threads". When a grammar calls a grammar that calls a grammar each call can expose its "first set" to the original called thread. This is a recursive statement that O_2linker handles by transitive closure. It calculates the grammar's "first set" exposures across thread calls starting from its "Start rule" which can have rules where their subrules call threads whose "first sets" also have thread calls. Part of the problem are epsilonable rules partaking in the grammar's "Start rule".

4. List of used threads within the grammar.
   This provides a call index for O_2linker's generated document. In the example they are the same as the transitive thread call list. But this is not always the case as there can be buried thread calls within the grammar that are not reachable by the "Start rule" evaluation: no epsilonable rules or has epsilonal rules but whose "follow set" do not contain a called thread expression.

5. `fsm-comments` is the grammar's `fsm`'s parameter.
   Its commentary shows up in its table of contents and section describing the grammar thread. It is the table of contents that i find useful particularly when you received a error message identifying the thread called. Depending on u, the thread name might be acronymed and with time u've forgotten what it means. Dada the table of content is your searching source.

11.2 Input file to O_2linker

Now there are many grammars, how would u declare them for O_2linker consumption? For the moment, u have to hand-code the file giving all the grammars to be linked. Ugh but this is the current state of affairs. Here is O_2's own file for O_2linker:

```
file-of-T-alphabet   "/usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.fsc"
emitfile             "/usr/local/yacco2/compiler/grammars/yacco2_fsc.cpp"

preamble
#include "yacco2.h"
#include "yacco2_T_enumeration.h"
#include "yacco2_k_symbols.h"
#include "yacco2_terminals.h"
#include "yacco2_characters.h"
```

```
using namespace NS_yacco2_T_enum;
using namespace NS_yacco2_err_symbols;
using namespace NS_yacco2_k_symbols;
using namespace NS_yacco2_terminals;
using namespace NS_yacco2_characters;
end-preamble

/**
* List of each grammar's emitted fsc file
* Their order is not important. Just their presence.
**/
"/usr/local/yacco2/compiler/grammars/rules_use_cnt.fsc"
"/usr/local/yacco2/compiler/grammars/enumerate_grammar.fsc"
"/usr/local/yacco2/compiler/grammars/enumerate_T_alphabet.fsc"
"/usr/local/yacco2/compiler/grammars/angled_string.fsc"
"/yacco2/compiler/grammars/bad_char_set.fsc"
"/usr/local/yacco2/compiler/grammars/c_comments.fsc"
"/usr/local/yacco2/compiler/grammars/c_literal.fsc"
"/usr/local/yacco2/compiler/grammars/c_string.fsc"
      ... elided code
"/usr/local/yacco2/compiler/grammars/o2_lcl_opt.fsc"
"/usr/local/yacco2/compiler/grammars/o2_lcl_opts.fsc"
"/usr/local/yacco2/compiler/grammars/o2_linker_opts.fsc"
```

11.2.1 O_2linker's input "fsc" file composition

There are 3 parts:

1. Preliminary info:
   `file-of-T-alphabet` is a keyword that introduces the T alphabet enumeration file name. Its "fsc" file is generated when u use the `-t` option to O_2. Have a reread on a grammar's `T-enumeration` construct. One of its parameters is the `file-name` for the emitted file. Its contents match the members of the T vocabulary. One of the cross-checks is the number of items in this file must match up with the number of Tes expressed in each grammar's emitted "fsc" file. A mismatch occurs when a grammar has been changed due to creating or removing a T member from the T alphabet in either the error or regular T class. This operation might change the T ordering and definitely the number of Tes. Without re-gening all the grammars, their tables will be compromised. I know as i've been bitten by irregular parsing behaviours by not re-gening either the thread tables by O_2linker nor re-gening all the grammars by O_2. After this u must i said must recompile their C++ code modules. So warned dear reader as i know the frustrations to some of my own memory lapses (an euphemism for stupidities).

   Here's `T-enumeration` "fsc" file sample for O_2's grammars:

   ```
   /*
    File: yacco2_T_enumeration.fsc
    Date and Time: Mon Feb 22 11:30:31 2010
   */
   T-alphabet
   LR1_questionable_shift_operator // c++ class name
   LR1_eog
   LR1_eolr
   LR1_parallel_operator
   LR1_reduce_operator
   LR1_invisible_shift_operator
   LR1_all_shift_operator
   LR1_fset_transience_operator
   raw_nul
       ... elided code
   Err_not_a_Rule                 // c++ class name
   ```

```
Err_empty_file
Err_not_lhs_pcnrl_mntr
end-T-alphabet
```

This list is used to recalculate the T's enumeration scheme by order of appearance as what O_2 did for its own tables. There is no significance attached to the use of the T's class name instead of its literal value. I took the easy way out in not having to deal with quoting properly the literal.

Each grammar's "fsc" file `list-of-native-first-set-terminals` construct supplies its direct Tes list. Indirect T members are calculated for the "Start rule" of the thread having epsilonal rules and thread calls. That is the thread called must also have its "first set" calculated. `emitfile` keyword identifies the file to be outputted.

2. `preamble/end-preamble` keywords.
   They bracket typically "include file' directives required to be added at the beginning to the emitted output file. In the example above, it contains Yac$_2o_2$'s definition, the related T vocabulary files, and their namespace definitions. Without the "preamble" items the emited file would have compile errors.

3. The list of each grammar's emitted "fsc" files.
   They are the fodder to O_2's linker. There is no significance placed on their inputted order: random as can be. Just make sure all your grammars are entered in its list.

11.3 O_2linker's outputted document: the soothsayer of threads

I'm not ashamed to say it: O_2linker emits *Cweb* code so that the *cweave* program generates the TEX file for its typesetting. "Literate Programming" is a treat to use. In my case why re-invent the wheel when time/expertise was not on my side and this excellent family of tools was open for use. *cweave* became the template model for document generation having its own set of typesetting directives. It is a form of script piping where the output of 1 tool becomes the input for the 2nd and so on exploiting each tool's manufacturing process. Here's my take on mapping "language content" into document form. It is a draft on O_2's own "literate programming" from a pragmatist perspective. Your own expressive thoughts should renaissance your document creations. [2]

o2linker_doc.pdf is the outputted document that is composed of 5 parts:

1. The table of content.
   A great summary of all the grammars written for monolithic and threaded grammars. The entries are entered in lexigraphical order first for threads followed by monolithic grammars. Each grammar's commentary comes from the grammar's `fsm-comments` parameter.

2. Introduction about the document.

3. The grammars.
   Each grammar entry has:

   - A brief description.
     U can bury TEX macros inside it for special effects.
   - Hierarchical list of called threads making up its first set.
     A grammar's "Start rule" could call a grammar whose "Start rule" calls another grammar and so on. This is equivalent to a rule's "first set" calculation with thread calls thrown in. A question for u: what is the "first set" if its "Start rule" reaches its lookahead expression and possibly the called grammars themselves reached their "lookahead" expression? Answer: The lookahead expression should be expressed in the grammar's "fsc" file. As `LR1_eolr` could be the only one in this expression and represents "all symbols including self" in the "first set", should `LR1_eolr` only be represented? But what does this represent to a parser's current token and the optimization

[2] What webbing would u call this? kweb? an lr(k) algorithm, typesetting tools ...

previously discussed? Nada. Because of this, the "first set" explodes to all the real Tes including error Tes as u could be grammaring on these as in post-error processing.

- List of calculated "first set" members.
- List of all called threads within the grammar.

4. Input "fsc" file contents given to O_2linker.

5. Index.
   An excellent cross reference of all the symbols used by all the grammars: T vocabulary, thread's name, C++ class name, and grammar calls who-called-whom.

11.3.1 Extracts of the document

Here is a snap shot of a O_2linker document taken from O_2. As it is quite large (27 pages), i sampled the various sections described.

1. O2linker Index of Grammars.

The grammars are sorted lexicographically into 2 parts: threads followed by the stand alone grammars. Each grammar's called threads graph is determined from their "list-of-transitive-threads" derived from this construct.

2. TH_angled_string — Angled string lexer: < . .. > with c ....

Angled string lexer: < ... > with c type escape sequences.

First set of called threads list.

TH_angled_string

First set: raw_less_than
Used threads: none

3. TH_bad_char_set — Bad source character set recognizer.

Bad source character set recognizer.

First set of called threads list.

TH_bad_char_set

First set: raw_nul raw_soh raw_stx raw_etx raw_eot raw_enq raw_ack raw_bel raw_bs raw_so raw_si raw_dle raw_dc1 raw_dc2 raw_dc3 raw_dc4 raw_nak raw_syn raw_etb raw_can raw_em raw_sub raw_esc raw_fs raw_gs raw_rs raw_us raw_del raw_x80 raw_x81 raw_x82 raw_x83 raw_x84 raw_x85 raw_x86 raw_x87 raw_x88 raw_x89 raw_x8a raw_x8b raw_x8c raw_x8d raw_x8e raw_x8f raw_x90 raw_x91 raw_x92 raw_x93 raw_x94 raw_x95 raw_x96 raw_x97 raw_x98 raw_x99 raw_x9a raw_x9b raw_x9c raw_x9d raw_x9e raw_x9f raw_xa0 raw_xa1 raw_xa2 raw_xa3 raw_xa4 raw_xa5 raw_xa6 raw_xa7 raw_xa8 raw_xa9 raw_xaa raw_xab raw_xac raw_xad raw_xae raw_xaf raw_xb0 raw_xb1 raw_xb2 raw_xb3 raw_xb4 raw_xb5 raw_xb6 raw_xb7 raw_xb8 raw_xb9 raw_xba raw_xbb raw_xbc raw_xbd raw_xbe raw_xbf raw_xc0 raw_xc1 raw_xc2 raw_xc3 raw_xc4 raw_xc5 raw_xc6 raw_xc7 raw_xc8 raw_xc9 raw_xca raw_xcb raw_xcc raw_xcd raw_xce raw_xcf raw_xd0 raw_xd1 raw_xd2 raw_xd3 raw_xd4 raw_xd5 raw_xd6 raw_xd7 raw_xd8 raw_xd9 raw_xda raw_xdb raw_xdc raw_xdd raw_xde raw_xdf raw_xe0 raw_xe1 raw_xe2 raw_xe3 raw_xe4 raw_xe5 raw_xe6 raw_xe7 raw_xe8 raw_xe9 raw_xea raw_xeb raw_xec raw_xed raw_xee raw_xef raw_xf0 raw_xf1 raw_xf2 raw_xf3 raw_xf4 raw_xf5 raw_xf6 raw_xf7 raw_xf8 raw_xf9 raw_xfa raw_xfb raw_xfc raw_xfd raw_xfe raw_xff
Used threads: none

4. TH_cweb_or_c_k — C++ or cweb type comments lexer.

C++ or cweb type comments lexer.

First set of called threads list.

TH_cweb_or_c_k

First set: raw_slash
Used threads: none

5. TH_c_comments — C++ type comments lexer.

C++ type comments lexer.

First set of called threads list.

TH_c_comments

First set: raw_slash
Used threads: none

6. TH_c_literal — C literal lexer.
C literal lexer.

First set of called threads list.

TH_c_literal

First set: raw_right_quote
Used threads: none

7. TH_c_string — C string lexer.
C string lexer.

First set of called threads list.

TH_c_string

First set: raw_dbl_quote
Used threads: none

8. TH_dbl_colon — This one's for the thread's name :: lexer.
This one's for the thread's name :: lexer.

First set of called threads list.

TH_dbl_colon

First set: raw_colon
Used threads: none

9. TH_eol — end-of-line recognizer — Unix, Mac, and Microsoft supported s... .
end-of-line recognizer — Unix, Mac, and Microsoft supported styles.

First set of called threads list.

TH_eol

First set: raw_lf raw_cr
Used threads: none

10. TH_err_symbols_ph_th — Parse Error vocabulary.
Parse Error vocabulary.

First set of called threads list.

TH_err_symbols_ph_th

TH_lint_balls

TH_ws

TH_eol

TH_c_comments

First set: LR1_questionable_shift_operator raw_ht raw_lf raw_vt raw_ff raw_cr raw_sp raw_open_bracket raw_slash
Used threads: TH_identifier TH_lint_balls TH_term_def_ph

11. TH_esc_seq — C type escape sequence recognizer.
C type escape sequence recognizer.

First set of called threads list.

TH_esc_seq

First set: raw_back_slash
Used threads: none

12. TH_fsm_class_phrase_th — Parse the fsm-class grammar construct.
Parse the fsm-class grammar construct.

First set of called threads list.

TH_fsm_class_phrase_th

TH_identifier

First set: LR1_questionable_shift_operator raw_A raw_B raw_C raw_D raw_E raw_F raw_G raw_H raw_I raw_J raw_K raw_L raw_M raw_N raw_O raw_P raw_Q raw_R raw_S raw_T raw_U raw_V raw_W raw_X raw_Y raw_Z raw_a raw_b raw_c raw_d raw_e raw_f raw_g raw_h raw_i raw_j raw_k raw_l raw_m raw_n raw_o raw_p raw_q raw_r raw_s raw_t raw_u raw_v raw_w raw_x raw_y raw_z

Used threads: TH_cweb_or_c_k TH_identifier TH_lint_balls TH_o2_sdc

13. TH_fsm_phrase_th — Parse grammar's fsm phrase along with its directive... .
Parse grammar's fsm phrase along with its directives.

First set of called threads list.

TH_fsm_phrase_th

TH_lint_balls

TH_ws

TH_eol

TH_c_comments

First set: LR1_questionable_shift_operator raw_ht raw_lf raw_vt raw_ff raw_cr raw_sp raw_open_bracket raw_slash

Used threads: TH_c_string TH_fsm_class_phrase_th TH_identifier TH_lint_balls

14. TH_identifier — Yacco2 identifiers lexer with symbol table lookup.
Yacco2 identifiers lexer with symbol table lookup.

First set of called threads list.

TH_identifier

First set: raw_A raw_B raw_C raw_D raw_E raw_F raw_G raw_H raw_I raw_J raw_K raw_L raw_M raw_N raw_O raw_P raw_Q raw_R raw_S raw_T raw_U raw_V raw_W raw_X raw_Y raw_Z raw_a raw_b raw_c raw_d raw_e raw_f raw_g raw_h raw_i raw_j raw_k raw_l raw_m raw_n raw_o raw_p raw_q raw_r raw_s raw_t raw_u raw_v raw_w raw_x raw_y raw_z

Used threads: none

15. TH_int_no — Integer number lexer.
Integer number lexer.

First set of called threads list.

TH_int_no

First set: raw_0 raw_1 raw_2 raw_3 raw_4 raw_5 raw_6 raw_7 raw_8 raw_9

Used threads: none

73. Crules_phrase — Dispatcher to parse the grammar's rules.
Dispatcher to parse the grammar's rules.

First set of called threads list.

Crules_phrase

TH_rules_phrase_th

TH_ws

TH_cweb_or_c_k

TH_eol

First set: LR1_questionable_shift_operator
Used threads: TH_rules_phrase_th

74. Crules_use_cnt — Optimization: Count "rules used" to lower new / de... .
Optimization: Count "rules used" to lower new / delete rule cycles while parsing.

First set of called threads list.

Crules_use_cnt

First set: rule_def
Used threads: none

75. Cterminals_phrase — Dispatcher to parse the terminals alphabet.
Dispatcher to parse the terminals alphabet.

First set of called threads list.

Cterminals_phrase

TH_terminals_phrase_th

TH_lint_balls

TH_ws

TH_eol

TH_c_comments

First set: LR1_questionable_shift_operator
Used threads: TH_terminals_phrase_th

79. First set control file (fsc) listing.
File : "yacco2.fsc"

```
              file-of-T-alphabet
"/usr/local/yacco2/compiler/grammars/yacco2_T_enumeration.fsc"
              emitfile "/usr/local/yacco2/compiler/grammars/yacco2_fsc.cpp"
              preamble
              #include "yacco2.h"
              #include "yacco2_T_enumeration.h"
              #include "yacco2_k_symbols.h"
              #include "yacco2_terminals.h"
              #include "yacco2_characters.h"
              using namespace NS_yacco2_T_enum;
              using namespace NS_yacco2_err_symbols;
              using namespace NS_yacco2_k_symbols;
              using namespace NS_yacco2_terminals;
              using namespace NS_yacco2_characters;
              end-preamble
              "/usr/local/yacco2/compiler/grammars/rules_use_cnt.fsc"
              "/usr/local/yacco2/compiler/grammars/enumerate_grammar.fsc"
              "/usr/local/yacco2/compiler/grammars/enumerate_T_alphabet.fsc"
              "/usr/local/yacco2/compiler/grammars/angled_string.fsc"
              "/usr/local/yacco2/compiler/grammars/bad_char_set.fsc"
              "/usr/local/yacco2/compiler/grammars/c_comments.fsc"
              "/usr/local/yacco2/compiler/grammars/c_literal.fsc"
              "/usr/local/yacco2/compiler/grammars/c_string.fsc"
              "/usr/local/yacco2/compiler/grammars/cweb_or_c_k.fsc"
              "/usr/local/yacco2/compiler/grammars/cweave_fsm_sdc.fsc"
              "/usr/local/yacco2/compiler/grammars/cweave_lhs_sdc.fsc"
              "/usr/local/yacco2/compiler/grammars/cweave_sdc.fsc"
              "/usr/local/yacco2/compiler/grammars/cweave_T_sdc.fsc"
              "/usr/local/yacco2/compiler/grammars/cweb_put_k_into_ph.fsc"
              "/usr/local/yacco2/compiler/grammars/dbl_colon.fsc"
              "/usr/local/yacco2/compiler/grammars/eol.fsc"
              "/usr/local/yacco2/compiler/grammars/epsilon_rules.fsc"
              "/usr/local/yacco2/compiler/grammars/err_symbols_ph.fsc"
              "/usr/local/yacco2/compiler/grammars/err_symbols_ph_th.fsc"
              "/usr/local/yacco2/compiler/grammars/esc_seq.fsc"
              "/usr/local/yacco2/compiler/grammars/eval_phrases.fsc"
              "/usr/local/yacco2/compiler/grammars/fsc_file.fsc"
              "/usr/local/yacco2/compiler/grammars/fsm_phrase.fsc"
              "/usr/local/yacco2/compiler/grammars/fsm_phrase_th.fsc"
              "/usr/local/yacco2/compiler/grammars/fsm_class_phrase_th.fsc"
              "/usr/local/yacco2/compiler/grammars/identifier.fsc"
              "/usr/local/yacco2/compiler/grammars/int_no.fsc"
              "/usr/local/yacco2/compiler/grammars/la_expr.fsc"
              "/usr/local/yacco2/compiler/grammars/la_expr_lexical.fsc"
              "/usr/local/yacco2/compiler/grammars/la_expr_src.fsc"
              "/usr/local/yacco2/compiler/grammars/la_lrk_T.fsc"
              "/usr/local/yacco2/compiler/grammars/lr1_k_phrase.fsc"
              "/usr/local/yacco2/compiler/grammars/link_cleanser.fsc"
              "/usr/local/yacco2/compiler/grammars/linker_id.fsc"
```

80. Index.

o2linker_doc.w

Date: October 27, 2014 at 12:08

File: o2linker_doc.w

Fsc : yacco2.fsc

11.3.2 So what value is this document to me?

I've not had to use it too often so u could consider it a tree mulcher or dust collector. But it does have value and it has oriented my debugging efforts. I've found it useful in the following few ways:

1. As a way to assess your overall grammar design by calls and whether to combine common or competing threads. It allows u to evaluate the scope between a forest versus the trees.

2. When an error message has happened and you're knee deep in thread calls, this is where the grammar call graphs help u overview the problem rather than dealing only in the narrow scope of error disclosure. At least the document provides a crude mapping on the inter-relationships of grammar calls. From the bottom-up the errant token could be crossed referenced against this document.

3. Why didn't that thread get called? Ohh my a O_2 bug? Possibly at least the evidence is in document form and not some compressed C++ coded table form. Though described in the following chapter, the individual documents express these tables well in non C++ terms.

Yes the document is sketchy. Some of my own wishes would be: who generated that T? Who uses that T? The individual grammar document deals with this but not at an overall design level. Stay tuned to future enhancements. Now onto the individual grammar documents: the jewels in the tiara?.

Chapter 12

Jewels of O_2

Yuo the reader have demonstrated patience. So lets get it on with these so called rhind stones. Each grammar has 2 outputted documents where xxx.pdf is your coding document and xxx_idx.pdf is your "lr table" supplement. O_2's print option "-p" generates the grammar's documents. Here is a sample from the "subrules_phrase.lex" grammar. There can be some running comments against each page to explain some of the typesetting highlighters: for example superscripting, underlining of symbols. The second document has its own symbol legends inside it with explanations specific to each lr table item.

12.1 Rhind stone cowboy: grammar document

The document is composed as follows:

1. The grammar's introduction, its `fsm`, and annotated pictorials per rule with your C++ code and any of your accompanying comments. Each grammar's production is drawn in similar fashion to Pascal's railroad diagrams.
2. The "first set control" file for O_2linker.
3. Emitted lr state network in readable form rather than its C++ code. It is the grammar's emitted state network compendium. Items like lookahead sets used in reducing a subrule use a symbolic symbol: for example LA 1, LA 2 etc. The 2nd document provides their contents and how they were calculated.

