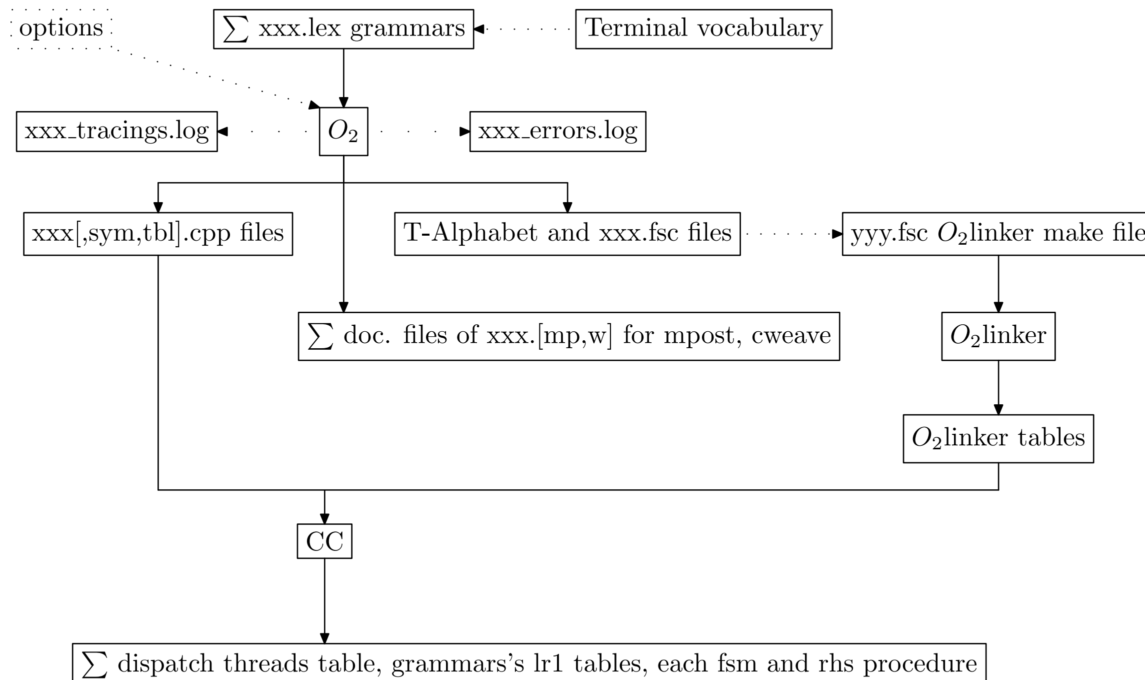
1. License.
Gened date: January 30, 2015
Copyright © 1998-2015 Dave Bone

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 O_2 — Yacco$_2$'s nickname.
The compiler / compiler's formal name is Yac_2o_2 but call me O_2. Yac_2o_2 can be morphed many ways; here are some hints: sound of a cold, molecular contortions. Do your own expletives.

3. Component overview running O_2.
I use a ".lex" extension to distinguish a grammar file. This is not hard coded. You can choose your own memory mnemonics for any of my files. The ".T" file extension identifies the Terminal vocabulary. Its components are described later in the document. The Lrk and Rc terminals are pre-assembled and reside in the "/usr/local/yacco2/library/grammars" account. My original thought was to allow the compiler writer to experiment with his own terminal definitions for all classes: LR constants, raw characters, errors, and terminals. From experience, only the last 2 classes are local to each language being defined.



Please note, the "header" files are absent from the diagram due to space constraints. The salient ones to note are:

1) the enumeration of the vocabulary's symbols
2) headers for fsm, and the terminal classifications: lrk,error,rc,and T

Per compiled "xxx.lex" grammar, the 2 status files "xxx_tracings.log" and "xxx_errors.log" hold the text-only compiled results lodged within the local grammar's folder. "T-Alphabet" gets gened when the "-t" option is inputted to gen the Terminal vocabulary. It is a file by defined order of all the terminals's literal names used as comments against the outputted lookahead tables to make sense of their compressed set definitions. Its file name is built from the grammar's "T-enumeration" construct using its filename and adding a ".fsc" extension. O_2^{linker}cross checks the number of terminals defined in the "no-of-T" value in each grammar's "fsc" file against this file. Out-of-sync values indicates that new terminals have been added to either the "error" or "terminal" class terminals vocabulary without gening up the grammar with the "-t" and/or "-err" option(s). The "yyy.fsc" file is the grammar writer's handcrafted file containing references to these "xxx.fsc" files and the T-Alphabet file for $O_2^{linker}$to compile. Its name can be anything but i use the ".fsc" as a memory jog. Please see O_2^{linker}'s documentation on its raison d'être and make file comments.

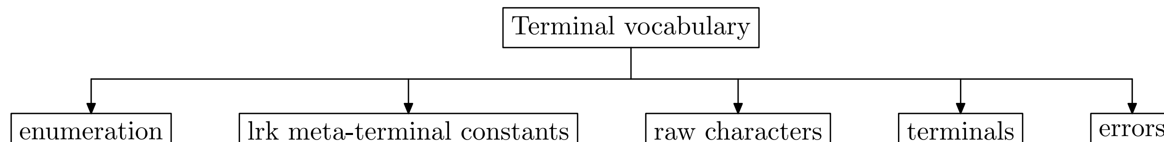
4. Tracing facilities.
Some of the more important tracing facilities are as follows where their mnemonic replaces the "xx":

TH — dynamic trace of the grammar's parse stack when its "debug option is true
T — trace terminals fetched across all grammars
AR — trace arbitration when grammar's debug switch is true
MSG — dynamic threading messages between the co-operatives

Please see O_2's library documentation concerning each tracing variable when set to 1 within your program by the programmer: "yacco2::YACCO2_xx__ = 1;" starts their specific scribblings. There are other less important trace variables not listed above. The turned on O_2's library trace facility will log to "xxx_tracings.log' file where 'xxx' represents the grammar being parsed without its extension. As the "xxx_tracings.log" is text-only content, this allows the use of a general text editor to browse its material. If the editor has indigestion due to its volume, a script can be written to postprocess it for study by the "sed" / "grep" combo or using just the "split" utility.

5. Grammar anatomy.
The grammar is composed of your traditional components: start rule, non-terminal vocabulary, terminal vocabulary, and 2 additional parts: fsm and syntax directed code. "fsm" (short for finite state machine) is a packaging agent. It houses all the grammar's software generated parts along with the c++ syntax directed code within the grammar associated with their directives. These directives are local to the grammar's rules, subrules, and possibly the grammar's start-run-finish sequence that is handled within the fsm. "fsm" supplies the grammar's c++ namespace, class name, and filename prefix to output the components to.

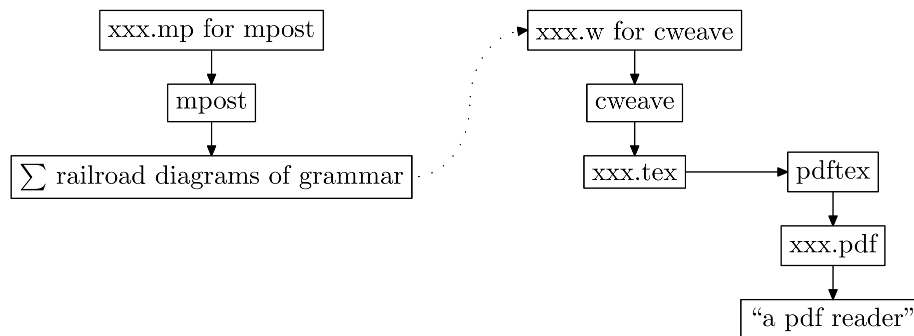
6. Terminal vocabulary.
From the diagram below, the "enumeration" component is a packaging agent that receives the outputed enumeration definitions for each terminal class. The counting scheme uses the natural numbers starting from 0 listing the "lrk" constants, followed by each of the other components's terminals. The last component "terminals" is your regular terminal definitions that gets assembled from the lexical or syntactical passes, and possibly out into etherland of abstraction. All vocabulary elements are tagged this way. It is the glue to all the emitted tables. For the record, each grammar's non-terminal vocabulary (rules) are enumerated after the terminal enumeration count and are defined within the grammar's fsm class definition. The rules's subrules are also enumerated and defined there. They are not dependent on the Terminal vocabulary.



7. Overview of generating the grammar's pdf and postscript(ps) documents.
There are 2 generated documents using the "-p" option emitting "cweave" content and associated "mpost" diagrams for compilation:

1) grammar with its syntax direct code,
emitted O_2linker file, and gened lr1 state network
2) various cross references against the grammar, and lr1 state network
  - symbols used from each rule's subrules symbol string position
  - additional information supporting the lr1 state network in the 1st document
    • each lr1 state's rules follow sets
    • reducing states subrules with references to their contributors' follow sets
    • global lookahead sets with their yield used by the parse reduce operation

The below diagram shows the manufacturing of a grammar's document.



Also for each "pdf" document generated there is a postscript document to remove the dependency of a "pdf reader" and its "gui" interaction when wanting to spool the document for print. For example on my Sun Solaris, the program "pdftops" takes a "pdf" document and creates an equivalent "ps" document. Spooling it to a print would use the command line "lp xxx.ps".

8. A sample O_2 script where the options are described.
O_2's input data template is [*options*] filename where options are optional as they have preconfigured values. Here are the switches that can be inputted to O_2 using the Unix approach to turn on the specific option. Each option must be inputted with its own − sign.
Options: if they are not present, do not generate

1) -t — generate the Terminal vocabulary
2) -err — generate the Error vocabulary
3) -lrk — generate the lr k vocabulary: Deprecated not supported
4) -rc — generate the Raw characters vocabulary: Deprecated not supported
5) -p — generate the grammar's documents

Points 1 and 2 are usually stable and do not need to be gened. Do so when these vocabularies have been modified. Don't forget to regen all the grammars and re-run O_2linker to reprocess the "fsc" files if the number of Terminals in the vocabularies has changed as all the Lookahead sets are now different along with the enumeration scheme that ties them together. Points 3 and 4 cannot be used as they reside in "/usr/local/yacco2/library/grammars" pregened. I included them as a memory jog to my experiments; if u try to input these deprecated options u'll get an error message. Other options were experimented with but found boarderline marginal: gen namespace, gen the grammar, and turn on debug of grammar instead of using the editor cycle to modify the grammar to be traced. Now namespace and grammar are always generated, and hello editor. So out damn spot.

Here is a batch command file that runs on a Microsoft's NT/XP desktop. The same can be done within a "Unix" flavour script language like "Bash". Though it does not illustrate a conditional test as to whether the script should continue when the grammar is faulty, O_2 returns a 0 to indicate a healthy grammar and a 1 to indicate a sick grammar: The gory details are in the error log.

```
 1:  rem file: o2.bat
 2:  rem compile O2 grammars
 3:  cd \yacco2\compiler\h2o\release
 4:  @echo ON
 5:
 6:  o2  -p -t -err /yacco2/compiler/grammars/enumerate_grammar.lex
 7:  mpost enumerate_grammar.mp
 8:  cweave enumerate_grammar.w
 9:  pdftex enumerate_grammar
10:
```

The above example uses line numbers delimited by ":" at the start of each line for commentary purposes. Line 3 sets the directory where O_2 resides and repository for the temporary files from O_2, "mpost" that draws the grammar diagrams, while "cweave" generates the "xxx.tex" file for "pdftex" who completes the document for an "Adobe" reader program: for example "xpdf" of open source or Adobe's reader. "cweave" is one of the programs by Donald E. Knuth and Silvio Levy from their book "The CWEB System of Structured Documentation". Go to the web site "www.tex.org" for more information on how to obtain "CWEB". The same comments apply to "mpost" written by John D. Hobby of 'Bell Labs". This is a remake of "MetaFont"language / "MetaPost" program by Donald E. Knuth. These programs are grrrreat! More people should be using them. The emitted grammar files get placed in the same directory of the inputted grammar to O_2.

Line 6 runs O_2 with its inputted grammar file "/yacco2/compiler/grammars/enumerate_grammar.lex" and switches to gen up the Terminal and Error vocabularies and gen a printed set of documents. O_2 also generates the documentation files "enumerate_grammar.mp" and "enumerate_grammar.w" files. Lines 7–9 are command lines to create the output document. In the example, "enumerate_grammar.pdf" is the final file document for printing. "enumerate_grammar.xx" are figure files generated by mpost from file "enumerate_grammar.mp". These files are referenced in the "enumerate_grammar.w" file by cweave who produces an "enumerate_grammar.tex" file for program pdftex. All this to say that there can be many generated files before the document is complete. Please note the other cross reference document is not shown but follows the same run pattern.

9. Some definitions.

Non-terminal:
This is your normal grammar definition. I interchange this term with "rule". They are the same. Depending on the context, i also use rule in the same sense of a grammar's production. To refine the context, the term "subrule" indicates one of a rule's productions.

Subrule:
Equivalent to a grammar's production. It is one of a rule's right-hand-side string of symbols drawn from the non-terminal or Terminal alphabets. The string can be empty indicating epsilon.

