1. License.
Gened date: January 31, 2015
Copyright © 1998-2015 Dave Bone

This Source Code Form is subject to the terms of the Mozilla Public License, v. 2.0. If a copy of the MPL was not distributed with this file, You can obtain one at http://mozilla.org/MPL/2.0/.

2. Summary of Yacco2's user library.
These are the building blocks of various definitions for all derived code emitted from Yacco2 with their runtime objects. All code blocks are genereted by *cweb*'s ctangle program drawn from their source file names having an extension of ".w". Points 8 and 9 are created from the *thread.w* source. The following are the outputted files:

1) *yacco2.h* — common definitions for all implementations and use
2) *yacco2.cpp* — common parts of yacco2's library created from this document
3) *wthread.cpp* — thread components
4) *wrc.cpp* — raw characters mapping into terminals
5) *wset.cpp* — set routines for the finite automaton tables
6) *wpp_core.cpp wproc_pp_core.cpp* — include code for generated pp threads
6.5) *wpp_core.cpp* thread, while *wproc_pp_core.cpp* procedure call version
7) *wtok_can.cpp* — specialized token containers: reads chr from file and string
8) *war_begin_code.h* — arbitrator's start code
9) *war_end_code.h* — arbitrator's end code
10) *wtree.cpp* — tree container, walkers, and functors

The 3 files generated outside this environment and referenced within Yacco2's library:

1) *yacco2_k_symbols.h* — lr k terminal definitions
2) *yacco2_characters.h* — raw character terminal definitions
3) *yacco2_T_enumeration.h* — enumeration of symbols

Some Yacco2 memorabilia:

1) yacco2 — library namespace
2) directory — "/usr/local/yacco2/library"
3) *wlibrary.w* — yacco2's *cweb* document
4) Look at the *Global macro definitions* and *Typedef* for limitations

At the end of this document is a *Notes to myself* section that you should read. These are a quasi set of ramblings on old / new reasons for changes, whys of the current implementation, and items for future redress. Please have a browse during this document reading. The notes are in an order of my programming thought zones while being developed.

3. Introduction to Yacco2's parse library.
Welcome to *Yacco2*'s library. This is the *oracle* of *typedefs*, *macros* of assorted functionality, and *constant* definitions. By having a common source code generator of definitions for the library, it should make this project easier to maintain and evolve. Instead of using the basic type definitions of the C++ language, I felt the **typedef** facility will make it easier to port the project onto another platform of nbit evolution — ahh the crazy world of bit envy 16...32...64 etc. Any inconsistency within the c language like **char** and its smorgasbord of flavors should be minimized by this approach to handling the pain-no-gain syndrome of supported systems.

Now I'm a fan of macros as it gives a nice way to dynamicly generate source code patterns. Unfortunately, the c language preprocessor was a hack that people are still living with while the PDP11 macro assembler facility of bygone years from now defunct Digital Equipment Corporation had class. All this to say, I am still using macros but trying to restrict their use. Within this project, macros provide the tracing facilities for the emitted grammar code, and the library's debug version. From experience with this library first written in C++, various refinements to the tracing output were needed. When one parses a large file having possibly hundreds of threaded grammars running dynamically within a session, if all tracing classifications are turned on, the traced session output can get rather large. Message tracing alone is very verbous but at least you have options to track down problems. This was very helpful when I relied on Microsoft's take on messaging. When threads become latent due to dropped messages (unexpressed limitation of the number of messages allowed in their Window queue), at least I could re-evaluate how I would roll my own. Well you'll see later how I re-implemented message queues with mutexes.

Now *cweb* provides various flavors of macros. Some macros use parametric substitution per c source line. A great feature of *cweb* is its code snippet insertion facility. The description of the code section provides a better reading of the code. One is not caught up with the details but the intent. I consider it a version of pseudo-programming in the real or is it a real coding in the pseudo? So the following is an re-engineering of Yacco2's library from C++ code to *cweb*.

4. Using Yacco2.
Where are those damn objects? Make sure your C++ compiler and linker are given the directions to where the # **include** `"yacco2.h"` file resides and Yacco2's appropriate object library. For example, the Yacco2 environment to use and link to are as follows:

/usr/local/yacco2/library - where the include file resides
/usr/local/yacco2/library/xxxx - where xxxx is debug or release for the object library

Within the "Visual studio C++" product, one can provide the appropriate directions within the project properties and preprocessor symbol definitions used to control code inclusion. One can also create an Environment variable in NT by going to the 'System panel', choosing 'System properties' followed by 'Advanced properties'. You can possiblely use 'Yacco2' and 'Yacco2lib' as the variable names: it's to your taste. The HP C++ product and linker can be expressed by command line parameters.

5. Overview of Yacco2's components:.
Still under thought construction — procastinating am i...?

6. Rules of the name.
There are not too many dictates. I try to give meaningful names to the components, be it methods, variables, or symbols. I lean a little too far in verbousity as in the Germanic description given to a symbol's name. Use of *cweb* will lower this trait. Cryptic names don't have a long life in their intent: future readings of the code usually requires a rebuilding of code comprehension. Typical coding comments are not enough. There are usually unspoken premises that trip up the programmer. This is why, for me, 'Literate programming' is the only way to go with its adjunct *mpost* diagrams (Meta Post). I say this in an asymtompic way as perfection is the carrot before the coder striving for a moment's perfection that is just a drop in the programming space. Too many programmers are stuck in the one dimension of code: 'just get it done' that becomes a debugging issue of learning that does not get reframed into documentation. Judge accordingly my attempt at the how,why,when,where,what,and whom are expressed. This is a quasi diary of my internal debats, mistakes, and evolutionary corrections in comprehension to programming Yacco2.

Rule number one: Use the imperative verb form to express a method name. For example to read or set a variable named xxx, the imperative actions can be *read_xxx* having no parameter, and *set_xxx* with it's appropriate parameter. From experience, overloading the method name by presence or absence of a parameter tempts error. I am more disciplined on the setting of variables due to past trapings. Regards to reading of a vaiable value, I'm more relaxed as you will see some variations.

You'll find for efficiency reasons, I access the variables directly instead of thru the wrapper function: yes I know the arguments of "OO" but inlining in my opinion got fumbled.

7. Legend of terms.
th - thread
pp - grammar requesting parallel parse
ar - arbitrator

8. The preprocessor coding game.
To cope with variations in source code, the C++ preprocessor's #if directives are used. The # **if**'s constant expression is used where appropriate values are tested using the # **if** / #elif preprocessor expressions. The *yacco2_compiler_symbols.h* file contains the 2 preprocessor symbols for compilation of O_2: `THREAD_LIBRARY_TO_USE__` — Pthreads(0) or Microsoft(1) thread library, and `THREAD_VS_PROC_CALL__` — run by thread(0) or by a procedure call(1). `THREAD_VS_PROC_CALL__` is an optimization attempt or a bailout when the platform being ported to has threading problems. Please see "Notes to myself" as to why it's been removed.

Initially the below symbols were used to control the inclusion of tracing code by the macro preprocessor. This really was a pain-in-the-???. As the number of options increased, how many O_2 library variations do u need? So now there are only 2 O_2 library flavours: clean-no-chafe tracing code and all-u-can-trace. To achieve this binary approach to O_2 libraries, **instead of conditionals**, **global tracing variables** are now used that are checked at runtime to exercise their tracing behaviors.

The run program that uses the O_2 library can use the *YACCO2_define_trace_variables* macro to generate the tracing variable definitions. U can still do it the hard way by individually coding each definition but why not use this short cut? So far these tracing global definitions take a binary value of 0 indicating do-not-trace while 1 means use it. There is a very slight run speed bump having their runtime presence within O_2's library and whether it's nobler to trace or not...but their benefits outweight their hiccups. One can turn on or off there use anywhere through one's code. Directory of variables:

`YACCO2_T__` — trace terminal when fetched
`YACCO2_TLEX__` — trace macros of emitted grammar: rules and user emergency macros
`YACCO2_MSG__` — trace thread messages
`YACCO2_MU_TRACING__` — trace acquire / release of trace mutex
`YACCO2_MU_TH_TBL__` — trace acquire / release mutex of thread table
`YACCO2_MU_GRAMMAR__` — trace acquire / release each grammar's mutex
`YACCO2_TH__` — trace the parse stack: fsa and syntax directed activities
`YACCO2_AR__` — trace arbitrator procedure
`YACCO2_THP__` — trace thread performance
`VMS__` — Alpha VMS port to correct their Pthread limitations
`VMS_PTHREAD_STACK_SIZE__` see bug's talk and *yacco2_compile_symbols.h*

They are enrobed by namespace yacco2. To set the trace variable be sure the namespace is delared: either explicitly as in:

yacco2 :: `YACCO2_T__` = 1;

or implicitly by a "using namespace yacco2;" statement somewhere preceding the assignment:

using namespace yacco2;
...
`YACCO2_T__` = 1;

9. Thread library use.
`THREAD_LIBRARY_TO_USE__` indicates what thread library to gen up. **It is a macro conditional symbol**. There are currently 2 libraries supported: Microsoft's thread support and the *Pthread* POSIX library. Both libraries have been used. The Pthread library of 32 and 64 bit flavours was tested on HP's VMS operating system — Alpha hardware, Apple's OS X PowerPC laptop, and Sun's Solaris Ultra M20 AMD 64 bit dual core work station. As `THREAD_LIBRARY_TO_USE__` is binary valued for now, the value 1 selects the Microsoft thread library while the value 0 selects the *Pthread* library.

10. Parsing trace variables.
To help in debugging a grammar, the following variables symbols are defined: `YACCO2_T__` , `YACCO2_TH__` , `YACCO2_TLEX__` , `YACCO2_MSG__` , `YACCO2_MU_GRAMMAR__` , and `YACCO2_AR__`. So far the tracing facilities fall into 3 catagories: trace the token when fetched, trace the message correspondence between threads, and trace the parsing stack of the grammar per action taken. Each symbol name tries by use of a suffix to indicate its functionality. For example, `_MSG__` suffix controls tracing of the messages between all threads and process. Specific arbitrator functor uses the `_AR__` suffix. These are workers supporting parallel parsing per grammar that require arbitration and thread control.

The symbols are all binary expressions where "1" (one) includes their functionality. As parallel parsing can use many threads, to refine the volume of traced output, macros that use these symbols `YACCO2_TLEX__` ,`YACCO2_TH__` , and `YACCO2_AR__` also test whether their associated grammar has the fsm's debug parameter value of 'true'. `YACCO2_TLEX__` symbol controls the specific tracings that are emitted by *Yacco2* in the C++ code per rule.

YACCO2_MU_xxx__ helps to verify that mutexes are properly acquired and released. There are 2 contexts that mutexes are used:

1) global mutexes — thread table and tracing
2) grammar mutex

To aid in identifying a grammar mutex, `(UN)LOCK_MUTEX_OF_CALLED_PARSER` external routines were created so that the grammar's context could be passed as a parameter. This allowed one to trace the grammar's name and assigned thread no. Why are `LOCK_MUTEX` and `UNLOCK_MUTEX` routines not sufficient? There are contexts where the parse context is too far down the chain of calls to pass the parser context or there is no parser context availible: eg, handle tracing by the grammar writer outside the parser context.

11. Thread performance.
To get a feel of why threads are a tad sluglish, the `YACCO2_THP__` conditional was invented. It allows one to see the serpentine meanderings of how the thread library works: flow control dodos.

When the environment is a single cpu, the flow control is how the cpu relinquishes control to the various threads. As cpus are added, this serpentine tracking becomes non-deterministic: That is, the traces are parallel or branched competing on the same race trace side-by-side where the number of lanes is the number of cpus actively running.

12. Section organization.
To control the output of various *cweb* code sections, the section names and their order are as follows:

⟨ Include files 14 ⟩,⟨ Type defs 16 ⟩,⟨ Structure defs 18 ⟩, and ⟨ External rtns and variables 22 ⟩. As include statements can take on different definitions: type, constant, structures, sometimes the dependency of the include file order is important particularly when the files are outside one's developmental control or there are circular references. For structures not defined yet but referenced, at the point of use, the standard C++ statement will be added infront of the to-be-defined structure. Maybe a bit imperfect but pratical. So this is my take...

13. C macros.
Conditionally defined macros for tracing. They are bracketed by the conditional preprocessor code controlling their inclusion.

⟨ c macros 13 ⟩ ≡ /∗ c macros ∗/

See also sections 630 and 631.

This code is used in section 35.

14. Include files.
To start things off, these are the Standard Template Library (STL) includes needed by Yacco2.

⟨ Include files 14 ⟩ ≡
⟨ iSTL 32 ⟩;

See also section 138.

This code is cited in section 12.

This code is used in section 35.

15. Global macro definitions. These are references throughout all Yacco2's *cweb* files. One definition to watch for is *SIZE_CAbs_lr1_sym*. It attempts to optimize the allocation of raw characters. Due to some of *CAbs_lr1_sym* items — the boolean and short ints, there are slack bytes generated when alignment for 64 bit support takes place for pointers on 8 byte boundries. `SIZE_RC_MALLOC` is used to eliminate dflt ctor of *CAbs_lr1_sym*.

```
#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 (#ffffffff) /* 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 /* 32 bit: 24..28 bytes, 64 bit: 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 /* rel 0 for now */
#define No_Token_start_pos Token_start_pos - 1 /* rel 0 for now */
#define CALLED_AS_THREAD true
#define CALLED_AS_PROC false
#define ACCEPT_FILTER true
#define BYPASS_FILTER false
```

16. Typedef definitions.
These are the basic types to aid in porting or maintaining the code. Other sections will add to this section as they get developed.

⟨ Type defs 16 ⟩ ≡
```
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];
```
See also sections 44, 124, 125, 139, 170, 316, 423, and 431.

This code is cited in section 12.

This code is used in section 35.

17. Recursion index for internal tracing of output.
Used to prefix spaces according to its count. Allows one to output messages to **lrclog** where the prefix number of spaces is the recursive call level.

```
#define Recursion_count() int RECURSION_INDEX__(0);
```

18. Structure definitions.

⟨ Structure defs 18 ⟩ ≡ /* structures */

See also sections 45, 51, 52, 53, 58, 78, 79, 80, 81, 82, 83, 104, 106, 107, 108, 112, 113, 114, 115, 117, 171, 184, 222, 429, 443, 444, 445, 446, 447, 448, 449, 526, 527, 528, 529, 530, 531, 532, and 533.

This code is cited in section 12.

This code is used in section 35.

19. Global external variables from yacco2's linker.

Apart from *PTR_LR1_eog__* which is defined by the *yacco2_k_symbols.lex* grammar, yacco2's linker generates the balance of these symbol definitions. All these symbols are covered by namespace yacco2. They are dangling references within this library that get resolved by the regular language linker from other objects when the program is built.

The first 5 symbols can only be defined by yacco2's linker due to the condition that all grammars and their threads must be known before these symbols can be defined specific to the developed language. Here we have a general piece of software that has dangling references of future knowns.

⟨ Global external variables from yacco2's linker 19 ⟩ ≡

```
    /* Global externals from yacco2's linker and yacco2_k_symbols.lex */
  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__;
```

This code is cited in section 109.

This code is used in section 35.

20. Global tracing variables.

See *The preprocessor coding game* for their meanings.

⟨ Global externals for yacco2 tracing variables 20 ⟩ ≡

```
  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__;
```

This code is used in section 35.

21. Global variables.

⟨ Global variables 21 ⟩ ≡ /* gbl variables */

See also sections 172, 424, 425, and 426.

This code is used in section 35.

22. External rtns.

⟨ External rtns and variables 22 ⟩ ≡ /* extern rtns + gbl variables */

See also sections 46, 140, 173, 211, 427, and 632.

This code is cited in section 12.

This code is used in section 35.

23. Using library's namespace yacco2. The acronyms should be obvious to the user within their context.

⟨ uns 23 ⟩ ≡
  using namespace yacco2;

This code is cited in section 666.

This code is used in sections 36, 76, 189, 193, 200, 203, and 209.

24. Begin namespace yacco2.

⟨ bns 24 ⟩ ≡
  namespace yacco2 {

This code is cited in section 666.

This code is used in section 35.

25. End namespace yacco2.

⟨ ens 25 ⟩ ≡
  } ;   /* end namespace yacco2 */

This code is cited in section 666.

This code is used in section 35.

26. Include Yacco2 header.

⟨ iyacco2 26 ⟩ ≡
```
#include "yacco2.h"
```

This code is used in sections 36, 42, 55, 76, 169, and 450.

27. Include Yacco2's raw characters header.

⟨ irc 27 ⟩ ≡
```
#include "yacco2_characters.h"
```

This code is used in sections 55 and 76.

28. Include Yacco2's constants header.

⟨ ilrk 28 ⟩ ≡
```
#include "yacco2_k_symbols.h"
```

This code is used in sections 55 and 76.

29. Include Yacco2's conditional compile control symbols header.

⟨ icompile??? 29 ⟩ ≡
```
#include "yacco2_compile_symbols.h"
```

This code is used in section 35.

30. Include Yacco2's arbitrator's begin code.

⟨ iar begin 30 ⟩ ≡
```
#include "war_begin_code.h"
```

This code is used in section 175.

31. Include Yacco2's arbitrator's end code.

⟨ iar end 31 ⟩ ≡
```
#include "war_end_code.h"
```

This code is used in section 175.

32. A wrapper file that brings in the required Standard Template Library (STL) containers used by Yacco2.

⟨ iSTL 32 ⟩ ≡
```
#include <stdlib.h>
#include <limits.h>
#include <assert.h>
#include "std_includes.h"
#include <time.h>
```
This code is used in section 14.

33. Accrue yacco2 code.

⟨ accrue yacco2 code 33 ⟩ ≡ /* accrue yacco2 code */

See also sections 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 72, 73, 74, 75, 120, 121, 122, 126, 130, 131, 133, 134, 135, 136, 230, 232, 233, 234, 236, 238, 240, 241, 243, 249, 265, 267, 269, 272, 279, 282, 283, 284, 285, 286, 288, 289, 297, 298, 300, 301, 302, 303, 305, 306, 307, 309, 310, 311, 312, 313, 315, 318, 319, 320, 322, 323, 324, 326, 327, 328, 330, 331, 332, 334, 336, 337, 338, 342, 343, 344, 345, 346, 347, 348, 350, 362, 364, 365, 366, 367, 368, 369, 371, 372, 375, 376, 385, 386, 393, 396, 399, 401, 402, 414, 418, 421, 422, 428, 430, 432, 433, and 636.

This code is used in section 36.

34. ***cweb* output of Yacco2's user library.**
The implementation code is emitted by *cweb*'s @c or @(operators throughout this discourse. Definitions etc are outputted to the common include file **yacco2** . *h*. All implementations will include this file into their implementation.

35. Create header file for Yacco2 library environment. Note, the "include search" directories for the C++ compiler has to be supplied.

⟨ `yacco2.h` 35 ⟩ ≡
  ⟨ copyright notice 565 ⟩;
#ifndef *yacco2_*
#define *yacco2_* 1
  ⟨ icompile??? 29 ⟩;
  ⟨ Preprocessor definitions ⟩
  ⟨ Include files 14 ⟩;
  ⟨ bns 24 ⟩;
  ⟨ Type defs 16 ⟩;
  ⟨ Global variables 21 ⟩;
  ⟨ Global externals for yacco2 tracing variables 20 ⟩;
  ⟨ Global external variables from yacco2's linker 19 ⟩;
  ⟨ Structure defs 18 ⟩;
  ⟨ External rtns and variables 22 ⟩;
  ⟨ ens 25 ⟩;
  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__*;
  };
  ⟨ c macros 13 ⟩;
#endif

36. Yacco2's library implementation.
Start the code output to **yacco2** . *cpp* by appending its include file.

⟨ `yacco2.cpp` 36 ⟩ ≡
  ⟨ copyright notice 565 ⟩;
  ⟨ iyacco2 26 ⟩;
  ⟨ uns 23 ⟩;
  ⟨ accrue yacco2 code 33 ⟩;

37. Constant definitions.
These are used by a hodge-podge of functionalities. The majority of the constants are enumerates: *LR1_Questionable_operator* to *LR1_Procedure_call_operator* are the lr constants. Some individual definitions below have comments relating their grammar's logical symbol. I did this as a memory jog to read the grammars. For example, to introduce parallelism into the grammar, the `|||` symbol is used. These constants allow one to efficiently test against an abstract symbol for its appropriate identity. Why test its identity?: to cast to a concrete object or to do conditional processing. Why not use C++ cast type operators — just too expensive in space and time! This is not a complaint but expressed from experience with Yacco2's environment — lots and lots of symbols and type cast operators lead to 'fat city'.

All grammar symbols (terminals and rules) have an emitted enumeration definition. The reason for these hardwired definitions is that they are also referenced within the Yacco2 compiler/compiler before I bootstrapped Yacco2 to compile its own grammars. These symbols will be explained when the code is developed. Possibly as I recast Yacco2 into *cweb*, these constants could be dropped for their generated look-alikes. Until then, they have earned their keep.

38. Enumerates. Events.

```
#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 /* |r| */
#define LR1_Invisible_shift_operator 5 /* |.| */
#define LR1_All_shift_operator 6 /* |+| */
#define LR1_FSET_transience_operator 7 /* |t| */
#define LR1_Procedure_call_operator 7 /* |t| */
```

39. The only reason for this section is to stop the appended slash to the last *cweb* macro above. This is a slight deviation in *cweb* code emission. Another abnormality is the use of the word "error" within C++ code: eg. enums. *cweb* has a slight clearing of the throat. So, just rename "error" to some other form: err ...

40. Enumeration of Alphabets — Terminals and Rules.

41. Enumeration.
The terminal alphabet is represented by the positive integers starting at zero. Lr constant terminals (meta terminals) are indicators of parsing situations like end-of-token stream reached, parallel parsing to take place, to different wild type shifts. None of these meta-terminals are found within the input language being parsed.

Raw characters represent the mapping from the 8 bit ASCII character into its raw character terminal. Both the meta and raw characters terminals are fixed and will never expand. They are therefore constant in their positions. Error terminals are internally generated situations produced by the parsing grammars manufactured by the grammar writer. They indicate the appropriate faulty situation detected and will grow in numbers as new error situations are developed. Regular terminals are composites that get created by the grammars from streams of other raw character terminals or composite terminals. They are evolutionary and come into existance from various passes made on the token streams: lexical to syntactic. Consequently, both errors and regular terminals are variable in their numbers as the grammar system is being developed.

To help speed up bottom-up parsing, the enumerate value of each terminal is computed to its compressed set key. This will be used in the various set operations like reduce, shift, and accept against the lookahead sets. The following *Set handling* section describes the details.

42. Set handling. This is an interesting section.
The original Yacco2 generated code to create each thread's tables at startup time by C++ templates. Well the 10 megabyte gorilla thumped its chest. By use of the marvelous book "Efficient C++" by Bulka and Mayhew, Yacco2 became a diet marvel. Have you heard any testimonials? No, well I'm now one. Go groan and sweat, your software will thank you for it and so will its life span.

As lookahead sets are rather sparse, to make set processing reasonably efficient, the following approach was implemented. The out-of-the-box binary search function is used to search a set. To minimize set size, the range of enumerated elements is divided up into 8 elements per partition where the remainder is the specific element.

Now why an 8 element partition? As Yacco2 currently uses 8 bit ASCII encoding and the density of the sets are sparse and my machine has 8 bits per byte, I felt that this was a reasonable compromise in the age of Aquarius. If the sets were more dense, then the number of elements per partition could be 16 or greater. As always, there is a compromise between space and speed. It's upto the person porting the software to decide. Hash tables were considered but I decided that space would be too wasteful.

Thought: Is there a dynamic hash faclity that rivals the set space but beats it in accessing speed? Other thoughts: use of complement sets if set size too big.

Elements are ordered in ascending sequence such that the set becomes a binary array of partitions. The binary functor takes two set structures: one is the key that is being searched within the set table, and the set table. To shrink the set size, *LR1_Eolr* is a special element used to signify 'use all terminals defined including self'. It's grammar tag is "eolr".

Output is directed to *wset.cpp*.

#define `SET_ELEM_NO_BITS` 8

⟨ `wset.cpp` 42 ⟩ ≡
  ⟨ copyright notice 565 ⟩;
  ⟨ iyacco2 26 ⟩;
  ⟨ accrue set code 43 ⟩;

43. Accrue set code.

⟨ accrue set code 43 ⟩ ≡ /* acrue set code */

See also section 47.

This code is used in section 42.

44. Some set types used in constructing search sets.

⟨ Type defs 16 ⟩ +≡
  typedef std ::**map** < **yacco2**::**USINT** , **yacco2**::**USINT** > *yacco2_set_type*;
  typedef yacco2_set_type ::**iterator** *yacco2_set_iter_type*;

45. Structure of a set.
Current implementation uses 2 bytes of 8 bit size. The first byte is the partition number with a range of 0..255. The 2nd byte is the elements where x in 2^x indicates its position within the byte. An element's position within the byte is its remainder of modulo `SET_ELEM_NO_BITS`. This set structure supports 2048 elements — 256 partitions by 8 elements. If there are more terminals to be supported, then there is 2 ways to increase the supported number of terminals: increase the partition size from a byte to an integer or expand the size of the number of elements per partition.

⟨Structure defs 18⟩ +≡
```
struct Set_entry {   /* set structure: byte no of set pairs, partition, set pair(s) */
  yacco2::UCHAR partition__;   /* whole no */
  yacco2::UCHAR elements__;   /* 7..0 in bit order due to remainder: 0 = 1 while 7 = 128 value */
};
struct Set_tbl {
  yacco2::UCHAR no_entries__;
  yacco2::Set_entry first_entry__[1];
};
```

46. Set element compare functor.
This is just your basic binary search functor whose address is passed to the binary search routine. The only interesting part is c's bitwise logical 'and' to determine if the element is in the 2nd byte of the structure. If the element is not found, it forces the search to continue down a cul-du-sac by returning a false 'less than' comparison.

Now i roll my own bsearch to speed things up. The compare functor is just too expensive in run time so out damn spot.

⟨External rtns and variables 22⟩ +≡
```
extern void create_set_entry(yacco2::USINT Enum_id, yacco2::Set_entry &Set);
```

47. From a terminal's enumeration create a set's key for searching.
This routine maps an enumeration into a set's co-ordinates.

⟨accrue set code 43⟩ +≡
```
extern void yacco2::create_set_entry(yacco2::USINT Enum_id, yacco2::Set_entry &Set)
{
  INT R = Enum_id % SET_ELEM_NO_BITS;
  Set.partition__ = Enum_id/SET_ELEM_NO_BITS;
  Set.elements__ = 1 << R;
}
```

48. *create_set_entry*.

⟨*create_set_entry* 48⟩ ≡
```
INT R = Enum_id % SET_ELEM_NO_BITS;
la_set.partition__ = Enum_id/SET_ELEM_NO_BITS;
la_set.elements__ = 1 << R;
```
This code is used in section 290.

49. *create_set_entry* **for** `RC`.

⟨*create_set_entry* **for** *Rc* 49⟩ ≡
```
INT R = sym→enumerated_id__ % SET_ELEM_NO_BITS;
sym→tok_co_ords__.set_entry__.partition__ = sym→enumerated_id__/SET_ELEM_NO_BITS;
sym→tok_co_ords__.set_entry__.elements__ = 1 << R;
```
This code is used in section 57.

50. *create_set_entry* **for CAbs_lr1_sym**.

⟨ *create_set_entry* **for CAbs_lr1_sym** 50 ⟩ ≡
INT $R = Enum\_id$ % `SET_ELEM_NO_BITS`;
$tok\_co\_ords\_\_.set\_entry\_\_.partition\_\_ = Enum\_id$ / `SET_ELEM_NO_BITS`;
$tok\_co\_ords\_\_.set\_entry\_\_.elements\_\_ = 1 \ll R$;

This code is used in section 60.

51. Table lookup functor. Inheritance earns its keep. See “Yacco2 - symbol table” document as an example of use.

⟨ Structure defs 18 ⟩ +≡

```
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 Toperator()(Tt) = 0;
      void turn_off_lkup()
      {
        lkup__ = OFF;    /* / yacco2::lrclog ¡¡ "TURN OFF TBLE LK" ¡¡ std::endl; */
      }
      ;
      void turn_on_lkup()
      {
        lkup__ = ON;    /* / yacco2::lrclog ¡¡ "TURN ON TBLE LK" ¡¡ std::endl; */
      }
      ;
      bool lkup()
      {
        return lkup__;
      }
      ;
      bool lkup__;
    }
    ;
```

52.

⟨ Structure defs 18 ⟩ +≡

```
typedef tble_lkup < yacco2::CAbs_lr1_sym * > tble_lkup_type;
```

53. Raw character mapper. Maps an 8 bit character into the raw character object. This is the raw character part of a grammar's terminal alphabet. To provide some performance, a static pool of objects is used instead of trashing malloc memory manager. Though it's a fixed size defined by `SIZE_RC_MALLOC` an overflow test at runtime throws an error if the memory pool is exhausted. All the raw character objects are of same size. Their differences comes in their genes: blue eyes, *id__*, *enumerated_id__*, and delete attributes. It is the same dog with the same spots of color being called by a different nickname. To improve startup performance where the array was being initialized to the default ctor layout that actually was useless, **CAbs_lr1_sym**'s default ctor has been eliminated. Now a raw character pool is used with casting to the newly minted **CAbs_lr1_sym**.

Output is directed to *wrc.cpp*.

```
⟨Structure defs 18⟩ +≡
  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::KCHARP chr_literal__[ASCII_8_BIT];
  };
```

54. Set up Raw characters malloc variables.

```
⟨accrue rc code 54⟩ ≡     /* acrue rc code */
  int yacco2::rc_map::current_rc_malloc_sub__(−1);
  char yacco2::rc_map::array_chr_sym__[SIZE_RC_MALLOC];
  yacco2::KCHARP yacco2::rc_map::chr_literal__[ASCII_8_BIT] = {"\x00","\x01","\x02","\x03",
      "\x04","\x05","\x06","\x07","\x08","\x09","\x0a","\x0b","\x0c","\x0d","\x0e","\x0f",
      "\x10","\x11","\x12","\x13","\x14","\x15","\x16","\x17","\x18","\x19","\x1a","\x1b",
      "\x1c","\x1d","\x1e","\x1f","\x20","\x21","\x22","\x23","\x24","\x25","\x26","\x27",
      "\x28","\x29","\x2a","\x2b","\x2c","\x2d","\x2e","\x2f","\x30","\x31","\x32","\x33",
      "\x34","\x35","\x36","\x37","\x38","\x39","\x3a","\x3b","\x3c","\x3d","\x3e","\x3f",
      "\x40","\x41","\x42","\x43","\x44","\x45","\x46","\x47","\x48","\x49","\x4a","\x4b",
      "\x4c","\x4d","\x4e","\x4f","\x50","\x51","\x52","\x53","\x54","\x55","\x56","\x57",
      "\x58","\x59","\x5a","\x5b","\x5c","\x5d","\x5e","\x5f","\x60","\x61","\x62","\x63",
      "\x64","\x65","\x66","\x67","\x68","\x69","\x6a","\x6b","\x6c","\x6d","\x6e","\x6f",
      "\x70","\x71","\x72","\x73","\x74","\x75","\x76","\x77","\x78","\x79","\x7a","\x7b",
      "\x7c","\x7d","\x7e","\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","\xf2","\xf3",
      "\xf4","\xf5","\xf6","\xf7","\xf8","\xf9","\xfa","\xfb","\xfc","\xfd","\xfe","\xff"};
```

See also section 56.

This code is used in section 55.

55. Output rc code.

```
⟨wrc.cpp 55⟩ ≡
  ⟨copyright notice 565⟩;
  ⟨iyacco2 26⟩;
  ⟨irc 27⟩;
  ⟨ilrk 28⟩;
  using namespace NS_yacco2_characters;
  ⟨accrue rc code 54⟩;
```

56. Map raw character to character symbol.
Place line detection by line feed. Call of this method requires the line number and character position. It determines the line boundary and augments their values.

```
⟨accrue rc code 54⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::rc_map::map_char_to_raw_char_sym
  (yacco2::UINT Char, yacco2::UINT File_no, yacco2::UINT Pos, UINT *Line_no, UINT
        *Pos_in_line)
  {
  map_char_to_symbol:
    ⟨Validate File no parameter 548⟩;
    if (Char ≥ rc_size_) {
      ⟨Error bad character mapping 562⟩;
      return 0;
    }
    ⟨Trace raw characters 646⟩;
    if (Char ≡ EOF_CHAR_SUB) {
      yacco2::PTR_LR1_eog__→tok_co_ords__.external_file_id__ = File_no;
      yacco2::PTR_LR1_eog__→tok_co_ords__.rc_pos__ = Pos;
      yacco2::PTR_LR1_eog__→set_line_no_and_pos_in_line(*Line_no, 1);
      return yacco2::PTR_LR1_eog__;
    }
    ⟨malloc raw characters from static pool instead of newing 57⟩;
    ++(*Pos_in_line);
    sym→set_line_no_and_pos_in_line(*Line_no, *Pos_in_line);
    if (Char ≡ LINE_FEED) {    /* set for next char */
      ++(*Line_no);
      *Pos_in_line = START_CHAR_POS;
    }
    return sym;
  }
```

57. Malloc raw characters from static pool instead of newing of Malloc.
Note: the raw character pool used to eliminate the default **CAbs_lr1_sym** ctor initialization of the array at start up time. Now it's just a raw cess pool waiting to evolve.

⟨ malloc raw characters from static pool instead of newing 57 ⟩ ≡
  ++**rc_map** :: *current_rc_malloc_sub__*;
  long *rc_sub* = *current_rc_malloc_sub__* ∗ *SIZE_CAbs_lr1_sym*;
  if (*rc_sub* > SIZE_RC_MALLOC) {
    ⟨ Error no more raw character storage 563 ⟩;
  }
  CAbs_lr1_sym ∗*sym* = (**CAbs_lr1_sym** ∗) &**rc_map** :: *array_chr_sym__*[*rc_sub*];
  sym→*id__* = **rc_map** :: *chr_literal__*[*Char*];
  sym→*enumerated_id__* = *Char* + START_OF_RC_ENUM;
  sym→*tok_co_ords__*.*external_file_id__* = *File_no*;
  sym→*tok_co_ords__*.*rc_pos__* = *Pos*;
  ⟨ *create_set_entry* **for** *Rc* 49 ⟩;

This code is used in section 56.

58. Abstract symbol class for all alphabets.
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 used in the grammar 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 dupples where each dupple is composed of a partition no 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 membership within a set.

⟨Structure defs 18⟩ +≡
```
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__;
    };
  };
```

59. Grammar abstract symbol implementation.
Why the 3 **CAbs_lr1_sym** constructors? The 1st **CAbs_lr1_sym** defines rules, the 2nd defines the terminals without the GPS, while the 3rd can be used by the grammar writer in the syntax directed code to create terminals having a GPS to its source file.

60. CAbs_lr1_sym constructor.

⟨ accrue yacco2 code 33 ⟩ +≡

yacco2 :: **CAbs_lr1_sym** :: **CAbs_lr1_sym**(**yacco2** :: **KCHARP** *Id*, **yacco2** :: `FN_DTOR` *Dtor*, **yacco2** :: **USINT** *Enum_id*, **yacco2** :: **Parser** ∗*P*, **bool** *Auto_delete*, **bool** *Affected_by_abort*)
: *id__*(*Id*), *dtor__*(*Dtor*), *enumerated_id__*(*Enum_id*), *auto_delete__*(*Auto_delete*), *affected_by_abort__*(*Affected_by_abort*) {
  rule_info__.parser__ = *P*;
  ⟨ *create_set_entry* **for CAbs_lr1_sym** 50 ⟩;
}

yacco2 :: **CAbs_lr1_sym** :: **CAbs_lr1_sym**(**yacco2** :: **KCHARP** *Id*, **yacco2** :: `FN_DTOR` *Dtor*, **yacco2** :: **USINT** *Enum_id*, **bool** *Auto_delete*, **bool** *Affected_by_abort*)
: *id__*(*Id*), *dtor__*(*Dtor*), *enumerated_id__*(*Enum_id*), *auto_delete__*(*Auto_delete*), *affected_by_abort__*(*Affected_by_abort*) {
  tok_co_ords__.rc_pos__ = 0;
  tok_co_ords__.line_no__ = 0;
  tok_co_ords__.external_file_id__ = 0;
  tok_co_ords__.pos_in_line__ = 0;
  tok_co_ords__.who_file__ = 0;
  tok_co_ords__.who_line_no__ = 0;
  ⟨ *create_set_entry* **for CAbs_lr1_sym** 50 ⟩;
}

yacco2 :: **CAbs_lr1_sym** :: **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*)
: *id__*(*Id*), *dtor__*(*Dtor*), *enumerated_id__*(*Enum_id*), *auto_delete__*(*Auto_delete*), *affected_by_abort__*(*Affected_by_abort*) {
  tok_co_ords__.rc_pos__ = *Rc_pos*;
  tok_co_ords__.line_no__ = 0;
  tok_co_ords__.external_file_id__ = *Ext_file_no*;
  tok_co_ords__.pos_in_line__ = 0;
  tok_co_ords__.who_file__ = 0;
  tok_co_ords__.who_line_no__ = 0;
  ⟨ *create_set_entry* **for CAbs_lr1_sym** 50 ⟩;
}

61. *enum_id_set_partition_no* **and** *enum_id_set_member***.**
A compressed set key.

⟨ accrue yacco2 code 33 ⟩ +≡

yacco2 :: **UCHAR** **yacco2** :: **CAbs_lr1_sym** :: *enum_id_set_partition_no*( ) **const**
{
  return *tok_co_ords__.set_entry__.partition__*;
}

yacco2 :: **UCHAR** **yacco2** :: **CAbs_lr1_sym** :: *enum_id_set_member*( ) **const**
{
  return *tok_co_ords__.set_entry__.elements__*;
}

62. *rhs_no_of_parms*. Number of elements contained in a rule's right hand side subrule.

⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2 :: **USINT** **yacco2** :: **CAbs_lr1_sym** :: *rhs_no_of_parms* ( )
  {
    return *rule_info__.rhs_no_of_parms__*;
  }

63. *parser*. Associated parser with the grammar being used.
A terminal symbol has no association with a parser apart from where it was constructed¿ Where as a rule does require this reference that gets assigned at construction time. So be ware as the parser variable is unionized!

⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2 :: **Parser** ∗**yacco2** :: **CAbs_lr1_sym** :: *parser* ( )
  {
    return *rule_info__.parser__*;
  }

64. *id*. Descriptive form of the symbol for tracing purposes.
For rules, this is optimized out when the grammar's debug switch is set to off. You must regenerate the grammar when you want to turn on the grammar's debug facilty. Just setting the C++ code for debug is not sufficient. Trust me.

⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2 :: **KCHARP** **yacco2** :: **CAbs_lr1_sym** :: *id* ( ) **const**
  {
    return *id__*;
  }

65. *enumerated_id*.
The iteration scheme for the terminal alphabet starts at 0 followed by the grammar's rules. Subrules enumeration start from 1. Their enumerates are mutually exclusive and are defined in the generated fsm class of the grammar.

⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2 :: **USINT** **yacco2** :: **CAbs_lr1_sym** :: *enumerated_id* ( ) **const**
  {
    return *enumerated_id__*;
  }

66. *set_enumerated_id*.

⟨ accrue yacco2 code 33 ⟩ +≡
  void **yacco2** :: **CAbs_lr1_sym** :: *set_enumerated_id* (**yacco2** :: **USINT** *Id*)
  {
    enumerated_id__ = *Id*;
  }

67. *set_affected_by_abort* **and** *affected_by_abort***.**
These are the writer and reader of the grammar's auto abort attribute 'AB' for the symbol.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::CAbs_lr1_sym::set_affected_by_abort(bool X)
{
  affected_by_abort__ = X;
}
bool yacco2::CAbs_lr1_sym::affected_by_abort() const
{
  return affected_by_abort__;
}
```

68. *set_auto_delete* **and CAbs_lr1_sym ::** *auto_delete***.**
These are the writer and reader of the grammar's auto delete attribute 'AD' for the symbol.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::CAbs_lr1_sym::set_auto_delete(bool X)
{
  auto_delete__ = X;
}
bool yacco2::CAbs_lr1_sym::auto_delete() const
{
  return auto_delete__;
}
```

69. *dtor***.**
Destructor function defined by the grammar writer for the symbol. Why not use the class genetics? A class is too expensive in its implementation. Your basic structure is sufficient with no virtual table overhead. Within this context, the dtor is rarely needed and it's upto Yacco2 to create when needed. See the *destructor* directive of the grammar.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::FN_DTORyacco2::CAbs_lr1_sym::dtor()
{
  return dtor__;
}
```

70. *set_rc*, *set_rc_pos*, **and** *rc_pos*.
These are the writers and reader of the terminal's co-ordinate. The only symbol that directly sets these values are the raw character symbols. All other symbols are composites built from raw character terminals. The co-ordinate parts can be individually set, or all parts of the co-ordinate can be copied from a previous symbol's co-ordinate. Normally their use comes from a parsing environment producing tokens built from a grammar but this is not a hardfast rule.

The reason why the parser address is passed to **CAbs_lr1_sym** :: *set_rc* is due to *eog*. It is shared across all token containers and all copied source files. This sharing behavior was taken to lower the new-delete overhead to creating of the terminal. Consequently there is no definite co-ordinate associated with this terminal and one must go to the previous token of the supplier to tack on the real co-ordinates + the number of previous terminals tried for a co-ordinate. The supplier context comes from the *parser__*.

The 2 GPS parameters allows parental histronics: Don't know if this is received well by the user of O_2 but it certainly helps to debug. This was added down the road and so the reason for the defaults in the prototype as to not disturb existing grammars. If the default is taken then the GPS is not set as it could be done elsewhere. *set_who_created* allows one to initially set or override previous settings.

Some marginal additives: parse stack co-ordinates for error tokens and "eog" association with from current token supplier. Added the situation if no token symbol to find for the "eog" token (no data entered at the command line), i force the command line co-ordinates instead of throwing up.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::CAbs_lr1_sym::set_rc(yacco2::CAbs_lr1_sym &Rc, yacco2::KCHARP
        GPS_FILE, yacco2::UINT GPS_LINE)
{
  if (GPS_FILE ≠ 0) {
    tok_co_ords__.who_file__ = GPS_FILE;
    tok_co_ords__.who_line_no__ = GPS_LINE;
  }
  if (Rc.tok_co_ords__.external_file_id__ > 0) {
    tok_co_ords__.external_file_id__ = Rc.tok_co_ords__.external_file_id__;
    tok_co_ords__.rc_pos__ = Rc.tok_co_ords__.rc_pos__;
    tok_co_ords__.line_no__ = Rc.tok_co_ords__.line_no__;
    tok_co_ords__.pos_in_line__ = Rc.tok_co_ords__.pos_in_line__;
    return;
  }
  return;
}
```

71. Does terminal have a legitimate co-ordinate?.
Do you see the moonwalk? This goes backwards through the supplier tokens looking for a source address. Inside the supplier routine is the validation on the requested subscript.

⟨ does terminal have a legitimate co-ordinate? yes set it and exit. no keep trying 71 ⟩ ≡

```
if (pt→tok_co_ords__.rc_pos__ ≠ 0) goto set_co_ordinates;
++bk_cnt;
−−prev_pos;
goto find_legitimate_terminal;
```

72.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::CAbs_lr1_sym::set_rc_pos(yacco2::UINT Pos)
{
  ⟨ Validate Pos parameter 546 ⟩;
  tok_co_ords__.rc_pos__ = Pos;
}
yacco2::UINT yacco2::CAbs_lr1_sym::rc_pos()
{
  return tok_co_ords__.rc_pos__;
}
```

73. *set_external_file_id* **and** *external_file_id*.

These are the writer and reader of the grammar's external file index used to reference the copied files descriptive name.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::CAbs_lr1_sym::set_external_file_id(yacco2::UINT File_no)
{
  ⟨ Validate File no parameter 548 ⟩;
  tok_co_ords__.external_file_id__ = File_no;
}
yacco2::UINT yacco2::CAbs_lr1_sym::external_file_id()
{
  return tok_co_ords__.external_file_id__;
}
```

74. Set line no, and character position routines.
These are the writer and reader to parts of the co-ordinate.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::CAbs_lr1_sym::set_line_no(yacco2::UINT Line_no)
{
  ⟨ Validate Line no parameter 545 ⟩;
  tok_co_ords__.line_no__ = Line_no;
}
yacco2::UINT yacco2::CAbs_lr1_sym::line_no()
{
  return tok_co_ords__.line_no__;
}
yacco2::UINT yacco2::CAbs_lr1_sym::pos_in_line()
{
  return tok_co_ords__.pos_in_line__;
}
void yacco2::CAbs_lr1_sym::set_pos_in_line(yacco2::UINT Pos_in_line)
{
  ⟨ Validate Pos in line parameter 547 ⟩;
  tok_co_ords__.pos_in_line__ = Pos_in_line;
}
void yacco2::CAbs_lr1_sym::set_line_no_and_pos_in_line(yacco2::UINT Line_no, yacco2::UINT
      Pos_in_line)
{
  ⟨ Validate Line no parameter 545 ⟩;
  ⟨ Validate Pos in line parameter 547 ⟩;
  tok_co_ords__.line_no__ = Line_no;
  tok_co_ords__.pos_in_line__ = Pos_in_line;
}
void yacco2::CAbs_lr1_sym::set_line_no_and_pos_in_line(yacco2::CAbs_lr1_sym &Rc)
{
  tok_co_ords__.line_no__ = Rc.tok_co_ords__.line_no__;
  tok_co_ords__.pos_in_line__ = Rc.tok_co_ords__.pos_in_line__;
}
```

75. *set_who_created*, *who_line_no*, *who_file*.
These are the writer and reader to parts of the co-ordinate giving the source that created the symbol.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::CAbs_lr1_sym::set_who_created(yacco2::KCHARP File, yacco2::UINT Line_no)
{
  tok_co_ords__.who_file__ = File;
  tok_co_ords__.who_line_no__ = Line_no;
}
yacco2::UINT yacco2::CAbs_lr1_sym::who_line_no()
{
  return tok_co_ords__.who_line_no__;
}
yacco2::KCHARP yacco2::CAbs_lr1_sym::who_file()
{
  return tok_co_ords__.who_file__;
}
```

76. Token container structure, templates, and functions.
The 2 specialized containers *tok_can* < `AST` ∗> for tree walks and *tok_can* < *ifstream* ∗> for raw character fetching have been optimized to eliminate the "jit" fetching of token for speed reasons: elimination of read mutex. See "Notes to myself" on discussion. This leaves the *tok_can* < *string* > as unsafe. It is used internally by the library to GPS tokens against their opened files to line / character position. Sooooo, Be Ware the ....

⟨`wtok_can.cpp` 76⟩ ≡
  ⟨copyright notice 565⟩;
  ⟨iyacco2 26⟩;
  ⟨irc 27⟩;
  ⟨ilrk 28⟩;
  using namespace NS_yacco2_characters;
  ⟨uns 23⟩;
  ⟨accrue *tok_can* code 77⟩;

77.
⟨accrue *tok_can* code 77⟩ ≡ /∗ accrued *tok_can* code ∗/
See also sections 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, and 103.
This code is used in section 76.

78. Sour Apple on template definition.
Circa December 2005, Apple C++ gcc 4.0 compiler honks on preprocessing the *Tok_can* template definition. The template has not been instantiated but its prototype definition preprocessed into a holding source macro for future code substitution and compiling — AKA instantiation. Unfortunately gcc 4.0 expects all prototype variables declared before preprocessing the template prototype takes place. For example variables `LOCK_MUTEX`, `UNLOCK_MUTEX`, *PTR_LR1_eog__*, and `YACCO2_T__` in template *Tok_can* below aggravates the compiler and gave me a headache. All other C++ compilers tried like Intel C++ 9.0, HP C++ 6.x and 7.1 for VMS Alpha, and Microsoft's Visual Studio c++ 7.0 and 2005 all work. Alas portability is extremely trying. Am i being bruised by software savants? The work around is declare these items before the template definition. See *Notes to myself* to their response and correct position to my perceived problem.

⟨Structure defs 18⟩ +≡
```
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);
```

79. *Tok_can* **template.**
tok_base forces regularity across the *tok_can* containers. *wtok_can.cpp* for *tok_can* containers of ifstream, string, and tree.

⟨ Structure defs 18 ⟩ +≡
```
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) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog << "YACCO2_T__::tok_can␣token␣eog:␣"
         << PTR_LR1_eog__ << "␣pos:␣" << Pos << FILE_LINE << std::endl;
        ⟨ release trace mu 390 ⟩;
      }
      return PTR_LR1_eog__;
    }
    CAbs_lr1_sym *tok_(0);
    if (r_w_cnt__ > 1) {
      ⟨ acquire token mu 391 ⟩tok_ = container__[Pos];
      ⟨ release token mu 392 ⟩
    }
```

```
  else {
    tok_ = container__[Pos];
  }
  if (YACCO2_T__ ≠ 0) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_T__::tok_can␣token:␣" ≪ tok_→id__
     ≪ "␣*:␣" ≪ tok_ ≪ "␣pos:␣" ≪ Pos
     ≪ "␣enum:␣" ≪ tok_→enumerated_id__ ≪ '"' ≪ tok_→id__ ≪ '"' ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
    EXTERNAL_GPSing(tok_)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪ tok_→tok_co_ords__.line_no__ ≪
        "␣GPS␣CHR␣POS:␣" ≪ tok_→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
  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__;
};
```

80. Specialized tok_can containers: ifstream and string.
They read character streams from external file or string contexts. The string container's contents can be added to dynamically (concatenated) using *set_string* procedure while parsing is taking place. The caveat is it must be before the end-of-string condition has been met. If a GPS token is passed to it at ctor creation time, the errors reported will be relative to the GPSed file. It tries hard to keep these co-ordinates relative to the spawning token who supplied the string: string new lines are not respected as this would throw off the error reporting relative the external file. *reuse_string* allows one to keep a global string token container and to reuse it.

⟨Structure defs 18⟩ +≡

```
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__;
};
```

81. Tree container and its related paraphernalia.
There's the functor for the tree walker that includes the stack, a user functor that executes when the node is visited, a set filter mechanism to include or exclude node types, and the tree node itself.

Filters are just sets of Tes enumerated ids of T vocabulary. All T types lr, rc, error, and T are allowed. A filter type of bypass or accept makes walking the trees easier in selecting T. A nil based filter implies all Tes are accepted.

The tree walkers supported are pre and post fix, and various flavours of breadth walks. A forest walk refines the scope of the tree to be walked even though the forest node can be linked to the tree.

Due to the nature of a binary tree, the infix tree walker is not supported. The tree structure is provided by the `AST` definition which is just a tree node wrapper for the grammar's vocabulary. Its content is abstracted to **CAbs_lr1_sym** as it has no psychic powers of the future grammar user. In tandom with the enumeration value of the abstracted symbol, the casting operator brings its out of the closet so-to-speak. Dominance is provided by the *lt_* link while *rt_* provides the equivalence link. To aid in walking the tree, the *pr_* link provides the backward link to its immediate caller. This link can be its older sibling, parent when its the first child, to nil when the node is the root of the tree.

⟨ Structure defs 18 ⟩ +≡

```
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_;    /* index */
  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;
};
```

82. Tree node definition AST.
Note on linkages:

1) lt parent to son relationship: dominant order
2) rt older to younger relationship: equivalence order
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.

⟨Structure defs 18⟩ +≡

```
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_;    /* caller who links to it */
yacco2::CAbs_lr1_sym *obj_;
bool wdelete_;
};
```

83. Tree tok_can⟨AST ∗⟩ container with accept / bypass *filters*.
The interesting part is use of the **int_set_type** filter and its companion *Accept_opt* in the constructor of the *tree_walker*. The **int_set_type** filter just contains the Terminal enumerations to either accept or bypass. If these parameters are defaulted, there is no **int_set_type** filter present so the complete tree is handed off for consumption of each node's content. Having *Accept_opt* **true** means accept only the items in the set while **false** means bypass the items found in the filter set when the tree is walked. This is a very powerful way to flatten a branching structure.

Please note *nodes_visited_* holds the terminals accepted by the filter in the traversal order. It is an array of **AST** ∗. To access a token's tree node, u need the container address. If a grammar is receiving its terminals by a walked tree, casting the container address to **tok_can**⟨**yacco2** :: **AST** ∗⟩ ∗ allows one to access the container's tree node vector: *nodes_visited*( ). The below code fetches the container's address from a piece of syntax directed code of a grammar's rule:

tok_can⟨**AST** ∗⟩ ∗*can* = (**tok_can**⟨**AST** ∗⟩ ∗) *parser*( )→*token_supplier*( );

To fetch a specific tree node of a token, u can use the container's ast function giving it the position within the container: Remember its relative to 0. For example u want to fetch the tree node associated with the 1st token using the above container:

AST ∗*first_tok_tree* = *can*→*ast*(0);

The other note is a shifted token on the parse stack is **not the current token**. Why? The current token is the lookahead token and the one u want is on parse stack! Here is a sample code snippet to get the shifted token's tree address using the above container with another way to fetch its tree:

AST ∗*t* = (∗*can*→*nodes_visited*( ))[*parser*( )→*current_token_pos*( ) − 1];

Why use parser's *current_token_pos*( ) instead of the container's *pos*( )? Good question: they are equivalent except when one is reusing the container to deliver tokens to another grammar. The recycled container's pos **contains the residue from the previous reads: its last token position**. Ugh but this is reality. The sundry tree routines can now be used to walk or fetch the contents of the local tree node.

Caveat: EOG Handling.
Make sure u add an *eog* node to the end of the tree so that proper end-of-tree handling is done. U do this by:

AST ∗*eog_t* = **new AST**(∗**yacco2** :: *PTR_LR1_eog__*);
then add the node to the end-of-the-tree using one of the tree linking routines

If it is not added, an *eog* token is returned but there is no associated tree node. So the last token read is not the lookahead but the previous (shifted) token. If u are using an accept filter, make sure the *eog* is included in the accept set so that *eog* gets its associated end-of-tree node. Please see "Tree containers, functors, and walkers" later in this document for their descriptions.

Another way to access the container and its contents.
Set up a filter and "for loop" the container to fill it up while the body of the for loop can done specific activity. This method can be done outside of the parsing activity or within "syntax directed code" of a grammar. Just give the tree and rip thru it using the filter.

```
// file: /yacco2/diagrams+etc/tokcanaccess.txt
  using namespace NS_yacco2_T_enum;
  using namespace NS_yacco2_terminals;
  using namespace yacco2;
  INT_SET_type filter;
  filter.insert(T_Enum::T_T_cweb_comment_);
  tok_can_ast_functor walk_functr;
  ast_prefix_1forest rule_walk(*tree_ptr,&walk_functr,&filter,ACCEPT_FILTER);
  tok_can<AST*> comments_can(rule_walk);// container
  for(int x(0);comments_can[x] != yacco2::PTR_LR1_eog__;++x){
```

```
      T_cweb_comment* k = (T_cweb_comment*)comments_can[x];
      (*Wfile) << k->comment_data()->c_str() << endl;
    }

```

⟨ Structure defs 18 ⟩ +≡

```
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_;
};
```

84. String tok_can⟨std :: string⟩ implementation.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
yacco2::tok_can<std::string>::tok_can()
: tok_base(1), pos_(0), have_1st_rec__(false), file_no__(MAX_USINT),
    line_no__(START_LINE_NO), pos_in_line__(START_CHAR_POS), string__(std::string()), eof_sym_(0),
    real_start_pos_in_line_(START_CHAR_POS), eof_pos_(0), gps__(0) { }
yacco2::tok_can<std::string>::tok_can(const char *String, CAbs_lr1_sym *GPS)
: tok_base(1), pos_(0), have_1st_rec__(false), file_no__(MAX_USINT),
    line_no__(START_LINE_NO), pos_in_line__(START_CHAR_POS), string__(String), eof_sym_(0),
    real_start_pos_in_line_(START_CHAR_POS), eof_pos_(0), gps__(GPS) {
  if (GPS == 0) return;
  line_no__ = GPS->tok_co_ords__.line_no__;
  pos_in_line__ = GPS->tok_co_ords__.pos_in_line__;
  file_no__ = GPS->tok_co_ords__.external_file_id__;
  real_start_pos_in_line_ = pos_in_line__;
}
void yacco2::tok_can<std::string>::set_gps(CAbs_lr1_sym *GPS)
{
  gps__ = GPS;
  if (GPS == 0) return;
  line_no__ = GPS->tok_co_ords__.line_no__;
  pos_in_line__ = GPS->tok_co_ords__.pos_in_line__;
  file_no__ = GPS->tok_co_ords__.external_file_id__;
  real_start_pos_in_line_ = pos_in_line__;
}
yacco2::CAbs_lr1_sym *yacco2::tok_can<std::string>::gps_used()
{
  return gps__;
}
yacco2::tok_can<std::string>::~tok_can()
{ }
bool yacco2::tok_can<std::string>::empty()
{
  if (string__.empty() == true) return YES;
  return NO;
}
void yacco2::tok_can<std::string>::reuse_string(const char *Str, CAbs_lr1_sym *GPS)
{
  string__.erase();
  string__ += Str;
  file_no__ = MAX_USINT;
  line_no__ = START_LINE_NO;
  pos_in_line__ = START_CHAR_POS;
  eof_sym_ = 0;
  real_start_pos_in_line_ = START_CHAR_POS;
  eof_pos_ = 0;
  if (GPS == 0) return;
  set_gps(GPS);
}
```

85. *Tok_can < string* , **std::vector > operator[].**
This is the heart of the container. Three things are of interest: the just-in-time character access, the 2 "eog" token symbols added to the end-of-file condition, and how to report errors inside the string relative to the file that provided the string: its contents cannot increment new line with character alignment. Why? When u report an error back to the original file containing the string, it is GPSed to it and not its contents. The string's line number stays the same while the line position increments towards the right without regard to the new line character. This allows the container to be handled like its brethern within the grammar context. Note: *map_char_to_raw_char_sym* maintains the line:character segmentation as the string is being read and so must be re-aligned afterwards. The file no reference to the outside source is hardwired using the MAX_USINT symbol when there is possibly no outside file referenced: eg, internal memory string for the parsing. A bit of a kludge (ahum) as this condition goes against the 0..n declaration for external file numbers. This is watched for when the external file out-of-bounds occurs: reported is "No external file".

```
⟨ accrue tok_can code 77 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::tok_can⟨std::string⟩::operator[](yacco2::UINT Pos)
  {
    CAbs_lr1_sym *sym(0);
    if (eof_pos_ ≡ EOF) return eof_sym_;
  fetch_char:
    if (have_1st_rec__ ≡ false) {
      have_1st_rec__ = true;
      pos_ = 0;
    }
    else {
      if (Pos ≤ pos_) {
        return container__[Pos];
      }
      ++pos_;
    }
    if (r_w_cnt__ > 1) {⟨ acquire token mu 391 ⟩}
    for ( ; ; ) {    /* fetch token somewhere in char stream */
      char c;
      if (pos_ ≥ string__.size()) {    /* eof: add two lrk eog */
        eof_pos_ = EOF;
        ++pos_;    /* 2nd eog pos, same token used */
        sym = RC__.map_char_to_raw_char_sym(EOF_CHAR_SUB, file_no__, pos_, &line_no__, &pos_in_line__);
        eof_sym_ = sym;
        container__.push_back(*sym);
        container__.push_back(*sym);
        return sym;
      }
      c = string__[pos_];
    convert_char_to_unsigned_value:
      unsigned char uc = c;
      UINT slno = line_no__;
      sym = RC__.map_char_to_raw_char_sym(uc, file_no__, pos_, &line_no__, &pos_in_line__);
      if (gps__ ≠ 0) {    /* re-align against the proxy token */
        line_no__ = slno;
        pos_in_line__ = real_start_pos_in_line_ + pos_;
      }
      container__.push_back(*sym);
      if (Pos ≡ pos_) break;
```

```
      ++pos_;
      continue;
    }
    ;
    if (r_w_cnt__ > 1) {⟨release token mu 392⟩}
    return sym;
  }
```

86. *Tok_can < string >***size.**

```
⟨accrue tok_can code 77⟩ +≡
  yacco2::UINT yacco2::tok_can⟨std::string⟩::size( )
  {
    return string__.size( );
  }
```

87. Balance of sundry routines.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
yacco2::UINT yacco2::tok_can<std::string>::pos()
{
  return pos_;
}
void yacco2::tok_can<std::string>::push_back(yacco2::CAbs_lr1_sym &Tok)
{
  container__.push_back(Tok);
}
void yacco2::tok_can<std::string>::clear()
{
  container__.clear();
  pos_ = 0;
  have_1st_rec__ = false;
  file_no__ = MAX_USINT;
  line_no__ = START_LINE_NO;
  pos_in_line__ = START_CHAR_POS;
  string__.clear();
  eof_sym_ = 0;
  real_start_pos_in_line_ = START_CHAR_POS;
  eof_pos_ = 0;
  gps__ = 0;
}
TOKEN_GAGGLE &yacco2::tok_can<std::string>::container()
{
  return container__;
}
void tok_can<std::string>::remove()
{ }
void yacco2::tok_can<std::string>::set_string(const char *String)
{
  string__ += String;
}
std::string *yacco2::tok_can<std::string>::string_used()
{
  return &string__;
}
;
```

88. External file tok_can⟨std :: *ifstream*⟩ implementation.
Removed the "jit" approach and now at *open_file* time the complete input is placed into the container. See "Notes to myself" on its discussion.

⟨ accrue **tok_can** code 77 ⟩ +≡
```
yacco2::tok_can<std::ifstream>::tok_can()
: tok_base(1), pos_(0), have_1st_rec__(false), eof_pos_(EOF), file_ok_(NO), line_no__(START_LINE_NO),
    pos_in_line__(START_CHAR_POS), file_name__(std::string()) { }
yacco2::tok_can<std::ifstream>::tok_can(const char *File_name)
: tok_base(1), pos_(0), have_1st_rec__(false), eof_pos_(EOF), file_ok_(NO), line_no__(START_LINE_NO),
      pos_in_line__(START_CHAR_POS), file_name__(File_name) {
  open_file();
}
yacco2::tok_can<std::ifstream>::~tok_can()
{
  if (file_ok_ ≡ YES) file__.close();
}
bool yacco2::tok_can<std::ifstream>::empty()
{
  if (have_1st_rec__ ≡ false) return YES;
  return NO;
}
```

89. *File_ok*.
By testing after the ctor has tried to open the file, one can do whatever is appropriate in a bad file situation. Originally a bad file condition was thrown. Now it's more gentle.