   The emitted tables also contain C++ comments of the grammar components making up table items. Using the document against the C++ tables should be adequate for the curious, the desperate, and the venturous.
4. Index of all grammar symbols along with your own index directives.
5. A cover sheet acting as a table of contents.

cweave generates the document while *mpost* draws the grammar's rules. This document is your normal reference when writing grammar code. If the grammar file is ill structured or the grammar is not an lr1 grammar then the raw text file is your only source for problem resolution. Not passing the lr1 constraint is currently a weakness in not at least being able to generate a document to review the grammar inconsistencies within the partially derived lr1 state part of the document: a future promise to correct this weakness.

Legend of graphic symbols:

1. A solid oval with symbol inside it identifies a T.
2. A solid rectangle encloses the specific rule.
3. The dotted form of the above symbols represent the thread call expression. The called thread is enclosed by the dotted rectangle.

4. Left recursion is represented by a dotted curve covering the repeating expression.

5. Due to possibly my ignorance or a *mpost* limitation, O_2 draws a small solid oval with a dot "." inside it to represent an epsilonal subrule: ϵ . Outside of the drawn graphics, the ϵ symbol is used particularly in the lr1 State Network.

6. Where there is "syntax directed code" associated with the subrule, each drawn subrule symbol is positional labeled starting from 1. This is the number used in your C++ stack frame reference: the prefixing access segment "sf→px__ ..." where x indicates the position to the specific stacked symbol.

Here is grammar subrules_phrase's document.

1. Copyright.

2. *subrules_phrase* **Thread.**
Parse subrule definitions.

3. Fsm Csubrules_phrase class.

4. Csubrules_phrase constructor directive.

⟨ Csubrules_phrase constructor directive 4 ⟩ ≡
```
subrules_phrase_ = 0;
```

5. Csubrules_phrase op directive.

⟨ Csubrules_phrase op directive 5 ⟩ ≡
```
if (subrules_phrase_ ≠ 0) {
  delete subrules_phrase_;
  subrules_phrase_ = 0;
}
subrules_phrase_ = new T_subrules_phrase;
subrules_phrase_→set_rc(*parser__→start_token__, __FILE__, __LINE__);
AST * t = new AST(*subrules_phrase_);
subrules_phrase_→phrase_tree(t);
subrule_no_ = 0;
```

6. Csubrules_phrase user-declaration directive.

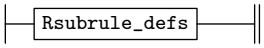
⟨ Csubrules_phrase user-declaration directive 6 ⟩ ≡
```
public: T_subrules_phrase * subrules_phrase_;
  int subrule_no_;
```

7. Csubrules_phrase user-prefix-declaration directive.

⟨ Csubrules_phrase user-prefix-declaration directive 7 ⟩ ≡
```
#include "lint_balls.h"
#include "eol.h"
#include "c_comments.h"
#include "identifier.h"
#include "c_string.h"
#include "subrule_def.h"
```

8. *Rsubrules_phrase* **rule.**

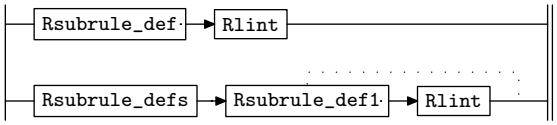
Rsubrules_phrase



⟨ Rsubrules_phrase subrule 1 op directive 8 ⟩ ≡
```
Csubrules_phrase * fsm = ( Csubrules_phrase * ) rule_info__.parser__→fsm_tbl__;
RSVP(fsm→subrules_phrase_);
fsm→subrules_phrase_ = 0;
```

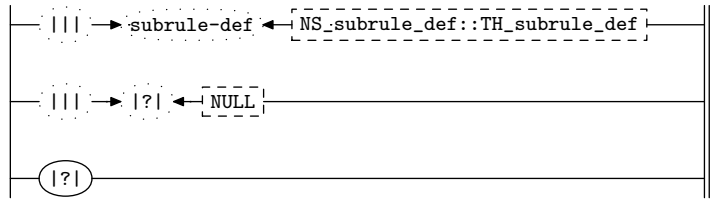
9. *Rsubrule_defs* **rule.**

Rsubrule_defs

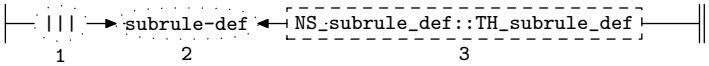


10. *Rsubrule_def* **rule.**

Rsubrule_def



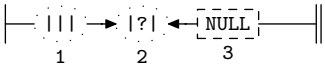
11. *Rsubrule_def* **'s subrule 1.**



⟨ Rsubrule_def subrule 1 op directive 11 ⟩ ≡

```
Csubrules_phrase * fsm = ( Csubrules_phrase * ) rule_info__.parser__→fsm_tbl__;
++fsm→subrule_no_;
sf→p2__→subrule_no_of_rule(fsm→subrule_no_);
fsm→subrules_phrase_→add_sr_to_subrules(sf→p2__);
```

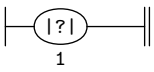
12. *Rsubrule_def* **'s subrule 2.**



⟨ Rsubrule_def subrule 2 op directive 12 ⟩ ≡

```
RSVP(sf→p2__);
rule_info__.parser__→set_stop_parse(true);
```

13. *Rsubrule_def* **'s subrule 3.**

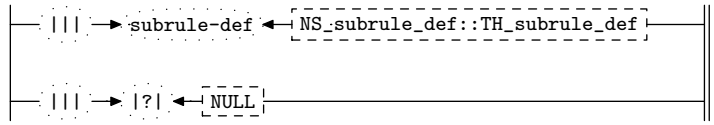


⟨ Rsubrule_def subrule 3 op directive 13 ⟩ ≡

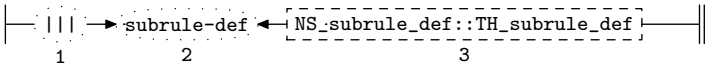
```
CAbs_lr1_sym * sym = new Err_no_sub_rule_present;
sym→set_rc(*rule_info__.parser__→current_token( ), __FILE__, __LINE__);
RSVP(sym);
rule_info__.parser__→set_stop_parse(true);
```

14. *Rsubrule_def1* **rule.**

Rsubrule_def1



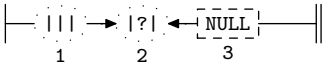
15. *Rsubrule_def1* **'s subrule 1.**



⟨ Rsubrule_def1 subrule 1 op directive 15 ⟩ ≡

```
Csubrules_phrase * fsm = ( Csubrules_phrase * ) rule_info__.parser__->fsm_tbl__;
++fsm->subrule_no_;
sf->p2__->subrule_no_of_rule(fsm->subrule_no_);
fsm->subrules_phrase_->add_sr_to_subrules(sf->p2__);
```

16. *Rsubrule_def1* **'s subrule 2.**

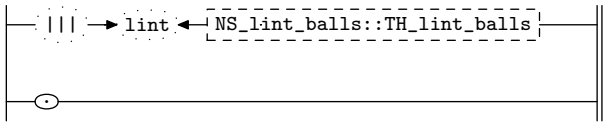


⟨ Rsubrule_def1 subrule 2 op directive 16 ⟩ ≡

```
RSVP(sf->p2__);
rule_info__.parser__->set_stop_parse(true);
```

17. *Rlint* **rule.**

Rlint



18. First Set Language for O_2^{linker}.

```
/*
 File: subrules_phrase.fsc
 Date and Time: Fri Jan  2 15:33:56 2015
*/
transitive   y
grammar-name "subrules_phrase"
name-space   "NS_subrules_phrase"
thread-name  "TH_subrules_phrase"
monolithic   n
file-name    "subrules_phrase.fsc"
no-of-T      569
list-of-native-first-set-terminals 1
   LR1_questionable_shift_operator
end-list-of-native-first-set-terminals
list-of-transitive-threads 1
  NS_subrule_def::TH_subrule_def
end-list-of-transitive-threads
list-of-used-threads 2
  NS_lint_balls::TH_lint_balls
  NS_subrule_def::TH_subrule_def
end-list-of-used-threads
fsm-comments
"Parse subrules: into the valley of ..."
```

12.2 Comments on the lr state table

To make the lr1 table more understandable, the following conventions are used. Each state has its "vectored in" symbol except for the closure state (state 1), and its type: shift only, reduce only, shift/reduce due to the subrules within it. If the "vectored in symbol" is ||| which indicates that threading is a happening, it is accompanied by the arbitrator-code procedure be it ϵ or the generated "AR_rule-name" procedure provided by one of the threading subrule's parent rule.

Each rule's subrule configuration is listed making up the state and is tagged as to how it got included in the state configuration: "c" for closure supplied by a rule in the configuration or a "t" as a transitioning subrule from another state. The subrule element column contains the current symbol in the subrule expression being worked on and its possibly suffixed symbol string following it. Two reasons for this: it quickly showed programming errors, visual clue to the subrule contents, and provided strings of symbols for a rule's follow set calculation. Prefixing this column are the rule's coordinates identifying it: rule number, its subrule number and current symbol position being worked on. Due to space the number of symbols printed are constrained by the column width.

The suffixed symbols contributing to a rule's follow set are underlined if its current symbol being worked on is a rule. Epsilonal rules in the subrule expression are marked by a superscript ϵ symbol against the rule name. The balance of the subrule configuration indicates the lr transitioning on the subrule symbols: what state number it was born in, its vectored into state number, followed by where the subrule is reduced. When the subrule configuration is being reduced, a symbolic LA set number is given. Due to space the "lr state network" supplement document: (xxx_idx.pdf where xxx is the grammar name) contains the contents of this symbolic LA set.

19. Lr1 State Network.

⇒ State: 1 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
c	Rsubrule_def		3	3	1	|?|		1	2	2	
c	Rsubrule_def		3	1	1	||| subrule-def NS_subrule_def::TH_subrule_def		1	3	5	
c	Rsubrule_def		3	2	1	||| |?| NULL		1	3	4	
c	Rsubrule_defs		2	2	1	Rsubrule_defs *Rsubrule_def*1		1	6	13	
c	Rsubrules_phrase		1	1	1	Rsubrule_defs		1	6	6	
c	Rsubrule_defs		2	1	1	Rsubrule_def *Rlint*$^{\epsilon}$		1	14	15	

⇒\|?\| State: 2 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def		3	3	2			1	0	2	1

⇒\|\|\| *arbitration−code*: ϵ State: 3 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def		3	2	2	|?|		1	4	4	
t	Rsubrule_def		3	1	2	subrule-def		1	5	5	

⇒\|?\| State: 4 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def		3	2	3			1	0	4	1

⇒*subrule−def* State: 5 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def		3	1	3			1	0	5	1

⇒*Rsubrule_defs* State: 6 state type: $^{s/r}$

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrules_phrase		1	1	2			1	0	6	2
c	Rsubrule_def1		4	1	1	||| subrule-def NS_subrule_def::TH_subrule_def		6	7	9	
c	Rsubrule_def1		4	2	1	||| |?| NULL		6	7	8	
t	Rsubrule_defs		2	2	2	Rsubrule_def1 *Rlint*$^{\epsilon}$		1	10	13	

⇒\|\|\| *arbitration−code*: ϵ State: 7 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def1		4	2	2	|?|		6	8	8	
t	Rsubrule_def1		4	1	2	subrule-def		6	9	9	

⇒\|?\| State: 8 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def1		4	2	3			6	0	8	1

⇒*subrule−def* State: 9 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def1		4	1	3			6	0	9	1

⇒*Rsubrule_def*1 State: 10 state type: $^{s/r}$

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
c	Rlint		5	2	1	ϵ		10	0	10	1
c	Rlint		5	1	1	||| lint NS_lint_balls::TH_lint_balls		10	11	12	

t	Rsubrule_defs	2	2	3	Rlint			1	13	13	

⇒||| *arbitration−code*: ϵ State: 11 state type: ^{s}

←	rule	→	R#	sr#	Po	←	subrule element	→	Brn	Gto	Red	LA
t	Rlint		5	1	2	lint			10	12	12	

⇒*lint* State: 12 state type: ^{r}

←	rule	→	R#	sr#	Po	←	subrule element	→	Brn	Gto	Red	LA
t	Rlint		5	1	3				10	0	12	1

⇒*Rlint* State: 13 state type: ^{r}

←	rule	→	R#	sr#	Po	←	subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_defs		2	2	4				1	0	13	1

⇒*Rsubrule_def* State: 14 state type: $^{s/r}$

←	rule	→	R#	sr#	Po	←	subrule element	→	Brn	Gto	Red	LA
c	Rlint		5	2	1	ϵ			14	0	14	1
c	Rlint		5	1	1	|||	lint NS_lint_balls::TH_lint_balls		14	11	12	
t	Rsubrule_defs		2	1	2	Rlint			1	15	15	

⇒*Rlint* State: 15 state type: ^{r}

←	rule	→	R#	sr#	Po	←	subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_defs		2	1	3				1	0	15	1

20. Index.

subrules_phrase Grammar

Date: January 2, 2015 at 15:39

File: subrules_phrase.lex Ns: NS_subrules_phrase

Version: 1.0 Debug: false

Grammar Comments: Type: Thread

Parse subrules: into the valley of ...

562 element(s) in Lookahead Expression below

eolr − |||

12.3 Rinds and stones: grammar's "lr1 state" supplement

This document expresses "the what" and "the how" of various lr state items's contents. Each grammar's supplement has the following naming convention: "grammar-name_idx.pdf" with the following sections:

1. A summary of the emitted lr1 states.
2. Symbols used.
3. "first set" per rule.
4. Each rule's generated subrules used throughout the generated state network.
5. List of reduced states with their reduce type: reduce/reduce, shift/reduce, mix of both.
6. Each state rule's "follow set" and how their content is derived. The local yield is listed along with inheritance from other rule follow sets.
7. Common merged lookahead sets with their content used per reducing subrule.

The document contains explanations pertaining to each expressed component. To conserve book space, the 2nd page of the LA 1 set's contents are not shown along with the LA 2 set as the notion of their content is adequate for u the reader to grasp.

1. Grammar symbols: Used cross reference.
Reference of each grammar's symbol used within each rule's productions. The index uses the tripple: rule name, its subrule no, and the symbol's position within the symbol string.

2. NS_lint_balls::TH_lint_balls:.
Rlint 1.3

3. NS_subrule_def::TH_subrule_def:.
Rsubrule_def 1.3 Rsubrule_def1 1.3

4. NULL thread:.
Rsubrule_def 2.3 Rsubrule_def1 2.3

5. Rlint:.
Rsubrule_defs 1.2 Rsubrule_defs 2.3

6. Rsubrule_def:.
Rsubrule_defs 1.1

7. Rsubrule_def1:.
Rsubrule_defs 2.2

8. Rsubrule_defs:.
Rsubrules_phrase 1.1 Rsubrule_defs 2.1

9. ϵ **:.**
Rlint 2.1

10. lint:.
Rlint 1.2

11. subrule-def:.
Rsubrule_def 1.2 Rsubrule_def1 1.2

12. |?|:.
Rsubrule_def 2.2 Rsubrule_def 3.1 Rsubrule_def1 2.2

13. |||:.
Rsubrule_def 1.1 Rsubrule_def 2.1 Rsubrule_def1 1.1 Rsubrule_def1 2.1 Rlint 1.1

14. Grammar Rules's First Sets.

15. *Rsubrules_phrase* **# in set: 2.**
|?| |||

16. *Rsubrule_defs* **# in set: 2.**
|?| |||

17. *Rsubrule_def* **# in set: 2.**
|?| |||

18. *Rsubrule_def1* **# in set: 1.**
|||

19. *Rlint*$^{\epsilon}$ **# in set: 1.**
|||

20. LR State Network.
List of productions with their derived LR state lists. Their subrule number and symbol string indicates the specific production being derived. The "▷" symbol indicates the production's list of derived states from its closured state. Multiple lists within a production indicate 1 of 2 things:

1) derived string that could not be merged due to a lr(1) conflict
2) partially derived string merged into another derived lr states

A partially derived string is indicated by the "merged into" symbol ↗used as a superscript along with the merged into state number.

21. Rsubrules_phrase.

1 Rsubrule_defs
▷ 1 6

22. Rsubrule_defs.

1 Rsubrule_def Rlint
▷ 1 14 15
2 Rsubrule_defs Rsubrule_def1 Rlint
▷ 1 6 10 13

23. Rsubrule_def.

1 ||| subrule-def NS_subrule_def::TH_subrule_def
▷ 1 3 5
2 ||| |?| NULL
▷ 1 3 4
3 |?|
▷ 1 2

24. Rsubrule_def1.

1 ||| subrule-def NS_subrule_def::TH_subrule_def
▷ 6 7 9
2 ||| |?| NULL
▷ 6 7 8

25. Rlint.

1 ||| lint NS_lint_balls::TH_lint_balls
▷ 10 11 12
▷ 14↗11
2 ϵ
▷ 10
▷ 14

26. List of reducing states.
The following legend indicates the type of reducing state.
Points 2–4 are states that must meet the lr(1) condition:
1) r — only 1 production reducing
2) r^2 — 2 or more reducing productions
3) s/r — shift and 1 reducing production
4) s/r^2 — shift and multiple reducing productions

⊂ 2^r 4^r 5^r $6^{s/r}$ 8^r 9^r $10^{s/r}$ 12^r 13^r $14^{s/r}$ 15^r

27. Lr1 State's Follow sets and reducing lookahead sets.
Notes on Follow set expressions:
1) The "follow set" for rule uses its literal name and tags its grammar rule rank number as a superscript. Due to space limitations, part of the follow set information uses the rule's literal name while the follow set expressions refers to the rule's rank number. This < rule name, rule rank number > tupple allows you the reader to decifer the expressions. Transitions are represented by S_xR_z whereby S is the LR1 state identified by its "x" subscript where other transient calculations occur within the LR1 state network. R indicates the follow set rule with the subscript "z" as its grammar rank number that contributes to the follow set.

The $\nearrow^x$ symbol indicates that a merge into state "x" has taken place. That is, the reduced subrule that depends on this follow set finds its follow set in 2 places: its birthing state that generated the sequence up to the merged into state, and the birthing state that generated the "merged into" state. So the rule's "follow set" calculation must also continue its calculation within the birth state generating the "x merged into" state.

State: 1 Follow Set contributors, merges, and transitions
← Follow set Rule → ← follow set symbols contributors →
Rsubrules_phrase[1]
Local follow set yield:
x06, x07, x08, x09, x0a, x0b, x0c, x0d, x0e, x0f, x10, U, x11, V, W, X, Y, Z, [, \ ,], ^, _, ‘, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, {, |, }, , x7f, x80, x81, x82, x83, x84, x85, x86, x87, x88, x89, x8a, x8b, x8c, x8d, x8e, x8f, x90, x91, x92, x93, x94, x95, x96, x97, x98, x99, x9a, x9b, x9c, x9d, x9e, x9f, xa0, xa1, xa2, xa3, xa4, xa5, xa6, xa7, xa8, xa9, xaa, xab, xac, xad, xae, xaf, xb0, xb1, xb2, xb3, xb4, xb5, xb6, xb7, xb8, xb9, xba, xbb, xbc, xbd, xbe, xbf, xc0, xc1, xc2, xc3, xc4, xc5, xc6, xc7, xc8, xc9, xca, xcb, xcc, xcd, xce, xcf, xd0, xd1, xd2, xd3, xd4, xd5, xd6, xd7, xd8, xd9, xda, xdb, xdc, xdd, xde, xdf, xe0, xe1, xe2, xe3, xe4, xe5, xe6, xe7, xe8, xe9, xea, xeb, xec, xed, xee, xef, xf0, xf1, eog, x12, x13, x14, x15, x16, x17, x18, xf2, x19, x1a, x1b, x1c, x1d, x1e, x1f, , !, ", # , xf3, xf4, xf5, xf6, xf7, xf8, xf9, xfa, xfb, xfc, xfd, xfe, xff, option-t, option-p, option-err, esc-seq, eol, basic-char, raw-char, comment, ws, angled-string, c-literal, c-string, xc-str, unquoted-string, file-inclusion, identifier, int-no, # raw-characters, # lr1-constant-symbols, # error-symbols, # ***, # AD, # AB, # parallel-la-boundary, # arbitrator-code, # parallel-parser, # parallel-thread-function, # parallel-control-monitor, # fsm, # fsm-id, # fsm-filename, # fsm-namespace, # fsm-class, # fsm-version, # fsm-date, # fsm-debug, # fsm-comments, # terminals, # T-enumeration, # file-name, # name-space, # sym-class, # rules, # lhs, # user-declaration, # user-prefix-declaration, # user-suffix-declaration, # constructor, # destructor, # op, # failed, # user-implementation, # user-imp-tbl, # user-imp-sym, # constant-defs, # terminals-refs, # terminals-sufx, # lrk-sufx, - > , # NULL, ::, block, syntax-code, fsm-class-phrase, fsm-phrase, parallel-parser-phrase, T-enum-phrase, terminal-def, table-entry, sym-tbl-report-card, terminals-phrase, error-symbols-phrase, lr1-k-phrase, rc-phrase, rule-lhs-phrase, parallel-monitor-phrase, rule-def, rules-phrase, subrule-def, subrules-phrase, T-in-stbl, refered-T, rule-in-stbl, refered-rule, transitive, grammar-name, thread-name, monolithic, no-of-T, list-of-native-first-set-terminals, end-list-of-native-first-set-terminals, list-of-transitive-threads, end-list-of-transitive-threads, emitfile, preamble, end-preamble, T-alphabet, end-T-alphabet, file-of-T-alphabet, T-attributes, tth-in-stbl, thread-attributes, th-in-stbl, kw-in-stbl, la-express-source, eosubrule, called thread eosubrule, null call thread eosubrule, cweb-comment, grammar-phrase, cweb-marker, lint, list-of-used-threads, end-list-of-used-threads, nested files exceeded, no end-of-code, no cmd-lne-data, no file-name, bad filename, bad filename to output grammar header, bad filename to output cpp, bad filename to output sym, bad filename to output tbl, bad filename to output enumeration

Non-terminal (rule) outside Rules's construct, misplaced or misspelt Rule or T outside of Rules defs, not a Rule in chained dispatcher expr, Empty file no grammar constructs present, term not a lhs or parallel-control-monitor kw, x00, x01, x02, x03, x04, x05, $, %, &, ', (,), *, +, ,, -, ., /, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, :, ;, < , =, > , ?, @, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T.