Please see the mavelous book "Formal Languages and Their Relationn to Automata" by Hopcroft and Ullman for a complete discussion on grammars and their makeup. Excellent reading for a 1968 vintage on automata.

Here are some basic definitions used by my lr1 generator.

First set:
Please see *first_set_rules.lex* grammar for a more thorough discussion. First set is the set of terminals that begins a string of symbols. When the symbol is a rule, then all its subrules contribute to the first set. This is a recursive definition as the rule's subrules can also bring in other rules' subrules string of symbols that contribute to it. If the string's start symbol is a rule and its epsilonable, then its right neighbour also contributes. Again if its a rule and epsilonable its right neighbour is a contributor: ahh recursive definitions.

Follow set:
The rule's first set of production strings to the right of a lr state's configuration. Here is a simple arithmetic grammar to illustrate follow sets for the "Closure-only" state where all production strings have their configuration position at their very beginning illustrated by ".". I use a form of Dewey decimal notation to reference the production's configuration. For example "E.1.2" means the production of rule "E" referencing subrule 1 of its second symbol is being refered to. In this example below the referenced symbol is +. How follow sets are arrived at is discussed in "Overview of O_2's state generated components".

Rule		*subrule's symbols*
S	$\rightarrow$	.E $\perp$
E	$\rightarrow$	.E + T
	$\rightarrow$	.T
T	$\rightarrow$	.T $*$ F
	$\rightarrow$	.F
F	$\rightarrow$	.(E)
	$\rightarrow$	.id

Rule	*follow set*	*R.SR.Pos of follow set*	*with transitions*
E	$\perp$ +	$\perp$ by S.1.2, + from E.1.2	$\perp$ +
$\uparrow$		E.2.2	
T	$*$	$*$ by T.1.2	$*$ $\perp$ +
$\uparrow$		T.2.2	
F	$\emptyset$		$*$ $\perp$ +

Table of follow sets for the "Start state" of the above grammar

There are 3 subtleties that are watched for in the follow set calculation:

1) rule symbol — use its “first set”
2) epsilonable rule symbol — continue to next symbol in follow string for assessment
3) end of symbol string reached — transition

Point 3 requires some explanation. Its condition means that the rule’s right-hand-side has been consumed (or is epsilon) so what’s it follow set? Nothing? No it’s the subrule’s rule that spawned it that provides more follow set context. This context resides in the “closure” state of this rule. So now there is a transition to this rule’s follow set. This is the transitive closure of spawning contexts. The Table of follow sets shows these transitions with the ↑ symbol. Epsilon rules are chameleon in nature: they supply their first sets and also disappear and so u must continue to the next symbol in the follow string to complete the follow set while observing the end-of-string condition to follow its transitions.

10. Catalogue of O_2's files.

Cweb Documents:

1) Yac$_2o_2$ parse library
2) O_2extern — external routines
3) Yac$_2o_2$stbl — symbol table

O_2's Input files to *cweb*:

1) *o2.w* — master file that starts things off
2) *intro.w* — introduction
3) *defs.w* — basic definitions to gen lr1 network
4) *prog.w* — O_2 cweb code
5) *bug.w* — confessions
6) *o2_defs.w* — details
7) *includes.w* — bring in those grammars for the parsing
8) *o2externs.w* — external routines

cweb generated files:

1) *o2.h* — compiler definitions
2) *o2.cpp* — O_2 program
3) *o2_defs.cpp* — structure implementations
4) *o2_externs.h* — global definitions used across O_2's source code

O_2's generated files where xxx is the grammar's name being compiled:

1) *xxx.fsc* — grammar's first set confessions for Linker
2) *xxx.h* — grammar's header file
3) *xxx.cpp* — automaton code
4) *xxxsym.cpp* — automaton symbols
5) *xxxtbl.cpp* — automaton's state definitions

Yac$_2o_2$ library memorabilia:

1) yacco2 — library namespace
2) "/usr/local/yacco2/library" — yacco2's library directory
3) < *yacco2.h* > — Yacco2's library header file
4) "library directory/xxxx" - xxxx is the debug or release of the object library

Dependency files from Yac$_2o_2$ sub-systems:

yacco2.h - basic definitions used by Yacco2
yacco2_T_enumeration.h - terminal enumeration for Yacco2's terminal grammar alphabet
yacco2_err_symbols.h - error terminal definitions from Yacco2's grammar alphabet
yacco2_characters.h - raw character definitions from Yacco2's grammar alphabet
yacco2_k_symbols.h - constant terminal definitions from Yacco2's grammar alphabet
yacco2_terminals.h - regular terminal definitions from Yacco2's grammar alphabet
∗. *h* - assorted grammar definitions from Yacco2 to parse
o2_externs.h - external support routines for O_2

Grammars

pass3.lex — lex and syntactic phase of grammar
la_expr_source.lex — lexical phase of lookahead expression
la_expr.lex – syntactic phase of lookahead expression
enumerate_T_alphabet.lex – logic grammar to assign each T a number from 0..n
epsilon_rules.lex – grammar determines epsilon per rule and pathological conditions
first_set.lex – logic grammar to calculate each rule's first set
prt_fs_of_rules.lex – logic grammar to print each rule's first set
enumerate_grammar.lex – dump aid: enumerate grammar's components

Globals

`LR1_STATES` — list of gened lr1 states
`LR1_COMMON_STATES` — common states map having same vectored into symbol
`START_OF_RULES_ENUM` — used in shift / reduce conflict evaluation

Comments:
My external routines use the all upper case approach to names. I know it's like shouting but it clues the reader where the heck the routine comes from. I could have tempered the all caps approach to a capital letter but i'm myopic and becoming visually golden in age. So my excuses to the reader for this tasteless approach.

11. O_2's language.
There are 3 languages that are actually parsed: 2 in preparation — command line and its contents, and the grammar file. A grammar is divided into 4 parts:

a) Finite automaton definition — basic statements about the grammar
b) Parallel parse that defines a threading grammar
c) Terminal vocabulary: errors, lr k, raw characters, and terminals
d) Rule definitions

```
 1:  /*
 2:  FILE:    eol.lex
 3:  Dates:   17 Juin 2003
 4:  Purpose: end-of-line recognizer
 5:  */
 6:  fsm
 7:  (fsm-id    "eol.lex",fsm-filename eol,fsm-namespace NS_eol
 8:  ,fsm-class Ceol
 9:  ,fsm-version "1.0",fsm-date "17 Juin 2003",fsm-debug "false"
10:  ,fsm-comments "end of line recognizer")
11:  parallel-parser
12:  (
13:    parallel-thread-function
14:      TH_eol
15:    ***
16:    parallel-la-boundary
17:      eolr // - "x0a" more efficient to use |.|
18:    ***
19:  )
20:  @"c:/yacco2/compiler/grammars/yacco2_include_files.lex"
21:
22:  rules{
23:  Reol AD AB(){
24:    -> Rdelimiters {
25:      rhs-op
26:        CAbs_lr1_sym* sym = new T_eol;
27:        sym->set_rc(*parser()->start_token(),*parser());
28:        sym->set_line_no_and_pos_in_line(*parser()->start_token());
29:        RSVP(sym)
30:      ***
31:      }
32:  }
33:
34:  Rdelimiters AD AB(){
35:    ->  "x0a"
36:    ->  "x0d" |.|
37:    ->  "x0d" "x0a"
38:  }
39:  }// end of rules
40:
```

The above source listing is an example of a threaded grammar. Starting each source line is a line number suffixed by ':' present only for discussion purposes. Line numbers 6–10 defines the fsm component. Lines 11–19 indicates that the grammar is a thread. Though the terminal vocabulary definitions are hidden by

line 20, it illustrates the file include feature of O_2. Lines 22–39 are the rule definitions. Each grammar's section has a defining keyword like "fsm", "parallel-parser", "rules" that introduces the part being defined.

12. C macros.
Originally there were conditionally defined trace variables that controlled the inclusion of trace code. This was a pain-in-the-seat so now they are global variables that test their values. I felt the slight speed bump merited the facility without the combinatorics of libraries needed for distribution. *YACCO2_define_trace_variables* macro defines these global variables used by O_2's tracing purposes. U can roll your own or just include the macro in your code. These variables are dormant until their values are not zero. Without their inclusion, a linker message of unresolved variable will be regurgitated: they must be present when using the O_2 library. It's an easy way to define them within your program. Please see O_2 library documentation for a discussion on each trace variable. To activate a specific tracing, assignment a non zero value to the selected trace variable: set it to 1. Here is their catalogue:

`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

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;

Each traced output line identifies its type by the trace variable turned on. As tracing can be very very volumnious, post evaluating the output thru a Bash type filter script makes the log output manageable. I say this from experience as some editors blow up due to the size of the traced file. Names withheld to protect the innocent.

13. External routines and globals.
General routines to get things going:

1) get control file and put into O_2's holding file
2) parse the command line
3) format errors
4) O_2's parse phrases — pieces of syntactic structures

These are defined by including *o2_externs.h*. Item 4 is driven out of the *pass3.lex* grammar. It demonstrates a procedural approach similar to recursive descent parsing technique.

The globals are:

a) *Error_queue* — global container of errors passed across all parsings
b) Switches from command line parse
c) Token containers for the parsing phases

⟨ External rtns and variables 13 ⟩ ≡

```
extern int RECURSION_INDEX__;
extern void COMMONIZE_LA_SETS( );
extern int NO_LR1_STATES;
extern STATES_SET_type VISITED_MERGE_STATES_IN_LA_CALC;
extern LR1_STATES_type LR1_COMMON_STATES;
extern CYCLIC_USE_TBL_type CYCLIC_USE_TABLE;
extern void Print_dump_state(state * State);
```

This code is used in section 163.

14. Main line of O_2.

```
⟨ accrue O2 code 14 ⟩ ≡
  YACCO2_define_trace_variables( );    /* Recursion_count( ); */
  int RECURSION_INDEX__(0);
  yacco2::CHART_SW('n');
  yacco2::CHARERR_SW('n');
  yacco2::CHARPRT_SW('n');
  yacco2::TOKEN_GAGGLE JUNK_tokens;
  yacco2::TOKEN_GAGGLE P3_tokens;
  yacco2::TOKEN_GAGGLE Error_queue;
  char Big_buf[BIG_BUFFER_32K];
  T_sym_tbl_report_card report_card;
  std::string o2_file_to_compile;
  std::string o2_fq_fn_noext;
  STBL_T_ITEMS_type STBL_T_ITEMS;
  STATES_type LR1_STATES;
  LR1_STATES_type LR1_COMMON_STATES;
  bool LR1_HEALTH(LR1_COMPATIBLE);
  int NO_LR1_STATES(0);
  STATES_SET_type VISITED_MERGE_STATES_IN_LA_CALC;
  CYCLIC_USE_TBL_type CYCLIC_USE_TABLE;
  int main(int argc, char *argv[])
  {
    cout << yacco2::Lr1_VERSION << std::endl;
    ⟨ setup O2 for parsing 17 ⟩;
    ⟨ fetch command line info and parse the 3 languages 19 ⟩;
    lrclog << yacco2::Lr1_VERSION << std::endl;
    ⟨ are all phases parsed? 34 ⟩;
    ⟨ epsilon and pathological assessment of Rules 29 ⟩;
    ⟨ dump aid: enumerate grammar's components 28 ⟩;
    ⟨ determine if la expression present. Yes parse it 35 ⟩;
    ⟨ get total number of subrules for elem_space size check 31 ⟩;
    ⟨ calculate rules first sets 32 ⟩;
    ⟨ calculate Start rule called threads first sets 33 ⟩;
    ⟨ generate grammar's LR1 states 39 ⟩;
    ⟨ is the grammar unhealthy? yes report the details and exit 40 ⟩;
    ⟨ determine each rule use count 37 ⟩;
    ⟨ emit FSA, FSC, and Documents of grammar 130 ⟩;    /* ⟨ print tree 132 ⟩; */
      /* ⟨ shutdown 16 ⟩; */
  exit: lrclog << "Exiting␣O2" << std::endl;
    return 0;
  }
```

See also section 162.

This code is used in section 164.

15. Some Programming sections.

16. Shutdown.
Prints out the thread table with their runtime activity, and calls each one of them to quitely remove themselves as threads. Within Unix this is not needed as the winddown duties of the process removes launched threads: That is why it is commented out. Uncommenting it provides the run statistics for the compiler writer to view reality in terms of performance stats.

```
⟨ shutdown 16 ⟩ ≡
  lrclog ≪ "Before␣thread␣shutdown" ≪ std::endl;
  yacco2::Parallel_threads_shutdown(pass3);
  lrclog ≪ "After␣thread␣shutdown" ≪ std::endl;
```

This code is cited in section 14.

17. Setup O_2 for parsing.

⟨ setup O_2 for parsing 17 ⟩ ≡
  ⟨ load O_2's keywords into symbol table 18 ⟩;

This code is used in section 14.