⟨ accrue **tok_can** code 77 ⟩ +≡
```
bool yacco2::tok_can<std::ifstream>::file_ok()
{
  return file_ok_;
}
```

90. *Tok_can < ifstream* , **std::vector > operator[].**
This is the heart of the container. Two things are of interest: the just-in-time character access, and the 2 "eog" token symbols added to the end-of-file condition. This allows the container to be handled like its brethern within the grammar context. Note: *map_char_to_raw_char_sym* maintains the line:character segmentation as the file is being read.

```
⟨ accrue tok_can code 77 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::tok_can⟨std::ifstream⟩::operator[](yacco2::UINT Pos)
  {
    if (file_ok_ ≡ NO) {
      char a[BUFFER_SIZE];
      yacco2::KCHARP msg = "tok_can<ifstream>operator[]␣trying␣to␣access␣file␣that␣is␣b\
          ad:␣%s,␣position␣%i␣";
      sprintf(a, msg, file_name__.c_str(), Pos);
      Yacco2_faulty_precondition(a, __FILE__, __LINE__);
      exit(1);
    }
    CAbs_lr1_sym *sym(0);
    if (eof_pos_ ≡ EOF ∧ Pos ≥ pos_) {
      return container__[pos_];
    }
  fetch_char:
    if (have_1st_rec__ ≡ false) {
      have_1st_rec__ = true;
      pos_ = 0;
    }
    else {
      if (Pos ≤ pos_) {
        return container__[Pos];
      }
      ++pos_;
    }
    if (r_w_cnt__ > 1) {⟨ acquire token mu 391 ⟩}
    for ( ; ; ) {    /* fetch token somewhere in char stream */
      char c;
      file__ ≫ c;
      if ((file__.good() ≡ false) ∨ (file__.eof() ≡ true)) {    /* eof: add two lrk eog */
        eof_pos_ = EOF;
        ++pos_;    /* 2nd eog pos, same token used */
        sym = RC__.map_char_to_raw_char_sym(EOF_CHAR_SUB, file_no__, pos_, &line_no__, &pos_in_line__);
        container__.push_back(*sym);
        container__.push_back(*sym);
        return sym;
      }
    convert_char_to_unsigned_value:
      unsigned char uc = c;
      sym = RC__.map_char_to_raw_char_sym(uc, file_no__, pos_, &line_no__, &pos_in_line__);
      container__.push_back(*sym);
      if (Pos ≡ pos_) break;
      ++pos_;
      continue;
```

```
    }
    ;
    if (r_w_cnt__ > 1) {⟨ release token mu 392 ⟩}
    return sym;
  }
```

91. *Tok_can < ifstream >***size.**
Due to the just-in-time attitude, the container's size has no meaning. Its size indicates the number of symbols currently in-process and not the total number of characters in the file stream. I guess I could try to use the file system to figure out its size but I'm not sure if this is portable as in the case of line delimiters: DEC versus ASCII. So, just fake it and allow the end-of-file situation deal with it. Use of the "maximum signed integer" constant does the trick in faking it as a very big text file. Who in their mind would create 2 billion characters?: ahhh wait for the XML crowd.

Now who in hell uses this test? My parser does in accessing the token containers by use of the constraint facility testing for possible subscript overflow.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
  yacco2::UINT yacco2::tok_can⟨std::ifstream⟩::size( )
  {
    return INT_MAX;
  }
```

92. tok_can⟨std::ifstream⟩::open_file.
This routine allows one to delay the use of an external file by declaring the container without the file name. Before its use, the file name is supplied by the *set_file_name* method and then the *open_file* method called. For example the container could be declared globally but one supplies the file to-be-read as in passing the file name thru the program's main parameter facility. Removed the "jit" attitude and now read all its input into the container for speeeeed reasons — this is not a William Borough's novel.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
void yacco2::tok_can<std::ifstream>::open_file()
{
  CAbs_lr1_sym *sym(0);
open_file:
  file__.open(file_name__.c_str(), std::ios::in);
  if (file__.is_open()) goto filename_opened;
  else goto filename_bad;
filename_opened:
  {
    file_ok_ = YES;
    ++yacco2::FILE_CNT__;
    have_1st_rec__ = true;
    pos_ = 0;
    if (yacco2::FILE_CNT__ == 1) {
      std::string empty;
      yacco2::FILE_TBL__.push_back(empty);
    }
    yacco2::STK_FILE_NOS__.push_back(yacco2::FILE_CNT__);
    file_no__ = yacco2::STK_FILE_NOS__.back();
    yacco2::FILE_TBL__.push_back(file_name__);
    eof_pos_ = 0;
  set_dont_skip_any_chars:
    file__ >> std::noskipws;
    for ( ; eof_pos_ != EOF; ++pos_) {
      char c;
      if (file__.good() == true) {
        file__ >> c;
      }
      if (file__.eof() == true) goto eoroad;
      if (file__.fail() == true) goto eoroad;
    convert_char_to_unsigned_value: unsigned char uc = c;
      sym = RC__.map_char_to_raw_char_sym(uc, file_no__, pos_, &line_no__, &pos_in_line__);
      container__.push_back(*sym);
    }
  }
filename_bad:
  {
    eof_pos_ = EOF;
    file_ok_ = NO;
    return;
  }
eoroad:
  {
    eof_pos_ = EOF;
```

```
      ++pos_;
      sym = RC__.map_char_to_raw_char_sym(EOF_CHAR_SUB, file_no__, pos_, &line_no__, &pos_in_line__);
      container__.push_back(*sym);    /* 2 eof added really 2 eogs */
      container__.push_back(*sym);
      return;
    }
  }
```

93. tok_can⟨std :: *ifstream*⟩ :: *close_file*.
This routine allows one close a file prematurely or to reuse the token container for another round of parsing.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
  void yacco2::tok_can<std::ifstream>::close_file()
  {
    if (file__.is_open()) {
      if (file_ok_ ≡ YES) {
        file__.close();
      }
    }
    file_ok_ = false;
  }
```

94. Balance of sundry routines.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
yacco2::UINT yacco2::tok_can<std::ifstream>::pos()
{
  return pos_;
}
void yacco2::tok_can<std::ifstream>::push_back(yacco2::CAbs_lr1_sym &Tok)
{
  container__.push_back(Tok);
}
void yacco2::tok_can<std::ifstream>::clear()
{
  container__.clear();
  pos_ = 0;
  have_1st_rec__ = false;
  eof_pos_ = EOF;
  file_ok_ = NO;
  line_no__ = START_LINE_NO;
  pos_in_line__ = START_CHAR_POS;
  file_name__.clear();
}
TOKEN_GAGGLE &yacco2::tok_can<std::ifstream>::container()
{
  return container__;
}
void yacco2::tok_can<std::ifstream>::remove()
{ }
std::string & yacco2::tok_can<std::ifstream>::file_name()
{
  return file_name__;
}
void yacco2::tok_can<std::ifstream>::set_file_name(const char *File_name)
{
  file_name__ += File_name;
}
```

95. Tree token container implementation tok_can⟨AST ∗⟩.
This is your tree container of tokens. A filter mechanism is passed to the template. It is just a set of terminal enumerates with it companion indicator of include or exclude the terminals in the tree traversal within the tree walker.

The traversal operator also keeps a subscript marker as to where its traversed. This allows one to interrogate the container for a token without having to re-traverse the tree. Excuse the acronym but it is a just-in-time delivery mechanism. If the subscript is within bounds of the container, it delivers the already traversed tree's token. Out-of-bounds will continue the tree traversal looking for the requested token-by-number. If the token number is not continuous, the container gets filled up with the inbetween tokens found in the traversal before returning the requested terminal. When the end-of-tree has been met, the *PTR_LR1_eog__* terminal is returned. This is in keeping with the other containers.

Optimization: remove jit for all input filled in at ctor.
This jit optimization is removed due to self modifying of tree nodes. Without this the old container that called the self modifying of a tree node contains the old T in its container. So in with the reader mutex and its slow down and out with the speed for self modifying tree nodes. Please read "Notes to myself" of item "Tree Modifying while walking them..." discussing the "how tos" of dealing with dynamic self-modifying tree setting.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
yacco2::tok_can<yacco2::AST *>::tok_can(yacco2::ast_stack &Walker)
: tok_base(1)
, pos_(0)
, have_1st_rec__(false)
, tree_end_reached__(false)
, nodes_visited_()
, accept_node_level_()
, traverse_(Walker) {
  operator[](0);    /* needed: ensures container has tried to get first T before its use */
}
yacco2::tok_can<yacco2::AST *>::~tok_can()
{ }
bool yacco2::tok_can<yacco2::AST *>::empty()
{
  return nodes_visited_.empty();
}
void yacco2::tok_can<yacco2::AST *>::clear()
{
  nodes_visited_.clear();
}
```

96. Tree container dispensor.
It delivers tokens by the numbers. At present, this number is relative to 0. Ugh!

If the tree node number is within the token container then return it. For token numbers outside the current container, the tree is traversed putting the accepted tokens into the container until either the end-of-tree is reached or the token requested is found. The container of tokens allows one to re-iterate many times over the token stream. It also optimizes the token stream by one-pass-only on the tree. An end-of-tree condition returns the *PTR_LR1_eog__* token back to the user. This is in the same spirit of the other token containers. It allows grammars to be written without any knowledge as to its input token stream.

```
⟨ accrue tok_can code 77 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::tok_can⟨yacco2::AST *⟩::operator[](yacco2::UINT Pos)
  {
    AST *t;
    CAbs_lr1_sym *tsym;
    AST *vnode;
    CAbs_lr1_sym *sym;
    ast_base_stack::s_rec *srec;
    if (tree_end_reached__ ≡ true) {
      if (Pos < pos_) goto in_bnds;
      if (YACCO2_T__ ≠ 0) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "YACCO2_T__::tok_can␣token␣eog:␣"
         ≪ PTR_LR1_eog__ ≪ "␣pos:␣" ≪ Pos ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
      sym = PTR_LR1_eog__;
      goto rtn_fnd_T;
    }
  first_time_accessed:
    if (have_1st_rec__ ≡ false) {
      have_1st_rec__ = true;
      goto out_bnds;
    }
  determine_where_t_is:
    if (Pos ≤ pos_) {     /* already in container */
      goto in_bnds;
    }
    ++pos_;     /* next node */
    goto out_bnds;
  in_bnds:
    ⟨ fetch and return token from container instead of tree 97 ⟩;
  out_bnds:
    if (r_w_cnt__ > 1) {⟨ acquire token mu 391 ⟩}
  get_tree_rec:
    {
      ⟨ traverse tree 100 ⟩;
      ⟨ end of traverse reached? yes rtn 101 ⟩;
      ⟨ put node in container 102 ⟩;
      if (Pos ≡ pos_) goto rtn_fnd_T;
      ++pos_;
      goto get_tree_rec;     /* keep filling container until Pos met */
    }
```

```
rtn_fnd_T:
  if (r_w_cnt__ > 1) {
    ⟨release token mu 392⟩;
  }
  return sym;
}
```

97. Fetch and return token from container instead of tree.
Prefetch next T and place in container when the current request is on its boundry and parallel readers are occuring.
Ip constraint: The sequential request always has the T inside its container.
Random request: Who'll need it? If it happens, the container's suitor count is checked and protected with a mutex.

```
⟨fetch and return token from container instead of tree 97⟩ ≡
  t = nodes_visited_[Pos];
  tsym = AST::content(*t);
  if (YACCO2_T__ ≠ 0) {
    ⟨acquire trace mu 389⟩;
    yacco2::lrclog << "YACCO2_T__::tok_can␣in-bnds␣already␣in␣container␣token:␣" << tsym→id__
      << "␣*:␣" << tsym << "␣pos:␣" << Pos << "␣id:␣" << tsym→id__ << "␣enum:␣" <<
        tsym→enumerated_id__ << FILE_LINE << std::endl;
    yacco2::lrclog << "\t\t::GPS␣FILE:␣";
    EXTERNAL_GPSing(tsym)yacco2::lrclog << "␣GPS␣LINE:␣" << tsym→tok_co_ords__.line_no__ <<
        "␣GPS␣CHR␣POS:␣" << tsym→tok_co_ords__.pos_in_line__ << FILE_LINE << std::endl;
    ⟨release trace mu 390⟩;
  }
  ⟨lookahead T needed? no rtn fnd t 98⟩;
```

This code is used in section 96.

98. Lookahead T needed? no rtn fnd t.
Lookahead is only needed when parallel reads are happening. If there is only one reader, it is always safe and can default to "jit" access.

```
⟨lookahead T needed? no rtn fnd t 98⟩ ≡
  if (r_w_cnt__ ≡ 1) return tsym;    /* no parallel suitors */
  if (Pos < pos_) return tsym;    /* not on the edge */
  ⟨acquire token mu 391⟩
  if (tree_end_reached__ ≡ true) {    /* ure parallel phatom got here before u */
  }
  else {
    if (Pos ≡ pos_) {    /* still needed as the other suitor could have looked ahead */
      ++pos_;
      ⟨traverse tree 100⟩;
      ⟨end of traverse reached for lookahead? no put T in container 99⟩;
    }
  }
  ⟨release token mu 392⟩return tsym;
```

This code is used in section 97.

99. End of traverse reached for lookahead?.

⟨ end of traverse reached for lookahead? no put T in container 99 ⟩ ≡

```
if (tree_end_reached__ ≠ true) {    /* test for other consumer's action */
  srec = traverse_.base_stk_.cur_stk_rec_;
  if (srec ≡ 0) {
    tree_end_reached__ = true;
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::tok_can␣token␣eog:␣"
       ≪ PTR_LR1_eog__ ≪ "␣pos:␣" ≪ pos_ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
  else {
    ⟨ put node in container 102 ⟩;
  }
}
```

This code is used in section 98.

100. Traverse tree.

⟨ traverse tree 100 ⟩ ≡

```
traverse_.exec( );
```

This code is used in sections 96 and 98.

101. End of traverse reached?.

⟨ end of traverse reached? yes rtn 101 ⟩ ≡

```
srec = traverse_.base_stk_.cur_stk_rec_;
if (srec ≡ 0) {
  tree_end_reached__ = true;
  if (YACCO2_T__ ≠ 0) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_T__::tok_can␣token␣eog:␣"
     ≪ PTR_LR1_eog__ ≪ "␣pos:␣" ≪ pos_ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
  sym = PTR_LR1_eog__;    /* end-of-tree */
  goto rtn_fnd_T;
}
```

This code is used in section 96.

102. Put node in container.

⟨ put node in container 102 ⟩ ≡
  vnode = *traverse_.base_stk_.cur_stk_rec_*→*node_*;
  sym = **AST** :: *content*(∗*vnode*);
  accept_node_level_.push_back(*traverse_.base_stk_.idx_*);
  nodes_visited_.push_back(*vnode*);
  if (`YACCO2_T__` ≠ 0) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2 :: *lrclog* ≪ `"YACCO2_T__::tok_can␣token:␣"` ≪ *sym*→*id__*
     ≪ `"␣*:␣"` ≪ *sym* ≪ `"␣pos:␣"` ≪ *pos_* ≪ `"␣requested␣pos:␣"` ≪ *Pos* ≪ `"␣node*:␣"` ≪ *vnode* ≪
       `"␣node␣content*:␣"` ≪ **AST** :: *content*(∗*vnode*) ≪ `FILE_LINE` ≪ **std** :: *endl*;
    ⟨ release trace mu 390 ⟩;
  }

This code is used in sections 96 and 99.

103. Balance of tree container routines.

⟨ accrue **tok_can** code 77 ⟩ +≡

```
yacco2::UINT yacco2::tok_can<yacco2::AST *>::pos()
{
  return pos_;
}
yacco2::UINT yacco2::tok_can<yacco2::AST *>::size()
{
  if (tree_end_reached__ == true) {
    return nodes_visited_.size();
  }
  return MAX_UINT;
}
void yacco2::tok_can<yacco2::AST *>::push_back(AST &Tok_ast)
{
  nodes_visited_.push_back(&Tok_ast);
  ++pos_;
}
void yacco2::tok_can<yacco2::AST *>::      /* defed due to template */
push_back(yacco2::CAbs_lr1_sym &Node)
{ }    /* but not meaningful in tree context */
yacco2::ast_stack &yacco2::tok_can<yacco2::AST *>::container()
{
  return traverse_;
}
std::vector<AST *> *yacco2::tok_can<yacco2::AST *>::nodes_visited()
{
  return &nodes_visited_;
}
void yacco2::tok_can<yacco2::AST *>::remove()
{
  nodes_visited_.pop_back();
  --pos_;
}
yacco2::AST *yacco2::tok_can<yacco2::AST *>::ast(yacco2::UINT Pos)
{
  if (Pos > pos_) return 0;
  return nodes_visited_.operator[](Pos);
}
yacco2::INT yacco2::tok_can<yacco2::AST *>::accept_node_level(yacco2::UINT Pos)
{
  if ((accept_node_level_.size() - 1) < Pos) return No_Token_start_pos;
  return accept_node_level_.operator[](Pos);
}
```

104. Structure and Rule Recycling Optimization.
To improve performance due to the birth-run-delete cyle of grammar rules on the parse stack, the following optimization is used: Stable of rule's symbol when created for recycling purposes. 2 concerns must be attended to:

1) the parse stack needs the Rule_s_reuse_entry ptr of the current rule
2) due to recursion, the recycle table per rule is sequentially searched

Please see *rules_use_cnt* grammar for a thorough discussion on how the rulle count is calculated for recycling.

Initially an array per specific grammar rule was generated. It had speed but would have been kludgey to handle overflow on the number of rules for reuse. Here is a note on the array [1] definition. The Sun compiler doesn't like the [] definition being open-ended. So I fake it. Each rule will be specifically defined within its namespace. But, Yac_2o_2 is a general library of routines. So my search uses the entry count to protect against a table overrun situation.

Take 2:
Though this approach is speedy when dealing with left recursion only, it did not have a saftey valve when the count was wrong: eg right recursion or flawed algorithm on determining rule recursion count. So i changed it to a stack/double list combo. The "in use" list acts like a stack but its lhs/rhs reduction pair leaves the lhs as the top item placed in the "in use" queue before its rhs items are removed from the "in use" list. Why? The lhs rule comes from the reduction when the rhs's symbols are still sitting on the parse stack. That is, lhs rule is created first, placed in the "in use" list before the rhs's symbols are popped from the parse stack. If the popped symbol is a rule it gets recycled and placed back into the "for use" stack for another round of reuse.

⟨Structure defs 18⟩ +≡

```
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_;     /* new rule symbol for recycling */
  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 {      /* grammar's stable of rules */
  int no_rules_entries_;
  Per_rule_s_reuse_table *per_rule_s_table_[1];
};
```

105. Finite automaton table definitions and their functions. These definitions support Yacco2's generated finite state automaton tables. A binary search is used on all tables: **Shift_tbl**, **Reduce_tbl**, and **State_s_thread_tbl**. Their structure contains the prefix giving the number of elements in the table, and the first record in the array. The elements are a concatenation of 'in ascending sequence' sorted records for the binary search.

106. State structure.
This represents the finite automaton state. The only wrinkles to your normal finite state definition are the entries supporting parallelism and the 2 meta terminals for the 'all shift' and 'invisible shift' functions. These extra shifts act like a normal shift requiring their own shift entries.

Parallelism is the `|||` grammatical expressions within the state calling threads. Each expression supplies the thread and the returned terminal be it successful or an error terminal. An aborted thread returns nothing. The expression itself requires 2 shifts: the `|||` followed by the winning terminal that the arbitrator has selected. Why is there not 3 shifts to include the thread used? I originally thought of this but it has no relevance to the expression parsed. The thread call is a pre-conditional condition to the T stream. If all the threads have aborted, then the `|||` terminal must be removed from the parse stack before trying the standard finite automaton's operations. The list of threads associated with the state needing launching completes the declaration of parallelism.

proc_call_shift__ has been added to deal with chained procedure calls. What the heck is that? It is a dispatcher of procedure calls reacting to the returned T. This grammatical structure allows one to call a thread, react on the returned T by calling a specific procedure. For example, this subrule `|||` "lhs" *TH_id Rdispatch_lhs*. The thread "id" is a identifier / symbol table lookup for keywords on a character token stream. The following *Rdispatch_lhs* becomes the dispatcher of called procedures based on the returned T "first set" is "lhs". *Rdispatch_lhs* subrule would be `|t|` "lhs-phrase" *PROC_TH_lhs_phrase* receiving the "lhs" start T. Its other subrules would be programmed to catch the errors. This "procedure call" sublety requires the called procedure to use the stacked returned T "lhs" as its current T and not the current T of the caller. Also it must set its own token position to 1 less the caller's current token position. There is an overlap on the input token stream whereby the characters used to create the "lhs" T are still in the supplier's token stream and not "lhs".

The other subtelty is a non-chained procedure call when the calling parser has only 1 thread to call so call it as a procedure and not as a thread to juice the optimization process.

questionable_shift__ is used in questionable situations like error detection points within a grammar. See notes to myself for an explanation.

⟨ Structure defs 18 ⟩ +≡

```
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__;   /* |t| */
  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__;   /* function for |t| */
  yacco2::Shift_entry *questionable_shift__;   /* |?| */
};
```

107. Shift table lookup.
The **Shift_tbl** is a binary array of **Shift_entry** of the finite state. The shift operation goes through a sequential list of ranked terminals trying always to shift first before trying to reduce. The ranking of potential shifts are:

1) current terminal being parsed
2) questionable shift terminal `|?|`
3) invisible meta terminal `|.|`
4) all shift terminal `|+|`

Their presence in the state's configuration dictates the shift operation. There are 4 individual search attempts to see whether the shift operation should take place. The numbered points indicates their ranking order: point 2 and 3 should be mutually exclusive.

The *goto__* in the *shift_entry* is your vanilla flavoured fsa 'go to' state. The actual state definition is laced with extra information to support parallel and conditional parsing. `|.|`is a bailout mechanism from ambiguous gramatical contexts. It can be used to describe an epsilon rule. How? Though there is a shift happening, there is no consumption of the token stream. Its use depends on the palative tastes of the grammar writer or the ingredients demanded by the grammar.

⟨Structure defs 18⟩ +≡
```
struct Shift_entry {
    yacco2::USINT id__;
    yacco2::State *goto__;
};
struct Shift_tbl {
    yacco2::USINT no_entries__;
    yacco2::Shift_entry first_entry__[1];
};
```

108. Reduce table entry.
The *Reduce_entry* gives the lookahead set number to be checked. The *rhs_id__* gives the subrule identity that will collapse to its left-handside rule. Where is the binary compare function? It is the set compare function. See *Set handling*.

⟨Structure defs 18⟩ +≡
```
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];
};
```

109. Threading Definitions.
Lots of merit but if it's not fast then this idea is side-lined or in football terms benched. To optimize the dispatching of threads, a global approach is required. This is resolved by Yacco2's linker. Why is a global approach needed? Sequential first set evaluation per thread within the state's configuation is just tooooo slowww. To properly assess the first sets of all threads, the linker must read the "fsc" files generated per thread by Yacco2. The linker can now apply the transience operator on the first sets where a thread can call another thread in its first set: the start (closure) state of the grammar could contain a call to a thread.

Thought:
How many stacked focuses does one need with fad out to see the forest from the trees? Programming demands this talent of Yoga reflection but how many times have u consciously observed oneself observing oneself... In this case, the tree scope lost to the forest, as the local optimizations discussed in *Notes to myself* had reached their effectiveness and I still needed more improvement.
Thought no 2:
Why wasn't this global approach thought of before now? Well I tried to get my threading ideas to work first. Thoughts of efficiency were not my first priority. Now reality of slowness demands gettting it to work faster. The speed approach is test the current token's enumeration id against a global "thread list having T in their first set" when paralellism is present within the finite automaton's current state's configuration. If there are threads with this first set item, then go thru the state's potential thread list looking to launch them. On an aside, common prefix threads will showup together in their common terminals. There should not be too many of these so the list should be short — normally one thread. To get speed, a thread id is required. It is the enumeration of all the thread grammars. This enumeration is done within Yacco2's Linker. As Yacco2 is local to the grammar being compiled, its local table must use indirection to get at this thread id. So u will see pointers to items that only get resolved by the language linker. See ⟨ Global external variables from yacco2's linker 19 ⟩ for the global symbols referenced within this library but generated by Yacco2's Linker.

Mutexes controlling the hoards:
1) **yacco2** :: `TOKEN_MU` - token dispensor access
2) **yacco2** :: `TRACE_MU` - used to log tracing
3) **yacco2** :: `TH_TBL_MU` - access thread dispatch table
4) **yacco2** :: `SYM_TBL_MU` - symbol table access

With my dual core AMD Sun work station, readonly access to the token dispensor requires a mutex `TOKEN_MU` to prevent thread residues poluting other threads accessing "at the same time" their tokens. My tracings re-affirmed my intuitions as to why it was not working in this configuration. Past portings onto Apple's OSx, VMS Alpha, and NT Windows all worked. In a single chip environment execution is normally sequential but in multi-chip environments parallel execution streams are dancing together on the same stage. `TOKEN_MU` ensures that each fetch to the token supplier is atomicly completed before others requests are serviced. Unfortunately this has a potential braking effect by throatling back to 1 only thread executing if there are multiple simultaneously token read requests happening until i can explore who / what causes the downstream polution. Currently my library is staticly declared and not declared as shared.??? Remember as multiple threads are launched by a parser, each thread's execution path is asynchronous in their token fetches even though each launched competing thread starts at the same position within the token stream. Please see "Notes to myself" on eliminating the "jit" token fetch.

`TRACE_MU` mutex ensures that the complete text traced is completely outputed. The atomicity is bracketed by the acquire / release cycle of the `TRACE_MU` mutex. This prevents interleaving of parallel thread loggings to occur. For example, i/o calls are fielded by the operating system; it is the operating system's decision as to who will run next.

`SYM_TBL_MU` is reserved for possible parallel symbol table access. `TH_TBL_MU` is the bouncer of the global thread table that registers launched threads. These thoroughbreds keep their engines running with environmental friendly octane while waiting for their next serve request that provides the needed pep to parallel parsing. As each access to the table is read / write, `TH_TBL_MU` keeps this critical region in tip-top shape.

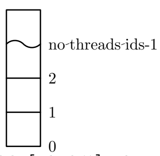
The following section discusses in detail how this table is used.

110. Critical region discussion surrounding *Parallel_thread_table***.**
Parallel_thread_table raison d'être is speed. Depending on the parsed context, threads are created dynamically. This stable of threads are reused on demand that eliminates the create-run-destroy cycle of a thread. Now it's create once, run as many times as needed, and exit when finished parsing. Nested thread calls like recursion is supported: thread A calls thread B calls thread A. Each thread in the list keeps an availability status: busy or idle. There are 2 parts to the global thread table:

1) *Parallel_thread_table* — the array of thread lists
2) `TH_TBL_MU` mutex — the guard dog controlling the crowds

thread_list :

- list<worker_thread_blk↑>
  - parser ↑ — grammar containing worker_thread_blk
  - status — idle, working, or exiting
  - run cnt — stats on how many times thread executed
  - thd id — thread id number



Parallel_thread_table[*thd#*].*thread_list*

The above figure depicts the thread table generated from O_2 linker. The 2 contexts requiring reader / writer access are:

1) grammar's launching or requesting threads to run
2) launched thread setting its work status back to idle or exiting

As an optimization, threads receive an unique ordered id from O_2 linker. This is just a lexigraphical ordering on their names allowing table access by subscript. The thread table is a single writer controlled by mutex primitives ⟨ acquire global thread table critical region 380 ⟩ and ⟨ release global thread table critical region 381 ⟩. These *cweb* sections are calls to the thread manager using the `TH_TBL_MU` mutex. To acquire control a launching grammar uses the ⟨ acquire global thread table critical region 380 ⟩ primitive. If someone else has possession on the resource, the thread manager places the requestor into a hold queue until the resource is freed. It is the thread manager that dispenses execution control.

Thread table possession:
Quick review:
A grammar's finite automata can contain lists of threads for the running within each state's context. To juice the running, each thread has a first set of tokens that start its parse. Potential thread launch evaluation uses the current token against these first sets to determine what threads should run.

So possession is $9/10^{th}$ of whose law? Now launch or run those threads by calling the thread manager — the "how" will be described later. New threads add their *worker_thread_blk** to the thread table without any care for critical region hygiene. The **Parser** object of the newly launched grammar does it from its *constructor*. Cuz the launching grammar has possession of the thread table and the launched threads are unique, there is no potential reader / writer destructive scribbling to the table. A thread's work status is maintained in the table depending on how they get run. "Just created" threads do a *push_back* of their *worker_thread_blk** into the thread list while "already created" threads set their *worker_thread_blk*'s status to busy that is already registered in the thread table's list. A grammar's potential thread list does not contain multiple requests of the same thread so that u'll never get a parallel set of identical threads spoiling the broth within the same launch list. Remember the table's granularity is by thread id subscript: So there is no conflict.

Note:
If the thread manager flips execution to a launched thread (single or multiple cpus don't matter) and this newly executed thread requires thread table access, it must call the ⟨ acquire global thread table critical

region 380 ⟩ that puts its request on hold until the resource is freed up. Eventually the original grammar releases control of the thread table by ⟨ release global thread table critical region 381 ⟩ that activates execution of the requestor.

Sleeping beauty:
Finally the calling grammar places itself into a wait state (is it ripper van winkle?) to be wakened by one of its called dwarfs. This is done by calling the ⟨ wait for event to arrive with no loop 394 ⟩ that releases the grammar's mutex, puts it on ice, and places its conditional variable into the thread manager's event wait queue. Freeing up of these "thread manager" variables allows its called threads to play with its calling grammar's critical region and to eventually wake it up. Remember, each called thread must go thru the acquiring / releasing of the called thread's mutex. U wouldn't want the dwarfs to screwup ogre's critcal region and the grammar writer's ire. Why the playing with the calling grammar's critical region away? Its called threads can report back their parse findings thru the "acceptance token" queue of the sleeping beauty. To wake up the ogre, the last thread finished executing calls primitive ⟨ signal thread to wake up and work 397 ⟩. How is this determined? The calling grammar's critical region has a launched thread count. Each called thread decrements it when completed regardless of its parsing outcome. When it hits zero, this indicates last thread to finish and so jostle the snoring beauty. The last duty of a running thread is ⟨ acquire global thread table critical region 380 ⟩, set its run status to idle ,⟨ release global thread table critical region 381 ⟩, and place itself into a wait state for another round of drinks: ⟨ wait for event to arrive with no loop 394 ⟩.

How does a called thread know its requestor?
Let's review the 2 situations:

1) create a thread
2) call an already created thread

There are 2 doors of entry into a thread. "Creation of a thread" is at the mercy of the thread manager to register the thread and prepare it for the calling. The only way information can be passed to the to-be created thread is thru a parameter passed to the called thread procedure by the thread manager. The calling grammar's **Parser** object address is passed as a parameter to `CREATE_THREAD` who passes it to the to-be-executed thread. Built within the thread code is the casting and extraction of the requestor's **Parser** object. Once the called thread is finished running, it puts itself into a wait state for its next marching order.

The 2nd port of entry.
U guessed it, the thread list contains the thread's **Parser** object that has been freed of its mutex and conditional variable put on ice. So the 2nd entry point is the ⟨ wait for event to arrive with no loop 394 ⟩. The calling grammar calls `SIGNAL_COND_VAR` to wake up the dwarf while the called thread uses the ⟨ signal thread to wake up and work 397 ⟩ to wake up the ogre that really calls `SIGNAL_COND_VAR`. Within the critical region of the "to be requested" thread is *pp_requesting_parallelism__* that holds the calling grammar's deposited critical region address. Note: thru out a parse a thread can be activated by different suitors. Each deposit by the requesting grammar leaves its tale for the dwarf.

Draining the thread swamp:
How does one get out of this infinite loop of wait for its marching order, do the parse, and wait again. This is Sambo and the tigers twirl: tail chasing ain't it? There is another marching order to exit-work. A bit of a subtlety here needs explaining: how does one know if the thread manager has placed all the toe tapping threads into a wait status within a single cpu environment? To let the swamp drain, a ⟨ pause for x seconds 181 ⟩ takes place that could be not effective but i'm trying: better yet would be to have a *pthread* procedure to do the act of bleeding... followed by a "stop work" order — it has other euphemisms. This is how the thread breaks out of its tail spin. The global *Parallel_threads_shutdown* procedure initiates the above and details the threads run stats and shutdown attempts. It is usually called from the "mainline" code of the program.

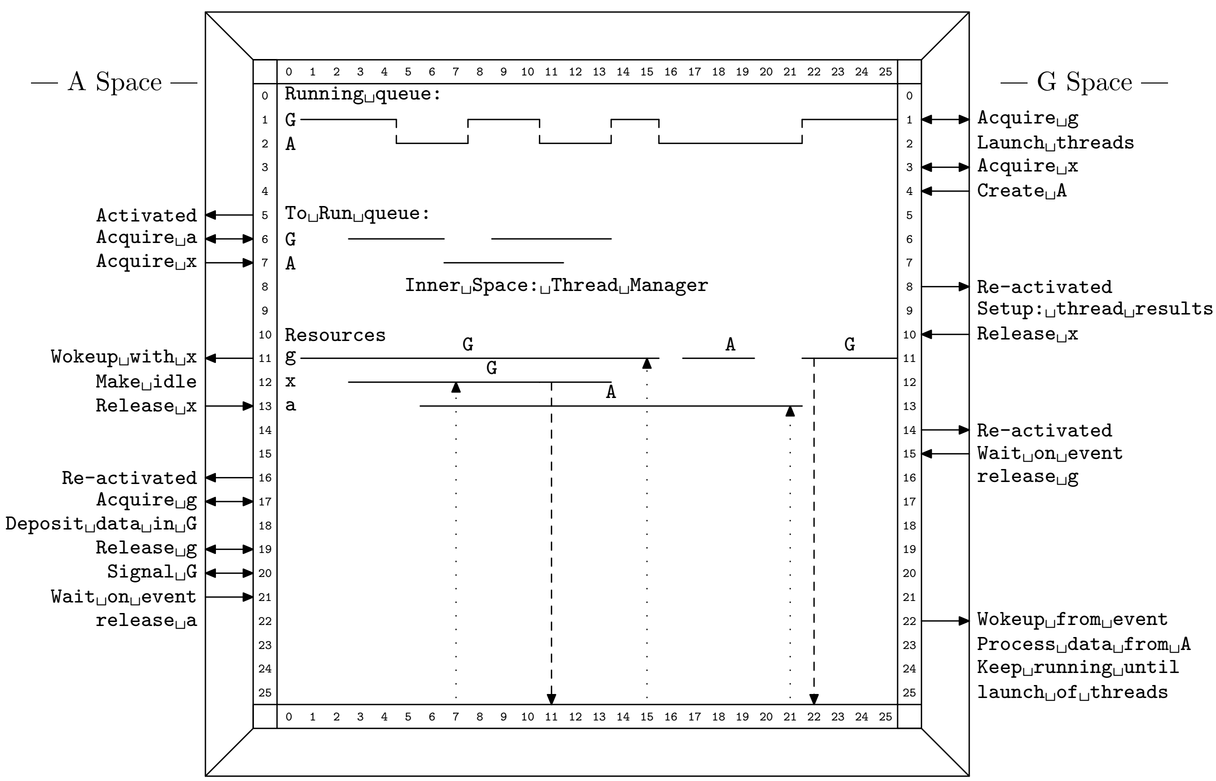
111. Diagrams, do we have diagrams — examples of critical region activity. Let a figure detail a 1000 words. In a single cpu environment, a process's execution sequence is sequential. To depict this using G as the process, A and B threads, and the critical region resources, i will use a box within a box concept to simulate multi-dimensions. Why a box? In one of the following examples there are 3 outer space dimensions representing G, A, and B. This really is a triangle but the running comments and activity vectors makes it easier to annotate using a box. An obelisk with its point removed represents all the dimensions.

Going from the outer to the inner parts of the obelisk, the outer walls are the process / thread spaces. Next, time rulers are the motes between outer and inner spaces. The court yard is the inner space (resource space). It contains the critical regions' resources, and execution queues — running and waiting to run.

Commented outer space events are registered aginst its time mark by vectors using an arrowhead to indicate the activity's direction into or out of the resource space. A double headed vector indicates the outer space call to the inner space that returns execution back to the calling outer space.

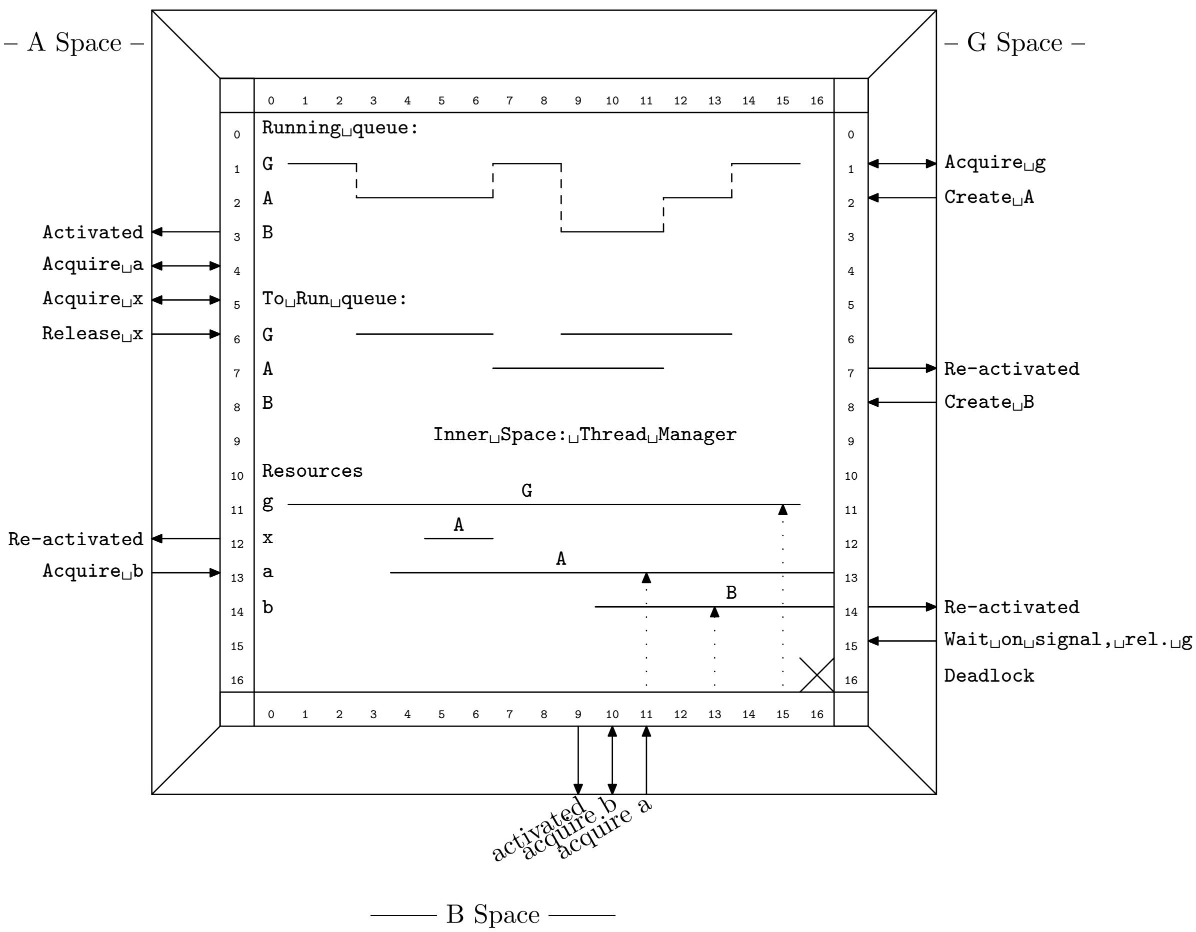
To indicate ownership and duration of time, each resource uses a line similar to the math open / close interval. The "running queue" also ties together the start/stop boundaries with a dashed vertical line to show continuity. Other resources have the owner above their time line marker. A dotted vertical vectored to the resource marks a request for ownership that is pending. Its converse uses a dashed line away from the resource marking the acquisition from a pending a request.

Example of threads being run by O_2.



A single thread A that gets launched and reports back to its caller G . The resource "x" is the global guard to the global thread table. Basic comments on the critical region components of G have been left out due to space. As previously described, an active thread count is maintained along with the acceptance token queue that the called threads deposit their results for G's arbitration code assessment. Lines 18 and 23 comments these situations with the bracketed acquisition. Line 18 shows the called thread A reporting its results within G's protected area. The signal variables of G and A have also been ommitted due to space. In the above example, it would not have mattered whether the launched thread started executing immediately with the calling grammar put on hold as the launching grammar G still has ownership on "x" that eventually the A will require and so it would be put into a pending state until the "x" resource could be re-allocated. In this illustration, G goes into a wait state to be signalled later by A. If the interweave of G's execution sequence was such that A was working and signaled G before G put itself into a wait-on-signal state, it is the thread library that pends the signal for when G finally requests it.

A Deadlock Example:



Some comments:
Deadlock is a graph of cyclicity. In the example, resource "x" is an intermediary used by the thread manager to relinquish execution control held by A when it releases "x". Process G then continues by creating thread B with its Acquire events on "b" and attempts on "a". Eventually thread A attempts its acquiring of "b". By sequencing the Acquire requests — Acquire(a) by A, Acquire(b) by B, Acquire(a) by B, and Acquire(b) by A, a cyclicity check could be done per Acquire to determine whether deadlock is met. The third Acquire(a) by B has the potential deadlock cyclic condition established. Because A is still running, it is not a conclusive deadlock as thread A could Release(a) to free up the cycle created by B. Only when thread A asks for "b" does it become a solid deadlock regardless of process G being able to run.

The simplest run death is G requesting a wait-on-signal when there are no other threads running that could wake it up — Sleeping beauty with no Prince to do resusitation.

112. Thread entry.
Just your basic attributes describing a thread. Each thread block is generated by the Linker. Remember, the thread ids are in lexigraphical order: upper / lower case are different. Only the Linker has access to all the threads to produce this order. Each thread entry block will have the Linker's manufactured thread name which will be referenced by the state's thread table and the global stable of threads. The thread entry will be identified by the following rule:

concatenate the letter "I" to the thread's name

For example, "*TH_eol*" is the end-of-line detector thread. Its variable name would be "*ITH_eol*" where the *TH_eol* value is taken from the grammar's "parallel-thread-function" component.

The reason for the *thread_array_record* having an array of **Thread_entry** ∗ is due to the thread entry name. It is referenced by the **State_s_thread_tbl** and can be referenced by the grammar writer when using the *spawn_thread_manually* procedure. The thread entry names are generated by Yacco2's Linker that is outside of Yacco2's library jurisdiction but used by it. This generation is specific per language being generated.

⟨ Structure defs 18 ⟩ +≡
```
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__;
};
```

113. Thread stable.

⟨ Structure defs 18 ⟩ +≡
```
struct thread_array_record {
  yacco2::USINT no_entries__;
  yacco2::Thread_entry *first_entry__[1];
};
```

114. State's thread table.
The thread entries are in sorted order. How? Though the list of potential threads order within the grammar are as programmed by the grammar writer, their names will be sorted lexigraphically. Hence their order in the table are relatively sorted.

The thread entry variable and its contents are generated by Yacco2's Linker.

⟨ Structure defs 18 ⟩ +≡
```
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];
};
```

115. Threads having terminal in first set.
Well here's the turbo charger of threads. It is generated by Yacco2's Linker. As the number of terminals defined is unknown to this general library, a spoofing technique is used.

Have a pointer to a structure that defines the running grammar's environment that contains another indirection to the local information. I use T as a generic symbol representing the individual terminals within the grammar's Terminal vocabulary. These 2 structures are:

1) terminal array pointing to the threads with T in the grammar's first set
2) the thread id list having T in their first set

This spoofing technique is:

⟨ Structure defs 18 ⟩ +≡
```
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];
};
```

116. Finite state machine definition.

117. *CAbs_fsm*.
It provides the basis for all grammar 'fsm' definitions. Yacco2 generates a specific 'fsm' per grammar derived from *CAbs_fsm*. The first 5 parameters are the grammar attributes extracted from the 'fsm' construct of the grammar. Parameters *Gened_date* thru to *Start_state* are specifics from the compiling of the grammar. For-your-information, the date and time as to when the grammar was compiled is passed by *Gened_date*.

Start_state parameter is the object address. *Start_state* is the "S" in your formal finite automaton definition.

⟨ Structure defs 18 ⟩ +≡
```
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__;
};
```

118. Trapping of Premature Parsing Failures — failed directive.
The “failed” directive within the “fsm” construct allows one to deal with premature aborts within a grammar. It makes it reeeeeeeal easy to trap errors instead of specifically trying to program within the grammar each potential abort position per T shift. It’s a “catch-all” last chance to provide an error response back from a threaded grammar to their calling grammars, or to place an error within the error queue of a monolithic grammar. A failed example:

```
fsm
(fsm-id "reset_rewrite_opt.lex",fsm-filename reset_rewrite_opt
,fsm-namespace NS_reset_rewrite_opt
,fsm-class Creset_rewrite_opt {
  user-prefix-declaration
#include "integer_no.h"
  ***
/@
Trap the failed option and return a bad option.
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* s = new LR1_err_bad_rsx_rms_opt;
      s->set_rc(*parser()->current_token(),*parser()
               ,"reset_rewrite_opt.lex",__LINE__);
      RSVP_FSM(s);
      return true;
  ***
}
,fsm-version "1.1",fsm-date "18 Oct. 2003",fsm-debug "true"
,fsm-comments "individual rsx/rms options")
parallel-parser
(
  parallel-thread-function
    TH_reset_rewrite_opt
  ***
  parallel-la-boundary
    "/" + "’"
  ***
)
```

119. Finite state machine implementation.

120. CAbs_fsm and ~CAbs_fsm.
Constructor and destructor of the finite state class.

⟨ accrue yacco2 code 33 ⟩ +≡
yacco2 :: **CAbs_fsm** :: **CAbs_fsm**
(**yacco2** :: **KCHARP** *Id*
, **yacco2** :: **KCHARP** *Version*
, **yacco2** :: **KCHARP** *Date*
, **bool** *Debug*
, **yacco2** :: **KCHARP** *Comments*
, **yacco2** :: **KCHARP** *Gened_date*
, **yacco2** :: **State** &*Start_state*)
: *id__*(*Id*), *version__*(*Version*), *date__*(*Date*), *gened_date__*(*Gened_date*), *debug__*(*Debug*), *comments__*(*Comments*), *start_state__*(&*Start_state*), *parser__*(0) { }

yacco2 :: **CAbs_fsm** :: ~**CAbs_fsm**( )
{ }

121. Fsm implementation.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::State *yacco2::CAbs_fsm::start_state()
{
  return start_state__;
}
yacco2::Parser *yacco2::CAbs_fsm::parser()
{
  return parser__;
}
void yacco2::CAbs_fsm::parser(yacco2::Parser &A)
{
  parser__ = &A;
}
yacco2::KCHARP yacco2::CAbs_fsm::gened_date()
{
  return gened_date__;
}
yacco2::KCHARP yacco2::CAbs_fsm::id()
{
  return id__;
}
yacco2::KCHARP yacco2::CAbs_fsm::version()
{
  return version__;
}
yacco2::KCHARP yacco2::CAbs_fsm::date()
{
  return date__;
}
bool yacco2::CAbs_fsm::debug()
{
  return debug__;
}
yacco2::KCHARP yacco2::CAbs_fsm::comments()
{
  return comments__;
}
```

122. *find_a_recycled_rule* **and** *recycle_rule*.

Each *fsm* is virtual and the concrete grammar's *fsm* gets gened up with its specific *reduce_rhs_of_rule*. It is here that the fetching of recycled rules are done. The popping of the parse stack by cleanup or a reduce operation recycles the rules. For the love of speed and environment, Recycle baby recycle!

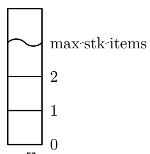
⟨ accrue yacco2 code 33 ⟩ +≡

```
void CAbs_fsm::find_a_recycled_rule(Per_rule_s_reuse_table *Reuse_rule_table, Rule_s_reuse_entry
        **Reuse_rule_entry)
{
  reuse_rule_list *rrl(0);
  if (Reuse_rule_table→for_use_list_ ≠ 0) {
    rrl = Reuse_rule_table→for_use_list_;
    (*Reuse_rule_entry) = rrl→reuse_rule_entry_;
    Reuse_rule_table→for_use_list_ = rrl→older_;
  }
  else {
    (*Reuse_rule_entry) = new Rule_s_reuse_entry( );
    rrl = &(*Reuse_rule_entry)→its_linked_list_;
    rrl→reuse_rule_entry_ = (*Reuse_rule_entry);
    rrl→per_rule_tbl_ptr_ = Reuse_rule_table;
  }
fnd_rrl: rrl→older_ = 0;
  rrl→younger_ = 0;
  if (Reuse_rule_table→in_use_list_ ≠ 0) {
    Reuse_rule_table→in_use_list_→younger_ = rrl;
    rrl→older_ = Reuse_rule_table→in_use_list_;
    Reuse_rule_table→in_use_list_ = rrl;
  }
  else {
    Reuse_rule_table→in_use_list_ = rrl;
  }
}
void CAbs_fsm::recycle_rule(Rule_s_reuse_entry *Rule_to_recycle)
{
  Per_rule_s_reuse_table *reuse_tbl = Rule_to_recycle→its_linked_list_.per_rule_tbl_ptr_;
  reuse_rule_list *iul = reuse_tbl→in_use_list_;
  reuse_rule_list *ful = reuse_tbl→for_use_list_;
  reuse_rule_list *rrl = &Rule_to_recycle→its_linked_list_;
  reuse_rule_list *older_rrl = rrl→older_;
  reuse_rule_list *younger_rrl = rrl→younger_;
    /* break bonds from "in use" and reattach to "for use" */
  rrl→younger_ = 0;
  rrl→older_ = reuse_tbl→for_use_list_;
  reuse_tbl→for_use_list_ = rrl;
  if (rrl ≡ iul) {    /* removal was end of iu list */
    reuse_tbl→in_use_list_ = older_rrl;
    if (older_rrl ≠ 0) {
      older_rrl→younger_ = 0;
    }
    return;
  }
  if (older_rrl ≡ 0) {    /* rechain the iu list */
```

```
      younger_rrl→older_ = 0;
      return;
    }
    younger_rrl→older_ = older_rrl;
    older_rrl→younger_ = younger_rrl;
  }
```

123. Parse stack environment.

parse_record:
- symbol↑
- state↑
- aborted boolean valued
- Rule_s_reuse_entry↑
  - recycled rule↑
  - used on stack — boolean valued



parse stack[].parse_record

Some general comments on the parse stack environment:
Firstly it's just an array of *parse_record* whereby the determinist push-down automaton straddles 2 array records: the first record contains the state address and its stacked symbol and the second record contains the goto state that it vectors to. To improve parsing speed, the rule's "birth-run-delete" cyle has been replaced by recycling of the rule: "birth once run forever" until the parser is shutdown. To do this a **Rule_s_reuse_entry** is kept per required number of recurse / use count per rule. This is determined by analysing the grammar and counting the rhs of each rule for the rule's use patterns. See *structure.w* of O_2library explaining this.

Each grammar locally contains its "rules's reuse" table. The *reduce_rhs_of_rule* procedure reads the recyled rules's table and returns the dupple containing the rule and the address within the recycle table containing the **Rule_s_reuse_entry**. Both components are pushed onto the parse stack frame. When the parse stack frame is popped due to a reduce of the rhs of a rule or due to an abort, each stack frame being popped is inspected for its symbol context: Rule or Terminal, or possibly nothing. If the symbol context is of Rule, the **Rule_s_reuse_entry**'s "in use" indicator is reset for recycling.

Another subtlety is that of "how to reset the rule's object"?.
In c++ terms, as the rule's class only has ctor and possibly a dtor that are implicitly called by the generated code, "how do u reset the object for reuse as this is not a copy situation?". Not to blame c++, this situation was not thought of until now by me. This requires an inspection of the grammar rule's definition for a grammar's "constructor" directive that usually does specific initializations at time of rule creation. If it does not exist, then there is nothing to be done unless the grammar writer has defaulted to the compiler's initialization code for the class's locally defined variables — as they say in French désolé. For me this unspoken initialization is not good as it is implicit and i prefer being forthright to my coding intentions. Given this, a "reuse type" ctor must be defined within the rule's class containing the constructor directive's code if required and called inside the preliminaries of *reduce_rhs_of_rule* procedure for the specific rule.

124. Parse record.
Cparse_record defines the record of the parse stack. Due to my way of *cweb* source code ordering, type definitions come before structure definitions. In this case,the structure definition is outputted as a type definition instead of as a structure.

"abort__" adjusted to "void*" from "bool" as my optimization on stack frame of individual structures being multiples got slack bytes generated when porting to a HP Aplha. So make sure all are of same size. Put back to bool.

⟨ Type defs 16 ⟩ +≡
```
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__;
};
```

125. Lr parse stack structure.
Why the home grown stack — SPEED. Templates are toooo slow with to many generalities.

⟨ Type defs 16 ⟩ +≡
```
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__;
};
```

126. Parse stack implementation.

⟨ accrue yacco2 code 33 ⟩ +≡

```
lr_stk::lr_stk()
{
  top_sub__ = 1;
  first_sf__ = &lr_stk__[1];
  top__ = first_sf__;
  first_state__ = 0;
  top__→state__ = 0;
  top__→symbol__ = 0;
  top__→aborted__ = 0;
  top__→rule_s_reuse_entry_ptr__ = 0;
}
void lr_stk::lr_stk_init(yacco2::State &S1)
{
  top_sub__ = 1;
  first_sf__ = &lr_stk__[1];
  top__ = first_sf__;
  first_state__ = &S1;
  top__→state__ = first_state__;
  top__→symbol__ = 0;
  top__→aborted__ = 0;
  top__→rule_s_reuse_entry_ptr__ = 0;
}
bool lr_stk::empty()
{
  if (top_sub__ < 1) return true;
  return false;
}
void lr_stk::push_symbol(yacco2::CAbs_lr1_sym &Sym)
{
  top__→symbol__ = &Sym;
}
void lr_stk::pop()
{
  --top_sub__;
  --top__;
}
void lr_stk::clean_up()
{
  top_sub__ = 1;
  first_sf__ = &lr_stk__[1];
  top__ = first_sf__;
  top__→symbol__ = 0;
  top__→aborted__ = 0;
  top__→state__ = first_state__;
  top__→rule_s_reuse_entry_ptr__ = 0;
}
```

127. **lr_stk** :: *clean_up*( ). Speed demon.

⟨ **lr_stk** :: *clean_up*( ) 127 ⟩ ≡
```
top_sub__ = 1;
top__ = first_sf__;
top__→symbol__ = 0;
top__→aborted__ = 0;
top__→state__ = first_state__;
top__→rule_s_reuse_entry_ptr__ = 0;
```

128. **lr_stk** :: *empty*( ). Speed demon.

⟨ **lr_stk** :: **lr_stk** :: *empty*( ) 128 ⟩ ≡
```
if (top_sub__ < 1) return true;
return false;
```

129. **lr_stk** :: *pop*( ). Speed demon.