← Follow set Rule → ← follow set symbols contributors →

Rsubrule_defs[2] $R_{2\cdot2\cdot1}$ $R_{1\cdot1\cdot1}$ $S_1 R_1$

Local follow set yield:

|||.

← Follow set Rule → ← follow set symbols contributors →

Rsubrule_def[3] $R_{2\cdot1\cdot1}$ $R_{2\cdot1\cdot2}$ $S_1 R_2$

Local follow set yield:

|||.

State: 6 Follow Set contributors, merges, and transitions

← Follow set Rule → ← follow set symbols contributors →

Rsubrule_def1[4] $R_{2\cdot2\cdot2}$ $R_{2\cdot2\cdot3}$ $S_1 R_2$

Local follow set yield:

|||.

State: 10 Follow Set contributors, merges, and transitions

← Follow set Rule → ← follow set symbols contributors →

Rlint[5] $R_{2\cdot2\cdot3}$ $\nearrow^{14}$ $S_1 R_2$

Local follow set yield:

State: 14 Follow Set contributors, merges, and transitions

← Follow set Rule → ← follow set symbols contributors →

Rlint[5] $R_{2\cdot1\cdot2}$ $S_1 R_2$

Local follow set yield:

28. Common Follow sets.

29. LA set: 1.

x06, x07, x08, x09, x0a, x0b, x0c, x0d, x0e, x0f, x10, U, x11, V, W, X, Y, Z, [, \ ,], ^, _, ‘, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, {, |, }, , x7f, x80, x81, x82, x83, x84, x85, x86, x87, x88, x89, x8a, x8b, x8c, x8d, x8e, x8f, x90, x91, x92, x93, x94, x95, x96, x97, x98, x99, x9a, x9b, x9c, x9d, x9e, x9f, xa0, xa1, xa2, xa3, xa4, xa5, xa6, xa7, xa8, xa9, xaa, xab, xac, xad, xae, xaf, xb0, xb1, xb2, xb3, xb4, xb5, xb6, xb7, xb8, xb9, xba, xbb, xbc, xbd, xbe, xbf, xc0, xc1, xc2, xc3, xc4, xc5, xc6, xc7, xc8, xc9, xca, xcb, xcc, xcd, xce, xcf, xd0, xd1, xd2, xd3, xd4, xd5, xd6, xd7, xd8, xd9, xda, xdb, xdc, xdd, xde, xdf, xe0, xe1, xe2, xe3, xe4, xe5, xe6, xe7, xe8, xe9, xea, xeb, xec, xed, xee, xef, xf0, xf1, eog, |r|, x12, x13, x14, x15, x16, x17, x18, xf2, x19, x1a, x1b, x1c, x1d, x1e, x1f, , !, ", # , xf3, xf4, xf5, xf6, xf7, xf8, xf9, xfa, xfb, xfc, xfd, xfe, xff, option-t, option-p, option-err, esc-seq, eol, basic-char, raw-char, comment, ws, angled-string, c-literal, c-string, xc-str, unquoted-string, file-inclusion, identifier, int-no, # raw-characters, # lr1-constant-symbols, # error-symbols, # ***, # AD, # AB, # parallel-la-boundary, # arbitrator-code, # parallel-parser, # parallel-thread-function, # parallel-control-monitor, # fsm, # fsm-id, # fsm-filename, # fsm-namespace, # fsm-class, # fsm-version, # fsm-date, # fsm-debug, # fsm-comments, # terminals, # T-enumeration, # file-name, # name-space, # sym-class, # rules, # lhs, # user-declaration, # user-prefix-declaration, # user-suffix-declaration, # constructor, # destructor, # op, # failed, # user-implementation, # user-imp-tbl, # user-imp-sym, # constant-defs, # terminals-refs, # terminals-sufx, # lrk-sufx, - > , # NULL, ::, block, syntax-code, fsm-class-phrase, fsm-phrase, parallel-parser-phrase, T-enum-phrase, terminal-def, table-entry, sym-tbl-report-card, terminals-phrase, error-symbols-phrase, lr1-k-phrase, rc-phrase, rule-lhs-phrase, parallel-monitor-phrase, rule-def, rules-phrase, subrule-def, subrules-phrase, T-in-stbl, refered-T, rule-in-stbl, refered-rule, transitive, grammar-name, thread-name, monolithic, no-of-T, list-of-native-first-set-terminals, end-list-of-native-first-set-terminals, list-of-transitive-threads, end-list-of-transitive-threads, emitfile, preamble, end-preamble, T-alphabet, end-T-alphabet, file-of-T-alphabet, T-attributes, tth-in-stbl, thread-attributes, th-in-stbl, kw-in-stbl, la-express-source, eosubrule, called thread eosubrule, null call thread eosubrule, cweb-comment, grammar-phrase, cweb-marker, lint, list-of-used-threads, end-list-of-used-threads, nested files exceeded, no end-of-code, no cmd-lne-data, no file-name, bad filename, bad filename to output grammar header, bad filename to output cpp, bad filename to output sym, bad filename to output tbl, bad filename to output enumeration header, bad filename for Errors vocabulary header, bad filename for Errors vocabulary implementation, bad cmd-opt, bad int-no, bad int-no range, no int present, bad eos, bad esc, comment-overrun, bad char, bad univ-seq, improper closing of rules construct, no identifier present, no directive present, duplicate directive, no syntax-code present, no open-parenthesis, no close-parenthesis, no fsm-id-present, no fsm-id-string present, no fsm-filename present, no fsm-filename id present, no comma present, no key-value present in definition, no fsm-namespace present, no fsm-namespace id present, no fsm-class present, no fsm-version present, no fsm-version string present, no fsm-date present, no fsm-date string present, no fsm-debug present, no fsm-debug string present, no fsm-comments present, no fsm-comments string present, invalid fsm-debug value, no parallel-thread-function, no parallel-control-monitor, no parallel thread function, no parallel-la-boundary, no parallel-la-boundary-expr, no ***, no parallel-code, no parallel-code-syntax-code, not an arbitration-code keyword, no open-brace, no close-brace, no constant-defs-directive present, no file-name present, no file-name-id present, no name-space present, no name-space-id present, no constant-defs-code present, no constant-defs keyword present, no terminal-def-code present, no symbol definition present, duplicate-entry in alphabet, already defined AB tag, already defined AD tag, improper directive, no sym-class present, no

30. LA set: 2.

x06, x07, x08, x09, x0a, x0b, x0c, x0d, x0e, x0f, x10, U, x11, V, W, X, Y, Z, [, \ ,], ^, _, ‘, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, {, |, }, , x7f, x80, x81, x82, x83, x84, x85, x86, x87, x88, x89, x8a, x8b, x8c, x8d, x8e, x8f, x90, x91, x92, x93, x94, x95, x96, x97, x98, x99, x9a, x9b, x9c, x9d, x9e, x9f, xa0, xa1, xa2, xa3, xa4, xa5, xa6, xa7, xa8, xa9, xaa, xab, xac, xad, xae, xaf, xb0, xb1, xb2, xb3, xb4, xb5, xb6, xb7, xb8, xb9, xba, xbb, xbc, xbd, xbe, xbf, xc0, xc1, xc2, xc3, xc4, xc5, xc6, xc7, xc8, xc9, xca, xcb, xcc, xcd, xce, xcf, xd0, xd1, xd2, xd3, xd4, xd5, xd6, xd7, xd8, xd9, xda, xdb, xdc, xdd, xde, xdf, xe0, xe1, xe2, xe3, xe4, xe5, xe6, xe7, xe8, xe9, xea, xeb, xec, xed, xee, xef, xf0, xf1, eog, x12, x13, x14, x15, x16, x17, x18, xf2, x19, x1a, x1b, x1c, x1d, x1e, x1f, , !, ", # , xf3, xf4, xf5, xf6, xf7, xf8, xf9, xfa, xfb, xfc, xfd, xfe, xff, option-t, option-p, option-err, esc-seq, eol, basic-char, raw-char, comment, ws, angled-string, c-literal, c-string, xc-str, unquoted-string, file-inclusion, identifier, int-no, # raw-characters, # lr1-constant-symbols, # error-symbols, # ***, # AD, # AB, # parallel-la-boundary, # arbitrator-code, # parallel-parser, # parallel-thread-function, # parallel-control-monitor, # fsm, # fsm-id, # fsm-filename, # fsm-namespace, # fsm-class, # fsm-version, # fsm-date, # fsm-debug, # fsm-comments, # terminals, # T-enumeration, # file-name, # name-space, # sym-class, # rules, # lhs, # user-declaration, # user-prefix-declaration, # user-suffix-declaration, # constructor, # destructor, # op, # failed, # user-implementation, # user-imp-tbl, # user-imp-sym, # constant-defs, # terminals-refs, # terminals-sufx, # lrk-sufx, - > , # NULL, ::, block, syntax-code, fsm-class-phrase, fsm-phrase, parallel-parser-phrase, T-enum-phrase, terminal-def, table-entry, sym-tbl-report-card, terminals-phrase, error-symbols-phrase, lr1-k-phrase, rc-phrase, rule-lhs-phrase, parallel-monitor-phrase, rule-def, rules-phrase, subrule-def, subrules-phrase, T-in-stbl, refered-T, rule-in-stbl, refered-rule, transitive, grammar-name, thread-name, monolithic, no-of-T, list-of-native-first-set-terminals, end-list-of-native-first-set-terminals, list-of-transitive-threads, end-list-of-transitive-threads, emitfile, preamble, end-preamble, T-alphabet, end-T-alphabet, file-of-T-alphabet, T-attributes, tth-in-stbl, thread-attributes, th-in-stbl, kw-in-stbl, la-express-source, eosubrule, called thread eosubrule, null call thread eosubrule, cweb-comment, grammar-phrase, cweb-marker, lint, list-of-used-threads, end-list-of-used-threads, nested files exceeded, no end-of-code, no cmd-lne-data, no file-name, bad filename, bad filename to output grammar header, bad filename to output cpp, bad filename to output sym, bad filename to output tbl, bad filename to output enumeration header, bad filename for Errors vocabulary header, bad filename for Errors vocabulary implementation, bad cmd-opt, bad int-no, bad int-no range, no int present, bad eos, bad esc, comment-overrun, bad char, bad univ-seq, improper closing of rules construct, no identifier present, no directive present, duplicate directive, no syntax-code present, no open-parenthesis, no close-parenthesis, no fsm-id-present, no fsm-id-string present, no fsm-filename present, no fsm-filename id present, no comma present, no key-value present in definition, no fsm-namespace present, no fsm-namespace id present, no fsm-class present, no fsm-version present, no fsm-version string present, no fsm-date present, no fsm-date string present, no fsm-debug present, no fsm-debug string present, no fsm-comments present, no fsm-comments string present, invalid fsm-debug value, no parallel-thread-function, no parallel-control-monitor, no parallel thread function, no parallel-la-boundary, no parallel-la-boundary-expr, no ***, no parallel-code, no parallel-code-syntax-code, not an arbitration-code keyword, no open-brace, no close-brace, no constant-defs-directive present, no file-name present, no file-name-id present, no name-space present, no name-space-id present, no constant-defs-code present, no constant-defs keyword present, no terminal-def-code present, no symbol definition present, duplicate-entry in alphabet, already defined AB tag, already defined AD tag, improper directive, no sym-class present, no

31. Index.

subrules_phrase_idx.w

Date: January 14, 2015 at 15:42

File: subrules_phrase_idx.w

12.4 Literate programming 5 aka 101

Please have a read on the following references: *Cweb* software tool set for literate programming [17], and "Literate Programming" [13]. The *Cweb* web document provides the index directives used by *cweave* in generating its document. O_2 uses them extensively. TEX macros are wideopen for your use and so u might have a look at the TEX reference [15]. A companion to "literate programming" is Metapost's drawing tool *mpost* program reference [6] written by John D. Hobby with its own drawing language. "The METAFONT book" by Donald E. Knuth [14] prepares u well on understanding/producing graphics. Its surgeon general warning is so true. I leave u to buy the book and draw your own...

The following macros I found useful to include in your documents:

1. Including text files examples with or without numbering the lines. The "cat" utility has an option "-n" to number the text lines.

   ```
   \let\setuplistinghook = \linenumberedlisting}
   \listing{"/yacco2/diagrams+etc/fsm_failed.txt"}
   ```

2. Including *mpost* drawings. Yac_2o_2 library uses *mpost* extensively. Some of O_2's grammars have customed drawings to complement my understanding.

   ```
   \convertMPtoPDF{"/yacco2/diagrams+etc/threading_defs.1"}{1}{1}
   ```

   This example reads the diagram "threading_defs.1" generated from the below *mpost* program:

   ```
   % file: threading_defs.mp
   input boxes
   pair sw,se,ne,nw;
   pair zx,zy;
   beginfig(1);
   u=1cc;
   % stbl
   sw:=(1u,0);
   se:=(2u,0);
   ne:=(2u,4u);
   nw:=(1u,4u);
   draw sw--se--ne--nw--cycle;
   %outline the tab;e
   for i=1 upto 3:
     zx:=(1u,i*u);
     zy:=(2u,i*u);
     if i = 3:
       draw zx{dir 45} .. .5[zx,zy] {dir -45} .. zy;
     else:
       draw zx--zy;
     fi
   endfor

   % label subscripts
   defaultscale:=.6;
   for i=0 upto 3:
     zy:=(2u,i*u);
     string a;
     if i=3:
       a := "no threads ids-1";
       label.rt(a,zy);
     else:
       a:=decimal(i);
       label.rt(a,zy);
     fi
   ```

```
endfor
defaultscale:=1;
label.bot(
         btex $Parallel\_thread\_table[thd $\#$].thread\_list$ etex
         ,(1u,0u)
         );

%outline table_entry
label.rt(btex $thread\_list:$ etex,(5u,6u));
        ...
label.rt(btex $\bullet\ $thd id\ --- thread id number etex,(8u,1u));
endfig;
end;
```

3. TeX reference. It is a typesetting mainstay: Macros and fonts and other excellent references, tutorials, and papers and TUG [19] and forums will get u acquainted for your own publishing efforts. Please consider joining TUG. It helps advance publishing efforts, and is good for everyone's health in this community and to your own learning experiences.

I will now address the points within a grammar where u can include your own optional comments for the grammar document and some of the typesetting directives used by *cweave*.

12.4.1 Points of interest: grammar stick-its

A typesetting comment starts with "/@" and closes off with a "@/". Its contents can be *cweave* directives, your drawings, and prose. All typesetting comments are optional. Being mum doesn't cut it for me. What about u? Here is a sample taken from the "epsilon_rules.lex" grammar.

```
/@
@** |epsilon_rules| grammar.\fbreak
Determine and check for the following:\fbreak
\ptindent{1: epsilon rules}
\ptindent{2: rules not deriving any string}
\ptindent{3: referenced rules are defined}
\ptindent{4: recursion cycling deriving only epsilon}
\ptindent{5: ambiguous subrules caused by multiple recursion cycling}
\fbreak
Epsilon definition:\fbreak
A rule having a subrule with no grammar symbols present:
ie, it derives an empty symbol string.
This is explicit epsilon.
A second condition is when a rule's subrule
has all its symbols as rules and they are
all epsiloned. I call this transient epsilon. It doesn't matter
how the condition is arrived at.
Not a third condition is the ``invisible-shift'' operator usage.
Though it's not part of the token stream per say it is normally used
to get out of ambiguous situations and so is part of the lookahead set.
Once upon a time i thought this was an
implicit epsilon: IT IS NOT.\fbreak
\fbreak
Some epsilon terminology:\fbreak
Noun: epsilon represents the empty string of symbols.
The Greek flavour ($\epsilon$) represents it visually.\fbreak
Adjective: epsilonable - has the empty symbol string condition
and non-epsilonable generates terminal strings.\fbreak
Verb: shake and bake the tenses.\fbreak
Verbal: epsiloned.\fbreak
\fbreak
```

```
This grammar demonstrates the elegance of flow control.
Review of a grammar's flow control:\fbreak
This depends on how the grammar tree
is walked. This grammar receives its
tokens in prefix formation:
a ``rule-def'' token is processed before
its ``subrule-def''.
Using sentences, a subrule's expression will be processed
before its left-handside rule.
This means the dependencies are in bottom-up left-to-right order of
token recognition.
One can use the prefix order to initialize the
various conditions, and
postfix the results from the bottom-up order.
A grammar of no tokens can be programmed using all epsiloned rules
whereby the rules act as logic triggers.
In this particular example, the grammar was originally
written to detect ``epsilon'' rules.
Now it detects whether all rules derive T. The grammar recognition
just loads up into a 2 dimensional list the rule x subrule x element.
Once the language is accepted, it then traverses the list to derive
the terminal strings per subrule.\fbreak
\fbreak
The Algorithm.\fbreak
Pass one: fill up the derives list\fbreak
Fill up the triplet list of all subrules:
rule $\otimes$ subrule $\otimes$ element.
One can view it as a 2 dimensional array where the
row is subrule dominant with its subordinate element columns.
The rule just tags along providing the parental anchor.
The grammar tree is walked top-down (prefix order) to
load up the derives list.
Only the ``rule-def'' and ``subrule-def'' come in as tokens
while the elements are filtered out.
This was done for efficiency reasons.
Within the grammar at the ``subrule-def'' logic point,
the ``subrule-def''  start element is fetched by
syntax directed code.
\fbreak
\fbreak
Pass 2: derive T for each subrule\fbreak
Try to derive terminal strings for each indeterminate subrule.
This stage does multiple passes thru the derives list
whereby each
pass is predicated on the fact that the previous pass
has a change in the status of a subrule
--- advance to the next element or
the subrule's elements have been consumed.
How is this a 2 dimensional walk? The subrules in the list are
the dominant axis while its elements are the
secondary axis. This is a dynamic grid of points being consumed.
The final outcome is either the grammar's subrules all derive
terminal strings and the derives list is empty
or the remainder of items are
the pathological subrules.\fbreak
\fbreak
Examples of Pathological conditions:\fbreak
Rule {\bf A} calls itself or
a variation of {\bf A} calls {\bf B} calls {\bf A} without
```

```
any terminal strings in their subrules.\fbreak
\fbreak
Please look at the following test grammars that
exercise the pathos...\fbreak
\ptindent{1: |ts_path1.lex|}
\ptindent{2: |ts_path1a.lex|}
\ptindent{3: |ts_path2.lex|}
\ptindent{4: |ts_path3.lex|}
\ptindent{5: |ts_path4.lex|}
\ptindent{6: |ts_path5.lex|}
\ptindent{7: |ts_path6.lex|}
\ptindent{8: |ts_path7.lex|}
They are exercised by |qa_alltests.sh| script command file.
@*2 Recursion and derives a string.\fbreak
Derives:\fbreak
Greek symbols are used to denote a mixed string of symbol(s) drawn from
the nonterminal, terminal vocabularies. The empty string is
represented by $\epsilon$. Derives is a relationship that starts with a
string of symbols and substitutes equivalent strings of symbols until
eventually no-more substitutions can take place. The resulting string
is either empty or a string of terminals. The order of substitution will
be from left to right. The $\longrightarrow$ symbol represents the
derives relation. Sometimes i will subscript the greek symbol
with $\epsilon$ denoting that it derives the empty string.
The $\not\longrightarrow$\ symbol represents ``no derives'' takes place.
I use this where a specific subrule pattern is being
discussed and the greek symbol is not present.

\fbreak
Left recursion: A \subrule \ A $\beta${} \fbreak
As shown, the rule {\bf A} calls itself from the left part of
its subrule. $\beta${} represents strings of Rules
and Terminals\fbreak
\fbreak
Right recursion: A \subrule \  $\alpha${} \  A \fbreak
The rule {\bf A} calls itself from the end of the subrule as shown.
$\alpha${} represents strings of Rules and Terminals.
\fbreak
\fbreak
@/
```

The above contents contains *cweave* directives and TEX macros like the following:

```
@** |epsilon_rules| grammar.\fbreak
```

Please acquaint yourself with *cweave*'s directives like sectioning, table of content entries, etc. The \fbreak[1] is a macro written by me to force a line break that contains the TEX command. The `@**` is a cweave directive having a `|...|` emphasis command.