18. Load O_2's keywords into symbol table.
Basic housekeeping. Originally a grammar recognized keywords by being in competition with the Identifier thread. Keyword thread only ran if its first set matched the starting character making up an identifier and keyword. Now it's blended into Identifier using the symbol table lookup that returns not only the identifier terminal but all other keyword entries put into the symbol table.

For now, only the keywords are cloned off as unique entities whilst all other entries are passed back from their symbol table with its source co-ordinates being overriden.

```
⟨ load O2's keywords into symbol table 18 ⟩ ≡
  LOAD_YACCO2_KEYWORDS_INTO_STBL();
```

This code is used in section 17.

19. Fetch command line info and parse the 3 languages.
The 3 separate languages to parse are:

1) fetching of the command line to place into a holding file
2) the command line in the holding file — grammar file name and options
3) the grammar file's contents

Items 1 and 2 are handled by external routines where fetching of the command line is crude but all-purpose whilst the command line language is specific to O_2.

```
⟨ fetch command line info and parse the 3 languages 19 ⟩ ≡
  ⟨ get command line, parse it, and place contents into a holding file 20 ⟩;
  ⟨ parse command line data placed in holding file 22 ⟩;
  ⟨ parse the grammar 26 ⟩;
```

This code is used in section 14.

20. Get command line, parse it, and place contents into a holding file. It uses a generic external routine to do this. The parse is very rudimentary. The command data is placed into a holding file provided by *Yacco2_holding_file* defined in the external library *o2_externs.h*. See *cweb* documents mentioned in the introduction regarding other support libraries. If the result is okay, set up O_2's library files for tracing.

```
⟨ get command line, parse it, and place contents into a holding file 20 ⟩ ≡
  GET_CMD_LINE(argc, argv, Yacco2_holding_file, Error_queue);
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
```

This code is used in section 19.

21. Do we have errors?. Check that error queue for those errors. Note, DUMP_ERROR_QUEUE will also flush out any launched threads for the good housekeeping or is it housetrained seal award? Trying to do my best in the realm of short lived winddowns.

```
⟨ if error queue not empty then deal with posted errors 21 ⟩ ≡
  if (Error_queue.empty() ≠ true) {
    DUMP_ERROR_QUEUE(Error_queue);
    return 1;
  }
```

This code is used in sections 20, 22, 26, 34, 36, 118, 119, 120, 121, 122, 123, 124, and 125.

22. Parse command line data placed in holding file.

```
⟨ parse command line data placed in holding file 22 ⟩ ≡
  YACCO2_PARSE_CMD_LINE(T_SW, ERR_SW, PRT_SW, o2_file_to_compile, Error_queue);
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
  ⟨ display to user options selected 25 ⟩;
  ⟨ extract fq name without extension 23 ⟩;
  ⟨ set up logging files 24 ⟩;
```

This code is used in section 19.

23. Extract fully qualified file name to compile without its extension. Used to access the generated first set control file for *cweb* documentation and O_2's tracings. Simple check, if the grammar file name does not contain a ".extension" then use the complete file name.

```
⟨ extract fq name without extension 23 ⟩ ≡
  std::string::size_type pp = o2_file_to_compile.rfind('.');
  if (pp ≡ std::string::npos) {
    o2_fq_fn_noext += o2_file_to_compile;
  }
  else {
    o2_fq_fn_noext += o2_file_to_compile.substr(0, pp);
  }
```

This code is used in section 22.

24. Set up O_2's logging files local to the parsed grammar. There are 2 stages. Stage 1 logs to "1lrerrors.log" and "1lrtracings" as the command line is being parsed — *o2_lcl_opts* and *o2_lcl_opt* grammars. It has no knowledge of the grammar file to parse. Stage 2 passed the command line parsing and the inputted grammar file name can be used to build the grammar's local O_2 tracing files. These log files are "xxx_tracings.log" and "xxx_errors.log" where the "xxx" is the grammar's base file name.

```
⟨ set up logging files 24 ⟩ ≡
  std::string normal_tracing(o2_fq_fn_noext.c_str());
  normal_tracing += "_tracings.log";
  std::string error_logging(o2_fq_fn_noext.c_str());
  error_logging += "_errors.log";
  yacco2::lrclog.close();
  yacco2::lrerrors.close();
  yacco2::lrclog.open(normal_tracing.c_str());
  yacco2::lrerrors.open(error_logging.c_str());
```

This code is used in section 22.

25. Display to user options selected.

```
⟨ display to user options selected 25 ⟩ ≡
  lrclog ≪ "Parse␣options␣selected:" ≪ std :: endl;
  lrclog ≪ "␣␣Gen␣T:␣" ≪ T_SW;
  lrclog ≪ "␣␣Gen␣Err:␣" ≪ ERR_SW;
  lrclog ≪ "␣␣Gen␣RC:␣" ≪ PRT_SW;
```

This code is used in section 22.

26. Parse the grammar.
Due to the syntax directed code not having legitimate grammars to parse it, a character-at-a-time parsing approach is used. This is a lexical and syntactic mix of parsing instead of your separate lexical, syntax parse stages. Why? I'll use a question as an answer. How do you recognize the '***' directive to end a c++ syntax directed code portion that is an unstructured sequence of characters? Well crawl at a character's pace per prefix accessment. This is why the bluring between lexical and syntatical boundaries. So walk-the-walk-and-talk of a lexical parser using recursive descent (for its single call of fame containing a bottom-up parse) tripped off by a bottom-up syntax directed code. What a mouthfull! Should mother use soap and a tooth brush to punish the child? Who is this mother anyway?

Within the *pass3.lex* grammar are procedure calls containing the parse phases. Each phase is called from within the syntax-directed-code of the recognized keyword: "fsm"," rules", etc. This demonstrates a bottom-up / top-down approach to parsing. Options are what it's all about. What's your choice?

```
⟨ parse the grammar 26 ⟩ ≡    /* yacco2 :: YACCO2_TH__ = 1; */    /* yacco2 :: YACCO2_MSG__ = 1; */
  using namespace NS_pass3;
  tok_can < std :: ifstream > cmd_line(o2_file_to_compile.c_str( ));
  Cpass3 p3_fsm;
  Parser pass3 (p3_fsm, &cmd_line, &P3_tokens, 0, &Error_queue, &JUNK_tokens, 0);
  pass3.parse( );
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
  ⟨ dump lexical and syntactic's outputted tokens 27 ⟩;
```

This code is used in section 19.

27. Dump lexical and syntactic's outputted tokens.

```
⟨ dump lexical and syntactic's outputted tokens 27 ⟩ ≡
  yacco2 :: TOKEN_GAGGLE_ITER i = P3_tokens.begin( );
  yacco2 :: TOKEN_GAGGLE_ITER ie = P3_tokens.end( );
  lrclog ≪ "Dump␣of␣P3␣tokons" ≪ endl;
  for (int yyy = 1; i ≠ ie; ++i) {
    CAbs_lr1_sym * sym = *i;
    if (sym ≡ yacco2 :: PTR_LR1_eog__) continue;
    lrclog ≪ yyy ≪ "::␣" ≪ sym→id__ ≪ "␣file␣no:␣" ≪ sym→tok_co_ords__.external_file_id__ ≪
        "␣line␣no:␣" ≪ sym→tok_co_ords__.line_no__ ≪ "␣pos:␣" ≪ sym→tok_co_ords__.pos_in_line__ ≪
        endl;
    ++yyy;
  }
```

This code is used in section 26.

28. Dump aid — Enumerate grammar's components.
As a reference aid to a grammar's components, each component has an enumerate value of "x.y.z" where x stands for the rule number, y is its subrule number, and z is the component number. The grammar's enumerated elements are "rule-def", "subrule-def", and components "refered-rule", "refered-T", and "eosubrule". The "rules-phrase" is not enumerated as it just ties all the forests together. An enumerate example is "1" standing for the Start rule. "1.2.2" goes to its 2nd subrule of component 2.

The grammar is read whereby all its forests are enumerated relative to one another.

⟨ dump aid: enumerate grammar's components 28 ⟩ ≡

```
set<int>␣enumerate_filter;
enumerate_filter.insert(T_Enum::T_rule_def_);
enumerate_filter.insert(T_Enum::T_T_subrule_def_);
enumerate_filter.insert(T_Enum::T_refered_T_);
enumerate_filter.insert(T_Enum::T_T_eosubrule_);
enumerate_filter.insert(T_Enum::T_refered_rule_);
enumerate_filter.insert(T_Enum::T_T_called_thread_eosubrule_);
enumerate_filter.insert(T_Enum::T_T_null_call_thread_eosubrule_);
using namespace NS_enumerate_grammar;
tok_can_ast_functor walk_the_plank_mate;
ast_prefix enumerate_grammar_walk(*rules_tree, &walk_the_plank_mate, &enumerate_filter, ACCEPT_FILTER);
tok_can < AST *> enumerate_grammar_can(enumerate_grammar_walk);
Cenumerate_grammar enumerate_grammar_fsm;
Parser enumerate_grammar(enumerate_grammar_fsm, &enumerate_grammar_can, 0, 0, &Error_queue);
enumerate_grammar.parse();
```

This code is used in section 14.

29. Epsilon and Pathological assessment of Rules.
Epsilon condition:
Rule contains an empty symbol string in a subrule. The only subtlety is when a rule has a subrule(s) containing all rules. If all the rules within that subrule are epsiloned, then this subrule is an epsilon and so turn on its rule as epsilonable.

Pathological Rule assessment:
Does a rule derive a terminal string? The empty string is included in this assessment. *epsilon_rules* grammar tells the whole story.

Note:
The tree is walked using discrete levels: Rules and Subrules. The subrule's elements are filtered out (not included) for the discrete rule traversal but is added within the rule's syntax directed code logic a subrule's element advancement. Element advancement bypasses the thread component expression. These are neat facilities provided by O_2 using the *tok_can* tree traversal containers.

⟨ epsilon and pathological assessment of Rules 29 ⟩ ≡

```
using namespace NS_epsilon_rules;
set<AST*> yes_pile;
set<AST*> no_pile;
list< pair<AST*,AST*> > maybe_list;
T_rules_phrase *rules_ph = O2_RULES_PHASE;
AST *rules_tree = rules_ph→phrase_tree();
set<int> filter;
filter.insert(T_Enum::T_T_subrule_def_);
filter.insert(T_Enum::T_rule_def_);
tok_can_ast_functor just_walk_functr;
ast_prefix rule_walk(*rules_tree, &just_walk_functr, &filter, ACCEPT_FILTER);
tok_can < AST *> rules_can(rule_walk);
Cepsilon_rules epsilon_fsm;
Parser epsilon_rules(epsilon_fsm, &rules_can, 0, 0, &Error_queue);
epsilon_rules.parse();
⟨ Print pathological symptoms but continue 30 ⟩;    /* ⟨ print tree 132 ⟩; */
```

This code is used in section 14.

30. Print pathological symptoms but continue.

⟨ Print pathological symptoms but continue 30 ⟩ ≡

```
if (Error_queue.empty() ≠ true) {
  DUMP_ERROR_QUEUE(Error_queue);
  Error_queue.clear();
  return 1;
}
```

This code is used in section 29.

31. Get the total number of subrules.
I'm lazy and don't want to distribute the count as the individual rules are being parsed so do it via the a tree walk on subrules. Why do it anyway? I've hardwired the *elem_space* table size against a constant *Max_no_subrules*. Why not allocate the table size dynamicly? Glad u asked as the malloc approach burped. Maybe there's mixed metaphores on malloc versus how the C++ new / delete allocation is done. Anyway this works and is reasonable.

⟨ get total number of subrules for *elem_space* size check 31 ⟩ ≡

```
set<int> sr_filter;
sr_filter.insert(T_Enum::T_T_subrule_def_);
ast_prefix sr_walk(*rules_tree, &just_walk_functr, &sr_filter, ACCEPT_FILTER);
tok_can < AST *> sr_can(sr_walk);
for (int xx(0); sr_can[xx] ≠ yacco2::PTR_LR1_eog__; ++xx) ;
O2_T_ENUM_PHASE→total_no_subrules(sr_can.size());
if (O2_T_ENUM_PHASE→total_no_subrules() > Max_no_subrules) {
  lrclog ≪ "Grammar's number of subrules: " ≪ O2_T_ENUM_PHASE→total_no_subrules() ≪
      " exceeds the allocated space for table elem_space: " ≪ Max_no_subrules ≪ endl;
  lrclog ≪ "This is a big grammar so please correct the grammar." ≪ std::endl;
  clog ≪ "Grammar's number of subrules: " ≪ O2_T_ENUM_PHASE→total_no_subrules() ≪
      " exceeds the allocated space for table elem_space: " ≪ Max_no_subrules ≪ endl;
  clog ≪ "This is a big grammar so please correct the grammar." ≪ std::endl;
  return 1;
}
```

This code is used in section 14.

32. Calculate each rule's first set.
Lov the discrete logic of a grammar to code algorithms. See *first_set_rules* grammar as it's really is simple in its logic: i'm getting there from all corners of the coding world. Not any more as i'm pruning the overhead so out my drafty thoughts and this grammar *first_set_rules*. Just iterate over the grammar tree for filtered *rule_def* nodes only.