⟨ **lr_stk** :: *pop*( ) 129 ⟩ ≡
```
−− top_sub__;
−− top__;
```

130. Parse stack implementation.

⟨ accrue yacco2 code 33 ⟩ +≡
```
Cparse_record *lr_stk::sf_by_sub(yacco2::UINT Sub)
{
  if ((Sub < 1) ∨ (Sub > MAX_LR_STK_ITEMS)) {
    char a[BUFFER_SIZE];
    yacco2::KCHARP msg = "lr_stk␣-␣sf_by_sub␣invalid␣sub:␣%i␣not␣in␣range␣1..%i";
    sprintf(a, msg, Sub, MAX_LR_STK_ITEMS);
    Yacco2_faulty_precondition(a, __FILE__, __LINE__);
    exit(1);
  }
  return &lr_stk__[Sub];
}
Cparse_record *lr_stk::sf_by_top(yacco2::UINT No)
{
  int s = top_sub__ − No;
  if (s < 1) {
    char a[BUFFER_SIZE];
    yacco2::KCHARP msg = "lr_stk␣-␣sf_by_top␣underflow␣top␣sub:␣%i,␣requested␣sub:␣%i\
        ␣<␣1";
    sprintf(a, msg, top_sub__, No);
    Yacco2_faulty_precondition(a, __FILE__, __LINE__);
    exit(1);
  }
  return &lr_stk__[s];
}
```

131. Parse stack implementation.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void lr_stk::push_state(yacco2::State &S1)
{
  if (top_sub__ ≥ MAX_LR_STK_ITEMS) {
    char a[BUFFER_SIZE];
    yacco2::KCHARP msg = "lr_stk␣-␣push␣overflow␣stack␣max:␣%i";
    sprintf(a, msg, MAX_LR_STK_ITEMS);
    Yacco2_faulty_precondition(a, __FILE__, __LINE__);
    exit(1);
  }
  ++top__;
  ++top_sub__;
  top__→state__ = &S1;
  top__→symbol__ = 0;
  top__→aborted__ = 0;
  top__→rule_s_reuse_entry_ptr__ = 0;
}
```

132. **lr_stk**::*push_state* — Speed demon.

⟨ lr_stk::push_state 132 ⟩ ≡

```
if (parse_stack__.top_sub__ ≥ MAX_LR_STK_ITEMS) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "lr_stk␣-␣push␣overflow␣stack␣max:␣%i";
  sprintf(a, msg, MAX_LR_STK_ITEMS);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
++parse_stack__.top__;
++parse_stack__.top_sub__;
parse_stack__.top__→state__ = Goto_state;
parse_stack__.top__→symbol__ = 0;
parse_stack__.top__→aborted__ = 0;
parse_stack__.top__→rule_s_reuse_entry_ptr__ = 0;
```

This code is used in section 349.

133. ***set_aborted* and *aborted* implementation.**

The *set_aborted* tags the parse stack record. It is used in conjunction with the symbol's *affected_by_abort* attribute. That is, the parallel parse aborted and it is cleaning up the partial effects of the parse: the symbol indirectly dictates the what's to be done.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Cparse_record::set_aborted(bool X)
{
  aborted__ = X;
}
bool yacco2::Cparse_record::aborted() const
{
  if (aborted__ ≡ 0) return false;
  return true;
}
```

134. *set_rule_s_reuse_entr* **and** *rule_s_reuse_entry* **implementation.**
Used in the optimization of a rule's recycled symbol. It is the rule's subscript into the fsm's *rules_reuse_table*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Cparse_record::set_rule_s_reuse_entry(yacco2::Rule_s_reuse_entry *Rule_s_reuse)
{
  rule_s_reuse_entry_ptr__ = Rule_s_reuse;
}
yacco2::Rule_s_reuse_entry *yacco2::Cparse_record::rule_s_reuse_entry()
{
  return rule_s_reuse_entry_ptr__;
}
```

135. *set_state* **and** *state* **implementation.**

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Cparse_record::set_state(yacco2::State *State_ptr)
{
  state__ = State_ptr;
}
yacco2::State *yacco2::Cparse_record::state()
{
  return state__;
}
```

136. *set_symbol* **and** *symbol* **implementation.**

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::CAbs_lr1_sym *yacco2::Cparse_record::symbol()
{
  return symbol__;
}
void yacco2::Cparse_record::set_symbol(yacco2::CAbs_lr1_sym *Symbol)
{
  symbol__ = Symbol;
}
```

137. Thread support library: native thread wrapper functions.
Supports both Microsoft's NT platform thread implementation and Pthreads. Pthreads has been tested on HP's VMS operating system, Apple's OS X platform, Ubuntu, and Sun Solaris 10 AMD workstation. See "Pthreads Programming" by Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrel. Easy read and well presented 2nd edition 1998.

There is only one thread type: grammar requesting parallelism — 'pp' is its acromyn for parallel parse. From a parallel parsing perspective, the parsing pushdown automaton detects parallelism by the presence of the thread list within the current parse state's configuration. It now handles the all the details from launching of the threads instead of the old way that used a middleman called the control monitor "cm" who attended to all details related to parallel parsing and waited for the completion of the threads, and passed the results to the arbitrator functor for its ruling, and then cleaned up the accept queue.¿

To communicate between threads, a message protocol was developed in tandem with critical regions: I now call it an event protocol. Per thread, possession of its critical region is controlled by a mutex — mu for short. To implement messaging a conditional variable (cv) is used having a companion variable indicating whether a event is received or not that is under mu control.

The event (message) protocol was developed to remove any reliance on the operating system. I was caught by Microsoft's message queue system with its quirks, limitations, and down right tantrums. These comments are circa 1997 and probably don't hold today... but the system dependency still does so here's my take on parsing events. Simple and not too challenging intellectually.

To reduce the size of the emitted cpp file, the thread implementation is outputted to *wthread.cpp* file. It's definitions etc are concatenated to the **yacco2** . *h* file which is used by every implementation.

The following diagrams illustrates the critical region structure per thread, and the message flows acting as events between the threads.

Critical regions:

Message flow:

138. Set up the required include files.

⟨Include files 14⟩ +≡
#**if** THREAD_LIBRARY_TO_USE__ ≡ 1
#**include** <windows.h>
#**include** <process.h>
#**elif** THREAD_LIBRARY_TO_USE__ ≡ 0
#**include** <pthread.h>
#**endif**

139. Basic types supporting thread development.

⟨ Type defs 16 ⟩ +≡

```
  typedef void *LPVOID;
#if THREAD_LIBRARY_TO_USE__ ≡ 1
#define _YACCO2_CALL_TYPE    /* __stdcall */
  typedef HANDLEMUTEX;
  typedef unsigned int THREAD_NO;
  typedef HANDLE THREAD;
  typedef HANDLE COND_VAR;
  typedef uintptr_tTHR;
  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    /* _YACCO2_CALL_TYPE */
  * Type_pp_fnct_ptr_voidp)(yacco2::LPVOID PP_requestor);
#elif THREAD_LIBRARY_TO_USE__ ≡ 0
#define _YACCO2_CALL_TYPE
  typedef pthread_mutex_tMUTEX;
  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;
```

140. Thread's External wrapper routines.
Access to the real thread control runtime library uses wrapper routines to aid in porting to another thread library.

⟨ External rtns and variables 22 ⟩ +≡

```
  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();
```

141. Thread library implementation.
The wrapper functions shields the native library routines from Yacco2's callings. I call this a little middling sir...

Please note, there is no exit or destroy thread wrapper routines. This is done automaticly when the thread returns to the operating system. For the duration of the parse, the thread stays within a work loop until it receives an "exit" message and its work status has been changed to `THREAD_TO_EXIT` by the requesting shutdown process. See *Parallel_threads_shutdown* routine. The exit message just interrupts the thread to start executing whose work loop condition has been broken. Basic hygiene takes place by the exiting thread and then it exits to the operating system with an appropriate return code.

142. Microsoft's NT thread implementation.
⟨ accrue thread code 142 ⟩ ≡
#if `THREAD_LIBRARY_TO_USE__` ≡ 1

See also sections 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 174, 175, 177, 178, 179, 180, 185, and 212.

This code is used in section 169.

143. Create mutex — `CREATE_MUTEX`.
Appropriate defaults:

1) security: default
2) initial owner: OFF = no, ON = yes
3) named mutex: default 0 is no

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::CREATE_MUTEX(yacco2::MUTEX & Mu)
{
  Mu = CreateMutex(0, OFF, 0);
}
```

144. Lock mutex — `LOCK_MUTEX`.
⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::LOCK_MUTEX(yacco2::MUTEX & Mu)
{
  WaitForSingleObject(Mu, INFINITE);
}
```

145. Lock mutex — `LOCK_MUTEX_OF_CALLED_PARSER`.
⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::LOCK_MUTEX_OF_CALLED_PARSER(yacco2::MUTEX & Mu, yacco2::Parser
        &parser, const char *Text)
{
  ⟨ Trace trying to acquire grammar's mutex 606 ⟩;
  WaitForSingleObject(Mu, INFINITE);
  ⟨ Trace acquired grammar's mutex 607 ⟩;
}
```

146. Unlock mutex — `UNLOCK_MUTEX`.
⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::UNLOCK_MUTEX(yacco2::MUTEX & Mu)
{
  ReleaseMutex(Mu);
}
```

147. Unlock mutex — UNLOCK_MUTEX_OF_CALLED_PARSER.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::UNLOCK_MUTEX_OF_CALLED_PARSER(yacco2::MUTEX & Mu, yacco2::Parser
        &parser, const char *Text)
{
  ⟨ Trace trying to release grammar's mutex 608 ⟩;
  ReleaseMutex(Mu);
  ⟨ Trace released grammar's mutex 609 ⟩;
}
```

148. Destroy mutex — DESTROY_MUTEX.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::DESTROY_MUTEX(yacco2::MUTEX & Mu)
{
  CloseHandle(Mu);
}
```

149. Create conditional variable — CREATE_COND_VAR.
Default settings:

1) security: default 0
2) initial cnt: 0 so that it can wait for a signal
3) max cnt: 1 so that it's 1:1
4) make unnamed variable: 0

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::CREATE_COND_VAR(yacco2::COND_VAR &Cv)
{
  COND_VAR xx = CreateSemaphore(0, 0, 1, 0);     /* 0: wait state */
  Cv = xx;
}
```

150. Conditional wait — COND_WAIT.
Default settings:

unlock mutex
wait on cv
lock mu

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::COND_WAIT(yacco2::COND_VAR &Cv, yacco2::MUTEX & Mu, yacco2::Parser
        &parser)
{
  ⟨ trace COND_WAIT entered 648 ⟩;
  UNLOCK_MUTEX_OF_CALLED_PARSER(Mu, parser, "␣of␣self␣by␣COND_WAIT()");
  WaitForSingleObject(Cv, INFINITE);
  LOCK_MUTEX_OF_CALLED_PARSER(Mu, parser, "␣of␣self␣from␣wakened␣COND_WAIT()");
  ⟨ trace COND_WAIT exit 649 ⟩;
}
```

151. Signal conditional variable — `SIGNAL_COND_VAR`.
Default settings:

1) cond. var ptr
2) release count: make 1
3) previous cnt: 0 means don't use previous cnt: so make 1:1

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::SIGNAL_COND_VAR(yacco2::Parser &To_thread, yacco2::Parser &parser)
{
  ⟨ trace SIGNAL_COND_VAR before call 650 ⟩;
  ReleaseSemaphore(To_thread.cv__, 1, 0);
  ⟨ trace SIGNAL_COND_VAR after call 651 ⟩;
}
```

152. Destroy conditional variable — `DESTROY_COND_VAR`.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::DESTROY_COND_VAR(yacco2::COND_VAR &Cv)
{
  CloseHandle(Cv);
}
```

153. Create thread — `CREATE_THREAD`.
Default settings:

1) security: default 0
2) stack size: default 0
3) function addr
4) Parm list addr
5) initflag default 0: start executing right away
6) thread id addr

When the thread is created, within the defining code body of the thread is a canned include file *wpp_core.h*. Its code sets all the variables related to thread activation: caller's parse context and launched number of threads. *pp_requesting_parallelism__* is the calling parser and so is *from_thread__*. The *no_competing_pp_ths__* is set from the calling parser's *no_requested_ths_to_run__*. *no_requested_ths_to_run__* is a readonly variable used to optimize mutex access / release of the calling parser's critical region. If the value is 1, there is no need to use the mutex.

⟨ accrue thread code 142 ⟩ +≡
```
extern yacco2::THR_result yacco2::CREATE_THREAD(yacco2::Type_pp_fnct_ptr Thread,
        yacco2::Parser &Parser_requesting_parallelism)
{
  yacco2::THREAD_NO thread_no;
  ⟨ trace CREATE_THREAD before call 652 ⟩;
  THR result = _beginthreadex(0, 0, (Type_pp_fnct_ptr_voidp) Thread, &Parser_requesting_parallelism, 0,
      &thread_no);
  ⟨ trace CREATE_THREAD after call 653 ⟩;
  return result;
}
```

154. Thread id — THREAD_SELF.

⟨ accrue thread code 142 ⟩ +≡
```
extern yacco2::THREAD_NO yacco2::THREAD_SELF()
{
  return GetCurrentThreadId();
}
```

155. Pthreads implementation.

⟨ accrue thread code 142 ⟩ +≡
```
#elif THREAD_LIBRARY_TO_USE__ ≡ 0
```

156. Create Mutex — CREATE_MUTEX.

When the thread is created, within the defining code body of the thread is a canned include file *wpp_core.h*. Its code sets all the variables related to thread activation: caller's parse context and launched number of threads. *pp_requesting_parallelism__* is the calling parser and so is *from_thread__*. The *no_competing_pp_ths__* is set from the calling parser's *no_requested_ths_to_run__*. *no_requested_ths_to_run__* is a readonly variable used to optimize mutex access / release of the calling parser's critical region. If the value is 1, there is no need to use the mutex.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::CREATE_MUTEX(yacco2::MUTEX & Mu)
{
  int result = pthread_mutex_init(&Mu, 0);
}
```

157. Lock mutex — LOCK_MUTEX.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::LOCK_MUTEX(yacco2::MUTEX & Mu)
{
  int result = pthread_mutex_lock(&Mu);
}
```

158. Lock mutex — LOCK_MUTEX_OF_CALLED_PARSER.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::LOCK_MUTEX_OF_CALLED_PARSER(yacco2::MUTEX & Mu, yacco2::Parser
        &parser, const char *Text)
{
  ⟨ Trace trying to acquire grammar's mutex 606 ⟩;
  int result = pthread_mutex_lock(&Mu);
  ⟨ Trace acquired grammar's mutex 607 ⟩;
}
```

159. Unlock mutex — UNLOCK_MUTEX.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::UNLOCK_MUTEX(yacco2::MUTEX & Mu)
{
  int result = pthread_mutex_unlock(&Mu);
}
```

160. Unlock mutex — UNLOCK_MUTEX_OF_CALLED_PARSER.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::UNLOCK_MUTEX_OF_CALLED_PARSER(yacco2::MUTEX &Mu, yacco2::Parser
        &parser, const char *Text)
{
  ⟨ Trace trying to release grammar's mutex 608 ⟩;
  int result = pthread_mutex_unlock(&Mu);
  ⟨ Trace released grammar's mutex 609 ⟩;
}
```

161. Destroy mutex — DESTROY_MUTEX.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::DESTROY_MUTEX(yacco2::MUTEX &Mu)
{
  int result = pthread_mutex_destroy(&Mu);
}
```

162. Create conditional variable — CREATE_COND_VAR.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::CREATE_COND_VAR(yacco2::COND_VAR &Cv)
{
  pthread_cond_init(&Cv, 0);
}
```

163. Conditional wait — COND_WAIT.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::COND_WAIT(yacco2::COND_VAR &Cv, yacco2::MUTEX &Mu, yacco2::Parser
        &parser)
{
  ⟨ trace COND_WAIT entered 648 ⟩;
  if (yacco2::YACCO2_MU_GRAMMAR__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog << parser.thread_no__ << "::" << parser.fsm_tbl__->id__ << "::" <<
        "␣before␣release␣mutex␣by␣pthread_cond_wait()" << __FILE__ << __LINE__ << std::endl;
    ⟨ release trace mu 390 ⟩;
  }
  pthread_cond_wait(&Cv, &Mu);
  ⟨ trace COND_WAIT exit 649 ⟩;
}
```

164. Signal conditional variable — SIGNAL_COND_VAR.

⟨ accrue thread code 142 ⟩ +≡

```
extern void yacco2::SIGNAL_COND_VAR(yacco2::Parser &To_thread, yacco2::Parser &parser)
{
  ⟨ trace SIGNAL_COND_VAR before call 650 ⟩;
  pthread_cond_signal(&To_thread.cv__);
  ⟨ trace SIGNAL_COND_VAR after call 651 ⟩;
}
```

165. Destroy conditional variable — DESTROY_COND_VAR.

⟨ accrue thread code 142 ⟩ +≡
```
extern void yacco2::DESTROY_COND_VAR(yacco2::COND_VAR &Cv)
{
  pthread_cond_destroy(&Cv);
}
```

166. Create thread — CREATE_THREAD. Experimenting with thread attributes by use of *pthread_attr_t* object and its methods: *pthread_attr_setstacksize*. If u want the default, pass *null* in the 2nd argument in *pthread_create*. This experiment is caused by VMS's tantrums when porting *pasxlator* translator to the Alpha platform. Circa 2002 – 2003, this worked under VMS 7.2 and their older C++ compiler 6.5.

⟨ accrue thread code 142 ⟩ +≡
```
extern yacco2::THR_result
yacco2::CREATE_THREAD
(yacco2::Type_pp_fnct_ptr Thread, yacco2::Parser &Parser_requesting_parallelism)
{
  ⟨ trace CREATE_THREAD before call 652 ⟩;
  yacco2::THREAD_NO thread_no;
  pthread_attr_t alpha_attr;
  pthread_attr_init(&alpha_attr);
#ifdef VMS__
  pthread_attr_setstacksize(&alpha_attr, VMS_PTHREAD_STACK_SIZE__);
#endif
  THR_result result = pthread_create(&thread_no, &alpha_attr, (Type_pp_fnct_ptr_voidp) Thread,
      &Parser_requesting_parallelism);
  pthread_detach(thread_no);
  ⟨ trace CREATE_THREAD after call 653 ⟩;
  return result;
}
```

167. Thread id — THREAD_SELF.

⟨ accrue thread code 142 ⟩ +≡
```
extern yacco2::THREAD_NO yacco2::THREAD_SELF()
{
  return pthread_self();
}
```

168. Close off the wrapper conditional code.

⟨ accrue thread code 142 ⟩ +≡
```
#endif
```

169. Yacco2's internal thread implementation.

⟨ `wthread.cpp` 169 ⟩ ≡
⟨ copyright notice 565 ⟩;
⟨ iyacco2 26 ⟩;
⟨ accrue thread code 142 ⟩;

170. Thread control runtime environment.
Thread control record for the thread pool table. This is used by Yacco2's global runtime table of spawned threads. This is a one-to-many relationship as the same thread can be running within a nested call chain. Very basic in its thread *worker_status*: working, waiting for work, and I'm out of here.

⟨ Type defs 16 ⟩ +≡
```
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;
```

171. worker_thread_blk structure.
grammar_s_parser__ is the grammar's parser. *status__* takes one of 3 states:

1) `THREAD_WAITING_FOR_WORK`
2) `THREAD_WORKING`
3) `THREAD_TO_EXIT`

Of import:
When the thread gets created, **worker_thread_blk** will enter the thread into the global thread table list. The table is a vector of precalculated thread numbers generated from Yacco2's linker. The launching grammar has mutual access to *Parallel_thread_table*. So the created thread can just deposit its **worker_thread_blk** address into the list.

⟨ Structure defs 18 ⟩ +≡
```
struct worker_thread_blk {
  worker_thread_blk( );     /* monolithic grammar */
  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( );
};
```

172. Global *Parallel_thread_table* declaration of use.
Maintains a list of launched threads with their availability. For efficiency, it is an array subscripted by the thread's id number. Why the list? This is a 1:m situation. Due to nested thread calls, a thread could be busy so another copy of the threads needs creation.

⟨ Global variables 21 ⟩ +≡
```
extern Parallel_thread_tbl_type Parallel_thread_table;
extern Parallel_thread_proc_call_table_type Parallel_thread_proc_call_table[MAX_NO_THDS];
```

173. Global routines declaration of use.

⟨ External rtns and variables 22 ⟩ +≡
```
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;
```

174. Global *Parallel_thread_table* definition.

⟨ accrue thread code 142 ⟩ +≡
```
yacco2::Parallel_thread_tbl_type yacco2::Parallel_thread_table(MAX_NO_THDS);
yacco2::Parallel_thread_proc_call_table_type yacco2::Parallel_thread_proc_call_table[MAX_NO_THDS];
```

175. Global Proxy arbitrator.
Used for manual parallelism. This is manually launched by the grammar writer's code within a grammar.

⟨ accrue thread code 142 ⟩ +≡
```
extern yacco2::THR _YACCO2_CALL_TYPE
yacco2::AR_for_manual_thread_spawning(yacco2::Parser *Caller_pp)
{
  std::string ar_name("AR_yacco2");
  ⟨ iar begin 30 ⟩;
  ⟨ No arbitration code present 176 ⟩;
  ⟨ iar end 31 ⟩;
}
```

176. No arbitration code present.
This condition exists when the accept queue has more than 1 accept token in the queue. What token should be accepted while the others are quitely put to heaven? Within Yacco2, it checks when the configuration state has more than 1 thread being launched, and there is no grammar writer code to select the winning token, before the throw code is emitted. Determining how the select code is present is currently crude. It checks to see that the *pp_accept_queue__* variable is present in the syntax directed code string: not present then emit the conditional wrapping of the throw condition.

⟨ No arbitration code present 176 ⟩ ≡
```
if (Caller_pp→th_accepting_cnt__ > 1) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "no␣arbitration␣code␣present␣in␣%s␣-␣accept␣token␣queue␣has␣\
      %i␣>␣1␣tokens␣to␣arbitrate␣on";
  sprintf(a, msg, ar_name.c_str(), Caller_pp→th_accepting_cnt__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```
This code is used in section 175.

177. worker_thread_blk initialization: monolithic grammar.
Part of its duties is to create the mutexs controling Yacco2's tables: symbol and thread list. To serialize traced output, a mutex is used to throatle back simultaneous multi-threads tracing into a single queue of buffer flush-out. STL does not control this. It is at the mercy of how threads are executed and how the operating system tic-tacs the clock and their output. Due to this whimsy of clock soundings, you can receive from different threads interspersed mixed snippets of traced code on the same line outputted. This is why all atomic traces are bracketed by the acquire / release of the trace mutex.

The mutex creation is done by the birth of a grammar object: each grammar contains a **Parser** component containing a **worker_thread_blk**. So there is no need for a special startup routine to use Yacco2's library.

⟨ accrue thread code 142 ⟩ +≡
```
yacco2::worker_thread_blk::worker_thread_blk()    /* monolithic grammar */
: grammar_s_parser__(0), status__(0), run_cnt__(1), thd_id__(0) {
  static bool init_gbl(OFF);
  if (init_gbl ≡ OFF) {
    init_gbl = ON;
    CREATE_MUTEX(yacco2::TH_TBL_MU);
    CREATE_MUTEX(yacco2::TRACE_MU);
    CREATE_MUTEX(yacco2::TOKEN_MU);
    CREATE_MUTEX(yacco2::SYM_TBL_MU);
  }
}
```

178. worker_thread_blk initialization: threaded grammar.
See HP*Alpha.CPLUSPLUS*/"**this**" *object mis* − *address* describing bug. It provides the reason for the change from *i.push_back*(**this**) to *i.push_back*(&*Grammar_s_parser*→*th_blk__*). ⟨ acquire global thread table critical region 380 ⟩ and ⟨ release global thread table critical region 381 ⟩ are not used in this context as the grammar requesting the threads to run has already acquired it!

⟨ accrue thread code 142 ⟩ +≡
```
yacco2::worker_thread_blk::worker_thread_blk(yacco2::Parser *Grammar_s_parser,
    yacco2::Parser *Calling_parser)    /* parallel grammar */
: grammar_s_parser__(Grammar_s_parser), status__(THREAD_WAITING_FOR_WORK), run_cnt__(1),
      thd_id__(grammar_s_parser__→thread_entry__→thd_id__) {
  status__ = THREAD_WORKING;
  Parallel_thread_list_type &i = Parallel_thread_table[grammar_s_parser__→thread_entry__→thd_id__];
  i.push_back(this);
  ⟨ Trace MSG thread being created 618 ⟩;
}
```

179. *set_waiting_for_work*.
It is the running thread who sets its own work status. Both ⟨ acquire global thread table critical region 380 ⟩ and ⟨ release global thread table critical region 381 ⟩ are used by the running thread in their local procedures *parallel_parse_successful* or *parallel_parse_unsuccessful*.

⟨ accrue thread code 142 ⟩ +≡
```
void yacco2::worker_thread_blk::set_waiting_for_work()
{
  ⟨ Trace MSG thread idle before setting waiting for work 616 ⟩;
  status__ = THREAD_WAITING_FOR_WORK;
  ⟨ Trace MSG thread idle after setting waiting for work 617 ⟩;
}
```

180. Global shutdown of threads.
Goes through the list of threads. Before doing 2 passes on the table, the routine pauses for x seconds to let the swamp drain: due to a single processor environment, there could still be threads outstanding in their winddown to-wait-for-work sequence. It then goes thru the thread list for threads waiting-for-work, these threads are given their pink notice.

The last pause is to allow the draining of the threads' output: flush those buffers. The 2nd pass thru the table is a sanity check. Any threads still outstanding are listed to Yacco2's output file *lrclog*. This notification allows the compiler writer to check out why.

```
⟨accrue thread code 142⟩ +≡
  extern void yacco2::Parallel_threads_shutdown(yacco2::Parser &PP)
  {
    ⟨acquire global thread table critical region 380⟩;
    int no_thds_to_shutdown(0);
    int no_ths_exited(0);
    ⟨pause for x seconds 181⟩;    /* let the other threads go into a wait state */
    ⟨Threads in table to potentially shutdown 182⟩;
    ⟨look for threads to shutdown 183⟩;
    ⟨pause for x seconds 181⟩;    /* allow the threads to close down */
    ⟨release global thread table critical region 381⟩;
    DESTROY_MUTEX(yacco2::TH_TBL_MU);
    DESTROY_MUTEX(yacco2::TRACE_MU);
    DESTROY_MUTEX(yacco2::TOKEN_MU);
    DESTROY_MUTEX(yacco2::SYM_TBL_MU);
  }
```

181. Pause for x seconds.

```
⟨pause for x seconds 181⟩ ≡
#if THREAD_LIBRARY_TO_USE__ ≡ 1
  Sleep(1000);
#elif THREAD_LIBRARY_TO_USE__ ≡ 0
  sleep(1);    /* from guy steele c ref bk, in seconds. */
#endif
```

This code is cited in section 110.

This code is used in section 180.

182. Threads in table to potentially shutdown.

```
⟨ Threads in table to potentially shutdown 182 ⟩ ≡
  Parallel_thread_tbl_iterator_type k = Parallel_thread_table.begin( );
  Parallel_thread_tbl_iterator_type ke = Parallel_thread_table.end( );
  for ( ; k ≠ ke; ++k) {
    Parallel_thread_list_iterator_type m = k→begin( );
    Parallel_thread_list_iterator_type me = k→end( );
    for ( ; m ≠ me; ++m) {
      ++no_thds_to_shutdown;
    }
  }
  yacco2::lrclog ≪ "Number␣of␣threads␣in␣table␣to␣shutdown:␣" ≪ no_thds_to_shutdown ≪
      __FILE__ ≪ __LINE__ ≪ std::endl;
  k = Parallel_thread_table.begin( );
  for ( ; k ≠ ke; ++k) {
    Parallel_thread_list_iterator_type m = k→begin( );
    Parallel_thread_list_iterator_type me = k→end( );
    for ( ; m ≠ me; ++m) {
      worker_thread_blk *tb = *m;
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "worker␣task␣in␣table␣tb*:␣" ≪ tb ≪ "␣thread␣id:␣" ≪
          tb→grammar_s_parser__→thread_no__ ≪ "::" ≪ tb→grammar_s_parser__→thread_name( ) ≪
          "␣run␣cnt:␣" ≪ tb→run_cnt__;
      switch (tb→status__) {
      case THREAD_WAITING_FOR_WORK:
        {
          yacco2::lrclog ≪ "␣waiting␣for␣work";
          break;
        }
      case THREAD_WORKING:
        {
          yacco2::lrclog ≪ "␣working";
          break;
        }
      case THREAD_TO_EXIT:
        {
          yacco2::lrclog ≪ "␣thread␣to␣exit";
          break;
        }
      default:
        {
          yacco2::lrclog ≪ "␣???␣thread␣status:␣" ≪ tb→status__;
          break;
        }
      }
      yacco2::lrclog ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
```

This code is used in section 180.

183. Look for threads to shutdown.

```
⟨ look for threads to shutdown 183 ⟩ ≡
  Parallel_thread_tbl_iterator_type i = Parallel_thread_table.begin( );
  Parallel_thread_tbl_iterator_type ie = Parallel_thread_table.end( );
  for ( ; i ≠ ie; ++i) {
    Parallel_thread_list_iterator_type j = i→begin( );
    Parallel_thread_list_iterator_type je = i→end( );
    for ( ; j ≠ je; ++j) {
      worker_thread_blk *tb = *j;
      if (tb→status__ ≡ THREAD_WAITING_FOR_WORK) {
        ⟨ acquire trace mu 389 ⟩;
        ++no_ths_exited;
        yacco2::lrclog ≪ "worker␣task␣to␣exit:␣" ≪ tb→grammar_s_parser__→thread_no__ ≪ "::" ≪
            tb→grammar_s_parser__→thread_name( ) ≪ "␣tb*␣" ≪ tb ≪ __FILE__ ≪ __LINE__ ≪
            std::endl;
        ⟨ release trace mu 390 ⟩;
        LOCK_MUTEX_OF_CALLED_PARSER(tb→grammar_s_parser__→mu__, *tb→grammar_s_parser__,
            "␣of␣called␣thread");
        tb→status__ = THREAD_TO_EXIT;
        PP.post_event_to_requesting_grammar(*tb→grammar_s_parser__, Shutdown, PP);
      }
      else {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "worker␣task␣not␣shutting␣down:␣" ≪ tb→grammar_s_parser__→thread_no__ ≪
            "::" ≪ tb→grammar_s_parser__→thread_name( ) ≪ "␣tb*␣" ≪ tb ≪ "␣status:␣" ≪
            tb→status__ ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
    }
    i→clear( );
  }
  Parallel_thread_table.clear( );
  yacco2::lrclog ≪ "Number␣of␣threads␣in␣table␣exiting:␣" ≪ no_ths_exited ≪
      "␣number␣of␣threads␣not␣shutting␣down:␣" ≪ no_thds_to_shutdown − no_ths_exited ≪
      __FILE__ ≪ __LINE__ ≪ std::endl;
```

This code is used in section 180.

184. *Caccept_parse* **Structure — Accept result from threads.**
Ahh, the smell of ??? Go tell it to cm. Jess the reality show syndrome. This message gets put into the accept queue of the requesting pp. This is a potential winner requiring the arbitrator to decide. Lets hope the judge is not of TVQ 'star acadamie' tabloids variety.

Changed the *accept_queue* from a mapped sturcture of keyed by the accept terminal's enumeration id to one of sequential list of local *Caccept_parse*. As non-determinism is small: potentially 2 or 3 occassionally Tes in the queue, i felt the sequential attitude appropriate instead of a mapped structure. The big improvement is to remove malloced *Caccept_parse* and use the copy into the local *Caccept_parse* of the accept queue.

⟨Structure defs 18⟩ +≡

```
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__;
};
```

185. Caccept_parse and ~Caccept_parse implementation.

⟨ accrue thread code 142 ⟩ +≡
```
yacco2::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)
{
  th_reporting_success__ = &Th_reporting_success;
  accept_token__ = &Accept_token;
  accept_token_pos__ = Accept_token_pos;
  la_token__ = &La_token;
  la_token_pos__ = La_token_pos;
}
yacco2::Caccept_parse::
Caccept_parse()
{
  th_reporting_success__ = 0;
  accept_token__ = 0;
  accept_token_pos__ = 0;
  la_token__ = 0;
  la_token_pos__ = 0;
}
void yacco2::Caccept_parse::initialize_it()
{
  th_reporting_success__ = 0;
  accept_token__ = 0;
  accept_token_pos__ = 0;
  la_token__ = 0;
  la_token_pos__ = 0;
}
void yacco2::Caccept_parse::fill_it(Caccept_parse &Accept_parse)
{
  th_reporting_success__ = Accept_parse.th_reporting_success__;
  accept_token__ = Accept_parse.accept_token__;
  accept_token_pos__ = Accept_parse.accept_token_pos__;
  la_token__ = Accept_parse.la_token__;
  la_token_pos__ = Accept_parse.la_token_pos__;
}
void yacco2::Caccept_parse::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)
{
  th_reporting_success__ = &Th_reporting_success;
  accept_token__ = &Accept_token;
```

$accept\_token\_pos\_\_ = Accept\_token\_pos;$
$la\_token\_\_ = \&La\_token;$
$la\_token\_pos\_\_ = La\_token\_pos;$
}

yacco2 :: **Caccept_parse** :: ~**Caccept_parse**( )
{ }

186. Thread code for arbitrator, and parallel parse.
The emitted files become the include files for the emitted threads and each finite automton's arbitrator. For the parallel parse thead, this is the core code loops that make it tick. The arbitrator code is the two pieces of bread that sandwich the grammar writer's selection code supplied from the *arbitrator − code* construct. The produced files are:

1) *wpp_core.cpp* — parallel parser include code for generated pp threads
2) *war_begin_code.h* — arbitrator's start code
3) *war_end_code.h* — arbitrator's end code

187. Arbitrator code generator — begin and end files: ***war_xxx_code.h*.**
The emitted code is the *pp_accept_queue*'s iteration to walk thru the potential tokens for consideration produced by the parallel threads inserted into the requesting grammar's accept queue. It is structured into 2 parts:

1) the startup variables to iterate thru the accept queue
2) the ending code of the iteration

Sandwiched between these 2 pieces of code is the arbitration logic supplied by the grammar writer that gets emitted for that specific state's configuration. Normally there is no code as the parallel request is deterministic with at most only one token returned by one of the launched threads.

188. Arbitrator begin code.
This is injected into the emitted arbitrators produced by Yacco2. The grammar writer's code follows this code. It is the discrimatory code used to select the winning accept terminal within the accept queue.

Arbitration is needed when there are competing parallel parses that return their accept terminals. A single entry only is checked first and returned before going into the arbitrated code selection. A sanity check is done on the accept queue whereby the accepted thread count **must equal** the number of accepted tokens placed into the queue.

The *Caller_pp* variable is the passed Parser pointer argument to the arbitration routine. It is the parser's context that includes the its critcal region supporting threading and the accept queue. Arbitration routine(s) generated out of the grammar have the following naming convention:

`AR_` concatenated with the rule name

An example of a routine is:

yacco2::THR `_YACCO2_CALL_TYPE` *NS_pass3* :: *AR_Rtok*(yacco2::Parser* *Caller_pp*);

The `_YACCO2_CALL_TYPE` is an internal definition specific to Microsoft call types. It is defined as *__stdcall* whereas in the other supported platforms it's value is empty.

⟨`war_begin_code.h` 188⟩ ≡
⟨copyright notice 565⟩;
⟨pp accept queue *war_begin_code* 189⟩;

189. ⟨pp accept queue *war_begin_code* 189⟩ ≡
⟨uns 23⟩;
int $i = 1$;
int $ie = Caller\_pp \rightarrow th\_accepting\_cnt\_\_$;
⟨Trace AR trace the starting of arbitration 625⟩;
This code is used in section 188.

190. Example of arbitrated grammar code.
The accept queue is sequentially searched in arbitrating on the enumerated id of the potential accepting Tes. The following example only gets executed when there are 2 or more accepting terminals in the queue. In this example, there are 2 independent parallelisms going on:

keyword versus identifier
floating point versus integer

They never intersect!

```
 1: ,parallel-control-monitor{
 2:   arbitrator-code
 3:     // arbitration
 4:     //              between
 5:     //      x               y                       winner
 6:     //      identifier      keyword                 keyword
 7:     //      fp no           integer                 fp no
 8:     //
 9:     using namespace NS_pas_T_enum;
10:     {
11:      for(i=1;i<=ie;++i){
12:       if(To_judge->pp_accept_queue__[i].accept_token__->enumerated_id__
13:              == NS_pas_T_enum::T_Enum::T_T_keyword_){
14:         goto arbitrated_parameter;
15:       }
16:      }
17:      for(i=1;i<=ie;++i){
18:       if(To_judge->pp_accept_queue__[i].accept_token__->enumerated_id__
19:              == NS_pas_T_enum::T_Enum::T_T_fp_pt_no_){
20:         goto arbitrated_parameter;
21:       }
22:      }
23:     }
24:   ***
25:   }
26:
```

Lines 11 and 12 above show 3 things:

1) i is the subscript to accept parse array's current contents
2) *Caller_pp* (Parser*) points to the critical region of the grammar
3) *pp_accept_queue__* contains the parallel results from the threads

The decision code only gets executed if there are 2 or more terminals placed into the accept queue for arbitration. This case is very rare but the above example illustrates dealing with non-determinism from 2 or more successful parallel parses. How can this come about?: Subset - superset — common prefixes. The example gives 2 examples of this that are tested for. The integer recognizes the whole number while the floating point continues with the fraction. One can argue that the grammar strategy was not very refined as the lookahead on the integer should not accept ".". You're right but this example is instructive and it was drawn from a real translator that was put together quickly. The moral is: u can be inefficient but effective with non-determinism.

Note, the items placed into the accept queue can contain error terminals forwarded to the calling grammar to do its own abort sequence.

191. Arbitrator end code.
Closes the iteration thru the accept queue. Originally i optimized injection code in case the grammar writer missed selecting the accepted T. This code was dependent on whether the specific state had multiple threads to launch. Now for clarity i have included a stopper procedure before the *arbitrated_parameter* label whereby it spews the gory details for the grammar writer's logic correction: Competing threads within the grammar have their names displayed while a thread with a "NULL" name is not a competing thread but allows one to be specific to an accepting token returned by one of the named threads.

Where is the accept queue drained of its contents? As potential terminals for arbitration are birthed from malloc (new), their sending to heaven should be epiphaned by "delete". This is done by the generic Parser code just after the call to the "Arbitrator". This is a code-bloat diet: Putting this in each generated arbitrator routine across all grammars would have been fat people community like the works of Spanish sculptor/painter Botero.

⟨ `war_end_code.h` 191 ⟩ ≡
  ⟨ copyright notice 565 ⟩;
  ⟨ pp accept queue *war_end_code* 192 ⟩;

192. ⟨ pp accept queue *war_end_code* 192 ⟩ ≡
  Caller_pp→*abort_no_selected_accept_parse_in_arbitrator*( );
arbitrated_parameter:
  Caller_pp→*arbitrated_token__* = &*Caller_pp*→*pp_accept_queue__*[*i*];
  Caller_pp→*pp_accept_queue_idx__* = *i*;
  ⟨ Trace AR stopped arbitrating 629 ⟩;
  return (**THR**) 1;

This code is used in section 191.

193. Parallel thread code: injection code for emitted pp *wpp_core.h***.**
This is the injector code for the manufactured parallel thread. Drawn from the just created file *wpp_core.h*. If it has been launched as a thread, "waiting-for-work" has been removed from the run loop and placed in the responding *parallel_parse_successful* and *parallel_parse_unsuccessful* procedures. This is an optimization: Ahhh the dragon trace of threading...

Even better is the check as to calling it as a thread or as a procedure. This depends on the number of threads to launch. If there is only one thread to run, this is called as a procedure instead of a thread. Do u see the friskiness in Yacco2? Well no, as threads now dominate.

Please see "Notes to myself" on running diatribe regarding optimization.

```
⟨wpp_core.h 193⟩ ≡
  ⟨copyright notice 565⟩;
  ⟨uns 23⟩;
  ⟨create communication variables 200⟩;
  ⟨create parser related variables and set them 202⟩;
  ⟨set parameter passed to pp as a message 201⟩;
  do {
    ⟨establish initial parser's token setting 199⟩;
    ⟨Trace pp start info 637⟩;
    ⟨let's parallel parse. do u? 198⟩;
    ⟨Trace stop of parallel parse message 639⟩;
    ⟨clean up parse stack but leave as ready to parse again 197⟩;
    ⟨house clean the parser and local communication variables 196⟩;
    ⟨Trace parallel thread waiting-to-do-work 642⟩;
    ⟨pp wait for work or shutdown message 195⟩;
    ⟨Trace pp received go start working message 643⟩;
  } while (pp_parser.th_blk__.status__ ≠ THREAD_TO_EXIT);
finished_working:
  ⟨winddown duties of pp 194⟩;
  ⟨Trace pp finished working 644⟩;
  UNLOCK_MUTEX_OF_CALLED_PARSER(pp_parser.mu__, pp_parser, "␣of␣called␣thread");
  return (THR) 1;
```

194. Winddown duties of pp.

```
⟨winddown duties of pp 194⟩ ≡
  pp_parser.clear_parse_stack();
```

This code is used in section 193.

195. Pp wait for work or shutdown message.

```
⟨pp wait for work or shutdown message 195⟩ ≡
  pp_parser.wait_for_event();
```

This code is used in section 193.

196. House clean the parser and local communication variables.
Their procedure calls replaced for speed.

```
⟨house clean the parser and local communication variables 196⟩ ≡
  pp_parser.use_all_shift__ = ON;
  pp_parser.abort_parse__ = OFF;
  pp_parser.stop_parse__ = OFF;
  pp_parser.has_questionable_shift_occured__ = OFF;
```

This code is used in section 193.

197. Clean up parse stack but leave as ready to parse again. The following points are done:
1) clean up trace activity: normally done when parse object destroyed
2) leave first record on stack for efficiency
3) make sure first stack symbol on stack checked for delete attribute

⟨clean up parse stack but leave as ready to parse again 197⟩ ≡
```
pp_parser.remove_from_stack(pp_parser.parse_stack__.top_sub__ − 1);
CAbs_lr1_sym *sym = pp_parser.top_stack_record()→symbol__;
if (sym ≠ 0) {
  if (sym→auto_delete__ ≡ ON) {
    ⟨Trace pp's last symbol on stack set as autodelete 640⟩;
    delete sym;
  }
  pp_parser.top_stack_record()→set_symbol(0);    /* keeping a clean stack */
}
pp_parser.parse_stack__.lr_stk_init(*pp_parser.fsm_tbl__→start_state__);
```
This code is used in section 193.

198. Let's parallel parse. do u?.

⟨let's parallel parse. do u? 198⟩ ≡
```
pp_parser.parallel_parse();
```
This code is cited in section 272.

This code is used in section 193.

199. Establish initial parser's token setting. When the thread is established and waiting to be wakenned, the calling grammar sets the following variables within the critical region of the called thread: *from_thread__*, *pp_requesting_parallelism__*, and *no_competing_pp_ths__*.

⟨ establish initial parser's token setting 199 ⟩ ≡

```
pp_parser.override_current_token(*pp_parser.pp_requesting_parallelism__→current_token(),
      pp_parser.pp_requesting_parallelism__→current_token_pos__);
pp_parser.set_start_token(*pp_parser.pp_requesting_parallelism__→current_token());
pp_parser.set_start_token_pos(pp_parser.pp_requesting_parallelism__→current_token_pos__);
pp_parser.top_stack_record()→set_symbol(pp_parser.current_token());
pp_parser.token_supplier__ = pp_parser.pp_requesting_parallelism__→token_supplier__;
pp_parser.token_producer__ = pp_parser.pp_requesting_parallelism__→token_producer__;
pp_parser.error_queue__ = pp_parser.pp_requesting_parallelism__→error_queue__;
pp_parser.recycle_bin__ = pp_parser.pp_requesting_parallelism__→recycle_bin__;
pp_parser.sym_lookup_functor__ = pp_parser.pp_requesting_parallelism__→sym_lookup_functor__;
pp_parser.supplier_r_w_cnt__ = pp_parser.pp_requesting_parallelism__→supplier_r_w_cnt__;
if (pp_parser.th_blk__.grammar_s_parser__ ≠ &pp_parser) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "parser's␣thd␣blk's␣pp␣addr␣!=␣itself␣thd:␣%i::%s";
  sprintf(a, msg, pp_parser.thread_no__, pp_parser.thread_name());
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
if (pp_parser.th_blk__.grammar_s_parser__→pp_requesting_parallelism__ ≠
      pp_parser.pp_requesting_parallelism__) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "caller's␣pp␣addr␣not␣=␣in␣called␣parser's␣thd␣blk␣ptr,␣and␣\
      its␣parser␣thd:␣%i::%s";
  sprintf(a, msg, pp_parser.thread_no__, pp_parser.thread_name());
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 193.

200. Create communication variables.

⟨ create communication variables 200 ⟩ ≡

```
char ma[SMALL_BUFFER_4K];
const char *pp_start = "YACCO2_MSG__::%i::%s␣start␣parsing\n";
const char *pp_stop = "YACCO2_MSG__::%i::%s␣stop␣parsing\n";
⟨ uns 23 ⟩;
```

This code is used in section 193.

201. Set parameter passed to pp as a message.

⟨ set parameter passed to pp as a message 201 ⟩ ≡

```
pp_parser.pp_requesting_parallelism__ = Caller_pp;
pp_parser.from_thread__ = Caller_pp;
pp_parser.no_competing_pp_ths__ = Caller_pp→no_requested_ths_to_run__;
```

This code is used in section 193.

202. Create parser related variables and set them.

⟨ create parser related variables and set them 202 ⟩ ≡
Parser *pp_parser*(*ssPARSE_TABLE*, *pp_thread_entry*, *Caller_pp*);

This code is used in section 193.

203. Procedure call: injection code for emitted pp *wproc_pp_core.h*.
This is the injector code for the manufactured called procedure instead of a thread. Even better is the check as to calling it as a thread or as a procedure. This depends on the number of threads to launch. If there is only one thread to run, this is called as a procedure instead of a thread. Do u see the friskiness in Yacco2? Well no, as threads now dominate.

Added improvements:
A `|t|` contruct has been added to do chained procedure calls: the 1st thread's returned T becomes the chained T for the next (chained) procedure call. I overloaded this symbol to support 2 contexts: O_2^{linker} and chained parsing calls. Why the overload? I only have 8 symbols reserved for the *LRk* symbol class and one context does not interfer with the other so i'm a bit lazy to possibly remove *eof* and double duty *eog* symbol where the file processing container templates us *eof*. Some parsing adjustments must be added to link the chained T with the chained procedure call as the the chained procedure must reference the shifted T of the calling parser as its start T and not the current T of the calling parser. *proc_call_funct__* has been added to the State's definition to support the chained call.

⟨ `wproc_pp_core.h` 203 ⟩ ≡
⟨ copyright notice 565 ⟩;
⟨ uns 23 ⟩;
⟨ create procedure communication variables 209 ⟩;
⟨ set procedure parameter passed to pp as a message 210 ⟩;
⟨ establish procedure initial parser's token setting 208 ⟩;
⟨ Trace procedure pp start info 638 ⟩;
⟨ let's procedure parallel parse. do u? 207 ⟩;
⟨ clean up procedure parse stack but leave as ready to parse again 206 ⟩;
⟨ house clean procedure the parser and local communication variables 205 ⟩;
finished_working:
⟨ winddown duties of procedure pp 204 ⟩;
⟨ Trace procedure pp finished working 645 ⟩;
return *rslt*;

204. Winddown duties of procedure pp.

⟨ winddown duties of procedure pp 204 ⟩ ≡
proc_parser→*clear_parse_stack*( );

This code is used in section 203.

205. House clean procedure the parser and local communication variables.

⟨ house clean procedure the parser and local communication variables 205 ⟩ ≡
proc_parser→*set_use_all_shift_on*( );
proc_parser→*set_abort_parse*(`OFF`);
proc_parser→*set_stop_parse*(`OFF`);
proc_parser→*has_questionable_shift_occured__* = `OFF`;

This code is used in section 203.

206. Clean up procedure parse stack but leave as ready to parse again. The following points are done:
1) clean up trace activity: normally done when parse object destroyed
2) leave first record on stack for efficiency
3) make sure first stack symbol on stack checked for delete attribute

⟨clean up procedure parse stack but leave as ready to parse again 206⟩ ≡

```
proc_parser→remove_from_stack(proc_parser→parse_stack__.top_sub__ − 1);
CAbs_lr1_sym *sym = proc_parser→top_stack_record()→symbol__;
if (sym ≠ 0) {
  if (sym→auto_delete__ ≡ ON) {
    ⟨Trace procedure pp's last symbol on stack set as autodelete 641⟩;
    delete sym;
  }
  proc_parser→top_stack_record()→set_symbol(0);    /* keeping a clean stack */
}
proc_parser→parse_stack__.lr_stk_init(*proc_parser→fsm_tbl__→start_state__);
```

This code is used in section 203.

207. Let's procedure parallel parse. do u?.

⟨let's procedure parallel parse. do u? 207⟩ ≡

```
THR_result rslt = proc_parser→parallel_parse();
```

This code is used in section 203.

208. Establish procedure parser's initial token setting. When the thread is established and waiting to be wakenned, the calling grammar sets the following variables within the critical region of the called thread: *from_thread__*, *pp_requesting_parallelism__*, and *no_competing_pp_ths__*.

Distinguish between chained procedure call and just a plain old thread call optimized by a procedure call. The chained T is the Caller parser's previous "go to" state. Its current token position is the tail character of the stacked T as the caller parser's current token context is the lookahead token and position returned from the called thread.

⟨establish procedure initial parser's token setting 208⟩ ≡

```
if (Caller_pp→top_stack_record()→state__→proc_call_addr__ ≡ 0) {    /* regular proc call */
  proc_parser→override_current_token(*Caller_pp→current_token(), Caller_pp→current_token_pos__);
  proc_parser→set_start_token(*Caller_pp→current_token());
  proc_parser→set_start_token_pos(Caller_pp→current_token_pos__);
  proc_parser→top_stack_record()→set_symbol(proc_parser→current_token());
}
else {    /* chained proc call */
  Cparse_record *pr =    /* curr stk pos is rel. 1 but access is rel to 0 UGH! */
  Caller_pp→get_stack_record(Caller_pp→current_stack_pos() − 2);
  int new_pos = Caller_pp→current_token_pos__ − 1;
  proc_parser→override_current_token(*pr→symbol__, new_pos);
  proc_parser→set_start_token(*pr→symbol__);    /* chained T */
  proc_parser→set_start_token_pos(new_pos);
}
proc_parser→token_supplier__ = Caller_pp→token_supplier__;
proc_parser→token_producer__ = Caller_pp→token_producer__;
proc_parser→error_queue__ = Caller_pp→error_queue__;
proc_parser→recycle_bin__ = Caller_pp→recycle_bin__;
proc_parser→sym_lookup_functor__ = Caller_pp→sym_lookup_functor__;
```

This code is used in section 203.

209. Create procedure communication variables.

⟨ create procedure communication variables 209 ⟩ ≡

```
char ma[SMALL_BUFFER_4K];
const char *pp_start = "YACCO2_MSG__::PROC::%i::%s start parsing\n";
const char *pp_stop = "YACCO2_MSG__::PROC::%i::%s stop parsing\n";
⟨ uns 23 ⟩;
```

This code is used in section 203.

210. Set procedure parameter passed to pp as a message.

⟨ set procedure parameter passed to pp as a message 210 ⟩ ≡

```
proc_parser→pp_requesting_parallelism__ = Caller_pp;
proc_parser→launched_as_procedure__ = true;
proc_parser→from_thread__ = Caller_pp;
proc_parser→no_competing_pp_ths__ = Caller_pp→no_requested_ths_to_run__;
```

This code is used in section 203.

211. Determine threads to launch by their first sets.
As an optimization before launching the thread, the thread's first set is checked to see if the start token, or the meta terminals `|+|` and `|.|` are present. Why are the meta terminals checked? `|+|` is the 'all shift' terminal used as a wild terminal facilty; it handles all terminals so even though the start token is not found in the first set, the wild token faclity indicates its presence. I do not check to see if the finite state automaton's "all shift" facility is on. Its presence in the first set is sufficient: testing the grammar's finite automaton to see if this facility is turned off is enough paranoia.

What about `|.|` the invisible shift meta terminal? In this case it denotes an epsilon rule within the start state configuration of the grammar so you better launch the thread as you do not know what's happening past that point when the token stream is being consumned. Yacco2's linker goes through this transient chain of first sets: internal discovery of what's after the `|.|` be it internal or external first sets from called threads. I should rely on the first set but as a precaution, I err to try it and if it doesn't work so what. It's a bit of overhead but at least it's better then not trying out the thread and having an irrate grammar writer to deal with. This type of grammatical situation is very rare but still needs checking.

This is a major optimization! The "pp" grammar checks in its parallel table list for the eligible threads that have the current terminal in their *first* **set**. If found, the parallel entry for those threads are added to the potential thread list. Only then does the parallel parse launch the threads. By absorbing the optimization into the "pp" thread it eliminates false thread starts. Now it's zippy-do-da. Do u hear the sirens? Hey u putting jell in y're hair?: Not zippy or whatever adjective or adverb expressed.

Take ...

⟨ External rtns and variables 22 ⟩ +≡
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*);

212. *find_threads_by_first_set*.
Work the global optimization of first sets and Terminals: See Yacco2's Linker. State's thread list against the T's thread list.

⟨ accrue thread code 142 ⟩ +≡
extern void yacco2 :: *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*)
{
  yacco2 :: **thread_array_record** ∗*thds* = (**yacco2** :: **thread_array_record** ∗) **yacco2** :: `THDS_STABLE__`;
  ⟨ determine if there is a bit map gened for state. no do it 213 ⟩;
  ⟨ define and set work variables of Terminal having threads 216 ⟩;
  ⟨ define and set state's dynamic work variables 214 ⟩;
  ⟨ search T's thd ids against State's thd id list. fnd add to-run thread list 217 ⟩;
}

213. Determine if there is a bit map gened for state. no do it.
As the grammar's state configuration is gened locally and has no knowledge about the global number of threads, its configuration has an indirection towards the thread entry having a pre-agreed to naming convention of the letter "I" concatenated with the thread name without its namespace. For example *ITH_eol* would be the global thread entry object for the "eol" grammar.

To make the thread launching efficient, a thread id bit map is used and searched. Cuz the state has just a list of **Thread_entry** pointers, this must be converted into the global bit map configuration. This is done per parallelism request. To offset each hit, the state's configuration contains a pointer for this dynamicly composed environment. As threads are more efficient than procedure calls, this is a one time inefficiency per state being gened on the fly. Now why again are threads more efficient? Cuz of objects and their rights of passage: Too much start-run-cleanup.

```
⟨determine if there is a bit map gened for state. no do it 213⟩ ≡
  static int no_of_gbl_thds(0);
  static int no_bit_mapped_words(0);
  static bool one_time(false);
  if (one_time ≡ false) {
    one_time = true;
    no_of_gbl_thds = thds→no_entries__;
    div_t x = div(no_of_gbl_thds, BITS_PER_WORD);
    if (x.rem ≠ 0) ++x.quot;
    no_bit_mapped_words = x.quot;
  }
  if (P_tbl.thd_id_bit_map__ ≡ 0) {
    ⟨define and set work variables of state threading table 215⟩;
    yacco2::ULINT(*maps) = (yacco2::ULINT(*))yacco2::BIT_MAPS_FOR_SALE__;
    P_tbl.thd_id_bit_map__ = (yacco2::ULINT(*)) & maps[yacco2::BIT_MAP_IDX__];
    yacco2::BIT_MAP_IDX__ += no_bit_mapped_words;
    if (yacco2::BIT_MAP_IDX__ > yacco2::TOTAL_NO_BIT_WORDS__) {
      char a[BUFFER_SIZE];
      yacco2::KCHARP msg = "Err␣no␣more␣bit␣maps:␣%i;␣adjust␣TOTAL_NO_BIT_WORDS␣in␣Link\
          er";
      sprintf(a, msg, yacco2::BIT_MAP_IDX__);
      Yacco2_faulty_precondition(a, __FILE__, __LINE__);
      exit(1);
    }
    div_t dd;
    for ( ; S_no_thd_entries > 0; −−S_no_thd_entries, ++S_cur_thread_entry_ptr) {
      yacco2::USINT S_thd_id = (*S_cur_thread_entry_ptr)→thd_id__;
      dd = div(S_thd_id, BITS_PER_WORD);
      ULINT bit_pos_value = 1 ≪ dd.rem;
      P_tbl.thd_id_bit_map__[dd.quot] |= bit_pos_value;
    }
  }
```

This code is cited in section 722.

This code is used in section 212.

214. Define and set state's dynamic work variables.

⟨ define and set state's dynamic work variables 214 ⟩ ≡

```
yacco2::ULINT S_cur_thd_id_map = P_tbl.thd_id_bit_map__[0];
```

This code is used in section 212.

215. Define and set work variables of state threading table.

⟨ define and set work variables of state threading table 215 ⟩ ≡

```
yacco2::Thread_entry **S_cur_thread_entry_ptr = (yacco2::Thread_entry **) &P_tbl.first_entry__;
yacco2::USINT S_no_thd_entries = P_tbl.no_entries__;
```

This code is used in section 213.

216. Define and set work variables of Terminal having threads.

⟨ define and set work variables of Terminal having threads 216 ⟩ ≡

```
yacco2::thd_ids_having_T *T_cur_thd_id_having_T_ptr;
yacco2::ULINT T_cur_thd_id_map;
T_array_having_thd_ids *t_array_having_thd_ids = (T_array_having_thd_ids *)
    yacco2::T_ARRAY_HAVING_THD_IDS__;

T_cur_thd_id_having_T_ptr = t_array_having_thd_ids→first_entry__[Current_T_id];
T_cur_thd_id_map = T_cur_thd_id_having_T_ptr→first_thd_id__[0];
```

This code is used in section 212.

217. Search T's thread ids against the State's thread entry list. fnd add to thread list. This is a linear search of segments. It is worked like a merge between two variable length lists of points. Its cost is linear bounded depending where the state's thread ids are relative to T's thread ids: before, within, or after. This linear bound can be 1 to the number of items in the largest list.

Both meta terminals `|+|` and `|.|` first sets get generated in Yacco2's linker. It is much more efficient to go thru a State and T list once. The expense is to explode the `|+|` meta terminal into all the terminals. This should be a rare occurance to have a thread's first set contain this meta terminal.

Bit maps are used: lets hear it for compression and possibly speed. To extract more speed, the inline assembler directive is used when developed on a Microsoft environment for the Intel 486 chipset. Without it, the bit map strategy is slower than the linear list. For the moment ⟨ extract thread ids from map and add their *thread_entry* to thread list 218 ⟩ is the portable piece of code until I improve the runtime strategy.

⟨ search T's thd ids against State's thd id list. fnd add to-run thread list 217 ⟩ ≡

```
int base_idx_for_thd_id_calc(0);
int cur_bit_word_idx(0);
do {
  yacco2::ULINT bit_map = T_cur_thd_id_map & S_cur_thd_id_map;
  if (bit_map ≠ 0) {
    base_idx_for_thd_id_calc = cur_bit_word_idx * BITS_PER_WORD;
    ⟨ extract thread ids from map and add their thread_entry to thread list 218 ⟩;
  }
  ++cur_bit_word_idx;
  T_cur_thd_id_map = T_cur_thd_id_having_T_ptr→first_thd_id__[cur_bit_word_idx];
  S_cur_thd_id_map = P_tbl.thd_id_bit_map__[cur_bit_word_idx];
} while (cur_bit_word_idx < no_bit_mapped_words);
```

This code is used in section 212.

218. Extract thread ids from map and add their *thread_entry* to thread list. Now the fun begins. What threads are to be run. The bits must be tested individually and their bit position converted into the their bit map vector co-ordinates: quotient * 32 + bit position.

For example, word 0, bit position 0 is thread id 0. Word 1 bit position 0 is thread id 32.

⟨extract thread ids from map and add their *thread_entry* to thread list 218⟩ ≡

```
yacco2::ULINT bit(1);
for (int bit_pos = 0; bit_pos ≤ BITS_PER_WORD_REL_0; ++bit_pos) {
  if (bit_map & bit) {
    ⟨add thread entry whose first set contains the current token 219⟩;
  }
  bit <<= 1;    /* next bit: rt to left order; insignificant to significant order */
}
```

This code is cited in section 217.

This code is used in section 217.

219. Add thread entry whose first set contains the current token.

⟨add thread entry whose first set contains the current token 219⟩ ≡

```
yacco2::USINT thd_id = base_idx_for_thd_id_calc + bit_pos;
Th_list.push_back(thds→first_entry__[thd_id]);
```

This code is used in section 218.

220. Ms Intel 486 Assembler extract thread ids from map and add their *thread_entry* to thread list.

⟨Ms Intel 486 assembler extract ids from map and add their *thread_entry* to thread list 220⟩ ≡

```
yacco2::Thread_entry *(*pte)[ ] = &thds→first_entry__;
yacco2::Thread_entry *te;
__asm
{
  pushad
  mov ebx, pte;    /* addr of thread stable[] of thread entries */
  mov esi, bit_map;    /* copy of bit map */
  mov edi, base_idx_for_thd_id_calc;
scn_bits: bsf eax, esi;    /* aex: idx of bit, esi: copied map to search */
  jz end_of_scan;    /* map completely scanned */
  btr esi, eax;    /* clear the fnd bit in map esi: the bit map, eax: the fnd bit pos to turn off */
  add eax, edi;    /* calced thd id */
  mov edx, [ebx][eax * 4];    /* fetch addr of thread entry */
  mov te, edx;    /* store the thread entry address */
}
Th_list.push_back(te);
__asm
{
  jmp scn_bits;    /* go scan more bits */
}
end_of_scan:
__asm
{
  popad;    /* clean up the dodos */
}
```

221. Parser Definitions — Pushdown Automaton. Just what you've been taught at university with its associated components:

parse stack
finite automaton tables

It supports 2 parsing paradigms: hohum and parallel.

The extras added to the pushdown automaton are the abort and stop parsing instructions, and the turning on and off of the wild shift facility. All 3 of these activities are controlled by the grammar writer's syntax directed code. They all get reset back to their initial settings when the thread completes parsing.

The abort parse is an abrupt way of killing the parse. It justs stops it. No result returned to the calling grammar. The stop parse is more refined in that one normally adds a terminal to the accept queue of the calling grammar before shutting down. If used, the all shift facilty needs to be turned off within some running context or else the terminal stream being parsed will overrun. This is protected against in the PDA but...

222. The parser structure.

```
⟨Structure defs 18⟩ +≡
  struct Parser { enum parse_result {
    erred, accepted, reduced, paralleled, no_thds_to_run
  };
  ⟨parser's internal variables 223⟩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 ) ;    /* parallel parser */
    Parser(yacco2::CAbs_fsm &Fsm_tbl, yacco2::Parser *Calling_parser);
      /* parallel parser: procedure called */
    ~Parser();
    ⟨PDA's defs 226⟩⟨Parser's containers defs 227⟩⟨Parser's token defs 229⟩⟨Parse's stack
        defs 228⟩⟨Parse's all shift, stop, and abort defs 225⟩
      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);
    ⟨Parallel parsing support definitions 224⟩} ;
```

223. Parser's internal variables.

⟨ parser's internal variables 223 ⟩ ≡

```
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__;
```

This code is used in section 222.

224. Parallel parsing support definitions.

⟨ Parallel parsing support definitions 224 ⟩ ≡

```
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();    /* how thread or procedure */
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_irregularites(yacco2::Caccept_parse &Calling_parm);
void abort_no_selected_accept_parse_in_arbitrator();
```

This code is used in section 222.

225. Parse's all shift, stop, and abort defs.

⟨ Parse's all shift, stop, and abort defs 225 ⟩ ≡

```
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);
```

This code is used in section 222.

226. PDA's defs.

⟨ PDA's defs 226 ⟩ ≡

```
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();
```

This code is used in section 222.

227. Parser's containers defs.

⟨ Parser's containers defs 227 ⟩ ≡

```
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);
```

This code is used in section 222.

228. Parse's stack defs.

⟨ Parse's stack defs 228 ⟩ ≡

```
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);    /* rel 0 */
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);    /* rel 0 */
```

This code is used in section 222.

229. Parser's token defs.

⟨ Parser's token defs 229 ⟩ ≡

```
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( );
```

This code is cited in section 708.

This code is used in section 222.

230. Parser Regular parser.
Runs a monolithic grammar: not a threaded grammar. i/o token containers are required whereas the threaded parser receives this information via a parameter at first thread startup or as a message within the calling parser. Not much is required in start up but to establish the runtime parse stack and fetch the first terminal for processing if it is available. How can it not be available? Well I support the empty language: moot but hugging theory.

Notice that the items imported are references instead of pointers. I'm trying it again. I hope that it works within the threaded environment. It didn't with cica Microsoft Visual studio 6 C++ compiler. Pointers were consistent.

cv__(0) and *mu__*(0) are removed from the initializer list due to linux honking.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::Parser::Parser
(yacco2::CAbs_fsm &Fsm_tbl
,yacco2::token_container_type *Token_supplier
,yacco2::token_container_type *Token_producer
,yacco2::UINT Token_supplier_key_pos
,yacco2::token_container_type *Error_queue
,yacco2::token_container_type *Recycle_bin
,yacco2::tble_lkup_type * Sym_lookup_functor
,bool Use_all_shift)
: fsm_tbl__(&Fsm_tbl)
, thread_name__(Fsm_tbl.id__)
, thread_entry__(0)
, token_supplier__(Token_supplier)
, token_producer__(Token_producer)
, error_queue__(Error_queue)
, recycle_bin__(Recycle_bin)
, current_token__(0)
, current_token_pos__(Token_supplier_key_pos)
, start_token__(0)
, start_token_pos__(Token_supplier_key_pos)
, sym_lookup_functor__(Sym_lookup_functor)
, abort_parse__(OFF)
, stop_parse__(OFF)
, use_all_shift__(Use_all_shift)
, has_questionable_shift_occured__(OFF)
, from_thread__(0)
, thread_no__(THREAD_SELF())
, cv_cond__(WAIT_FOR_EVENT)
, th_blk__()
, pp_accept_queue_idx__(0)
, pp_accept_queue__()
, th_active_cnt__(0)
, th_accepting_cnt__(0)
, pp_requesting_parallelism__(0)
, msg_id__(0)
, arbitrated_token__(0)
, no_competing_pp_ths__(0)
, no_requested_ths_to_run__(0)
, th_lst__()
, launched_as_procedure__(false)
, supplier_r_w_cnt__(1)
```

```
{
  CREATE_COND_VAR(cv__);
  CREATE_MUTEX(mu__);
  LOCK_MUTEX_OF_CALLED_PARSER(mu__, *this, " of self");
  parse_stack__.lr_stk_init(*Fsm_tbl.start_state__);
  for (int x = 0; x < pp_accept_queue_size; ++x) {
    pp_accept_queue__[x].initialize_it();
  }
  if (token_supplier__ ≠ 0) {
    supplier_r_w_cnt__ = token_supplier__→r_w_cnt__;
  }
  fsm_tbl__→parser(*this);
  Fsm_tbl.parser(*this);
  if (Token_supplier ≠ 0) {
    current_token__ = get_spec_token(current_token_pos__);
  }
  else {
    current_token__ = yacco2::PTR_LR1_eog__;
  }
  start_token__ = current_token__;
  ⟨check for empty language. yes, just exit 231⟩;
  parse_stack__.lr_stk_init(*fsm_tbl__→start_state__);
  if (YACCO2_T__ ≠ 0) {
    if (current_token__ ≡ 0) return;   /* no tokens */
    ⟨acquire trace mu 389⟩;
    yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪ "::" ≪
        " enum: " ≪ current_token__→enumerated_id__ ≪ ' ' ≪ '"' ≪ current_token__→id__ ≪ '"' ≪
        " pos: " ≪ current_token_pos__ ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\t\t::GPS FILE: ";
    EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ " GPS LINE: " ≪
        current_token__→tok_co_ords__.line_no__ ≪ " GPS CHR POS: " ≪
        current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨release trace mu 390⟩;
  }
}
```

231. Check for empty language.

```
⟨check for empty language. yes, just exit 231⟩ ≡
  if (current_token__ ≡ 0) return;
```

This code is used in section 230.

232. Parser Parallel parser.
The parse containers are all global. One can set up some of these containers for local requirements within the threaded grammar. Threaded grammar use this constructor. Elsewhere the threaded code is developed exposing its deployment. The calling grammar's parse object provides all the gory details to parse with its current token, token position, and token dispensor.

At initial startup, the token co-ordinates — dispensor, token, and position set — will be set within the **Parser**. The parse thread awakened by a message will have in its critical region the requestor's parallel parser address. Within the request for work loop, the messaged parser will extract from the calling parser its token assemble — dispensor, token, and position set

The error, recycle containers are optional. All these containers are taken from the monolithic parser that started the rave. Use of recursion to create a new i/o token containers is permissible. It's up to the designer. Lets hear it for openness! Don't be too cheery boy due to the following: *cv__*(0) and *mu__*(0) are removed from the initializer list due to linux honking.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Parser::Parser (yacco2::CAbs_fsm &Fsm_tbl
  , yacco2::Thread_entry &Thread_entry , yacco2::Parser *Calling_parser )
      : fsm_tbl__(&Fsm_tbl)
      , thread_name__(Thread_entry.thread_fnct_name__)
      , thread_entry__(&Thread_entry)
      , token_supplier__(0)
      , token_producer__(0)
      , current_token__(0)
      , current_token_pos__(0)
      , start_token__(0)
      , start_token_pos__(0)
      , recycle_bin__(0)
      , sym_lookup_functor__(0)
      , abort_parse__(OFF)
      , stop_parse__(OFF)
      , use_all_shift__(YES)
      , has_questionable_shift_occured__(OFF)
      , from_thread__(0)
      , thread_no__(THREAD_SELF())
      , cv_cond__(EVENT_RECEIVED)
      , th_blk__(this, Calling_parser)
      , pp_accept_queue__()
      , pp_accept_queue_idx__(0)
      , th_active_cnt__(0)
      , th_accepting_cnt__(0)
      , pp_requesting_parallelism__(0)
      , msg_id__(0)
      , arbitrated_token__(0)
      , no_competing_pp_ths__(0)
      , no_requested_ths_to_run__(0)
      , th_lst__()
      , launched_as_procedure__(false)
      , supplier_r_w_cnt__(0)
      {
        CREATE_COND_VAR(cv__);
        CREATE_MUTEX(mu__);
        LOCK_MUTEX_OF_CALLED_PARSER(mu__, *this, "␣of␣self");
        fsm_tbl__→parser(*this);
```

```
Fsm_tbl.parser(*this);
parse_stack__.lr_stk_init(*fsm_tbl__→start_state__);    /* no token yet */
for (int x = 0; x < pp_accept_queue_size; ++x) {
  pp_accept_queue__[x].initialize_it();
}
}
```

233. Parser Procedure call: Parallel parser.
Same as the parallel thread parser except for the registry of the thread into the *Parallel_thread_table* and setting how its called.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Parser::Parser(yacco2::CAbs_fsm &Fsm_tbl
  , yacco2::Parser *Calling_parser)
  : fsm_tbl__(&Fsm_tbl)
  , thread_name__(Fsm_tbl.id__)
  , thread_entry__(0)
  , token_supplier__(0)
  , token_producer__(0)
  , current_token__(0)
  , current_token_pos__(0)
  , start_token__(0)
  , start_token_pos__(0)
  , recycle_bin__(0)
  , sym_lookup_functor__(0)
  , abort_parse__(OFF)
  , stop_parse__(OFF)
  , use_all_shift__(YES)
  , has_questionable_shift_occured__(OFF)
  , from_thread__(0)
  , thread_no__(THREAD_SELF())
  , cv_cond__(EVENT_RECEIVED)
  , th_blk__()
  , pp_accept_queue__()
  , pp_accept_queue_idx__(0)
  , th_active_cnt__(0)
  , th_accepting_cnt__(0)
  , pp_requesting_parallelism__(0)
  , msg_id__(0)
  , arbitrated_token__(0)
  , no_competing_pp_ths__(0)
  , no_requested_ths_to_run__(0)
  , th_lst__()
  , launched_as_procedure__(true)
  , supplier_r_w_cnt__(0)
  {
    CREATE_COND_VAR(cv__);
    CREATE_MUTEX(mu__);
    LOCK_MUTEX_OF_CALLED_PARSER(mu__, *this, "␣of␣self");
    fsm_tbl__→parser(*this);
    Fsm_tbl.parser(*this);
    parse_stack__.lr_stk_init(*fsm_tbl__→start_state__);   /* no token yet */
    for (int x = 0; x < pp_accept_queue_size; ++x) {
      pp_accept_queue__[x].initialize_it();
    }
  }
```

234. ~Parser.
General house keeping by popping the stack. Popping allows the firing off of the start rule and automatic garbage collection.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::Parser::~Parser()
{
  clear_parse_stack();
  DESTROY_COND_VAR(cv__);
  DESTROY_MUTEX(mu__);
}
```

235. Parser — PDA's implementation.

236. Shift.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::shift(yacco2::Shift_entry &SE)
{
  ⟨ Reserve and get current stack record 352 ⟩;
  ⟨ set parse stack symbol to current token 242 ⟩;
  yacco2::State *Goto_state = SE.goto__;
  ⟨ add_to_stack 349 ⟩;
  ⟨ Trace TH the parse stack configuration 581 ⟩;
  get_next_token();
}
```

237. Find shift entry.

⟨ find shift entry 237 ⟩ ≡
```
yacco2::Shift_entry *se(0);
if (pr→state__→shift_tbl_ptr__ ≠ 0) se = find_cur_T_shift_entry();
```
This code is used in sections 251 and 271.