Here are the insertion points within a grammar open for *cweave* directives, your jabbertalk, and graphic diagrams.

1. Before the **fsm** keyword. This acts as the introduction to the document.
2. Before the **rules** keyword. This acts as a break in the document to introduce the rule definitions. Comments might be on the overall rules parsing strategy, basic description to algorithms being developed or used within the grammar, grammar to grammar interactions that are out of the ordinary, symbol table activities (scoping activity as in the Pascal language driven by grammar points), logic point boundaries. [2]

[1] I know that TEX has such a macro but back in time i wanted to learn to write macros and also a nemonic of my patios.
[2] Remember the "Alzheimer's coding syndrome? Your coding thoughts deteriorate over time.

3. Before a rule definition. This acts as a break in the document for the specific rule. Possibly to introduce some subtleties used by the specific rule definition to follow. It is more of a general introduction to what is being achieved by the rule.

4. Just after the rule name. This is an introduction to the specific rule.

5. Before any O_2 directive.

6. Inside the C++ syntax directed code declaration and implementation directives. This allows one to break up large syntax code into manageable and commented and formatted code sections. The code directive can include any number of typesetting comments: `/@...@/`.

For terminals and Errors, the following outlines its comment insertion points.

1. Before their **terminals** or **error-symbols** construct. This acts as the introduction to the document.

2. Before a T or error **(sys-class** definition. That is the insert point comes after the defining T literal and optional AD AB attributes. This is an introduction to the specific T.

3. Before any T directive like **user-declaration**, **user-implementation**, **constructor**, **op**, or **destructor**.

12.5 A grammar's outlook: Pictures and Annotations

The annotated pictorials is the grammar's "literate programming" document. This is the Yang of the grammar's raw text file to typeset its document. A ".lex" file uses the same *cweave* directives as in a *Cweb*'s ".w" file. There are the canned directives generated by O_2 to shape the document along with your own directives used to include pictures, break up code boundaries, and add better cross referencing or notes for the Index. U are open to draw from TEX's macro programming facilities. If u are really into producing typeset content, please have a read of Donald E. Knuth's "The TeXbook" [15]. I'm not trying to push this on u only making u aware that these facilities are open to your use/inclusion in grammar documents. My leanings allow u the user to tailor your own tastes into a grammar document. U can reprogram the *mpost* code that draws to railroad diagrams, frontend your own directives to shape your documents, or register their manufacture into some backend repository. This is open season to grilling.

Diagram 1.2 shows the intermediate steps to generating the document for *mpost* diagrams included in a ".w" file for *cweave* consumption. This should be your programming source to study the grammar. This is an indirection as the document makes it readable while the ".lex" file still needs to be source edited. The frequency of generating the document is of your own choosing. U might spend most of your time in a text editor/gen the grammar/compile/run cycle. If the grammar is complex, or the mini algorithms inside it are subtle, I suggest u lean more to the generated document to eyeball the formatted source for debug assessment.

O_2 is a substitute to *Cweb*'s *ctangle* program that emits C++ code with the "#line" directives as reference points into the "literate programming" file. O_2 emits straight C++ code. Your debugging reference points are the grammar's entities: `fsm`, rule names, and local object methods and variables. As an aside, O_2 was written using the full *Cweb* programming paradigm: *cweave* for document generation and *ctangle* for C++ code generation. I post-processed *ctangle*'s C++ "#line" directives as comments as i found it much easier to debug against the C++ code instead of the document's source. Going back to the "literate programming" file is easy with these commented out directives in place: line number with its source file reference.

12.5.1 Emitted LR state network

It is a very detailed description in table form describing the makeup of each lr derived state. Each state lists its subrules contents, their string of symbols, and a vector list of where each item was state born, next vectored into state, and what state the subrule reduces. In states having reduced items, the lookahead set is identified. These sets and their contents are listed in the companion document. The neat thing about this table is its extensive use of typesetting facilities like underlining, and superscripting to enhance

understanding. For example the symbol vectoring into this state is identified, the state type expressed, symbol strings underlined that contribute to a rule's follow set. I put forth this thought: shouldn't compiler type outputs use typesetting/cross referencing facilities to enhance understanding rather than current flattened data reports that can be borderline useless? I choose expressiveness on-demand over the easy way out to limiting effort.

12.6 Making some sense out of these lr tables?

O_2 never emits bad code. Where did that comment come from and who is the author of this falsity? Questioning the outcome in bottom-up parsing was one of my concerns. I felt it only fair to the end user that tracks be left around allowing one to assess the outcome.

At least booboos or nobler thoughts to error correction can be studied and dealt with. When developing O_2, i needed to study my attempts at getting it right. Regression tests were done using a test-driver with lots of different grammars. This tested the error reporting traps by my grammars. There were times when the tables needed study in an uncompressed form. Here this document details the different items i felt required certifying that the emitted lr tables were correct. This crossed various attempts at C++ table content, the C++ structures being compiled into, to the dynamic optimizations while running.

Part of this documenting required C++ comments sprinkled throughout the emitted C++ tables. For example threads got a lexicographical call-id that got compressed into bit maps. What and how: do u know what the bit maps were composed of without the thread's name as comments? The same comments apply to the other grammar entities: Tes, rules, and the state numbers making up the tables. Unless u like to de-engineer bits which is my last choice i tried to make things as easy as possible to certify the emitted code. At least with the 2 sources to the tables: document and C++ tables, this bisected the assessment between noble intent and deeds: reality. It allowed O_2's weaknesses to be exposed more easily to me. Back to booboos, there ain't any now but i'm sure crow will be my serving.

Regarding dynamic patching to tables, this thought is still running through my head. For the moment i leave this to u to experiment with. I will gladly open up my ears to u.

12.6.1 Symbol registry

This is your google to the stars or to the symbolic strings of symbols. First the directives are crossed referenced against the grammar's entities `fsm` and rules. Rules are crossed referenced giving recursive use, definition, and position within subrules. Used Tes indicate what rule/subrule(s) and symbol position across the grammar.

12.6.2 Random comments

1. Rules first sets.
   They list their booty. Other calculations using "first sets" have each rule's booty. For example a thread's "first set", a state's "first set", follow sets where it is an operand in these other calculations.

2. Lr network.
   List of productions with their derived LR state list configurations.

3. List of reducing states.
   It describes all the reducing states and their types: shift/reduce, reduce/reduce, mixed type of previous 2. These reducing states have references to the lookahead sets calculated per reducing subrule.

4. Follow sets.
   Follow sets are strings of symbols used to calculate the reducing lookahead sets whose booty are Tes. Each state is listed with each rule's follow set giving its local booty: "first set" and inheritance vectors receiving other state's rules follow set contents. It allows one to manually recalculate whether the follow set graphs are correct against O_2's follow set calculation used per reducing subrule.

5. List of lookahead sets.
   This is a list of the follow sets referenced by the reducing subrules with their contents. As an optimization, it merges common sets together to save space. These lookahead sets are derived by traversing the reducing rules's follow set graphs. The 1st document describes the grammar and the lr state table which contains references to these common lookahead sets.

6. Index.
   As some of the expressed items like rules were too verbose in compressed form, there is a cross reference to these compressed rules against their literal names.

7. Table of contents.
   Your GPS to where those symbols are defined and used.

12.7 Where are Errors and Tes documents?

Their documents are not automatically generated by O_2. This was a implementation decision as these grammar items get rarely modified after the initial development flurry. Their table of contents are useful as a reference particularly when you want to get at your own C++ methods contained in them.

Their "Err.w" and "T.w" files are created when the "-err" or "-t" options are given to O_2. To generate the "pdf" document, u must run the following programs manually or create your own bash script. This generates the "Err.pdf" document.

```
cweave Err.w
pdftex Err.tex
```

Substitute "T" in place of "Err" in the above script to generate the T document. All terminal classes allow the full range of typesetting directives: drawings, index directives, macros, etc [3].

[3]The lrk meta T constants document is **/usr/local/yacco2/docs/Lrk.pdf**. It is constant and can not be generated. The why is my take on freezing its definition where at the beginning of O_2 development there was a -lrk parameter to do this. Not any more.

Chapter 13

Pieces of thought: odds and ends

This is post-clean-up commentary on my cerebral droppings. It is an eclectic collection on various loose ends threaded throughout the book left to jabber on and u the reader having this book's learnings to critique/appreciate:

1. Symbol table support in Yac_2o_2 library.
2. Destructor support in rules.
3. Rule recycling.
4. Syntax directed nondeterminism: to thread or not to?

13.1 Symbol table's Yac_2o_2 companion

Why should there be a symbol table portal by Yac_2o_2?
T morphing and counting:
What does this mean?

If a T has been created under some context like lexing but takes on a different personality later in the parsing stages, this is T morphing. An example would help! Let's look at the function call. Defining the function as an externally referenced prototype, the function name is just an identifier. It is constructed from letters, numbers, and possibly some other characters like underscore or dollar sign. When the lexical stage transforms these character sequences into an identifier the context to which it belongs still has not been fully recognized. This T is placed into the token stream as an identifier and not as a function-id. Most compilers will post process the identifier within the function call statement to verify that the identifier is actually a function-id. Their grammar recognizing the function call statement is less sensitive to what it is being parsing: it is just an identifier. But an identifier could also be a variable name or a type name... U get the notion. Parsing at the semantic level, the token stream is insensitive to this function-id context. My sensitivities are to be more specific at the point within the grammar expressions by using the proper symbol.

How can u substitute the fetched identifier T as a function-id?
At container access time by the parser, T morphing on the fetched token against a symbol table lookup would resolve it.

You do not need this functionality if u are not dynamicly remorphing the fetched T. Your symbol table management is still independent of the Yac_2o_2 library. What/when/where/how it gets used removes any dependency on this dynamic library facility. So u can skip this section but for the curious here's the scoop. Let's review what this facility allows u to do, where/when/how the functor fires, and what u can do to trace its activities. The functor supported in Yac_2o_2 library is defined as:

```
template <typename T> class tble_lkup
  :public std::unary_function<T,T>{
  public:
```

```
      tble_lkup():lkup__(ON){};
      ~tble_lkup(){};
      virtual T operator()(T t)=0;
      void turn_off_lkup(){
       lkup__ = OFF;
     };
     void turn_on_lkup(){
       lkup__ = ON;
     };
     bool lkup(){return lkup__;};
     bool lkup__;
  };
  typedef tble_lkup<yacco2::CAbs_lr1_sym*> tble_lkup_type;
```

It is a template class functor that requires one to derive from when defining your own symbol table functor. The "operator" method is the functor part that gets fired. It gets called only when the functor's class "lkup__" variable is "on". "lkup__" allows u to control whether the functor should be servicing the fetched T access request. Co-ordinating the on/off of the symbol table facility can be done by a grammar's syntax directed code. The Pascal translator uses epsilonal rules strictly written to co-ordinate the turning off and on its lookup capability within the statement context being recognized. U might question why? Well the Pascal language has different symbol scoping contexts: with statement, nested functions, formal function passed parameters, global symbol context. This is a partial list of its symbol scopes. Each context being recognized depends on the degree of symbol table scoping lookup to be used. Use of this table symbol facility has served the Pascal translator well. It co-ordinated its own internal symbol table management with O_2's library symbol table functor.

The grammar "start points" like the **fsm**'s *op* directive is a good place to turn on/off the dynamic morphing capability by changing the "lkup__" value. It is not a hard and fast rule that u do it within the grammars's logic points. My sensitivities are to be more specific at the point within the grammar expressions rather than a post evaluation process. O_2 allows one to control this. If there are 2 competing grammar threads with different symbol contexts competing, u can control the symbol table management with the SYM_TBL_MU mutex.

Here are the 2 procedures to control your private use. As symbol table management is a critical region/service, u must, must use the SYM_TBL_MU mutex defined in the O_2 library to control and release its resource.

```
  LOCK_MUTEX(SYM_TBL_MU);
    ... your code
  UNLOCK_MUTEX(SYM_TBL_MU);
```

In the Pascal translator, the above 2 procedures are called by their epsilonal grammar rules and turned on/off scoping within it own symbol table management facility. Your grammar rules between these calls will deliver symbols according to your code dictates.

13.1.1 Rolling rolling rolling down the functor

The Yac$_2o_2$ library leaves this half built functor definition for u to derive from. It derives from STL's "unary_function" while adding the on/off table lookup control. Here's an example taken from the Pascal translator:

```
class pas_tble_lkup : public tble_lkup<CAbs_lr1_sym*>{
 public:
  pas_tble_lkup();
  ~pas_tble_lkup();
  CAbs_lr1_sym* operator()(CAbs_lr1_sym* T);// symbol table functor
  CAbs_lr1_sym* fnd_tid_in_sym_tbl(const char* Tid);
  CAbs_lr1_sym* try_remapping_t(CAbs_lr1_sym* T
                               ,T_sym_tbl_report_card& Report);
  void add_t_to_sym_tbl(KCHARP Tid,CAbs_lr1_sym* T);
```

```
   void try_remapping_t_with_all_scopes(Parser* Parser
                                        ,CAbs_lr1_sym* Sym);
  private:
   bool is_t_an_id(CAbs_lr1_sym* T);
 };
```

Notice it derives from the template class "**tble_lkup < CAbs_lr1_sym* >** " whose contents are the abstract symbol pointer: "CAbs_lr1_sym*". This is the base symbol from which all grammar symbols are derived from: Rules, and Tes which are symbol table applicable. U can add any number of methods/variables to your functor class but i repeat u must define the functor's operator as its forced definition comes from the base template C++'s **virtual** keyword attached to the "operator" method:

```
  virtual T operator()(T t)=0;
```

Without defining it will incur the wrath of the C++ compiler.

13.1.2 Pascal's symbol table functor implementation

Here is the Pascal's functor's definition. I left various remnants of past tracings inside it as C++ comments to show to u gentle reader what i did to certify my efforts. The remapping of the types come about by the appropriate Pascal definitions like procedure and function names, compound fields, type and variable definitions. Use of T morphing made the Pascal grammars more sensitive to parsing statements and to their readability:

```
CAbs_lr1_sym*
pas_tble_lkup::
operator()(CAbs_lr1_sym* T){
#ifdef _DEBUG_PRE_COND
  if(T == 0){
      CHARP msg =
        "Symbol ptr is zero";
      Lr1_Error_faulty_precondition(msg,__FILE__,__LINE__);
  }
#endif
   ///lrclog << "LOOKUP TOKEN: " << T->id() << endl;
  if(lkup() == false){
   ///lrclog << "LOOKUP TURNED OFF " << endl;
    return T;
  }
  if (is_t_an_id(T) == false){
   ///lrclog << "LOOKUP TOKEN NOT ID " << endl;
    return T;
  }
  T_sym_tbl_report_card report_card;
  CAbs_lr1_sym* sym = try_remapping_t(T,report_card);
  if (sym == 0){
  ///lrclog << "LOOKUP TOKEN NOT IN TABLE " << endl;
    return T;
  }

  T_identifier* id = (T_identifier*)(T);
  // clone the returned type, but how?
  //sym->rc(T);// add its physical co-ordinates
  switch (sym->enumerated_id()){
    case T_Enum::T_T_const_id_:{
      T_const_id* st_entry = (T_const_id*)(sym);
      AST* cd_reft = crt_cd_ref_ast(st_entry,id);
      AST* cidt = AST::get_1st_son(*cd_reft);
      T_const_id* nsym = (T_const_id*)AST::content(*cidt);
      nsym->set_rc(*T,__FILE__,__LINE__);
```

```
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_type_id_:{
    T_type_id* st_entry = (T_type_id*)(sym);
    AST* td_reft = crt_td_ref_ast(st_entry,id);
    AST* tidt = AST::get_1st_son(*td_reft);
    T_type_id* nsym = (T_type_id*)AST::content(*tidt);
    nsym->set_rc(*T,__FILE__,__LINE__);
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_var_id_:{
    T_var_id* st_entry = (T_var_id*)(sym);
    AST* vd_reft = crt_vd_ref_ast(st_entry,id);
    AST* vidt = AST::get_1st_son(*vd_reft);
    T_var_id* nsym = (T_var_id*)AST::content(*vidt);
    nsym->set_rc(*T,__FILE__,__LINE__);
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_proc_id_:{
    T_proc_id* st_entry = (T_proc_id*)(sym);
    AST* pd_reft = crt_pd_ref_ast(st_entry,id);
    AST* pidt = AST::get_1st_son(*pd_reft);
    T_proc_id* nsym = (T_proc_id*)AST::content(*pidt);
    nsym->set_rc(*T,__FILE__,__LINE__);
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_func_id_:{
    T_func_id* st_entry = (T_func_id*)(sym);
    AST* fud_reft = crt_fud_ref_ast(st_entry,id);
    AST* fidt = AST::get_1st_son(*fud_reft);
    T_func_id* nsym = (T_func_id*)AST::content(*fidt);
    nsym->set_rc(*T,__FILE__,__LINE__);
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_fld_id_:{
    T_fld_id* st_entry = (T_fld_id*)(sym);
    AST* fd_reft = crt_fd_ref_ast(st_entry,id);
    AST* fidt = AST::get_1st_son(*fd_reft);
    T_fld_id* nsym = (T_fld_id*)AST::content(*fidt);
    nsym->set_rc(*T,__FILE__,__LINE__);
    nsym->ref_no = report_card.i_->second.ref_no_;
    return nsym;
  }
  case T_Enum::T_T_proxy_type_:{
    T_proxy_type* st_entry = (T_proxy_type*)(sym);
    AST* npt = new AST();
    T_proxy_type* np = new T_proxy_type(st_entry->identifier(),npt);
    np->set_rc(*T);
    return np;
  }
  default: {
    return T;
  }
}
```

```
  return T;
}
```

So let us head scratch a bit on the above code.

Where is the actual symbol table and its activities: I don't see them? How does the functor get called by the Parser?

One of the parameters to a Parser is the symbol table functor. Please see section 2.5.7 describing the parameters to **Parser**. The 7th parameter is the symbol table functor. When present, **Parser**'s "get_next_token" and "get_spec_token" methods will call the functor. If the functor facility has been turned off, it will be bypassed by the token fetch routines. Appropriate tracing details are dormant within the Yac_2o_2 library and activated when the "yacco2::YACCO2_T__" is turned on. It will give a before T trace and possibly its T after-value if the T was remapped.

13.1.3 Relating to Pascal's symbol table

Well this is nice but how does all of the above relate to symbol table handling? Part of the methods accompanying the functor have access to the globally defined symbol table facility. Here is an example from the above functor class:

```
CAbs_lr1_sym*
pas_tble_lkup::
fnd_tid_in_sym_tbl(KCHARP Tid){
  ///lrclog << "========>TO fnd_tid_in_sym_tbl key: " << Tid << endl;
  string Uid_name(Tid);
  transform(Uid_name.begin(),Uid_name.end(),Uid_name.begin(),tolower);

  T_sym_tbl_report_card report_card;
  LOCK_MUTEX(yacco2::TH_TBL_MU); // dont allow other grsmmsrs to launch
    pastbl_cpp::find_sym_using_scopes_stk(&report_card,Uid_name.c_str());
  UNLOCK_MUTEX(yacco2::TH_TBL_MU);
  if(report_card.status_==T_sym_tbl_report_card::failure){
  return 0;
  }
  if(report_card.action_==T_sym_tbl_report_card::not_fnd){
  return 0;
  }
  return report_card.sym_entry_;
}
```

I'll not get into my Pascal symbol table implementation details. What is important is mutex use for exclusion within this **symbol table functor**: "yacco2::TH_TBL_MU" controlled by the "LOCK_MUTEX" and its release "UNLOCK_MUTEX" methods. This prevents ugly things happening when threaded parallelism is happening. The TH_TBL_MU mutex prevents other parallel potental threads from running and do their own symbol table management. That is, threading non-determinism might take place and their T view being different. Hence the reason for this mutex use to globally controll thread launching when a specific grammar has asked and gotten controll on the symbol table. Within the grammars, the symbol table mutex SYM_TBL_MU will be held/released for its own symbol table activity. Next section demonstrates this.