⟨ calculate rules first sets 32 ⟩ ≡

```
set<int> fs_filter;
fs_filter.insert(T_Enum::T_rule_def_);
ast_prefix fs_rule_walk(*rules_tree, &just_walk_functr, &fs_filter, ACCEPT_FILTER);
tok_can < AST *> fs_rules_can(fs_rule_walk);
for (int xx(0); fs_rules_can[xx] ≠ yacco2::PTR_LR1_eog__; ++xx) {
  rule_def* rd = (rule_def*)fs_rules_can[xx];
  GEN_FS_OF_RULE(rd);
}
```

This code is used in section 14.

33. Calculate Start rule's called threads first set list.
It calculates the "called threads" first set for the "to be emitted xxx.fsc" file. The neat wrinkle is the epsilonable rule that requires same transience left-to-right moves thru the subrule expressions. This is fodder to O_2^{linker}that builds each thread's first set from the "list-of-native-first-set-terminal" and "list-of-transitive-threads" constructs. The final outcome of $O_2^{linker}$is an optimized list of threads per terminal. The calculation goes across the Start rule and its closured rules to determine the list of called threads. This list can be ϵ . In the "Start rule" is the contents for "list-of-transitive-threads".

⟨ calculate Start rule called threads first sets 33 ⟩ ≡

```
rule_def * start_rule_def = ( rule_def * ) fs_rules_can.operator[](0);
GEN_CALLED_THREADS_FS_OF_RULE(start_rule_def);
```

This code is used in section 14.

34. Are all Grammar phases parsed?.
As i parse the individual phrases by their keyword presence without using a grammar to sequence each phase, now is the time to see if all the parts are present in the grammar. This is a simple iteration on the posted `O2_PHRASE_TBL` to fetch their phrase terminals and to put them thru a post grammar sequencer.

I changed how the tokens are fetched from fill the container by iterating the *O2_xxx* phases to reading the grammar's tree. Why? Cuz i implicitly changed to on-the-fly enumeration of their values while they were being parsed. If their order was changed then their appropriate enumerates are out-of-alignment. For example if the raw character classification came before the "lrk" definitions, this would be catastrophic due to the down stream semantics' dependency on their correct enumerates.
A bird's view of O_2's phases: indent shows node's dependency

::1 grammar-phrase grammar-phrase file 2:0: line 24:4: sym*: 0122B598
 ::2 fsm-phrase fsm-phrase file 2:766: line 24:4: sym*: 01220BA0
 ::3 T-enum-phrase T-enum-phrase file 4:1069: line 32:14: sym*: 01272500
 ::4 lr1-k-phrase lr1-k-phrase file 5:1727: line 44:21: sym*: 011F0360
 ::5 rc-phrase rc-phrase file 6:303: line 13:15: sym*: 01270C98
 ::6 error-symbols-phrase error-symbols-phrase file 7:1026: line 34:14: sym*: 0257F388
 ::7 terminals-phrase terminals-phrase file 8:474: line 15:10: sym*: 011F1458
 ::8 rules-phrase rules-phrase file 2:1708: line 60:6: sym*: 02FB3AA8

Notice i walk the tree by *ast_prefix_wbreadth_only*. This visits the start node "grammar-phrase" and only its immediate children by the "breadth-only" qualifier.

⟨ are all phases parsed? 34 ⟩ ≡

```
set<int> phase_order_filter;
phase_order_filter.insert(T_Enum::T_T_fsm_phrase_);
phase_order_filter.insert(T_Enum::T_T_enum_phrase_);
phase_order_filter.insert(T_Enum::T_T_lr1_k_phrase_);
phase_order_filter.insert(T_Enum::T_T_rc_phrase_);
phase_order_filter.insert(T_Enum::T_T_error_symbols_phrase_);
phase_order_filter.insert(T_Enum::T_T_terminals_phrase_);
phase_order_filter.insert(T_Enum::T_T_rules_phrase_);
tok_can_ast_functor orderly_walk;
ast_prefix_wbreadth_only evaluate_phase_order(*GRAMMAR_TREE, &orderly_walk, &phase_order_filter,
    ACCEPT_FILTER);
tok_can<AST *> phrases_can(evaluate_phase_order);
using namespace NS_eval_phrases;
Ceval_phrases eval_fsm;
Parser eval_phrases(eval_fsm, &phrases_can, 0, 0, &Error_queue, 0, 0);
eval_phrases.parse();
⟨ if error queue not empty then deal with posted errors 21 ⟩;
```

This code is used in section 14.

35. Thread's end-of-token stream: Lookahead expression post evaluation.
If the grammar contains the 'parallel-parser' construct, then it is considered a thread. As a refinement, this construct allows one to fine-tune the lookahead boundaries of the grammar in its own contextual way. As this construct is declared before the grammar's vocabulary definitions — rules and terminals, the expression must be kept in raw character token format with some lexems removed like comments. Only after all the grammar has been recognized can the lookahead expression be parsed properly: the terms in the expression must relate to T-in-stbl, rule-in-stbl, and the + or − expression operators.

Squirrelled away in the 'parallel-parser' terminal is the raw token stream of the lookahead expression. The strategy used is to fetch the appropriate parsed phase token from the O_2 phase table and then deal with its locally defined pieces of information. Originally these parse phases were kept in the global symbol table but now they are contained in its own table. Why? Well how do u guard against a grammar writer defining a terminal whose key could be a synomyn to one of my internal parse phases? Regardless of how clever one is to naming keys, separation between my internal tables and the global symbol table has a 100% assurance of no conflict.

First set Criteria:

1) Element is a Terminal, use its calculated enumeration value
2) If the element is eolr, then use all calculated enumeration values
3) Element is a Rule, use its calculated First set terminals

Before the Lookahead first set can be calculated, the terminal vocabulary must be traversed and assigned an enumeration value per terminal. The grammar's rules must also have their first sets calculated before the lookahead expression can be calculated.

The lookahead logic within its grammar(s) is two fold:

a) parse the lookahead expression for kosher syntax
b) calculate the lookahead's first set from the expression

The error checks are for an ill-formed expressions, and for an empty first set calculation: for example, 'a' - 'a', or 'b' - 'eolr', and epsilon Rules used in the lookahead expression. This calculated first set is then used down stream in the finite state automata (FSA) generation of the grammar.

⟨ determine if la expression present. Yes parse it 35 ⟩ ≡

```
if (O2_PP_PHASE ≠ 0) {
   ⟨ parse la expression and calculate its first set 36 ⟩;
}
```

This code is used in section 14.

36. Parse the la expression and calculate its first set.

```
⟨ parse la expression and calculate its first set 36 ⟩ ≡
  T_parallel_parser_phrase *pp_ph = O2_PP_PHASE;
  if (pp_ph→la_bndry() ≡ 0) {
    CAbs_lr1_sym *sym = new Err_pp_la_boundary_attribute_not_fnd;
    sym→set_rc(*pp_ph);
    Error_queue.push_back(*sym);
    ⟨ if error queue not empty then deal with posted errors 21 ⟩;
  }
  T_parallel_la_boundary *la_bndry = pp_ph→la_bndry();
  yacco2::TOKEN_GAGGLE *la_srce_tok_can = la_bndry→la_supplier();
  yacco2::TOKEN_GAGGLE la_tok_can_lex;
  yacco2::TOKEN_GAGGLE la_expr_tok_can;
  using namespace NS_la_expr_lexical;
  Cla_expr_lexical la_expr_lex_fsm;
  Parser la_expr_lex_parse(la_expr_lex_fsm, la_srce_tok_can, &la_tok_can_lex, 0, &Error_queue,
      &JUNK_tokens, 0);
  la_expr_lex_parse.parse();
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
  using namespace NS_la_expr;
  Cla_expr la_expr_fsm;
  Parser la_expr_parse(la_expr_fsm, &la_tok_can_lex, &la_expr_tok_can, 0, &Error_queue, &JUNK_tokens, 0);
  la_expr_parse.parse();
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
```

This code is used in section 35.

37. Determine rule use count: Optimization.
To improve performance, the rules (Productions) symbols are newed once and recycled when needed. To ensure that there are enough recycled rules available, the gramar is traversed and their uses counted. If recursion is present within the rule, this adds one more use. The grammar tree is traversed looking only for "rule-def", "subrule-def", and "refered-rule" tokens.

```
⟨ determine each rule use count 37 ⟩ ≡
  lrclog << "Evaluate␣rules␣count" << endl;
  using namespace NS_rules_use_cnt;
  set<int>␣rules_use_cnt_filter;
  rules_use_cnt_filter.insert(T_Enum::T_T_subrule_def_);
  rules_use_cnt_filter.insert(T_Enum::T_rule_def_);
  rules_use_cnt_filter.insert(T_Enum::T_refered_rule_);
  tok_can_ast_functor rules_use_walk_functr;
  ast_prefix rules_use_walk(*GRAMMAR_TREE, &rules_use_walk_functr, &rules_use_cnt_filter, ACCEPT_FILTER);
  tok_can < AST *> rules_use_can(rules_use_walk);
  Crules_use_cnt rules_use_cnt_fsm;
  Parser rules_use_cnt(rules_use_cnt_fsm, &rules_use_can, 0, 0, &Error_queue);
  rules_use_cnt.parse();
```

This code is used in section 14.

38. Generate grammar's LR1 states.
The global lr states list `LR1_STATES` is added to dynamicly as each closure state/vector gens their states. `LR1_HEALTH` is the diagnostic of the parsed grammar.

39. Driver generating lr1 states.
Goes thru the lr state list looking for closure states to gen. Note: a closure state gens its transitive states. A part from the "closure only" state (start state), all other states contain 2 contexts: transitive core items, and possibly added to closured items. As the list is read, it evaluates the possible state for gening by seeing if there are **closured** items needing to be gened. There are 3 possible outcomes to this evaluation:

1) items not gened: goto of item is nil.
2) items completed due to right boundedness from a previous gen closure state / vector context.
3) partially gened items due to common prefix of a previous closure state/vector context.

Point 1 + 3 need gening. Point 1 is your regular generation context. Point 3 requires walking thru its right side symbols to where its goto state needs gening (nil). From there its gening proceeds as normal within its own closure state/vector context.

During each state closure part/vectors pass, lr kosherness is tested within each closure state/vector gening context. A non lr(1) verdict is returned immediately within the gening closure state/vector context. The balance of the closure state/vectors to gen are not completed.

```
⟨generate grammar's LR1 states 39⟩ ≡
  AST * start_rule_def_t = AST::get_1st_son(*rules_tree);
  state * gening_state = new state(start_rule_def_t);
  gen_context gening_context(0, −1);
  STATES_ITER_type si = LR1_STATES.begin( );
  STATES_ITER_type sie = LR1_STATES.end( );
    /* list added to dynamicly as each gening context created */
  for ( ; si ≠ sie; ++si) {
    gening_state = *si;
    gening_context.for_closure_state_ = gening_state;
    gening_context.gen_vector_ = −1;
    lrclog ≪ "lr␣state␣driver␣considered␣state:␣" ≪ gening_context.for_closure_state_→state_no_ ≪
        "␣for␣vector:␣" ≪ gening_context.gen_vector_ ≪ endl;
    LR1_HEALTH = gening_state→gen_transitive_states_for_closure_context(gening_context, *gening_state,
        *gening_state);
    if (LR1_HEALTH ≡ NOT_LR1_COMPATIBLE) {
      ⟨is the grammar unhealthy? yes report the details and exit 40⟩;
    }
  }     /* ⟨print dump state 135⟩; */
  ⟨commonize la sets 41⟩;     /* please put back at sign if u want to trace */
    /* ⟨print dump state 135⟩; */     /* ⟨print dump common states 134⟩; */
```

This code is used in section 14.

40. Is the grammar unhealthy? yes report the details and exit.

```
⟨ is the grammar unhealthy? yes report the details and exit 40 ⟩ ≡
  if (LR1_HEALTH ≡ NOT_LR1_COMPATIBLE) {
    yacco2::lrclog ≪ "===>Please␣check␣Grammar␣dump␣file:␣" ≪ normal_tracing.c_str() ≪
        "␣for␣Not␣LR1␣details" ≪ endl;
    std::cout ≪ "===>Please␣check␣Grammar␣dump␣file:␣" ≪ normal_tracing.c_str() ≪
        "␣for␣Not␣LR1␣details" ≪ endl;
    yacco2::lrclog ≪ "Not␣LR1␣---␣check␣state␣conflict␣list␣of␣state:␣" ≪
        gening_state→state_no_ ≪ "␣for␣details" ≪ endl;
    ⟨ print dump state 135 ⟩;
    ⟨ print dump common states 134 ⟩;
    return 1;
  }
```

This code is used in sections 14 and 39.