238. Invisible shift. Its symbol |.|.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::invisible_shift(yacco2::Shift_entry &SE)
{
  ⟨ Reserve and get current stack record 352 ⟩;
  ⟨ set parse stack symbol to invisible shift operator 239 ⟩;
  yacco2::State *Goto_state = SE.goto__;
  ⟨ add_to_stack 349 ⟩;
  ⟨ Trace TH the parse stack configuration 581 ⟩;
}
```

239. Set parse stack symbol to invisible shift operator.

⟨ set parse stack symbol to invisible shift operator 239 ⟩ ≡
```
pr→symbol__ = NS_yacco2_k_symbols::PTR_LR1_invisible_shift_operator__;
```
This code is used in section 238.

240. Questionable shift. Its symbol is |?|. Note, as it is used for error situations though it acts like a wild token as in |+|, it does not advance to the next token in the parse stream! It must be explicitly done by the grammar writer. I haven't head wrestled "error processing / correction" yet.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::questionable_shift(yacco2::Shift_entry &SE)
{
  has_questionable_shift_occured__ = ON;
  ⟨ Reserve and get current stack record 352 ⟩;
  ⟨ set parse stack symbol to current token 242 ⟩;
  yacco2::State *Goto_state = SE.goto__;
  ⟨ add_to_stack 349 ⟩;
  ⟨ Trace TH the parse stack configuration 581 ⟩;
}
```

241. All shift.
The current terminal and not `|+|` is placed onto the parse stack. The fsm's 'go to' state is the vectored `|+|` symbol.

⟨ accrue yacco2 code 33 ⟩ +≡
void yacco2 :: **Parser** :: *all_shift* (**yacco2** :: **Shift_entry** &`SE`)
{
⟨ Reserve and get current stack record 352 ⟩;
⟨ set parse stack symbol to current token 242 ⟩;
yacco2 :: **State** ∗*Goto_state* = `SE`.*goto__*;
⟨ *add_to_stack* 349 ⟩;
⟨ Trace TH the parse stack configuration 581 ⟩;
get_next_token( );
}

242. Set parse stack symbol to current token.

⟨ set parse stack symbol to current token 242 ⟩ ≡
pr→*symbol__* = *current_token__*; /∗ state's shift symbol ∗/

This code is used in sections 236, 240, and 241.

243. Reduce. The reduce.

⟨ accrue yacco2 code 33 ⟩ +≡
yacco2 :: **Parser** :: **parse_result yacco2** :: **Parser** :: *reduce* (**yacco2** :: **Reduce_entry** &`RE`)
{
⟨ execute subrule with it directives and create rule 244 ⟩;
⟨ pop rule's rhs subrule from parse stack 246 ⟩;
⟨ put rule onto parse stack 247 ⟩;
⟨ find rule's shift entry in fsm 248 ⟩;
⟨ Validate if rule shift symbol in fsm table 559 ⟩;
⟨ put goto state onto parse stack, and return accepted or reduced result 245 ⟩;
}

244. Execute the subrule, its directives, and create the rule.
Inside the rule's constructor is the *lhs* − *constructor* directive code. The top of the stack address is passed to *reduce_rhs_of_rule* to efficiently calculate the subrule's parameters as its just an array of **Cparse_record**. This is a tricky-dicky, now no politics, cuz I'm really fetching the first component of the stack record which is its grammatical symbol. See notes on the real story. Added a rule recycling program to speed up parser due to new hit on birth-run-delete cycle. See **Recycled_rule_struct** discussion.

⟨ execute subrule with it directives and create rule 244 ⟩ ≡
Rule_s_reuse_entry ∗*rule_rec1* (0);
Rule_s_reuse_entry ∗∗*rule_rec* = &*rule_rec1* ;
fsm_tbl__→*reduce_rhs_of_rule* (`RE`.*rhs_id__*, *rule_rec*);

This code is used in section 243.

245.

⟨ put goto state onto parse stack, and return accepted or reduced result 245 ⟩ ≡

```
yacco2::State *Goto_state = se→goto__;
⟨ add_to_stack 349 ⟩;
⟨ Trace TH the parse stack configuration 581 ⟩;
if (se→goto__→state_no__ ≡ 1)
   return Parser::accepted;
return Parser::reduced;
```

This code is used in section 243.

246.

⟨ pop rule's rhs subrule from parse stack 246 ⟩ ≡

```
remove_from_stack((*rule_rec)→rule_→rule_info__.rhs_no_of_parms__);
```

This code is used in section 243.

247.

⟨ put rule onto parse stack 247 ⟩ ≡

```
parse_stack__.top__→set_symbol((*rule_rec)→rule_);   /* stack state's rule shift symbol */
parse_stack__.top__→set_rule_s_reuse_entry(*rule_rec);
```

This code is used in section 243.

248.

⟨ find rule's shift entry in fsm 248 ⟩ ≡

```
Shift_entry *se(0);
if (parse_stack__.top__→state__→shift_tbl_ptr__ ≠ 0)
   se = find_R_or_paralleled_T_shift_entry((*rule_rec)→rule_→enumerated_id__);
```

This code is used in section 243.

249. Regular parse.
This parse comes from a non-threaded grammar executed from a process. One can use recursion to start many parse streams. In fact, processing of include files is done this way with an appropriate nested file count limit to prevent overruns.

Added *failed* call to monolithic grammar as it becomes a global way to handle an aborted parse. For example, a general error message could be put into the error queue by the monolithic grammar. This becomes a cheap way to deal with invalid token sequences. At least it pin points where it occured by a general error message. The proper refinement is to go to each grammar and program the catching of the error by use of the `|.|` terminal or the `|+|` terminal within the subrule. How refined do u want to go or be or not to go? that is the ?

⟨accrue yacco2 code 33⟩ +≡
yacco2 :: **Parser** :: **parse_result** **yacco2** :: **Parser** :: *parse*( )
{
⟨check for empty language. yes, exit as accepted 250⟩;
⟨fire off fsm's op directive 252⟩;
parse_result *result*;
read_token_stream:
{
⟨process tokens 251⟩;
}
parse_successful:
return **Parser** :: *accepted*;
parse_unsuccessful:
fsm_tbl__→*failed*( ); /* ?sdc from grammar writer for the error queue */
⟨Trace TH straight parse error 590⟩;
cleanup_stack_due_to_abort( );
return **Parser** :: *erred*;
}

250. Check for empty language.

⟨check for empty language. yes, exit as accepted 250⟩ ≡
if (*current_token__* ≡ 0) **return** **Parser** :: *accepted*;

This code is used in section 249.

251. Process tokens.

⟨ process tokens 251 ⟩ ≡
  ⟨ Reserve and get current stack record 352 ⟩;
  if (*stop_parse__* ≡ ON) {
    cleanup_stack_due_to_abort( );   /* quasi controlled abort */
    goto *parse_successful*;
  }
  if (*abort_parse__* ≡ ON) **goto** *parse_unsuccessful*;
  State ∗*cur_state* = *pr*→*state__*;
  ⟨ dispatch to parallel, or proc call, or straight parsing 254 ⟩;
parallel_parsing:
  ⟨ try parallel parse. no threads-to-run go straight 255 ⟩;
  ⟨ is parallel parsing successful? If so reduce the |||phrase 256 ⟩;
  ⟨ parallel parsing unsuccessful. So, set up + go to straight parsing 258 ⟩;
proc_call_parsing:
  {
    ⟨ try proc call parse. no threads-to-run go straight 259 ⟩;
    ⟨ is proc call parsing successful? If so reduce the |t|phrase 260 ⟩;
    ⟨ proc call parsing unsuccessful. So, set up + go to straight parsing 262 ⟩;
  }
straight_parsing:
  ⟨ find shift entry 237 ⟩;
  ⟨ try various shift types. if executed go to process next token in token stream 253 ⟩;
  ⟨ find reduce entry 263 ⟩;
  ⟨ try reduce 264 ⟩;
  goto *parse_unsuccessful*;

This code is used in section 249.

252. Fire off fsm's op directive.

This is the fsm's directive that gets run when the parser starts up. As a parallel parser is within a run loop, each time it starts running this directive gets called. It is a directive that allows the grammar writer to preset or pre-evaluate appropriate events. For example, it is used in the Pascal translator to pre-evaluate by symbol table lookup the passed identifier token. If it is morphed, the new token is then used in the parse. Good stuff.

⟨ fire off fsm's op directive 252 ⟩ ≡
  fsm_tbl__→*op*( );

This code is used in sections 249 and 269.

253. Try various shift types.
The parser favours a shift before a reduce operation. There are 4 types of shifts. The regular shift found in the token stream and 3 meta terminal shifts — |?| questionable, |.| invisible, and |+| all of which are not found in the token stream. The rank of shifts is conditionally checked for their presence within the current parse state with their test order being regular, followed by questionable, invisible, and all shift. The all shift is controlled by the parser's 'all shift' facility. If this facility was not present, the parse would always overrun the token stream. The turning on and off is controlled by the syntax directed code of the parsing grammar.
Comment:
See bug's comment.

⟨ try various shift types. if executed go to process next token in token stream 253 ⟩ ≡

```
if (se ≠ 0) {
  shift(*se);
  goto read_token_stream;
}
if (cur_state→questionable_shift__ ≠ 0) {
    /* guard against perpetual machine using |?|and last token "eog" */
  if (has_questionable_shift_occured__ ≡ ON) {    /* previous state action */
    ⟨ Invalid |?|instead of |+|use 543 ⟩;
  }
  questionable_shift(*cur_state→questionable_shift__);
  goto read_token_stream;
}
if (cur_state→inv_shift__) {
  invisible_shift(*cur_state→inv_shift__);
  goto read_token_stream;
}
if (use_all_shift__ ≡ ON) {
  if (cur_state→all_shift__ ≡ 0) { }
  else {    /* guard against overrun of token dispensor using |+| */
    if (current_token__→enumerated_id__ ≡ LR1_Eog)
    {
      use_all_shift__ = OFF;    /* turn off the all shift operator */
      all_shift(*cur_state→all_shift__);
    }
    else {
      all_shift(*cur_state→all_shift__);
      goto read_token_stream;
    }
  }
}
```

This code is cited in section 738.

This code is used in sections 251 and 271.

254. Dispatch to parallel, proc call, or straight parsing.

⟨ dispatch to parallel, or proc call, or straight parsing 254 ⟩ ≡

```
⟨ Validate any token for parsing 544 ⟩;
if (cur_state→parallel_shift__ ≠ 0) goto parallel_parsing;
if (cur_state→proc_call_shift__ ≠ 0) goto proc_call_parsing;
else goto straight_parsing;
```

This code is used in sections 251 and 271.

255. Try parallel parse.
It checks whether there are threads to be run by their first set. If not, the *no_thds_to_run* result is returned so go do some straight parsing.

⟨ try parallel parse. no threads-to-run go straight 255 ⟩ ≡
  result = *start_parallel_parsing*(∗*cur_state*);
  if (*result* ≡ *no_thds_to_run*) **goto** *straight_parsing*;

This code is used in sections 251 and 271.

256. Is parallel parsing successful?. If so reduce the ||| phrase. The wrinkle is whether a chained procedure call is present. This extends the subrule expression until after the chained procedure call and then it is reduced.

⟨ is parallel parsing successful? If so reduce the ||| phrase 256 ⟩ ≡
  if (*result* ≡ *paralleled*) {
    if (*parse_stack__.top__→state__→proc_call_shift__* ≠ 0) {
      cur_state = *parse_stack__.top__→state__*;
      goto *proc_call_parsing*; /∗ chained proc call so reduce later ∗/
    }
    ⟨ find parallel reduce entry 257 ⟩;
    ⟨ Validate reduce entry 560 ⟩;
    ⟨ Get current stack record 353 ⟩;
    ⟨ try reduce 264 ⟩;
  }

This code is used in sections 251 and 271.

257. find parallel reduce entry.

⟨ find parallel reduce entry 257 ⟩ ≡
  Reduce_entry ∗*re*(0);
  if (*parse_stack__.top__→state__→reduce_tbl_ptr__* ≠ 0) *re* = *find_parallel_reduce_entry*( );

This code is used in section 256.

258. Parallel parsing unsuccessful.
So, set up + go to straight parsing.

⟨ parallel parsing unsuccessful. So, set up + go to straight parsing 258 ⟩ ≡
  ⟨ Trace TH failed parallel try straight parse 588 ⟩;
  ⟨ Get current stack record 353 ⟩;
  goto *straight_parsing*;

This code is used in sections 251 and 271.

259. Try proc call parse.
It checks whether there is a proc call entry in state. If not, the *no_thds_to_run* result is returned so go do some straight parsing.

⟨ try proc call parse. no threads-to-run go straight 259 ⟩ ≡

```
THR_result rslt = chained_proc_call_parsing(*cur_state);    /* result = rslt; */
switch (rslt) {
case erred: goto straight_parsing;
case no_thds_to_run: goto straight_parsing;
default:
  {
    result = paralleled;
    break;
  }
}
```

This code is used in sections 251 and 271.

260. Is proc call parsing successful?. If so reduce the |t| phrase.

⟨ is proc call parsing successful? If so reduce the |t| phrase 260 ⟩ ≡

```
if (result ≡ paralleled) {
  ⟨ find proc call reduce entry 261 ⟩;
  ⟨ Validate reduce entry 560 ⟩;
  ⟨ Get current stack record 353 ⟩;
  ⟨ try reduce 264 ⟩;
}
```

This code is used in sections 251 and 271.

261. find proc call reduce entry.

⟨ find proc call reduce entry 261 ⟩ ≡

```
Reduce_entry *re(0);
if (parse_stack__.top__→state__→reduce_tbl_ptr__ ≠ 0) re = find_proc_call_reduce_entry();
```

This code is used in section 260.

262. Proc call parsing unsuccessful.
So, set up + go to straight parsing.

⟨ proc call parsing unsuccessful. So, set up + go to straight parsing 262 ⟩ ≡

```
⟨ Trace TH failed proc call try straight parse 589 ⟩;
⟨ Get current stack record 353 ⟩;
goto straight_parsing;
```

This code is used in sections 251 and 271.

263. find reduce entry.

⟨ find reduce entry 263 ⟩ ≡

```
Reduce_entry *re(0);
if (parse_stack__.top__→state__→reduce_tbl_ptr__ ≠ 0) re = find_reduce_entry();
```

This code is used in sections 251 and 271.

264. Try reduce.
The stop parse is checked after the reduce syntax directed code has been run. Provides a little more flexibility to the grammar writer's actions.

```
⟨ try reduce 264 ⟩ ≡
  if (re ≠ 0) {
    result = reduce(*re);
    if (stop_parse__ ≡ ON) {
      cleanup_stack_due_to_abort();      /* quasi controlled abort */
      goto parse_successful;
    }
    if (abort_parse__ ≡ ON) goto parse_unsuccessful;
    if (result ≡ Parser::reduced) goto read_token_stream;
    if (result ≡ Parser::accepted) goto parse_successful;
  }
```

This code is cited in section 719.

This code is used in sections 251, 256, 260, and 271.

265. Parallel shift.
A parallel shift has the following stack configuration:
|||, followed by |+|, |?|, or newly minted terminal
It places the parallel terminal onto the parse stack even though it is not part of the input token stream. I felt that it should faithfully follow the grammatical expression.

This is the tailend of the parallel parse that shifts the arbitrated symbol onto the parse stack. Please note the conditional 2nd attempt on the |+|. If it is present in the current state configuration, then the shift is successful. The only subtlety is in the arbitration code. What happens if there are many returned terminals? There has to be a choice made or the first item in the accept queue gets returned. Should this be a run-time-error if the arbitration code does not select the many to one situation? As parallelism is quasi-random in execution order so are the terminal placements in the accept queue. Where a single processor seems to work, a multi-processor can lead to different results per execution. The grammar should honk with a mildly acidic warning. It does now — see note.

Note: Support for |?|— questionable shift operator.
This is like the meta |+|terminal but it allows the grammar write to state that the returned T is an error. In the pecking order of shift presence, the returned T is tested first for its presence within the state. If it is not found then the meta shift terminals are tested in the following order: |?|, |+|.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::parallel_shift(yacco2::CAbs_lr1_sym &Accept_terminal)
  {
    ⟨ Reserve and get current stack record 352 ⟩;
    Shift_entry *se(0);
    if (pr→state__→shift_tbl_ptr__ ≠ 0)
      se = find_R_or_paralleled_T_shift_entry(Accept_terminal.enumerated_id__);
    if (se ≠ 0) goto set_stack_to_symbol_being_shifted;
    se = pr→state__→questionable_shift__;
    if (se ≠ 0) goto set_stack_to_symbol_being_shifted;
    se = pr→state__→all_shift__;
    if (se ≠ 0) goto set_stack_to_symbol_being_shifted;
    ⟨ Error shift symbol not fnd in fsm table 558 ⟩;
  set_stack_to_symbol_being_shifted:
    ⟨ shift parallel's returned symbol and goto state 266 ⟩;
  }
```

266. Shift parallel's returned symbol and goto state.

⟨shift parallel's returned symbol and goto state 266⟩ ≡
pr→symbol__ = &*Accept_terminal*; /* state's |||shift symbol */
yacco2 :: **State** **Goto_state* = *se→goto__*;
⟨ *add_to_stack* 349⟩; /*¡Trace TH the parse stack configuration¿; */

This code is used in section 265.

267. Proc call shift.
A proc call shift has the following stack configuration:
|t|, |+|or |?|or newly minted terminal
It places the proc call terminal onto the parse stack even though it is not part of the input token stream. I felt that it should faithfully follow the grammatical expression.

This is the tailend of the proc call parse that shifts the arbitrated symbol onto the parse stack. Please note the conditional 2nd attempt on the |+|or |?|to catch the eye as an error. If it is present in the current state configuration, then the shift is successful. The only subtlety is in the arbitration code. What happens if there are many returned terminals? There has to be a choice made or the first item in the accept queue gets returned. Should this be a run-time-error if the arbitration code does not select the many to one situation? As parallelism is quasi-random in execution order so are the terminal placements in the accept queue. Where a single processor seems to work, a multi-processor can lead to different results per execution. The grammar should honk with a mildly acidic warning. It does now — see note.

⟨accrue yacco2 code 33⟩ +≡
void yacco2 :: **Parser** :: *proc_call_shift*(**yacco2** :: **CAbs_lr1_sym** &*Accept_terminal*)
{
⟨Reserve and get current stack record 352⟩;
Shift_entry **se*(0);
if (*pr→state__→shift_tbl_ptr__* ≠ 0)
se = *find_R_or_paralleled_T_shift_entry*(*Accept_terminal.enumerated_id__*);
if (*se* ≠ 0) **goto** *set_stack_to_symbol_being_shifted*;
se = *pr→state__→all_shift__*;
if (*se* ≠ 0) **goto** *set_stack_to_symbol_being_shifted*;
se = *pr→state__→questionable_shift__*;
if (*se* ≠ 0) **goto** *set_stack_to_symbol_being_shifted*;
⟨Error shift symbol not fnd in fsm table 558⟩;
set_stack_to_symbol_being_shifted:
⟨shift proc call's returned symbol and goto state 268⟩;
}

268. Shift proc call's returned symbol and goto state.

⟨shift proc call's returned symbol and goto state 268⟩ ≡
pr→symbol__ = &*Accept_terminal*; /* state's |t|shift symbol */
yacco2 :: **State** **Goto_state* = *se→goto__*;
⟨ *add_to_stack* 349⟩; /*¡Trace TH the parse stack configuration¿; */

This code is used in section 267.

269. Parallel parse.
The control loop consuming the parallel tokens.

⟨ accrue yacco2 code 33 ⟩ +≡
yacco2 :: **Parser** :: **parse_result** **yacco2** :: **Parser** :: *parallel_parse* ( )
{
⟨ fire off fsm's op directive 252 ⟩;
parse_result *result*;
⟨ check for empty language. yes unsuccessful parallel parse 270 ⟩;
read_token_stream:
{
⟨ process parallel tokens 271 ⟩;
}
parse_successful:
return *parallel_parse_successful* ( );
parse_unsuccessful:
return *parallel_parse_unsuccessful* ( );
}

270. Check for empty language. yes unsuccessful parallel parse.

⟨ check for empty language. yes unsuccessful parallel parse 270 ⟩ ≡
if (*current_token__* ≡ 0) **goto** *parse_unsuccessful*;
goto *read_token_stream*;

This code is used in section 269.

271. Process parallel tokens.

```
⟨ process parallel tokens 271 ⟩ ≡
  ⟨ Reserve and get current stack record 352 ⟩;
  if (stop_parse__ ≡ ON) {
    cleanup_stack_due_to_abort( );    /* quasi controlled abort */
    goto parse_successful;
  }
  if (abort_parse__ ≡ ON) goto parse_unsuccessful;
  State *cur_state = pr→state__;
  ⟨ dispatch to parallel, or proc call, or straight parsing 254 ⟩;
parallel_parsing:
  ⟨ try parallel parse. no threads-to-run go straight 255 ⟩;
  ⟨ is parallel parsing successful? If so reduce the |||phrase 256 ⟩;
  ⟨ parallel parsing unsuccessful. So, set up + go to straight parsing 258 ⟩;
proc_call_parsing:
  {
    ⟨ try proc call parse. no threads-to-run go straight 259 ⟩;
    ⟨ is proc call parsing successful? If so reduce the |t|phrase 260 ⟩;
    ⟨ proc call parsing unsuccessful. So, set up + go to straight parsing 262 ⟩;
  }
straight_parsing:
  ⟨ find shift entry 237 ⟩;
  ⟨ try various shift types. if executed go to process next token in token stream 253 ⟩;
  ⟨ find reduce entry 263 ⟩;
  ⟨ try reduce 264 ⟩;
  goto parse_unsuccessful;
```

This code is used in section 269.

272. Parallel parse successful.
Put the accept message into the requesting grammar's accept queue. It checks whether it is the last active thread stopping. If so, it wakes up the requesting grammar by an event.

Notice the ⟨set thread status if launched as a thread 273⟩ is placed in the following parallel parse procedures: *parallel_parse_successful* and *parallel_parse_unsuccessful*. This is done to optimize the number of threads run instead of after the thread has cleanised itself from parsing in the thread loop. See *Parallel thread code* loop. ⟨set thread status if launched as a thread 273⟩ was just after the ⟨let's parallel parse. do u? 198⟩. Here's the take, when a event is sent to the requesting grammar, the thread library can restart executing the calling grammar while in a single cpu environment the parallel thread is put on hold to complete its duties some time later. Now the grammar requesting parallelism can continue its parse that can again request parallelism that can contain the thread that is winding down. Due to the winding down thread's status being busy, another copy of the thread is created and run. A little softshoe please...

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Parser::parse_result yacco2::Parser::parallel_parse_successful( )
  {
    ⟨ Trace TH current token, and accepted terminal wrapper 595 ⟩;
    if (launched_as_procedure__ ≡ true) {
      ⟨ reduce requesting grammar's active threads count 280 ⟩;
      ⟨ insert token into requesting grammar's accept queue 278 ⟩;
      clean_up( );
      return Parser::accepted;
    }
    else {
      ⟨ set thread status if launched as a thread 273 ⟩;
      ⟨ acquire parallelism requesting grammar's mutex if required 275 ⟩;
      ⟨ reduce requesting grammar's active threads count 280 ⟩;
      ⟨ insert token into requesting grammar's accept queue 278 ⟩;
      clean_up( );
      ⟨ notify requesting grammar if launched as a thread 274 ⟩;
      ⟨ release parallelism requesting grammar's mutex if required 276 ⟩;
      return Parser::accepted;
    }
  }
```

273. Set thread status if launched as a thread.

```
⟨ set thread status if launched as a thread 273 ⟩ ≡
  ⟨ acquire global thread table critical region 380 ⟩;
  th_blk__.set_waiting_for_work( );
  ⟨ release global thread table critical region 381 ⟩;
```

This code is cited in section 272.

This code is used in sections 272 and 279.

274. Notify requesting grammar if launched as a thread.

```
⟨ notify requesting grammar if launched as a thread 274 ⟩ ≡
  ⟨ notify parallelism requesting grammar if last thread to complete 277 ⟩;
```

This code is used in sections 272 and 279.

275. Acquire parallelism requesting grammar's mutex if required.
If there is only 1 thread running, the critical region is down graded to just a local context. This is an optimization to minimize "acquire-release" of mutexes.

⟨ acquire parallelism requesting grammar's mutex if required 275 ⟩ ≡
  LOCK_MUTEX_OF_CALLED_PARSER(*pp_requesting_parallelism__→mu__*, ∗**this**, "␣of␣calling␣grammar");

This code is used in sections 272 and 279.

276. Release parallelism requesting grammar's mutex if required. This is an optimization to minimize "acquire-release" of mutexes. *no_competing_pp_ths__* is a read-only variable that gets set when the thread is called. It eliminates the called thread having to acquire the mutex of the calling grammar to determine whether only 1 thread launched.

⟨ release parallelism requesting grammar's mutex if required 276 ⟩ ≡
  UNLOCK_MUTEX_OF_CALLED_PARSER(*pp_requesting_parallelism__→mu__*, ∗**this**, "␣of␣calling␣grammar");

This code is used in sections 272 and 279.

277. Notify parallelism requesting grammar if last thread to complete.

⟨ notify parallelism requesting grammar if last thread to complete 277 ⟩ ≡
  if (*have_all_threads_reported_back*( ) ≡ YES) {
    ⟨ Trace MSG all threads reported back 621 ⟩;
    post_event_to_requesting_grammar(∗*pp_requesting_parallelism__*, *Accept_parallel_parse*, ∗**this**);
  }
  else {
    ⟨ Trace MSG not all threads reported back 622 ⟩;
  }

This code is used in section 274.

278. Insert token into requesting grammar's accept queue.

⟨ insert token into requesting grammar's accept queue 278 ⟩ ≡
  pp_requesting_parallelism__→put_T_into_accept_queue(*pp_rsvp__*);

This code is used in sections 272 and 282.

279. Parallel parse unsuccessful.
If it is the last active thread, it wakes up the requesting grammar via a message. Otherwise, it just winds down without any message: a bit of an optimization to lowering messages between friends.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::Parser::parse_result yacco2::Parser::parallel_parse_unsuccessful()
{
  ⟨ check failed directive for possible acceptance 281 ⟩;
  ⟨ Trace TH parallel parse current token when an error has occured 596 ⟩;
  if (launched_as_procedure__ ≡ true) {
    ⟨ reduce requesting grammar's active threads count 280 ⟩;
    goto fire_off_error_functor;
  }
  else {
    ⟨ set thread status if launched as a thread 273 ⟩;
    ⟨ acquire parallelism requesting grammar's mutex if required 275 ⟩;
    ⟨ reduce requesting grammar's active threads count 280 ⟩;
    ⟨ notify requesting grammar if launched as a thread 274 ⟩;
    ⟨ release parallelism requesting grammar's mutex if required 276 ⟩;
  }
fire_off_error_functor:
  cleanup_stack_due_to_abort();
  clean_up();
  return Parser::erred;
}
```

280. Reduce requesting grammar's active threads count.

⟨ reduce requesting grammar's active threads count 280 ⟩ ≡
```
⟨ Trace TH before parallel parse thread message count reduced 598 ⟩;
−− pp_requesting_parallelism__→th_active_cnt__;
if (supplier_r_w_cnt__ > 1) {
  −− pp_requesting_parallelism__→supplier_r_w_cnt__;
  if (token_supplier__→r_w_cnt__ > 1) {
    ⟨ acquire token mu 391 ⟩;
    −− token_supplier__→r_w_cnt__;
    ⟨ release token mu 392 ⟩;
  }
}
⟨ Trace TH after parallel parse thread message count reduced 599 ⟩;
```
This code is used in sections 272 and 279.

281. Check failed directive for possible acceptance.
A fsm *failed* directive was added to allow for a last chance attempt at an aborted thread parse. One can return an error token to the calling grammar making its look like a successful parse via syntax directed code of the *failed* directive. It's not a panacea but hey it helps.

⟨ check failed directive for possible acceptance 281 ⟩ ≡
```
if (fsm_tbl__→failed() ≡ true) {
  return parallel_parse_successful();
}
```
This code is used in sections 279 and 283.

282. Proc call parse successful.
Put the accept message into the requesting grammar's accept queue. Just return back to callr.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Parser::parse_result yacco2::Parser::proc_call_parse_successful( )
{
  ⟨ Trace TH current token, and accepted terminal wrapper 595 ⟩;
  ⟨ insert token into requesting grammar's accept queue 278 ⟩;
  clean_up( );
  return Parser::accepted;
}
```

283. Proc call parse unsuccessful.
If it is the last active thread, it wakes up the requesting grammar via a message. Otherwise, it just winds down without any message: a bit of an optimization to lowering messages between friends.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Parser::parse_result yacco2::Parser::proc_call_parse_unsuccessful( )
{
  ⟨ check failed directive for possible acceptance 281 ⟩;
  ⟨ Trace TH proc call parse current token when an error has occured 597 ⟩;
  goto fire_off_error_functor;
fire_off_error_functor:
  cleanup_stack_due_to_abort( );
  clean_up( );
  return Parser::erred;
}
```

284. Find current T shift entry.
Algo. binary search 6.2.1 from Knuth Vol. 3. A little speed to eliminate the passing of the enumerate value. A quick test showed approximately the sequential search is faster than the binary search when the table population is less than 72.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Shift_entry *yacco2::Parser::find_cur_T_shift_entry( )
  {
    ⟨ Reserve and get current stack record 352 ⟩;
    yacco2::USINT Enum_id = current_token__→enumerated_id__;
    State *State_ptr = pr→state__;
    Shift_tbl *st = State_ptr→shift_tbl_ptr__;
    yacco2::USINT cnt = st→no_entries__;
    Shift_entry_array_type *shft_entry_array = (Shift_entry_array_type *) &st→first_entry__;
    yacco2::Shift_entry *k_entry;
    if (cnt > SEQ_SRCH_VS_BIN_SRCH_LIMIT) goto bin_srch;
    for (int x = 0; x < cnt; ++x) {
      k_entry = &(*shft_entry_array)[x];
      if (Enum_id ≡ k_entry→id__) return k_entry;
      if (Enum_id < k_entry→id__) break;
    }
  eolr_seq:
    for (int x = 0; x < cnt; ++x) {
      k_entry = &(*shft_entry_array)[x];
      if (LR1_Eolr ≡ k_entry→id__) return k_entry;
      if (LR1_Eolr < k_entry→id__) return 0;
    }
    return 0;
  bin_srch: int lower = 1;
    int upper = cnt;
    int seg_ln;
    int mid_pt;
    int mid_pt_rel0;
  B2:    /* calc mid pt */
    if (upper < lower) goto eolr_srch;
    seg_ln = upper + lower;
    mid_pt = seg_ln ≫ 1;
    mid_pt_rel0 = mid_pt − 1;
    k_entry = &(*shft_entry_array)[mid_pt_rel0];
  B3:    /* compare */
    if (Enum_id ≡ k_entry→id__) return k_entry;
    if (Enum_id > k_entry→id__) goto B5;
  B4:    /* adjust upper */
    upper = mid_pt − 1;
    goto B2;
  B5:    /* adjust lower */
    lower = mid_pt + 1;
    goto B2;
  eolr_srch:    /* see if all T in set */
    lower = 1;
    upper = st→no_entries__;
  B2_eolr:    /* calc mid pt */
```

```
    if (upper < lower) return 0;
    seg_ln = upper + lower;
    mid_pt = seg_ln >> 1;
    mid_pt_rel0 = mid_pt - 1;
    k_entry = &(*shft_entry_array)[mid_pt_rel0];
    if (LR1_Eolr == k_entry->id__) return k_entry;
    if (LR1_Eolr > k_entry->id__) goto B5_eolr;
  B4_eolr:      /* adjust upper */
    upper = mid_pt - 1;
    goto B2_eolr;
  B5_eolr:      /* adjust lower */
    lower = mid_pt + 1;
    goto B2_eolr;
    return 0;
  }
```

285. Find Rule or paralleled returned T shift entry.
Algo. binary search 6.2.1 from Knuth Vol. 3.

⟨accrue yacco2 code 33⟩ +≡

```
yacco2::Shift_entry *yacco2::Parser::find_R_or_paralleled_T_shift_entry(yacco2::USINT Enum_id)
{
  ⟨Reserve and get current stack record 352⟩;
  State *State_ptr = pr→state__;
  Shift_tbl *st = State_ptr→shift_tbl_ptr__;
  yacco2::USINT cnt = st→no_entries__;
  Shift_entry_array_type *shft_entry_array = (Shift_entry_array_type *) &st→first_entry__;
  yacco2::Shift_entry *k_entry;
  if (cnt > SEQ_SRCH_VS_BIN_SRCH_LIMIT) goto bin_srch;
  for (int x = 0; x < cnt; ++x) {
    if (x ≥ cnt) break;
    k_entry = &(*shft_entry_array)[x];
    if (Enum_id ≡ k_entry→id__) return k_entry;
    if (Enum_id < k_entry→id__) break;
  }
eolr_seq:
  for (int x = 0; x < cnt; ++x) {
    if (x ≥ cnt) break;
    k_entry = &(*shft_entry_array)[x];
    if (LR1_Eolr ≡ k_entry→id__) return k_entry;
    if (LR1_Eolr < k_entry→id__) return 0;
  }
  return 0;
bin_srch: int lower = 1;
  int upper = cnt;
  int seg_ln;
  int mid_pt;
  int mid_pt_rel0;
B2:   /* calc mid pt */
  if (upper < lower) goto eolr_srch;
  seg_ln = upper + lower;
  mid_pt = seg_ln ≫ 1;
  mid_pt_rel0 = mid_pt − 1;
  k_entry = &(*shft_entry_array)[mid_pt_rel0];
B3:   /* compare */
  if (Enum_id ≡ k_entry→id__) return k_entry;
  if (Enum_id > k_entry→id__) goto B5;
B4:   /* adjust upper */
  upper = mid_pt − 1;
  goto B2;
B5:   /* adjust lower */
  lower = mid_pt + 1;
  goto B2;
eolr_srch:   /* see if all T in set */
  lower = 1;
  upper = st→no_entries__;
B2_eolr:   /* calc mid pt */
  if (upper < lower) return 0;
```

```
    seg_ln = upper + lower;
    mid_pt = seg_ln >> 1;
    mid_pt_rel0 = mid_pt - 1;
    k_entry = &(*shft_entry_array)[mid_pt_rel0];
    if (LR1_Eolr == k_entry->id__) return k_entry;
    if (LR1_Eolr > k_entry->id__) goto B5_eolr;
  B4_eolr:    /* adjust upper */
    upper = mid_pt - 1;
    goto B2_eolr;
  B5_eolr:    /* adjust lower */
    lower = mid_pt + 1;
    goto B2_eolr;
    return 0;
  }
```

286. *add_set_to_map*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
  void add_set_to_map(yacco2::yacco2_set_type &Map, int Partition, int Element)
  {
    yacco2::yacco2_set_iter_type e = Map.find(Partition);
    if (e == Map.end()) {
      Map[Partition] = Element;
    }
    else {
      int se = e->second;
      int v = se + Element;
      e->second = v;
    }
  }
```

287. Reduce Attempts.
The following points detail the order of reduce attempts. Apart from point 1 which is the regular reduce attempt, points 2 and 3 use various meta terminals attempts for different parsing contexts.

1) current token — standard lr(1) reduce
2) meta Tes except |?|, eog, and |||
   in set — eolr, |r|, |.|, |+|, and |t|
3) Only |?| for forced lr(0) reduction

Point 2 is sensitive to the next state's shift attempts — be it wild or ϵ . Point 3 is a specific attempt at drawing the reader's eye to errors within the grammar. It is used in 2 situations:

a) shift with its syntax directed code to deal with the error
b) when in another rule's follow set enforce a reduction

Point b covers the situation whereby the subrule to be reduced will reduce and shift the rule into its next parse state which contains the |?| where the error will be dealt with by its syntax directed code. It is a forcefull reduce instead of considering it an error which it is due to the bad lookahead T by prolonging the error situation to be dealt with by the next parse state environment. This allows the parsing to continue (shift favoured) and to catch the error in the |?| "shift operation" of the new current parse state.

288. Find |?| in reduce lookahead to force a LR(0) reduction.
Algo. binary search 6.2.1 from Knuth Vol. 3. What do u do when the lookahead is faulty (current token) and u want the state's subrule to reduce so as to force the parser into the rule's shift state which deals with the |?| error? Remember the |?| sym has been properly calculated in the lookahead set for the reduce to take place as it is part of the follow set symbol string in the grammar! This is my experiment.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Reduce_entry *yacco2::Parser::find_questionable_sym_in_reduce_lookahead()
  {
    ⟨ Reserve and get current stack record 352 ⟩;
    State *State_ptr = pr→state__;
    UCHAR partition;
    UCHAR element;
    int lower;
    int upper;
    int seg_ln;
    int mid_pt;
    int mid_pt_rel0;
    yacco2::Set_entry *k_entry;
    Reduce_tbl *rt = State_ptr→reduce_tbl_ptr__;
    yacco2::USINT cnt_of_reducing_subrules = rt→no_entries__;
    Reduce_entry *re = (Reduce_entry *) &rt→first_entry__;
    yacco2::Set_tbl *pla_set;
    yacco2::INT no_set_pairs;
    for (yacco2::UINT x = 1; x ≤ cnt_of_reducing_subrules; ++x, ++re) {
      pla_set = re→la_set__;
      no_set_pairs = pla_set→no_entries__;
      Set_entry_array_type *set_entry_array = (Set_entry_array_type *) &(pla_set→first_entry__);
      if (no_set_pairs > SEQ_SRCH_VS_BIN_SRCH_LIMIT) goto QUE_srch;
      for (int x = 0; x < no_set_pairs; ++x) {
        k_entry = &(*set_entry_array)[x];
        if (LRK_LA_QUE_SET.partition__ ≡ k_entry→partition__) {
          if (LRK_LA_QUE_SET.elements__ & k_entry→elements__) {
            return re;
          }
          else {
            break;   /* next reducing rule; not in set */
          }
        }
        if (LRK_LA_QUE_SET.partition__ < k_entry→partition__) break;
      }
      continue;   /* next re */
    QUE_srch:   /* see if meta |?| in set */
      lower = 1;
      upper = no_set_pairs;
    B2_que:   /* calc mid pt */
      if (upper < lower) return 0;
      seg_ln = upper + lower;
      mid_pt = seg_ln ≫ 1;
      mid_pt_rel0 = mid_pt − 1;
      k_entry = &(*set_entry_array)[mid_pt_rel0];
      if (LRK_LA_QUE_SET.partition__ ≡ k_entry→partition__) {
```

```
        if (LRK_LA_QUE_SET.elements__ & k_entry→elements__) {
          return re;
        }
        else {
          continue;    /* this reducing rule not it so next reducing subrule */
        }
      }
      if (LRK_LA_QUE_SET.partition__ > k_entry→partition__) goto B5_que;
    B4_que:    /* adjust upper */
      upper = mid_pt − 1;
      goto B2_que;
    B5_que:    /* adjust lower */
      lower = mid_pt + 1;
      goto B2_que;
    }
    return 0;
  }
```

289. *find_reduce_entry*.
Use own bsearch to speed things up — too much overhead in generic bsearch. See Knuth algo. — variant used shift entry lookup. The reduce table contains a sequential list of potential reducing subrules. Each lookahead set is composed of pairs of set partition with its elements. Each entry is a 2 byte of compressed format. The number of pairs in the table is the 1st byte in the reducing set structure.

The algorithm is potentially a 2 pass over the number of potential reducing subrules in the state. The pecking order is find the current token within the reducing state followed by other attempts of meta symbols, and last the `|?|` symbol.
Pass 1: Is current token in one of the subrule lookahead sets.
If yes then exit with the appropriate reduce entry for that found reducing subrule.

Pass 2: Is the Meta set elements found within one of the reducing subrules?
The Meta symbol LA set elements are Eolr, `|.|`, `|+|`, `|t|`, and `|.|`. If yes then exit with the appropriate subrule's reduce entry having found a meta symbol.

Last gasp: Is `|?|` in the LA sets?.

As an optimization i implicitly use the current token who already has with it the compressed set key to be searched against the lookahead set.

A wrinkle is support of the `|?|`— questionable situations. *has_questionable_shift_occured__* flags its use and so returns the 1st entry as it is a lr(0) context. It is not dependent on the lookahead symbol with its context search.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::Reduce_entry *yacco2::Parser::find_reduce_entry()
  {
    ⟨ Reserve and get current stack record 352 ⟩;
    State *State_ptr = pr→state__;
    UCHAR partition = current_token__→tok_co_ords__.set_entry__.partition__;
    UCHAR element = current_token__→tok_co_ords__.set_entry__.elements__;
    int cp = partition;
    int ce = element;
    Reduce_tbl *rt = State_ptr→reduce_tbl_ptr__;
    yacco2::USINT cnt_of_reducing_subrules = rt→no_entries__;
    Reduce_entry *re = (Reduce_entry *) &rt→first_entry__;
    yacco2::Set_tbl *pla_set;
    yacco2::INT no_set_pairs;
    int lower;
    int upper;
    int seg_ln;
    int mid_pt;
    int mid_pt_rel0;
    yacco2::Set_entry *k_entry;
    if (has_questionable_shift_occured__ ≡ ON) {
      return re;
    }
    ⟨ Pass1: find current tok in potential reducing subrules and exit if fnd 291 ⟩;
    ⟨ Pass2: find meta symbols in potential reducing subrules and exit if fnd 294 ⟩;
    return find_questionable_sym_in_reduce_lookahead();
  }
```

290. Create element's key set.

⟨ create element's key set to be searched in reduce set 290 ⟩ ≡
```
Set_entry la_set;
⟨ create_set_entry 48 ⟩;
```

291. Pass1: find current tok in potential reducing subrules.
Rip thru the potential subrules list looking for mister current token. If found return its subrule's reduce entry. If not found against the subrules reducing LAs then it drops out of the loop and gives controll to Pass2.

⟨ Pass1: find current tok in potential reducing subrules and exit if fnd 291 ⟩ ≡
```
{
Pass1_reduce:
  re = (Reduce_entry *) &rt→first_entry__;
  for (yacco2::UINT x = 1; x ≤ cnt_of_reducing_subrules; ++x, ++re) {
    pla_set = re→la_set__;
    no_set_pairs = pla_set→no_entries__;
    Set_entry_array_type *set_entry_array = (Set_entry_array_type *) &(pla_set→first_entry__);
    if (no_set_pairs > SEQ_SRCH_VS_BIN_SRCH_LIMIT) {
      ⟨ binary search for token in current subrule la 293 ⟩;
    }
    else {
      ⟨ sequential search for token in current subrule la 292 ⟩;
    }
  }
}
```
This code is used in section 289.

292. Sequential search for token in current subrule la.

⟨ sequential search for token in current subrule la 292 ⟩ ≡
```
for (int xx = 0; xx < no_set_pairs; ++xx) {
  k_entry = &(*set_entry_array)[xx];
  if (partition ≡ k_entry→partition__) {
    if (element & k_entry→elements__) {
      return re;
    }
    else {
      break;    /* next reducing rule; not in set */
    }
  }
  if (partition < k_entry→partition__) break;
}
```
This code is used in section 291.

293. Binary search for token in current subrule la.

⟨ binary search for token in current subrule la 293 ⟩ ≡

```
{
bin_srch_cur_tok:
   lower = 1;
   upper = no_set_pairs;
B2:      /* calc mid pt */
   if (upper < lower) goto srch_end_cur_tok;
   seg_ln = upper + lower;
   mid_pt = seg_ln >> 1;
   mid_pt_rel0 = mid_pt - 1;
   k_entry = &(*set_entry_array)[mid_pt_rel0];
B3:      /* compare */
   if (partition == k_entry->partition__) {
      if (element & k_entry->elements__) {
         return re;
      }
      else {
         goto srch_end_cur_tok;      /* T not in LA */
      }
   }
   if (partition > k_entry->partition__) goto B5;
B4:      /* adjust upper */
   upper = mid_pt - 1;
   goto B2;
B5:      /* adjust lower */
   lower = mid_pt + 1;
   goto B2;
srch_end_cur_tok: ;
}
```

This code is used in section 291.

294. Pass2: find meta symbols in potential reducing subrules.
Rip thru the potential subrules list looking for meta symbols. If found return its subrule's reduce entry. If not found against the subrules reducing LAs then it drops out of the loop and gives controll to the last Gasp.

⟨ Pass2: find meta symbols in potential reducing subrules and exit if fnd 294 ⟩ ≡

```
{
   re = (Reduce_entry *) &rt->first_entry__;
   for (yacco2::UINT x = 1; x <= cnt_of_reducing_subrules; ++x, ++re) {
      pla_set = re->la_set__;
      no_set_pairs = pla_set->no_entries__;
      Set_entry_array_type *set_entry_array = (Set_entry_array_type *) &(pla_set->first_entry__);
      if (no_set_pairs > SEQ_SRCH_VS_BIN_SRCH_LIMIT) {
         ⟨ binary search for meta symbol in current subrule la 296 ⟩;
      }
      else {
         ⟨ sequential search for meta symbol in current subrule la 295 ⟩;
      }
   }
}
```

This code is used in section 289.

295. Sequential search for meta symbol in current subrule la.

⟨ sequential search for meta symbol in current subrule la 295 ⟩ ≡

```
{
seq_meta:
  for (int x = 0; x < no_set_pairs; ++x) {
    k_entry = &(*set_entry_array)[x];
    if (LRK_LA_EOLR_SET.partition__ ≡ k_entry→partition__) {
      if (LRK_LA_EOLR_SET.elements__ & k_entry→elements__) {
        return re;
      }
      else {
        break;     /* next reducing rule; not in set */
      }
    }
    if (LRK_LA_EOLR_SET.partition__ < k_entry→partition__) break;
  }
}
```

This code is used in section 294.

296. Binary search for meta symbol in current subrule la.

⟨ binary search for meta symbol in current subrule la 296 ⟩ ≡

```
{
bin_srch_meta: lower = 1;
  upper = no_set_pairs;
Meta_srch:   /* see if meta Ts in set */
  lower = 1;
  upper = no_set_pairs;
B2_meta:   /* calc mid pt */
  if (upper < lower) {
    continue;   /* next subrule return 0; */
  }
  seg_ln = upper + lower;
  mid_pt = seg_ln >> 1;
  mid_pt_rel0 = mid_pt - 1;
  k_entry = &(*set_entry_array)[mid_pt_rel0];
  if (LRK_LA_EOLR_SET.partition__ == k_entry->partition__) {
    if (LRK_LA_EOLR_SET.elements__ & k_entry->elements__) {
      return re;
    }
    else {
      continue;   /* this reducing rule no meta so next reducing subrule */
    }
  }
  if (LRK_LA_EOLR_SET.partition__ > k_entry->partition__) goto B5_meta;
B4_meta:   /* adjust upper */
  upper = mid_pt - 1;
  goto B2_meta;
B5_meta:   /* adjust lower */
  lower = mid_pt + 1;
  goto B2_meta;
}
```

This code is used in section 294.

297. *find_parallel_reduce_entry*.

See "Notes to myself". This is a lr(0) reduction. So pick up the first entry in the table. This forces a reduction to take place regardless of the "lookahead" token. It allows the calling parser to complete the reduction and then use the "shift" mechanism of |.|, |+| to catch errors.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Reduce_entry *yacco2::Parser::find_parallel_reduce_entry()
{
  ⟨Reserve and get current stack record 352⟩;
  State *State_ptr = pr->state__;
  Reduce_tbl *rt = State_ptr->reduce_tbl_ptr__;
  Reduce_entry *re = (Reduce_entry *) &rt->first_entry__;
  return re;
}
```

298. *find_proc_call_reduce_entry*.
See "Notes to myself". This is a lr(0) reduction. So pick up the first entry in the table. This forces a reduction to take place regardless of the "lookahead" token. It allows the calling parser to complete the reduction and then use the "shift" mechanism of |.|, |+| to catch errors.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Reduce_entry *yacco2::Parser::find_proc_call_reduce_entry()
{
  ⟨ Reserve and get current stack record 352 ⟩;
  State *State_ptr = pr→state__;
  Reduce_tbl *rt = State_ptr→reduce_tbl_ptr__;
  Reduce_entry *re = (Reduce_entry *) &rt→first_entry__;
  return re;
}
```

299. Start token routines.

300. *start_token*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::CAbs_lr1_sym *yacco2::Parser::start_token()
{
  return start_token__;
}
```

301. *set_start_token*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_start_token(CAbs_lr1_sym &Token)
{
  start_token__ = &Token;
}
```

302. *start_token_pos*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::UINT yacco2::Parser::start_token_pos()
{
  return start_token_pos__;
}
```

303. *set_start_token_pos*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_start_token_pos(yacco2::UINT Pos)
{
  start_token_pos__ = Pos;
}
```

304. All shift routines.
These routines control how the parser reacts to the |+| all shift terminal. As this terminal is never in the token stream, it is a condition that the parser checks within the current state's configuration. If the facility is on and the 'all shift' terminal is present in the current PDA's state, then the parser shifts the terminal. Not on or present, the parser tries the next inline operation which is a reduce. The parser favors shifting over reducing. It is turned on both at initialization time and reset time after a parallel parse.

It is up to the grammar writer to turn off this facility. To shutoff this facility, usually the syntax directed code tests for a specific terminal by its enumeration id during the shift operation. Shuting off of the facility allows the grammar to complete instead of sitting in an open loop of consuming terminals until an overrun occurs against the token stream.

305. *set_use_all_shift_on*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_use_all_shift_on()
{
  use_all_shift__ = ON;
}
```

306. *set_use_all_shift_off*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::set_use_all_shift_off()
{
  use_all_shift__ = OFF;
}
```

307. *use_all_shift*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
bool yacco2::Parser::use_all_shift()
{
  return use_all_shift__;
}
```

308. Parser symbol table functor and abort, stop routines.

309. *sym_lookup_functor*.
This is your imported functor used to do token remapping: another term for symbol table handling. The functor is specific to the language being parsed. It has been tested against the Pascal language and Yacco2's grammar. Of course *cweb* was used to develop these symbol tables.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::tble_lkup_type *yacco2::Parser::sym_lookup_functor()
{
  return sym_lookup_functor__;
}
```

310. *abort_parse*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
bool yacco2::Parser::abort_parse()
{
  return abort_parse__;
}
```

311. *set_abort_parse*.
Used to abort abruptly a parse. Not too subtle. Directs the parser to do its abort-winddown thing.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_abort_parse(bool Abort)
{
  abort_parse__ = Abort;
}
```

312. *stop_parse*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
bool yacco2::Parser::stop_parse()
{
  return stop_parse__;
}
```

313. *set_stop_parse*.
Used to stop a parse. This is much more refined as one can place an error token into the accept queue for grammatical error processing and come to a gentle stop. This is a refinement to an abort. It does the same thing as abort in its cleanup except that it is considered a successful parse. This process is a grammar writer's statement within syntax directed code whereas the abort comes from 2 sources: the grammar writer's syntax directed code or an invalid token stream causing the parse thread to abort. Cavate: You still must use the `RSVP` macro to place the token into the accept queue. If you don't, you'll get a runtime check due to the accepted token (current lookahead token) having the same lookahead token boundary.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_stop_parse(bool Stop)
{
  stop_parse__ = Stop;
}
```

314. Parser's FSM support routines.

315. *fsm_tbl***.**
Just the fsm automaton ensemble.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::CAbs_fsm *yacco2::Parser::fsm_tbl()
{
  return fsm_tbl__;
}
```

316. Parse containers. The four containers are:

Token supplier — input token stream to parser
Token producer — receives output from the parser for next stage processing
Error — container of error terminals
Recycle — ecological bio-degradable

As containers are template driven due to the diversity of inputs, there are 2 typedefs describing containers. **token_container_type** is a **tok_can** based template that other containers inherit from. Used by the **error** queue is the **TOKEN_GAGGLE** container based on a vector template.

The 2 variants of an input queue are the source file to raw character conversion, and the regular supplier queue that feeds the lexical and syntatic parse stages. These are specialized templates.

Another container handles tree related structures with their associated walkers and terminal filter functors. This allows one to process a tree as a stream of tokens that get digested by a grammar. The filter has a complement indicator as to include or exclude the terminal types in the filter set. This eases the declaration task of the compiler writer. Given a large population of terminal types, the set of exclusion terminal enumerates minimizes the effort of unwanted terminals in the parse stream. The same holds for a small number of terminals for processing using inclusion. See *tree containers*.

```
⟨ Type defs 16 ⟩ +≡
  struct Caccept_parse;
#define pp_accept_queue_size 8
  typedef yacco2::Caccept_parse pp_accept_queue_type[pp_accept_queue_size];
```

317. Supplier container.
This is your standard token dispensor that feeds a parser. Due to parallelism, there is only 1 supplier of tokens. Somewhere in the call chain of the threads there is a token dispensor. It is always supplied by the calling grammar to its threads. The container is a “many reader” to the called threads of parallelism.

318. *token_supplier*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::token_container_type *yacco2::Parser::token_supplier( )
  {
    return token_supplier__;
  }
```

319. *set_token_supplier*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::set_token_supplier(yacco2::token_container_type &Token_supplier)
  {
    token_supplier__ = &Token_supplier;
  }
```

320. *add_token_to_supplier*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::add_token_to_supplier(yacco2::CAbs_lr1_sym &Token)
  {
    if (token_supplier__→r_w_cnt__ > 1) ⟨ acquire token mu 391 ⟩;
    token_supplier__→push_back(Token);
    if (token_supplier__→r_w_cnt__ > 1) ⟨ release token mu 392 ⟩;
  }
```

321. Producer container.
Receiver of outputted terminals from a parse stage. It normally becomes a supplier queue to a down stream parse stage.

322. *token_producer*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::token_container_type *yacco2::Parser::token_producer()
{
  return token_producer__;
}
```

323. *set_token_producer*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_token_producer(yacco2::token_container_type &Token_producer)
{
  token_producer__ = &Token_producer;
}
```

324. *add_token_to_producer*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::add_token_to_producer(yacco2::CAbs_lr1_sym &Token)
{
  ⟨ acquire token mu 391 ⟩;
  token_producer__->push_back(Token);
  ⟨ release token mu 392 ⟩;
}
```

325. Recycle container.
A holder of tokens that need to be postprocessed. Typical use is to remove tokens out of a token stream but will be re-integrated later back into some other token stream. For example, a translator that retargets one language into another and the comments need re-integrating back into the targetted output.

326. *recycle_bin*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::token_container_type *yacco2::Parser::recycle_bin()
{
  return recycle_bin__;
}
```

327. *set_recycle_bin*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_recycle_bin(yacco2::token_container_type &Recycle_bin)
{
  recycle_bin__ = &Recycle_bin;
}
```

328. *add_token_to_recycle_bin*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::add_token_to_recycle_bin(yacco2::CAbs_lr1_sym &Token)
{
  ⟨ acquire token mu 391 ⟩;
  recycle_bin__→push_back(Token);
  ⟨ release token mu 392 ⟩;
}
```

329. Error queue.
Just a holding container for error terminals. I use this container to express warnings and errors within Yacco2. If one is creative, error sentences can be outputted that will be later parsed by an error grammar. This is how Yacco2 handles its errors outputted to the grammar writer by matching the errors to the source file co-ordinates. The error queue is just another input queue to be parsed. Error sentences can be expressed be it of a single token to a complete language of various structures. To process the errors, it can be as simple as iterating through the container, to use a grammar having only the 'all shift' facility, to grammars describing the error language.

330. *set_error_queue*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::set_error_queue(yacco2::token_container_type &Error_queue)
{
  error_queue__ = &Error_queue;
}
```

331. *error_queue*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::token_container_type *yacco2::Parser::error_queue()
{
  return error_queue__;
}
```

332. *add_token_to_error_queue*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::add_token_to_error_queue(yacco2::CAbs_lr1_sym &Token)
{
  ⟨ Validate error queue 557 ⟩;
  ⟨ acquire token mu 391 ⟩;
  error_queue__→push_back(Token);
  ⟨ release token mu 392 ⟩;
}
```

333. Accept queue `RSVP`, `RSVP_FSM`, `RSVP_WLA` macro use comments.
This is an array where the arbitrator's syntax directed code tests against it for the specific presence of an accepted token. For example, the terminals 'identifier' and 'keyword' are parallel competitors. The arbitrator needs to test if the keyword is present to throw away the identifier.

The `RSVP` macro is used to added to the parser's accept queue from within the grammar's rule context. The `RSVP_WLA` macro is used to added to the parser's accept queue and to use its lookahead parameters instead of the defaults. The `RSVP_FSM` macro is used to added to the parser's accept queue from within the fsm's context. *put_T_into_accept_queue* is another way to do it.

334. Put potential Caccept_parse into accept queue.
Caccept_parse is just a carrier of the real terminal contained inside it. The parallel thread submitting its result to the accept queue already has ownership of *pp_requesting_parallelism__*'s mutex. *pp_accept_queue__* is an array where the 0 subscript does nothing.

The parameter is needed as this is the context of the called thread who is placing its contents into the calling thread's accept queue.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::put_T_into_accept_queue(yacco2::Caccept_parse &Calling_parm)
{
  ++th_accepting_cnt__;
  if (th_accepting_cnt__ < pp_accept_queue_size) {
    pp_accept_queue__[th_accepting_cnt__].fill_it(Calling_parm);   /* copy its contents */
  }
  else {   /* throw error */
    abort_accept_queue_irregularites(Calling_parm);
  }
}
```

335. Token Get routines: specific stack token, next token in stream. A word on the subscript used to access a container's content. I'm not a fan of relative-to-zero situations. I count by 1 and a 2 and a... Lawrence Welk anyone? Just because its more efficient to access an array by relative-to-zero subscripts doesn't mean that I must adhold to this. So what are the options. Sit quite and be efficient... ugh. Hear my teeth grinding? Subtract 1 from the subscript every time the container is being accessed: a bit too expensive — what, can't u count this way? Put a boggus record at container creation time into the zero position of the container. Humm — consider it a bs record: before start. Now what are the merits: no calculation required, Dave can count, and no off-by-one situations. Now the demerits: extra space, must watch to skip over the first item in the container if iterators are used. Oh well. Come on u old dog or is it Humpty Dumpty had a great... No, one is one and that's it. For now the relative-to-zero works.

To integrate symbol table facilities into the Yacco2, a functor was created. Appropriate *cweb* macros were written to easy the pain. *Remap_token* retargets the token read from the input stream. It clones off the token having the same source co-ordinates. Its logic est tres simple:

1) is there a symbol table functor present: no return token fetched
2) is symbol table lookup turned off: yes return token fetched
3) try look up: if returned token is nil return the fetched token
4) return the looked up token

There are 2 companion *cweb* macros: *Remap_set_result_and_return* and *Remap_return_result*. The first macro takes the symbol table's returned token and sets it as the parser's current token and returns the new token. *Remap_return_result* just returns the retargeted token used by *get_spec_token* which is a random query of a token stream. Remapped tokens eventually get put into other token containers for down stream processing.

```
#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;
```

336. *get_spec_stack_token*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::Parser::get_spec_stack_token(yacco2::UINT Pos)
  {
    if (Pos > MAX_LR_STK_ITEMS) return 0;    /* is_pos_within_bnds */
    Cparse_record *pr = parse_stack__.sf_by_sub(Pos);
    return pr→symbol__;
  }
```

337. *get_next_token***.** Due to the "jit" accessing the mutex guarding the container read is NEEDED. Tests between not "jit" versus "jit" with mutex yielded just 3 seconds difference across 80 compiles. SO KEEP IT.

Some subtle comments on overflow per token container.
The container template implements the access [] operator which guards against overflow. It returns the "eog" token to indicate end-of-token stream reached. In this context the end-of-token stream depends on the specific container. From a tree container's perspective, the container's size is open-ended and its internal tree walking stack determines whether it has been reached. It returns the maximum unsigned integer value within its size method which forces a call using the access operator []. So the size method is not quite accurate though the other containers are.

But what is your problem Dave? When porting to VMS/Alpha, the implemented virtual method of the container template did not execute the **TOKEN_GAGGLE** container's virtual operator [] which tests its internal state before accessing its own internal stl array container's access operator. **TOKEN_GAGGLE** container is specificly declared for the "Error queue" while all the other containers used in parsing like Supplier and Producer are abstract **tok_base** type which forces the compiler to call the implemented virtual table of the container to deal with size, [] and other methods. **tok_base** enforces regularity. When parsing the "Error queue" aka **TOKEN_GAGGLE** using a grammar/Parser approach, the native container's [] operator and not the virtual method was called and so aborted on "array bounds exceeded" error. This is why the pre and post overflow evaluation before calling the container's access [] operator. The first check is "has overflow already happened" and so don't increment *current_token_pos__*, just reset the *current_token__* to "eog" and exit. This keeps the internal token subscript accurate. The post overflow evaluation is after the *current_token_pos__* increment to see if it just reached the end-of-token stream condition and so set *current_token__* to "eog" and exit.