13.1.4 Calling dhem sybol facalties

Calls to the symbol table global methods can be sprinkled throughout your grammars syntax directed code. The Pascal language has nested contexts as to where a symbol gets resolved. These contexts can be turned on or off at grammar points: example, with statement, the formal parameter names of function parameters, nested procedure definitions and their own constant, type, and variable definitions. An example from the Pascal translator:

```
/*
FILE:  pas_plist_decl.lex
Dates: 16 July 2001
       7 Nov. 2001 tested
       29 Aug. 2003 - change name to indicate declaration as opposed to
               parm list in call statement
       23 Jun 2004 - remove sym. table on/off. use proper sym. tble
       23 Oct. 2007 - added back table on/off re due to:
               id list could have a name as one of the type or constants!
               ie scope control
Test file:  pas_plist_decl.dat
Purpose:
  Parameter list declaration from Wirth modified for bottom-up parse
*/
fsm
(fsm-id "pas_plist_decl.lex",fsm-filename pas_plist_decl,fsm-namespace NS_pas_plist_decl
,fsm-class Cpas_parameter_list {
  user-prefix-declaration
using namespace std;
using namespace yacco2;
using namespace NS_pas_terminals;
#include "pas_var_decl.h"
#include "pas_proc_plst_decl.h"
#include "pas_func_plst_decl.h"
#include "externs.h"
***
  op
    LOCK_MUTEX(yacco2::SYM_TBL_MU);
      pastbl_cpp::use_only_top_nested_scope();
    UNLOCK_MUTEX(yacco2::SYM_TBL_MU);
  ***
}
,fsm-version "1.0",fsm-date "9 May 2001",fsm-debug "false"
,fsm-comments "parm list for program, and func./proc.")
parallel-parser
(
  parallel-thread-function
    TH_pas_plist_decl
  ***
  parallel-la-boundary
    eolr
  ***
)
@"/pascalxlator/pas_include_files.T"

rules{
Rpas_plist_decl (){
  -> Rexp  {
    op // return parm-list structure
      AST* p1 = sf->p1__->type_;
      sf->p1__->type_ = 0;
      RSVP(AST::content(*p1));
    ***
    }
}

Rexp (
lhs {
  user-declaration
```

```
    public:
    AST* type_;
  ***
  }
){
  -> "(" Rpms ")"  {
    op
      AST* p1t = new AST(*sf->p1__);
      AST* p2 = sf->p2__->type_;
      AST* p2c = sf->p2__->eoc_;
      sf->p2__->type_ = 0;
      AST* p3t = new AST(*sf->p3__);

      type_ = new AST;
      T_parm_list_decl* pl = new T_parm_list_decl(type_);
      pl->set_rc(*sf->p1__,__FILE__,__LINE__);
      AST::set_content(*type_,*pl);
      AST::crt_tree_of_2sons(*type_,*p1t,*p2);
      AST::join_sts(*p2c,*p3t);
    ***
    }
}

Rpms (
lhs {
  user-declaration
    public:
    AST* type_;
    AST* eoc_;
  ***
  }
){
  -> Rpms Rneeded_semi_colon Rpm {
    op
      AST* p1 = sf->p1__->type_;
      AST* p1e = sf->p1__->eoc_;
      AST* p2t = sf->p2__->type_;
      AST* type_t = AST::add_child_at_end(*p1e,*p2t);
      AST* p3 = sf->p3__->type_;
      sf->p1__->type_ = 0;
      sf->p3__->type_ = 0;

      type_ = p1;
      AST::join_sts(*p1e,*p3);
      eoc_ = p3;
    ***
    }
  -> Rpm {
    op
      AST* p1 = sf->p1__->type_;
      sf->p1__->type_ = 0;
      type_ = p1;
      eoc_ = p1;
    ***
    }
}

Rneeded_semi_colon  (
lhs {
```

```
  user-declaration
    public:
    AST* type_;
  ***
  }
){
  -> ";"   {
    op
      type_ = new AST(*sf->p1__);
    ***
    }
  -> |?|   {
    op
      CAbs_lr1_sym* sym = new Err_missing_semi_colon;
      sym->set_rc(*parser()->current_token(),__FILE__,__LINE__);
      RSVP(sym);
      parser()->set_stop_parse(true);
    ***
    }
}

Rpm (
lhs {
  user-declaration
    public:
    AST* type_;
    AST* eoc_;
  ***
  }
){
  -> ||| "var-decl" NS_pas_var_decl::TH_pas_var_decl {
    op
      type_ = sf->p2__->ast();
      eoc_ = sf->p2__->eoc();
      LOCK_MUTEX(yacco2::SYM_TBL_MU);
        pastbl_cpp::use_only_top_nested_scope();
      UNLOCK_MUTEX(yacco2::SYM_TBL_MU);
    ***
    }
  -> ||| "proc-decl" NS_pas_proc_plst_decl::TH_pas_proc_plst_decl {
    op
      type_ = sf->p2__->ast();
      eoc_ = type_;
      LOCK_MUTEX(yacco2::SYM_TBL_MU);
        pastbl_cpp::use_only_top_nested_scope();
      UNLOCK_MUTEX(yacco2::SYM_TBL_MU);
    ***
    }
  -> ||| "func-decl" NS_pas_func_plst_decl::TH_pas_func_plst_decl {
    op
      type_ = sf->p2__->ast();
      eoc_ = type_;
      LOCK_MUTEX(yacco2::SYM_TBL_MU);
        pastbl_cpp::use_only_top_nested_scope();
      UNLOCK_MUTEX(yacco2::SYM_TBL_MU);
    ***
    }
  ->   ||| |?| NULL {
    op
```

```
      RSVP(sf->p2__);
      parser()->set_stop_parse(true);
    ***
    }
}
}// end of rules
```

Sprinkled above you see "**pastbl_cpp::**" prefix which is the namespace given to the Pascal symbol table. The "use_only_top_nested_scope" procedure call is just one of its procedures controlling the "scoping stack" for the symbol table search space. This grammar example shows the subrule and **fsm** *op* directive code calls to the symbol table interface.

13.2 Destructor tear downs

A question:
What use does the **destructor** directive do in a rule definition when Dave u've been raging about recycling/efficiencies and the rule never gets deleted back to heap heaven?

Well u have a point dear reader. Nada as it never gets destroyed until the end as in the Doors song. But Tes could have some use for it. It is not very common in its use but it at least satisfies my own coding template logics: before, during, and aftermath. This is the last chance before the entity goes to recycled memory purgatory?

An example from Pascal's T class both in grammar and emitted C++ form.

```
"basic-char"  // "basic source character set"
(sym-class  T_basic_char{
  user-declaration
    public:
    T_basic_char(yacco2::CAbs_lr1_sym* Basic_char);
    yacco2::CAbs_lr1_sym* basic_char()const;
    void zero_out_basic_char();
    private:yacco2::CAbs_lr1_sym* basic_char_;
  ***
  destructor
    delete R->basic_char();
  ***
  user-implementation
    T_basic_char::
    T_basic_char(yacco2::CAbs_lr1_sym* Basic_char)
      T_CTOR("basic-char",T_Enum::T_T_basic_char_,&dtor_T_basic_char,false,false)
     {basic_char_ = Basic_char;}
    void T_basic_char::zero_out_basic_char(){basic_char_ = 0;}
    yacco2::CAbs_lr1_sym* T_basic_char::basic_char()const{return basic_char_;}
  ***
  }
)

/**
 * c++ form
 **/
T_basic_char::
  T_basic_char(yacco2::CAbs_lr1_sym* Basic_char)
  T_CTOR("basic-char",T_Enum::T_T_basic_char_,&dtor_T_basic_char,false,false)
     {basic_char_ = Basic_char;}
  void T_basic_char::zero_out_basic_char(){basic_char_ = 0;}
  yacco2::CAbs_lr1_sym* T_basic_char::basic_char()const{return basic_char_;}
  void T_basic_char::dtor_T_basic_char(yacco2::VOIDP This,yacco2::VOIDP P){
```

```
    T_basic_char* R = (T_basic_char*)(This);
    delete R->basic_char();
  }
```

The 2 forms are given together to see the emitted parameters added to the destructor method and to see extra code added to aid the grammar writer. The **destructor** directive constructs its destructor method by prefixing the name with "dtor_" and suffixing the balance of the name with its C++ class name. The 2 parameters give u the context where its comes from and to reestablish by casting to it its own entity. The internal **R** variable is the casting back into itself. This allows the destructor code to get at the internal variable "basic_char_" for its recycling. Also the **Parser** context is given by the **P** parameter. Though not referenced in the example, it allows the grammar writer other contexts to possibly deal with. Again your perplexed non-botoxed brow queries the reason for this writing. Some possibilities to its existence though a conjuring i go: Notifying or logging to a file, coding hygiene, awareness of a marginally useful/useless facility. Okay white caned u are Dvae.

13.3 Rule recycling: What your mother/method never told u

This is a set of watch-outs to that evil can-evil Optimization. As rules are created once through the **new** verb, the template run model of create, initialize, flush out is never completed. The circular pattern is dead-ended by never completing the cycle. So things to watch for are as follows:

1. When a rule is created and re-attached upon a recycled rule being re-activated, how do i ensure that its local contents are always initialized properly? Use its **op** directive. This duplicates the initial creation initialization of a rule once-only. Thereafter it acts like a new born rule. Warnings, make sure pointers are niled out or u could be in for a surprise. Make sure any STL types are emptied. For example strings and their contents.

2. As it never dies, how can it clean-up from its popping off the stack? These are kludged comments:

   (a) The subrule being reduced has its symbols on the stack. The reduced-to rule can do clean up duties by calling the stacked symbols own clean up methods. This is not too nice as it now has a parental interrelationship to software toilet training. Now the burden is on the parent rule to deal with its stacked children. If there is not an urgent need to the cleanup, then leave it to the rule re-activation to clean up. This might not be acceptable as the recycled rule is dormant and will never get cleaned up after its last fling in Paris. So there are remnants of droppings still lingering. As an aside, once upon a time "stack" symbol pushing/popping did have their own special set of grammar directives. They were deemed useless and so recycled into my own mindless waste-bin.

   (b) Alas dear reader, u are warned and u are required to come up with your own creative ways to deal with this situation. Your thoughts might be to re-activate the new/dispose of rules but a 25% performance hit is too expensive. To add another directive to the rule definition just for occasional convenience does not merit it.

My experience with other systems written with O_2 indicate that this should not be too much of an inconvenience to the grammar writer.

13.4 Syntax directed nondeterminism: to thread or not to?

Ahh the title is a fancy expression on having 2 or more threads soliloquizing at the same time. Let's review what is happening, how to get it going, and what to do with the results. Somewhere a grammar has entered a subrule's syntax directed code or u just want to fire up some grammar logic snippet. Grammars normally start off by a monolithic grammar. It then deals with threads somewhere within its symbol sequences: aka subrules. Outside of this, u can manually call a grammar thread as described in the Threads chapter 7. 2 choices u have to run thread snippets: create a monolithic grammar that calls dthem threads or call a thread manually that calls them threads. Snippets of independent symbol logic sequencers run manually only allow u to launch 1 thread. So how do u manually launch more then 2 threads at a time. The direct

route is a cul-du-sac: u can't; It's one thread only. But u can call manually a threaded grammar whose start rule's "first set" contains threaded grammar expressions to be launched. Its a bit indirect but there u have it: a threaded grammar that acts as a thread dispatcher. Just remember that the 2 competing threads being launched must have equivalent "first set" terminals or u will be questioning the outcome with directed expletives at O_2 or more so towards the author.

Now syntax directed code within these launched threads can also launch other threaded grammars using the Parser to do it or manually within syntax directed code: and away u go [1]. With all this double dribbling, remember to arbitrate on the outcomes when multiple returned Tes are passed back to its launching grammar that could be nested within other grammars. This is a recursive statement as snippets of symbol sequences are themselves launching snippets of sequences.

[1] Is this a Gleasonism? or your sentiments on my comments?

Chapter 14

Con·clu·sion: hal·le·lu·jah

My pipes are huffing and a puffing and your ears a'ringing. The chorus taken with encores and the white cane allowing me off the stage. A thank you dear reader for your patience and putting up with my wanderings on Yac$_2o_2$. Hopefully your readings have gained you the tap to dance the O_2's tune.

Now a close down on "how to" sceanarios working with O_2 in an UNIX environment. These are quick *bash* vignettes to stop the mutterings towards the author and page flipping in the age of Web searching without the advertisements. So on with the theater Dave as my teeth mashings are now down to the gums.

14.1 A grammar's C++ clone: be it evil or not

Let's see if a specific grammar is kosher. I will use Yac$_2o_2$'s file structure to illustrate. Of course u would substitute your own development folders to compile your grammar. I will fully qualify the path to execute Yac$_2o_2$'s programs. Here is the book reference on setting up *bash*'s **PATH** variable: section 1.6.1 page 6

```
cd /usr/local/yacco2/compiler/grammars
/usr/local/yacco2/bin/o2 eol.lex
```

/usr/local/yacco2/compiler/grammar/o2grammars.sh *bash* script will generate all your grammars at one time for C++ compiler consumption given the appropriate parameter(s). U can clone this and modify it to your own whimsies within your development folder.

```
cd /usr/local/yacco2/compiler/grammars
# to gen C++ code per grammar
.  o2grammars.sh # no parms just gen grammars found
.  o2grammars.sh -t -err # gen grammars and Tes, Errors
```

The last script above includes the "-t" and "-err" parameters. U add either one of these parameters whenever the grammar's Terminal vocabulary has been changed by an terminal's addition, modification, or removal from either Terminal classes Meta-terminals or Errors. Not doing so leads to the emitted C++'s vocabulary files being not up-todate! Of course u remembered but for my wanning memory, the Terminal's enumeration scheme and rules building upon it will be different when plus or minusing terminals: each grammar's emitted lr(1) tables are now whackey or the C++es code for these modified terminals need to be regened.

14.2 A grammar's Mr. Dressup

To come out.

```
cd /usr/local/yacco2/compiler/grammars
# to gen pdf, ps per grammar
.  o2grammars.sh -p # printed cosmetics of your grammars
```

The generated "pdf" and "ps" files are copied back to your grammars folder.

```
cd /tmp/o2
# move them to your own docs folder
cp *pdf /usr/local/yacco2/docs/ # example docs folder of yacco2
cp *ps /usr/local/yacco2/docs/ # example docs folder of yacco2
```

U can then move them wherever u want or just lazer them for your eyes. Here are the script commands to generate an individual grammar the hard way by manually entering each command. This example gens the **eol.lex** grammar's documents:

```
cd /usr/local/yacco2/compiler/grammars
# to gen a specific grammar's pdf,ps
/usr/local/yacco2/bin/o2 -p eol.lex
mpost eol.mp
cweave eol.w
pdftex eol.tex
pdftops eol.pdf
cweave eol_idx.w
pdftex eol_idx.tex
pdftops eol_idx.pdf
```

These above commands are run within **/usr/local/yacco2/compiler/grammars**. This means that any ancillary generated files by the programs like *mpost, cweave,* O_2 need to be manually removed — ugh! So I wrote another script for u that generates an individual grammar. An options menu is outputted which u select. It runs the approriate programs within the "/tmp/o2" directory thus not requiring clean up duty. The gened files are moved back to your starting folder: for O_2 **/usr/local/yacco2/compiler/grammars**. Here's the newly built script **gen1grammar.sh**:

```
cd /usr/local/yacco2/compiler/grammars
.  gen1grammar.sh eol.lex # emit your c++ jewel
```

14.3 Cross talk amongst grammars

So let's get your grammars talking amongst themselves. U remembered didn't u? This is O_2linker's claim-to-fame. It takes each grammar's "fsc" file when u C++ed each grammar. Then O_2linker outputs its own grammars's assessment for the native linker to piece together those grammar threads. U wrote your O_2linker's input file referencing each grammar's "fsc" file. Let's get it on Dave! Here is O_2's own grammar registery: **yacco2.fsc**.

```
cd /usr/local/yacco2/compiler/grammars
# gening fsces object file that is glue for the native operating system linker
/usr/local/yacco2/bin/o2linker yacco2.fsc
```

Have a look in **yacco2.fsc** contents or **/usr/local/yacco2/grammars-testsuite/testout.fsc**. There is no magic. Just itemizing the grammars u composed with a preamble of header files defining the Terminal vocabulary and the grammars' symbols enumeration. So what can go wrong with this staging. U might have forgotten to include a grammar's "fsc" file in your handcrafted file for O_2linker's consumption.

Now u are ready to link together your main compiler program with all those grammar object files.

14.4 Linking your compiler into an executable

Are there any special requirements to do this? Not at all. Just that u must include the "**libc++**" library allowing its magic, the **pthread** library for your own grammars to strut, and of course Yac$_2o_2$'s own library **libyacco2** to manage the parsing environment, but most important don't forget to include your own grammars library. The **makefile_APPLE** *bash* script in folder **/usr/local/yacco2/compiler/o2/** shows how it creates the O_2 executable. Take notice on the C++ and linker commands to assemble O_2 to guide u for your own compiler. In the **ld** command, there are references to Yac$_2o_2$'s libraries — yacco2, o2grammars, and libraries C++ and pthread. Ditto comments apply to O_2linker's build script found in folder **/usr/local/yacco2/o2linker**.

14.5 Self study: Some readings to be done

Some gentle guidelines. Reading the grammar pdfes will give u a sense of how "literate programming" can be used to express itself against a grammar's logic/language. What other contexts can this be applied to? Your own creativity awaits. O_2's program document: **o2.pdf** is the road map of O_2's parsing stages. U'll see within the various parsing techniques employed: bottom-up and top-down — what the fudge? Yes threads allows one to parse ribbons of different terminal types in parallel at different parsing contexts: lexing, symantic, outputting, or gosh darn errors. Each parsing stage uses grammars and their syntactic logic applied towards some output objective: self applying self against self.

Now the Ents: trees beautiful trees!
Enough waxing Dave. Give us a bloody reading list with comments to study. So, sir we can lobotomize your diatribes!

Okay here it is:
The generated O_2linker's document: **o2linker_doc.pdf** provides a table of contents of each grammar with their summaries. A road map to grammar confusion? Not i says the sage of ingredients.

o2.pdf is the compiler/compiler documented program. Pay attention to the mainline code as it provides the various parsing stages and their grammars. Observe their token container types and whether they are input, output, or both to the parsers. As u near the end of the mainline code, the grammars used become semantic in their use: kalbassa rolls of different ingredients: to the stewing. The same comments apply to O_2linker program: **o2linker.pdf** with its own grammars to parse its linker language and emit the cross relationships for grammar conversations. As the linker's language is small, this might be a better study as the program is small but uses the same coding regimen with each parsing stage having its own specific grammars as logic automata.

14.5.1 A small reading list to explore:

My assumed order of import could be completely off the mark where u are concerned. Your own learning maelstrom on threads and currents about O_2 is what it is all-about. Take the list with some salt as your own discoveries / learnings will beacon your own insights to its use.

- **wlibrary.pdf** — A **must** read / reference to use Yac$_2o_2$'s API. Aired "literate programming" example with its dirty linen.
- **o2linker_doc.pdf** O_2linker gened document: TOC of all grammars. Your own documents will guide others about your own creations.
- **o2.pdf** the compiler / compiler program and its companion **o2linker.pdf**. They expose my attempt at "literate programming". At least they are organic in their maintenance history.
- **pass3.pdf** grammar — lexical parsing stage: thread use with error capture. Does "literate programming" of grammars merit your approval?
- **xxx_phrase.pdf** how a specific parse phase grammar can be accomplished. There are many ways to skin a language: some more ad hoc than structured.
- **rules_use_cnt.pdf** grammar used like a called subroutine with tree container use. A neat way to clothe an algorithm.
- **epsilon_rules.pdf** grammar determining epsilonal rules within the parsed grammar: logic automaton. Another take of a wolf in lamb's cloth?
- **la_expr.pdf** grammar used to analyse the thread's lookahead expression. A post evaluation on already parsed expression.
- **yacco2_stbl.pdf** Yac$_2o_2$'s symbol table code. Literate programming example on take one: symbol table management.

- **o2_externs.pdf** some of Yac$_2o_2$'s external routines that use and are called by grammars.
- **o2_err_hdlr.pdf** error grammar processing O_2's grammars faulty discoveries.
- **testout.pdf** a simple compiler/translator that uses Yac$_2o_2$'s library for your study.

Included in the distribution are all the grammars gened documents illustrating "literate programming". The shadows coming out with pride. Have a read as they allow u now the compiler writer to acquaint yourself with O_2's logic paradign: be it good, bad, or just plain ugly. They are there for your curiosity or barbs.

Happy compiling
Amen...