41. Commonize LA Sets — Combine common sets as a space saver.
Go thru the lr states looking for reduced subrules. Their lookahead sets have already been calculated so by set equality determine common la sets by reading thru the registry for its soul mate. This common reference to same sets minimizes space in the emitted lr state tables. The index number per set in the COMMON_LA_SETS registry will be used as part of each generated la set's name. This is why the found index number is deposited per reduced subrule. When the state tables get emitted, this index number + 1 is used in the gened lookahead's name as i prefer its name to be relative to 1.

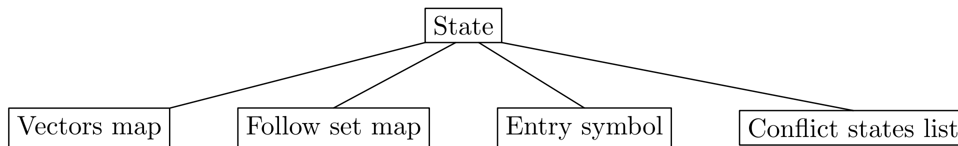
```
⟨ commonize la sets 41 ⟩ ≡
  COMMONIZE_LA_SETS();
```

This code is used in section 39.

42. Overview of O_2's state generated components.
O_2 generates the components making up the automaton and the first set language for O_2Linker to compile. These files are the header definition of the grammar, the "first set" file for O_2Linker, and the implementations of the automata (fsm), it's symbols, and the fsm's states.

Depending on the switches inputted, O_2 can generate the Terminal vocubulary defined for the grammar environment: the individual terminal classifications of errors, lr constants, raw characters, and Terminals. As a global reference to all defined terminals, an enumeration scheme is emitted.

43. LR1 definitions.



vectors_map[eno]: symbol's enumerate

- state's element list
- state_element ↑
  - grammar tree node ↑
  - closure state ↑
  - go_to state ↑
  - previous state ↑
  - reduced state ↑
  - previous state element
  - next state element
  - LA set ↑
  - Common LA set index

follow_set_map[eno]: rule's enumerate

- follow_set_element ↑
- rule no • rule def tree ↑
- state element ↑ • it's state ↑
- follow set of T−in−stbl ↑
- transitions: follow_set_element ↑
- merges:

Let's review what makes up a state:

1) subrules's specific element — state's to gen vectors
2) rules's follow set
3) state's entry symbol — Start state has no entry symbol
4) state's list of conflict states

A state is a set of productions (subrules) where each production's current symbol being worked on is some position along its string. A state from the arithmetic grammar discussed earlier could be represented by the following example where the "." indicates the position within the production's string being worked on in the state:

$S \rightarrow E\ .\ \perp$
$E \rightarrow E\ .\ + T$

The above state has 2 productions where each symbol being worked on is in position 2 of their respective strings. These are items in the state having their production configuration of subrule ⊗ string position. Sometimes I shall call each entry a **state element** rather than an item. A rule's follow set gets created when it is present in point 1: ie, the state's element is a rule and its follow set is the string to its right that generates teminals. Please see at the beginning of this document the "follow set" definition. Point 3's entry symbol identifies the symbol used to gen the state and to quickly help in determining whether two states are equivalent for potential state merging. Point 4 is a requirement to support merging of states from 2 different closured-part state networks. It supplies the lr1 states that have reduce / reduce or shift conditions that require the lr1 compatibility check. When there is a proposed merger from 2 different closured-part contexts, it is the union of their follow sets that gives the reducing subrules their lookahead sets. Consequently the lr1 conflict states of the "merged into context" must be evaluated for lr1 compatibility.

Parts of a state:

1) Closured
2) Transitive

A closured part are all the state's items whose elements start their strings. They have been brought into the state by the "closure" operation caused by a state's element being a rule. A transitive part are those productions whose elements are to the right of the start element. Items used to generate a new state are called **"core items"**.

State generation:
All states are generated from a closured part of a state. Its productions are walked along their strings producing transitive states until their strings are completely consumed. This holds for the "Start state" that starts things off by generating all it transitive children. Thereafter each transitive state is visited and assessed for its closured components that then generates its own transitive states. This goes on until all the generated states have been visited.

Contexts:
1) follow sets
2) production's reduced lookahead set
A production's reduction occurs when all its string has been recognized. For it to reduce, it depends on the context of its follow set within its birthing closured state: This is the lr1 compatibility context that is refered to as **lookahead**. When there is no conflict of interest between competing productions (reduces with possibly shifts) within the state, this becomes a lr(0) situation. Without regard for the lookahead context this now shifts the error detection to the state that must deal with the lookahead as the current terminal for shifting. This strategy is used when state merges takes place. Instead of exploding the number of states sensitive to only its own lookahead context, mergers combine the follow set contexts as long as there is no state incompatibilities created. 2 or more competing reducing productions requires their follow set contexts to resolve the reducing conflict: reduce / reduce or reduce / shift. Shifts of symbols are local to the reducing state.

Of course lookaheads are context sensitive according to each productions birthing states. In LR(1) terms, the lookahead is deterministic and provided by the follow sets having only 1 symbol string as lookahead.

Follow set and right bounded condition:
This condition is where a rule is the last symbol in its production string. Its closured productions inherit the follow set of the production string(s) that closured it. These follow sets are found in the gening closured state environment. Consequently right bounded closured productions must be gened in case it could produce a conflict state. Why? Merges taking place above this to-be-gened production from a different closured state generation could produce a conflict as the merger is not aware that one of its transitive states has a future conflict condition dependent on these merged follow set contexts.
As an example please see David Spector from "SIGPLAN VOL 23 DEC/88" where my gened "lr1_sp5.lex" grammar illustrates this condition.

Epsilon rules and right bounded condition:
If the last symbol is a rule and is epsilonable, then the right bounded condition moves left inwards from the end of the symbol string to the next right-to-left symbol. Now if that symbol is a rule it is considered a right bounded requiring generation within the current closured state environment. This is a recursive definition: right bounded condition gens closures having the right bounded condition that also requires immediate generation. When it comes time to gen the closured-part state of the right bounded components, they will have been already gened and their conflict states entered against their gening closured-part state environment.

Significance of right bounded condition:
It demands that its future closured state generation be associated with the generating closured state that created it. Restated: It must be generated prematurely by its spawning closured state. This way any of its transitive states that have the lr1 conflict condition will get placed in the conflict state list of the generating closured state so that a proposed merger relative to the original closured state is aware of the potential conflict and checked accordingly.

Some synonyms:
"Closure-only" state:
A state where all its state elements are configurations with their start symbol. This is your one and only Start state.

“Transitive” state:
A state where at least one state element is not the starting symbol of a production’s string.

“Closured-only-part” of a state:
All state’s elements whose symbols start the subrule string. Synonym: “closured state”.

“Closured-only-subrules” of a state:
Productions’s symbol strings brought into the state by the closure operation caused by a state’s element being a rule. The “closured-only” part are those subrules birthed within this state to generate all its “closured-only” subrules transitive states.

“Conflict state”:
A state having at least 2 items where at least one of the productions is reducing.

Building a state core:
There are only 2 contexts that provide the generation fodder for a state:
1) Start (closure-only) state — Start rule’s grammar tree definition
2) Transitive states — generated from a closured state
The “closured-part” of a state generates all its transitive states from its closure subrules regardless of the type of state — Start or transitive. Point 1 starts things off. It generates all its transitive states. Point 2 deals with transitive states from point 1 that have closured-only residues that need generating. Of course these newly generated transitive states could be merged into the existing lr1 state network if they meet the lr1 compatibility criteria. Eventually the newly added transitive states will be assessed for their “closured-part” generation.

Some Merge points:
First, only conflict states are tested. They are supplied by their associated closured generating state. When a merge takes place, the state being absorbed by the older closured network deposits its follow set info against the merged into state.

Second, the conflict states of the “merged into” state network must also be added to the gening closured state’s conflict state list. Why? If the state was not merged, eventually all its gened states would have the equivalent conflict states as the proposed merger. The only refinement to this is conflict states should only be added that are eventually generated from the “merged into” state. Now if future mergers are proposed into this newly closured state’s network, the conflict states of the absorbing network will also be there for the testing.

In summary, Lr1 state generation is discrete in its generation passes. Pass one: generate all the states for the start state from its “closured-only” subrules. Pass two and greater deals with “closured-only” parts of transitive states that have not been completely gened. Remember a subrule is associated with its birth state that brought it into existance. These transitive states are of previous passes. Each transitive pass looks for the next transitive state to generate until all its lr state network have been built. The “transitive state pass” generates all its “closured-only” subrules independently of the past generations.

Now the state implementation bedevils this definition as does Goethic churches — one usually does not see the infrastructure required to build it unless the project ran out of money and stands unfinished but open to its engineering secrets. So here’s the scaffolding for my sanity. A note on the following type defs sections: to make “cweave” behave in formatting its the document — a slight ahem until i debug / correct “cweave”. The cause is templates that came after the original program was written.

44. *gen_context* **definition/implementation.**
The context identifying the closure state and vector combo gening its states. This context is needed to prevent same closure state merges whose vectors are different but generate common states having different follow sets. If merged the contributing contexts could make it non lr1. See David Spector's paper "Efficicent Full Lr1 Parser Generators": G2 example. The context is maintained per state that gened it and per state's subrules vectors: *state_element*.

```
⟨ Structure implementations 44 ⟩ ≡
gen_context :: gen_context(state ∗ S, Voc_ENO Ve): for_closure_state_(S), gen_vector_(Ve)
  { }
```

See also sections 46, 48, 50, 52, 53, 54, 55, 57, 58, 60, 62, 63, 66, 67, 68, 69, 70, 72, 73, 74, 75, 76, 78, 79, 81, 82, 87, 88, 89, 91, 92, 93, 98, 99, 100, 101, 104, 105, 113, and 115.

This code is used in section 141.

45. *state_element* **definition/implementation.**
Basic building block of a state's set of subrules' string symbols. Laced throughout *state_element* are linkages between the past, present, and future of its lr1 state generation. This is the scaffolding to build the state network. *sr_def_element_* **is for tracing purposes only**. I could have gone the long way by getting the tree node's content and then fetch its definition but this makes life easier when truth telling takes place — **terminal-def or rule-def**.

The *la_set_* only gets created at the end-of-string point. It's a fast way to scratch-pad potential merges and the lr1 breathalyzer test.

46. *state_element* **implementation.**

```
⟨ Structure implementations 44 ⟩ +≡
state_element :: state_element(AST ∗ Elem):
        cs_vector_combo_gening_it_(0, −1), sr_element_(Elem), sr_def_element_(0), its_enum_id_(−1),
            subrule_def_(0), closure_state_(0), goto_state_(0), previous_state_(0), reduced_state_(0), self_state_(0),
            previous_state_element_(0), next_state_element_(0), la_set_(0), common_la_set_idx_(−1)
  {
    ⟨ determine entry symbol 47 ⟩;
  }
```

47. Determine entry symbol.
Eases tracing of lr states easier instead of just displaying its enumerated value.

```
⟨ determine entry symbol 47 ⟩ ≡
  CAbs_lr1_sym * sym = AST::content(*Elem);
  Voc_ENO id = sym→enumerated_id__;
  switch (id) {
  case T_Enum::T_refered_rule_:
    {
      ⟨ get cast referenced rule 108 ⟩;
      rule_def * rd = rr→its_rule_def();
      sr_def_element_ = rd;
      its_enum_id_ = rd→enum_id();
      break;
    }
  case T_Enum::T_T_eosubrule_:
    {
      ⟨ get cast referenced eosubrule 110 ⟩;
      sr_def_element_ = eos;
      its_enum_id_ = eos→enumerated_id__;
      la_set_ = new LA_SET_type();
      break;
    }
  case T_Enum::T_T_null_call_thread_eosubrule_:
    {
      ⟨ get cast referenced null called thread eosubrule 111 ⟩;
      sr_def_element_ = eos;
      its_enum_id_ = eos→enumerated_id__;
      break;
    }
  case T_Enum::T_T_called_thread_eosubrule_:
    {
      ⟨ get cast referenced called thread eosubrule 112 ⟩;
      sr_def_element_ = eos;
      its_enum_id_ = eos→enumerated_id__;
      break;
    }
  case T_Enum::T_refered_T_:
    {
      ⟨ get cast referenced T 109 ⟩;
      T_terminal_def * td = rt→its_t_def();
      sr_def_element_ = td;
      its_enum_id_ = td→enum_id();
      break;
    }
  }
```

This code is used in section 46.

48. *∼state_element.*

⟨ Structure implementations 44 ⟩ +≡
```
state_element::~state_element()
{
  if (la_set_ ≠ 0) delete la_set_;
}
```

49. Lookahead Comments.
Please see the *la_express* grammar as to how it calculates the thread's end-of-parse stream lookahead. The "eolr" metaterminal is discussed in length as to what it represents and how it is exploded when lookahead expressions are used in a thread grammar "parallel-la-boundary" construct.

50. *add_fs_setA_to_LA.*
Substitute "parallel-reduce-operator for "parallel-operator" to eliminate the ambiguity between look ahead for reduction purposes versus calling thread for shift purposes.