Extracting the token from the container:
So now the Parser's token container needs to be called to get its next token with the incremented subscript. It is up to the token container's implementation to determine whether the token is within its internal stl's container's bounds. The subscript is checked against the stl container's size method for the overflow condition and to take appropriate action which is return the "eog" token back. Finally the internal stl's container is accessed by its [] operation to extract the called for token.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::Parser::get_next_token(){
      if (token_supplier__ ≡ 0) return 0;      /* is_there_a_token_supplier: */
      if (token_supplier__→empty() ≡ true) {      /* out-of-bnds: protect current pos */
        current_token__ = yacco2::PTR_LR1_eog__;
        return current_token__;
      }
      if (current_token_pos__ ≥ token_supplier__→size()) {      /* out-of-bnds: protect current pos */
        current_token__ = yacco2::PTR_LR1_eog__;
        return current_token__;
      }
      ++current_token_pos__;
      if (current_token_pos__ ≥ token_supplier__→size()) {      /* out-of-bnds: protect current pos */
        current_token__ = yacco2::PTR_LR1_eog__;
        return current_token__;
      }
      if (YACCO2_T__ ≠ 0) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪
            "␣get_next_token::␣pos␣to␣fetch:␣" ≪ current_token_pos__ ≪ FILE_LINE ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
```

```
}
current_token__ = (*token_supplier__)[current_token_pos__];
if (YACCO2_T__ ≠ 0) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪
      "␣get_next_token::␣pos:␣" ≪ current_token_pos__ ≪ "␣␣enum:␣" ≪
      current_token__→enumerated_id__ ≪ '␣' ≪ '"' ≪ current_token__→id__ ≪ '"' ≪
      "␣token␣fetched*:␣" ≪ current_token__ ≪ FILE_LINE ≪ std::endl;
  yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
  EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
      current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
      current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
  if (yacco2::YACCO2_MU_TRACING__) {
    yacco2::lrclog ≪ "YACCO2_MU_TRACING__::Releasing␣trace␣mu" ≪ FILE_LINE ≪ std::endl;
  }
  EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
      current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
      current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
Remap_token(current_token__)
if ((YACCO2_T__ ≠ 0) ∧ (sym_lookup_functor__ ≠ 0)) {
  if (sym_lookup_functor__→lkup__ ≡ ON ≠ 0) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪
        "␣get_next_token::␣pos:␣" ≪ current_token_pos__ ≪ "␣␣enum:␣" ≪
        current_token__→enumerated_id__ ≪ '␣' ≪ "␣after␣remap␣" ≪ '"' ≪
        current_token__→id__ ≪ '"' ≪ "␣token␣fetched*:␣" ≪ current_token__ ≪ FILE_LINE ≪
        std::endl;
    yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
    EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
        current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
        current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
Remap_set_result_and_return(current_token__)
}
```

338. *get_spec_token*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::CAbs_lr1_sym *yacco2::Parser::get_spec_token(yacco2::UINT Pos){
    ⟨ Validate if parser's supplier exists 552 ⟩;
    ⟨ Validate if subscript within supplier's bnds 553 ⟩;
    ⟨ any tokens in container? no return nil ptr 339 ⟩;
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name( ) ≪ "::" ≪
          "␣get_spec_token␣pos:␣" ≪ Pos ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
    CAbs_lr1_sym *token = (*token_supplier__)[Pos];
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name( ) ≪ "::" ≪
          "␣get_spec_token:␣returned␣token␣" ≪ token→id__ ≪ "␣pos:␣" ≪ Pos ≪ "␣enum:␣" ≪
          token→enumerated_id__ ≪ '"' ≪ token→id__ ≪ '"' ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
      EXTERNAL_GPSing(token)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪ token→tok_co_ords__.line_no__ ≪
          "␣GPS␣CHR␣POS:␣" ≪ token→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
    Remap_token(token)
    if ((YACCO2_T__ ≠ 0) ∧ (sym_lookup_functor__ ≠ 0)) {
      if (sym_lookup_functor__→lkup__ ≡ ON ≠ 0) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name( ) ≪ "::" ≪
            "␣get_spec_token:␣returned␣token␣" ≪ token→id__ ≪ "␣pos:␣" ≪ Pos ≪
            "␣enum:␣" ≪ token→enumerated_id__ ≪ "␣after␣remap␣" ≪ '"' ≪ token→id__ ≪ '"' ≪
            FILE_LINE ≪ std::endl;
        yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
        EXTERNAL_GPSing(token)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
            token→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪ token→tok_co_ords__.pos_in_line__ ≪
            FILE_LINE ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
    }
    Remap_return_result
    }
```

339. Any tokens in container?. no return nil ptr.

⟨ any tokens in container? no return nil ptr 339 ⟩ ≡

```
if (token_supplier__→empty( ) ≡ YES) return 0;
```

This code is used in section 338.

340. Parse stack routines. Currently the subscript to access the stack is relative to ONE.

341. *cleanup_stack_due_to_abort*. The last item on the stack is left so that the thread can be re-used. This is why its one less for the popping. The thread sits idle, twirling its whatever until a requesting grammar asks to be serviced.

342. *cleanup_stack_due_to_abort*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::cleanup_stack_due_to_abort( )
{
  yacco2::INT stack_items_to_process = parse_stack__.top_sub__ − 1;
  if (stack_items_to_process > 0) {
    remove_from_stack(stack_items_to_process);
  }
  set_abort_parse(OFF);
  set_stop_parse(OFF);
}
```

343. *current_stack_pos*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::INT yacco2::Parser::current_stack_pos( )
{
  return parse_stack__.top_sub__;
}
```

344. *parse_stack*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::lr_stk *yacco2::Parser::parse_stack( )
{
  return &parse_stack__;
}
```

345. *top_stack_record*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::Cparse_record *yacco2::Parser::top_stack_record( )
{
  if (parse_stack__.top_sub__ < 1) return 0;    /* if(parse_stack__.empty() == YES) return 0; */
  ⟨ Reserve and get current stack record 352 ⟩;
  return pr;
}
```

346. *get_stack_record*.
The subscript of stack is rel 1 not 0 while the request is rel to 0. In between counting strategies: Ugh!

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::Cparse_record *yacco2::Parser::get_stack_record(yacco2::INT Pos)
{
  ⟨ Validate subscript not ≤ 0 554 ⟩;
  if (Pos ≥ (parse_stack__.top_sub__)) return 0;
  return parse_stack__.sf_by_sub(Pos + 1);
}
```

347. *no_items_on_stack*.
Twist no oliver, it returns one less than whats on the stack. The reason is the first stack record, which is the *start state* of the finite automaton, is always maintained for optimization reasons. This allows the parser to begin just start when its re-commissioned to work. Normally calling *no_items_on_stack* is a general way to winddown the parse be it successful or aborted.

⟨ accrue yacco2 code 33 ⟩ +≡
```
yacco2::INT yacco2::Parser::no_items_on_stack()
{
  return parse_stack__.top_sub__;
}
```

348. Add state to parse stack *add_to_stack*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::add_to_stack ( yacco2::State &State )
    {
      parse_stack__.push_state(State);
      ⟨ Trace TH the parse stack configuration 581 ⟩;
    }
```

349. Add to parse stack — Speed Demon.

⟨ *add_to_stack* 349 ⟩ ≡
```
⟨ lr_stk::push_state 132 ⟩;    /* ¡Trace TH the parse stack configuration¿; */
```
This code is used in sections 236, 238, 240, 241, 245, 266, 268, 417, and 420.

350. Remove items from the parse stack *remove_from_stack*.
Parse stack is a LIFO order of $< state* : sym* >$ configuration pairs. The parse stack configuration for S1 shifting 'a' into S2 has 2 records. The first record contains as an example without the pointer $1 : `a'$. Symbol 'a' is the shift item that takes the finite state from state 1 into state 2. The second record contains the entered state $2 : nil$. There is no symbol as the next parse action has not happened.

This routine also cleans up aborted parses. It always leaves the first parse record on the stack as an optimization as the thread is snapping its fingers for the next message request to parse.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::remove_from_stack(yacco2::INT No_to_remove)
{
  ⟨ Validate parse stack number of removal items 555 ⟩;
  ⟨ Validate parse stack removal for underflow 556 ⟩;
  ⟨ Check parse stack for epsilon removal. yes exit 351 ⟩;
  ⟨ Remove items from the parse stack 361 ⟩;
}
```

351. Check parse stack for epsilon removal *remove_from_stack*.

⟨ Check parse stack for epsilon removal. yes exit 351 ⟩ ≡
```
if (No_to_remove ≡ 0) {
  ⟨ Trace TH when an epsilon rule is being reduced 582 ⟩;
  return;
}
```
This code is used in section 350.

352. Reserve and get current stack record.

⟨ Reserve and get current stack record 352 ⟩ ≡
  Cparse_record *$*pr$* = *parse_stack__.top__*;

This code is used in sections 236, 238, 240, 241, 251, 265, 267, 271, 284, 285, 288, 289, 297, 298, 345, and 362.

353. Get current stack record.

⟨ Get current stack record 353 ⟩ ≡
  pr = *parse_stack__.top__*;

This code is used in sections 256, 258, 260, 262, 356, and 361.

354. Initialize stack record.

⟨ Initialize stack record 354 ⟩ ≡
  pr→symbol__ = 0;
  pr→aborted__ = 0;
  pr→rule_s_reuse_entry_ptr__ = 0;

This code is used in sections 356 and 361.

355. Pop parse stack.

⟨ Pop parse stack 355 ⟩ ≡
  −−*parse_stack__.top_sub__*;
  −−*parse_stack__.top__*; /* parse_stack__.pop(); */

This code is used in section 356.

356. Clean up parse stack record and pop from stack.
When the state is popped, the exposed record is the state:symbol pair used by the finite automaton to map into the state just popped.

⟨ Clean up parse stack record and pop state from stack exposing symbol record 356 ⟩ ≡
  ⟨ Initialize stack record 354 ⟩;
  ⟨ Pop parse stack 355 ⟩;
  ⟨ Get current stack record 353 ⟩; /* symbol record */

This code is used in section 361.

357. Check for zeroed out symbol on parse stack.
This situation can happen if the grammar user plays with the stack's symbols. Once apon a time, meta symbols were zeroed out to protect from deletion due to their re-cycled nature: for example the parallel and invisible shift symbols are created once and recycled many times throughout the parse history. Now these symbols are protected by having their *auto_delete* attribute turned off.

⟨ Check for zeroed out symbol on parse stack. If so goto next element to remove 357 ⟩ ≡
  if (*pr→symbol__* ≡ 0) {
    ⟨ Trace TH zeroed out symbol situation when popped from parse stack 584 ⟩;
    goto *next_stack_element_to_remove*;
  }

This code is used in section 361.

358. Is popping symbol auto deleted?.
This deals with the grammar symbol's 'AD' attribute. Due to MSN and their bug brigade, , the delete arttribute is commented out. So the memory heap just grows but with no occasional aborts. When the parser stops, it's left to the operating system to reset the heap allocated to the program.

⟨ Is popping symbol auto deleted? then deal with it and goto next element to remove 358 ⟩ ≡
```
if (pr→rule_s_reuse_entry_ptr__ ≠ 0) {
  fsm_tbl__→recycle_rule(pr→rule_s_reuse_entry_ptr__);
  pr→rule_s_reuse_entry_ptr__ = 0;    /* wipe off the rule from the "in use" slate */
}
else {
  if (pr→symbol__→auto_delete__ ≡ ON) {
    ⟨ Trace TH advise when symbol deleted due to AD switch 586 ⟩;
    if (pr→symbol__→dtor__ ≠ 0) (*pr→symbol__→dtor__)(pr→symbol__, this);
    delete pr→symbol__;
    pr→symbol__ = 0;    /* keep that stack clean */
    goto next_stack_element_to_remove;
  }
}
```
This code is used in section 361.

359. Check for aborted parse situation.
If the parse record is clean, then goto next element to remove.

⟨ Check for aborted parse situation. If clean goto next element to remove 359 ⟩ ≡
```
if (pr→aborted__ ≡ 0) goto next_stack_element_to_remove;
```
This code is used in section 361.

360. Deal with auto abort.
This is the grammar symbol's 'AB' attribute. It checks to see if there is a destructor function to run.

⟨ Deal with auto abort 360 ⟩ ≡
```
if (pr→rule_s_reuse_entry_ptr__ ≠ 0) {
  fsm_tbl__→recycle_rule(pr→rule_s_reuse_entry_ptr__);
  pr→rule_s_reuse_entry_ptr__ = 0;    /* wipe off the rule from the "in use" slate */
}
else {
  if (pr→symbol__→affected_by_abort__ ≡ OFF) goto next_stack_element_to_remove;
  if (pr→symbol__→dtor__ ≠ 0)
    (*pr→symbol__→dtor__)(pr→symbol__, this);
  delete pr→symbol__;
}
```
This code is used in section 361.

361. Remove items from the parse stack.
The remove routine is a straddler. The number of records to pop is the appropriate grammar's subrule: all the king's men... The straddler part is how the PDA works: the top record is the state just entered. The symbol that vectored into it is one back. This is the straggler. So one is popping the vectored into state leaving the exposed symbol record. This holds for accepted and aborted parse situations. The Start state record is always on the stack: even at parse shutdown as there is nothing to clean up.

```
⟨Remove items from the parse stack 361⟩ ≡
  Cparse_record *pr;
  ⟨Get current stack record 353⟩;
  ⟨Trace TH remove items from the parse stack configuration 579⟩;
  while (No_to_remove > 0) {
    ⟨Trace TH popped state no 583⟩;
    ⟨Clean up parse stack record and pop state from stack exposing symbol record 356⟩;
    ⟨Check for zeroed out symbol on parse stack. If so goto next element to remove 357⟩;
    ⟨Trace TH exposed symbol on parse stack 585⟩;
    ⟨Is popping symbol auto deleted? then deal with it and goto next element to remove 358⟩;
    ⟨Check for aborted parse situation. If clean goto next element to remove 359⟩;
    ⟨Trace TH advise when auto abort happening 587⟩;
    ⟨Deal with auto abort 360⟩;
    ⟨Initialize stack record 354⟩;
  next_stack_element_to_remove:
    --No_to_remove;
  }
  ⟨Trace TH finished removing items from the parse stack configuration 580⟩;
```

This code is used in section 350.

362. *clear_parse_stack*.

```
⟨accrue yacco2 code 33⟩ +≡
  void yacco2::Parser::clear_parse_stack()
  {
    yacco2::INT s = parse_stack__.top_sub__ − 1;   /* always leave 1st parse record */
    if (s > 0) remove_from_stack(s);
    if (s ≡ 0) {   /* cleanse possible acceptance start rule */
      ⟨Reserve and get current stack record 352⟩;
      if (pr→rule_s_reuse_entry_ptr__ ≠ 0) {   /* don't need hanging around like a dirty smell */
        pr→rule_s_reuse_entry_ptr__ = 0;   /* already recycled */
      }
    }
  }
```

363. Token Get, Reset, Override Flavours: *current_token*, *reset_current_token*, **etc.**

364. *current_token*.
It checks whether it has a symbol table lookup functor. If it does not exist or the facility is turned off, the current terminal is returned. The table lookup will try to remap a generic terminal. The terminal remapped can be anything. This is dependent on the functor written for the language being compiled.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2::CAbs_lr1_sym *yacco2::Parser::current_token(){Remap_token(current_token__)
      Remap_set_result_and_return(current_token__)
      }
```

365. Reset current token.
reset_current_token 15 micro seconds of fame by re-aligning the calling parser's current token's co-ordinate within the token stream using the *Pos* parameter.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::reset_current_token(yacco2::UINT Pos)
  {
    ⟨ Validate if parser's supplier exists 552 ⟩;
    ⟨ Validate if subscript within supplier's bnds 553 ⟩;
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪ "::" ≪
          "␣reset_current_token␣pos:␣" ≪ Pos ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
    current_token_pos__ = Pos;
    current_token__ = (*token_supplier__)[Pos];
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪ "::" ≪
          "␣reset_current_token:␣token␣to:␣" ≪ current_token__→id__ ≪ "␣pos:␣" ≪
          current_token_pos__ ≪ "␣enum:␣" ≪ current_token__→enumerated_id__ ≪ '"' ≪
          current_token__→id__ ≪ '"' ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
      EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
          current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
          current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
```

366. *override_current_token*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::override_current_token(yacco2::CAbs_lr1_sym &Token, yacco2::UINT Pos)
  {
    current_token_pos__ = Pos;
    current_token__ = &Token;
  }
```

367. *override_current_token_pos*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::override_current_token_pos(yacco2::UINT Pos)
{
  current_token_pos__ = Pos;
}
```

368. *current_token_pos*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::UINT yacco2::Parser::current_token_pos()
{
  return current_token_pos__;
}
```

369. Get shift's next token *get_shift_s_next_token*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::get_shift_s_next_token()
{
  get_next_token();
}
```

370. Thread name of grammar that is a thread. Monolithic grammars use their "fsm" name.

371. *thread_name*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::KCHARP yacco2::Parser::thread_name()
{
  return thread_name__;
}
```

372. Thread entry.
Contains all the dirt about the thread. This entry is *nil* if its a monolithic grammar. This entry's thread id is used as the key into the parallel thread global table.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Thread_entry *yacco2::Parser::thread_entry()
{
  return thread_entry__;
}
```

373. Thread "hows and whys" on thread activation.
There are just 2 critical region classifications:

1) launched threads' table
2) each grammar's threading region

Each grammar's threading region supports the framework for inter-thread communications: messaging (re: events) and acceptance token queue — tokens passed back as results from a thread's execution.
Messaging components:
The *th_active_cnt__* and *th_accepting_cnt__* are variables that are dynamicly set at each thread launch invocation within the launching grammar. The number of attempted parallel parses is indicated by the *th_active_cnt__* which is the launched number of threads. As each thread stops processing, it decrements the counter of the launching grammar. When the counter reaches 0, it is that thread's responsibility to notify the sleeping *pp* parser by event to wake up and assess the parallel parse results. *th_accepting_cnt__* is the number of accept messages placed into the message queue by successful parallel parses. This number can be 0 indicating that all the attempted parallel parses have failed.

Originally the control monitor was the go between for the grammar requesting parallelism and the threads controlled by it. Now the requesting grammar launches the threads given by the its fa's configuration state. A little optimization is done by the requesting grammar: only launch threads whose first set contains the current token. The launching first checks if the thread is in the global thread table and that it is available for work.

To further the pursuit of speed, variables *no_competing_pp_ths__* and *no_requested_ths_to_run__* determine how the threads should be executed within the local context of the launched grammar. If there is only 1 thread to launch, it is executed as a procedure call without the thread baggage and its critical region entourage (not any more: pure thrrreading in the scotish roll of "r"). Why the 2 variables? *no_competing_pp_ths__* tells the current thread how many others are competing and have been launched by the requesting grammar. Without it being local the threaded grammar needs to acquire the mutex of its caller to determine the number of launched threads. It is a read-only variable that receives its value from the requesting grammar's *no_requested_ths_to_run__* variable at start up time. If this grammar requests parallelism, it sets its own *no_requested_ths_to_run__* variable and calls the appropriate threads who in term set their *no_competing_pp_ths__* variable at their invocation time. The nesting of threads requires this 2 variable approach: read-only, and read/write along with the optimization requirement.

The last part to the flow of messages between threads and the launching grammar is the waking up of the calling grammar. The launching grammar waits on "the wakeup" event posted by the last completed execution of the launched threads Originally there were many posted messages due to the above middlemen but this was streamlined to just wake up the grammar requesting parallelism. It then checks the critical region variable *th_accepting_cnt__* as to whether any of the launched threads were successful.

Why are there variants on "Wait for an event with or no loop"? Cuz of "pthread" implementations. It depends on how the library deals with messages for an intended thread that has not gone into the waiting stupor. Some "pthread" implementation will queue up the potential message while others just drop it. It's a question of how to sync the wait. If the "pthread" supports a future thread eventually getting to wait on the message and the called thread has already fired off the message, this pooled "to be awakened" message will be be forwarded to the thread asking to be put on hold. Your choice.

374. How to call a thread.

375. Procedure call: *start_procedure_call*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::THR_result Parser::start_procedure_call(yacco2::State &S)
{
  th_active_cnt__ = 1;
  no_requested_ths_to_run__ = 1;
  ⟨ Trace MSG start by procedure call 614 ⟩;
  THR_result rslt = (*S.proc_call_addr__)(this);
  ⟨ Trace MSG return from by procedure call 615 ⟩;
  return rslt;
}
```

376. Manually: *spawn_thread_manually*.

There is no checking on the first set of the thread. It just runs it. Allows the grammar writer to explicitly run a thread.

⟨ accrue yacco2 code 33 ⟩ +≡

```
bool yacco2::Parser::spawn_thread_manually(yacco2::USINT Thread_id)
{
  yacco2::thread_array_record *thd_stable = (yacco2::thread_array_record *) THDS_STABLE__;
  Thread_entry **thd_tbl = (Thread_entry **) &thd_stable→first_entry__;
  int no_thds = thd_stable→no_entries__ − 1;    /* rel to 0 */
  if ((Thread_id > no_thds)) {
    char a[BUFFER_SIZE];
    yacco2::KCHARP msg = "spawn_thread_manually␣thread␣id:␣%i␣out␣of␣bounds␣0␣to␣%i:␣\
        no␣thread␣available";
    sprintf(a, msg, Thread_id, no_thds);
    Yacco2_faulty_precondition(a, __FILE__, __LINE__);
    exit(1);
  }
  th_lst__.clear();
  Thread_entry *pe = thd_tbl[Thread_id];
  th_lst__.push_back(pe);
  return start_threads();
}
```

377. Start threads: *start_threads***.**
The grammar has already determined what threads to launch before calling this routine. See ⟨determine if there are threads to run 378⟩ for details. It supplies this threads thru its own private list. It searches through the global table for a thread tapping its toes to some ipod beat. If the thread is not in the table, the thread is created and passed back. If the thread is found and it's snapping its fingers for service— garçon, then it is taken, marked in the table as working, and passed back.

The last condition is the thread is found but not available to work as it already is working. This situation is nested parallelism which is equivalent to recursion used by top down parses. So, create the thread and enter it in the global table list of same thread, run it, and return.

Question. Why do you use a global mutex to protect the global thread table? As I do not know how a template runtime library controls multi-access, this is an assurance that there is no destruction or strange behaviours caused by multiple cpu systems or hyper thread systems. This might be overkill but it can be fine tuned when ported to a specific platform having standard template library thread safety. Just comment out the contents of ⟨acquire global thread table critical region 380⟩ and ⟨release global thread table critical region 381⟩.

Dance of the thread / procedure samba.
Sirens of speed are calling. The procedure call happenns when there is only 1 thread to call so its sidekick doubles for him. What happens when this sidekick is called recursively? For speed reasons, the called procedure's fsm table is static and global. Rephrased having the fsm table locally defined in the procedure takes on the ctor / use / dtor overhead. So? Well recursion becomes a destructive action on the singular fsm table. 2 or more chefs adding salt to the same pot without their knowledge of the other. Now detect whether the procedure is in use so that its thread partner does the strutting.

378. Determine if there are threads to run by current token.

```
⟨determine if there are threads to run 378⟩ ≡
  th_lst__.clear();
  find_threads_by_first_set(id_of_T, th_lst__, *S.state_s_thread_tbl__);
```

This code is cited in section 377.

This code is used in section 421.

379. Are there threads to run?. no exit with no-thds-to-run result.

```
⟨are there threads to run?. no exit with no-thds-to-run result 379⟩ ≡
  if (th_lst__.empty() ≡ YES) return Parser::no_thds_to_run;
```

This code is used in section 421.

380. Acquire global thread table critical region.

```
⟨acquire global thread table critical region 380⟩ ≡
  if (yacco2::YACCO2_MU_TH_TBL__) {
    ⟨acquire trace mu 389⟩;
    yacco2::lrclog ≪ "␣-->␣Attempting␣to␣acquire␣thread␣table␣Mutex" ≪ FILE_LINE ≪ std::endl;
    ⟨release trace mu 390⟩;
  }
  LOCK_MUTEX(yacco2::TH_TBL_MU);
  if (yacco2::YACCO2_MU_TH_TBL__) {
    ⟨acquire trace mu 389⟩;
    yacco2::lrclog ≪ "␣-->␣Acquired␣thread␣table␣Mutex" ≪ FILE_LINE ≪ std::endl;
    ⟨release trace mu 390⟩;
  }
```

This code is cited in sections 110, 178, 179, and 377.

This code is used in sections 180, 273, and 384.

381. Release global thread table critical region.

```
⟨ release global thread table critical region 381 ⟩ ≡
  if (yacco2::YACCO2_MU_TH_TBL__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog << "␣-->␣Attempting␣to␣release␣thread␣table␣Mutex" << FILE_LINE << std::endl;
    ⟨ release trace mu 390 ⟩;
  }
  UNLOCK_MUTEX(yacco2::TH_TBL_MU);
  if (yacco2::YACCO2_MU_TH_TBL__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog << "␣-->␣Released␣thread␣table␣Mutex" << FILE_LINE << std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is cited in sections 110, 178, 179, and 377.

This code is used in sections 180, 273, and 384.

382. Determine disposition of thread in global thread table.

There are 3 possibilities:

1) thread not in global table so needs to be created
2) all threads of same name busy so need to create another copy - nested situation
3) thread loitering around so put it to work

```
⟨ determine disposition of thread 382 ⟩ ≡
  int thread_disposition(0);
  Parallel_thread_list_type &i = Parallel_thread_table[pe→thd_id__];
  Parallel_thread_list_iterator_type j;
  Parallel_thread_list_iterator_type je;
  worker_thread_blk *tb;
  if (i.empty() ≡ true) {
    thread_disposition = NO_THREAD_AT_ALL;
    goto dispatch_disposition;
  }
  j = i.begin();
  je = i.end();
  for ( ; j ≠ je; ++j) {
    tb = *j;
    ⟨ Trace threads in launched list 619 ⟩;
    if (tb→status__ ≡ THREAD_WAITING_FOR_WORK) {
      thread_disposition = THREAD_WAITING_FOR_WORK;
      goto dispatch_disposition;
    }
  }
  thread_disposition = ALL_THREADS_BUSY;
  goto dispatch_disposition;
```

This code is used in section 384.

383. Dispatch on thread availability.
Note at the time of thread creation, it will fill in its operating system's "thread no" returned from `THREAD_SELF` procedure. Also the thread's *pp_requesting_parallelism__*, *from_thread__*, and *no_competing_pp_ths__* gets filled in by the canned *wpp_core.h* code. So this is why u do not see these variables set in the code parts of `NO_THREAD_AT_ALL` , and `ALL_THREADS_BUSY`.

```
⟨dispatch on thread availability: busy, available, and create one 383⟩ ≡
  switch (thread_disposition) {
  case THREAD_WAITING_FOR_WORK:
    {
      LOCK_MUTEX_OF_CALLED_PARSER(tb→grammar_s_parser__→mu__, ∗tb→grammar_s_parser__, "␣of␣self");
      tb→status__ = THREAD_WORKING;
      ++tb→run_cnt__;
      tb→grammar_s_parser__→pp_requesting_parallelism__ = this;
      tb→grammar_s_parser__→no_competing_pp_ths__ = this→no_requested_ths_to_run__;
      tb→grammar_s_parser__→from_thread__ = this;
      ⟨Trace MSG found thread in thread pool waiting to be run 611⟩;
      UNLOCK_MUTEX_OF_CALLED_PARSER(tb→grammar_s_parser__→mu__, ∗tb→grammar_s_parser__,
          "␣of␣self");
      SIGNAL_COND_VAR(∗tb→grammar_s_parser__, ∗this);
      break;
    }
  case NO_THREAD_AT_ALL:
    {
      ⟨Trace MSG thread not found in global thread pool 613⟩;
      THR_result result = CREATE_THREAD(pe→thread_fnct_ptr__, ∗this);
      break;
    }
  case ALL_THREADS_BUSY:
    {
      ⟨Trace MSG thread fnd but all busy, so launch another one 612⟩;
      yacco2::THR_result result = CREATE_THREAD(pe→thread_fnct_ptr__, ∗this);
      break;
    }
  }
```

This code is used in section 384.

384. Request threads to work.
It goes thru the thread list of the current fa's state configuration. If there is only 1 thread to be run, it calls it as a procedure rather than as a thread. The crowd is going mad... A little Fraggle Roc. I got to keep that white cane from removing me off the stage.

Why the "`VMS__`" macro variable? Don't ask, HP fumbled the pthread library implementation and the procedure call interfers with their pananoia. Blow ups on what they think is recursion to same mutex whereby a called procedure can then down the grammar call chain call itself again but the thread is launched as a thread. There is no interference on mutex recursion: each instantiation of a thread / procedure call contains its own mutex / conditional variable. Oh well enough of the core dump reguritation. Also see their stutter on the *pthread_attr_t* variable that does not default properly on stack size. It really blows its brains out even with their debugger as the firing up of the threads can't even get the registers created and so nada on the debugger scene with bad exception thrown.

```
⟨request threads to work 384⟩ ≡
  th_active_cnt__ = th_lst__.size( );
  no_requested_ths_to_run__ = th_active_cnt__;
  yacco2_threads_to_run_iter_type i = th_lst__.begin( );
  yacco2_threads_to_run_iter_type ie = th_lst__.end( );
  USINT new_r_w_cnt = supplier_r_w_cnt__ + no_requested_ths_to_run__ − 1;
  if (new_r_w_cnt > 1) {
    if (supplier_r_w_cnt__ ≡ 1) {
      if (token_supplier__ ≠ 0) {
        token_supplier__→r_w_cnt__ = new_r_w_cnt;
      }
    }
    else {
      if (token_supplier__ ≠ 0) {
        ⟨acquire token mu 391⟩;
        token_supplier__→r_w_cnt__ = new_r_w_cnt;
        ⟨release token mu 392⟩;
      }
    }
  }
  Thread_entry *pe = *i;
  ⟨acquire global thread table critical region 380⟩;
#ifndef VMS111__
  if (no_requested_ths_to_run__ > 1) goto thread_call;
procedure_call:
  {
    if (Parallel_thread_proc_call_table[pe→thd_id__].proc_call_in_use__ ≡ true) {
      ⟨Trace MSG proc call in use so call its thread 623⟩;
      goto thread_call;
    }
    Parallel_thread_proc_call_table[pe→thd_id__].proc_call_in_use__ = true;
    ⟨release global thread table critical region 381⟩;
    ⟨Trace MSG start by procedure call 614⟩;
    THR_result rslt = (*pe→proc_thread_fnct_ptr__)(this);
    ⟨acquire global thread table critical region 380⟩;
    Parallel_thread_proc_call_table[pe→thd_id__].proc_call_in_use__ = false;
    ⟨release global thread table critical region 381⟩;
    ⟨Trace MSG return from by procedure call 615⟩;
```

```
    return CALLED_AS_PROC;
  }
#endif
thread_call:
  {
    for ( ; i ≠ ie; ++i) {
      pe = *i;
      ⟨ Trace thread to be launched 620 ⟩;
      ⟨ determine disposition of thread 382 ⟩;
    dispatch_disposition:
      ⟨ dispatch on thread availability: busy, available, and create one 383 ⟩;
      ⟨ Trace TH parallel parse thread start communication 591 ⟩;
    }
  }
  ⟨ release global thread table critical region 381 ⟩;
  return CALLED_AS_THREAD;
```

This code is cited in section 742.

This code is used in section 385.

385. *start_threads*.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  bool yacco2::Parser::start_threads( )
  {
    ⟨ Trace MSG start thread 610 ⟩;
    ⟨ request threads to work 384 ⟩;
  }
```

386. Call arbitrator: *call_arbitrator***.**
No distinction made between automatically launched thread and its manual breathern. A pre-canned arbitrator *AR_for_manual_thread_spawning* is used that just returns the first item in the queue cuz there is no specialized selective code. There is a check as to more than one accept message within the queue that produces a thrown error.

Note the optimization code: If there is only 1 parallel thread within the configuration and there is no arbritration code present, then no arbitrator code for that grammar's state configuration is emitted by Yacco2. Also if only 1 T accepting then don't call the arbitrator function.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::call_arbitrator(yacco2::Type_pp_fnct_ptr The_judge)
{
  if (th_accepting_cnt__ ≡ 1) {    /* optimize no arbitration needed */
    arbitrated_token__ = &pp_accept_queue__[1];
    pp_accept_queue_idx__ = 1;
    return;
  }
  (*The_judge)(this);
}
```

387.

⟨ Optimized code call arbitrator 387 ⟩ ≡
```
if (The_judge ≡ 0) {    /* arbitrator not present in grammar */
  arbitrated_token__ = &pp_accept_queue__[1];
  pp_accept_queue_idx__ = 1;
}
if (The_judge ≠ 0) {    /* arbitrator present due to code in grammar */
  if (th_accepting_cnt__ ≡ 1) {    /* optimize no arbitration needed */
    arbitrated_token__ = &pp_accept_queue__[1];
    pp_accept_queue_idx__ = 1;
    return;
  }
  (*The_judge)(this);
  return;
}
arbitrated_token__ = &pp_accept_queue__[1];
pp_accept_queue_idx__ = 1;
```

388. Pedestrian routines for threading.

389. Acquire trace mu.
Used to serialize trace output. Sometimes the traced output is skewed due to the threading. The output to a global container is not thread safe, so make it by use of a mutex.

⟨ acquire trace mu 389 ⟩ ≡
```
LOCK_MUTEX(yacco2::TRACE_MU);
if (yacco2::YACCO2_MU_TRACING__) {
  yacco2::lrclog << "YACCO2_MU_TRACING__::Acquired␣trace␣mu" << FILE_LINE << std::endl;
}
```
This code is used in sections 79, 96, 97, 99, 101, 102, 163, 182, 183, 230, 337, 338, 365, 380, 381, 401, 402, 497, 539, 579, 580, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 626, 628, 629, 633, 634, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 648, 649, 650, 651, 652, and 653.

390. Release trace mu.

⟨ release trace mu 390 ⟩ ≡
```
if (yacco2::YACCO2_MU_TRACING__) {
  yacco2::lrclog << "YACCO2_MU_TRACING__::Releasing␣trace␣mu" << FILE_LINE << std::endl;
}
UNLOCK_MUTEX(yacco2::TRACE_MU);
```
This code is cited in section 747.

This code is used in sections 79, 96, 97, 99, 101, 102, 163, 182, 183, 230, 337, 338, 365, 380, 381, 401, 402, 497, 539, 579, 580, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 626, 628, 629, 633, 634, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 648, 649, 650, 651, 652, and 653.

391. Acquire token mu.
Used to serialize token reading.

⟨ acquire token mu 391 ⟩ ≡
```
LOCK_MUTEX(yacco2::TOKEN_MU);
```
This code is used in sections 79, 85, 90, 96, 98, 280, 320, 324, 328, 332, and 384.

392. Release token mu.

⟨ release token mu 392 ⟩ ≡
```
UNLOCK_MUTEX(yacco2::TOKEN_MU);
```
This code is used in sections 79, 85, 90, 96, 98, 280, 320, 324, 328, 332, and 384.

393. Wait for event: *wait_for_event*.

⟨ accrue yacco2 code 33 ⟩ +≡
```
void yacco2::Parser::wait_for_event()
{
  ⟨ Trace MSG thread waiting for message 601 ⟩;
#if THREAD_LIBRARY_TO_USE__ ≡ 1
  ⟨ wait for event to arrive with no loop 394 ⟩;
#else
  ⟨ wait for event to arrive with no loop 394 ⟩;
#endif
  ⟨ Trace MSG message received 602 ⟩;
}
```

394. Wait for an event to arrive with no loop.
This is a free-for-all loop, in my case only 1:1. The conditional variable and its associated data value is protected by the mutex. The calling thread has possession of the called thread's mutex. It does its thing in the critical region of the called thread by depositing the message and setting the conditional variable's data indicator to `EVENT_RECEIVED`. It releases the called thread's critical region and signals the thread library to wake up the called thread thru a conditional variable. `SIGNAL_COND_VAR` is the wrapper function to do this with the passed in variable being the selected thread to wakeup.

The wakened thread has now in its possession its critical region protecting the conditional variable and associated message indicator.

⟨ wait for event to arrive with no loop 394 ⟩ ≡
  `COND_WAIT`(*cv__*, *mu__*, ∗**this**);
  cv_cond__ = `WAIT_FOR_EVENT`;

This code is cited in sections 110 and 395.

This code is used in section 393.

395. Wait for an event to arrive with loop.
This is a free-for-all loop, in my case only 1:1. The conditional variable and its associated data value is protected by the mutex. The calling thread has possession of the called thread's mutex. It does its thing in the critical region of the called thread by depositing the message and setting the conditional variable's data indicator to `EVENT_RECEIVED`. It releases the called thread's critical region and signals the thread library to wake up the called thread thru a conditional variable. `SIGNAL_COND_VAR` is the wrapper function to do this with the passed in variable being the selected thread to wakeup.

The wakened thread has now in its possession its critical region protecting the conditional variable and associated message indicator. But to be in good keeping, I used Pthread's recommendation to protect against spurious interrupts. This is why the wait loop tests the message indicator. If it was a spurious event, it quitely goes back to sleep waiting for that prince charming to... To protect against false messages received, the condition is set right after the loop. THIS DOES NOT WORK IN HP's Alpha. That is why *wait_for_event*( ) uses ⟨ wait for event to arrive with no loop 394 ⟩ in its macro conditional.

⟨ wait for event to arrive with loop 395 ⟩ ≡
  while (*cv_cond__* ≡ `WAIT_FOR_EVENT`) {
    `COND_WAIT`(*cv__*, *mu__*, ∗**this**);
  }
  cv_cond__ = `WAIT_FOR_EVENT`;

396. *post_event_to_requesting_grammar*.
The calling thread already has the write access to the called thread's critical region. Note: All messages are synchronous in nature

1) A thread waits for an event. There is only one thread that will reply.
2) The replying thread already has the caller's mutex in its posession.

Therefore, the called grammar's mutex only needs releasing before it gets wakened by the `SIGNAL_COND_VAR` routine. It interrupts the thread runtime library with the thread's conditional variable.

⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2 :: **Parser** :: *post_event_to_requesting_grammar*
  (**yacco2** :: **Parser** &*To_thread*
  , **yacco2** :: **INT** *Message_id*
  , **yacco2** :: **Parser** &*From_thread*)
  {
    ⟨ Trace posting from - to thread info 603 ⟩;
    ⟨ deposit sender's co-ordinates and event in called thread's critical region 398 ⟩;
    ⟨ signal thread to wake up and work 397 ⟩;
  }

397. Signal thread to wake up and work.
This is the wake up event for the thread library to activate the thread from slumber.

⟨ signal thread to wake up and work 397 ⟩ ≡
⟨ Trace signaled grammar to wakeup while releasing its mutex 604 ⟩;
SIGNAL_COND_VAR(*To_thread*, ∗**this**);
⟨ Trace wakened grammar with its acquired mutex 605 ⟩;

This code is cited in section 110.

This code is used in section 396.

398. Deposit sender's co-ordinates and event in called thread's critical region.

⟨ deposit sender's co-ordinates and event in called thread's critical region 398 ⟩ ≡
To_thread.*from_thread__* = &*From_thread*;
To_thread.*msg_id__* = *Message_id*;

This code is used in section 396.

399. *have_all_threads_reported_back*.
Each thread has the responsibility to check whether it is the last thread to finish processing launched by the requesting grammar. There is no distinction on success or failure. If it is the last thread to complete, it must report back via an event to the grammar requesting parallelism. If this is not done, well you've heard of Rip Van Winkle? The requestor grammar and its dwarfs will sleep forever but not the grammar writer. Trust me, 'after you circles' of politness, or in computer terms the '5 dining philosophers' is down right hard to solve.

⟨ accrue yacco2 code 33 ⟩ +≡
bool yacco2 :: **Parser** :: *have_all_threads_reported_back*( )
{
  if (*pp_requesting_parallelism__*→*th_active_cnt__* ≡ 0) **return** YES;
  return NO;
}

400. Paranoid routines — Aborts.

401. *abort_accept_queue_irregularites*.
Provide logic clues to grammar writer. At least give the writer the grammar's state, list of threads launched, and accept tokens to figure out logic bug.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::abort_accept_queue_irregularites(yacco2::Caccept_parse &Calling_parm)
{
  ⟨ acquire trace mu 389 ⟩;
  char a[BUFFER_SIZE];
  int i = 1;
  int ie = th_accepting_cnt__;
  KCHARP grammar_having_logic_bug = "abort_accept_queue_\
      irregularites␣""-␣Overflow␣on␣accept␣queue␣Grammar␣name:␣%s␣in␣parse␣state:␣%i";
  sprintf(a, grammar_having_logic_bug, fsm_tbl__→id__, top_stack_record()→state__→state_no__);
  yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
  yacco2::lrclog ≪ "␣List␣of␣launched␣threads" ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
  KCHARP thread_in_launched_list = "␣␣␣␣-␣%s";
  yacco2_threads_to_run_iter_type ii = th_lst__.begin();
  yacco2_threads_to_run_iter_type iie = th_lst__.end();
  for ( ; ii ≠ iie; ++ii) {
    Thread_entry *pe = *ii;
    sprintf(a, thread_in_launched_list, pe→thread_fnct_name__);
    yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
  }
  yacco2::lrclog ≪ "␣List␣of␣potential␣accept␣parse␣Tes" ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
  KCHARP no_of_accept_tokens_in_queue = "␣␣␣␣no␣of␣accept␣tokens␣in␣queue:␣%i";
  sprintf(a, no_of_accept_tokens_in_queue, th_accepting_cnt__);
  yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
  KCHARP accept_queue_tokens = "␣␣␣␣-␣id:␣%s,␣token␣position:␣%i";
  for ( ; i ≤ ie; ++i) {
    sprintf(a, accept_queue_tokens, pp_accept_queue__[i].accept_token__→id__,
        pp_accept_queue__[i].accept_token_pos__);
    yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
  }
  ⟨ release trace mu 390 ⟩;
  KCHARP msg = "Overflow␣on␣Accept␣queue␣no␣of␣items:␣%i␣not␣eq␣to␣thread␣a\
      ccepting␣cnt:␣%i\n""This␣means␣more␣than␣1␣thread␣adding␣same␣accept␣token␣into\
      ␣queue?";
  sprintf(a, msg, th_accepting_cnt__ + 1, th_accepting_cnt__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

402. *abort_no_selected_accept_parse_in_arbitrator*.
Provide logic clues to grammar writer. At least give the writer the grammar's state, list of threads launched, and accept tokens to figure out logic bug.

```
⟨ accrue yacco2 code 33 ⟩ +≡
  void yacco2::Parser::abort_no_selected_accept_parse_in_arbitrator()
  {
    ⟨ acquire trace mu 389 ⟩;
    char a[BUFFER_SIZE];
    int i = 1;
    int ie = th_accepting_cnt__;
    KCHARP grammar_having_logic_bug = "abort_no_selected_accept_parse_in_arbit\
        rator␣""-␣No␣selected␣accept␣T␣Grammar␣name:␣%s␣in␣parse␣state:␣%i";
    sprintf(a, grammar_having_logic_bug, fsm_tbl__→id__, top_stack_record()→state__→state_no__);
    yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "␣List␣of␣launched␣threads" ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
    KCHARP thread_in_launched_list = "␣␣␣␣-␣%s";
    yacco2_threads_to_run_iter_type ii = th_lst__.begin();
    yacco2_threads_to_run_iter_type iie = th_lst__.end();
    for ( ; ii ≠ iie; ++ii) {
      Thread_entry *pe = *ii;
      sprintf(a, thread_in_launched_list, pe→thread_fnct_name__);
      yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
    }
    yacco2::lrclog ≪ "␣List␣of␣potential␣accept␣parse␣Tes" ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
    KCHARP no_of_accept_tokens_in_queue = "␣␣␣␣no␣of␣accept␣tokens␣in␣queue:␣%i";
    sprintf(a, no_of_accept_tokens_in_queue, th_accepting_cnt__);
    yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
    KCHARP accept_queue_tokens = "␣␣␣␣-␣id:␣%s,␣token␣position:␣%i";
    for ( ; i ≤ ie; ++i) {
      sprintf(a, accept_queue_tokens, pp_accept_queue__[i].accept_token__→id__,
          pp_accept_queue__[i].accept_token_pos__);
      yacco2::lrclog ≪ a ≪ FILE_LINE ≪ std::endl;
    }
    ⟨ release trace mu 390 ⟩;
    KCHARP msg = "No␣selected␣accept␣parse␣T␣no␣of␣items:␣%i␣\n";
    sprintf(a, msg, th_accepting_cnt__);
    Yacco2_faulty_precondition(a, __FILE__, __LINE__);
    exit(1);
  }
```

403. Lets parse do u?.

404. Common parsing code.

405. Clean up aborted parallel parse and exit erred.

⟨ clean up aborted parallel parse and exit erred 405 ⟩ ≡

```
clean_up();
return Parser::erred;
```

This code is used in sections 418, 421, and 422.

406. Exit as paralleled.
The passed back token co-ordinates are the token, position in the token stream, and the lookahead token and its position in the token stream. This is lodged in *arbitrated_token__* taken from the *accept_queue__*. The accepted token is determined by the arbitrator. Why the 2 token co-ordinates? The returned terminal is a digested statement of one or more consumed tokens in the token stream. Its token position is usually the first terminal passed for the parallel parsing: The position used the stamp the returned token can be anywhere within the position bounds of the just consummed tokens. The lookahead co-ordinates is the current token for future use. It has the same meaning as the lookahead set used by a reduce operation.

⟨ clean up and exit as paralleled 406 ⟩ ≡

```
clean_up();
return Parser::paralleled;
```

This code is used in section 422.

407. Wait for parallelism response if required.

⟨ wait for parallelism response if required 407 ⟩ ≡

```
if (how_thread_called ≡ CALLED_AS_THREAD) {
  wait_for_event();
}
```

This code is used in sections 421 and 422.

408. Extract accept parse's token Caccept_parse.
It extracts the arbitrated accept parse's token, and zeroes out its presence from the accept queue. This protects against the accept parse cleanup process deleting it as it dutifully erases all potential accept tokens in its queue.

⟨ remove accepted token reference from **Caccept_parse** and delete **Caccept_parse** 408 ⟩ ≡

```
arbitrated_token__→accept_token__ = 0;
```

This code is used in sections 418 and 421.

409. Dispatch on parallel result.

⟨ dispatch on parallel result 409 ⟩ ≡

```
if (th_accepting_cnt__ ≠ 0) goto parallelism_successful;
else goto parallelism_unsuccessful;
```

This code is used in sections 421 and 422.

410. Re-align token stream to la boundry.

⟨ re-align token stream to la boundry 410 ⟩ ≡

```
override_current_token(*arbitrated_token__→la_token__, arbitrated_token__→la_token_pos__);
```

This code is used in sections 418 and 421.

411. Re-align current token stream to accept token co-ordinates.

⟨ re-align current token stream to accept token co-ordinates 411 ⟩ ≡
override_current_token(∗*arbitrated_token__*→*accept_token__*, *arbitrated_token__*→*accept_token_pos__*);

This code is used in sections 418 and 421.

412. Allocate T id to search with.

⟨ allocate T id to search with 412 ⟩ ≡
yacco2 :: **USINT** *id_of_T* = *current_token__*→*enumerated_id__*;

This code is used in section 421.

413. Startup those threads. On your mark, get set, ...

⟨ startup those threads 413 ⟩ ≡
bool *how_thread_called* = *start_threads*( );

This code is used in section 421.

414. Clean up parallelism scribbles: *clean_up***.**

Sanitize for another round of parallel parses. Its variables are re-initialized, and potential accept messages deleted from the queue. It is rare that there is many accept messages in the queue. But when it happens, arbitration zeroed out the winner from the list leaving the balance of messages to be flushed out. The winning message is handed off to the requesting grammar to digest. *no_competing_pp_ths__* is not cleared as it's a read-only variable set by the grammar requesting parallelism.

⟨ accrue yacco2 code 33 ⟩ +≡

```
void yacco2::Parser::clean_up()
{
  if (th_accepting_cnt__ > 1) {     /* delete losers */
    for (int x = 1; x ≤ th_accepting_cnt__; ++x) {
      if (x ≡ pp_accept_queue_idx__) continue;
      if (pp_accept_queue__[x].accept_token__→auto_delete() ≡ YES) {
        delete pp_accept_queue__[x].accept_token__;
      }
      pp_accept_queue__[x].initialize_it();
    }
  }
  th_active_cnt__ = 0;
  th_accepting_cnt__ = 0;
  pp_accept_queue_idx__ = 0;
}
```

415. Chained procedure call parsing: *chained_proc_call_parsing***.**

Procedure call parsing's logic:

1) if |t| is present in the state.

This is a subrule expression that links the prefix symbol to an explicit procedure call. Its a top-down attitude to parsing with the efficiency of a procedure call. Though thread calls are neat they have their runtime inefficiences caused by their launching requirements: registers setup, address paging domains etc. Until thread calls become hardwire-support equivalent in procedure call speed this allows one to fiddle. See *pass3.lex* grammar dealing with O_2's include file expression.

416. Dispatch on proc call result.

⟨ dispatch on proc call result 416 ⟩ ≡
  if $(result \equiv th\_accepting\_cnt\_\_ \neq 0)$ **goto** *proc_call_successful*;
  else goto *proc_call_unsuccessful*;

This code is used in section 418.

417. Shift |t| onto parse stack.

⟨ shift proc call operator on to pp's parsing stack 417 ⟩ ≡
  top_stack_record( )→*set_symbol*(**NS_yacco2_k_symbols** :: *PTR_LR1_fset_transience_operator__*);
  State ∗*Goto_state* = *S.proc_call_shift__*→*goto__*;
  ⟨ *add_to_stack* 349 ⟩; /∗ ¡Trace TH the parse stack configuration¿; ∗/

This code is used in section 418.

418. *chained_proc_call_parsing*.

⟨ accrue yacco2 code 33 ⟩ +≡
  yacco2 :: **THR_result yacco2** :: **Parser** :: *chained_proc_call_parsing*(**yacco2** :: **State** &*S*)
  {
    THR_result *result* = *start_procedure_call*(*S*);
    ⟨ Trace TH request thread received message from parallel thread 594 ⟩;
    ⟨ dispatch on proc call result 416 ⟩;
  proc_call_successful:
    {
      ⟨ shift proc call operator on to pp's parsing stack 417 ⟩;
      ⟨ re-align current token stream to accept token co-ordinates 411 ⟩;
      ⟨ Trace TH accepted token info 592 ⟩;
      proc_call_shift(∗*arbitrated_token__*→*accept_token__*);
      ⟨ re-align token stream to la boundry 410 ⟩;
      ⟨ Trace TH re-aligned token stream la boundry info 593 ⟩;
      ⟨ remove accepted token reference from **Caccept_parse** and delete **Caccept_parse** 408 ⟩;
      clean_up( );
      return Parser :: *paralleled*;
    }
  proc_call_unsuccessful:
    ⟨ clean up aborted parallel parse and exit erred 405 ⟩;
  }

419. Start parallel parsing: *start_parallel_parsing*.

start parallel parsing's logic:

1) determine by first set evalution if there are threads. exit if none.
2) parser spawns the parallel parser threads and waits for results
3) dispatching of the Arbitrator. Arbitration is local per state

420. Shift (|||) onto parse stack.

⟨ shift parallel operator on to pp's parsing stack 420 ⟩ ≡
  top_stack_record( )→*set_symbol*(**NS_yacco2_k_symbols** :: *PTR_LR1_parallel_operator__*);
  Goto_state = *S.parallel_shift__*→*goto__*;
  ⟨ *add_to_stack* 349 ⟩; /∗ ¡Trace TH the parse stack configuration¿; ∗/

This code is used in section 421.

421. *start_parallel_parsing*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Parser::parse_result yacco2::Parser::start_parallel_parsing(yacco2::State &S)
{
  yacco2::State *Goto_state;
  ⟨ allocate T id to search with 412 ⟩;
  ⟨ determine if there are threads to run 378 ⟩;
  ⟨ are there threads to run?. no exit with no-thds-to-run result 379 ⟩;
  ⟨ startup those threads 413 ⟩;
wait_for_response:
  ⟨ wait for parallelism response if required 407 ⟩;
  ⟨ Trace TH request thread received message from parallel thread 594 ⟩;
  ⟨ dispatch on parallel result 409 ⟩;
parallelism_successful:
  ⟨ shift parallel operator on to pp's parsing stack 420 ⟩;
  if (S.state_s_thread_tbl__→ar_fnct_ptr__ ≡ 0) {
    arbitrated_token__ = &pp_accept_queue__[1];
    pp_accept_queue_idx__ = 1;
  }
  else {
    call_arbitrator(S.state_s_thread_tbl__→ar_fnct_ptr__);
  }   /* Validate accept message; */
  ⟨ re-align current token stream to accept token co-ordinates 411 ⟩;
  ⟨ Trace TH accepted token info 592 ⟩;
  parallel_shift(*arbitrated_token__→accept_token__);
  ⟨ re-align token stream to la boundry 410 ⟩;
  ⟨ Trace TH re-aligned token stream la boundry info 593 ⟩;
  ⟨ remove accepted token reference from Caccept_parse and delete Caccept_parse 408 ⟩;
  clean_up();
  return Parser::paralleled;
parallelism_unsuccessful:
  ⟨ clean up aborted parallel parse and exit erred 405 ⟩;
}
```

422. *start_manually_parallel_parsing*.
This facility allows one to do parallel parsing from syntax directed code within a grammar. For example, one might test a returned terminal whose lookahead expressions need parsing. This is a context sensitive way to process text dynamically. The Yacco2 compiler uses this approach to process its directives' syntax directed code. Here is a code sample using it.

```
/*
 file: /yacco2/diagrams+etc/threadmanualcall.txt
 Example of a subrule calling a thread manually.
 Taken from grammar pass3.lex before explicit procedure call of threads
 construct invented.
 The start_manually_parallel_parsing function uses the
 thread's id generated from O2linker as its key to find the thread to launch.
*/
  -> "@"
  /@
  \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.
 @/
  {
    op
      using namespace NS_prefile_include;
      using namespace NS_yacco2_T_enum;

      Parser::parse_result result =
        rule_info__.parser__->
        start_manually_parallel_parsing(ITH_prefile_include.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_file_inclusion_) {
            T_file_inclusion* finc = (T_file_inclusion*)(rtn_tok);
            CAbs_lr1_sym* err = finc->error_sym();
            if(err != 0) {
              rule_info__.parser__->set_abort_parse(true);
              ADD_TOKEN_TO_ERROR_QUEUE(*finc);
              ADD_TOKEN_TO_ERROR_QUEUE(*finc->error_sym());
              finc->error_sym(0);
              return;
            }
            rule_info__.parser__->
              override_current_token(*accept_parm.la_token__
                                     ,accept_parm.la_token_pos__);
             bool result =
              PROCESS_INCLUDE_FILE
                  (*rule_info__.parser__
                  ,*finc,*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(*finc);//file name inside
            return;
        }
        // catch all  errors
        rule_info__.parser__->set_abort_parse(true);
      ***
      }
```

⟨ accrue yacco2 code 33 ⟩ +≡

Parser :: **parse_result** **yacco2** :: **Parser** :: *start_manually_parallel_parsing*
(**yacco2** :: **USINT** *Thread_id*)
{
  bool *how_thread_called* = *spawn_thread_manually*(*Thread_id*);
  ⟨ wait for parallelism response if required 407 ⟩;
  ⟨ Trace TH request thread received message from parallel thread 594 ⟩;
  ⟨ dispatch on parallel result 409 ⟩;
parallelism_successful:
  {
    if (**yacco2** :: *PTR_AR_for_manual_thread_spawning* ≡ 0) {
      arbitrated_token__ = &*pp_accept_queue__*[1];
      pp_accept_queue_idx__ = 1;
    }
    else {
      call_arbitrator(**yacco2** :: *PTR_AR_for_manual_thread_spawning*);
    } /* Validate accept message; */
    ⟨ Trace TH accepted token info 592 ⟩;
    ⟨ clean up and exit as paralleled 406 ⟩;
  }
parallelism_unsuccessful:

```
    ⟨ clean up aborted parallel parse and exit erred 405 ⟩;
  }
```

423. Yacco2 global variables.
A hodge-podge of entities and procedures supporting tracing, files processed with recursion support, threading tables and their first sets, low-level character mapping, and low level mutual exclusion controlling access to threads, tracing, and symbol table management.

Access control: Bouncer / doorman.
By their name `TRACE_MU`, `MUTEXTH_TBL_MU`, and `MUTEXSYM_TBL_MU` are mutexes for crowd control for tracing, thread table management, and symbol table access. `THDS_STABLE__` and `THDS_FSET_BY_T__` are data structures generated by Yacco2's Linker that get resolved to the specific use of this library. They are dangling references.

File management:
`FILE_TBL__` is a dictionary of file names that have been opened during the compile process. It's key is the file number component to the symbol's GPS in the source file. `FILE_CNT__` is the current file number being processed. It starts from 0 due to C++'s vector requirement used by `FILE_TBL__`. The **tok_can** template containers use these variables: ie, raw character symbol processing. `STK_FILE_NOS__` is a stack of nested `FILE_CNT__` file numbers used to re-establish processing of the file following its include statement.

⟨ Type defs 16 ⟩ +≡
  typedef std :: **vector**⟨**std** :: *string*⟩ **gbl_file_map_type**;

424. Global variables.

⟨ Global variables 21 ⟩ +≡
  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 :: `MUTEXTRACE_MU`;
  extern yacco2 :: `MUTEXTH_TBL_MU`;
  extern yacco2 :: `MUTEXSYM_TBL_MU`;
  extern yacco2 :: `MUTEXTOKEN_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__`;

425. `LRK_LA_EOLR_SET`.
Used by *find_reduce_entry* for meta termials lookahed set. Meta-terminals are 8 elements that start off the enumeration scheme. Therefore they all fit within one partition. `|?|`, eog, and `|||` are left out of the lookahead set leaving eolr, `|.|`, `|+|`,`|t|`, and `|r|`.

⟨ Global variables 21 ⟩ +≡
  extern yacco2 :: **Set_entry** `LRK_LA_EOLR_SET`;

426. `LRK_LA_QUE_SET` for error enforcement.
Used by *find_questionable_sym_in_reduce_lookahead* for forced reduce to handle error detection. It forces the reducing subrule to reduce cuz the `|?|` symbol in in its follow set. That is the shifted into parse state for the recuded rule contains the `|?|` symbol used for error catching.

⟨ Global variables 21 ⟩ +≡
  extern yacco2 :: **Set_entry** `LRK_LA_QUE_SET`;

427. Global routines.

⟨ External rtns and variables 22 ⟩ +≡

```
extern void Delete_tokens(yacco2::TOKEN_GAGGLE &Tks, bool Do_delete = OFF);
extern void Clear_yacco2_opened_files_dictionary( );
```

428. Global variables implementations.

⟨ accrue yacco2 code 33 ⟩ +≡

```
std::list<std::string> yacco2::O2_LOGICALS__;
yacco2::gbl_file_map_type yacco2::FILE_TBL__;
std::vector<yacco2::UINT> yacco2::STK_FILE_NOS__;
yacco2::UINT yacco2::FILE_CNT__(0);
yacco2::rc_map yacco2::RC__;
yacco2::Type_pp_fnct_ptr yacco2::PTR_AR_for_manual_thread_spawning(0);
  /* split lines: cuz Apple's latest compiler bug */
  /* No matching literal operator for call to 'operator"' date macro */   /* with arguments of types
    'const char*' and 'unsigned long', and no matching literal operator template */   /* */
yacco2::KCHARP yacco2::Lr1_VERSION = "O2 version: 1.0 Distribution Date: "
__DATE__"\n";
yacco2::KCHARP yacco2::O2linker_VERSION = "O2linker version: 1.0 Distribution Date: "
__DATE__"\n";
yacco2::MUTEX yacco2::TOKEN_MU;
yacco2::MUTEX yacco2::TRACE_MU;
yacco2::MUTEX yacco2::TH_TBL_MU;
yacco2::MUTEX yacco2::SYM_TBL_MU;
std::ofstream yacco2::lrclog("1lrtracings.log");
std::ofstream yacco2::lrerrors("1lrerrors.log");
yacco2::Set_entry yacco2::LRK_LA_EOLR_SET = {0, #f4};   /* eolr, |r|, |.|, |+|, and |t| */
yacco2::Set_entry yacco2::LRK_LA_QUE_SET = {0, #01};   /* elem 1 is |?| so 2 ⊕ 0 */
```

429. Runtime errors.

It supplies all the error objects that get thrown within yacco2's environment. Presently, my design is crude: no design but a list of error events.

⟨ Structure defs 18 ⟩ +≡

```
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__);
};
```

430. Runtime error messages implementations.

⟨ accrue yacco2 code 33 ⟩ +≡

```
yacco2::Source_info::
Source_info(yacco2::KCHARP File, yacco2::UINT Line)
: file__(File), line__(Line) {
  w_info();
}
void yacco2::Source_info::
w_info()
{
  yacco2::lrclog << " Version: " << yacco2::Lr1_VERSION << " thrown from source file: " <<
      file__ << " line: " << line__ << __FILE__ << __LINE__ << std::endl;
  std::cout << " Version: " << yacco2::Lr1_VERSION << " thrown from source file: " <<
      file__ << " line: " << line__ << __FILE__ << __LINE__ << std::endl;
}
yacco2::Yacco2_faulty_precondition::
Yacco2_faulty_precondition(yacco2::KCHARP Message, yacco2::KCHARP File, yacco2::UINT
    Line)
: Source_info(File, Line) {
  yacco2::lrclog << "Yacco2_faulty_precondition: " << Message << ' ' << __FILE__ << ": " <<
      __LINE__ << std::endl;
  std::cout << "Yacco2_faulty_precondition: " << Message << std::endl;
}
yacco2::Yacco2_faulty_postcondition::
Yacco2_faulty_postcondition(yacco2::KCHARP Message, yacco2::KCHARP
    File, yacco2::UINT Line)
: Source_info(File, Line) {
  yacco2::lrclog << "Yacco2_faulty_postcondition: " << Message << ' ' << __FILE__ << ": " <<
      __LINE__ << std::endl;
  std::cout << "Yacco2_faulty_postcondition: " << Message << std::endl;
}
```

431. Global garbage sweeper.
Simple container whose contents are deleted. As one parses, somewhere the newly minted tokens needed to be deleted. The container maintains a one-to-one reference to the symbol which gets deleted by this routine. Due to the “lr k symbols” being globally defined in global space rather than being their creation by the new operator, there is protective code to prevent their deletion.

Depending on how the “Raw Characters” are built, they could also be bypassed. For now, the “global garbage” sweeper is not very good: a map template just is down right slow. So I must revisit my thought and come up with a better data structure to use.

I bypass this routine as the cost of building the thread index is TOOOOOooo slow and occassionally buggy from the template implementation.