Appendix A: an unabridged grammar example

Here is a threaded grammar text file before it receives typesetting surgery. It is part of O_2's threaded grammars menagerie with its typesetting comments left in. The numbers starting each line act for tongue wagging. It recognizes the end-of-line characters and their combinations dependent on the Operating system run under. Lines 8, 13, 20 are a grammar's keyword structures for fsm, thread, and rules. Syntax directed code are contained in the *op* directives like line 24 to 31 inclusive. Typesetting directives examples are lines 4–7 introducing the document, 19 introducing the rules section, while 35 introduces the specific rule along with lines 37–40. One of its subrules is typeset commented by line 44.

```
 1 /* FILE:    eol.lex  Date: 17 Juin 2003
 2    Purpose: end-of-line recognizer
 3 */
 4 /@
 5   @** Eol Thread.\fbreak
 6   Fame: end-of-line matching for Unix, Mac, or Windows.
 7 @/
 8 fsm (fsm-id "eol.lex",fsm-filename eol
 9     ,fsm-namespace NS_eol,fsm-class Ceol,fsm-version "1.0"
10     ,fsm-date "6/2003",fsm-debug "false"
11     ,fsm-comments "end-of-line matcher."
12     )
13 parallel-parser (
14                   parallel-thread-function TH_eol ***
15                   parallel-la-boundary eolr ***
16                 )
17 @"/usr/local/yacco2/compiler/grammars/yacco2_T_includes.T"
18
19 /@ @** Rules.\fbreak @/
20 rules{
21     Reol (){
22       -> Rdelimiters {
23           /@ Return the |eol| token back to the caller. @/
24         op
25          CAbs_lr1_sym* sym = NS_o2_terms::PTR_eol__;
26          sym->set_rc(*parser()->start_token__
27          ,rule_info__.parser__,__FILE__,__LINE__);
28          sym->set_line_no_and_pos_in_line
29          (*parser()->start_token__);
30          RSVP(sym);
31        ***
32        }
33     }
34
35     /@ @** |Rdelimiters| rule.\fbreak @/
36     Rdelimiters
37       /@Eol Variants. \invisibleshift escapes from shift/reduce
```

```
38        conflict caused by x0d common prefix. As it's not in the token
39        stream, it deals well with this conflict type.
40       @/
41       ()
42       {
43         ->
44            /@ lf: Unix\fbreak@/
45            "x0a"
46         ->
47       /@ cr: Mac.
48         Note \invisibleshift removes shift/reduce conflict
49          caused by the lookahead boundary of |eolr|.\fbreak
50       @/
51            "x0d" |.|
52         -> /@ cr:lf Windows\fbreak@/ "x0d" "x0a"
53       }
54 }// end of rules
```