⟨ Structure implementations 44 ⟩ +≡
```
void state_element::add_fs_setA_to_LA(follow_element & Fe, LA_SET_type & La_to_fill_in)
{
  FOLLOW_SETS_ITER_type i = Fe.follow_set_.begin();
  FOLLOW_SETS_ITER_type ie = Fe.follow_set_.end();
  for ( ; i ≠ ie; ++i) {
    LA_SET_ITER_type j = La_to_fill_in.find(*i);
    if (j ≡ La_to_fill_in.end()) {
      T_in_stbl *t_sym = *i;
      ⟨ is there back to back thread calls? 51 ⟩;
    }
  }
}
```

51. Is there back to back thread call?.
Before the thread call reduce was made lr(0), it used the `|r|` meta terminal to reduce the first thread call as the `|||` operator within a state was ambiguous in the 2 contexts — run the thread or reduce the called thread.

⟨ is there back to back thread calls? 51 ⟩ ≡
```
if (t_sym→t_def()→enum_id() ≠ LR1_PARALLEL_OPERATOR) {
  La_to_fill_in.insert(t_sym);
}
else {
  using namespace yacco2_stbl;
  T_sym_tbl_report_card report_card;
  find_sym_in_stbl(report_card, *LR1_REDUCE_OPERATOR_LITERAL);
  T_in_stbl* td = (T_in_stbl*)report_card.tbl_entry_->symbol_;
  La_to_fill_in.insert(td);
}
```
This code is used in section 50.

52. *calc_la* — **fill the reduced element's la set by walking follow set graph.**
Fill in the lookahead set for a reduced subrule by walking its follow sets. I protect against merges vs right bounded transitions that could cycle by `VISITED_MERGE_STATES_IN_LA_CALC`. The gened la is checked empty so indicate as bad.

⟨Structure implementations 44⟩ +≡

```
bool state_element::calc_la(state_element & La_to_fill_in)
{
  if (La_to_fill_in.la_set_ ≡ Λ) {
    return false;    /* no set to fill;so not lr1 */
  }
  if (La_to_fill_in.reduced_state_ ≠ La_to_fill_in.self_state_) return true;
  VISITED_MERGE_STATES_IN_LA_CALC.clear( );
  La_to_fill_in.la_set_→clear( );
  state * cs = La_to_fill_in.closure_state_;
  CAbs_lr1_sym * sym = AST::content(*La_to_fill_in.sr_element_);
  T_ENO id = sym→enumerated_id__;
  switch (id) {
  case T_Enum::T_T_eosubrule_:
    {
      ⟨get cast referenced eosubrule 110⟩;
      rule_def * rd = eos→its_rule_def( );
      RULE_ENO r_id = rd→enum_id( );
      S_FOLLOW_SETS_ITER_type i = cs→state_s_follow_set_map_.find(r_id);
      follow_element * fe = i→second;
      add_fs_setA_to_LA(*fe, *La_to_fill_in.la_set_);
      if (fe→transitions_.empty( ) ≠ true) {
        fill_la_from_transition(La_to_fill_in, fe→transitions_);
      }
      if (fe→merges_.empty( ) ≠ true) {
        fill_la_from_merge(La_to_fill_in, fe→merges_, r_id);
      }
      break;
    }
  case T_Enum::T_T_null_call_thread_eosubrule_:
    {
      break;
    }
  case T_Enum::T_T_called_thread_eosubrule_:
    {
      break;
    }
  }
  if (La_to_fill_in.la_set_→empty( ) ≡ true) {
    return false;    /* not lr1 as cant have an empty la set */
  }
  else {
    return true;    /* la set ok */
  }
}
```

53. *fill_la_from_merge*.

⟨ Structure implementations 44 ⟩ +≡

```
void state_element::fill_la_from_merge(state_element &La_to_fill_in, MERGES_type &Merge,
        RULE_ENO Rule_no)
{
  MERGES_ITER_type i = Merge.begin();
  MERGES_ITER_type ie = Merge.end();
  for ( ; i ≠ ie; ++i) {
    state *cs = *i;
    STATES_SET_ITER_type ii = VISITED_MERGE_STATES_IN_LA_CALC.find(cs);
    if (ii ≠ VISITED_MERGE_STATES_IN_LA_CALC.end()) return;
    VISITED_MERGE_STATES_IN_LA_CALC.insert(cs);
    S_FOLLOW_SETS_ITER_type i = cs→state_s_follow_set_map_.find(Rule_no);
    follow_element *fe = i→second;
    add_fs_setA_to_LA(*fe, *La_to_fill_in.la_set_);
    if (fe→transitions_.empty() ≠ true) {
      fill_la_from_transition(La_to_fill_in, fe→transitions_);
    }
    if (fe→merges_.empty() ≠ true) {
      fill_la_from_merge(La_to_fill_in, fe→merges_, fe→rule_no_);
    }
  }
}
```

54. *fill_la_from_transition*.

⟨ Structure implementations 44 ⟩ +≡

```
void state_element::fill_la_from_transition(state_element &La_to_fill_in, TRANSITIONS_type &Transition)
{
  TRANSITIONS_ITER_type i = Transition.begin();
  TRANSITIONS_ITER_type ie = Transition.end();
  for ( ; i ≠ ie; ++i) {
    follow_element *fe = *i;
    add_fs_setA_to_LA(*fe, *La_to_fill_in.la_set_);
    if (fe→transitions_.empty() ≠ true) {
      fill_la_from_transition(La_to_fill_in, fe→transitions_);
    }
    if (fe→merges_.empty() ≠ true) {
      fill_la_from_merge(La_to_fill_in, fe→merges_, fe→rule_no_);
    }
  }
}
```

55. *find_state_element_s_rule_no*.

Used for state merging to calculate a reducing subrule's lookahead set.

⟨ Structure implementations 44 ⟩ +≡

```
RULE_ENO state_element::find_state_element_s_rule_no()
{
  return subrule_def_→its_rule_def()→enum_id();
}
```

56. Follow set definition for a rule.
The input set of strings for the rule's follow set are provided by the state's *S_VECTORS_type* that is a map of 3 generic enumerate types — "rule-defs", "T-defs", and "eosubrule" variants. Of particular interest in this map are the "rule-def"s. The *state_element*s associated with the rule are the GPS into each subrule's symbol string. One can view this as the state's contributors list to generate both its lr states and the referenced rules' follow sets for this state. Now these input follow set strings are the strings to its right of its GPS. This is supplied thru the symbol's grammar next brother tree node.

57. Follow set implementation.

```
⟨ Structure implementations 44 ⟩ +≡
  AST * follow_element :: rule_def_t( )
  {
    return rule_def_t_;
  }
  state * follow_element :: its_state( )
  {
    return its_state_;
  }
follow_element ::follow_element(state * State): rule_no_(−1), rule_def_t_(0), its_state_(State)
  { }
follow_element ::follow_element(RULE_ENO Rule_no, state_element & State_elem, AST & Rule_def_t):
        rule_no_(Rule_no), rule_def_t_(&Rule_def_t), its_state_(State_elem.closure_state_)
  { }
  void follow_element :: add_follow_set_contributor(AST * SR_element)
  {
    sr_elements_.push_back(SR_element);
  }
```

58. *add_follow_set_transition***.**
Find eosubule's lhs rule. This gives the "rule no" to fetch its follow set element from its closure state's follow set map. Remember, epsilon stays within its closured state whilst a reducing subrule **needs to go back to the start of its symbol string** for its spawning rule and hence its follow set.

```
⟨ Structure implementations 44 ⟩ +≡
  void follow_element :: add_follow_set_transition(state_element & State_elem, T_eosubrule & Eos)
  {
    rule_def * rd = Eos.its_rule_def( );
    RULE_ENO eno = rd→enum_id( );
    S_FOLLOW_SETS_ITER_type fsi = State_elem.closure_state_→state_s_follow_set_map_.find(eno);
    if (fsi ≡ State_elem.closure_state_→state_s_follow_set_map_.end( )) {
      return;
    }
    follow_element * fe = fsi→second;
    ⟨ left recursion on rule check — out damn spot 59 ⟩;
    transitions_.push_back(fe);
  }
```

59. Left recursion on rule check.
Don't want to cycle on the same state spot: S1.A transitions on S1.A caused by a grammar's rule having left rule recursion.

⟨ left recursion on rule check — out damn spot 59 ⟩ ≡

```
if ((rule_no_ ≡ eno) ∧ (its_state_→state_no_ ≡ State_elem.closure_state_→state_no_)) return;
```

This code is used in section 58.

60. Add the terminal to the follow set.

⟨ Structure implementations 44 ⟩ +≡

```
void follow_element::add_T_to_follow_set(AST * Refered_T)
{
  ⟨ get refered-t 61 ⟩;
  follow_set_.insert(t→t_in_stbl());
}
```

61. Get refered-t.

⟨ get refered-t 61 ⟩ ≡

```
refered_T*␣t␣=␣(refered_T*)AST::content(*Refered_T);
```

This code is used in section 60.

62. *remove_merge_closure_info*.

⟨ Structure implementations 44 ⟩ +≡

```
void follow_element::remove_merge_closure_info()
{
  merges_.pop_front();
}
```

63. *add_merge_closure_info*.
Watch for rule having subrules that are merged with common closure state. U should only have 1 such state in list so throw out duplicates.

⟨ Structure implementations 44 ⟩ +≡

```
void follow_element::add_merge_closure_info(state & To_merge_closure_state)
{
  state * tm = &To_merge_closure_state;
  MERGES_ITER_type i = merges_.begin();
  MERGES_ITER_type ie = merges_.end();
  for ( ; i ≠ ie; ++i) {
    state * s = *i;
    if (s ≡ tm) return;
  }
  merges_.push_front(&To_merge_closure_state);
}
```

64. State definition/implementation.
*vectored_into_by_elem_* is the goto element from the spawning state that enters this state. The "xxx-def" symbol is provided by *vectored_into_by_elem_sym_* that is used for tracing purposes. It is one of the elements in determining whether 2 states are equal. I use the symbol's defining enumerate value which was enumerated across all the Grammar's vocabulary: Rules and Terminals. `START_STATE_ENUMERATE` symbol representing -1 is used to accommodate a "closure only" state where there is no symbol entering the start state as a Grammar's vocabulary enumeration begins at 0.

*closure_rule_list_* provides referenced rules in the state to complete the state's elements. *follow_rule_list_* is a fast way to deal with building follow sets for the state as it is a list of rule numbers that are keys into *state_s_to_vector* that indirectly supplies the follow string contexts.

To support rules recycling optimization, a quasi closure state for any rule of the grammar has been added. Why the addition? Recycling of rules requires a use count derived from recursion and subrules references to the rule. My first attempt was wrong as i did not take into account that a rhs subrule could have a referenced rule that could be indirectly referenced by (derived by) a suffixed referenced rule. So i need to derive the state containing the closured items and them analyse its content to see whether indirect referencing is taking place. So create ctor of state with no tree and a *closure_only_derives* method.

65. State's map of "to vector" elements.
S_VECTORS_type is the state's map of "to vector" elements of "rule-ref", "T-ref", and *eosubrule*. These elements produce the "goto" state eminating out of the lr1 state. This is a white lie as the *eosubrule* eminates nothing. It represents either the epsilon condition if its the first element of a subrule or a fully consumed subrule: its string of symbols has been consumed and so to be reduced. "rule-ref", "T-ref" are proxies to their definitions whereby their enumerated values are unique.

The second part of the map is the list of **same state elements** having identical enumerated keys. These vectors are the fodder to generate the next set of states eminating from this state and all the "closured-only part" states progeny. The list is sorted by the AST address inside the state's emlement so that state equivalences can be determined. U might raise the point: doesn't it matter what order the elements are placed inside the state to generate the lr1 state network: FIFO? NO! Let's review why.

1) "closured only" state composed of 1st position only subrules' elements.
2) this state's follow sets are static: first set from strings to rt of refered rules.
3) only the closured-only subrules are fully generated at the same time.
4) transitive states only continue gening their subrules from the closured state.
5) the resulting lr1 states are evaluated for lr1 conflicts.
6) apply the logic above to gened states having incomplete gened closured-only parts.

It is point 2 that is interesting: the birthing closured states of its reducing subrules supplies their lookahead. This means the closured state is generated completely before an assessment needs to take place. The lr1 assessment determines whether the gened states are lr(1) compatible. This check goes only against states that have reducing subrules so that the reduce / reduce and shift / reduce conditions can be verified.

How is the lr1 condition evaluated? Easy, the reducing subrule's rule within its birthing closured state contains its follow set: ie its lookahead terminals. All it takes is to make sure that the intersection of all the reducing subrules' follow sets is empty and that the state's shift terminals are not in any of the reducing subrules' follow sets. This shift set can be considered an invisible follow set that is applied at the same time to the other reducing follow sets. Keeping a list of conflicting states within the "closure-only or part" state when a state merge is proposed allows one to apply this lr1 condition for compatibility against the potential merged follow sets. Remember a gened state is produced out of its closured state. Thus mergers mean use the follow sets of each closure state. The "state to merge into" already has it list of lr1 conflict states in its associated gening closured state that need checking before Mr. Goodwrench nods.