⟨ Type defs 16 ⟩ +≡

```
typedef std::set<yacco2::CAbs_lr1_sym *> set_of_objs_type;
typedef set_of_objs_type::iterator set_of_objs_iter_type;
```

432. *Delete_tokens*.

⟨ accrue yacco2 code 33 ⟩ +≡

```
extern void yacco2::Delete_tokens(yacco2::TOKEN_GAGGLE &Tks, bool Do_delete)
{
  return;
  using namespace NS_yacco2_k_symbols;
  static yacco2::set_of_objs_type deleted_syms;
  static yacco2::set_of_objs_type dont_delete_syms;
  static bool onetime(OFF);
  if (onetime ≡ OFF) {
    onetime = ON;
    dont_delete_syms.insert(PTR_LR1_eolr__);
    dont_delete_syms.insert(PTR_LR1_questionable_shift_operator__);
    dont_delete_syms.insert(PTR_LR1_eog__);
    dont_delete_syms.insert(PTR_LR1_parallel_operator__);
    dont_delete_syms.insert(PTR_LR1_invisible_shift_operator__);
    dont_delete_syms.insert(PTR_LR1_all_shift_operator__);
    dont_delete_syms.insert(PTR_LR1_fset_transience_operator__);
  }
  if (Do_delete ≡ ON) {
    set_of_objs_iter_type k = deleted_syms.begin();
    set_of_objs_iter_type ke = deleted_syms.end();
    for ( ; k ≠ ke; ++k) {
      CAbs_lr1_sym *sym = *k;    /* /delete sym; */
    }
    return;
  }
  TOKEN_GAGGLE_ITER i = Tks.begin();
  TOKEN_GAGGLE_ITER ie = Tks.end();
  for ( ; i ≠ ie; ++i) {
    yacco2::CAbs_lr1_sym *sym = *i;
    yacco2::set_of_objs_iter_type j;
    j = deleted_syms.find(sym);
    if (j ≠ deleted_syms.end()) continue;    /* already deleted */
    j = dont_delete_syms.find(sym);
    if (j ≠ dont_delete_syms.end()) continue;
    deleted_syms.insert(sym);
  }
}
```

433. *Clear_yacco2_opened_files_dictionary*.

Allows one to have multiple parse sessions. This clears the previous parse attempt. Give me an example of why u need this? Consider a XML language recognizer that is continuously being called to process a say Soap session. Each session is a new parsing bout.

⟨ accrue yacco2 code 33 ⟩ +≡

```
extern void yacco2::Clear_yacco2_opened_files_dictionary()
{
  yacco2::FILE_TBL__.clear();
  yacco2::STK_FILE_NOS__.clear();
  yacco2::FILE_CNT__ = 0;
}
```

434. 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 behavior 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 behavior 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. Ugh. 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 exclussion 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!

435. Tree walker's traversal with filter mechanism.

⟨ tree walker's traversal with filter mechanism 435 ⟩ ≡

```
  advance( );   /* status advance */
  int_set_iter_type i;
  CAbs_lr1_sym *sym;
tree_traverse:
  {
    if (base_stk_.cur_stk_rec_ ≡ 0) return;
    if (base_stk_.cur_stk_rec_→act_ ≠ ast_base_stack::visit) {
      ⟨ Go to next t 437 ⟩;
    }
    sym = AST::content(*base_stk_.cur_stk_rec_→node_);
    if (base_stk_.filter_ ≡ 0) ⟨ Go to accept t 438 ⟩;
  filter_node:
    ⟨ see if just read node's content is in filter set 442 ⟩;
    ⟨ dispatch on filter type: accept or reject filter 436 ⟩;
  reject_filter:
    ⟨ is node's content found in bypass filter? yes next t, no accept t 441 ⟩;
  accept_filter:
    ⟨ is node's content in accept filter? no next t, yes accept t 440 ⟩;
  next_t:
    advance( );   /* go fetch next node as current */
    goto tree_traverse;
  }
accept_t: ⟨ fire off visit functor 439 ⟩;
  return;
```

This code is used in sections 455, 458, 461, 464, 467, and 470.

436. Dispatch on filter type: accept or reject filter.

⟨ dispatch on filter type: accept or reject filter 436 ⟩ ≡

```
  if (base_stk_.accept_opt_ ≡ true) goto accept_filter;
  else goto reject_filter;
```

This code is used in section 435.

437. Go to next t.

⟨ Go to next t 437 ⟩ ≡

```
  goto next_t;
```

This code is used in sections 435, 440, and 441.

438. Go to accept t.

⟨ Go to accept t 438 ⟩ ≡

```
  goto accept_t;
```

This code is used in sections 435, 440, and 441.

439. Fire off visit functor.

⟨ fire off visit functor 439 ⟩ ≡
```
yacco2::functor_result_type rr = base_stk_.action_→operator()(&base_stk_);
switch (rr) {
case yacco2::bypass_node: goto next_t;
case yacco2::accept_node: return;
case yacco2::stop_walking:
  {
    base_stk_.cur_stk_rec_ = 0;
    return;
  }
}
```
This code is used in section 435.

440. Is node's content found in accept filter? no next t, yes accept t.

⟨ is node's content in accept filter? no next t, yes accept t 440 ⟩ ≡
```
if (i ≡ base_stk_.filter_→end()) ⟨ Go to next t 437 ⟩;
⟨ Go to accept t 438 ⟩;
```
This code is used in section 435.

441. Is node's content found in bypass filter? yes next t, no accept t.

⟨ is node's content found in bypass filter? yes next t, no accept t 441 ⟩ ≡
```
if (i ≠ base_stk_.filter_→end()) ⟨ Go to next t 437 ⟩;
⟨ Go to accept t 438 ⟩;
```
This code is used in section 435.

442. See if just read node's content is in filter set.

⟨ see if just read node's content is in filter set 442 ⟩ ≡
```
i = base_stk_.filter_→find(sym→enumerated_id__);
```
This code is used in section 435.

443. *ast_postfix* **tree walker.**

⟨ Structure defs 18 ⟩ +≡
```
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();
};
```

444. **Prefix tree walker.**

⟨ Structure defs 18 ⟩ +≡
```
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();
};
```

445. Postfix tree walker of self only.
The forest in its name indicates that it is considered a stand alone tree. It will not follow it's brother links.

⟨Structure defs 18⟩ +≡
```
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();
};
```

446. Prefix tree walker of a forest.
This only walks itself and its underlings. It does not follow its brother link.

⟨Structure defs 18⟩ +≡
```
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();
};
```

447. Breadth only tree walker.
Deal with self and younger siblings.

⟨Structure defs 18⟩ +≡
```
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();
};
```

448. Prefix with breadth only tree walker.
Parental walk with immediate children.

⟨Structure defs 18⟩ +≡
```
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();
};
```

449. Moon walking — get ancestry for a specific node.
This walk goes up a tree looking for its ancestral goal node. It fills the list in youngest to oldest order where the last node being the goal node. The goal node allows u to stop partway thru the global tree: ie somewhere within a context. If no filter set is passed it defaults to all Tes accepted. The resultant list of ancestral nodes can be empty.

If a functor is provided, it allow one to fine-tune the acceptance of an ancester or to recurse on its own tree walking: no inter-family feuds allowed?!

⟨Structure defs 18⟩ +≡

```
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_;
};
```

450. Tree implementations.

⟨wtree.cpp 450⟩ ≡
  ⟨copyright notice 565⟩;
  ⟨iyacco2 26⟩;
  ⟨accrue tree code 451⟩;

451. Accrue tree code.

⟨accrue tree code 451⟩ ≡ /* accrue tree code */

See also sections 452, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 501, 502, 503, 504, 505, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 525, 534, 535, 536, 537, 538, 539, 540, and 541.

This code is used in section 450.

452. ast_base_stack implementation.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::ast_base_stack::ast_base_stack(Type_AST_functor *Action, yacco2::int_set_type
        *Filter, bool Accept_opt)
  : idx_(No_Token_start_pos), stk_(std::vector<s_rec>()), action_(Action), cur_stk_rec_(0), filter_(Filter),
    accept_opt_(Accept_opt) { }
yacco2::ast_base_stack::ast_base_stack()
  : idx_(No_Token_start_pos), stk_(std::vector<s_rec>()), action_(0), cur_stk_rec_(0), filter_(0),
    accept_opt_(YES) { }
yacco2::ast_stack::ast_stack(Type_AST_functor *Action, yacco2::int_set_type *Filter, bool
        Accept_opt)
  : base_stk_(Action, Filter, Accept_opt) { }
void yacco2::ast_base_stack::pop()
{
  if (stk_.empty() ≡ YES) return;
  --idx_;
  stk_.pop_back();
  if (stk_.empty() ≡ YES) {
    idx_ = No_Token_start_pos;
    cur_stk_rec_ = 0;
    return;
  }
  cur_stk_rec_ = &stk_[idx_];
}
void yacco2::ast_base_stack::push(AST &Node, ast_base_stack::n_action Action)
{
  ++idx_;
  stk_.push_back(yacco2::ast_base_stack::s_rec());
  cur_stk_rec_ = &stk_[idx_];
  cur_stk_rec_→node_ = &Node;
  cur_stk_rec_→act_ = Action;
}
yacco2::INT yacco2::ast_base_stack::cur_stk_index()
{
  return idx_;
}
yacco2::ast_base_stack::s_rec *yacco2::ast_base_stack::cur_stk_rec()
{
  return cur_stk_rec_;
}
yacco2::ast_base_stack::s_rec *yacco2::ast_base_stack::stk_rec(yacco2::INT I)
{
  if (I > idx_) return 0;
  return &stk_[I];
}
```

453. Tree walker implementations.

454. ast_postfix.
This is your regular postfix tree walker of a complete tree.

⟨ accrue tree code 451 ⟩ +≡
yacco2 :: **ast_postfix** :: **ast_postfix**(**AST** &*Forest*, **Type_AST_functor** ∗*Action*, **yacco2** :: **int_set_type** ∗*Filter*, **bool** *Accept_opt*)
: **yacco2** :: **ast_stack**(*Action*, *Filter*, *Accept_opt*) {
base_stk_.push(*Forest*, **ast_base_stack** :: *init*);
}

455. ast_postfix exec.
Originally this was a switch statement handling the 5 states. As this is a 80/20 situation, the if statement is more efficient: no need for the specifics.

⟨ accrue tree code 451 ⟩ +≡
void yacco2 :: **ast_postfix** :: *exec*( )
{
⟨ tree walker's traversal with filter mechanism 435 ⟩;
}

456. ast_postfix advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_postfix::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      AST *down = AST::get_1st_son(*base_stk_.cur_stk_rec_→node_);
      if (down ≡ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;   /* bypass left */
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::left;
      base_stk_.push(*down, ast_base_stack::init);
      return;
    }
  case ast_base_stack::left:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::visit:
    {
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≡ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;   /* bypass */
        return;
      }
      base_stk_.pop();
      base_stk_.push(*rt, ast_base_stack::init);
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::eoc:
    {
      base_stk_.pop();
      return;
    }
  }
}
```

457. ast_prefix.
Prefix walk of complete tree.

⟨ accrue tree code 451 ⟩ +≡
yacco2 :: **ast_prefix** :: **ast_prefix**(**AST** &*Forest*, **Type_AST_functor** ∗*Action*, **yacco2** :: **int_set_type** ∗*Filter*, **bool** *Accept_opt*)
: **yacco2** :: **ast_stack**(*Action*, *Filter*, *Accept_opt*) {
base_stk_.push(*Forest*, **ast_base_stack** :: *init*);
}

458. ast_prefix exec.

⟨ accrue tree code 451 ⟩ +≡
void yacco2 :: **ast_prefix** :: *exec*( )
{
⟨ tree walker's traversal with filter mechanism 435 ⟩;
}

459. ast_prefix advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_prefix::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::left:
    {
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::visit:
    {
      AST *lt = AST::get_1st_son(*base_stk_.cur_stk_rec_→node_);
      if (lt ≠ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::left;
        base_stk_.push(*lt, ast_base_stack::init);
        return;
      }
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::eoc:
    {
      base_stk_.pop();
      return;
    }
  }
}
```

460. ast_postfix_1forest.
Forest postfix walk. Do not go outside its bounds.

⟨ accrue tree code 451 ⟩ +≡
yacco2 :: **ast_postfix_1forest** :: **ast_postfix_1forest**(**AST** &*Forest*, **Type_AST_functor** ∗*Action*, **yacco2** :: **int_set_type** ∗*Filter*, **bool** *Accept_opt*)
: **yacco2** :: **ast_stack**(*Action*, *Filter*, *Accept_opt*) {
base_stk_.push(*Forest*, **ast_base_stack** :: *init*);
}

461. ast_postfix_1forest exec.

⟨ accrue tree code 451 ⟩ +≡
void yacco2 :: **ast_postfix_1forest** :: *exec*( )
{
⟨ tree walker's traversal with filter mechanism 435 ⟩;
}

462. ast_postfix_1forest advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_postfix_1forest::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      AST *down = AST::get_1st_son(*base_stk_.cur_stk_rec_→node_);
      if (down ≡ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;    /* bypass left */
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::left;
      base_stk_.push(*down, ast_base_stack::init);
      return;
    }
  case ast_base_stack::left:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::visit:
    {
      AST *rt(0);
      if (base_stk_.idx_ ≠ 0)    /* only traverse the forest */
        rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≡ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;    /* bypass */
        return;
      }
      base_stk_.pop();
      base_stk_.push(*rt, ast_base_stack::init);
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::eoc:
    {
      base_stk_.pop();
      return;
    }
  }
}
```

463. ast_prefix_1forest.
Forest prefix walk. Do not go outside its bounds.

⟨ accrue tree code 451 ⟩ +≡
yacco2 :: **ast_prefix_1forest** :: **ast_prefix_1forest**(**AST** &*Forest*, **Type_AST_functor** **Action*, **yacco2** :: **int_set_type** **Filter*, **bool** *Accept_opt*)
: **yacco2** :: **ast_stack**(*Action*, *Filter*, *Accept_opt*) {
base_stk_.push(*Forest*, **ast_base_stack** :: *init*);
}

464. ast_prefix_1forest exec.

⟨ accrue tree code 451 ⟩ +≡
void yacco2 :: **ast_prefix_1forest** :: *exec*( )
{
⟨ tree walker's traversal with filter mechanism 435 ⟩;
}

465. ast_prefix_1forest advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_prefix_1forest::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::left:
    {
      AST *rt(0);
      if (base_stk_.idx_ ≠ 0)    /* only traverse the forest */
        rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::visit:
    {
      AST *lt = AST::get_1st_son(*base_stk_.cur_stk_rec_→node_);
      if (lt ≠ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::left;
        base_stk_.push(*lt, ast_base_stack::init);
        return;
      }
      AST *rt(0);
      if (base_stk_.idx_ ≠ 0)    /* only traverse the forest */
        rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::eoc:
    {
      base_stk_.pop();
```

```
      return;
    }
  }
}
```

466. ast_breadth_only.
Walk self and its younger brothers.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::ast_breadth_only::ast_breadth_only(AST &Forest, Type_AST_functor
        *Action, yacco2::int_set_type *Filter, bool Accept_opt)
: yacco2::ast_stack(Action, Filter, Accept_opt) {
  base_stk_.push(Forest, ast_base_stack::init);
}
```

467. ast_breadth_only exec.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_breadth_only::exec( )
{
  ⟨ tree walker's traversal with filter mechanism 435 ⟩;
}
```

468. ast_breadth_only advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_breadth_only::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::left:
    {
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::visit:
    {
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::eoc:
    {
      base_stk_.pop();
      return;
    }
  }
}
```

469. ast_prefix_wbreadth_only.
Walk self who is a parent and its immediate children.

⟨ accrue tree code 451 ⟩ +≡
```
yacco2::ast_prefix_wbreadth_only::ast_prefix_wbreadth_only(AST &Forest,
        Type_AST_functor *Action, yacco2::int_set_type *Filter, bool Accept_opt)
: yacco2::ast_stack(Action, Filter, Accept_opt) {
   base_stk_.push(Forest, ast_base_stack::init);
}
```

470. ast_prefix_wbreadth_only exec.

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::ast_prefix_wbreadth_only::exec()
{
   ⟨ tree walker's traversal with filter mechanism 435 ⟩;
}
```

471. ast_prefix_wbreadth_only advance.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::ast_prefix_wbreadth_only::advance()
{
  if (base_stk_.cur_stk_rec_ ≡ 0) return;
  switch (base_stk_.cur_stk_rec_→act_) {
  case ast_base_stack::init:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::visit;
      return;
    }
  case ast_base_stack::left:
    {
      if (base_stk_.idx_ ≡ 0) {
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
        return;
      }
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::visit:
    {
      if (base_stk_.idx_ ≡ 0) {
        AST *lt = AST::get_1st_son(*base_stk_.cur_stk_rec_→node_);
        if (lt ≡ 0) {
          base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
          return;
        }
        base_stk_.cur_stk_rec_→act_ = ast_base_stack::left;
        base_stk_.push(*lt, ast_base_stack::init);
        return;
      }
      AST *rt = AST::brother(*base_stk_.cur_stk_rec_→node_);
      if (rt ≠ 0) {
        base_stk_.pop();
        base_stk_.push(*rt, ast_base_stack::init);
        return;
      }
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
    }
  case ast_base_stack::right:
    {
      base_stk_.cur_stk_rec_→act_ = ast_base_stack::eoc;
      return;
```

```
      }
    case ast_base_stack::eoc:
      {
        base_stk_.pop();
        return;
      }
    }
  }
```

472. ast_moonwalk_looking_for_ancestors.

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::ast_moonwalk_looking_for_ancestors::ast_moonwalk_looking_for_ancestors(AST
        &Moonchild, USINT Goal, Type_AST_ancestor_list & Ancestors_list, Type_AST_functor
        *Functor, yacco2::int_set_type *Filter, bool Accept_opt)
  : moonchild_(&Moonchild), goal_(Goal), ancestor_list_(&Ancestors_list), functor_(Functor),
      filter_(Filter), filter_type_(Accept_opt), filter_provided_(NO) {
    if (Filter ≠ 0) filter_provided_ = YES;
  }
```

473. *let_s_functor*.

It's returned value indicates either stop the tree walk, or continue the walk and what to do with the visited node — accept it or bypass.

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::functor_result_type yacco2::ast_moonwalk_looking_for_ancestors::let_s_functor(AST
        *Parent)
  {
    functor_result_type functor_result;
    yacco2::ast_base_stack abs;
    abs.push(*Parent, ast_base_stack::init);
    return functor_→operator()(&abs);
  }
```

474. *deal_with_functor*.
If the Parent passes the grade it's added to the ancestry list. Returning a "NO" indicates to terminate the tree walking while a "YES" is keep-it-going thriller.

⟨ accrue tree code 451 ⟩ +≡
```
bool yacco2::ast_moonwalk_looking_for_ancestors::deal_with_functor(AST *Parent)
{
  if (functor_ ≠ 0) {
    functor_result_type functor_result = let_s_functor(Parent);
    switch (functor_result) {
    case accept_node:
      {
        ancestor_list_→push_back(Parent);
        return YES;
      }
    case bypass_node:
      {
        return YES;
      }
    case stop_walking:
      {
        return NO;
      }
    }
  }
  else {
    ancestor_list_→push_back(Parent);
    return YES;
  }
  return YES;
}
```

475. *let_s_moonwalk*.
Do those backward moves on the tree like MJ.

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::ast_moonwalk_looking_for_ancestors::let_s_moonwalk()
{
  functor_result_type functor_result;
  AST *cnode = moonchild_;
  AST *parent(0);
  while (cnode ≠ 0) {
    parent = AST::get_parent(*cnode);
    bool continue_waldo = deal_with_parent(parent);
    if (continue_waldo ≡ NO) return;
    cnode = parent;
  }
}
```

476. *deal_with_parent*.
Returning a "NO" indicates to terminate the tree walking.

⟨ accrue tree code 451 ⟩ +≡

```
bool yacco2::ast_moonwalk_looking_for_ancestors::deal_with_parent(AST *Parent)
{
  if (Parent ≡ 0) {    /* orphan? */
    return NO;
  }
  CAbs_lr1_sym *tsym = AST::content(*Parent);
  USINT id = tsym→enumerated_id();
  if (id ≡ goal_) {
    ancestor_list_→push_back(Parent);
    return NO;    /* finished going thru tree as goal found */
  }
  ⟨ Dispatch on use-of-filter 477 ⟩;
no_filter_so_accept_all_Tes:
  {
    return deal_with_functor(Parent);
  }
filtered_Tes:
  {
    int_set_iter_type i = filter_→find(id);
    if (i ≡ filter_→end()) {
      if (filter_type_ ≡ ACCEPT_FILTER) return YES;
      return deal_with_functor(Parent);
    }    /* found T in filter */
    if (filter_type_ ≡ BYPASS_FILTER) return YES;
    return deal_with_functor(Parent);
  }
}
```

477. Dispatch on use-of-filter.

⟨ Dispatch on use-of-filter 477 ⟩ ≡

```
if (filter_provided_ ≡ NO) goto no_filter_so_accept_all_Tes;
else goto filtered_Tes;
```

This code is used in section 476.

478. Build and restructure trees.

479. *restructure_2trees_into_1tree*. .

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::restructure_2trees_into_1tree(AST &S1, AST &S2)
{
  AST *s2lt = AST::get_1st_son(S2);
  AST::zero_1st_son(S2);
  AST::crt_tree_of_2sons(S2, S1, *s2lt);
  return &S2;
}
```

480. Create trees *crt_tree_of_1son*—*crt_tree_of_9sons*.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::crt_tree_of_1son(yacco2::AST &Parent, yacco2::AST &S1)
{
  yacco2::AST::join_pts(Parent, S1);
}
void yacco2::AST::crt_tree_of_2sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
        &S2)
{
  yacco2::AST::join_pts(Parent, S1);
  yacco2::AST::join_sts(S1, S2);
}
void yacco2::AST::crt_tree_of_3sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
        &S2, yacco2::AST &S3)
{
  yacco2::AST::join_pts(Parent, S1);
  yacco2::AST::join_sts(S1, S2);
  yacco2::AST::join_sts(S2, S3);
}
void yacco2::AST::crt_tree_of_4sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
        &S2, yacco2::AST &S3, yacco2::AST &S4)
{
  yacco2::AST::join_pts(Parent, S1);
  yacco2::AST::join_sts(S1, S2);
  yacco2::AST::join_sts(S2, S3);
  yacco2::AST::join_sts(S3, S4);
}
void yacco2::AST::crt_tree_of_5sons(yacco2::AST &Parent, AST &S1, yacco2::AST
        &S2, yacco2::AST &S3, yacco2::AST &S4, yacco2::AST &S5)
{
  yacco2::AST::join_pts(Parent, S1);
  yacco2::AST::join_sts(S1, S2);
  yacco2::AST::join_sts(S2, S3);
  yacco2::AST::join_sts(S3, S4);
  yacco2::AST::join_sts(S4, S5);
}
void yacco2::AST::crt_tree_of_6sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
        &S2, yacco2::AST &S3, yacco2::AST &S4, yacco2::AST &S5, yacco2::AST &S6)
{
  yacco2::AST::join_pts(Parent, S1);
  yacco2::AST::join_sts(S1, S2);
  yacco2::AST::join_sts(S2, S3);
  yacco2::AST::join_sts(S3, S4);
  yacco2::AST::join_sts(S4, S5);
  yacco2::AST::join_sts(S5, S6);
}
void yacco2::AST::crt_tree_of_7sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
        &S2, yacco2::AST &S3, yacco2::AST &S4, yacco2::AST &S5, yacco2::AST
        &S6, yacco2::AST &S7)
{
```

```
    yacco2::AST::join_pts(Parent, S1);
    yacco2::AST::join_sts(S1, S2);
    yacco2::AST::join_sts(S2, S3);
    yacco2::AST::join_sts(S3, S4);
    yacco2::AST::join_sts(S4, S5);
    yacco2::AST::join_sts(S5, S6);
    yacco2::AST::join_sts(S6, S7);
  }
  void yacco2::AST::crt_tree_of_8sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
          &S2, yacco2::AST &S3, yacco2::AST &S4, yacco2::AST &S5, yacco2::AST
          &S6, yacco2::AST &S7, yacco2::AST &S8)
  {
    yacco2::AST::join_pts(Parent, S1);
    yacco2::AST::join_sts(S1, S2);
    yacco2::AST::join_sts(S2, S3);
    yacco2::AST::join_sts(S3, S4);
    yacco2::AST::join_sts(S4, S5);
    yacco2::AST::join_sts(S5, S6);
    yacco2::AST::join_sts(S6, S7);
    yacco2::AST::join_sts(S7, S8);
  }
  void yacco2::AST::crt_tree_of_9sons(yacco2::AST &Parent, yacco2::AST &S1, yacco2::AST
          &S2, yacco2::AST &S3, yacco2::AST &S4, yacco2::AST &S5, yacco2::AST
          &S6, yacco2::AST &S7, yacco2::AST &S8, yacco2::AST &S9)
  {
    AST::join_pts(Parent, S1);
    AST::join_sts(S1, S2);
    AST::join_sts(S2, S3);
    AST::join_sts(S3, S4);
    AST::join_sts(S4, S5);
    AST::join_sts(S5, S6);
    AST::join_sts(S6, S7);
    AST::join_sts(S7, S8);
    AST::join_sts(S8, S9);
  }
```

481. *content* **of node.**

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::CAbs_lr1_sym *yacco2::AST::content(yacco2::AST &Node)
  {
    return Node.obj_;
  }
```

482. *zero_1st_son* **link.**

⟨ accrue tree code 451 ⟩ +≡

```
  void yacco2::AST::zero_1st_son(yacco2::AST &Node)
  {
    Node.lt_ = 0;
  }
```

483. *zero_2nd_son* **link.**

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::zero_2nd_son(yacco2::AST &Node)
{
  yacco2::AST *lt = Node.lt_;
  if (lt ≡ 0) {
    yacco2::KCHARP msg = "zero_2nd_son␣2nd␣son's␣1st␣son␣Node␣ptr␣is␣zero";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  lt→rt_ = 0;
}
```

484. *zero_brother* **link.**

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::zero_brother(yacco2::AST &Node)
{
  Node.rt_ = 0;
}
```

485. *zero_content*.

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::zero_content(yacco2::AST &Node)
{
  Node.obj_ = 0;
}
```

486. *set_content* **of node.**

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::set_content(yacco2::AST &Node, yacco2::CAbs_lr1_sym &Sym)
{
  Node.obj_ = &Sym;
}
```

487. *zero_previous* **link.**

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::zero_previous(yacco2::AST &Node)
{
  Node.pr_ = 0;
}
```

488. *set_content_wdelete*: **mark node's content to be deleted when node deleted.**

⟨ accrue tree code 451 ⟩ +≡
```
void yacco2::AST::set_content_wdelete(yacco2::AST &Node, yacco2::CAbs_lr1_sym &Sym)
{
  Node.obj_ = &Sym;
  Node.wdelete_ = true;
}
```

489. *set_previous* **link.**

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::set_previous(yacco2::AST &Node, yacco2::AST &Previous_node)
{
  Node.pr_ = &Previous_node;
}
```

490. *wdelete* **is node's contents marked as to-be-deleted?.**

⟨ accrue tree code 451 ⟩ +≡

```
bool yacco2::AST::wdelete(yacco2::AST &Node)
{
  return Node.wdelete_;
}
```

491. *wdelete* **set delete attribute: true or false.**

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::wdelete(yacco2::AST &Node, bool Wdelete)
{
  Node.wdelete_ = Wdelete;
}
```

492. Fetch various tree nodes: *brother***.**

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::brother(yacco2::AST &Node)
{
  return Node.rt_;
}
```

493. *previous* **node: returns its heritage parent or older brother.**
Returns either the older brother or parent if the brother is first in the chain. A root node returns NIL. The difference between *previous* and *get_older_sibling* is in how it treats the oldest brother node. *get_older_sibling* does not return its parent node but returns NIL.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::previous(yacco2::AST &Node)
{
  return Node.pr_;
}
```

494. Birth, pruning, and death of a tree node: AST.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST::AST()
: lt_(0), rt_(0), pr_(0), obj_(0), wdelete_(false) { }
yacco2::AST::AST(yacco2::CAbs_lr1_sym &Obj)
: lt_(0), rt_(0), pr_(0), obj_(&Obj), wdelete_(false) { }
yacco2::AST::~AST()
{
  if (wdelete_ ≡ true) {
    delete obj_;
  }
}
```

495. *join_pts***: parent to son bonding.**

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::join_pts(yacco2::AST &Parent, yacco2::AST &Child)
{
  if (Parent.lt_ ≠ 0) {
    yacco2::KCHARP msg = "join_pts␣Parent␣lt␣ptr␣not␣zero";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Parent ≡ &Child) {
    yacco2::KCHARP msg = "join_pts␣Parent␣and␣child␣nodes␣are␣the␣same";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  Parent.lt_ = &Child;
  Child.pr_ = &Parent;
}
```

496. *join_sts***: brother to brother bonding.**

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::join_sts(yacco2::AST &Elder_sibling, yacco2::AST &Younger_sibling)
{
  if (Elder_sibling.rt_ ≠ 0) {
    yacco2::KCHARP msg = "join_sts␣Elder_sibling␣rt␣ptr␣not␣zero";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Elder_sibling ≡ &Younger_sibling) {
    yacco2::KCHARP msg = "join_sts␣Left␣and␣Right␣nodes␣are␣the␣same";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  Elder_sibling.rt_ = &Younger_sibling;
  Younger_sibling.pr_ = &Elder_sibling;
}
```

497. *ast_delete*: **delete the tree node.**

```
⟨ accrue tree code 451 ⟩ +≡
  void yacco2::AST::ast_delete(yacco2::AST &Node, bool Due_to_abort)
  {
    if (YACCO2_T__ ≠ 0) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_T__::ast_DELETE␣Node␣to␣be␣deleted*:␣" ≪ &Node ≪
          "␣Abort␣switch:␣" ≪ Due_to_abort ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
    if (&Node ≡ Node.lt_) {
      yacco2::KCHARP msg = "ast_delete␣recursion␣to␣self␣Node";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    if (&Node ≡ Node.rt_) {
      yacco2::KCHARP msg = "ast_delete␣Right␣recursion␣to␣self␣Node";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    yacco2::CAbs_lr1_sym *sym = Node.obj_;
    if (YACCO2_T__ ≠ 0) {
      if (sym ≠ 0) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "YACCO2_T__::ast_DELETE␣Node␣to␣be␣deleted*:␣" ≪ &Node ≪
            "␣sym*:␣" ≪ sym ≪ "␣id:␣" ≪ sym→id__ ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
    }
    if (YACCO2_T__ ≠ 0) {
      if (Node.lt_) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "YACCO2_T__::call␣ast_DELETE␣Node␣by␣LEFT␣node␣to␣be␣deleted*:␣" ≪
            Node.lt_ ≪ "␣by␣node*:␣" ≪ &Node ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
        AST::ast_delete(*Node.lt_, Due_to_abort);
      }
      if (Node.rt_) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "call␣ast_DELETE␣Node␣by␣RIGHT␣node␣to␣be␣deleted*:␣" ≪ Node.rt_ ≪
            "␣by␣node*:␣" ≪ &Node ≪ __FILE__ ≪ __LINE__ ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
        AST::ast_delete(*Node.rt_, Due_to_abort);
      }
    }
    if (sym ≠ 0) {      /* is there a sym to work on. if the delete process */
        /* was originally started by delete sym is 0 */
      if (Due_to_abort ≡ true) {
        if (sym→affected_by_abort() ≡ true) {
          if (YACCO2_T__ ≠ 0) {
            ⟨ acquire trace mu 389 ⟩;
```

```
          yacco2::lrclog << "YACCO2_T__::ast_DELETE node's object deleted due to ABORT: " <<
              sym->id__ << __FILE__ << __LINE__ << std::endl;
          ⟨ release trace mu 390 ⟩;
        }
        delete sym;   /* protects against recycled bin deleting its items */
        Node.obj_ = 0;
      }
      ;
    }
    else {   /* normal throes of death */
      delete sym;
      Node.obj_ = 0;
    }
  }
  delete &Node;
  ⟨ acquire trace mu 389 ⟩;
  lrclog << "ast_DELETE Node deleted*: " << &Node << __FILE__ << __LINE__ << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

498. *find_depth*.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::find_depth(AST &Node, yacco2::INT Enum)
{
  if (&Node ≡ Node.lt_) {
    yacco2::KCHARP msg = "find_depth␣Left␣recursion␣to␣self␣Node";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Node ≡ Node.rt_) {
    yacco2::KCHARP msg = "find_depth␣Right␣recursion␣to␣self␣Node";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (Node.obj_ ≡ 0) {
    yacco2::KCHARP msg = "find_depth␣Tree's␣oject␣is␣zero";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  yacco2::CAbs_lr1_sym *sym = Node.obj_;
  if (sym→enumerated_id__ ≡ Enum) return &Node;
  if (Node.lt_ ≠ 0) {
    yacco2::AST *rtn = find_depth(*Node.lt_, Enum);
    if (rtn ≠ 0) return rtn;
  }
  if (Node.rt_ ≠ 0) {
    yacco2::AST *rtn = find_depth(*Node.rt_, Enum);
    if (rtn ≠ 0) return rtn;
  }
  return 0;
}
```

499. *find_breadth*.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::find_breadth(yacco2::AST &Node, yacco2::INT Enum)
{
  if (&Node ≡ Node.lt_) {
    yacco2::KCHARP msg = "find_breadth␣Left␣recursion␣to␣self␣Node";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Node ≡ Node.rt_) {
    yacco2::KCHARP msg = "find_breadth␣Right␣recursion␣to␣self␣Node";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (Node.obj_ ≡ 0) {
    yacco2::KCHARP msg = "find_breadth␣Tree's␣object␣is␣zero";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  yacco2::CAbs_lr1_sym *sym = Node.obj_;
  if (sym→enumerated_id__ ≡ Enum) return &Node;
  if (Node.rt_ ≠ 0) {
    yacco2::AST *rtn = find_breadth(*Node.rt_, Enum);
    if (rtn ≠ 0) return rtn;
  }
  return 0;
}
```

500. Tree relinking routines: before, between, after and other sundries.

501. *relink*.
This drops the old link and re-welds the previous node to the new node. The relationships between the previous and old node are erased. No memory meltdown but pure lobotomy with 2 scoops.

```
⟨ accrue tree code 451 ⟩ +≡
  void yacco2::AST::relink(yacco2::AST &Previous, yacco2::AST &Old_to, yacco2::AST
        &New_to)
  {
    if (&Previous ≡ &Old_to) {
      yacco2::KCHARP msg = "relink␣Previous␣ptr␣==␣Old␣ptr";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    if (&Previous ≡ &New_to) {
      yacco2::KCHARP msg = "relink␣Previous␣ptr␣==␣New␣ptr";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    if (&Old_to ≡ &New_to) {
      yacco2::KCHARP msg = "relink␣Old␣ptr␣==␣New␣ptr";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    if (Previous.rt_ ≡ &Old_to) {
      Old_to.pr_ = 0;
      Previous.rt_ = &New_to;
      New_to.pr_ = &Previous;
      return;
    }
    Old_to.pr_ = 0;
    Previous.lt_ = &New_to;
    New_to.pr_ = &Previous;
  }
```

502. *relink_between*.
This wedges the new node inbetween the previous and old node. Depending on the relationship between the previous and old node, the same relationship is maintained; this can be parental or brotherly love. The new node becomes the older brother to the old node.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::relink_between(yacco2::AST &Previous, yacco2::AST &Old_to, yacco2::AST
        &New_to)
{
  if (&Previous ≡ &Old_to) {
    yacco2::KCHARP msg = "relink_between␣Previous␣ptr␣==␣Old␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Previous ≡ &New_to) {
    yacco2::KCHARP msg = "relink_between␣Previous␣ptr␣==␣New␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (&Old_to ≡ &New_to) {
    yacco2::KCHARP msg = "relink_between␣Old␣ptr␣==␣New␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (Previous.rt_ ≡ &Old_to) {
    Old_to.pr_ = &New_to;
    Previous.rt_ = &New_to;
    New_to.pr_ = &Previous;
    New_to.rt_ = &Old_to;
    return;
  }
  if (Previous.lt_ ≡ &Old_to) {
    Old_to.pr_ = &New_to;
    Previous.lt_ = &New_to;
    New_to.pr_ = &Previous;
    New_to.rt_ = &Old_to;
    return;
  }
  yacco2::KCHARP msg = "ast_relink_between␣Previous␣node␣does␣not␣have␣lt␣or␣rt␣of␣\
      Old";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

503. *relink_after*.
This adds the new node as the previous node's immediate younger brother. If there was a younger brother already established, it re-aligns these relations. There is no politeness; just raw butting in.

```
⟨ accrue tree code 451 ⟩ +≡
  void yacco2::AST::relink_after(yacco2::AST &Previous, yacco2::AST &To)
  {
    if (&Previous ≡ &To) {
      yacco2::KCHARP msg = "relink_after␣Previous␣ptr␣==␣To␣ptr";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    if (Previous.rt_ ≡ 0) {    /* eoc */
      Previous.rt_ = &To;
      To.pr_ = &Previous;
      return;
    }
    AST *rt = Previous.rt_;
    if (rt→pr_ ≡ &Previous) {
      rt→pr_ = &To;
      Previous.rt_ = &To;
      To.pr_ = &Previous;
      To.rt_ = rt;
      return;
    }
    yacco2::KCHARP msg = "relink_after␣Previous␣Node␣does␣not␣have␣lt␣or␣rt␣of␣Old";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
```

504. *relink_before*.

The new node is added before the 'Before' node. Depending on the Before's node relationship as either the oldest child or a younger sibling, *relink_before* maintains this relationship with the *New_to* node while the 'Before' node becomes *New_to*'s younger brother.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::relink_before(yacco2::AST &Before, yacco2::AST &New_to)
{
  if (&Before ≡ &New_to) {
    yacco2::KCHARP msg = "relink_before␣Before␣ptr␣==␣New␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  if (Before.pr_ ≡ 0) {      /* eoc */
    Before.pr_ = &New_to;
    New_to.rt_ = &Before;
    return;
  }
  yacco2::AST *pr = Before.pr_;
  if (pr→lt_ ≡ &Before) {
    pr→lt_ = &New_to;
    New_to.pr_ = pr;
    New_to.rt_ = &Before;
    Before.pr_ = &New_to;
    return;
  }
  if (pr→rt_ ≡ &Before) {
    pr→rt_ = &New_to;
    New_to.pr_ = pr;
    New_to.rt_ = &Before;
    Before.pr_ = &New_to;
    return;
  }
  yacco2::KCHARP msg = "relink_before␣Before␣node␣does␣not␣have␣lt␣or␣rt␣of␣Old";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

505. *replace_node*.
Substitute the Old node with the By node. Remap all the relations to the By node and wipe out relationships in Old node leaving it as an orphan.

⟨ accrue tree code 451 ⟩ +≡

```
void yacco2::AST::replace_node(yacco2::AST &Old, yacco2::AST &By)
{
  if (&Old ≡ &By) {
    yacco2::KCHARP msg = "replace_node␣Old␣ptr␣==␣By␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  yacco2::AST *prev = Old.pr_;
  yacco2::AST *rt = Old.rt_;
  if (prev→rt_ ≡ &Old) {
    prev→rt_ = &By;
    By.pr_ = prev;
    By.rt_ = rt;
    if (rt ≠ 0) rt→pr_ = &By;
    Old.rt_ = 0;
    Old.pr_ = 0;
    return;
  }
  if (prev→lt_ ≡ &Old) {
    prev→lt_ = &By;
    By.pr_ = prev;
    By.rt_ = rt;
    if (rt ≠ 0) rt→pr_ = &By;
    Old.rt_ = 0;
    Old.pr_ = 0;
    return;
  }
  By.rt_ = Old.rt_;
  Old.rt_ = 0;
}
```

506. Various tree node routines.

507. *add_son_to_tree*.
Just wedge the new kid as an oldest child with the Parent node. If the Parent node is childless... well congratulations. If there are already children, well let the probate officer deal with the squawkes.

```
⟨ accrue tree code 451 ⟩ +≡
  void yacco2::AST::add_son_to_tree(yacco2::AST &Parent, yacco2::AST &Son)
  {
    AST *p_lt = Parent.lt_;
    if (p_lt ≡ 0) {
      Parent.lt_ = &Son;
      Son.pr_ = &Parent;
      return;
    }
    Parent.lt_ = &Son;
    Son.pr_ = &Parent;
    Son.rt_ = p_lt;
    p_lt→pr_ = &Son;
  }
```

508. *add_child_at_end*.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::add_child_at_end(yacco2::AST &Tree, yacco2::AST &Child)
  {
    yacco2::AST *cur_youngest_child = AST::get_child_at_end(Tree);
    if (cur_youngest_child ≡ 0) {
      AST::join_pts(Tree, Child);
    }
    else {
      AST::join_sts(*cur_youngest_child, Child);
    }
    return &Child;
  }
```

509. *get_spec_child*.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::get_spec_child(yacco2::AST &Tree, yacco2::INT Cnt)
{
  if (Cnt ≤ 0) {
    yacco2::KCHARP msg = "get_spec_child␣Node␣Cnt␣is␣<=␣0";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
  yacco2::INT pos(0);
  yacco2::AST *ct = Tree.lt_;
  for ( ; ct ≠ 0; ct = ct→rt_) {
    ++pos;
    if (pos ≡ Cnt) return ct;
  }
  return 0;
}
```

510. Get specific son node by number.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::get_1st_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 1);
}
yacco2::AST *yacco2::AST::get_2nd_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 2);
}
yacco2::AST *yacco2::AST::get_3rd_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 3);
}
yacco2::AST *yacco2::AST::get_4th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 4);
}
yacco2::AST *yacco2::AST::get_5th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 5);
}
yacco2::AST *yacco2::AST::get_6th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 6);
}
yacco2::AST *yacco2::AST::get_7th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 7);
}
yacco2::AST *yacco2::AST::get_8th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 8);
}
yacco2::AST *yacco2::AST::get_9th_son(yacco2::AST &Node)
{
  return get_spec_child(Node, 9);
}
```

511. *get_child_at_end*. Go thru the parent's children looking for the youngest.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::get_child_at_end(yacco2::AST &Tree)
  {
    yacco2::AST *ct = Tree.lt_;
    yacco2::AST *pct(0);
    for ( ; ct ≠ 0; ct = ct→rt_) {
      pct = ct;
    }
    return pct;
  }
```

512. *get_youngest_sibling*.
If there is no younger brother then a `NIL` pointer is returned indicating such condition. It is up to the user to check the validity.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::get_youngest_sibling(yacco2::AST &Tree)
  {
    yacco2::AST *start = &Tree;
    yacco2::AST *younger_sibling = start;
    for ( ; younger_sibling ≠ 0; ) {
      if (younger_sibling→rt_ ≡ 0) break;
      younger_sibling = younger_sibling→rt_;
    }
    if (start ≡ younger_sibling) return 0;
    return younger_sibling;
  }
```

513. *get_younger_sibling*.
It goes right along the brother chain looking for the brother x youngest to him.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::get_younger_sibling(yacco2::AST &Child, yacco2::INT Pos)
  {
    if (Pos ≤ 0) {
      yacco2::KCHARP msg = "get_younger_sibling␣Pos␣<=␣0";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    int cnt(0);
    yacco2::AST *younger_sibling = Child.rt_;
    for ( ; younger_sibling ≠ 0; younger_sibling = younger_sibling→rt_) {
      ++cnt;
      if (cnt ≡ Pos) return younger_sibling;
    }
    return 0;
  }
```

514. *get_older_sibling*: **returns only older brother.**
It goes to its left along the brother chain in older order. If it is the first in the breadth chain, well, it's the end and returns a nil ptr.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::get_older_sibling(yacco2::AST &Child, yacco2::INT Pos)
  {
    if (Pos ≥ 0) {
      yacco2::KCHARP msg = "get_older_sibling␣Pos␣>=␣0";
      Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
      exit(1);
    }
    int cnt(0);
    AST *older_sibling = Child.pr_;
    for ( ; older_sibling ≠ 0; older_sibling = older_sibling→pr_) {
      −− cnt;
      if (cnt ≡ Pos) return older_sibling;
    }
    return 0;
  }
```

515. *get_parent*: **child guidance required.**

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::AST *yacco2::AST::get_parent(yacco2::AST &Tree)
  {
    yacco2::AST *cnode = &Tree;
    yacco2::AST *older_sibling = cnode→pr_;
    for ( ; older_sibling ≠ 0; cnode = older_sibling, older_sibling = cnode→pr_) {
      if (older_sibling→rt_ ≠ cnode) return older_sibling;    /* */
    }
    return 0;
  }
```

516. *common_ancestor*: **Are we distant ?.**

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::common_ancestor(yacco2::Type_AST_ancestor_list &ListA,
          yacco2::Type_AST_ancestor_list &ListB){ Type_AST_ancestor_list *a;
    Type_AST_ancestor_list *b;
    if (ListA.size() < ListB.size()) {
      a = &ListA;
      b = &ListB;
    }
    else {
      b = &ListA;
      a = &ListB;
    }
    Type_AST_ancestor_list
        ::iterator ai = a→begin(); Type_AST_ancestor_list
            ::iterator aie = a→end(); Type_AST_ancestor_list
                ::iterator bi;
            Type_AST_ancestor_list
                ::iterator bie;
            for ( ; ai ≠ aie; ++ai) {
              bi = b→begin();
              bie = b→end();
              for ( ; bi ≠ bie; ++bi) {
                AST *A = *ai;
                AST *B = *bi;
                if (A ≡ B) return A;
              }
            }
            return 0; }
```

517. *divorce_node_from_tree*.
Never pretty but civil in its settlement.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::divorce_node_from_tree(yacco2::AST &Node)
{
  yacco2::AST *bpr = Node.pr_;
  yacco2::AST *brt = Node.rt_;
  ⟨ remove node's association from tree 524 ⟩;
  ⟨ dispatch to node association: forest, among brothers, or parental 518 ⟩;
forest:
  ⟨ handle a forest situation, with or without a younger brother 522 ⟩;
amongst_brothers:
  ⟨ handle sibling relationship 523 ⟩;
parental_guidance:
  ⟨ handle parent / sibling relationship 519 ⟩;
}
```

518. Dispatch to node association.
The following points are the sequences checked on the removed node's relationship within the tree structure.

1) forest — only node or oldest node in the forest
2) middle or youngest node in the forest
3) parent with one or more children

⟨ dispatch to node association: forest, among brothers, or parental 518 ⟩ ≡
```
if (bpr ≡ 0) goto forest;
if (bpr→rt_ ≡ &Node) goto amongst_brothers;
if (bpr→lt_ ≡ &Node) goto parental_guidance;
```
This code is used in section 517.

519. Handle parent / sibling relationship. Is it an only child? If so, then remove the parent relationship. If there are brothers, then re-align the relationships in both the parent and the younger child.

⟨ handle parent / sibling relationship 519 ⟩ ≡
```
⟨ only child? yes make parent childless and exit 520 ⟩;
⟨ re-bond younger child with parent and exit with child 521 ⟩;
```
This code is used in section 517.

520. Only child? yes make parent childless and exit.

⟨ only child? yes make parent childless and exit 520 ⟩ ≡
```
if (brt ≡ 0) {
  bpr→lt_ = 0;
  return 0;
}
```
This code is used in section 519.

521. Re-bond younger child with parent.

⟨ re-bond younger child with parent and exit with child 521 ⟩ ≡
```
bpr→lt_ = brt;
brt→pr_ = bpr;
return brt;
```
This code is used in section 519.

522. Handle a forest situation, with or without a younger brother.
It is considered a forest if there is no older brother attached to it. If it has a brother, disconnect the younger brother's association from he removed node, and back back the younger brother.

⟨ handle a forest situation, with or without a younger brother 522 ⟩ ≡
```
if (brt ≡ 0) return 0;   /* onlynode; */
brt→pr_ = 0;
return brt;
```
This code is used in section 517.

523. Handle sibling relationship. This situation is:
a $\rightarrow b \rightarrow ?$ where ? is either nil or a node. So, relink node a with its younger brother c.

⟨ handle sibling relationship 523 ⟩ ≡
```
bpr→rt_ = brt;
if (brt ≠ 0) brt→pr_ = bpr;
return brt;
```
This code is used in section 517.

524. Remove node's association from tree.

⟨ remove node's association from tree 524 ⟩ ≡

```
Node.pr_ = 0;
Node.rt_ = 0;
```

This code is used in section 517.

525. *clone_tree*.

logic: walk the tree in prefix

Ip:

1) To - node to copy
2) Calling - predecessor: if 0, no predecesor. it's the root
3) Relation - join options: init,left,right

Op:

new tree with each new node's content being a duplicate

The new tree is a complete copy. The tree nodes are fresh from the malloc bakery with their contents being the same.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::AST *yacco2::AST::clone_tree(yacco2::AST &Node_to_copy, yacco2::AST
        *Calling_node, yacco2::ast_base_stack::n_action Relation)
{
  yacco2::AST *new_t = new yacco2::AST(*yacco2::AST::content(Node_to_copy));
    /* copy node */
  switch (Relation) {    /* how to join */
  case ast_base_stack::init: break;    /* root */
  case ast_base_stack::left:
    {
      if (Calling_node ≠ 0) {
        AST::join_pts(*Calling_node, *new_t);
      }
      break;
    }
  case ast_base_stack::right:
    {
      if (Calling_node ≠ 0) {
        AST::join_sts(*Calling_node, *new_t);
      }
      break;
    }
  }
  if (Node_to_copy.lt_ ≠ 0) AST::clone_tree(*Node_to_copy.lt_, new_t, ast_base_stack::left);
  if (Node_to_copy.rt_ ≠ 0) AST::clone_tree(*Node_to_copy.rt_, new_t, ast_base_stack::right);
  return new_t;
}
```

526. Some tree functors: remove, insert back, print a tree, etc.
These functors are examples of how to create your own functor. *prt_ast_functor* prints out a tree in indented format. *fire_a_func_functor* just calls a procedure passing it the current tree node. *str_ast_functor* claim to fame is in its use of the `BYPASS_FILTER` given the many abstract meta-terminal 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 an 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 orginally 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.

⟨Structure defs 18⟩ +≡
```
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_;
};
```

527. *tok_can_ast_functor*.

⟨Structure defs 18⟩ +≡
```
struct tok_can_ast_functor : public Type_AST_functor {
    functor_result_type operator()(yacco2::ast_base_stack *Stk_env);
};
```

528. *tok_can_ast_no_stop_functor*.

⟨Structure defs 18⟩ +≡
```
struct tok_can_ast_no_stop_functor : public Type_AST_functor {
    functor_result_type operator()(yacco2::ast_base_stack *Stk_env);
};
```

529. *tok_can_ast_bypass_functor*.

⟨Structure defs 18⟩ +≡
```
struct tok_can_ast_bypass_functor : public Type_AST_functor {
    functor_result_type operator()(yacco2::ast_base_stack *Stk_env);
};
```

530. *prt_ast_functor*.

⟨ Structure defs 18 ⟩ +≡

```
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_;
};
```

531. *fire_a_func_ast_functor*.

⟨ Structure defs 18 ⟩ +≡

```
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_;
};
```

532. *str_ast_functor* — **build up source string.**

⟨ Structure defs 18 ⟩ +≡

```
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];
};
```

533. *remove_unwanted_ast_functor*.

⟨ Structure defs 18 ⟩ +≡

```
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_;
};
```

534. Implementation of some functors. remove_unwanted_ast_functor.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::functor_result_type
        yacco2::remove_unwanted_ast_functor::operator()(yacco2::ast_base_stack
        *Stk_env)
{
  stk_env_ = Stk_env;
  srec_ = stk_env_→cur_stk_rec_;
  idx_ = stk_env_→idx_;
  cnode_ = srec_→node_;
  yacco2::CAbs_lr1_sym *sobj = AST::content(*cnode_);
  if (sobj ≡ 0) return accept_node;
  if (sobj→tok_co_ords__.external_file_id__ ≤ 1) return accept_node;
  idx_ = stk_env_→idx_;
  if (stk_env_→idx_ ≡ 0) {    /* 1st entry of complete tree */
    return accept_node;
  }
  return bypass_node;    /* cuz: apple's symantic error */
}
void yacco2::remove_unwanted_ast_functor::possible_delete()
{
  yacco2::INT pidx = idx_ − 1;
  if (pidx < 0) return;
  ast_base_stack::s_rec *psrec = stk_env_→stk_rec(pidx);
  yacco2::AST *psnode = psrec→node_;
  yacco2::AST *srt = AST::brother(*cnode_);
  switch (psrec→act_) {
  case ast_base_stack::left:
    {
      if (srt ≠ 0) {    /* replace current record with rt node: shift left tree */
        yacco2::AST::relink(*psnode, *cnode_, *srt);
        srec_→node_ = srt;
        srec_→act_ = ast_base_stack::init;
        return;
      }
      yacco2::AST::zero_1st_son(*psnode);
      srec_→act_ = ast_base_stack::eoc;    /* deleted node: complete its seq */
      return;
    }
  case ast_base_stack::right:
    {
      if (srt ≠ 0) {
        yacco2::AST::relink(*psnode, *cnode_, *srt);
        srec_→node_ = srt;
        srec_→act_ = ast_base_stack::init;
        return;
      }
      yacco2::AST::zero_brother(*psnode);
      srec_→act_ = ast_base_stack::eoc;    /* deleted node: complete its seq */
      return;
```

```
      }
    default:
      {
        return;
      }
    }
}
yacco2::remove_unwanted_ast_functor::~remove_unwanted_ast_functor()
{ }
```

535. Insert items back into a tree.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::functor_result_type
        yacco2::insert_back_recycled_items_functor::operator()(yacco2::ast_base_stack
        *Stk_env)
  {
    stk_env_ = Stk_env;
    srec_ = stk_env_→cur_stk_rec_;
    idx_ = stk_env_→idx_;
    cnode_ = srec_→node_;
    yacco2::CAbs_lr1_sym *top_node_sym = AST::content(*cnode_);
    yacco2::CAbs_lr1_sym *node_sym = AST::content(*insert_node_);
    if (node_sym→tok_co_ords__.rc_pos__ ≤ top_node_sym→tok_co_ords__.rc_pos__) return accept_node;
    return bypass_node;    /* cuz: apple's symantic error */
  }
  void yacco2::insert_back_recycled_items_functor::insert_node(yacco2::AST &Inode)
  {
    insert_node_ = &Inode;
  }
  yacco2::AST *yacco2::insert_back_recycled_items_functor::new_root()
  {
    return new_root_;
  }
  void yacco2::insert_back_recycled_items_functor::insert_before()
  {
    if (stk_env_→idx_ > 0) goto overlay;
  root_change:
    new_root_ = insert_node_;
  overlay:    /* overlay cur node with new node to insert */
    srec_→node_ = insert_node_;
    srec_→act_ = ast_base_stack::right;
    AST::join_sts(*insert_node_, *cnode_);
      /* adj visited node: default to visit cuz next ast could be ¡ than it */
    stk_env_→push(*cnode_, ast_base_stack::visit);
  adj_prev_caller:
    if (stk_env_→idx_ ≡ 0) return;    /* only root */
    yacco2::INT pi = idx_ − 1;
    yacco2::ast_base_stack::s_rec *pcur_rec = stk_env_→stk_rec(pi);
    yacco2::AST *pnode = pcur_rec→node_;
    switch (pcur_rec→act_) {
    case yacco2::ast_base_stack::left:
      {
        yacco2::AST::zero_1st_son(*pnode);
        yacco2::AST::join_pts(*pnode, *insert_node_);
        return;
      }
    case yacco2::ast_base_stack::right:
      {
        yacco2::AST::zero_brother(*pnode);
        yacco2::AST::join_sts(*pnode, *insert_node_);
```

```
        return;
      }
    }
    return;
  }
```

536. tok_can_ast_functor continue looping thru the tree.

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::functor_result_type yacco2::tok_can_ast_functor::operator()(ast_base_stack *Stk_env)
  {
    return accept_node;   /* stop looping thru ast */
  }
```

537. tok_can_ast_no_stop_functor stop looping thru the tree.

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::functor_result_type yacco2::tok_can_ast_no_stop_functor::operator()(ast_base_stack
          *Stk_env)
  {
    return stop_walking;   /* continue looping thru ast */
  }
```

538. tok_can_ast_bypass_functor.

⟨ accrue tree code 451 ⟩ +≡

```
  yacco2::functor_result_type yacco2::tok_can_ast_bypass_functor::operator()(ast_base_stack
          *Stk_env)
  {
    yacco2::ast_base_stack::s_rec *srec = Stk_env→cur_stk_rec_;
    yacco2::AST *cnode = srec→node_;
    yacco2::CAbs_lr1_sym *sym = AST::content(*cnode);
    if (sym→tok_co_ords__.external_file_id__ > 1) return bypass_node;
        /* contine the walk, not wanted */
    return bypass_node;
  }
```

539. prt_ast_functor.

⟨ accrue tree code 451 ⟩ +≡

```
yacco2::functor_result_type yacco2::prt_ast_functor::operator()(yacco2::ast_base_stack
        *Stk_env)
  {
    stk_env_ = Stk_env;
    srec_ = stk_env_→cur_stk_rec_;
    idx_ = stk_env_→idx_;
    yacco2::INT pidx = idx_ − 1;
    cnode_ = srec_→node_;    /* std::string how; */
    if (pidx ≤ 0) goto prt_prefix;
    {
      ast_base_stack::s_rec *psrec = stk_env_→stk_rec(pidx);
      if (psrec→act_ ≡ ast_base_stack::left) {
        how_[0] = 'l';
      }
      else {
        how_[0] = 'r';
      }
      how_[1] = 't';
      how_[2] = (char) 0;
    }
  prt_prefix:
    ⟨ acquire trace mu 389 ⟩;
    yacco2::INT no_lt(0);
    for (yacco2::INT x = 0; x ≤ idx_; ++x)
      if (stk_env_→stk_rec(x)→act_ ≡ ast_base_stack::left) ++no_lt;
    for (yacco2::INT x = 0; x ≤ no_lt; ++x) (*ofile_) ≪ "␣";
    (*ofile_) ≪ ++cnt_ ≪ "::" ≪ '␣';
    ⟨ release trace mu 390 ⟩;
  call_prt_func:
    (*prt_funct_)(cnode_);
    return accept_node;    /* continue looping thru ast */
  }
yacco2::prt_ast_functor::prt_ast_functor(PF Func, std::ofstream * Ofile): prt_funct_(Func), cnt_(0)
  {
    if (Ofile ≡ 0) {
      ofile_ = &yacco2::lrclog;
    }
    else {
      ofile_ = Ofile;
    }
  }
  void yacco2::prt_ast_functor::reset_cnt()
  {
    cnt_ = 0;
  }
```

540. fire_a_func_ast_functor.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::functor_result_type
        yacco2::fire_a_func_ast_functor::operator()(yacco2::ast_base_stack
        *Stk_env)
  {
    stk_env_ = Stk_env;
    srec_ = stk_env_→cur_stk_rec_;
    idx_ = stk_env_→idx_;
    yacco2::INT pidx = idx_ − 1;
    cnode_ = srec_→node_;
  call_prt_func:
    (*a_funct_)(cnode_);
    return accept_node;    /* continue looping thru ast */
  }
yacco2::fire_a_func_ast_functor::fire_a_func_ast_functor(PF Func): a_funct_(Func)
  { }
```

541. str_ast_functor.

```
⟨ accrue tree code 451 ⟩ +≡
  yacco2::functor_result_type yacco2::str_ast_functor::operator()(yacco2::ast_base_stack
        *Stk_env)
  {
    stk_env_ = Stk_env;
    srec_ = stk_env_→cur_stk_rec_;
    idx_ = stk_env_→idx_;
    yacco2::INT pidx = idx_ − 1;
    cnode_ = srec_→node_;    /* std::string how; */
    if (pidx ≤ 0) goto prt_prefix;
    {
      ast_base_stack::s_rec *psrec = stk_env_→stk_rec(pidx);
      if (psrec→act_ ≡ ast_base_stack::left) {
        how_[0] = 'l';
      }
      else {
        how_[0] = 'r';
      }
      how_[1] = 't';
      how_[2] = (char) 0;
    }
  prt_prefix:
    call_prt_func:
    (*prt_funct_)(cnode_, this);
    return accept_node;    /* continue looping thru ast */
  }
yacco2::str_ast_functor::str_ast_functor(PF Func): prt_funct_(Func)
  {
    source_str_.clear();
  }
```

542. Constraints. Validation code snippets.
This is the source collector of all constraints used across Yacco2's code. Why one place insead of keeping the code close to the routines using them? Good question. Code comprehension demands that the code be within the the reading periphery of the programmer. But, code clutter can remove this advantage to understanding. *cweb* provides a better way to do it. You can still use the code clutter approach but it provides a better way. Just describe the code block with intention and reference it. No need to keep the code near by! Gardening chores are tidier, one-place-only to correct and improve.

543. Invalid use of |?| instead of |+| symbol.

```
⟨ Invalid |?| instead of |+| use 543 ⟩ ≡
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Bad␣use␣of␣|?|␣instead␣of␣|+|␣symbol␣or␣epsilon␣sub\
      rule.␣""Correct␣%s␣grammar,␣parse␣state:␣%i.␣Cannot␣continue␣parsing.";
  sprintf(a, msg, fsm_tbl__→id__, parse_stack__.top__→state__→state_no__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
```

This code is cited in section 700.

This code is used in section 253.

544. Validate any token for parsing.

```
⟨ Validate any token for parsing 544 ⟩ ≡
  if (current_token__ ≡ 0) {
    yacco2::KCHARP msg = "Error␣-␣current␣token␣ptr␣zero.␣Cannot␣continue␣parsing.";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
```

This code is used in section 254.

545. Validate Line no parameter.

```
⟨ Validate Line no parameter 545 ⟩ ≡
  if (Line_no < 1) {
    yacco2::KCHARP msg = "Error␣-␣Line_no␣not␣1␣or␣greater";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
```

This code is used in section 74.

546. Validate Pos parameter.

```
⟨ Validate Pos parameter 546 ⟩ ≡
  if (Pos < 1) {
    yacco2::KCHARP msg = "Error␣-␣Pos␣not␣1␣or␣greater";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
  }
```

This code is used in section 72.

547. Validate Pos in line parameter.

⟨ Validate Pos in line parameter 547 ⟩ ≡
```
if (Pos_in_line < 1) {
    yacco2::KCHARP msg = "Error␣-␣Pos_in_line␣not␣1␣or␣greater";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
}
```
This code is used in section 74.

548. Validate File no parameter.

⟨ Validate File no parameter 548 ⟩ ≡
```
if (File_no < 1) {
    yacco2::KCHARP msg = "Error␣-␣File_no␣not␣1␣or␣greater";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
}
```
This code is used in sections 56 and 73.

549. Validate any symbol for co-ordinate setting to relate to?.

⟨ Validate any symbol for co-ordinate setting to relate to? 549 ⟩ ≡
```
if (pt ≡ 0) {
    ;
    yacco2::KCHARP msg = "Error␣-␣no␣supplier␣symbol␣found␣to␣relate␣to␣for␣co-ordina\
        te␣setting";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
}
```

550. Validate parser's finite state table.

⟨ Validate parser's finite state table 550 ⟩ ≡
```
if (parser→fsm_tbl__ ≡ 0) {
    yacco2::KCHARP msg = "Error␣-␣parser's␣finite␣state␣table␣is␣zero␣ptr";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
}
```
This code is used in section 636.

551. Validate that parser stack is not empty.

⟨ Validate that parser stack is not empty 551 ⟩ ≡
```
if (parser→parse_stack__.top_sub__ < 1) {
    yacco2::KCHARP msg = "Error␣-␣parser's␣stack␣is␣empty";
    Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
    exit(1);
}
```
This code is used in section 636.

552. Validate if parser's supplier exists.

⟨ Validate if parser's supplier exists 552 ⟩ ≡

```
if (token_supplier__ ≡ 0) {
  yacco2::KCHARP msg = "Error␣-␣parser's␣supplier␣is␣zero␣ptr";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in sections 338 and 365.