Appendix B — O_2's API

Here is O_2's header file: is this the French connection to your parsing?

```
#ifndef yacco2_
#define yacco2_ 1
#include "yacco2_compile_symbols.h"
#define START_OF_LRK_ENUMERATE 0
#define END_OF_LRK_ENUMERATE 7
#define START_OF_RC_ENUMERATE END_OF_LRK_ENUMERATE+1
#define END_OF_RC_ENUMERATE START_OF_RC_ENUMERATE+256-1
#define START_OF_ERROR_ENUMERATE END_OF_RC_ENUMERATE+1
#define SEQ_SRCH_VS_BIN_SRCH_LIMIT 71
#define MAX_UINT 1024*1024*1024*4-1
#define MAX_USINT 256*256-1
#define MAX_LR_STK_ITEMS 256
#define C_MAX_LR_STK_ITEMS MAX_LR_STK_ITEMS+1
#define BITS_PER_WORD 32
#define BITS_PER_WORD_REL_0 BITS_PER_WORD-1
#define MAX_NO_THDS 1024
#define START_OF_RC_ENUM 8
#define SIZE_CAbs_lr1_sym 56
#define NO_CAbs_lr1_sym_ENTRIES 1024*1024
#define SIZE_RC_MALLOC NO_CAbs_lr1_sym_ENTRIES*SIZE_CAbs_lr1_sym
#define ASCII_8_BIT 256
#define START_LINE_NO 1
#define START_CHAR_POS 0
#define LINE_FEED 10
#define EOF_CHAR_SUB 256
#define YES true
#define NO false
#define ON true
#define OFF false
#define BUFFER_SIZE 1024*4
#define BIG_BUFFER_32K 1024*32
#define SMALL_BUFFER_4K 1024*4
#define THREAD_WORKING 0
#define THREAD_WAITING_FOR_WORK 1
#define ALL_THREADS_BUSY 2
#define NO_THREAD_AT_ALL 3
#define THREAD_TO_EXIT 4
#define EVENT_RECEIVED 0
#define WAIT_FOR_EVENT 1
#define Token_start_pos 0
#define No_Token_start_pos Token_start_pos-1
#define CALLED_AS_THREAD true
#define CALLED_AS_PROC false
#define ACCEPT_FILTER true
#define BYPASS_FILTER false
```

```
#define FORCE_STK_TRACE 0
#define COND_STK_TRACE 1
#define Accept_parallel_parse 1
#define Shutdown 2
#define LR1_Questionable_operator 0
#define LR1_Eog 1
#define LR1_Eolr 2
#define LR1_Parallel_operator 3
#define LR1_Reduce_operator 4
#define LR1_Invisible_shift_operator 5
#define LR1_All_shift_operator 6
#define LR1_FSET_transience_operator 7
#define LR1_Procedure_call_operator 7 \

#define SET_ELEM_NO_BITS 8
#define Remap_token(Token)  \
if(sym_lookup_functor__==0) return Token; \
if(sym_lookup_functor__->lkup__==OFF) { \
return Token; \
} \
CAbs_lr1_sym*x= sym_lookup_functor__->operator() (Token) ; \
if(x==0) return Token;
#define Remap_set_result_and_return(Token)  \
Token= x; \
return Token;
#define Remap_return_result return x;
#define T_CTOR(A,B,C,D,E)  \
:CAbs_lr1_sym(A,C,B,D,E)
#define T_CTOR_RW(A,B,C,D,E,F,G)  \
:CAbs_lr1_sym(A,C,B,D,E,F,G)
#define YACCO2_define_trace_variables()  \
int yacco2::YACCO2_T__(OFF) ; \
int yacco2::YACCO2_TLEX__(OFF) ; \
int yacco2::YACCO2_MSG__(OFF) ; \
int yacco2::YACCO2_TH__(OFF) ; \
int yacco2::YACCO2_AR__(OFF) ; \
int yacco2::YACCO2_THP__(OFF) ; \
int yacco2::YACCO2_MU_TRACING__(OFF) ; \
int yacco2::YACCO2_MU_TH_TBL__(OFF) ; \
int yacco2::YACCO2_MU_GRAMMAR__(OFF) ; \

#define ADD_TOKEN_TO_RECYCLE_BIN(Token)  \
rule_info__.parser__->add_token_to_recycle_bin(Token)
#define DELETE_T_SYM(T)  \
if(T!=0) { \
if(T->enumerated_id__> END_OF_RC_ENUMERATE) { \
if(T->dtor__!=0) { \
(*T->dtor__) (T,0) ; \
} \
delete T; \
} \
}
#define RSVP(Token)  \
rule_info__.parser__->pp_rsvp__.fill_it(*rule_info__.parser__ \
,*Token,Token->tok_co_ords__.rc_pos__ \
,*rule_info__.parser__->current_token__ \
,rule_info__.parser__->current_token_pos__)  \

#define RSVP_WLA(Token,LATOK,LAPOS)  \
```

```
rule_info__.parser__->pp_rsvp__.fill_it(*rule_info__.parser__ \
,*Token,Token->tok_co_ords__.rc_pos__ \
,*LATOK \
,LAPOS)  \

#define RSVP_FSM(Token)  \
parser__->pp_rsvp__.fill_it(*parser__ \
,*Token,Token->tok_co_ords__.rc_pos__ \
,*parser__->current_token__,parser__->current_token_pos__)  \

#define ADD_TOKEN_TO_PRODUCER_QUEUE(TOKEN)  \
rule_info__.parser__->add_token_to_producer(TOKEN)
#define ADD_TOKEN_TO_ERROR_QUEUE(TOKEN) rule_info__.parser__->add_token_to_error_queue(TOKEN)
#define ADD_TOKEN_TO_ERROR_QUEUE_FSM(TOKEN) parser__->add_token_to_error_queue(TOKEN)
#include <stdlib.h>
#include <limits.h>
#include <assert.h>
#include "std_includes.h"
#include <time.h>
#if THREAD_LIBRARY_TO_USE__ == 1
#include <windows.h>
#include <process.h>
#elif THREAD_LIBRARY_TO_USE__ == 0
#include <pthread.h>
#endif

namespace yacco2{
typedef const char*KCHARP;
typedef unsigned char UCHAR;
typedef char CHAR;
typedef UCHAR*UCHARP;
typedef unsigned short int USINT;
typedef short int SINT;
typedef CHAR*CHARP;
typedef const void*KVOIDP;
typedef void*VOIDP;
typedef int INT;
typedef unsigned int UINT;
typedef unsigned int ULINT;
typedef void(*FN_DTOR)(VOIDP This,VOIDP Parser);
typedef UCHARP LA_set_type;
typedef LA_set_type LA_set_ptr;
struct CAbs_lr1_sym;
struct State;
struct Parser;
struct Shift_entry;
struct Shift_tbl;
struct Reduce_tbl;
struct State_s_thread_tbl;
struct Thread_entry;
struct T_array_having_thd_ids;
struct Set_entry;
struct Recycled_rule_struct;
struct Rule_s_reuse_entry;
typedef Shift_entry Shift_entry_array_type[1024*100];
typedef Set_entry Set_entry_array_type[1024*100];
typedef std::map<yacco2::USINT,yacco2::USINT> yacco2_set_type;
typedef yacco2_set_type::iterator yacco2_set_iter_type;
```

```
struct Cparse_record{
 void set_aborted(bool X);
 bool aborted()const;
 yacco2::CAbs_lr1_sym*symbol();
 void set_symbol(yacco2::CAbs_lr1_sym*Symbol);
 yacco2::State*state();
 void set_state(yacco2::State*State_no);
 void set_rule_s_reuse_entry(yacco2::Rule_s_reuse_entry*Rule_s_reuse);
 yacco2::Rule_s_reuse_entry*rule_s_reuse_entry();
 yacco2::CAbs_lr1_sym*symbol__;
 yacco2::State*state__;
 bool aborted__;
 yacco2::Rule_s_reuse_entry*rule_s_reuse_entry_ptr__;
};

struct lr_stk{
lr_stk();
 void lr_stk_init(yacco2::State&S1);
 void push_state(yacco2::State&S1);
 void push_symbol(yacco2::CAbs_lr1_sym&Sym);
 bool empty();
 void pop();
 void clean_up();
 Cparse_record*sf_by_sub(yacco2::UINT Sub);
 Cparse_record*sf_by_top(yacco2::UINT No);
 Cparse_record lr_stk__[C_MAX_LR_STK_ITEMS];
 yacco2::UINT top_sub__;
 Cparse_record*top__;
 Cparse_record*first_sf__;
 State*first_state__;
};

typedef void*LPVOID;
#if THREAD_LIBRARY_TO_USE__ == 1
 #define _YACCO2_CALL_TYPE
 typedef HANDLE MUTEX;
 typedef unsigned int THREAD_NO;
 typedef HANDLE THREAD;
 typedef HANDLE COND_VAR;
 typedef uintptr_t THR;
 typedef int THR_result;
 typedef THR(
 _YACCO2_CALL_TYPE
 *Type_pp_fnct_ptr)(yacco2::Parser*PP_requestor);
 typedef THR_result(
 _YACCO2_CALL_TYPE
 *Type_pc_fnct_ptr)(yacco2::Parser*PP_requestor);
 typedef THR(
  --stdcall
  *Type_pp_fnct_ptr_voidp)(yacco2::LPVOID PP_requestor);
#elif THREAD_LIBRARY_TO_USE__ == 0
 #define _YACCO2_CALL_TYPE
 typedef pthread_mutex_t MUTEX;
 typedef pthread_t THREAD_NO;
 typedef pthread_cond_t COND_VAR;
 typedef void*LPVOID;
 typedef LPVOID THR;
 typedef int THR_result;
 typedef pthread_t THREAD;
```

```
 typedef THR(*Type_pp_fnct_ptr)(yacco2::Parser*PP_requestor);
 typedef THR(*Type_pp_fnct_ptr_voidp)(yacco2::LPVOID PP_requestor);
 typedef THR_result(*Type_pc_fnct_ptr)(yacco2::Parser*PP_requestor);
#endif

typedef std::vector<yacco2::Thread_entry*> yacco2_threads_to_run_type;
typedef yacco2_threads_to_run_type::iterator yacco2_threads_to_run_iter_type;

struct worker_thread_blk;
typedef std::list<yacco2::worker_thread_blk*> Parallel_thread_list_type;
typedef Parallel_thread_list_type::iterator Parallel_thread_list_iterator_type;
typedef std::vector<yacco2::Parallel_thread_list_type> Parallel_thread_tbl_type;
typedef Parallel_thread_tbl_type::iterator Parallel_thread_tbl_iterator_type;
struct called_proc_entry{
 bool proc_call_in_use__;
};
typedef called_proc_entry Parallel_thread_proc_call_table_type;

struct Caccept_parse;
#define pp_accept_queue_size 8
typedef yacco2::Caccept_parse pp_accept_queue_type[pp_accept_queue_size];
typedef std::vector<std::string> gbl_file_map_type;
typedef std::set<yacco2::CAbs_lr1_sym*> set_of_objs_type;
typedef set_of_objs_type::iterator set_of_objs_iter_type;
extern Parallel_thread_tbl_type Parallel_thread_table;
extern Parallel_thread_proc_call_table_type Parallel_thread_proc_call_table[MAX_NO_THDS];
extern std::list<std::string> O2_LOGICALS__;
extern std::ofstream lrclog;
extern std::ofstream lrerrors;
extern yacco2::KCHARP Lr1_VERSION;
extern yacco2::KCHARP O2linker_VERSION;
extern yacco2::MUTEX TRACE_MU;
extern yacco2::MUTEX TH_TBL_MU;
extern yacco2::MUTEX SYM_TBL_MU;
extern yacco2::MUTEX TOKEN_MU;
extern yacco2::gbl_file_map_type FILE_TBL__;
extern yacco2::UINT FILE_CNT__;
extern std::vector<yacco2::UINT> STK_FILE_NOS__;
struct rc_map;
extern yacco2::rc_map RC__;
extern yacco2::Set_entry LRK_LA_EOLR_SET;
extern yacco2::Set_entry LRK_LA_QUE_SET;
extern int YACCO2_T__;
extern int YACCO2_TLEX__;
extern int YACCO2_MSG__;
extern int YACCO2_TH__;
extern int YACCO2_AR__;
extern int YACCO2_THP__;
extern int YACCO2_MU_TRACING__;
extern int YACCO2_MU_TH_TBL__;
extern int YACCO2_MU_GRAMMAR__;
extern void*THDS_STABLE__;
extern void*T_ARRAY_HAVING_THD_IDS__;
extern void*BIT_MAPS_FOR_SALE__;
extern int TOTAL_NO_BIT_WORDS__;
extern int BIT_MAP_IDX__;
extern CAbs_lr1_sym*PTR_LR1_eog__;

struct Set_entry{
```

```
 yacco2::UCHAR partition__;
 yacco2::UCHAR elements__;
};
struct Set_tbl{
 yacco2::UCHAR no_entries__;
 yacco2::Set_entry first_entry__[1];
};


template<typename Functor>
struct functor2{
 struct functor{};
 void operator()(Functor*Func){Func->operator()();};
};
template<typename T> class tble_lkup:public std::unary_function<T,T>{
 public:
  tble_lkup():lkup__(ON){};
  ~tble_lkup(){};
  virtual T operator()(T t)= 0;
  void turn_off_lkup(){
   lkup__= OFF;
  };
 void turn_on_lkup(){
  lkup__= ON;
 };
 bool lkup(){return lkup__;};
 bool lkup__;
};

typedef tble_lkup<yacco2::CAbs_lr1_sym*> tble_lkup_type;

struct rc_map{
 enum rc_size{rc_size_= ASCII_8_BIT+1};
 yacco2::CAbs_lr1_sym*
  map_char_to_raw_char_sym(yacco2::UINT Char,yacco2::UINT File,yacco2::UINT Pos
  ,UINT*Line_no,UINT*Pos_in_line);
 static char array_chr_sym__[SIZE_RC_MALLOC];
 static INT current_rc_malloc_sub__;
 static yacco2::CHARP chr_literal__[ASCII_8_BIT];
};

struct CAbs_lr1_sym{
 CAbs_lr1_sym(
  yacco2::KCHARP Id
  ,yacco2::FN_DTOR Dtor
  ,yacco2::USINT Enum_id
  ,bool Auto_delete
  ,bool Affected_by_abort);

 CAbs_lr1_sym(
  yacco2::KCHARP Id
  ,yacco2::FN_DTOR Dtor
  ,yacco2::USINT Enum_id
  ,bool Auto_delete
  ,bool Affected_by_abort
  ,yacco2::USINT Ext_file_no
  ,yacco2::UINT Rc_pos);

 CAbs_lr1_sym(
```

```
  yacco2::KCHARP Id
  ,yacco2::FN_DTOR Dtor
  ,yacco2::USINT Enum_id
  ,yacco2::Parser*P
  ,bool Auto_delete= false
  ,bool Affected_by_abort= false);

 yacco2::KCHARP id()const;
 yacco2::USINT enumerated_id()const;
 void set_enumerated_id(yacco2::USINT Id);
 void set_auto_delete(bool X);
 bool auto_delete()const;
 void set_affected_by_abort(bool X);
 bool affected_by_abort()const;
 yacco2::UINT rc_pos();
 void set_rc_pos(yacco2::UINT Pos);
 yacco2::UINT external_file_id();
 void set_external_file_id(yacco2::UINT File);
 void set_rc(yacco2::CAbs_lr1_sym&Rc
  ,yacco2::KCHARP GPS_FILE= __FILE__,yacco2::UINT GPS_LINE= __LINE__);
 yacco2::UINT line_no();
 void set_line_no(yacco2::UINT Line_no);
 yacco2::UINT pos_in_line();
 void set_pos_in_line(yacco2::UINT Pos_in_line);
 void set_line_no_and_pos_in_line(yacco2::CAbs_lr1_sym&Rc);
 void set_line_no_and_pos_in_line(yacco2::UINT Line_no,yacco2::UINT Pos_in_line);
 void set_who_created(yacco2::KCHARP File,yacco2::UINT Line_no);
 yacco2::UINT who_line_no();
 yacco2::KCHARP who_file();
 yacco2::Parser*parser();
 yacco2::FN_DTOR dtor();
 yacco2::USINT rhs_no_of_parms();
 yacco2::KCHARP id__;
 yacco2::FN_DTOR dtor__;
 yacco2::USINT enumerated_id__;
 bool auto_delete__;
 bool affected_by_abort__;
 UCHAR enum_id_set_partition_no()const;
 UCHAR enum_id_set_member()const;

 struct tok_co_ordinates{
  yacco2::KCHARP who_file__;
  yacco2::UINT who_line_no__;
  yacco2::UINT rc_pos__;
  yacco2::UINT line_no__;
  yacco2::USINT external_file_id__;
  yacco2::USINT pos_in_line__;
  Set_entry set_entry__;
 };
 struct rule_info{
  yacco2::Parser*parser__;
  yacco2::USINT rhs_no_of_parms__;
 };
 union{
  tok_co_ordinates tok_co_ords__;
  rule_info rule_info__;
 };
};
```

```
extern void LOCK_MUTEX(yacco2::MUTEX&Mu);
extern void UNLOCK_MUTEX(yacco2::MUTEX&Mu);
extern void LOCK_MUTEX_OF_CALLED_PARSER
 (yacco2::MUTEX&Mu,yacco2::Parser&parser,const char*Text);
extern void UNLOCK_MUTEX_OF_CALLED_PARSER
 (yacco2::MUTEX&Mu,yacco2::Parser&parser,const char*Text);

struct tok_base{
 tok_base(USINT RW):r_w_cnt__(RW){};
 virtual yacco2::UINT size()= 0;
 virtual yacco2::CAbs_lr1_sym*operator[](yacco2::UINT Pos)= 0;
 virtual void push_back(yacco2::CAbs_lr1_sym&Tok)= 0;
 virtual void clear()= 0;
 virtual bool empty()= 0;
 USINT r_w_cnt__;
};

template<typename Container> class tok_can:public tok_base{
 public:
 typedef Container value_type;
 typedef typename Container::size_type size_type;
 typedef typename Container::difference_type difference_type;

 typedef typename Container::iterator iterator;
 typedef typename Container::const_iterator const_iterator;
 typedef typename Container::reverse_iterator reverse_iterator;
 typedef typename Container::const_reverse_iterator const_reverse_iterator;

 typedef typename Container::pointer pointer;
 typedef typename Container::const_pointer const_pointer;
 typedef typename Container::reference reference;
 typedef typename Container::const_reference const_reference;
 tok_can():tok_base(1),pos__(0){};
 ~tok_can(){};
 yacco2::CAbs_lr1_sym*operator[](yacco2::UINT Pos){
  if(Pos>=container__.size()){
   if(YACCO2_T__!=0){
     LOCK_MUTEX(yacco2::TRACE_MU);
     if(yacco2::YACCO2_MU_TRACING__){
      yacco2::lrclog<<"YACCO2_MU_TRACING__::Acquired trace mu"<<std::endl;
     }
     yacco2::lrclog<<"YACCO2_T__::tok_can token eog: "
      <<PTR_LR1_eog__<<" pos: "<<Pos<<std::endl;

     if(yacco2::YACCO2_MU_TRACING__){
       yacco2::lrclog<<"YACCO2_MU_TRACING__::Releasing trace mu"<<std::endl;
     }
     UNLOCK_MUTEX(yacco2::TRACE_MU);
   }
   return PTR_LR1_eog__;
  }
  CAbs_lr1_sym*tok_(0);
  if(r_w_cnt__> 1){
    LOCK_MUTEX(yacco2::TOKEN_MU);
    tok_= container__[Pos];
    UNLOCK_MUTEX(yacco2::TOKEN_MU);
  }else{
   tok_= container__[Pos];
  }
```

```
  if(YACCO2_T__!=0){
    LOCK_MUTEX(yacco2::TRACE_MU);
    if(yacco2::YACCO2_MU_TRACING__){
      yacco2::lrclog<<"YACCO2_MU_TRACING__::Acquired trace mu"<<std::endl;
    }

    yacco2::lrclog<<"YACCO2_T__::tok_can token: "<<tok_->id__
     <<" *: "<<tok_<<" pos: "<<Pos <<" enum: "<<tok_->enumerated_id__
     <<'"'<<tok_->id__<<'"' <<std::endl;
    yacco2::lrclog <<"\t\t::GPS FILE: ";
    if(tok_->tok_co_ords__.external_file_id__!=MAX_USINT)
     yacco2::lrclog<<
            yacco2::FILE_TBL__[tok_->tok_co_ords__.external_file_id__].c_str();
    else
     yacco2::lrclog<<" No external file";

    yacco2::lrclog <<" GPS LINE: " <<tok_->tok_co_ords__.line_no__
      <<" GPS CHR POS: " <<tok_->tok_co_ords__.pos_in_line__ <<std::endl;

    if(yacco2::YACCO2_MU_TRACING__){
      yacco2::lrclog<<"YACCO2_MU_TRACING__::Releasing trace mu"<<std::endl;
    }
    UNLOCK_MUTEX(yacco2::TRACE_MU);
  }
  return tok_;
 };

 yacco2::UINT pos(){return pos__;};
 yacco2::UINT size(){return container__.size();};
 bool empty(){return container__.empty();};
 void push_back(yacco2::CAbs_lr1_sym&Tok){container__.push_back(&Tok);};
 void remove(){};
 void clear(){container__.clear();};
 Container&container(){return container__;};
 iterator begin(){return container__.begin();};
 iterator end(){return container__.end();};
 private:
  yacco2::UINT pos__;
  bool have_1st_rec__;
  Container container__;
};

typedef tok_base token_container_type;
typedef tok_can<std::vector<yacco2::CAbs_lr1_sym*> > GAGGLE;
typedef GAGGLE::iterator GAGGLE_ITER;
typedef GAGGLE TOKEN_GAGGLE;
typedef GAGGLE_ITER TOKEN_GAGGLE_ITER;

template<> class tok_can<std::ifstream> :public yacco2::tok_base{
 public:
  tok_can();
  tok_can(const char*File_name);
  ~tok_can();
  std::string&file_name();
  void set_file_name(const char*File_name);
  yacco2::CAbs_lr1_sym*operator[](yacco2::UINT Pos);
  yacco2::UINT pos();
  yacco2::UINT size();
  bool empty();
```

```
  void push_back(yacco2::CAbs_lr1_sym&Tok);
  void remove();
  void clear();
  TOKEN_GAGGLE&container();
  bool file_ok();
  void open_file();
  void close_file();
 private:
  std::ifstream file__;
  yacco2::UINT pos_;
  bool have_1st_rec__;
  std::ios::int_type eof_pos_;
  bool file_ok_;
  UINT line_no__;
  UINT pos_in_line__;
  TOKEN_GAGGLE container__;
  std::string file_name__;
  yacco2::UINT file_no__;
};

template<> class tok_can<std::string> :public yacco2::tok_base{
 public:
  tok_can();
  tok_can(const char*String,CAbs_lr1_sym*GPS= 0);
  ~tok_can();
  void set_string(const char*String);
  void reuse_string(const char*String,CAbs_lr1_sym*GPS= 0);
  yacco2::CAbs_lr1_sym*operator[](yacco2::UINT Pos);
  yacco2::UINT pos();
  yacco2::UINT size();
  bool empty();
  void push_back(yacco2::CAbs_lr1_sym&Tok);
  void remove();
  void clear();
  TOKEN_GAGGLE&container();
  std::string*string_used();
  void set_gps(CAbs_lr1_sym*Gps);
  yacco2::CAbs_lr1_sym*gps_used();
 private:
  std::string string__;
  yacco2::UINT pos_;
  bool have_1st_rec__;
  std::ios::int_type eof_pos_;
  UINT line_no__;
  UINT pos_in_line__;
  TOKEN_GAGGLE container__;
  CAbs_lr1_sym*eof_sym_;
  yacco2::UINT file_no__;
  int real_start_pos_in_line_;
  yacco2::CAbs_lr1_sym*gps__;
};

struct AST;
struct ast_base_stack;
typedef std::set<yacco2::INT> int_set_type;
typedef int_set_type::iterator int_set_iter_type;
typedef std::vector<yacco2::AST*> ast_vector_type;
typedef std::vector<yacco2::INT> ast_accept_node_type;
typedef enum{bypass_node,accept_node,stop_walking}functor_result_type;
```

```
typedef ast_vector_type Type_AST_ancestor_list;

template<class T> struct ast_functor{
 virtual functor_result_type operator()(T Ast_env)= 0;
};

typedef ast_functor<yacco2::ast_base_stack*> Type_AST_functor;
struct ast_base_stack{
 typedef enum n_action{init,left,visit,right,eoc}n_action_;
 struct s_rec{
  AST*node_;
  n_action_ act_;
 };
 ast_base_stack();
 ast_base_stack(Type_AST_functor*Action,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 s_rec*stk_rec(yacco2::INT I);
 void pop();
 void push(AST&Node,ast_base_stack::n_action Action);
 yacco2::INT cur_stk_index();
 s_rec*cur_stk_rec();
 yacco2::INT idx_;
 std::vector<s_rec> stk_;
 Type_AST_functor*action_;
 s_rec*cur_stk_rec_;
 yacco2::int_set_type*filter_;
 bool accept_opt_;
};

struct ast_stack{
 ast_stack(Type_AST_functor*Action,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 ast_base_stack base_stk_;
 virtual void exec()= 0;
 virtual void advance()= 0;
};

struct AST{
 AST(yacco2::CAbs_lr1_sym&Obj);
 AST();
 ~AST();
 static AST*restructure_2trees_into_1tree(AST&S1,AST&S2);
 static void crt_tree_of_1son(AST&Parent,AST&S1);
 static void crt_tree_of_2sons(AST&Parent,AST&S1,AST&S2);
 static void crt_tree_of_3sons(AST&Parent,AST&S1,AST&S2,AST&S3);
 static void crt_tree_of_4sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4);
 static void crt_tree_of_5sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4,AST&S5);
 static void crt_tree_of_6sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4,AST&S5,AST&S6);
 static void crt_tree_of_7sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4,AST&S5,AST&S6
  ,AST&S7);
 static void crt_tree_of_8sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4,AST&S5
  ,AST&S6,AST&S7,AST&S8);
 static void crt_tree_of_9sons(AST&Parent,AST&S1,AST&S2,AST&S3,AST&S4,AST&S5
  ,AST&S6,AST&S7,AST&S8,AST&S9);
 static void join_pts(AST&Parent,AST&Sibling);
 static void join_sts(AST&Elder_sibling,AST&Younger_sibling);
 static void ast_delete(AST&Node,bool Due_to_abort= false);
 static AST*find_depth(AST&Node,yacco2::INT Enum);
 static AST*find_breadth(AST&Node,yacco2::INT Enum);
 static yacco2::CAbs_lr1_sym*content(AST&Node);
 static AST*get_1st_son(AST&Node);
```

```
 static AST*get_2nd_son(AST&Node);
 static AST*get_3rd_son(AST&Node);
 static AST*get_4th_son(AST&Node);
 static AST*get_5th_son(AST&Node);
 static AST*get_6th_son(AST&Node);
 static AST*get_7th_son(AST&Node);
 static AST*get_8th_son(AST&Node);
 static AST*get_9th_son(AST&Node);
 static AST*get_spec_child(AST&Tree,yacco2::INT Cnt);
 static AST*get_child_at_end(AST&Tree);
 static AST*add_child_at_end(AST&Tree,AST&Child);
 static AST*get_younger_sibling(AST&Child,yacco2::INT Pos);
 static AST*get_older_sibling(AST&Child,yacco2::INT Pos);
 static AST*get_youngest_sibling(AST&Child);
 static AST*get_parent(AST&Child);
 static AST*common_ancestor
           (Type_AST_ancestor_list&ListA,Type_AST_ancestor_list&ListB);

 static AST*brother(AST&Node);
 static AST*previous(AST&Node);
 static void zero_1st_son(AST&Node);
 static void zero_2nd_son(AST&Node);
 static void zero_brother(AST&Node);
 static void zero_previous(AST&Node);
 static void zero_content(AST&Node);
 static void set_content(AST&Node,yacco2::CAbs_lr1_sym&Sym);
 static void set_content_wdelete(AST&Node,yacco2::CAbs_lr1_sym&Sym);
 static void set_previous(AST&Node,AST&Previous_node);
 static void wdelete(AST&Node,bool Wdelete);
 static bool wdelete(AST&Node);
 static void replace_node(AST&Old_to,AST&New_to);
 static void relink(AST&Previous,AST&Old_to,AST&New_to);
 static void relink_between(AST&Previous,AST&Old_to,AST&New_to);
 static void relink_after(AST&Previous,AST&New_to);
 static void relink_before(AST&Previous,AST&New_to);
 static void add_son_to_tree(AST&Parent,AST&Son);
 static AST*divorce_node_from_tree(AST&Node);
 static AST*clone_tree(AST&Node_to_copy,AST*Calling_node
                      ,ast_base_stack::n_action Relation= ast_base_stack::init);
 AST*lt_;
 AST*rt_;
 AST*pr_;
 yacco2::CAbs_lr1_sym*obj_;
 bool wdelete_;
};

template<> class tok_can<yacco2::AST*> :public yacco2::tok_base{
 public:
  tok_can(ast_stack&Walker);
  ~tok_can();

  yacco2::CAbs_lr1_sym*operator[](yacco2::UINT Pos);
  yacco2::UINT pos();
  yacco2::UINT size();
  bool empty();
  void push_back(yacco2::AST&Node);
  void push_back(yacco2::CAbs_lr1_sym&Node);
  void remove();
  void clear();
```

```
  yacco2::ast_stack&container();
  std::vector<yacco2::AST*> *nodes_visited();
  yacco2::AST*ast(yacco2::UINT Pos);
  yacco2::INT accept_node_level(yacco2::UINT Pos);

 private:
  volatile yacco2::UINT pos_;
  bool have_1st_rec__;
  bool tree_end_reached__;
  yacco2::ast_vector_type nodes_visited_;
  yacco2::ast_accept_node_type accept_node_level_;
  yacco2::ast_stack&traverse_;
};

struct Per_rule_s_reuse_table;
struct reuse_rule_list;
struct reuse_rule_list{
 reuse_rule_list():younger_(0),older_(0),reuse_rule_entry_(0),per_rule_tbl_ptr_(0){};
 reuse_rule_list*younger_;
 reuse_rule_list*older_;
 Rule_s_reuse_entry*reuse_rule_entry_;
 Per_rule_s_reuse_table*per_rule_tbl_ptr_;
};
struct Rule_s_reuse_entry{
 reuse_rule_list its_linked_list_;
 CAbs_lr1_sym*rule_;
 Rule_s_reuse_entry():rule_(0){};
 ~Rule_s_reuse_entry(){
  if(rule_==0)return;
  delete rule_;
 };
};

struct Per_rule_s_reuse_table{
 reuse_rule_list*in_use_list_;
 reuse_rule_list*for_use_list_;
 Per_rule_s_reuse_table():in_use_list_(0),for_use_list_(0){};
};

struct Fsm_rules_reuse_table{
 int no_rules_entries_;
 Per_rule_s_reuse_table*per_rule_s_table_[1];
};

struct State{
 yacco2::UINT state_no__;
 yacco2::Shift_entry*parallel_shift__;
 yacco2::Shift_entry*all_shift__;
 yacco2::Shift_entry*inv_shift__;
 yacco2::Shift_entry*proc_call_shift__;
 yacco2::Shift_tbl*shift_tbl_ptr__;
 yacco2::Reduce_tbl*reduce_tbl_ptr__;
 yacco2::State_s_thread_tbl*state_s_thread_tbl__;
 yacco2::Type_pc_fnct_ptr proc_call_addr__;
 yacco2::Shift_entry*questionable_shift__;
};

struct Shift_entry{
 yacco2::USINT id__;
```

```
 yacco2::State*goto__;
};
struct Shift_tbl{
 yacco2::USINT no_entries__;
 yacco2::Shift_entry first_entry__[1];
};

struct Reduce_entry{
 yacco2::Set_tbl*la_set__;
 yacco2::USINT rhs_id__;
};
struct Reduce_tbl{
 yacco2::USINT no_entries__;
 yacco2::Reduce_entry first_entry__[1];
};

struct Thread_entry{
 yacco2::KCHARP thread_fnct_name__;
 yacco2::Type_pp_fnct_ptr thread_fnct_ptr__;
 yacco2::USINT thd_id__;
 yacco2::Type_pc_fnct_ptr proc_thread_fnct_ptr__;
};

struct thread_array_record{
 yacco2::USINT no_entries__;
 yacco2::Thread_entry*first_entry__[1];
};

struct State_s_thread_tbl{
 yacco2::USINT no_entries__;
 yacco2::Type_pp_fnct_ptr ar_fnct_ptr__;
 yacco2::ULINT(*thd_id_bit_map__);
 yacco2::Thread_entry*first_entry__[1];
};

struct thd_ids_having_T{
 yacco2::ULINT first_thd_id__[1];
};
struct T_array_having_thd_ids{
 yacco2::USINT no_of_T__;
 yacco2::thd_ids_having_T*first_entry__[1];
};

class CAbs_fsm{
 public:
  virtual void op()= 0;
  virtual bool failed()= 0;
  yacco2::KCHARP id();
  yacco2::KCHARP version();
  yacco2::KCHARP date();
  bool debug();
  yacco2::KCHARP comments();
  yacco2::KCHARP gened_date();
  yacco2::State*start_state();
  virtual~CAbs_fsm();
  virtual void reduce_rhs_of_rule
   (yacco2::UINT Sub_rule_no,yacco2::Rule_s_reuse_entry**Recycled_rule)= 0;
  yacco2::Parser*parser();
  void parser(yacco2::Parser&A);
```

```
  void find_a_recycled_rule(Per_rule_s_reuse_table*Reuse_rule_table
       ,Rule_s_reuse_entry**Reuse_rule_entry);
  void recycle_rule(Rule_s_reuse_entry*Rule_to_recycle);
 protected:
  CAbs_fsm(yacco2::KCHARP Id
  ,yacco2::KCHARP Version
  ,yacco2::KCHARP Date
  ,bool Debug
  ,yacco2::KCHARP Comments
  ,yacco2::KCHARP Gened_date
  ,yacco2::State&Start_state
  );
 public:
  yacco2::KCHARP id__;
  yacco2::KCHARP version__;
  yacco2::KCHARP date__;
  bool debug__;
  yacco2::KCHARP comments__;
  yacco2::KCHARP gened_date__;
  yacco2::State*start_state__;
  yacco2::Parser*parser__;
};

struct worker_thread_blk{
 worker_thread_blk();
 worker_thread_blk(yacco2::Parser*Grammar_s_parser,yacco2::Parser*Calling_parser);
 yacco2::Parser*grammar_s_parser__;
 int status__;
 int run_cnt__;
 int thd_id__;
 void set_waiting_for_work();
};

struct Caccept_parse{
 Caccept_parse
  (yacco2::Parser&Th_reporting_success
  ,yacco2::CAbs_lr1_sym&Accept_token
  ,yacco2::UINT Accept_token_pos
  ,yacco2::CAbs_lr1_sym&La_token
  ,yacco2::UINT La_token_pos);
 Caccept_parse();
 void initialize_it();
 void fill_it(Caccept_parse&Accept_parse);
 void fill_it(yacco2::Parser&Th_reporting_success
 ,yacco2::CAbs_lr1_sym&Accept_token
 ,yacco2::UINT Accept_token_pos
 ,yacco2::CAbs_lr1_sym&La_token
 ,yacco2::UINT La_token_pos);

 ~Caccept_parse();
 yacco2::Parser*th_reporting_success__;
 yacco2::CAbs_lr1_sym*accept_token__;
 yacco2::UINT accept_token_pos__;
 yacco2::CAbs_lr1_sym*la_token__;
 yacco2::UINT la_token_pos__;
};


struct Parser{
```

```
enum parse_result{erred,accepted,reduced,paralleled,no_thds_to_run};

yacco2::CAbs_fsm*fsm_tbl__;
yacco2::KCHARP thread_name__;
yacco2::Thread_entry*thread_entry__;
yacco2::token_container_type*token_supplier__;
yacco2::token_container_type*token_producer__;
yacco2::token_container_type*recycle_bin__;
yacco2::token_container_type*error_queue__;
yacco2::lr_stk parse_stack__;
yacco2::CAbs_lr1_sym*current_token__;
yacco2::UINT current_token_pos__;
yacco2::CAbs_lr1_sym*start_token__;
yacco2::UINT start_token_pos__;
yacco2::tble_lkup_type*sym_lookup_functor__;
bool abort_parse__;
bool stop_parse__;
bool use_all_shift__;
bool has_questionable_shift_occured__;
yacco2::Parser*from_thread__;
yacco2::THREAD_NO thread_no__;
yacco2::COND_VAR cv__;
yacco2::MUTEX mu__;
int cv_cond__;
yacco2::worker_thread_blk th_blk__;

yacco2::pp_accept_queue_type pp_accept_queue__;
int pp_accept_queue_idx__;
yacco2::INT th_active_cnt__;
yacco2::INT th_accepting_cnt__;

yacco2::Parser*pp_requesting_parallelism__;
yacco2::INT msg_id__;
yacco2::Caccept_parse*arbitrated_token__;
yacco2::Caccept_parse pp_rsvp__;
int no_competing_pp_ths__;
int no_requested_ths_to_run__;
yacco2::yacco2_threads_to_run_type th_lst__;
bool launched_as_procedure__;
USINT supplier_r_w_cnt__;

Parser(yacco2::CAbs_fsm&Fsm_tbl
 ,yacco2::token_container_type*Token_supplier
 ,yacco2::token_container_type*Token_producer
 ,yacco2::UINT Token_supplier_key_pos= Token_start_pos
 ,yacco2::token_container_type*Error_queue= 0
 ,yacco2::token_container_type*Recycle_bin= 0
 ,yacco2::tble_lkup_type*Sym_lookup_functor= 0
 ,bool Use_all_shift= ON);
Parser(yacco2::CAbs_fsm&Fsm_tbl
 ,yacco2::Thread_entry&Thread_entry
 ,yacco2::Parser*Calling_parser);
Parser(yacco2::CAbs_fsm&Fsm_tbl
 ,yacco2::Parser*Calling_parser);
~Parser();

parse_result parse();
void shift(yacco2::Shift_entry&SE);
void invisible_shift(yacco2::Shift_entry&SE);
```

```
void questionable_shift(yacco2::Shift_entry&SE);
void all_shift(yacco2::Shift_entry&SE);
void parallel_shift(yacco2::CAbs_lr1_sym&Accept_terminal);
void proc_call_shift(yacco2::CAbs_lr1_sym&Accept_terminal);
parse_result reduce(yacco2::Reduce_entry&RE);
parse_result parallel_parse();
parse_result proc_call_parse();
parse_result start_parallel_parsing(yacco2::State&S);
THR_result chained_proc_call_parsing(yacco2::State&S);
parse_result start_manually_parallel_parsing(yacco2::USINT Thread_id);
yacco2::Shift_entry*find_cur_T_shift_entry();
yacco2::Shift_entry*find_R_or_paralleled_T_shift_entry(yacco2::USINT Enum_id);
yacco2::Reduce_entry*find_questionable_sym_in_reduce_lookahead();
yacco2::Reduce_entry*find_reduce_entry();
yacco2::Reduce_entry*find_parallel_reduce_entry();
yacco2::Reduce_entry*find_proc_call_reduce_entry();

yacco2::token_container_type*token_supplier();
void set_token_supplier(yacco2::token_container_type&Token_supplier);
yacco2::token_container_type*token_producer();
void set_token_producer(yacco2::token_container_type&Token_producer);
yacco2::token_container_type*recycle_bin();
void set_recycle_bin(yacco2::token_container_type&Recycle_bin);
void set_error_queue(yacco2::token_container_type&Error_queue);
yacco2::token_container_type*error_queue();
void add_token_to_supplier(yacco2::CAbs_lr1_sym&Token);
void add_token_to_producer(yacco2::CAbs_lr1_sym&Token);
void add_token_to_recycle_bin(yacco2::CAbs_lr1_sym&Token);
void add_token_to_error_queue(yacco2::CAbs_lr1_sym&Token);

void get_shift_s_next_token();
yacco2::CAbs_lr1_sym*get_next_token();
yacco2::CAbs_lr1_sym*get_spec_token(yacco2::UINT Pos);
yacco2::CAbs_lr1_sym*current_token();
yacco2::CAbs_lr1_sym*start_token();
void set_start_token(yacco2::CAbs_lr1_sym&Start_tok);
yacco2::UINT start_token_pos();
void set_start_token_pos(yacco2::UINT Pos);
void reset_current_token(yacco2::UINT Pos);
void override_current_token(yacco2::CAbs_lr1_sym&Current_token,yacco2::UINT Pos);
void override_current_token_pos(yacco2::UINT Pos);
yacco2::UINT current_token_pos();

void cleanup_stack_due_to_abort();
yacco2::lr_stk*parse_stack();
yacco2::INT no_items_on_stack();
yacco2::Cparse_record*get_stack_record(yacco2::INT Pos);
yacco2::Cparse_record*top_stack_record();
void remove_from_stack(yacco2::INT No_to_remove);
void add_to_stack(yacco2::State&State_no);
yacco2::INT current_stack_pos();
void clear_parse_stack();
yacco2::CAbs_lr1_sym*get_spec_stack_token(yacco2::UINT Pos);

void set_use_all_shift_on();
void set_use_all_shift_off();
bool use_all_shift();
bool abort_parse();
void set_abort_parse(bool Abort);
```

```
 bool stop_parse();
 void set_stop_parse(bool Stop);

 yacco2::CAbs_fsm*fsm_tbl();
 void fsm_tbl(yacco2::CAbs_fsm*Fsm_tbl);

 yacco2::tble_lkup_type*sym_lookup_functor();
 Parser::parse_result parallel_parse_successful();
 Parser::parse_result parallel_parse_unsuccessful();
 Parser::parse_result proc_call_parse_successful();
 Parser::parse_result proc_call_parse_unsuccessful();
 bool spawn_thread_manually(yacco2::USINT Thread_id);

 yacco2::Parser*from_thread();
 yacco2::KCHARP thread_name();
 yacco2::Thread_entry*thread_entry();
 void post_event_to_requesting_grammar
  (yacco2::Parser&To_thread
  ,yacco2::INT Message_id
  ,yacco2::Parser&From_thread);
 void wait_for_event();
 bool start_threads();
 THR_result start_procedure_call(yacco2::State&S);
 void put_T_into_accept_queue(yacco2::Caccept_parse&Parm);
 void clean_up();
 void call_arbitrator(yacco2::Type_pp_fnct_ptr The_judge);
 bool have_all_threads_reported_back();
 void abort_accept_queue_irregularities(yacco2::Caccept_parse&Calling_parm);
 void abort_no_selected_accept_parse_in_arbitrator();
};

struct Source_info{
 Source_info(yacco2::KCHARP File,yacco2::UINT Line);
 void w_info();
 yacco2::KCHARP file__;
 yacco2::INT line__;
};
struct Yacco2_faulty_precondition:Source_info{
 Yacco2_faulty_precondition(yacco2::KCHARP Message
  ,yacco2::KCHARP File= __FILE__
  ,yacco2::UINT Line= __LINE__);
};

struct Yacco2_faulty_postcondition:Source_info{
 Yacco2_faulty_postcondition(yacco2::KCHARP Message
  ,yacco2::KCHARP File= __FILE__
  ,yacco2::UINT Line= __LINE__);
};

struct ast_postfix:public ast_stack{
 ast_postfix(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void exec();
 void advance();
};

struct ast_prefix:public ast_stack{
 ast_prefix(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
```

```
 void exec();
 void advance();
};

struct ast_postfix_1forest:public ast_stack{
 ast_postfix_1forest(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void exec();
 void advance();
};

struct ast_prefix_1forest:public ast_stack{
 ast_prefix_1forest(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void exec();
  void advance();
};

struct ast_breadth_only:public ast_stack{
ast_breadth_only(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void exec();
 void advance();
};

struct ast_prefix_wbreadth_only:public ast_stack{
 ast_prefix_wbreadth_only(AST&Forest,Type_AST_functor*Action
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void exec();
 void advance();
};

struct ast_moonwalk_looking_for_ancestors{
 ast_moonwalk_looking_for_ancestors
  (AST&Moonchild,USINT Goal,Type_AST_ancestor_list&Ancestors,Type_AST_functor*Functor
  ,yacco2::int_set_type*Filter= 0,bool Accept_opt= true);
 void let_s_moonwalk();
 bool deal_with_parent(AST*Parent);
 functor_result_type let_s_functor(AST*Parent);
 bool deal_with_functor(AST*Parent);
 AST*moonchild_;
 USINT goal_;
 Type_AST_ancestor_list*ancestor_list_;
 Type_AST_functor*functor_;
 yacco2::int_set_type*filter_;
 bool filter_type_;
 bool filter_provided_;
};

struct insert_back_recycled_items_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
 void insert_node(yacco2::AST&Inode);
 yacco2::AST*new_root();
 void insert_before();
 private:
  yacco2::ast_base_stack*stk_env_;
  yacco2::INT idx_;
  yacco2::AST*cnode_;
  yacco2::ast_base_stack::s_rec*srec_;
```

```
  yacco2::AST*insert_node_;
  yacco2::AST*new_root_;
};

struct tok_can_ast_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
};

struct tok_can_ast_no_stop_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
};

struct tok_can_ast_bypass_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
};

struct prt_ast_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
 typedef void(*PF)(AST*);
 prt_ast_functor(PF Func,std::ofstream*Ofile= 0);
 void reset_cnt();
 private:
  yacco2::ast_base_stack*stk_env_;
  yacco2::INT idx_;
  yacco2::AST*cnode_;
  yacco2::ast_base_stack::s_rec*srec_;
  PF prt_funct_;
  yacco2::INT cnt_;
  char how_[3];
  std::ofstream*ofile_;
};

struct fire_a_func_ast_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
 typedef void(*PF)(AST*);
 fire_a_func_ast_functor(PF Func);
 private:
  yacco2::ast_base_stack*stk_env_;
  yacco2::INT idx_;
  yacco2::AST*cnode_;
  yacco2::ast_base_stack::s_rec*srec_;
  PF a_funct_;
};

struct str_ast_functor:public Type_AST_functor{
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
 typedef void(*PF)(AST*,Type_AST_functor*);
 str_ast_functor(PF Func);
 std::string source_str_;
 private:
  yacco2::ast_base_stack*stk_env_;
  yacco2::INT idx_;
  yacco2::AST*cnode_;
  yacco2::ast_base_stack::s_rec*srec_;
  PF prt_funct_;
  char how_[3];
};

struct remove_unwanted_ast_functor:public Type_AST_functor{
```

```
 functor_result_type operator()(yacco2::ast_base_stack*Stk_env);
 void possible_delete();
 ~remove_unwanted_ast_functor();

 private:
  yacco2::ast_base_stack*stk_env_;
  yacco2::INT idx_;
  yacco2::AST*cnode_;
  yacco2::ast_base_stack::s_rec*srec_;
};

extern void create_set_entry(yacco2::USINT Enum_id,yacco2::Set_entry&Set);
extern void CREATE_MUTEX(yacco2::MUTEX&Mu);
extern void LOCK_MUTEX(yacco2::MUTEX&Mu);
extern void UNLOCK_MUTEX(yacco2::MUTEX&Mu);
extern void LOCK_MUTEX_OF_CALLED_PARSER
    (yacco2::MUTEX&Mu,yacco2::Parser&parser,const char*Text);
extern void UNLOCK_MUTEX_OF_CALLED_PARSER
    (yacco2::MUTEX&Mu,yacco2::Parser&parser,const char*Text);
extern void DESTROY_MUTEX(yacco2::MUTEX&Mu);
extern void CREATE_COND_VAR(yacco2::COND_VAR&Cv);
extern void COND_WAIT(yacco2::COND_VAR&Cv,yacco2::MUTEX&Mu,yacco2::Parser&parser);
extern void SIGNAL_COND_VAR(yacco2::Parser&To_thread,yacco2::Parser&parser);
extern void DESTROY_COND_VAR(yacco2::COND_VAR&Cv);
extern yacco2::THR_result
CREATE_THREAD(yacco2::Type_pp_fnct_ptr Thread,yacco2::Parser&Parser_requesting_parallelism);
extern THREAD_NO THREAD_SELF();

extern void Parallel_threads_shutdown(yacco2::Parser&PP);
extern yacco2::THR
  _YACCO2_CALL_TYPE AR_for_manual_thread_spawning(yacco2::Parser*Caller_pp);
extern yacco2::Type_pp_fnct_ptr PTR_AR_for_manual_thread_spawning;

extern void find_threads_by_first_set
 (yacco2::USINT Current_T_id
 ,yacco2::yacco2_threads_to_run_type&Th_list
 ,yacco2::State_s_thread_tbl&P_tbl);

extern void Delete_tokens(yacco2::TOKEN_GAGGLE&Tks,bool Do_delete= OFF);
extern void Clear_yacco2_opened_files_dictionary();
extern bool trace_parser_env(Parser*parser,bool Trace_type);
};

namespace NS_yacco2_k_symbols{
 extern yacco2::CAbs_lr1_sym*PTR_LR1_questionable_shift_operator__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_eog__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_eolr__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_parallel_operator__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_fset_transience_operator__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_invisible_shift_operator__;
 extern yacco2::CAbs_lr1_sym*PTR_LR1_all_shift_operator__;
};

#define ssNEW_TRACEss(ssPss,ssQss) \
yacco2::lrclog << "\t!!!!! new adr: " << (void*)ssPss \
<< " " << #ssQss << std::endl; \
yacco2::lrclog << "\tfile: " << __FILE__ << " line: " << __LINE__ << std::endl;
#define ssP_TRACEss(ssPss,ssQss) \
yacco2::lrclog \
```

```
<< '\t' << Parse_env->thread_no__ \
<< "\t!!!!! new adr: " << (void*)ssPss \
<< " " << #ssQss << std::endl; \
yacco2::lrclog \
<< "\tfile: " << __FILE__ << " line: " << __LINE__ << std::endl;
#define sstrace_terminalsss \
if(yacco2::YACCO2_TLEX__){\
bool to_trace_or_not_to =  trace_parser_env(rule_info__.parser__,FORCE_STK_TRACE); \
if(to_trace_or_not_to == true){\
yacco2::lrclog \
<< "\tYACCO2_TLEX__::" << rule_info__.parser__->thread_no__ \
<< rule_info__.parser__->fsm_tbl__->id__ << "::" << id__ << "::op()\n"; \
}\
}
#define sstrace_rulesss \
if(yacco2::YACCO2_TLEX__){\
 bool to_trace_or_not_to =  trace_parser_env(rule_info__.parser__,FORCE_STK_TRACE); \
if(to_trace_or_not_to == true){\
  yacco2::lrclog << "\tYACCO2_TLEX__::" << rule_info__.parser__->thread_no__ \
  << "::" << rule_info__.parser__->fsm_tbl__->id__ << "::" << id__ << "::op()\n"; \
}\
}
#define sstrace_sub_rulesss \
if(yacco2::YACCO2_TLEX__){\
  bool to_trace_or_not_to =  trace_parser_env(rule_info__.parser__,FORCE_STK_TRACE); \
 if(to_trace_or_not_to == true){\
 yacco2::lrclog << "\tYACCO2_TLEX__::" << rule_info__.parser__->thread_no__ \
  << "::" << rule_info__.parser__->fsm_tbl__->id__ << "::" << id__ << "::op()\n"; \
}\
}
#define sstrace_stack_rtnsss \
if(yacco2::YACCO2_TLEX__){\
 bool to_trace_or_not_to =  trace_parser_env(Parse_env,FORCE_STK_TRACE)); \
if(to_trace_or_not_to == true){\
 yacco2::lrclog \
 << "\tYACCO2_TLEX__::" << Parse_env->thread_no__ \
 << "::" << Parse_env->fsm_tbl__->id__ << "::op() sym: " << id__ << std::endl; \
}\
}

#define sstrace_parallel_supportss(ssPROC_NAME) \
if(yacco2::YACCO2_TLEX__){\
  Parser* _ap =  parser_of_parallel_support(); \
  yacco2::lrclog  << "YACCO2_TLEX__::" << pthread_self() << "::" \
  << _ap->fsm_tbl__->id__ << "::" \
  << ' ' << #ssPROC_NAME << " this:: " << this << std::endl; \
  yacco2::lrclog  << "\tYACCO2_TLEX__:: parser_of_parallel_support:: " << _ap << std::endl; \
  yacco2::lrclog  << "\tself thread id:: " << thread_no__ << std::endl; \
  yacco2::lrclog \
<< "\tYACCO2_TLEX__:: embedded thread id:: " << embedded_thread_no() << std::endl;\
  }

#define  sstrace_parallel_support_envss(ssPROC_NAME) \
if(yacco2::YACCO2_TLEX__){\
  yacco2::lrclog  << "YACCO2_TLEX__::" << GetCurrentThreadid__ << "::" \
  << fsm_tbl__->id__ << "::" \
  << ' ' << #ssPROC_NAME << " this:: " << this << std::endl; \
  yacco2::lrclog  << "\tYACCO2_TLEX__:: self thread id:: " << thread_no__ << std::endl;\
  }
```

```
#endif
```