Key: element's enumerate: "rule-def", "T-def", and "eosubrule"
Elements in list: state's elements that contain a grammar's tree node address

66. State implementation.

⟨ Structure implementations 44 ⟩ +≡

67. *state*(AST ∗ *Start_rule_t*).

⟨ Structure implementations 44 ⟩ +≡

```
state::state(AST * Start_rule_t): cs_vector_combo_gening_it_(0, -1),
        vectored_into_by_elem_(START_STATE_ENUMERATE), vectored_into_by_elem_sym_(0), state_no_(0),
        closure_state_birthing_it_(0), state_type_(0), arbitrator_name_(0)
  {
    create_start_state(*Start_rule_t);
    add_state_to_gbl_lr1_state_tbls(this);
  }
```

68. *state*(): **for closure only state of derives.**

⟨ Structure implementations 44 ⟩ +≡

```
  state::state()
  : cs_vector_combo_gening_it_(0, -1), vectored_into_by_elem_(START_STATE_ENUMERATE),
      vectored_into_by_elem_sym_(0), state_no_(0), closure_state_birthing_it_(0), state_type_(0),
      arbitrator_name_(0) { }
```

69. *state*(AST & *Vectored_into_id_t*) — **Create transitive state.**

⟨ Structure implementations 44 ⟩ +≡

```
state::state(Voc_ENO Eno, CAbs_lr1_sym * Entry_sym): cs_vector_combo_gening_it_(0, -1),
        vectored_into_by_elem_(Eno), vectored_into_by_elem_sym_(Entry_sym), state_no_(0),
        closure_state_birthing_it_(0), state_type_(0), arbitrator_name_(0)
  { }
```

70. *closure_only_derives* — **Create a closure only derives state.**
Gen a derives only state for a rule so that the rule's recycle count is correct for indirect references. Used by *rules_use_cnt.lex* grammar.

⟨ Structure implementations 44 ⟩ +≡

```
  void state::closure_only_derives(AST * Rule_tree)
  {
    gen_context gening_context(0, -1);
    add_rule_s_subrules_to_state(*Rule_tree, gening_context, *this);
    add_closure_rules_subrules_to_state(gening_context, *this);
  }
```

71. Generate states.

72. *add_element_to_state_vector*.
The *state_s_vector_*[enumerate of element] is positive except when its "eosubrule". Why? Cuz i'm (re)cursing on this terminal in 2 ways: as an end-of-subrule condition for a production and in grammars that are referencing it: is this devine? So when it's a real end-of-string situation i make the key negative. When it's a "refered-T" then use its contained "terminal-def" positive image that could be the containment of "eosubrule".

The element list is sorted on its AST address. Why the order? This makes sure that 2 elements having been brought into the state by different order will equate when state merges are proposed.

⟨Structure implementations 44⟩ +≡

```
void state::add_element_to_state_vector(Voc_ENO Elem_id, state_element & Elem)
{
  S_VECTORS_ITER_type i = state_s_vector_.find(Elem_id);
  if (i ≡ state_s_vector_.end()) {
    state_s_vector_[Elem_id] = S_VECTOR_ELEMS_type();
    i = state_s_vector_.find(Elem_id);
  }
  S_VECTOR_ELEMS_type & el = i→second;
  if (el.empty() ≡ true) {
    el.push_back(&Elem);
    return;
  }
  S_VECTOR_ELEMS_ITER_type j = el.begin();
  S_VECTOR_ELEMS_ITER_type je = el.end();
  for ( ; j ≠ je; ++j) {
    state_element * se = *j;
    if (Elem.sr_element_ < se→sr_element_) {
      el.insert(j, &Elem);
      return;
    }
  }
  el.push_back(&Elem);
}
```

73. *add_closure_rules_subrules_to_state*.
Why the *closure_rule_list_.end*( ) in the loop? As i'm adding items to it while iterating thru it, i play it safe by testing each cycle for the end-of-container condition via the function call rather than a local variable set before the iteration.

⟨ Structure implementations 44 ⟩ +≡
```
void state::add_closure_rules_subrules_to_state(gen_context &Possible_gen_context, state &Closure_state)
{
  CLOSURE_RULES_type processed_rules_set;
loop_until_empty: ;
  if (closure_rule_list_.empty() ≡ true) return;
  CLOSURE_RULES_ITER_type i = closure_rule_list_.begin();
  if (processed_rules_set.find(*i) ≠ processed_rules_set.end()) {
    closure_rule_list_.erase(*i);
    goto loop_until_empty;
  }
  processed_rules_set.insert(*i);
  rule_def *rd = (*i)→r_def();
  AST *t = rd→rule_s_tree();
  add_rule_s_subrules_to_state(*t, Possible_gen_context, Closure_state);
  goto loop_until_empty;
}
```

74. *add_rule_to_closure_list*.

⟨ Structure implementations 44 ⟩ +≡
```
void state::add_rule_to_closure_list(rule_in_stbl *Rule_in_stbl)
{
  CLOSURE_RULES_ITER_type i = closure_rule_list_.find(Rule_in_stbl);
  if (i ≡ closure_rule_list_.end()) {
    closure_rule_list_.insert(Rule_in_stbl);
    derives_closure_rule_list_.insert(Rule_in_stbl);
    rule_def *rd = Rule_in_stbl→r_def();
    if (rd→closure_rules_making_up_first_set()→empty() ≡ true) return;
    CLOSURE_RULES_type *cr = rd→closure_rules_making_up_first_set();
    CLOSURE_RULES_ITER_type j = cr→begin();
    CLOSURE_RULES_ITER_type je = cr→end();
    for ( ; j ≠ je; ++j) {
      rule_in_stbl *ris = *j;
      i = closure_rule_list_.find(ris);
      if (i ≡ closure_rule_list_.end()) {
        derives_closure_rule_list_.insert(ris);
        closure_rule_list_.insert(ris);
      }
    }
  }
}
```

75. *add_rule_s_subrules_to_state*.
Add rule's productions to state due to closured operation.

A wrinkle: if the possible gen context has a gen vector of -1, this means the state's closure elements are not right bounded and are not assiciated with generating context and so its gen context will be out its own state and own vector.

⟨ Structure implementations 44 ⟩ +≡

```
void state::add_rule_s_subrules_to_state(AST &Start_Rule_def_t, gen_context &Possible_gen_context,
        state &Closure_state_associate_with)
{
  AST *subrules_t = AST::get_1st_son(Start_Rule_def_t);
  AST *first_element_t(0);
  CAbs_lr1_sym *first_element(0);
  Voc_ENO id(START_STATE_ENUMERATE);
  Voc_ENO cs_id(START_STATE_ENUMERATE);
  for ( ; subrules_t ≠ 0; subrules_t = AST::brother(*subrules_t)) {
    T_subrule_def* srd = (T_subrule_def*)AST::content(*subrules_t);
    first_element_t = AST::get_1st_son(*subrules_t);
    first_element = AST::content(*first_element_t);
    state_element *se = new state_element(first_element_t);
    se→subrule_def_ = srd;
    se→self_state_ = this;
    se→closure_state_ = this;
    se→closured_state_gening_it_ = &Closure_state_associate_with;
    id = first_element→enumerated_id__;
    switch (id) {
    case T_Enum::T_refered_rule_:
      {
        refered_rule* rr = (refered_rule*)first_element;
        rule_def *rd = rr→its_rule_def();
        rule_in_stbl *ris = rr→Rule_in_stbl();
        RULE_ENO r_id = rd→enum_id();
        cs_id = r_id;
        se→cs_vector_combo_gening_it_.gen_vector_ = r_id;
        add_element_to_state_vector(r_id, *se);
        add_rule_to_follow_list(r_id);
        add_rule_to_closure_list(ris);
        break;
      }
    case T_Enum::T_T_eosubrule_:
      {
        T_ENO t_id = T_Enum::T_T_eosubrule_;
        cs_id = t_id;
        se→reduced_state_ = this;
        se→cs_vector_combo_gening_it_.gen_vector_ = t_id;
        add_element_to_state_vector(−t_id, *se);
        break;
      }
    case T_Enum::T_T_null_call_thread_eosubrule_:
      {
        T_ENO t_id = T_Enum::T_T_null_call_thread_eosubrule_;
        cs_id = t_id;
        se→reduced_state_ = this;
```

```
          se→cs_vector_combo_gening_it_.gen_vector_ = t_id;
          add_element_to_state_vector(−t_id, *se);
          break;
        }
      case T_Enum::T_T_called_thread_eosubrule_:
        {
          T_ENO t_id = T_Enum::T_T_called_thread_eosubrule_;
          cs_id = t_id;
          se→reduced_state_ = this;
          se→cs_vector_combo_gening_it_.gen_vector_ = t_id;
          add_element_to_state_vector(−t_id, *se);
          break;
        }
      case T_Enum::T_refered_T_:
        {
          refered_T*␣rt␣=␣(refered_T*)first_element;
          T_terminal_def * td = rt→its_t_def();
          T_ENO t_id = td→enum_id();
          cs_id = t_id;
          se→cs_vector_combo_gening_it_.gen_vector_ = t_id;
          add_element_to_state_vector(t_id, *se);
          break;
        }
      }
      if (Possible_gen_context.gen_vector_ ≠ −1) {
        se→cs_vector_combo_gening_it_ = Possible_gen_context;
      }
      else {
        se→cs_vector_combo_gening_it_.for_closure_state_ = this;
        se→cs_vector_combo_gening_it_.gen_vector_ = cs_id;
      }
    }
  }
```

76. *crt_core_items_of_state*.
Add subrules to the new state being created. The iterators walk the vector list of the spawning state.

Right bounded check:
1) Rx $\rightarrow \alpha$. Ra t
2) Ry $\rightarrow \beta$. Ra
3) Rz $\rightarrow$. Ra Rb

The period indicates where within the string of the production the current vector is. Point 1 is not right bounded due to t representing a terminal. Point 2 is right bounded as the follow set of its Ra hits the end-of-string condition and so must transition back along to Ry's follow sets. Point 3 could be right bounded if Rb is epsilonable. Cuz of point 2, the newly generated state's closure rules will be associated with the current closure state generation.

⟨ Structure implementations 44 ⟩ +≡

```
bool state::crt_core_items_of_state(S_VECTOR_ELEMS_ITER_type & Iter_begin,
        S_VECTOR_ELEMS_ITER_type & Iter_end, gen_context & Gening_context)
{
  bool rt_bnded(false);
  AST * to_element_t(0);
  CAbs_lr1_sym * to_element(0);
  Voc_ENO id(START_STATE_ENUMERATE);
  for ( ; Iter_begin ≠ Iter_end; ++Iter_begin) {
    state_element * from_se = *Iter_begin;
    to_element_t = AST::brother(*from_se→sr_element_);
    bool se_rt_bnded_condition = is_str_rt_bnded(to_element_t);
    if (se_rt_bnded_condition ≡ true) rt_bnded = se_rt_bnded_condition;
    to_element = AST::content(*to_element_t);
    switch (to_element→enumerated_id__) {
    case T_Enum::T_T_null_call_thread_eosubrule_:
      {     /* bypass */
        to_element_t = AST::brother(*to_element_t);
        se_rt_bnded_condition = is_str_rt_bnded(to_element_t);
        if (se_rt_bnded_condition ≡ true) rt_bnded = se_rt_bnded_condition;
        to_element = AST::content(*to_element_t);
        break;
      }
    case T_Enum::T_T_called_thread_eosubrule_:
      {     /* bypass */
        to_element_t = AST::brother(*to_element_t);
        se_rt_bnded_condition = is_str_rt_bnded(to_element_t);
        if (se_rt_bnded_condition ≡ true) rt_bnded = se_rt_bnded_condition;
        to_element = AST::content(*to_element_t);
        break;
      }
    }
    state_element * se = new state_element(to_element_t);
    se→subrule_def_ = from_se→subrule_def_;
    se→self_state_ = this;
    se→closure_state_ = from_se→closure_state_;
    se→closured_state_gening_it_ = Gening_context.for_closure_state_;
      /* se→cs_vector_combo_gening_it_ = from_se→cs_vector_combo_gening_it_; */
    se→cs_vector_combo_gening_it_ = Gening_context;
```

```
    if (se_rt_bnded_condition ≡ true) {
      if (from_se→closured_state_gening_it_ ≠ Gening_context.for_closure_state_) {
          /* common prefix syndrome: keep it pure: same as from se */
        se→cs_vector_combo_gening_it_ = from_se→cs_vector_combo_gening_it_;
        se→closured_state_gening_it_ = from_se→closured_state_gening_it_;
      }
      else {
        se→cs_vector_combo_gening_it_ = Gening_context;
        se→closured_state_gening_it_ = Gening_context.for_closure_state_;
      }
    }
    from_se→goto_state_ = se→self_state_;
    se→previous_state_ = from_se→self_state_;
    from_se→next_state_element_ = se;
    se→previous_state_element_ = from_se;
    id = to_element→enumerated_id__;
    ⟨add subrule's element to the being gened state's vector 77⟩;
  }
  return rt_bnded;
}
```

77. Add subrule's element to the being gened state's vector.

⟨ add subrule's element to the being gened state's vector 77 ⟩ ≡

```
switch (id) {
case T_Enum::T_refered_rule_:
  {
    refered_rule*␣rr␣=␣(refered_rule*)to_element;
    rule_def *rd = rr→its_rule_def();
    rule_in_stbl *ris = rr→Rule_in_stbl();
    RULE_ENO r_id = rd→enum_id();
    add_element_to_state_vector(r_id, *se);
    add_rule_to_follow_list(r_id);
    add_rule_to_closure_list(ris);
    break;
  }
case T_Enum::T_T_eosubrule_:
  {
    T_ENO t_id = T_Enum::T_T_eosubrule_;
    add_element_to_state_vector(-t_id, *se);
    break;
  }
case T_Enum::T_T_null_call_thread_eosubrule_:
  {
    T_ENO t_id = T_Enum::T_T_null_call_thread_eosubrule_;
    add_element_to_state_vector(-t_id, *se);
    break;
  }
case T_Enum::T_T_called_thread_eosubrule_:
  {
    T_ENO t_id = T_Enum::T_T_called_thread_eosubrule_;
    add_element_to_state_vector(-t_id, *se);
    break;
  }
case T_Enum::T_refered_T_:
  {
    refered_T*␣rt␣=␣(refered_T*)to_element;
    T_terminal_def *td = rt→its_t_def();
    T_ENO t_id = td→enum_id();
    add_element_to_state_vector(t_id, *se);
    break;
  }
}
```

This code is used in section 76.