553. Validate if subscript within supplier's bnds.

⟨ Validate if subscript within supplier's bnds 553 ⟩ ≡

```
if (Pos > token_supplier__→size()) {
  yacco2::KCHARP msg = "Error␣-␣Pos␣out␣of␣bounds␣against␣supplier";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in sections 338 and 365.

554. Validate subscript not ≤ 0. The subscript must be a positive integer. This condition is now controlled by *Token_start_pos* macro. The original subscript is relative to 0. My preference is relative to 1. So, provide a mechanism to change in one place so that these conditions can be experimented with.

⟨ Validate subscript not ≤ 0 554 ⟩ ≡

```
if (Pos < Token_start_pos) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Subcript␣Pos␣value␣<␣%i␣---␣out␣of␣bounds";
  sprintf(a, msg, Token_start_pos);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 346.

555. Validate parse stack number of removal items.

⟨ Validate parse stack number of removal items 555 ⟩ ≡

```
if (No_to_remove < 0) {
  yacco2::KCHARP msg = "Error␣-␣parse␣stack␣number␣of␣removal␣items␣<␣0";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 350.

556. Validate parse stack removal for underflow.

⟨ Validate parse stack removal for underflow 556 ⟩ ≡

```
if (parse_stack__.top_sub__ < 1) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "FSM␣id:␣%s␣parse␣stack␣empty!";
  sprintf(a, msg, fsm_tbl__→id__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
if (No_to_remove > MAX_LR_STK_ITEMS) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Underflow␣FSM␣id:␣%s␣on␣parse␣stack␣size:%i␣removal\
      ␣request:␣%i␣";
  sprintf(a, msg, fsm_tbl__→id__, MAX_LR_STK_ITEMS, No_to_remove);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 350.

557. Validate error queue.

⟨ Validate error queue 557 ⟩ ≡

```
if (error_queue__ ≡ 0) {
  yacco2::KCHARP msg = "Error␣-␣Trying␣to␣add␣to␣Parser␣error_queue__␣which␣is␣zero\
      ␣ptr";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 332.

558. Error shift symbol not fnd in fsm table.

The reason for not using a Error type T is that this is below the language being parsed. It would force having a pre-canned error terminal in the error class of the language being defined: lr constants and rc terminals are enough cement. That's the short of it.

⟨ Error shift symbol not fnd in fsm table 558 ⟩ ≡

```
if (se ≡ 0) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Can't␣find␣symbol␣to␣shift␣in␣FSM␣id:␣%s␣state:␣%i␣\
      sym-id:␣%i␣literal:␣%s";
  CAbs_lr1_sym *xxx = current_token();
  sprintf(a, msg, fsm_tbl__→id__, pr→state__→state_no__, xxx→enumerated_id(), xxx→id());
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  yacco2::KCHARP msg2 = "T␣co-ordinates:␣file:␣%s␣GPS␣LINE:␣%i␣GPS␣CHR␣POS:␣%i";
  sprintf(a, msg2, xxx→tok_co_ords__.external_file_id__ ≠ MAX_USINT ?
      yacco2::FILE_TBL__[xxx→tok_co_ords__.external_file_id__].c_str() : "␣No␣external␣file",
      xxx→tok_co_ords__.line_no__, xxx→tok_co_ords__.pos_in_line__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in sections 265 and 267.

559. Validate if rule shift symbol in fsm table.

⟨ Validate if rule shift symbol in fsm table 559 ⟩ ≡

```
if (se ≡ 0) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Cant␣find␣rule␣shift␣in␣state␣FSM␣id:␣%s␣state:␣%i␣\
      rule␣id:␣%i␣";
  sprintf(a, msg, fsm_tbl__→id__, parse_stack__.top__→state__→state_no__, (*rule_rec)→rule_→enumerated_id__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in section 243.

560. Validate reduce entry.

⟨ Validate reduce entry 560 ⟩ ≡

```
if (re ≡ 0) {
  char a[BUFFER_SIZE];
  yacco2::KCHARP msg = "Error␣-␣Cant␣find␣parallel␣sym␣reduce␣in␣FSM␣id:␣%s␣state:␣\
      %i␣token:␣%s␣subs:␣%i␣";
  sprintf(a, msg, fsm_tbl__→id__, pr→state__→state_no__, current_token__→id__, current_token_pos__);
  Yacco2_faulty_precondition(a, __FILE__, __LINE__);
  exit(1);
}
```

This code is used in sections 256 and 260.

561. Validate accept message.

⟨ Validate accept message 561 ⟩ ≡

```
if (arbitrated_token__→accept_token_pos__ ≡ arbitrated_token__→la_token_pos__) {
  yacco2::KCHARP msg = "Error␣-␣Parallel␣token␣boundry␣same␣as␣LA␣token␣boundry";
  Yacco2_faulty_precondition(msg, __FILE__, __LINE__);
  exit(1);
}
```

562. Error bad character mapping.

⟨ Error bad character mapping 562 ⟩ ≡

```
char a[BUFFER_SIZE];
yacco2::KCHARP msg = "Error␣-␣Bad␣char␣mapping␣chr␣value:␣%i";
sprintf(a, msg, Char);
Yacco2_faulty_precondition(a, __FILE__, __LINE__);
exit(1);
```

This code is used in section 56.

563. Error no more raw character storage.

⟨ Error no more raw character storage 563 ⟩ ≡

```
char a[BUFFER_SIZE];
yacco2::KCHARP msg = "Error␣-␣Sorry␣run␣out␣of␣raw␣character␣storage:␣need␣to␣reg\
    en␣Yacco2:␣%i";
sprintf(a, msg, SIZE_RC_MALLOC);
Yacco2_faulty_precondition(a, __FILE__, __LINE__);
exit(1);
```

This code is used in section 57.

564. Macro definitions.
I use macros of C++ and *cweb* variety. Their use covers terminal constructor initialization, tracing of flow control events, parse stack configuration and syntax directed directives, utilities to deal with specific parse situations or results, and aid macros to debug grammars.

As log trace files can be volumous, i placed within each logged message the macro variable's name that controls its output. For example, `YACCO2_MSG__` controls signalling between threads as in wait-for-wakeup message from one of the called threads etc. I'll see how refined this is by use of an UNIX shell's scripting language like "bash" with piping. I'll let u posted.

565. Copyright.
⟨copyright notice 565⟩ ≡ /* copyright */

This code is used in sections 35, 36, 42, 55, 76, 169, 188, 191, 193, 203, and 450.

566. ***EXTERNAL_GPSing* macro is used to print out T's external file.**
The external file comes from **tok_can** container use that registers the external files processed with `FILE_TBL__`. A created T has a subscript reference into this stack. Sanity check must exist against the `FILE_TBL__` registrar or a possible out-of-subscript error could be thrown.

One misuse is to process the "command line" input where the input is written to a holding file. A hardwiring of 1 for the file is used as the "holding file" is the first file inputted to *Yacco2*. But if the holding file name is illegal, a T error of "bad file inputted" created with this file reference as crap. The other potential error is the CLI inputted file is non existent and creating the error T referenced to the holding file which has not been registered with `FILE_TBL__` thru **tok_can**⟨*ifstream*⟩ container create also becomes poop-poop.

Now u defined an Error_handler grammar to trace out those errors expecting to see the traced output with the external file name and its contents line references. Say the "holding file" exists with the command line data placed there but never registered the holding file with the **tok_can**. Hence the **non registering of the CLI holding file** will not be printed by the parser / Error processing grammar. See the "./grammar-testsuite/testout.pdf" program as an example of "command line processing" to avoid the above errors.

```
#define EXTERNAL_GPSing(TOK__)
    if (TOK__→tok_co_ords__.external_file_id__ < yacco2::FILE_TBL__.size()) {
      yacco2::lrclog ≪ yacco2::FILE_TBL__[TOK__→tok_co_ords__.external_file_id__].c_str();
    }
    else {
      yacco2::lrclog ≪ "␣EXTERNAL_GPSing␣-␣No␣external␣file␣registered␣to␣use" ≪
          "␣stack␣subscript:␣" ≪ TOK__→tok_co_ords__.external_file_id__;
    }
```

567. **`FILE_LINE` macro source file co-ordinates for tracing.**
Add the file and line number to the dynamic tracing output. Allows one to go to the source code if things are askew. Gum stuck to your shoe but hey it's an indication.

```
#define FILE_LINE '␣' ≪ __FILE__ ≪ ":␣" ≪ __LINE__
```

568. T_CTOR macro is used by the terminal defs supplied to the grammar.
When a terminal definition needs to be customized, the grammar writer can roll his own class definition. It just initializes the base variables within the class constructor's implementation. Its name is composed of *T* indicating for terminals, and the CTOR uses the C++ naming convention to indicate that it belongs to the class constructor. Please have a look at Yacco2's *yacco2_k_symbols.lex* file that defines the lr constants definitions for a demonstration of use. For the moment there are 5 parameters: A..E. Originally there was more to handle the push-pop-lookahead functors. From Yacco2's use, these functors were never needed. It was only during my Master's thesis that they got their 15 minutes of fame.

Parameter A: provides the terminal's literal name for tracing

Parameter B is the enumerated value

It is symbolically gened by prefixing an T_ to the 'C++' name of the terminal and ending it off with a _ suffix. This is described in *Enumeration of Alphabets*.

Parameter C is the address of the class destructor function or nil

I know, this should be automatically detected by Yacco2's parse generator but for now this is reality: still outstanding.

parser__ is the associated parser for the grammar used by the grammar's rules. As the **CAbs_lr1_sym** is a base structure for both the terminals and rules of the grammar, it has no associated parser for the terminals as terminals are nomadic by nature. Normally *tok_co_ords__*'s attributes are overriden by a raw character co-ordinates. Terminals are composites of other basic entities like raw character terminals.

Parameter D is auto delete boolean value of ON or OFF

Parameter E is auto abort boolean value of ON or OFF

An auto delete attribute indicates that the terminal is deleted when popped from the parse stack. When an abort of a parse occurs, this attribute when turned on indicates that the object should be deleted. It's a 'clean up your own mess' attribute. Both paramaters relate to the terminal's 'AD' and 'AB' grammatical attributes. An example of T_CTOR use is:

T_CTOR("labeled-stmt",*T_T_labeled_stmt_*,&*dtor_T_labeled_stmt*,OFF,ON)

T_CTOR_RW macro handles the raw character terminals. The additional 2 parameters F, and G are the source file index and character position within the file. Please look at Yacco2's *yacco2_characters.lex* file to see an example of T_CTOR_RW use. The *Yacco2* runtime environment maintains an index of files included into the source grammar. FILE_TBL__ is a vector of file index and the external filename literal. FILE_CNT__ is the matching external variable used by the include file grammar that stacks them when nested include statements come into and out of scope. From the raw character classes, the GPS of the character is passed in as parameters. A specialized **tok_can** template for 'file to raw character' object mapping handles this task.

569. T_CTOR, T_CTOR_RW macros.

```
#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)
```

570. Define *YACCO2_define_trace_variables*.
See "The C++ preprocessor coding game" regarding the individual tracing variable functionality.

```
#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);
```

571. Token placement macros.
They are used by the grammar writer within syntax directed code sections of a grammar to place a token into appropriate queues:

recycle container — used to delete or re-integrate tokens back into a parse stream
accept container — tokens returned by launched threads for arbitration
producer container - tokens outputted for other parse stages
error container — a container of accrued error tokens
supplier of tokens — token stream that a grammar parses

572. `ADD_TOKEN_TO_RECYCLE_BIN`**.**
This is a holding pen for possibly re-use of the token that has been pulled out of the token stream. It is a minor facility but has poco merit.

#define `ADD_TOKEN_TO_RECYCLE_BIN`(*Token*) *rule_info__.parser__→add_token_to_recycle_bin*(*Token*)

573. `DELETE_T_SYM` **macro.**
This macro deletes a T when passed by pointer. It only allows Tes that are from either Error or Meta-terminal classes. This guards against the erasing of preallocated Tes of LR k or RC (raw chacacter) classes. They are preallocated from the memory heap for speed. It checks whether the symbol's *dtor__* static method is present and calls it. This allows a delete chain calling of other dependents or other dependencies when the *destructor* directive is used within the T grammar definition. Why this route to T symbol deletes rather than c++'s dtor: ∼*T*( )? Mixed into the fray is my `AB` abort parse stack cleanup. Whether its of any use this is my experiment. It required the stack frame pointer as the 2nd parameter. For the just plain way to deletes, this macro eases the complaints without the stack frame pointer. Depending on how your compiler/translator runs, deleting of Tes could be left to the process teardown. If your language recognizer is always on and being invoked like an Internet protocol, then T hygiene is required or those memory leaks will haunt u.

```
#define 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;
      }
    }
```

574. Add token to an accept queue: `RSVP`, `RSVP_FSM`, `RSVP_WLA` macros.
`RSVP` places a token into the calling grammar's accept queue that requested this thread. It can be placed anywhere in the syntax directed code of the grammar except within the grammar's fsm context where you use the `RSVP_FSM` macro. The `RSVP_WLA` allows u to override the lookahead bounds instead of taking as default the current token. A quick review of messages, threads, and accept tokens:

the calling grammar: 1:m launching of threads
accept queue: 0:m potential tokens returned by launched threads
1 wakeup event to calling grammar by thread finished last in execution

Arbitration is used by the caller grammar when it is re-activated by an event (message) from the thread finishing last in execution order: the status message "accept parallel parse" is posted to just wake up the calling grammar regardless of the overall parse success by the launched threads. It is the *th_accepting_cnt__* that determines whether the threads were successful or not. It only occurs when there are items in the accept queue. If none of the launched threads are successful in their parsing, then the calling grammar will attempt to go through its conditional parsing (non-determinism). Arbitration is the associated code within the grammar's fsm state that launched the threads. It rules on possibly more than one accept token being returned. A little french spices up this ho-hum macro.

Why is there an accepted token position? Remember the current token in the thread's parse stream is now the future position in the token stream to continue parsing from for the calling grammar. In a long stream of characters that makes up the accept token, usually its the start token's position passed to the called thread that is used to GPS it's position within the token stream. The current token context (I call the "lookahead context") is provided by the last 2 parameters for the **Caccept_parse**. It is this lookahead context of the accepted token that is used to continue parsing within the calling grammar. The arbitrated token is parallel shifted and its accompaning lookahead boundry is then used to reduce the parallel shift's subrule expression. All other potential accept tokens are flushed out of the accept queue.

#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__*)

575. `ADD_TOKEN_TO_PRODUCER_QUEUE`.
This allows one to output from a parse a terminal stream that becomes a supplier queue for another grammar to parse.

#define `ADD_TOKEN_TO_PRODUCER_QUEUE(TOKEN)` *rule_info__.parser__→add_token_to_producer*(`TOKEN`)

576. `ADD_TOKEN_TO_ERROR_QUEUE` **and** `ADD_TOKEN_TO_ERROR_QUEUE_FSM`.
This becomes a holding queue that can be processed by a error grammar. It's a nice way to format parsing errors. It is the context that determines which macro to use.

#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`)

577. Generated finite state automaton macros.
They are included in the C++ code of each rule emitted by Yacco2. Their names are sufficient to explain their intent. Why the wrapping of the macros within the @= ... @> construct instead of a plain macro "# **define**" definition? Glad u asked. The *cweave* version "7.5.5" on a Mac emits code that *pdftex* Version 3.141592-1.30.4-2.2 honks: too many "}″ or "$". So this is my workaround until i have time to get a higher version of cweave.

Note: the *cweb* Microsoft flavour works. More rumblings from within my quest to port Yacco2. Screw the port. i need to read it.
Brought back *cweb* macros as they work on the Mac now.

```
#define ssNEW_TRACEss(ssPss, ssQss) yacco2::lrclog << "\t!!!!! new adr: " << (void *)
        ssPss << " " << #ssQss << ' ' << __FILE__ << ':' << __LINE__ << 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 << FILE_LINE << 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__ << FILE_LINE << std::endl;
    }
    }
```

578. Pushdown automaton's flow control macros.
They are placed in stragetic places for operations accept, reduce, shift, and abort. As there are many points being traced, a little explanation is required to give some semblance of order. The messages outputted go to a log file named 'tracings.log'. What type of name is this? The prefix 1 sorts the file to the top of a directory. The balance of the name was an attempt to say lr output of clog type. Ugh. This will be changed.

Messages logged fall into the parsing configuration that tries to give a semblance of a stack. It prints the stack content in bottomup order. A sample of the trace is:

. .1500 :: *rule_def_phrase*.*lex* :: 1 −− *identifier*→3

The dots indicate the number of items on the stack to be displayed, followed by the thread's identity — a runtime thread number and the grammar's name being traced. Following this are the stacked items displayed in bottom-to-top order. Each item contains the finite state that it is in, a vector containing the stacked item and the finite state's shift into state no.

Other traces will try to output regular sentences so that they can be parsed by a grammar or a scripting language. This will allow one to digest intelligently the interplay between the grammar, and the appropriate running threads. As there are many threads simultaneously running, this will help in consolidating the reported tracings.

579. `T_0` trace remove items from the parse stack.

```
⟨ Trace TH remove items from the parse stack configuration 579 ⟩ ≡
  if (YACCO2_TH__) {
    if (fsm_tbl__→debug__ ≡ ON) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
          "::␣Popping␣items␣from␣stack␣#␣to␣pop:␣" ≪ No_to_remove ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
```

This code is used in section 361.

580. *T_0a* trace finished removing items from the parse stack.

```
⟨ Trace TH finished removing items from the parse stack configuration 580 ⟩ ≡
  if (YACCO2_TH__) {
    if (fsm_tbl__→debug__ ≡ ON) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
          "::␣Finished␣Popping␣items␣from␣stack" ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
```

This code is used in section 361.

581. `T_1` trace the parse stack if the grammar is requesting to be debugged. The returned debug switch's value is dropped.

```
⟨ Trace TH the parse stack configuration 581 ⟩ ≡
  if (YACCO2_TH__) {
    bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  }
```

This code is used in sections 236, 238, 240, 241, 245, and 348.

582. T_2 trace when an epsilon rule is being reduced.

⟨ Trace TH when an epsilon rule is being reduced 582 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪ "::epsilon" ≪
        FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 351.

583. T_3 trace the state no when popped from the parse stack.

⟨ Trace TH popped state no 583 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__;
    yacco2::lrclog ≪ "::" ≪ fsm_tbl__→id__ ≪ "::popped␣state::␣";
    yacco2::lrclog ≪ pr→state__→state_no__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 361.

584. T_4 trace when invisible shift symbol popped from stack.

Because this symbol is universal, ?? chk why zeroed instead of not having AD on?

⟨ Trace TH zeroed out symbol situation when popped from parse stack 584 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
        "::exposed␣rule/terminal::␣NULL␣due␣to␣invisible␣shift" ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 357.

585. T_5 trace exposed symbol on parse stack.

```
⟨ Trace TH exposed symbol on parse stack 585 ⟩ ≡
  if (YACCO2_TH__) {
    bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
    if (to_trace_or_not_to ≡ YES) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
          "::exposed␣rule/terminal::␣" ≪ parse_stack__.top__→symbol__→id__ ≪ '␣' ≪
          parse_stack__.top__→symbol__ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
```

This code is used in section 361.

586. T_6 trace top item on the parse stack when auto-delete switch on.
This is the grammatical attribute AD requesting deletion when popped from the parse stack.

```
⟨ Trace TH advise when symbol deleted due to AD switch 586 ⟩ ≡
  if (YACCO2_TH__) {
    bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
    if (to_trace_or_not_to ≡ YES) {
      if (parse_stack__.top__→symbol__→auto_delete__ ≡ YES) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
            "::AD␣deleting␣exposed␣rule/terminal::␣" ≪ parse_stack__.top__→symbol__→id__ ≪ '␣' ≪
            parse_stack__.top__→symbol__ ≪ FILE_LINE ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
    }
  }
```

This code is used in section 358.

587. *T_6a* trace top item on the parse stack when auto-abort switch on.
This occurs usually at abort time of a threaded parse. It can occur when the grammar writer has not removed the appropriate objects from being checked by a destructor directive for deletion in a grammar rule.

```
⟨ Trace TH advise when auto abort happening 587 ⟩ ≡
  if (YACCO2_TH__) {
    bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
    if (to_trace_or_not_to ≡ YES) {
      if (parse_stack__.top__→symbol__→affected_by_abort__ ≡ YES) {
        ⟨ acquire trace mu 389 ⟩;
        yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
            "::AB␣deleting␣exposed␣rule/terminal::␣" ≪ parse_stack__.top__→symbol__→id__ ≪ '␣' ≪
            parse_stack__.top__→symbol__ ≪ FILE_LINE ≪ std::endl;
        ⟨ release trace mu 390 ⟩;
      }
    }
  }
```

This code is used in section 361.

588. T_7 trace when threading failed: try straight parse.

⟨ Trace TH failed parallel try straight parse 588 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
        "::#############␣TRY␣STRAIGHT␣try_straight_due_to_aborted_parallel" ≪
        "␣reset␣token␣pos:␣" ≪ current_token_pos__ ≪ "␣reset␣token:␣" ≪ current_token__→id__ ≪
        FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RESET␣FILE:␣";
    EXTERNAL_GPSing(current_token())yacco2::lrclog ≪ "␣GPS␣RESET␣LINE:␣" ≪
        current_token()→tok_co_ords__.line_no__ ≪ "␣GPS␣RESET␣CHR␣POS:␣" ≪
        current_token()→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 258.

589. T_7 trace when proc call failed: try straight parse.

⟨ Trace TH failed proc call try straight parse 589 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
        "::#############␣TRY␣STRAIGHT␣try_straight_due_to_aborted_parallel" ≪
        "␣reset␣token␣pos:␣" ≪ current_token_pos__ ≪ "␣reset␣token:␣" ≪ current_token__→id__ ≪
        FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RESET␣FILE:␣";
    EXTERNAL_GPSing(current_token())yacco2::lrclog ≪ "␣GPS␣RESET␣LINE:␣" ≪
        current_token()→tok_co_ords__.line_no__ ≪ "␣GPS␣RESET␣CHR␣POS:␣" ≪
        current_token()→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 262.

590. T_11 straight parse error.
How and why NIL pointer? protects when the

⟨ Trace TH straight parse error 590 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪ "::";
    yacco2::lrclog ≪ "############straight␣parse-error␣current␣token␣" ≪ '"' ≪
        current_token()→id__ ≪ '"' ≪ "␣pos:␣" ≪ current_token_pos__ ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RESET␣FILE:␣";
    EXTERNAL_GPSing(current_token())yacco2::lrclog ≪ "␣GPS␣RESET␣LINE:␣" ≪
        current_token()→tok_co_ords__.line_no__ ≪ "␣GPS␣RESET␣CHR␣POS:␣" ≪
        current_token()→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 249.

591. T_14 trace parallel parse thread startup communication.

⟨ Trace TH parallel parse thread start communication 591 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_TH__::" ≪ "requestor␣of␣parallelism*␣:␣" ≪
        "␣pp␣id:␣" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪ "␣parallel␣PP␣started:␣" ≪
        pe→thread_fnct_name__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in section 384.

592. T_17 trace accepted token info.

⟨ Trace TH accepted token info 592 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::||||||||||ACCEPTED␣token␣POS:␣" ≪
        arbitrated_token__→accept_token_pos__ ≪ "␣token*:␣" ≪ arbitrated_token__→accept_token__ ≪
        "␣token:␣" ≪ arbitrated_token__→accept_token__→id__ ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣FILE:␣";
    EXTERNAL_GPSing(arbitrated_token__→accept_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
        arbitrated_token__→accept_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
        arbitrated_token__→accept_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪
        "::||||||||||ACCEPTED␣la␣token␣POS:␣" ≪ arbitrated_token__→la_token_pos__ ≪
        "␣la␣token:␣" ≪ arbitrated_token__→la_token__→id__ ≪ std::endl;
    yacco2::lrclog ≪ "\t" ≪ thread_no__ ≪ "::GPS␣LA␣FILE:␣";
    EXTERNAL_GPSing(arbitrated_token__→la_token__)yacco2::lrclog ≪ "␣GPS␣LA␣LINE:␣" ≪
        arbitrated_token__→la_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣LA␣CHR␣POS:␣" ≪
        arbitrated_token__→la_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in sections 418, 421, and 422.

593. Trace re-aligned token stream la boundry info.

⟨ Trace TH re-aligned token stream la boundry info 593 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪
        "::|||re-aligned␣token␣stream␣la␣boundry␣POS:␣" ≪ current_token_pos__ ≪
        "␣la␣token:␣" ≪ current_token__→id__ ≪ FILE_LINE ≪ std::endl;
    yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RE-ALIGN␣FILE:␣";
    EXTERNAL_GPSing(current_token__)yacco2::lrclog ≪ "␣GPS␣RE-ALIGN␣␣LINE:␣" ≪
        current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣RE-ALIGN␣CHR␣POS:␣" ≪
        current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
}
```

This code is used in sections 418 and 421.

594. T_18 trace requesting grammar's received message from a thread.

⟨Trace TH request thread received message from parallel thread 594⟩ ≡
```
if (YACCO2_TH__) {
  if (no_requested_ths_to_run__ > 1) {
    bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
    if (to_trace_or_not_to ≡ YES) {
      ⟨acquire trace mu 389⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪ "::" ≪
          "parallel␣parsing␣received␣message␣from␣id:" ≪ from_thread__→thread_no__ ≪ "::" ≪
          from_thread__→thread_name( ) ≪ FILE_LINE ≪ std::endl;
      ⟨release trace mu 390⟩;
    }
  }
}
```
This code is used in sections 418, 421, and 422.

595. T_22 and *T_22a* trace returned thread accept info.

⟨Trace TH current token, and accepted terminal wrapper 595⟩ ≡
```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    if (current_token( ) ≠ 0) {
      ⟨acquire trace mu 389⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::";
      yacco2::lrclog ≪ fsm_tbl__→id__ ≪ "::";
      yacco2::lrclog ≪ "accept-parallel-parse␣current␣token␣" ≪ '"' ≪ current_token( )→id__ ≪
          '"' ≪ "␣pos:␣" ≪ current_token_pos__ ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ "␣accept␣tok:␣" ≪ pp_rsvp__.accept_token__→id__ ≪
          "␣tok␣pos:␣" ≪ pp_rsvp__.accept_token_pos__ ≪ "␣la␣tok:␣" ≪ pp_rsvp__.la_token__→id__ ≪
          "␣la␣tok␣pos:␣" ≪ pp_rsvp__.la_token_pos__ ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "␣thru␣fsm->␣parser*:␣" ≪ fsm_tbl__→parser( ) ≪ std::endl;
      yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣ACCEPT␣FILE:␣";
      EXTERNAL_GPSing(pp_rsvp__.accept_token__)yacco2::lrclog ≪ "␣GPS␣ACCEPT␣␣LINE:␣" ≪
          pp_rsvp__.accept_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣ACCEPT␣␣CHR␣POS:␣" ≪
          pp_rsvp__.accept_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣ACCEPT␣LA␣FILE:␣";
      EXTERNAL_GPSing(pp_rsvp__.la_token__)yacco2::lrclog ≪ "␣GPS␣ACCEPT␣␣LA␣LINE:␣" ≪
          pp_rsvp__.la_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣ACCEPT␣LA␣CHR␣POS:␣" ≪
          pp_rsvp__.la_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      ⟨release trace mu 390⟩;
    }
  }
}
```
This code is used in sections 272 and 282.

596. T_23 **trace parallel parse current token when an error has occured.**

⟨ Trace TH parallel parse current token when an error has occured 596 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    if (current_token()) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::";
      yacco2::lrclog ≪ fsm_tbl__→id__ ≪ "::";
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ "#############parallel␣parse-error␣curre\
          nt␣token␣" ≪ current_token()→id__ ≪ "␣pos:␣" ≪ current_token_pos__ ≪ "␣enum␣id:␣" ≪
          current_token()→enumerated_id__ ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RESET␣FILE:␣";
      EXTERNAL_GPSing(current_token())yacco2::lrclog ≪ "␣GPS␣RESET␣LINE:␣" ≪
          current_token()→tok_co_ords__.line_no__ ≪ "␣GPS␣RESET␣CHR␣POS:␣" ≪
          current_token()→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
}
```

This code is used in section 279.

597. T_23 **trace proc call parse current token when an error has occured.**

⟨ Trace TH proc call parse current token when an error has occured 597 ⟩ ≡

```
if (YACCO2_TH__) {
  bool to_trace_or_not_to = trace_parser_env(this, COND_STK_TRACE);
  if (to_trace_or_not_to ≡ YES) {
    if (current_token()) {
      ⟨ acquire trace mu 389 ⟩;
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ thread_no__ ≪ "::";
      yacco2::lrclog ≪ fsm_tbl__→id__ ≪ "::";
      yacco2::lrclog ≪ "YACCO2_TH__::" ≪ "#############parallel␣parse-error␣curre\
          nt␣token␣" ≪ current_token()→id__ ≪ "␣pos:␣" ≪ current_token_pos__ ≪ "␣enum␣id:␣" ≪
          current_token()→enumerated_id__ ≪ FILE_LINE ≪ std::endl;
      yacco2::lrclog ≪ "\tYACCO2_TH__::" ≪ thread_no__ ≪ "::GPS␣RESET␣FILE:␣";
      EXTERNAL_GPSing(current_token())yacco2::lrclog ≪ "␣GPS␣RESET␣LINE:␣" ≪
          current_token()→tok_co_ords__.line_no__ ≪ "␣GPS␣RESET␣CHR␣POS:␣" ≪
          current_token()→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
      ⟨ release trace mu 390 ⟩;
    }
  }
}
```

This code is used in section 283.

598. T_24 trace before parallel parse thread message count reduced. This allows one to see if threading mutexes etc are behaving.

⟨Trace TH before parallel parse thread message count reduced 598⟩ ≡
```
if (YACCO2_MSG__) {
   ⟨acquire trace mu 389⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::";
   yacco2::lrclog << fsm_tbl__→id__ << "::";
   yacco2::lrclog << "␣called␣thread␣reducing␣thread␣active␣count␣of␣caller␣thread␣" <<
       pp_requesting_parallelism__→thread_no__ << "::" << pp_requesting_parallelism__→fsm_tbl__→id__ <<
       "␣active␣thread␣count::" << pp_requesting_parallelism__→th_active_cnt__ << FILE_LINE <<
       std::endl;
   ⟨release trace mu 390⟩;
}
```
This code is used in section 280.

599. T_25 trace parallel parse current token when an error has occured.

⟨Trace TH after parallel parse thread message count reduced 599⟩ ≡
```
if (YACCO2_MSG__) {
   ⟨acquire trace mu 389⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::";
   yacco2::lrclog << fsm_tbl__→id__ << "::";
   yacco2::lrclog << "␣called␣thread␣after␣reducing␣thread␣active␣count␣of␣caller␣thread␣" <<
       pp_requesting_parallelism__→thread_no__ << "::" << pp_requesting_parallelism__→fsm_tbl__→id__ <<
       "␣active␣thread␣count::" << pp_requesting_parallelism__→th_active_cnt__ << FILE_LINE <<
       std::endl;
   ⟨release trace mu 390⟩;
}
```
This code is used in section 280.

600. Message macros.

They trace the correspondence between various threads. Here are the thread relationships:

grammar calling its spawned threads

launched threads to the grammar requesting thread service

These macros are very verbous but it allows one to analyse whether messages have been dropped. Typically dropped messages come about when an event depends on a specific result and the order of execution within the threads can change the current terminal mapping such that executing produces possibly an aborted thread parse. For example when a terminal is fetched with dynamic symbol table evaluation taking place, depending on the sequence execution of the threads errant terminal delivery can occur. This is a critical region problem between the competing threads. To fix the problem, either eliminate the competition of threads between themselves by blending into one thread the competing grammatical sentences, or use a MUTEX to tame the eradic behavior.

To control messaging back to the requesting grammar when all threads have finished processing, an activity thread count under the control of its MUTEX is referenced by each launched thread. The responsibility of who responds back to the grammar requesting parallelism when all threads are done be it success or failure, is left to the individual threads launched. When a thread finishes work, it goes into the critical region of the requesting grammar and decrements the active thread count. It also checks if the activity count is zero indicating that it is the last thread in the house to lock up so wake up the requesting grammar.

601. TT_1 trace thread waiting for message.

⟨ Trace MSG thread waiting for message 601 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
   ⟨ acquire trace mu 389 ⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::" << thread_name() <<
      "␣-->WAIT_FOR_EVENT␣" << "␣#␣threads␣to␣run::␣" << no_requested_ths_to_run__ <<
      "␣#␣active␣threads:␣" << th_active_cnt__ << "␣#␣competing␣threads:␣" <<
      no_competing_pp_ths__ << FILE_LINE << std::endl;
   ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 393.

602. TT_2 trace message received.

Protect against procedure call that has wound down and destroyed itself before the calling grammar can trace it. Only trace returned call from threads.

⟨ Trace MSG message received 602 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
   ⟨ acquire trace mu 389 ⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::" << thread_name() <<
      "␣MESSAGE␣RECEIVED␣from␣" << from_thread__→thread_no__ << "::" <<
      from_thread__→thread_name() << FILE_LINE << std::endl;
   ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 393.

603. TT_4 trace posting from - to thread info.

⟨ Trace posting from - to thread info 603 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
   ⟨ acquire trace mu 389 ⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << From_thread.thread_no__ << "::" <<
      From_thread.thread_name() << "␣----->␣Posting␣message␣to:␣" << To_thread.thread_no__ <<
      "::" << To_thread.thread_name() << FILE_LINE << std::endl;
   ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 396.

604. *TT_4a* trace signaled grammar to wakeup.

⟨ Trace signaled grammar to wakeup while releasing its mutex 604 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
   ⟨ acquire trace mu 389 ⟩;
   yacco2::lrclog << "YACCO2_MSG__::" << From_thread.thread_no__ << "::" <<
      From_thread.thread_name() << "␣----->␣before␣SIGNAL_COND_VAR()␣to␣sig\
      nal␣wakeup␣grammar␣for:␣" << To_thread.thread_no__ << "::" << To_thread.thread_name() <<
      FILE_LINE << std::endl;
   ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 397.

605. *TT_4b* **trace wakened grammar with its acquired mutex.**

```
⟨ Trace wakened grammar with its acquired mutex 605 ⟩ ≡
  if (yacco2::YACCO2_MSG__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ From_thread.thread_no__ ≪ "::" ≪
        From_thread.thread_name() ≪ "␣----->␣after␣SIGNAL_COND_VAR()␣to␣waken␣grammar␣of␣" ≪
        To_thread.thread_no__ ≪ "::" ≪ To_thread.thread_name() ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in section 397.

606. *TT_4c* **trace trying to acquire grammar's mutex.**

```
⟨ Trace trying to acquire grammar's mutex 606 ⟩ ≡
  if (yacco2::YACCO2_MU_GRAMMAR__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MU_GRAMMAR__::" ≪ parser.thread_no__ ≪ "::" ≪ parser.fsm_tbl__→id__ ≪
        "::" ≪ "␣trying␣to␣acquire␣mutex" ≪ Text ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in sections 145 and 158.

607. *TT_4d* **trace acquired grammar's mutex.**

```
⟨ Trace acquired grammar's mutex 607 ⟩ ≡
  if (yacco2::YACCO2_MU_GRAMMAR__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MU_GRAMMAR__::" ≪ parser.thread_no__ ≪ "::" ≪ parser.fsm_tbl__→id__ ≪
        "::" ≪ "␣acquired␣mutex" ≪ Text ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in sections 145 and 158.

608. *TT_4e* **trace trying to release grammar's mutex.**

```
⟨ Trace trying to release grammar's mutex 608 ⟩ ≡
  if (yacco2::YACCO2_MU_GRAMMAR__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MU_GRAMMAR__::" ≪ parser.thread_no__ ≪ "::" ≪ parser.fsm_tbl__→id__ ≪
        "::" ≪ "␣trying␣to␣release␣mutex" ≪ Text ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in sections 147 and 160.

609. *TT_4f* **trace released grammar's mutex.**

```
⟨ Trace released grammar's mutex 609 ⟩ ≡
  if (yacco2::YACCO2_MU_GRAMMAR__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MU_GRAMMAR__::" ≪ parser.thread_no__ ≪ "::" ≪ parser.fsm_tbl__→id__ ≪
        "::" ≪ "␣released␣mutex" ≪ Text ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in sections 147 and 160.

610. TT_5 trace start thread.

⟨ Trace MSG start thread 610 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪ "::" ≪ fsm_tbl__→id__ ≪
      "␣-->␣start␣threads" ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 385.

611. TT_6 trace of found thread in thread pool waiting to be run. The pool of threads is dynmically built by thread requests. When a thread finishes work, instead of stopping, it goes into hibernation with an appropriate status indicating its availability. This is an optimization to speed up parallel processing. There can be many threads of the same name waiting for work due to recursion.

⟨ Trace MSG found thread in thread pool waiting to be run 611 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪
      "::-->␣parallel␣thread␣worker␣fnd␣in␣thread␣table␣CALL␣WORKER:␣" ≪
      tb→grammar_s_parser__→thread_name() ≪ "␣thread␣obj*:" ≪ tb→grammar_s_parser__ ≪
      "␣parm*:␣" ≪ tb→grammar_s_parser__→pp_requesting_parallelism__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 383.

612. TT_7 due to recursion trace no thread available in thread pool. This comes about when a thread calls a thread who calls a previous thread. I call this situation "nested parallelism". The grammar of Pascal's railroad diagrams is an example of such situations. It is detected due to the thread (thread id number) already has an entry in the thread pool but there are no available threads ready to run so... launch another thread.

⟨ Trace MSG thread fnd but all busy, so launch another one 612 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪ "::" ≪ pe→thread_fnct_name__ ≪
      "␣-->␣parallel␣thread␣fnd␣in␣thread␣table␣BUT␣ALL␣ARE␣BUSY␣" ≪ FILE_LINE ≪
      std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 383.

613. TT_8 trace thread not found in global thread pool.

⟨ Trace MSG thread not found in global thread pool 613 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪ "::" ≪ pe→thread_fnct_name__ ≪
      "␣-->␣parallel␣thread␣NOT␣fnd␣in␣thread␣table" ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 383.

614. Trace start thread by procedure call instead of threading.

⟨ Trace MSG start by procedure call 614 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪ "␣calling␣PROC::"
     /* ≪ pe→thread_fnct_name__ */
   ≪ "␣-->␣before␣procedure␣call" ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 375 and 384.

615. Trace return from procedure call instead of threading.

⟨ Trace MSG return from by procedure call 615 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ thread_no__ ≪ "::" ≪ thread_name() ≪
      "␣returned␣from␣PROC::"    /* ≪ pe→thread_fnct_name__ */
   ≪ "␣result:␣" ≪ rslt ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 375 and 384.

616. Trace thread idle before setting waiting for work.

⟨ Trace MSG thread idle before setting waiting for work 616 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ this→grammar_s_parser__→thread_no__ ≪ "::" ≪
      this→grammar_s_parser__→thread_name() ≪ "␣th_blk*:␣" ≪ this ≪ "this␣pp*:␣" ≪
      this→grammar_s_parser__ ≪ "pp*:␣" ≪ grammar_s_parser__ ≪ "pp^th␣blk*:␣" ≪
      &grammar_s_parser__→th_blk__ ≪ "␣#:␣" ≪ thd_id__ ≪ "␣st:␣" ≪ status__ ≪
      "␣before␣setting␣waiting␣for␣work" ≪ '␣' ≪ grammar_s_parser__→thread_no__ ≪ "::" ≪
      grammar_s_parser__→fsm_tbl__→id__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 179.

617. Trace thread idle after setting waiting for work.

⟨ Trace MSG thread idle after setting waiting for work 617 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ this→grammar_s_parser__→thread_no__ ≪ "::" ≪
      this→grammar_s_parser__→thread_name() ≪ "␣th_blk*:␣" ≪ this ≪ "this␣pp*:␣" ≪
      this→grammar_s_parser__ ≪ "pp*:␣" ≪ grammar_s_parser__ ≪ "pp^th␣blk*:␣" ≪
      &grammar_s_parser__→th_blk__ ≪ "␣#:␣" ≪ thd_id__ ≪ "␣st:␣" ≪ status__ ≪
      "␣after␣setting␣waiting␣for␣work␣" ≪ grammar_s_parser__→thread_no__ ≪ "::" ≪
      grammar_s_parser__→fsm_tbl__→id__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 179.

618. Trace thread being created.

⟨ Trace MSG thread being created 618 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << this→grammar_s_parser__→thread_no__ << "::" <<
      this→grammar_s_parser__→thread_name( ) << "␣th_blk*:␣" << this << "␣pp␣this:␣" <<
      this→grammar_s_parser__ << "␣this^pp^th_blk:␣" << &this→grammar_s_parser__→th_blk__ <<
      "pp*:␣" << grammar_s_parser__ << "pp^th␣blk*:␣" << &grammar_s_parser__→th_blk__ <<
      "␣#:␣" << thd_id__ << "␣thread␣created␣" << grammar_s_parser__→thread_no__ << "::" <<
      grammar_s_parser__→thread_name( ) << "␣of␣grammar:␣" << grammar_s_parser__→fsm_tbl__→id__ <<
      FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 178.

619. Trace threads in launched list.

⟨ Trace threads in launched list 619 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << tb→grammar_s_parser__→thread_no__ << "::" <<
      tb→grammar_s_parser__→thread_name( ) << "␣th_blk*:␣" << this << "␣th_blk*:␣" << tb <<
      "␣grammar␣parser:␣" << tb→grammar_s_parser__ << "␣#:␣" << tb→thd_id__ << "␣st:␣" <<
      tb→status__ << "␣thds␣in␣launched␣list␣" << FILE_LINE << std::endl;
  yacco2::lrclog << "------->" << tb→grammar_s_parser__→thread_no__ << "::" <<
      tb→grammar_s_parser__→fsm_tbl__→id__ << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 382.

620. Trace thread to be launched.

⟨ Trace thread to be launched 620 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << "pe*:␣" << pe << "␣thread␣name:␣" <<
      pe→thread_fnct_name__ << "␣thread␣proc*:␣" << pe→thread_fnct_ptr__ << "␣thread␣id:␣" <<
      pe→thd_id__ << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 384.

621. All threads reported back.

⟨ Trace MSG all threads reported back 621 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::" << thread_name() <<
      "␣all␣threads␣reported␣back␣" << "␣Caller␣parser::" <<
      pp_requesting_parallelism__→thread_no__ << "::" << pp_requesting_parallelism__→thread_name() <<
      "␣#␣competing␣thds:␣" << pp_requesting_parallelism__→no_requested_ths_to_run__ <<
      "␣#␣active␣thds:␣" << pp_requesting_parallelism__→th_active_cnt__ << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 277.

622. NOT all threads reported back.

⟨ Trace MSG not all threads reported back 622 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::" << thread_name() <<
      "␣NOT␣all␣threads␣reported␣back␣" << "␣Caller␣parser::" <<
      pp_requesting_parallelism__→thread_no__ << "::" << pp_requesting_parallelism__→thread_name() <<
      "␣#␣competing␣thds:␣" << pp_requesting_parallelism__→no_requested_ths_to_run__ <<
      "␣#␣active␣thds:␣" << pp_requesting_parallelism__→th_active_cnt__ << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 277.

623. Call procedure but in use.

⟨ Trace MSG proc call in use so call its thread 623 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_MSG__::" << thread_no__ << "::" << thread_name() <<
      "␣PROC␣CALL␣ALREADY␣IN␣USE␣so␣call␣its␣thread␣" << "␣Caller␣parser::" <<
      pp_requesting_parallelism__→thread_no__ << "::" << pp_requesting_parallelism__→thread_name() <<
      FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 384.

624. Arbitrator macros.
These are the syntax directed code directives within a grammar's rules that arbitrate between the returned results of that finite state's configuration. They are gened as individual procedures per finite state configuration due to parallelism. To refine this family of message traces, they test whether their grammar has the debug switch turned on.

625. `TAR_1` trace the starting of arbitration.

⟨ Trace AR trace the starting of arbitration 625 ⟩ ≡
  ⟨ pp accept queue AR 626 ⟩;

This code is used in section 189.

626.

⟨ pp accept queue AR 626 ⟩ ≡
```
if (yacco2::YACCO2_AR__) {      /* trace_parser_env(Caller_pp, FORCE_STK_TRACE); */
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_AR__::" << Caller_pp->thread_no__ << "::" << ar_name <<
      " start arbitrating" << FILE_LINE << std::endl;
  yacco2::lrclog << "YACCO2_AR__::" << "number of accept tokens: " <<
      Caller_pp->th_accepting_cnt__ << FILE_LINE << std::endl;
  int ii = 1;
  for ( ; ii <= Caller_pp->th_accepting_cnt__; ++ii) {
    yacco2::lrclog << "\t terminal in accept queue: " <<
        Caller_pp->pp_accept_queue__[ii].accept_token__->id__ << " token pos: " <<
        Caller_pp->pp_accept_queue__[ii].accept_token_pos__ << FILE_LINE << std::endl;
  }
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 625.

627. `TAR_2` trace no arbitration required.
This occurs when only 1 accept terminal is in the accept queue

⟨ Trace AR no arbitration required 627 ⟩ ≡
  ⟨ trace AR pp accept queue no arbitration required 628 ⟩;

628. ⟨ trace AR pp accept queue no arbitration required 628 ⟩ ≡
```
if (yacco2::YACCO2_AR__) {      /* trace_parser_env(Caller_pp, FORCE_STK_TRACE); */
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_AR__::" << Caller_pp->thread_no__ << "::" << ar_name <<
      " No Arbitration required" << FILE_LINE << std::endl;
  yacco2::lrclog << "YACCO2_AR__::" << "number of accept tokens: " <<
      Caller_pp->th_accepting_cnt__ << FILE_LINE << std::endl;
  int ii = 1;
  for ( ; ii <= Caller_pp->th_accepting_cnt__; ++ii) {
    yacco2::lrclog << "\t YACCO2_AR__:: terminal in accept queue: " <<
        Caller_pp->pp_accept_queue__[ii].accept_token__->id__ << " token pos: " <<
        Caller_pp->pp_accept_queue__[ii].accept_token_pos__ << FILE_LINE << std::endl;
  }
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 627.

629. TAR_3 trace stopped arbitrating.
This occurs when only 1 accept terminal is in the accept queue

⟨ Trace AR stopped arbitrating 629 ⟩ ≡
```
if (yacco2::YACCO2_AR__) {
    /* bool to_trace_or_no_to = trace_parser_env(Caller_pp, FORCE_STK_TRACE); */
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_AR__::" << Caller_pp->thread_no__ << "::" << ar_name <<
      " stop arbitrating" << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 192.

630. TLEX macros — roll-your-own tracing macros.
These are "roll your own" macros for when the going get 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 probably 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 *cout* are rudimentary but also effective.

⟨c macros 13⟩ +≡
```
#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 << FILE_LINE <<
        std::endl;
    yacco2::lrclog << "\tself thread id:: " << thread_no__ << FILE_LINE << std::endl;
    yacco2::lrclog << "\tYACCO2_TLEX__:: embedded thread id:: " << embedded_thread_no() <<
        FILE_LINE << std::endl;
  }
```

631. Print interplay between requesting grammar and launched thread. A roll your own descriptor is passed to the macro.

⟨c macros 13⟩ +≡
```
#define sstrace_parallel_support_envss(ssPROC_NAME)
  if (yacco2::YACCO2_TLEX__) {
    yacco2::lrclog << "YACCO2_TLEX__::" << GetCurrentThreadid__ << "::" << fsm_tbl__->id__ << "::" <<
        ' ' << #ssPROC_NAME << " this:: " << this << FILE_LINE << std::endl;
    yacco2::lrclog << "\tYACCO2_TLEX__:: self thread id:: " << thread_no__ << FILE_LINE <<
        std::endl;
  }
```

632. ***trace_parser_env*** **traces the parsing stack.**
It check whether the thread has its debug switch on. If it does, it does its own thing. It returns the thread's debug grammar switch for other trace macros to test whether they should do their trace dance.

⟨External rtns and variables 22⟩ +≡
```
  extern bool trace_parser_env(Parser *parser, bool Trace_type);
```

633. Print parse stack prefix.

⟨Print parse stack prefix 633⟩ ≡
```
  ⟨acquire trace mu 389⟩;
  for (UINT x = 1; x <= parser->parse_stack__.top_sub__; ++x) yacco2::lrclog << ".";
  yacco2::lrclog << parser->thread_no__;
  yacco2::lrclog << "::";
  yacco2::lrclog << parser->fsm_tbl__->id__ << "::";
  ⟨release trace mu 390⟩;
```
This code is used in section 636.

634. Print items on parse stack in FILO order.

⟨ Print items on parse stack 634 ⟩ ≡
```
⟨ acquire trace mu 389 ⟩;
Cparse_record *i = parser→parse_stack__.first_sf__;
Cparse_record *ie = parser→parse_stack__.top__;
for (int xxx(1); i ≠ ie; i = parser→parse_stack__.sf_by_sub(++xxx)) {
  yacco2::lrclog ≪ i→state__→state_no__ ≪ "--";
  CAbs_lr1_sym *sym = i→symbol__;
  if (sym) yacco2::lrclog ≪ sym→id__ ≪ "->␣";
  else yacco2::lrclog ≪ "ZEROED␣OUT␣SYMBOL" ≪ "->␣";
}
yacco2::lrclog ≪ ie→state__→state_no__;
yacco2::lrclog ≪ FILE_LINE ≪ std::endl;
⟨ release trace mu 390 ⟩;
```
This code is used in section 636.

635. Should grammar be traced?.
The debug switch supplied by the grammar is checked. If it's turned on then allow tracing. This check lowers the volume outputted. It's a spot check on 'what the hell is going wrong' with my grammar.

⟨ Should grammar be traced? no ta ta 635 ⟩ ≡
```
if (Trace_type ≡ COND_STK_TRACE) {
  if (parser→fsm_tbl__→debug__ ≡ OFF) return NO;
}
```
This code is used in section 636.

636. *trace_parser_env* **implementation.**
There are 2 contexts that stack tracing can take place:

1) within the grammar controlled by YACCO2_TH__ trace variable
2) forced stack trace used by other trace variables

⟨ accrue yacco2 code 33 ⟩ +≡
```
extern bool yacco2::trace_parser_env(Parser *parser, bool Trace_type)
{
  ⟨ Validate parser's finite state table 550 ⟩;
  ⟨ Validate that parser stack is not empty 551 ⟩;
  ⟨ Should grammar be traced? no ta ta 635 ⟩;
  ⟨ Print parse stack prefix 633 ⟩;
  ⟨ Print items on parse stack 634 ⟩;
  return YES;
}
```

637. Trace pp start info.
This is the tabloid giving all the gory details about the parallel thread, its requesting grammar, and the starting token stream to-be-parsed.

⟨ Trace pp start info 637 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  sprintf(ma, pp_start, pp_parser.thread_no__, pp_thread_entry.thread_fnct_name__);
  yacco2::lrclog << ma;
  Parser *pp_ = pp_parser.pp_requesting_parallelism__;
  yacco2::lrclog << "YACCO2_MSG__::" << pp_parser.thread_no__ << "::" << pp_parser.thread_name() <<
      "␣requesting␣parser*:␣" << pp_ << FILE_LINE << std::endl;
  yacco2::lrclog << "\tYACCO2_MSG__::" << pp_parser.thread_no__ << "::" <<
      pp_parser.thread_name() << "␣Caller's␣#␣threads␣to␣run::␣" <<
      pp_→no_requested_ths_to_run__ << "␣Caller's␣#␣active␣threads:␣" << pp_→th_active_cnt__ <<
      "␣Self␣#␣competing␣threads:␣" << pp_parser.no_competing_pp_ths__ << FILE_LINE << std::endl;
  yacco2::lrclog << "\tYACCO2_MSG__::" << pp_parser.thread_no__ << "::" <<
      pp_parser.thread_name() << "␣passed␣token*:␣" << pp_→current_token() << '"' <<
      pp_parser.current_token()→id__ << '"' << "␣pos:␣" << pp_parser.current_token_pos__ <<
      FILE_LINE << std::endl;
  yacco2::lrclog << "\t\t::GPS␣FILE:␣";
  EXTERNAL_GPSing(pp_parser.current_token__)yacco2::lrclog << "␣GPS␣LINE:␣" <<
      pp_parser.current_token__→tok_co_ords__.line_no__ << "␣GPS␣CHR␣POS:␣" <<
      pp_parser.current_token__→tok_co_ords__.pos_in_line__ << FILE_LINE << std::endl;
  ⟨ release trace mu 390 ⟩;
}
if (yacco2::YACCO2_T__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog << "YACCO2_T__::" << pp_parser.thread_no__ << "::" <<
      pp_parser.thread_name() << "␣token*:␣" << pp_parser.current_token__ << "␣enum:␣" <<
      pp_parser.current_token__→enumerated_id__ << "␣pos:␣" << pp_parser.current_token_pos__ << '␣' <<
      '"' << pp_parser.current_token__→id__ << '"' << FILE_LINE << std::endl;
  yacco2::lrclog << "\t\t::GPS␣FILE:␣";
  EXTERNAL_GPSing(pp_parser.current_token__)yacco2::lrclog << "␣GPS␣LINE:␣" <<
      pp_parser.current_token__→tok_co_ords__.line_no__ << "␣GPS␣CHR␣POS:␣" <<
      pp_parser.current_token__→tok_co_ords__.pos_in_line__ << FILE_LINE << std::endl;
  yacco2::lrclog << "\tGPS␣LINE:␣" << pp_parser.current_token()→tok_co_ords__.line_no__ <<
      "␣GPS␣CHR␣POS:␣" << pp_parser.current_token()→tok_co_ords__.pos_in_line__ << FILE_LINE <<
      std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 193.

638. Trace procedure pp start info.
This is the tabloid giving all the gory details about the parallel thread, its requesting grammar, and the starting token stream to-be-parsed.

⟨ Trace procedure pp start info 638 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  sprintf(ma, pp_start, proc_parser→thread_no__, called_proc_name);
  yacco2::lrclog ≪ ma;
  Parser *pp_ = proc_parser→pp_requesting_parallelism__;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ proc_parser→thread_no__ ≪ "::" ≪
      proc_parser→thread_name() ≪ "␣requesting␣parser*:␣" ≪ pp_ ≪ FILE_LINE ≪ std::endl;
  yacco2::lrclog ≪ "\tYACCO2_MSG__::PROC::" ≪ proc_parser→thread_no__ ≪
      "::" ≪ proc_parser→thread_name() ≪ "␣Caller's␣#␣threads␣to␣run::␣" ≪
      pp_→no_requested_ths_to_run__ ≪ "␣Caller's␣#␣active␣threads:␣" ≪ pp_→th_active_cnt__ ≪
      "␣Self␣#␣competing␣threads:␣" ≪ proc_parser→no_competing_pp_ths__ ≪ FILE_LINE ≪
      std::endl;
  yacco2::lrclog ≪ "\tYACCO2_MSG__::PROC::" ≪ proc_parser→thread_no__ ≪ "::" ≪
      proc_parser→thread_name() ≪ "␣passed␣token*:␣" ≪ pp_→current_token() ≪ '"' ≪
      proc_parser→current_token()→id__ ≪ '"' ≪ "␣pos:␣" ≪ proc_parser→current_token_pos__ ≪
      FILE_LINE ≪ std::endl;
  yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
  EXTERNAL_GPSing(proc_parser→current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
      proc_parser→current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
      proc_parser→current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
if (yacco2::YACCO2_T__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_T__::" ≪ proc_parser→thread_no__ ≪ "::" ≪
      proc_parser→thread_name() ≪ "␣token*:␣" ≪ proc_parser→current_token__ ≪ "␣enum:␣" ≪
      proc_parser→current_token__→enumerated_id__ ≪ "␣pos:␣" ≪ proc_parser→current_token_pos__ ≪
      '␣' ≪ '"' ≪ proc_parser→current_token__→id__ ≪ '"' ≪ FILE_LINE ≪ std::endl;
  yacco2::lrclog ≪ "\t\t::GPS␣FILE:␣";
  EXTERNAL_GPSing(proc_parser→current_token__)yacco2::lrclog ≪ "␣GPS␣LINE:␣" ≪
      proc_parser→current_token__→tok_co_ords__.line_no__ ≪ "␣GPS␣CHR␣POS:␣" ≪
      proc_parser→current_token__→tok_co_ords__.pos_in_line__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 203.

639. Trace stop of parallel parse message.

⟨ Trace stop of parallel parse message 639 ⟩ ≡
```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  sprintf(ma, pp_stop, pp_parser.thread_no__, pp_thread_entry.thread_fnct_name__);
  yacco2::lrclog ≪ ma;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in section 193.

640. Trace pp's last symbol on stack set as autodelete.

⟨ Trace pp's last symbol on stack set as autodelete 640 ⟩ ≡

```
if (yacco2::YACCO2_TH__) {
  THREAD_NO tid = pp_parser.thread_no__;
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_TH__::␣" ≪ "sym␣to␣be␣deleted:␣" ≪ tid ≪ "::" ≪
      pp_parser.fsm_tbl__→id__ ≪ "::" ≪ sym→id__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 197.

641. Trace procedure pp's last symbol on stack set as autodelete.

⟨ Trace procedure pp's last symbol on stack set as autodelete 641 ⟩ ≡

```
if (yacco2::YACCO2_TH__) {
  THREAD_NO tid = proc_parser→thread_no__;
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_TH__::␣" ≪ "sym␣to␣be␣deleted:␣" ≪ tid ≪ "::" ≪
      proc_parser→fsm_tbl__→id__ ≪ "::" ≪ sym→id__ ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 206.

642. Trace parallel thread waiting-to-do-work.

⟨ Trace parallel thread waiting-to-do-work 642 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ pp_parser.thread_no__ ≪ "::" ≪
      pp_thread_entry.thread_fnct_name__ ≪ "␣==>PP␣waiting␣for␣work:␣" ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 193.

643. Trace pp received go start working message.

⟨ Trace pp received go start working message 643 ⟩ ≡

```
if (yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ pp_parser.thread_no__ ≪ "::" ≪
      pp_thread_entry.thread_fnct_name__ ≪ "␣==>PP␣go␣process␣work:␣" ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```

This code is used in section 193.

644. Trace pp finished working.

```
⟨ Trace pp finished working 644 ⟩ ≡
  if (yacco2::YACCO2_MSG__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MSG__::" ≪ pp_parser.thread_no__ ≪ "::" ≪
        pp_thread_entry.thread_fnct_name__ ≪ "␣==>PP␣finished␣working" ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in section 193.

645. Trace procedure pp finished working.

```
⟨ Trace procedure pp finished working 645 ⟩ ≡
  if (yacco2::YACCO2_MSG__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_MSG__::PROC::" ≪ proc_parser→thread_no__ ≪ "::" ≪ called_proc_name ≪
        "␣==>procedure␣finished␣working" ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in section 203.

646. Trace raw characters.

```
⟨ Trace raw characters 646 ⟩ ≡
  if (yacco2::YACCO2_TLEX__) {
    ⟨ acquire trace mu 389 ⟩;
    yacco2::lrclog ≪ "YACCO2_TLEX__::␣" ≪ "chr:␣" ≪ Char ≪ "␣File:␣" ≪ File_no ≪ "␣Pos:␣" ≪
        Pos ≪ FILE_LINE ≪ std::endl;
    ⟨ release trace mu 390 ⟩;
  }
```

This code is used in section 56.

647. Thread performance macros.
They allow one to see how the thread library stops and starts the threads by their output.

648. Entered into waiting for an event.

⟨ trace COND_WAIT entered 648 ⟩ ≡
```
if (yacco2::YACCO2_THP__ ∨ yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_THP__␣||␣yacco2::YACCO2_MSG__::" ≪ parser.thread_no__ ≪ "::" ≪
      parser.thread_name() ≪ "␣COND_WAIT␣entered␣into␣Wait␣on␣event␣" ≪ FILE_LINE ≪
      std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 150 and 163.

649. Exit out of waiting for an event.

⟨ trace COND_WAIT exit 649 ⟩ ≡
```
if (yacco2::YACCO2_THP__ ∨ yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_THP__␣||␣yacco2::YACCO2_MSG__::" ≪ parser.thread_no__ ≪ "::" ≪
      parser.thread_name() ≪ "␣COND_WAIT␣exit␣on␣event␣" ≪ "␣cv_cond:␣" ≪ parser.cv_cond__ ≪
      "␣no␣competing␣thds:␣" ≪ parser.no_competing_pp_ths__ ≪ "␣no␣active␣thds:␣" ≪
      parser.from_thread__→th_active_cnt__ ≪ "␣from:␣" ≪ parser.from_thread__→thread_no__ ≪ "::" ≪
      parser.from_thread__→thread_name() ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 150 and 163.

650. Before SIGNAL_COND_VAR.

⟨ trace SIGNAL_COND_VAR before call 650 ⟩ ≡
```
clock_t start_;
if (yacco2::YACCO2_THP__ ∨ yacco2::YACCO2_MSG__) {
  start_ = clock();
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_THP__␣||␣yacco2::YACCO2_MSG__::" ≪ parser.thread_no__ ≪ "::" ≪
      parser.thread_name() ≪ "␣to␣signal␣SIGNAL_COND_VAR:␣" ≪ To_thread.thread_no__ ≪ "::" ≪
      To_thread.thread_name() ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 151 and 164.

651. After SIGNAL_COND_VAR.

⟨ trace SIGNAL_COND_VAR after call 651 ⟩ ≡
```
if (yacco2::YACCO2_THP__ ∨ yacco2::YACCO2_MSG__) {
  ⟨ acquire trace mu 389 ⟩;
  clock_t stop_ = clock();
  yacco2::lrclog ≪ "YACCO2_THP__␣||␣yacco2::YACCO2_MSG__::" ≪ parser.thread_no__ ≪ "::" ≪
      parser.thread_name() ≪ "␣after␣SIGNAL_COND_VAR␣:␣" ≪ stop_ − start_ ≪ "␣to:␣" ≪
      To_thread.thread_no__ ≪ "::" ≪ To_thread.thread_name() ≪ FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 151 and 164.

652. Before CREATE_THREAD.

⟨ trace CREATE_THREAD before call 652 ⟩ ≡
```
clock_t start_ = clock();
if (yacco2::YACCO2_THP__) {
  ⟨ acquire trace mu 389 ⟩;
  yacco2::lrclog ≪ "YACCO2_THP__::␣" ≪ Parser_requesting_parallelism.thread_no__ ≪ "::" ≪
      Parser_requesting_parallelism.thread_name() ≪ "␣before␣CREATE_THREAD" ≪ FILE_LINE ≪
      std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 153 and 166.