Appendix C: raw character mapping

.1 Canned enumerates

Due to space, all the enumerate labels are omitted from the tables below. To derive any of them use the struct name and prefix it with `T_` and suffix it with `_`. Drawing from the `eog` symbol, its enumerate label would be `T_LR1_eog_`.

.1.1 Lr constant symbols

Grammar literal	Struct name
|?|	LR1_questionable_shift_operator
eog	LR1_eog
eolr	LR1_eolr
|||	LR1_parallel_operator
|r|	LR1_reduce_operator
|.|	LR1_invisible_shift_operator
|+|	LR1_all_shift_operator
|t|	LR1_fset_transience_operator

Table 1: Lr constant symbols

`|r|` is the only symbol not to be used by the grammar writer. Also note that the above literal values are not quoted though they could have been. O_2 parsing of the grammar supports both symbol monikers. The only caveat is if the literal has some form of white space like spaces or tabs then it must be quoted.

.1.2 Raw characters symbols

The following tables 3–5 have the ASCII display character as ther literal value. Their struct name also uses its customed name.

Grammar literal	Struct name	Grammar literal	Struct name
"x00"	raw_nul	"x01"	raw_soh
"x02"	raw_stx	"x03"	raw_etx
"x04"	raw_eot	"x05"	raw_enq
"x06"	raw_ack	"x07"	raw_bel
"x08"	raw_bs	"x09"	raw_ht
"x0a"	raw_lf	"x0b"	raw_vt
"x0c"	raw_ff	"x0d"	raw_cr
"x0e"	raw_so	"x0f"	raw_si
"x10"	raw_dle	"x11"	raw_dc1
"x12"	raw_dc2	"x13"	raw_dc3
"x14"	raw_dc4	"x15"	raw_nak
"x16"	raw_syn	"x17"	raw_etb
"x18"	raw_can	"x19"	raw_em
"x1a"	raw_sub	"x1b"	raw_esc
"x1c"	raw_fs	"x1d"	raw_gs
"x1e"	raw_rs	"x1f"	raw_us

Table 2: Raw characters symbols: x00--x1f

Grammar literal	Struct name	Grammar literal	Struct name
" "	raw_sp	"!"	raw_exclam
"\""	raw_dbl_quote	"#"	raw_no_sign
"$"	raw_dollar_sign	"%"	raw_percent
"&"	raw_ampersign	"'"	raw_right_quote
"("	raw_open_bracket	")"	raw_close_bracket
"*"	raw_asteric	"+"	raw_plus
","	raw_comma	"-"	raw_minus
"."	raw_period	"/"	raw_slash
"0"	raw_0	"1"	raw_1
"2"	raw_2	"3"	raw_3
"4"	raw_4	"5"	raw_5
"6"	raw_6	"7"	raw_7
"8"	raw_8	"9"	raw_9
":"	raw_colon	";"	raw_semi_colon
"<"	raw_less_than	"="	raw_eq
">"	raw_gt_than	"?"	raw_question_mark

Table 3: Raw characters symbols: x20--x3f

Grammar literal	Struct name	Grammar literal	Struct name
"@"	raw_at_sign		
"A"	raw_A	"B"	raw_B
"C"	raw_C	"D"	raw_D
"E"	raw_E	"F"	raw_F
"G"	raw_G	"H"	raw_H
"I"	raw_I	"J"	raw_J
"K"	raw_K	"L"	raw_L
"M"	raw_M	"N"	raw_N
"O"	raw_O	"P"	raw_P
"Q"	raw_Q	"R"	raw_R
"S"	raw_S	"T"	raw_T
"U"	raw_U	"V"	raw_V
"W"	raw_W	"X"	raw_X
"Y"	raw_Y	"Z"	raw_Z
"["	raw_open_sq_bracket	"\\"	raw_back_slash
"]"	raw_close_sq_bracket	"^"	raw_up_arrow
"_"	raw_under_score	"‘"	raw_left_quote

Table 4: Raw characters symbols: x40--x60

Grammar literal	Struct name	Grammar literal	Struct name
"a"	raw_a	"b"	raw_b
"c"	raw_c	"d"	raw_d
"e"	raw_e	"f"	raw_f
"g"	raw_g	"h"	raw_h
"i"	raw_i	"j"	raw_j
"k"	raw_k	"l"	raw_l
"m"	raw_m	"n"	raw_n
"o"	raw_o	"p"	raw_p
"q"	raw_q	"r"	raw_r
"s"	raw_s	"t"	raw_t
"u"	raw_u	"v"	raw_v
"w"	raw_w	"x"	raw_x
"y"	raw_y	"z"	raw_z
"{"	raw_open_brace	"|"	raw_vertical_line
"}"	raw_close_brace	"~"	raw_tilde
"x7f"	raw_del		

Table 5: Raw characters symbols: x61--x7f

The above table's exception to its literal values is the last entry: "x7f". Please see the next table's comments regarding this coding pattern.

The pattern above for all the conditional display characters is: xyy where each y digit draws from the hexadecimal digits 0–9, a–f. For the balance of the raw characters ranged between xa0–xff, substitute the

Grammar literal	Struct name	Grammar literal	Struct name
"x80"	raw_x80	"x81"	raw_x81
"x82"	raw_x82	"x83"	raw_x83
"x84"	raw_x84	"x85"	raw_x85
"x86"	raw_x86	"x87"	raw_x87
"x88"	raw_x88	"x89"	raw_x89
"x8a"	raw_x8a	"x8b"	raw_x8b
"x8c"	raw_x8c	"x8d"	raw_x8d
"x8e"	raw_x8e	"x8f"	raw_x8f

Table 6: Raw characters symbols: x80--x8f

hexdigits to arrive at their grammar literal and enumerate label. For example, to manufacture the character for decimal value 254, its hex equivalent is "fe". So this character's grammar literal is `"xfe"`, its struct name `raw_xfe`, and its enumerate label `T_raw_xfe_`.

Bibliography

[1] Hans Christian Andersen. *New Fairy Tales*. 1844.

[2] antlr.org. ANTLR Parser Generator
www.antlr.org.

[3] Lewis Carroll. *Alice's Adventures in Wonderland*. 1865.

[4] dinosaur.computertools.net. The Lex and Yacc Page
http://dinosaur.computertools.net.

[5] gnu.org. GNU Operating System
www.gnu.org.

[6] John D. Hobby. *A User's Manual for MetaPost*. AT&T Bell Laboratories, Murray Hill, NJ 07974.

[7] John E. Hopcroft and Jeffrey D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley Publishing Company, Reading, MA, USA, 1969.

[8] Guy Steele James Gosling, Bill Joy. *The JAVA(TM) Language Specification*. Addison-Wesley Publishing Company, 1996.

[9] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report, Second Edition*. Springer-Verlag, 1974.

[10] S. C. Johnson. Yacc — yet another compiler-compiler. Technical Report 32, Murray Hill, NJ 07974, 1975.

[11] Nicolai M. Josuttis. *The C++ Standard Library — A Tutorial and Reference*. Addison-Wesley Publishing Company, 2004.

[12] Donald E. Knuth. On the translation of languages from left to right. In *Information and Control*, volume 8 of *6*, pages 607–639, 1965.

[13] Donald E. Knuth. *Literate Programming*. Stanford, California: Center for the Study of Language and Information, 1992.

[14] Donald E. Knuth. *The METAFONTbook*. Addison-Wesley Publishing Company, Reading, MA, USA, 1996.

[15] Donald E. Knuth. *The TeXbook*. Addison-Wesley Publishing Company, Reading, MA, USA, 1997.

[16] Donald E. Knuth. *The Art of Computer Programming Volume 1 Fundamental Algorithms Third Edition*. Addison-Wesley Publishing Company, Reading, MA, USA, 2002.

[17] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0*. Addison-Wesley Publishing Company, Reading, MA, USA, 1993.

[18] Bjarne Stroustrup. *The C++ Programming Language Special Edition*. Addison-Wesley Publishing Company, 2000.

[19] tug.org. TeX Users Group web site
www.tug.org.

[20] wikipedia. http://en.wikipedia.org/wiki/
Literate_programming.

[21] wikipedia. http://en.wikipedia.org/wiki/POSIX_Threads.

Index