653. After CREATE_THREAD.

⟨ trace CREATE_THREAD after call 653 ⟩ ≡
```
if (yacco2::YACCO2_THP__) {
  ⟨ acquire trace mu 389 ⟩;
  clock_t stop_ = clock();
  yacco2::lrclog ≪ "YACCO2_THP__::␣" ≪ Parser_requesting_parallelism.thread_no__ ≪ "::" ≪
      Parser_requesting_parallelism.thread_name() ≪ "␣after␣CREATE_THREAD:␣" ≪ stop_ − start_ ≪
      FILE_LINE ≪ std::endl;
  ⟨ release trace mu 390 ⟩;
}
```
This code is used in sections 153 and 166.

654. Notes to myself . .. Decisions.

655. Evaluate if extern "C" should be used in *Set element compare* **functor.**
Cuz its a closed system, there is no need to make the C++ functor global for other languages. So remove "C".

656. Cleanup from failed parallel parse.
As the local parallel parse does not affect the parser requesting parallelism, there is no save/reset action needed on its token stream variable *current_token__* and position. So remove the paranonia code.

657. Verfiy if all successful threads consume a token even if its....
just a remapper on the current token. For example in the Pascal translator, the lookahead token might need a re-verification by the symbol table across all scopes. So call the thread who tries the remapping and returns the result be it the same or remapped.

Now what. Is result in terms of processing the token stream and the new lookahead? I got it from the grape vine that... yup. As per normal — consumption takes place!

658. Manual arbitrator how does it work?.
It's a proxy just returning the 1st token in the accept queue. *AR_for_manual_thread_spawning* is a canned proxy arbitrator for this purpose. There is no judging code. It's a teflon special — nothing sticks to it; just pass back the first item in the queue. *spawn_thread_manually* function sets up this default. Corrected the *call_arbitrator* who originally jamed the first parm into the accept queue. Now, call the arbitrator given for both types normal and manual threads.

Though arbitrator function is a single procedure for that state configuration, it must service all the nested threads with this configuration. I still use the msg as a parameter for calling the function. It makes things simpler and consistent: generic parameter passed that needs casting to its real self. Note: arbitrator is not multi-threaded as there is only 1 copy of itself but it is re-entrant. So when two or more competing nested threads require its services, I leave it up to the operating system to deal with parallelism. It probably throatles back to single process but how many situations are there that use nested competing parses of same grammatical expressions?

659. *Ccm_to_ar* **message needed?.**
I ask the question in light that an arbitrator is a global procedure and not a thread. Yes it is needed as it containes the info to arbitrate. Like what? The cm providing the accept queue for review. Should the parm be a message type? No, but it keeps it simple Dave.

660. Why (CHARP) instead of Cparse_record definition in....
the *reduce_rhs_of_rule* function? Well back in time, u got it, Microsoft's compiler was a honking. So if you look at the generated code for a concrete *reduce_rhs_of_rule*, you'll see how it games itself down thru the stack equating the subrule's parameters in LIFO. Does it still hold this quirk? Don't know until I retry. At the moment, I have too many other things to complete.

Well I'm bitting the ??? to make things faster. Rewrote the stack and corrected for speed the emitted code of the rhs subrules. eliminate (CHARP):
3 Oct. 2005
Added rule recycling to speed up parsing due to the rule's birth-run-delete cycle.
June 2007

661. Why nil ptr test in `T_11`?.
Originally some symbols pushed onto the stack were zeroed out to protect from abort cleanups etc. This situation does not exist anymore. So rid it ghost busters.

662. Clean up parallel parse in control monitor instead of grammar....
requesting parse. It's just cleaner and closer to the action. Here are my original thoughts. Some house keeping is done. The cleanup is to pop the `|||` symbol from the attempted parallel parses. It could have been done in the control monitor who was the creator of this but I felt that spreading this cleanup to the control monitor was potentially spreading the mess.

Dictum: keep the effects' cleanup as close to the affect. Is this an Occam?

663. Conversion of control monitor and parallel parse code.
This is the injection code included into the outputted grammar modules from Yacco2. Conversion cleaned up dregs from cm handling of a |?| dynamic code request. A thought of minimal value where there are other means better to cope with this type of situation. Now what is this situation? How do you cope with a parsing situation like the syntax directed code that needs parsing? There is no assigned set of grammars to properly parse the C++ code. So, do a dynamic parse looking for a dynamicly calculated lookahead token to stop the parse-by-character situation.

Now the good stuff. The *cweb* worked first time both in the control monitors and parallel grammar threads. Let the applause begin.

664. Why is there an abort attribute in the parse stack record?.
If there is a symbol on the parse stack with 'affected by an abort parse' turned on, the cleanup of an aborted parse will delete the symbol like an "auto delete" when it pops the parse stack.

665. Make all yacco2's types, structures etc housed within yacco2's namespace.
The 'INT' type is also used by Microsoft. So, add 'yacco2::' qualifier to all definitions and implementations. This way there is no conflict of interest when porting to other environments.

Correct also the implementations to be qualified by namespace yacco2. There are 2 ways to do this. Firstly, be explicit per implementation. Secondly, enrobe the implementations with a namespace '{' ... '}' construct. To each their own ... you'll see both approaches depending on my mood.

For the moment, files *wcm_core.h* and *wpp_core.h* are not explicitly qualified by yacco2::. This allows the old current code that uses this to be compiled until the *cweb* version is completely finished. The current system does not include everything within the yacco2 namespace.

666. Make enclosure of namespace yacco2 explicit in implementation part of code.
Eliminates assumptions. ⟨ bns 24 ⟩ and ⟨ ens 25 ⟩ bracket the code to be housed within yacco2's namespace. All implementation code contains this start/stop definition. The code *wcm_code.h*, *wpp_core.h*, *war_begin_code.h*, and *war_end_code.h* are just that snippets and so are contained within another implementation. They still use ⟨ uns 23 ⟩.

667. The old version of terminal enumeration:.
The terminal alphabet is represented by the whole numbers both positive and negative. Both errors and regular terminals are open ended in their expansion capabilities as they are the left and right end points in the terminal enumeration scheme. Error terminals grow towards minus infinity while the regular terminals expand positively. The balance or pivot point of the terminal alphabet is 'eog' that starts the meta terminals. Meta terminals are indicators of parsing situations like end-of-token stream reached, parallel parsing to take place, to different wild type shifts. None of these meta-terminals are found within the input language being parsed.

The *Base_enum_of_T* parameter of 'fsm' is the starting point of the enumerated terminals. Due to the current enumeration scheme, its value is required to map a terminal's enumeration id into a set's co-ordinates. This is a bit of a hack as each grammar contains this starting point. The hack comes about from an out-of-sync condition when new errors affecting this start point has been defined and all grammars have not been recompiled and passed thru Yacco2's linker. The consequence is the parser when run will have strange things happen because of the wrong enumeration mapping to the terminals that are buried in the old finite automaton's tables. Trust me, I'm the guinea pig. Regenerate all the grammars.

Raw characters represent the mapping from the 8 bit ASCII character into its raw character terminal. Error terminals are internally generated situations produced by the parsing grammars manufactured by the grammar writer. They indicate the appropriate faulty situation detected. Regular terminals are composites. They get created by the grammars from streams of other raw character terminals or composite terminals. They are evolutionary and come into existance from various passes made on the token streams: lexical to syntactic to semantic.

Reason to change:
Why this type of mapping instead of the positive integers? Reality is there is no difference apart from using the range of numbers and how they expand. Both meta and raw character terminals are constant in size. It is the other two types that expand or evolve as one is developing the language recognizer. Either way of enumerating the terminals, when an error or a new regular terminal is created, all the grammars need to be regenerated due to the change in the lookahead sets. Hindsight critiques that a start seed buried in the grammar's finite state automaton definition is required. So get rid of it! The better design is to enumeration from 0. This eliminates the mapping from the negative space into the positive space of the set co-ordinates.

Take 2: Here is the new mapping: meta-terminals, raw characters, errors, and finally the regular terminals. There is no need to map into the positive space before calculating a terminal's lookahead set co-ordinates. Just use the enumerate value to translate to the set's partition and element!

668. Tree token template container.
Well let's try passing references instead of pointers. I hope that the compilers are kinder to me within the threaded environment. This certainly saves alot of constraint checking. 14 Oct. 2004.

669. Add in Yacco2 arbitration requiring code on the possibility of....
2 or more terminals in the accept queue with no arbitration taking place. That is, it defaults to the first terminal in the queue. The compilation check requires the checking of their first sets for the common prefix condition. At the moment, this does not take place due to the yacco2 / linker loop. Yacco2's linker generates the transitive first sets for the threads that call other threads. So this check is is a post condition beyond the compiler/compiler. At present, Yacco2 issues a warning and use at your own risk.

At runtime, there still needs a look-over-your-shoulder throw condition. This will be implemented in the arbitration code. This is done — 26 Oct. 2004 in Yacco2 generator. There is an optimization done before the throw code is appended to the arbitration thread:

1) more than 1 thread must be dispatched — thread with a name: NULL name bypassed
2)no arbitration code supplied by the grammar writer

670. Rework of thread management.
At present it is spread between the global implementation of independent methods and the table of spawned threads, and the worker thread record structure.

671. To check: does stop msg have wait/reply mechanism?.
In the shutdown? no.

672. Change tree container to a specialized version of tok_can⟨AST ∗⟩.
This makes things more consistent. Now, all u see are specialization containers. So why did u not do it in the beginning? This container was an after thought. It was written to support a Pascal translator to re-target a preprocessed Pascal variant using Oregon Software's compiler to Dec aka Compac aka HP Pascal. As there were special extensions to the Oregon Pascal, a complete front end compiler was needed to build a source tree of the program so that the source code could be morphed. There were lots of sinning go on. Well the outcome was this family of tree walkers and container. So what! Why did u not write a template specialization? Probably too deep into getting it done without the thought to whether it has any generalization. The other containers using string and ifstream did specialize but... 11 Nov. 2004. Now to correct the grammars that use the old container *tok_can_ast*.

673. Eliminate the control monitor.
The middleman is too expensive as a thread due to the current threading model. This helps in optimizing the run performance of Yacco2. To do this meant moving all the responsiblities of the control monitor into the grammar requesting parallelism. This plumbing is within **Parser**. Part of the demolition meant throwing out the messages between the various components — pp between cm between th. Now the message is the media or is it the **Parser**? The requesting **Parser** just passes itself to the grammar threads. It contains the pertinent token stream variable: token and position (current values) within the stream, and all the token containers — supplier, producer, recycling bin, and the error container (refuge shelter). Also removed was the distinction between the containers — parallel versus monolithic. As parallel grammars just graft onto the current token scene, there is no need to make the distinction except at their start up time that grabs their containers' addresses from the spawning parser. They are just readers of the tokens and not writers. Now what about error tokens? They should not be added to the error queue but should be passed back to the calling grammars within the **Caccept_parse** object. The arbitrator of the calling grammar determines what should be done. If u need to add to it then use the guard dog approach or is it the drake? "i get no respect" so choose your mutex before doing your thing.

Done 23 Nov. 2004. Performance gain: 30 percent.

674. Eliminate *pp_support__* as a thread optimization.
All info in now contained in **Parser**. Depending on how the thread is started — monolithic or parallel, the appropriate parse containers are imported either thru the contructor or via the passed parameter.

675. Another thread optimization.
If only 1 parallel thread asked to perform, one does not need to acquire / release the lock of the requesting grammar to report success or failure.

Look I'm trying to make threading closer to recursive descent in performance. Date: 3 Dec. 2004. Well I'm crawling out of the swamp... darwinism? If there is just one thread to be run, why not call it as a recursive descent procedure instead of the thread route. We'll see what the cost of thread modulation is against the procedure call approach and its object creation / destruction overhead. Take 200.1... 9 percent run improvement of procedure call over threads.

676. a N * 2.
Eliminate the number of times that the token container is read does miracles. Now let's look at my myopia. There was a single pass, call it P1, to break up the character stream into line segments followed by the lexical segment called P3. Why? I was lazy and wanted all down stream tokens to be properly tagged in file no - line number pairings. Why lazy? The P1 pass ensured that the tokens where properly GPSed. I did not have to deal with the vagaries of "how is a line delimited?". It was handled in one place: the "eol" thread, and could be retargeted to other dealings. Now the logic is hardwired for now to the "line-feed" definition based on Ascii encoding. By combining the 2 passes (P1 + P3), the number of reads on a N character stream is halved.

Now lets look at the raw character to symbol translation. Again this is a 2 traverse mechanism that reads from a file its characters that are translated into symbols. It should have been a just-in-time read like the tree traversals. Each character request fetches the character from the file and then calls the character translator to do the cosmetic make over. This definitely improves the "file include" process. This is a reduction from 37 seconds to 15 seconds. Not bad: a 2.something zinger.

Now for the overhead of raw caharcters to symbol objects. Judging from the cursor winking, this could be another 10 second improvement. Wait and see... Ladies and gentlemen and the winner is ... 37 seconds down to ??? Maestro the envelope please. 15 seconds! A 22 percent improvement against the 100 second starting point but 2.something faster against the 37 seconds. Slimefast ain't got nothing on us. As the song says — looking for xxx in all the wrong places.

Now what about the cost of symbol creates and std::map usuage in the thread library and the garbage collector? I'll see what I can do. I must approach the recursive descent speed zone or this thought experiment on parallel parsing is just that — religated to the empirical sidelines. A second string something and excuse the pun.

677. Remove *unique_id__* from CAbs_lr1_sym.
It's original purpose was a birthing number to give a count to the number of symbols produced and as a partial order. Never used so out damn thoughts! Dieting and speed is in.

678. Okay guys Yacco2 is starting to smoke.
Here's another improvement. Firstly I was looking in the wrong places: String copy was thought to be a major cause but it turns out that its a minor overhead. Globalization of the character storage is good at the cost of saftey but not a really really big stopper.

So here's the scoop: First set evaluation goes thru INDIVIDUALLY each potential thread contained in the state's configuration list.
If there are many potential threads to-be-run assessed on a per character basis — ouch. All one has to do is gather the threads into a consolidation thread to have only attempted pass on the first set of the consolidation thread. Yacco2's linker consolidates this first set of referenced threads. If the threads are orthogonal to one another (there is no common prefix), then the single first test lowers the cholestoral levels.

With this insight, now to modify the grammars like: pass3, lint, syntax directed code gatherer etc. Jan. 1/2005. Well this had limited improvement. Not what was expected so see *Global Parallel table entry* where it explains how Yacco2's linker became involved. Jan. 6/2005. Speed improvement — ???.

679. Slim down the CAbs_lr1_sym space.
This is the base component to all other symbols. Originally I had associated the parser across all symbols: Terminals and Rules. This fattened the space by 4 bytes. With a shrinking of some variables to short integer and unionizing the rule's variables, I brought down the space bloat from 36 bytes to...24 bytes. So what? Well, this allows more raw characters to be stored in a prefixed array rather than a template container.
3 Jan. 2005.

680. Grammar as a logic sequencer: Allow no token containers.
What type of improvement is this? By passing in pointers to the parser, does this not open up more programming mistakes? Could but hear my reasons please. This lets the grammar writer program grammars as logic sequencers using epsilon rules and related syntax directed code. If the writer is very creative, behavioural terminals could be defined and put into a token container for parsing: each to their own. See *enumerate_T_alphabet.lex* as an example of this use.
15 Aug. 2005

681. Logic bug: same accept token added to accept queue more than once.
Help the needy, the grammar has launched multiple threads and these threads have returned the same token. This condition is caught by the number of accept tokens in queue is not equal to the number of threads reporting success. The needy? well i was caught with this logic bug. See *Arbitrator code generator* where logic check resides.
13 Dec. 2005

682. Porting of *cweb* code.
Make sure the the @i include construct uses quoted file names. Without the quotes, the mac version of *cweave* has a slight stammer. The Microsoft flavour works.

See *Generated finite state automaton macros* for more stumblings from within. The c macro definition workaround works but the references to the macros are not placed into the Index.
16 Dec. 2005

683. *cweave* C++ code.
Removed ending semi-colon from `RSVP` macro to have *cweave* print out these type of token macros onto its own line. So make sure u add a ";" following their use.
8 Jan. 2006

684. *failed* directive added in the *fsm* construct.
I felt the grammar writer should be given a last-chance to deal with failed parses. Why? For example, my *yacco2_lcl_option* needs to deal with options having multiple letters. Now how do u program these options whose via prefix is faulty? For example, option -err has -e and -er as the potential option but are in error. One could explode on the combinatorial code within a grammar to deal with each evolving prefixe or force the calling thread to handle the failed thread with some form of epsilon in the grammar code. This is crude so why not field a returned error terminal? To do this i needed a directive of last-chance to be tried in the *parallel_parse_unsuccessful* procedure. For the moment, it is only supported in a thread grammar. Possibly i'll look at the monolithic grammar and what it means in particular for error correction.
8 Mar. 2006

Verified that *failed* directive works in a *monolithic* grammar. 2 thumbs up for consistency. Just make sure that a "failed" directive within a monolithic grammar places the Error T in the "Error queue" via the `ADD_TOKEN_TO_ERROR_QUEUE_FSM` macro and not `RSVP_FSM` macro: this places the error into the "accept queue" which is wrong.
15 Jun. 2014

685. More token info for tracing.
Added to token trace macros the GPS of the source. This allows one to see where within the source things are occurring.
22 Mar. 2006

686. Added to the CAbs_lr1_sym definition a "who created" GPS.
Comes in handy when errors are throw but from where? Errors are directed to the source file with no fingering as who the grammar was that generated it. So it's up to the grammar writer to tell it as it is. Now the *O2_err_hdlr* grammar can spread the word so to speak... if it is available. See *set_who_created*, *who_file*, and *who_line_no*.
22 Mar. 2006

687. Rewrote tok_can⟨AST ∗⟩ due to global functor firing.
Originally i had the filter mechanism within the **tok_can⟨AST ∗⟩** container. This lead to the functor being fired by the advance routine regardless of whether the tree node was rejected or not. Why the oversight? i did this to quickly knockoff the tree container. Now it's in the tree walker where it should be. This way the functor only gets fired if the tree node fetched is accepted by the filter or there is no filter.
17 Apr. 2006

688. Adjusted array of "[]" declaration.
Originally i defined arrays of unknown size as type variable-name[]. Porting to Sun did not like this. So my delimma was "how to define a base table structure for each table for threads, shifting, reducing etc?". The emitted cpp tables were explicitly sized in their definitions for the "bsearch" function to act on but my generic search code was open-ended having no knowledge of each table's size.
Solution:
Create a base definition of only 1 entry:

```
// array_def.txt solution to open-ended array definition
struct Shift_tbl {
  yacco2::USINT          no_entries__;
  yacco2::Shift_entry  first_entry__[1];
};
```

22 Dec. 2006

689. More porting issues dealing with threads and syncing signals.
When there was only 1 thread requested to run, i optimized out the mutex acquire / release cycle and left the Caller parser and the Called thread to complete their launch cycle by a) Caller parser goes into a wait state by *pthread_cond_wait* and b) the Called thread signaling the Caller parser by *pthread_cond_signal*.

What happens when:
A calls only 1 thread B and B completes before A puts itself into a Wait stupor. IE, B will be signalling A to wake up. It depends on the Pthread implementation. Some will queue it up for the wait signal to happen and then pass it back immediately to the Caller while Sun drops the signal and so ..... hear the zzzzzs from the sleeping beast and the anxiety from the compiler writer while waiting and wait....

Conclusion: Remove the optimization and just use proper acquire / release hygiene to deal with syncing between friends. As procedure calls are slower then thread calls due to "oo" variable initialization and destructor clean up , I'll just remove completely the conditional `THREAD_VS_PROC_CALL__`. My tracing works VERY WELL to diagnose this problem. Here here.

Dregs of past thoughts:
`THREAD_VS_PROC_CALL__` thread versus procedure call performance.
It must be defined as it is a preprocessor conditional symbol! There is a cost of calling a thread versus a procedure call. What is it is the reason for this symbol. When there is only one thread to be launched, this becomes a procedure call instead of a thread. Where I'm the doubting Thomas, is the cost of objects birthing and dying greater than having a thread startup and put on reserve for other calls?

`THREAD_VS_PROC_CALL__` of 0 calls threads and 1 calls procedures. The winner is procedure-call by 9 percent. NOT ANY MORE! It's threads cuz of oo's overhead in those damn objects and their rights of passage.

16 Jan. 2007

690. Changed back to passing Parser as a pointer for tracing purposes.
When the going get debugged, it a hell-of-lot-better to see what the pointer is pointing to in the debug session rather than just an address. Maybe a weakness in the Sun Studio debugger but so what. This will allow me to see if i'm clobbering memory by the data per parser environment.
29 June 2007

691. Some more optimizations.
The grammar suite takes 1:50 minutes. Now to improve.

692. 1) precalculates a compressed set key from a terminal's enumerate id.
This eliminates everytime a reduce takes place mapping the terminal's enumerate id into a compressed set key format so that the lookahead set can be searched. Its a tradeoff towards space for speed. Adjusted **CAbs_lr1_sym** to contain and manufacture the compressed set key. The performance improvement is approximately 20% — 35 seconds on grammar suite.

693. 2) eliminate passing shift's element enumerate value.
Split the *find_shift_entry* into 2 contexts:

1) current T context
2) Rule or returned T from parallelism context

The 2 routines are *find_R_or_paralleled_T_shift_entry* and *find_cur_T_shift_entry*. 5 seconds improvement on grammar suite.

694. 3) eliminating the tok_can reader mutex — nope.
Well here's the scoop. The **tok_can** templates are "just in time" (jit) in accessing their contents. What does this mean? For example, **tok_can**⟨*ifstream*⟩ container is a wrapper to access raw characters of a file returning the raw character transformed into raw character token placed into its secondary container for possible reuse. If the read request has the token in its internal container — container inside a wrapper container, then it returns it via the inside template container's operator[xxx]. Now for the "jit", if the [xxx] request is not inside its internal container, **tok_can**⟨*ifstream*⟩ calls the ifstream object to fetch the next character. For far so good but put this into a multithreaded context where there are 2 or more cpus running at-the-same-time. Now the **tok_can**⟨*ifstream*⟩ ifstream object becomes a critical region. What is the critical region part?: its subscript. Even though my *get_next_token* request is reader only against the **tok_can**⟨ ⟩ container, this container itself is a reader/writer depending on the context — reader if it has the request squirelled away in its token container, but a writer when it does not contain the request and must access the ifstream object. An optimization test was conducted, no "jit" character accessing by the **tok_can**⟨ ⟩ (all the characters were read at time of open before any read requests were done) versus the "jit" with guarded mutex. Though the winner was no "jit" by only 3 seconds over 80 compiles, it was not worth the gain over a slighlty unsafe attitude. I would have needed to adjust all **tok_can**⟨*xxx*⟩ variants to remove the "jit" unsafe condition.
August 2007

695. Elimination of reader mutex for optimization reasons.
The Ides of nagging made me do it for speed. So mutex control has been eliminated from the "jit" containers that are now not "jit". These template containers now do a double read across their input as the cost of the read mutex is tooooo slow: 3/80%. I'm putting into my subconsious the problem to find a better silicon / hardware solution to critical region control. I'll have a look at the overhead using Sun's "dtrace" facility not only for mutex overhead but also other optimizations that can be done to O_2 to approach top-down parsing speeds — ie O_2batch versus O_2: O_2 is approximately 4 times slower. Don't know if this is an accumulation of c++ and templates etc against a bare bones O_2batch "c" language approach?
Sept. 2007

696. Parallel thread reduction should be lr(0).
Here's the scoop: if a thread's lookahead boundry is a superset of what should follow, the returned lookahead token could be in error. As O_2's reduce operation looks to find its boundry dependent of the faulty lookahead, guess what it throwns an error due to the lookahead token not found in the reduce table of the calling grammar. So create a new *find_parallel_reduce* procedure that just returns the first **Reduce_entry** to complete the reduce. It effectively is lr(0): no concern for the following token context!

Now the error can be dealt with by programming the shift operation within the grammar using either `|+|` or `|.|` to capture the faulty parse point and to report a specific error against the GPS of the returned lookahead token.
Oct. 2007

697. Make *accept_queue* more efficient.
Make it a fixed array of local **Caccept_parse** for 2 reasons:

1) eliminate the new / use / delete cycle: malloc is too slow
2) don't need a map but just a sequential queue

This gives a 13 percent inprovement.
Nov. 2007

698. Use Procedure call when only 1 thread needs to be run.
The mutex / thread paraphrenalia is tooooo slow compared to a procedure call. This thought was nagging me since my 1st O_2 compiler written by recursive descent. It became my bench mark that thread parsing was measured against. Yes i'm aware of the bottom-up optimization by Ullman but i'm not there yet in digesting the optimized requirements to lower the push / pop overhead by consolidation of subrules and their syntax directed code that need some form of sequential sequencer when the consolidation consequence must get exercised.

Now why come back to this subject anyway? Those nagging optimization muses! I eliminated the mutex controlls due to threads and my critical regions; there is a 1:1 activity taking place whereby the calling of the procedure by the requesting grammar passes the right to the called procedure to enter its critical region when needed without the paranoia of duality destructive conditions. By making the **Parser** and its evil grammar fsm twin global and by mallocing them within the called procedure, the overhead should be lessened. Mastro the envelop please. And the winner is: 25% faster. How was this measured? My Apple laptop where running times between threads only against the hybrid approach where taken using the *o2grammars.bat* script.
Dec. 28, 2007

699. Thread's start-up attributes for stack size and system scope?.
I played with *pthread_attr_setstacksize* and *pthread_attr_setscope* attributes to improve possibly speed and fat deposits. Well the *pthread_attr_setscope*'s setting of `PTHREAD_SCOPE_SYSTEM` made things worse as this was an aggregate of all things considered. Procedure calls of threads by threads made the run environment too sensitive to this unknown size mix. The result can produce a `SIGSEGV`. Experimenting by increasing the *stack size* delayed the problem but bloated the run size. As always the cure was easy: just remove this fiddling and default to the runtime attributes of the local *pthread* implementation. On the Sun Solaris, the stack size for all threads is 1 megabyte — more than enough.
Apr. 2008

700. Error detection within a grammar: new `|?|` symbol introduced.
`|?|` was created to handle questionable situations like error detection points within a grammar. It can be expressed as a normal shift terminal or within the returned T of a `|||` thread expression. As the lookahead symbol is questionable, using the `|+|` or `|.|` symbol to handle error detection has one weakness: its subrule reduce operation depends on the lookahead set which the current T could be not in this LA set. Consequently the reduction could possibly will not action. Introducting the new symbol draws the reader's eye to the error point with the grammar. The reduce is a lr(0) context which means no dependency on the current symbol and so the subrule always reduces! This allows the grammar writer to coerse the parser's behaviour by the subrule reducing syntax directed code.
Warning:
The current token is **not advanced** so perpetual motion on the same token spot could occur if one is using the `|?|` to act like a `|+|`. ⟨Invalid `|?|`instead of `|+|`use 543⟩ has been created to detect and stop the parse process. So be warned.
June 2008

701. Speed wonderful speed in “Oliver Twist” and not William Burroughs.
Well the rule recycling works now. No more new(s)... Just recycle them grammar rules. The envelope please ... 25% speed improvement from 32 to 24 seconds against all them grammars. As time shrinks there seems to be an asymtotic return on performance improvements. But this one is good; no really very good. I'm only 4–5 seconds away from the recursive descent bench mark. It's malloc! and its mutual exclusion that is very very expensive by the following “dtrace” outpout.

0 57766 lmutex_lock:entry
 libc.so.1‘lmutex_lock
 libc.so.1‘malloc+0x25
 libCrun.so.1‘void*operator new(unsigned long)+0x2e
 o2‘void NS_o2_sdc::Co2_sdc::reduce_rhs_of_rule(...*)+0x282

The above trace also brought out my sloppiness in proper code emmissions per grammar's *reduce_rhs_of_rule* routine. I never stored the newed rule so each time the grammar was run the used rules were recreated — uck.

Dec. 2008

702. Improve dumped data when Shift T not found in parse table.
See where it is thrown. Though this is a grammar writer's lack of error catching in his grammar, at least dump out the info on T: its enumerated id and literal. Now the info dump contains the grammar in question, its current parse state, and the T details. Why isn't it using a Error class T and to use O_2's generic error queue dump facility? Cuz this is below the user's language: remember this is a generic interface without any knowledge of what's being built on top of it. And I didn't want to force yet another canned set of T definitions like lr constant and rc.

Feb. 2009

703. VMS spits core dumps when its thread stack is exceeded.
Ahh recursion is sometimes devine but not when the stack limit is exceeded thinking its a runaway recursion call when A recurses on itself without any stop recursion detection. So U must increase the `VMS_PTHREAD_STACK_SIZE__` symbol in the *yacco2_compile_symbols.h* file and rebuild the O_2 library. The allocated thread stack size was 128k before the Pascal translator starting to choke due to better symbol table management that increased the *pas_variable* grammar run size when dynmically creating the statement variable's symbol table components. double ugh but this is reality.

Feb. 2009

704. Caught by your short and curly — local variables in grammar rule.
The short of it is the recycling of rules to new once reuse forever. The consequence is the rule gets recycled and if u have not reinitialized the variable aka an array or table then the past dregs of invocation will haunt u. Either crate the variable in the “fsm” grammar construct or reinitialize in the rule's construct directive. Better yet do it in the rule's “op” directive before the variable is being used. Do u really want the curly part? Of course not so where did it grab u Dave? Grammar *la_express* to calculate the lookahead expression. Rule reuse happens on “+”, “-” expressions: eolr - ”.”.

Feb. 2009

705. Add a complete trace on fetching a T when symbol functor in use.
When the *tble_lkup_type* token fetching in its various forms attempts to remap the raw T, i just traced the fetched T before the potential remapping took place. If the symbol table functor is in place and turned on then the after attempt is now also traced. This was highlighted when i wrote a Pascal translator with a syntax directed symbol table scope handling and my myopic test was the problem as i put an externally defined function within a local procedure. Boy my misfits never cease to entertain. This seems to be my problem where the original test item was faulty. I guess u could say my grammars should have caught this faux-pas but they were not written to catch all sins but to remap one correct Pascal program into another correct Pascal variant. Some error reporting is being done but the more others use it the more retrofitting of error reporting is taking place. More for the weary when problems prevail.

Feb. 2009

706. Add right recursion support for rule recycling.
Well how did i treat this? I detected full rule use consumption and outputted a message to the grammar writer that all the allocated rules were in use and exited with a message. Please see grammar *rules_use_cnt.lex* as to how it counts number of rules in a left recursion scenario. Well this was not good as right recursion has its place in parsing though it hits hard on the parse stack. How so? Before the rule can be reduced it keeps pushing aka shifting until its lookahead boundary is met. So if the parse exceeds the fixed stack size it will still honk with an abrupt message and quick stage exit. Staying within the stack allocation is fine. See `MAX_LR_STK_ITEMS` as to the parse stack allocation: adjust accordingly.

Feb. 2009

707. Changed input order of T Vocabulary — exchanged T with Error T.
Why the change? This allows the grammar writer to write independent compiler/grammar combos — Eg front end lexing of Unicode, so that the front-end creates the external token container for the other compiler/parser combo to digest. Currently all token containers are memory only template derived. With this change the parser/grammar(s) T Vocabulary now appends the Errors at the end of T Vocabulary enumeration scheme. The second parser/grammars combo must include the first T definitions in their own T Vocabulary in the exact order defined by the first parser. From there it can build its own T Vocabulary of additional Tes and Error symbols. Another way is to remap the enumerated ids of the first parser's tokens into the ordering scheme of the second parser. Use of the token read functor associated with a read token container to remap Tes at read time. It could just change the "enumerate_id" value of the old token into the current parser's T Vocabulary mapping. It could also create a new token but this itself is overkill unless one is remapping the token into another different token type: for example remapping an "identifier" token into a keyword by use of a symbol table lookup.
Caveat.
Currently the O_2 library has globally defined symbols that get resolved at linker time. So one cannot run mutiply defined independent threads of parsers with having exclusive use of O_2. O_2's implementation contains multiple independent parsers sharing the same O_2 library and only 1 super set of Tes defined for all parse stages. For example, the command line to O_2 gets parsed by its own grammars and their outputted tokens become downstream fodder for the suite of grammars used to parse the inputted grammar file.

There is still work to be done to consolidate O_2's external symbols into a structure containing indirect pointers to these symbols that are currently resolved by the linker (ld). 1st thought:
1) have a local structure initialized to these pointers.
2) register this structure of pointers with the runtime library of O_2 before any parsing begins.
3) each independent parser can run in its own thread
2nd Thought:
1) use a fork process where the token containers are passed somehow as input to the subprocess that fills its booty. This thought is similar to the spawning of a grammar as a thread or its optimized procedure call.

May. 2009

708. Tree container is out-of-sorts from self modifying trees.
Well its back to just-in-time (JIT) reading of the tree **tok_can**⟨**AST** ∗⟩ as the following example outlines why:
Given a grammar that reads a specific T type like "call-stmt" and u want to change its younger brother to a different T. What happens during the parse? The current T is shifted onto the parse stack and the lookahead T is fetched becoming the current token. This LA T will be a "call-stmt" possibly used to reduce the shifted T "rhs" subrule. The problem is the container has the unmodified reference of the lookahead T. Now within the grammar's syntax-directed-code u process the younger brother nodes to which u changed some of the tree's content. If u are unlucky, the LA T's id gets changed. Irrational behaviour could occur: the parser doesn't reduce properly or possiblely as the T type is different from the parse stack frame entry of "call-stmt", this acts like an uninitialized object having random behaviour.

So what can one do? i corrected the **tok_can**⟨**AST** ∗⟩ container to JIT reading of its Tes and implemented the *remove* method that pops the last entry from the container. If u are modifying the T type of the tree: ie replacing the tree node's content with another T type, now the grammar writer must add syntax-directed-code to remove the LA T from the container, re-align the current token position to the shifted T position, and do a "get_next_token" to fetch the proper LA T thus maintaining the integrity of the parser. All this sounds like a lot of work but here is an example of such coding:
An example:

```
 1:  /*
 2:  file: /yacco2/diagrams+etc/treemodify.txt
 3:  Example of re-aligning the parser's LA T when dynamically modifying the tree
 4:  */
 5:  tok_can<yacco2::AST*>* ct
 6:              = (tok_can<yacco2::AST*>*)parser()->token_supplier();
 7:  ct->remove();//drop the la T as i could have morphed this into a comment
 8:  parser()->override_current_token_pos(parser()->current_token_pos()-1);
 9:  parser()->get_next_token();
10:
```

The code above is taken from a grammar's "rule" syntax-directed-code. The rule has a reference to the parser environment and doesn't have to go thru the "fsm" route to get at the token supplier. lines 5–6 gets the tree token container from the parser and casts it to a tree container. Lines 7–8 removes the last T from the container and re-aligns the parser's current token position to the shifted T position. Note: All token containers have subscripted token access starting from 0. Line 9 fetches the new LA T for the parser to continue merrily along its way. There are other ways to re-align the LA T: Please see ⟨Parser's token defs 229⟩. All this for dynamic modifying of trees: good stuff!

May 2009

709. Multiple Reader/Writer improvement to supplier container.
Historics: JIT fetching of tokens from an "ifstream" container demanded locking when the request was not in the container. Consider 2 parallel threads A and B competing where their read requests to the container are simultaneous: A on cpu 1 and B on cpu 2 and their requests are not in the container. The critical region becomes the physical i/o to the "ifstream" object when the request was not within the container. So what did i do? experiment 1 was remove the JIT attitude and read all the "ifstream" characters into the container at file open time. Now the container becomes a read-only with no need to use locking. So "ifstream" issue is solved but what about a tree container with T filtering? It is a JIT container that requires locking protection as u do not want to walk the complete tree filling it up before the first read request. Also consider a self modifying tree. What? The Pascal translator required the following:
The HP "delete" call statement had to be removed and replaced with a raised signal variable so that its future close statement could deal with it using a "delete disposition" clause within a modified close. This future close tree node was morphed into a conditional subtree dealing with "to delete or not to delete" issue. Without the JIT attitude the tree walker has remnants of the before tree surgery. The container could contain items that are no longer valid due to this modification.

Back to the JIT and Quick overview of mutual exclusion.
When a writer in introduced, locking protection is required if there is more than one simultaneous accessor to the container. If there are only readers JIT still demands writing to the container before the read request can be satified. No lock protection is required when only one suitor is active. Within the parsing environment, all threads are co-operative and must house clean when completing their task even though they might abort. By keeping a reader/writer count against the container and per parser, the supplier container lock usage can be optimized according to the simultaneous number of accessors.

What about the other containers: recycle-bin, error, and producer? Do they require lock protection? Yes they do when they are being filled and yes when they are acting as a supplier container. As they are more infrequently used, i leave the locking mechanism with the "add_token_to_xxx" procedures where xxx is one of "error_queue", "producer", or "recycle_bin". For occassional back door T adding to the supplier, the "add_token_to_supplier" procedure is lock optimized on simulatneous accessors as the supplier container maintains its suitor count.

June 2009

710. Removed *grammar_stk_state_no__* from the CAbs_lr1_sym definition.
The original thought was to capture the parse stack number at time of T creation for error tracing. The thought was half baked as what happens when a T is created outside of the parsing environment — no parse stack? So out half-baked! If the grammar writer needs this information, it can be programmed explicitly by the grammar writer by adding the appropriate attributes to the error T being logged.

June 2009

711. Note on what's in the token container and its size.
The "end-of-grammar" condition signaled by the *PTR_LR1_eog__* T is not an element of the container. Why? It acts as a conditional being only-the-lonely as only the Tes in the token stream are contained. So u are warned. If u are testing the token container for size — for example u walked a tree container with filtering and u are testing whether the 2 Tes and the "end-of-grammar" condition are there, u should test the container's size for 2 elements and not 3. Why all this verbage? whispers to myself.

June 2009

712. Sets: Sequential versus binary search optimization.
Well what is the break-over point when to use a sequential search on an ordered table versus a binary search? This question came up when i wanted to improve set handling: aka shift, reduce operations within the fsa state. Try to paper out the result! I finally wrote a simple program to gather stats on the break-out point. Surprizingly it was 72 elements. The test used a table of elements having a multiple of 3 as 1*3, 2*3, etc. The population went from 1 to 128 elements, and for each element in the table, a spanned search key of +,-, and = the element key was done. This was run against each search type to find out the break-over point on instruction costs. Now all state searches have a dual strategy tested against the `SEQ_SRCH_VS_BIN_SRCH_LIMIT` constant as to what search type to use.

July 2009

713. Change T containers's subscripting to unsigned integer or my subtle stupidities.
Why the change from signed to unsigned integers for size, subscripting? Depending on the stl template library, there will be unresolved references to method like "size" that returns unsigned.

Stupidity number 1: overloading the subscript range: subscript $< 0 \Rightarrow$ have not accessed container for T, before first time access, etc. U get the notion. Due to this, "first-time-accessed", and "end-of-container-reached" attributes were needed. Tree walking with filtering needed special attention in the "do i already have a T in the container?" and "end-of-tree-reached". That is, a request could be asked to fetch a specific T after the "end-of-tree" has already been reached.

2nd stupidity: not commenting / documenting that a Parser expects that the T is already been fetched before it requests it. This showed up in my haha finetuning of my logic on tree containers and the discrete logic grammars getting nada input: dead end T.

Cost to my overloading, about 8 hours of work to farret out these subtleties. I know its rather simple but this is my twilight zone of stupidity.

Nov. 2009

714. Porting to Microsoft: Visual Studio 8.
Some not so happy comments on 32 bit console application:
1) They got it wrong when it comes to C runtime (CRT) and their different calling types: __cdecl, __stdcall and how their libraries static or dynamic were built. The threaded library needs __stdcall, while the main program needs __cdecl. Each library draws from its own memory pool depending on what library type u are using. So build everthing using __cdecl and fine-tune the call to "_beginthredex" with __stdcall.
2) U better choose the right type of multi-threading "/MT" or "/MD" or Klack-klack-klack? Well trial-by-error discovered "/MT" is the right one and not their choosen default.
3) Forums are thin on quality but lots of verbage on multi-threading: Try looking up exit code (255).
4) U better use "/force:multiple" to allow all those common c++ rtns to coalesce.
5) Last, their Release libraries don't work! its blows up before the program "main" is entered into. So the port has the porky version but it works!

Alas poor fool for thinking they improved on this from Visual Studio 5 to 8. It was trial-by-the-blind using the various combinations to get it going. Better cosmetic documents but of same software quality ilk. Well my tea reading is this: cica 2003 was move to the CLR / C sharp development and leave as is the 32 bit console application code. Let the street hawkers spin their new tails of enchantment to follow them. Anyway the port is done but tooth mashing ain't fun.

Nov. 2009

715. Mutexing the containers.
A review:
1) All containers start with one owner. Therefore the 1st fetch is safe.
2) All sequential reads from a container is safe.
3) After a T is delivered from its container, the container checks nto see if the request was for its last T inside it. If so the container will do a future request by itself and not by the consumer. That is it is pushing the race condition ahead to maintain saftey to the consumer.
4) This future read i call lookahead. It contains the mutex mechanism to protect from 2 or more suitors. So what happens when 2 consumers request the same last T? Well there could be 2 potential lookaheads attempted. Only 1 lookahead T added to the T pool. What happens if the lookahead request hits the end-of-T-stream? The mutex protect checks for this.

Nov. 2009

716. Some refinements to source file/line tracings.
External file print sourcing improved, added source file/line to dynamic tracing. Cleaned up "Generated finite state automaton macros" from "c type macros" back to cweb macro.
See *EXTERNAL_GPSing* and `FILE_LINE` macros with appropriate comments.

Jun. 20014

717. Bugs in all their splender.

718. Error on "file-overrun".
Where the meta terminals 'eog' or 'eof' have no co-ordinates assigned to them and the error token being generated needs a real co-ordinte assigned to it. The **tok_can**⟨*ifstream*⟩ **operator**[] did not respect the requested token subscript when the end-of-file was reached. It always returned the 'eog' token.

Now if the requested subscript $\leq$ the container's *pos_* the appropriate token is returned from that the associated error terminal will associate to the previous real terminal returned. The container is walked backwards looking for Mr. Right.
Jan. 1/2005.

June 2008
"eof" has been end-of-the-line for `|?|`.

719. Parallel parse assumed that the grammar would do more....
than just parse and accept a single `|||`phrase. This showed up when I implemented a consolidated grammar to reduce the First set testing to launch threads.
Fix: replace *reduce* with ⟨ try reduce 264 ⟩.
Jan. 1/2005.

720. Parallel thread table aborts when program winds down.
This is a Microsoft problem as it's a simple template of map of thread strings and list of current threads available.
Jan. 1/2005.

721. `|+|`and end-of-container.
Ahh the Ides of March — what do u do when the "all shift" facility is on and u reached the "eog" or "eof" token: the end of the container? Originally I turned off the "all shift" facility and returned without executing the *all_shift* procedure if present in the configuration state. Overruns in any context are not liked. Well an improvement to this situation is to turn off the facility and still execute the *all_shift* if its present in the state's configuration. This allows the grammar writer to use this facility as an error handler.
Mar. 15/2005.

722. Test availability of `BIT_MAPS_FOR_SALE__`.
Finally getting around to refining the constraints by adding an extern indicating the total number of words for sale. When bit maps need generating — just-in-time manufacture per fsm state calling threads, the global `BIT_MAP_IDX__` is the accrued number of maps already created. It is this value that is measured for overflow against `TOTAL_NO_BIT_WORDS__`. See ⟨ determine if there is a bit map gened for state. no do it 213 ⟩ for implementation. A thrown error will be generated.
Apr. 10/2005.

723. Monolithic grammar's *start_token* should be set in constructor.
This error showed up when a standalone grammar was calling out of its first set a thread that should have run and didn't. The grammar highlighting the error was properly programmed but used the *start_token* procedure as a reference to set the error token co-ordinates. This type of error means either Yacco2's Linker did not generate properly its first sets, or the grammar writer did not regen the first sets using Linker after adding or subtracting terminals from the Terminal vocabulary.

Now one can set it 2 ways: by calling one of these procedures *start_token* or *current_token*.
May 10/2005.

724. Mismatched file number associated with error token co-ordinates.
Well this is just a dumb error! Like all others.
History:
To support nested file includes, 2 globals were used: `FILE_CNT__` and `NESTED_FILE_CNT__` to be efficiently clever. How so? I did not want to push, pop, and pant a stack. As new files were being processed, their literal names were kept in a map: file number and its description. Of course this could be a vector but my file number starts from one due to my bias on counting; I'll stick with the bias but fiddle the vector after this.

Now `FILE_CNT__` is an incrementing number while `NESTED_FILE_CNT__` is the nested level of includes. U guessed it the file number being associated with the error was the nested level and not `FILE_CNT__`. So just stack the `FILE_CNT__` at time of file processing and use the stack depth to guard against run away file recursion.
19 May, 2005.

725. Validate accept message against the new lookahead token position.
With experience, this reality check is not needed. Why? Error tokens can be returned from a thread with no consumption of the token stream occuring. The check came about when threads were being developed with the assumption that tokens returned consumed the current token stream which is not the case as one could post process tokens and forward post an error past the current token position or re-align the error outside of the token stream being read.

Now with a more creative approach to error handling and threads working properly, this check is too restraining. So beware as it can still happen.
26 May, 2005.

726. Linux bug — dropping namespace yacco2 :: on extern "C" referenced objects.
The **yacco2** namespace wrapped the below globals to manage threads. They get defined by Yacco2's linker. Now the shaker: originally I referenced these globals by using "C" extern. I used this approach to indicate that other languages could get a hold of them though the real use of extern "C" is for functions and the order of parameters pushed onto the calling stack. Unfortunately when porting **yacco2** to Linux these globals were not resolved by the regular language linker. The *wthread.cpp* code that referenced them compiled but emitted object code without the **yacco2** :: prefix.

1) extern "C" void* `THDS_STABLE__`;
2) extern "C" void* `T_ARRAY_HAVING_THD_IDS__`;
3) extern "C" void* `BIT_MAPS_FOR_SALE__`;
4) extern "C" int `TOTAL_NO_BIT_WORDS__`;
5) extern "C" int `BIT_MAP_IDX__`;
6) extern "C" **CAbs_lr1_sym*** *PTR_LR1_eog__*;

The fix: drop the "C" from the above extern statements. The object code now contains the **yacco2** :: prefix to these globals.
25 July, 2005.

727. Why me the ginea pig using other C++ compiler foibles?.
Linux ugh what's it good for? absolutely... as the song goes. The out-of-the-box C++ compiler generates unreolved references that are due to its template processing. Going thru g++ to assembler output only and looking for the undefined references from their STL and using the "nm" facility to see an object's symbols just doesn't help.

So the moral of this story is to try another compiler like Intel C++? or should I become involved with the free-open-source movement. For now my time is limited and so I will take the first option.
31 July 2005

728. MS C++ problems.
While converting to the dynamic approach to tracing, MS C++ compile hit the wall. It's symbol table management got confused in symbols that had common prefixes. Enough of my rants — detour no: xxx. At least I can still keep going instead of the more fundlemental problem posted about Linux and the unresolved ctors from template instantiation.
5 Aug. 2005

729. Regular parse and no input container: just parsed the empty language.
To support grammars as logic sequencers, i forgot to force a *current_token__* = **yacco2**::*PTR_LR1_eog__*; against the current token within the parser ctor when no input token container inputted. Even though there is no token consumption taking place, the parser starts things off by fetching the first token. If there is no token present, the ctor of the parser does not set up for parsing: parse stack etc, but exits as if an empty language had been parsed.

Correction:
In this case there is no token so i force the meta terminal *eog* to indicate the end-of-the-token-stream: a bit of a hack as regular parsing expects to receive its input from a token container but works as there is no token consumption by this particular grammar. This approach represents properly the empty language string when the grammar / parser consumes the token stream.

Observation:
As this is a very simple correction, why wasn't it programmed properly? Again the forest versus the trees situation. Local patch without overall assessment of how parsing requires a token. Now i'm being hard on myself as it was caught with my 1st test try but the observation still holds.
16 Aug. 2005

730. MS 7.0 heap delete bug....
I commented out the delete statement so that things at least work.
31 Oct. 2005 Goulish wonders...

731. MS 7.0 bug pranks.
For now bypass the *delete_tokens* request by returning immediately out of the routine.
31 Oct. 2005 wonders never cease...

732. Intel C++ release 9.
History:
Well, Intel's VTune is an excellent product that works first time. So from this experience and my problems with Red Hat's gcc compiler weaknesses of not compiling proper code, MS compiler 7.0 having little displays of irregularity lead me to try out Intel's compiler products particularly when Apple is endorsing their chips — chip wars with salt and vinegar? Well the install was easy and the anticipation high as to performance, optimization, and space. Crunch crunch crunch — that's the sound of the man ... Enough of my mental droppings... mumblings in karaoke. Hear's the scoup (intended): The compiler is approximately 3 times slower than MS compiler.
Code bloat is in fat city — 5.5 times bigger. My program is 675k using MS versus 3350k for Intel

The killer, the code produced does not handle a multi-threaded program and its contexts. It loses its proper thread run context. This did take place in Visual Studio 6.0 but they corrected this in release 7. As O_2 starts with no threads — on demand, the thread table of workers grows dynamically according to jit source context. Now the lost context, when a thread finishes it work, it sets its working status back to waiting-for-work. This setting does not happen with Intel's version of O_2. So the thread table keeps growing to approximately 2k threads created and then the program goes into a deadly wait state where all parties are politely nodding.

Upon debugging this in 2 ways: log all the events textually (let's hear it for my tracings: all events turned on — messaging between events, arbitration, tokens fetched, etc) and use of Intel's source code debugger, 2 things came out: the Intel debugger gets lost upon single stepping the source code for *set_waiting_for_work* and the smoking gun displayed its evidence as more common threads got created like *eol* where they were always busy even after their completion.

All this in 3 hours of high expectations to the sobering truth that C++ compilers are gum and shoe laced together in a top-down affair. Now Sunday 4 December, my clean up to bring me back to living with MS C++ and its little tantrums. At least it compiles fast, and my program runs in release mode. In debug mode, MS C++ has a bit of a problem with its memory re-cycling at program-exit time but this is now tolerated as there is nowhere to go for me at present. Hey what about Apple? i'll see how they do regarding top-down compiling. What about HP/Compaq/Dec? It worked 2 years ago so my porting of the Pascal translator will be the test with HP's new STL.

Alas i'm becoming more convinced of formal methods to compiling. This certainly saddens me a lot in year end 2005... about the Intel's state of affairs regarding compiling? or was it just their C++ implementation? I just don't know as the song goes but C++ certainly is a dog of a language to get right particularly when porting to different platforms exposes different compiler weaknesses. Wait till the meta-language crew start exhorting their virtues. Just try single stepping those songs!
4 Dec. 2005

733. Apple's cough in handling template definition.
See *Sour Apple on***template** *definition* for an explanation of why the slight arberation and work around.
13 Dec. 2005

Apple's response was fast and polite. They quoted the C++ Standard showing that this was left to the implementors and that their interpretation was appropriate. Upon reading Standard, they are right. The others (Microsoft, Intel, HP) use a more general approach and in my opinion would be the direction i would take dealing with glorified macros but kukos to the Open Source implementation. My correction was minimal to place all referenced variables before the defining template shell.
Feb. 2006

734. HP Alpha C++ "this" object mis-address.
See **worker_thread_blk** *initialization* : *threaded grammar*.
The launched thread places the **worker_thread_blk** "this" pointer within the *Parallel_thread_table* for thread reuse. Unfortunately the address of this object is not the same as the address within the containing grammar's parser object. Apple and Microsoft got it right!

The fix:
As the parser object containing it is also passed for tracing purposes, i now fetch its address thru the parser's object.
10 May 2006.

Take 1.329...
The problem was $ctor()$ producing a temporary variable that became $ctor\ (\ ctor(x)\ \&\ )$ in the initialization list of a defining ctor. Eg, box A contains box B where box B has only B(x) ctor. $A::A() :\ b\_(B(x))\ \{\}$; is the problem.
The ctor of box B in the list produces a temporary variable and C++ creates an implicit default ctor of $B()$ and an implicit copy $ctor\ (\ B\ \&\ )$. Why did u not just program $A::A() :\ \ldots\ b\_(x)\ \{\}$;? where the argument to the $b\_$ variable in the list is a regular ctor declaration? U got it, this is circa code of 1998 where the C++ compilers were not so good and that was the only way to initialize the variable in the list. Now 3 flavors of MS C++ compilers, 1 old Alpha compiler, and Apple's compiler morphed the code seeing that a temporary variable is not needed and respected the old way of compiling. Alas the vagaries of the past the present the future.
20 July, 2006

735. Rule reuse but forgot to remove the "AD" from each grammar.
For speed, the mallocing of rules is too expensive so i calculated its re-use count. See *rules_use_cnt* grammar on how it's done. The push / pop of the parse stack's symbols having each rule's "AD" auto delete attribute turned on got deleted every time it was popped. Consequence: any reference to the rule became a ghost reference.

Solution: just remove the attribute declaration from each rule within their grammars.
Nov. 2007

736. Recursion on "Procedure call" of a thread.
Ugly things happen as the thread's cloned "procedure call" is **not re-entrant** due to ctor / run / dtor overhead. Its fsm table is global and can only support 1 call at a time. This is a design decision for speed reasons. Needed is a recursion detection table *Parallel_thread_proc_call_table* to register call attempts for all threads. When called as a procedure turn on the use and remove the registered use after it has return from the call. This table is mutex protected unfortunately but necessary due to parallelism.
Apr. 2008

737. VMS misqueue on Mutex Recursion and Pthread stacksize.
Ugly things happen as the thread is activated. The pthread's default stack size *pthread_attr_t* variable does not set the stack size properly. causes the pthread library to throw up. So explicitly set it using the *pthread_attr_setstacksize* procedure before the pthread create.

The second more serious issue is its detection of what it thinks is recursion on a single use mutex. It's reaction is down right violent — spews of core dump and attempts at calming the hoard with information messages of potential inaccuracies. This reaction is illussionary as this is not so. Each thread or its singular procedure partner has their own private copy for the control message Mutex and Conditional variables. This was tested on Unix out-of-the-box Pthread library variants (Sun and Apple) without this hacking or is it gagging? So just remove the "procedure call" optimization for VMS and make it a thread call.
Aug. 2008

738. `|?|` used instead of `|+|` making it a perpetual motion machine.
Guard against `|?|`as it does not advance *get_next_token* so the parse keeps on going dancing at the same token spot: this is perpetual motion machine — swap file eventually fills up and Boom Ca Boom. Sometimes the grammar writer is using improperly the `|?|`instead of an epsilon rule. So how to detect this? Well if the *has_questionable_shift_occured__* has been previous set, then stop parsing instead of aborting. Should i message or not to message that is the question. I'll message the errant grammar and parse stack state where the problem was detected. The grammar writer should use the `|+|` symbol.
Patched ⟨ try various shift types. if executed go to process next token in token stream 253 ⟩.
Sept. 2008

739. Rule reuse Code emmission did not store the newed rule in its recycle table.
I did not store the newed grammar rule in its recycle table. This was brought out using a marvellous tool call dtrace from Sun. Well the thought was right but my details were wrong — like the kid who runs ahead in thought while learning to crawl.

The other part to rule recycling is making sure local grammar rule's variables are re-initialized as the past dribbles will effect the present. Speak clearly boy! Example, in *la_expr* grammar *Ra* and *Rt* rules contain the local set *fset_* variable. This holds the terminal in the lookahead expression so that set "union and difference" expressions can take place. Having a recycled rule with this set not cleared will contain its past history. This is the cost of an optimization: 25% improvement so be forewarned.

Dec. 2008

740. String template container did not set the *eof_pos_* variable and random boom.
The sky is falling. As the string container didn't set this variable, random droppings other than EOF meant that at least a first read on the string container would take place. Well u guessed it. As it was never read the eof symbol was not set and so nil pointer on the returned token. At least the file container set *eof_pos_* properly. Alas just sloppiness Dave and a swill to u.
Mar. 2009

741. TOKEN_GAGGLE's virtual table access [] operator not respected.
This showed up in an xml/message dispatcher system written for VMS/Alpha. The "Error queue" being parsed was getting an array out-of-bounds error when the end-of-token stream was reached. THE **TOKEN_GAGGLE**'S ACCESS [] PROTECTS AGAINST THIS. But the internal container used aka STL's array container was being called directly. This problem only occured in c++ VMS/Alpha port. Sun, Apple, Linux flavours all worked by respecting the virtual table of the abstracted **tok_base**. So tighter checks within the *get_next_token* Parse method is done ensuring the *current_token__* is always set on an empty container or any of the overflow checks.

Originally *current_token__* was set only when the overflow was first detected. As a post evaluation, the Parser "Error queue" which was originally declared as a **TOKEN_GAGGLE** is now declared as an abstract **tok_base** just like the other containers Supplier, Producer, and Recycle bin. This allows the language designer to use a different Error container like trees. In conclusion, though not a bug but a porting weakness, this modification makes the Parser more flexible. So Dave your fixed Error thoughts are virtualized.
Oct. 2010

742. Procedure calls in VMS revisited: thread versus procedure.
Revisited the optimization on procedure calling of grammars when only 1 grammar is to be called. This is a major improvement over thread calls. Well this is the scoop. Make sure that the stack paramater to the VMS linker is adequate or not fun abortive things happen within a called thread that u know works. This happened to a command that was parsed properly using the same called thead while the other command to be parsed aborted.

Second, make sure thare are no overruns in a std type container happening. Somehow VMS only has a problem guarding against an overrun which is properly guarded against within O_2's library. For now the code in ⟨ request threads to work 384 ⟩ has renamed the conditional variable `VMS_` to `VMS111__` so that it is not used. I'm keeping it there as a reminder to possible future reguritations.
Nov. 2010

743. Size of tree container — number of items in container.
What is the size of the tree container? It depends whether its end-of-tree has been reached. So put a conditional test in its size method: return `MAX_UINT` if tree walk is still in process. End-of-tree reached then return the size of its internal container.
Feb. 2011

744. *Find_reduce_entry* current token not found.
My to my stupidity. The searching for the subrule reducing was optimized. Not to get into my stupidity but the meta symbols were found before the next subrule's LA set was searched. The correct search is 2.5 passes — find the current tok against the potential subrules. Followed by meta symbols against a new round of potental subrules list , and then the last gasp `|?|`is search if the previous passes not met.
Nov. 2012

745. Date macro use — Apple LLVM C++ compiler.
Version 5.1 (clang-503.0.40) (based on LLVM 3.4svn).
This is caused when the version literal per O_2linker and O_2 is built. See "runtime_env.w" file for details.
Must split lines or delimit by spaces when concatenating the macro '__DATE__' by bounded literals.
Example: "xxx" __DATE__ "yyy" //works cuz spaces
Without the spaces the compiler thinks its a template mistake with this error:
No matching literal operator for call to 'operator"' __DATE__ with arguments of types 'const char*' and 'unsigned long', and no matching literal operator template.
Apr. 2014

746. Eog symbol not gpsing on external file and internal line no.
Here's the stik. I was playing around with the Pager_1.lex grammar. To make it interesting, the T vocabulary files were changed. By mistake the Error T vocabulary file did not have a close off brace: }. So the right error was thrown but the file co-ordinates were 0 and did not reference the external file!

Looking at the **tok_can**⟨**std** :: *ifstream*⟩ container, the end-of-file indicator was passing the appropriate file references. So what the heck? Well to the rescue, yacco2::YACCO2_T__ tracing of Tes. In all the gory details and low and behold the "eog" had no external references. Well the culprit was *map_char_to_raw_char_sym* that draws from its premade raw character pool and makes a T symbol. It was passed the appropriate external file's co-ordinates but...

To quicken raw character mapping to a **CAbs_lr1_sym** symbol a premade *PTR_LR1_eog* symbol was just returned without setting the passed-in file co-ordinates.

Man Dave you sure r a winner!
May 2014

747. Cleaned up Arbitrator's `YACCO2_AR__` tracings.
2 items corrected:

1) misplaced ⟨ release trace mu 390 ⟩; in `TAR_2` walking accept-queue. The 5 computer nerds waiting
2) commented out for `TAR_1` − 3 macros use of *trace_parser_env*

Oct 2014

748. 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 inapproproiate 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 specifiying the errant T to catch. This last option can be very daunting when one has 500+ T 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 burried 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
Rule Rqueshift → `|?|`...
will catch the error with appropriate syntax directed code directive

Caution: The ranking of meta-terminal shifts: 1 and a 2 and a 3 —`|?|`, `|.|`, `|+|`
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-ambigous 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 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: `|+|`or `|.|`. For the moment i have not issued an error message on this situation.

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.

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 parsing of the active grammar
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 successfull. 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.

Pinpointing where the error occured in the source file
Built into O_2 is the facility to tag each T with its approriate source file's GPS — filename, line number, and character position. These co-ordinates 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**.

Some subtleties on making the errant T fire off the error catching syntact 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 syntact 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 parse stack, O_2 emits within each subrule's c++ code the stack frame with each subrule's symbol string assigned to "*sf→pxx__*" where xx is the symbol's string position. This is the difference to `|+|`: `|+|` depends on the lookahead set to reduce. Now what then is the advantage to using `|+|`? One can test its under-its-hood T's enumerate value and then take error action or stop use of the `|+|` facility that allows the grammar to continue parsing up to the "start rule". As it's a wild symbol shifter, it really lowers the grammar's parse tables sizes and eases the grammar writer's typing.

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 successfull 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 a subset / superset competition, or an accept and error combo is quite acceptable and for 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. `|+|` demands 2 be subtracted as the current T is the new lookahead T. So u've been warned.

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 back to the user 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 successfull "why complete the parsing thru to 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_abort_parse(*true*)
set_stop_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.

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 stopping of the parse immediately. So u'll see in some the suggested grammars *set_use_all_shift_off* method being called to get out of this perpetual motion and possiblely continue up the parse chain to the "start rule". Here is a list of some O_2 grammars having error handling and premature stopping of a parse to learn from.

1) *o2_lcl_opts.lex* and called thread *o2_lcl_opt.lex* — command line parser
2) *la_express.lex* — *set_abort_parse*(*true*) thread's la expression parser
3) *c_string.lex* — semantic example stopping a parse and programmed fsa

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 grammar *o2_lcl_opts.lex* of a monolitic 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 find/grep/xargs combo to settle your appetite against O_2's grammars.

The last word, amen and happy parsing.
Remember that the normal flow of errors should be placed into the "error queue" and then post processed to report its findings. `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 in an abort state. But each grammar does a resetting to a clean slate for its next round of calling either by "procedure call" if no nesting calls of itself is occuring or by the heavy thread call. Hygiene is important so the cat washes itself for the next eating.

749. Index.

WLIBRARY