78. *create_start_state* — **Create start state.**

```
⟨ Structure implementations 44 ⟩ +≡
  void state::create_start_state(AST & Start_rule_t)
  {
    gen_context gening_context(0, −1);
    add_rule_s_subrules_to_state(Start_rule_t, gening_context, *this);
    add_closure_rules_subrules_to_state(gening_context, *this);
    crt_start_rule_s_follow_set(Start_rule_t);
    create_follow_sets_of_state();
    this→state_type_ = determine_reduced_state_type(this);    /* Print_dump_state(this); */
  }
```

79. *gen_transitive_states_for_closure_context*.

Loop thru the state's closure/vector where the subrule's start position is the first symbol. The loop depends on the gening closure state/vector. When a state is constructed, the closured rules are associated with either:

1) itself state/vector
2) a right bounded rule context that associates with the gening context creating the state

```
⟨ Structure implementations 44 ⟩ +≡
  bool state::gen_transitive_states_for_closure_context
  (gen_context & For_gening_context, state & For_closure_state, state & State)
  {
    ⟨ Increment and printout Recursion counter 137 ⟩;
    lrclog ≪ "gen_transitive_states_for_closure_context␣for␣closure␣state:␣" ≪
        For_gening_context.for_closure_state→state_no_ ≪ endl;
    S_VECTORS_ITER_type i = State.state_s_vector_.begin();
    S_VECTORS_ITER_type ie = State.state_s_vector_.end();
    for ( ; i ≠ ie; ++i) {    /* read state's goto symbols */
      Voc_ENO eno = i→first;
      ⟨ unchain my reduce states if end-of-subrule and continue to next item 80 ⟩;
      For_gening_context.gen_vector_ = eno;
      ⟨ Printout Recursion counter 139 ⟩;
      lrclog ≪ "␣for␣vector:␣" ≪ For_gening_context.gen_vector_ ≪ endl;
      bool continue_gening = gen_a_state(For_gening_context, For_closure_state, State, i);
      if (continue_gening ≡ NOT_LR1_COMPATIBLE) {
        ⟨ Decrement Recursion counter 140 ⟩;
        return NOT_LR1_COMPATIBLE;
      }
    }
    ⟨ Decrement Recursion counter 140 ⟩;
    return LR1_COMPATIBLE;    /* gened states ok */
  }
```

80. Unchain my reduced states if end-of-subrule and continue to next item. Rip thru the subrule's symbols depositing its reduced state as eyeball info.

```
⟨unchain my reduce states if end-of-subrule and continue to next item 80⟩ ≡
  if (eno ≡ −T_Enum :: T_T_eosubrule_) {
    S_VECTOR_ELEMS_type elem_list = i→second;
    S_VECTOR_ELEMS_ITER_type j = elem_list.begin();
    S_VECTOR_ELEMS_ITER_type je = elem_list.end();
    for ( ; j ≠ je; ++j) {
      state_element *se = *j;
      se→reduced_state_ = se→self_state_;
      state *reduced_state = se→self_state_;
      while (se→previous_state_element_ ≠ 0) {
        se→previous_state_element_→reduced_state_ = reduced_state;
        se = se→previous_state_element_;
      }
    }
    continue;    /* eosubrule: onto next vector to gen in closure state's vector loop */
  }
```

This code is used in sections 79 and 81.

81. *gen_transitive_states_balance_for_closure_vector*.

```
⟨Structure implementations 44⟩ +≡
  bool state :: gen_transitive_states_balance_for_closure_vector
  (gen_context &Gen_context, state &For_closure_state, state &Goto_state)
  {
    ⟨Increment and printout Recursion counter 137⟩;
    lrclog ≪ "gen_transitive_states_balance_for_closure_vector␣for␣<" ≪
        Gen_context.for_closure_state_→state_no_ ≪ "," ≪ Gen_context.gen_vector_ ≪
        ">␣goto␣state:␣" ≪ Goto_state.state_no_ ≪ endl;
    S_VECTORS_ITER_type i = Goto_state.state_s_vector_.begin();
    S_VECTORS_ITER_type ie = Goto_state.state_s_vector_.end();
    for ( ; i ≠ ie; ++i) {    /* read state's goto symbols */
      Voc_ENO eno = i→first;
      ⟨unchain my reduce states if end-of-subrule and continue to next item 80⟩;
      bool continue_gening = gen_a_state(Gen_context, For_closure_state, Goto_state, i);
      if (continue_gening ≡ NOT_LR1_COMPATIBLE) {
        ⟨Decrement Recursion counter 140⟩;
        return NOT_LR1_COMPATIBLE;    /* stop gening as not lr1 */
      }
    }
    ⟨Decrement Recursion counter 140⟩;
    return LR1_COMPATIBLE;
  }
```

82. *gen_a_state*.
The only wrinkle is when a previous closured state has dragged along a common prefix subrule that is not part of its productions being gened. Call this a premature production generation: the production could be partially gened up to where the common prefix differs or the production completely gened as its right-hand-side string of symbols was contained in the other closure state gened productions. When this premature production's closure state is finally generating all its productions, common conflict states from a past closure generation that include premature gened productions must also be assessed for state conflicts. There is a commonality between the conflict states per gened closure states due to their productions that contributed to this conflict state list and so these common states must be added to the premature closure state's conflict state list. The premature gened production could be reducing whilst its common production that dragged it along could be shifting or reducing. The reverse could also be happening: the being gened production could be reducing while the premature production shifting. Thus a proposed merge into a state of this closure state must also have these common conflict states assessed for lr1ness.

A good example of this is situation is Deremer and Pennello's paper on "Efficient computation of lalr(1) lookahead sets" ACM Transactions on Programming Language and Systems: Vol. 4 no. 4 October 1982 Page 632. Please see my gened grammar "lalr_dp1.lex" illustrating this.

Pathological grammar condition:
Not lr1 grammar where the gening closure state network is not lr1. So must flag the *gen_a_state* with a returned result: continue or stop.

To stop infinite looping within its own closure state / vector being gened, make sure new state being added is lr1 compatible! Instead of analysing the gened network for lr1 compatibility after it is built, do a compatibilty test while it is being built. This stops the infinite looping context!

There are 2 not lr1 compatibilty contexts:
1) can a state be merged into another closure state's network
2) can not merge and its is not a kosher lr(1) grammar
Point 1) tests the merge for incompatibility and rejects the merge. This does not mean that the grammar is incompatible but that the merged contexts make it incompatible. Point 2) is a state within its own closure state/vector environment which is not okay.

```
⟨Structure implementations 44⟩ +≡
  bool state::gen_a_state(gen_context & For_gening_context, state & For_closure_state,
        state & Requesting_state, S_VECTORS_ITER_type & Elem_iter)
  {
    ⟨Increment and printout Recursion counter 137⟩;
    lrclog ≪ "gen_a_state for <" ≪ For_gening_context.for_closure_state→state_no_ ≪ "," ≪
        For_gening_context.gen_vector_ ≪ "> requesting state: " ≪ Requesting_state.state_no_ ≪ endl;
    gen_context associated_rt_bnded_cs(0, −1);
    Voc_ENO eno = Elem_iter→first;
    S_VECTOR_ELEMS_type elem_list = Elem_iter→second;
    S_VECTOR_ELEMS_ITER_type i = elem_list.begin();
    S_VECTOR_ELEMS_ITER_type ie = elem_list.end();
    bool compatible(false);
    for ( ; i ≠ ie; ++i) {    /* read symbol's element list to gen */
      state_element * se = *i;
      ⟨is state's element associated with gened closure state? no bypass 83⟩;
      ⟨common prefix gened goto state? yes deal with its goto state 84⟩;
      ⟨create a new state 85⟩;
      ⟨can new state be merged into state network? yes erase its existance and exit 86⟩;
      add_state_to_gbl_lr1_state_tbls(s);
      ⟨Printout Recursion counter 139⟩;
```

```
      lrclog ≪ "gen_a_state␣for␣<" ≪ For_gening_context.for_closure_state→state_no_ ≪ "," ≪
          For_gening_context.gen_vector_ ≪ ">␣requesting␣state:␣" ≪ Requesting_state.state_no_ ≪
          "␣NEW␣STATE␣CREATED:␣" ≪ s→state_no_ ≪ endl;
      add_state_to_conflict_states_list_if(For_gening_context, *s);
      compatible = is_state_lr1_compatible(*s);    /* is new state to be added lr1 compatible? */
      if (compatible ≡ NOT_LR1_COMPATIBLE) {    /* added */
        ⟨Decrement Recursion counter 140⟩;
        return NOT_LR1_COMPATIBLE;
          /* stop gening: note state added before test as if not lr1 reports properly why */
      }    /* Print_dump_state(s); */
      s→state_type_ = determine_reduced_state_type(s);
      bool gen_ok = gen_transitive_states_balance_for_closure_vector(For_gening_context, For_closure_state,
          *s);
      ⟨Decrement Recursion counter 140⟩;
      return gen_ok;    /* state gened so finished going thru element list */
    }
    ⟨Decrement Recursion counter 140⟩;
    return LR1_COMPATIBLE;
  }
```

83. Is element vector associated with the current closure state being gened?.

```
⟨is state's element associated with gened closure state? no bypass 83⟩ ≡
  if (((se→cs_vector_combo_gening_it_.for_closure_state_ ≡ For_gening_context.for_closure_state_) ∧
        (se→cs_vector_combo_gening_it_.gen_vector_ ≡ For_gening_context.gen_vector_)) ≠ true) {
    ⟨Printout Recursion counter 139⟩;
    lrclog ≪ "gen_a_state␣Bypass␣subrule␣as␣its␣gening␣<" ≪
        se→cs_vector_combo_gening_it_.for_closure_state_→state_no_ ≪ "," ≪
        se→cs_vector_combo_gening_it_.gen_vector_ ≪ ">␣different␣then␣gening␣<" ≪
        For_gening_context.for_closure_state_→state_no_ ≪ "," ≪ For_gening_context.gen_vector_ ≪ ">" ≪
        endl;
    continue;
  }
```

This code is used in section 82.

84. Is element gened from common prefix of an earlier closure state gen?.

```
⟨common prefix gened goto state? yes deal with its goto state 84⟩ ≡
  if (se→goto_state_ ≠ 0) {
    ⟨Printout Recursion counter 139⟩;
    lrclog ≪ "gen_a_state␣subrule␣COMMON␣PREFIX␣state␣gened␣by␣a␣differen\
        t␣gening␣context.␣␣<" ≪ se→cs_vector_combo_gening_it_.for_closure_state_→state_no_ ≪ "," ≪
        se→cs_vector_combo_gening_it_.gen_vector_ ≪ ">␣goto␣state:␣" ≪ se→goto_state_→state_no_ ≪ endl;
    add_state_to_conflict_states_list_if(For_gening_context, *se→goto_state_);
    bool gen_ok = gen_transitive_states_balance_for_closure_vector(For_gening_context, For_closure_state,
        *se→goto_state_);
    ⟨Decrement Recursion counter 140⟩;
    return gen_ok;
  }
```

This code is used in section 82.

85. Create a new state.

```
⟨create a new state 85⟩ ≡
  state *s = new state(eno, se→sr_def_element_);
  s→closure_state_birthing_it_ = For_gening_context.for_closure_state_;
  s→cs_vector_combo_gening_it_ = For_gening_context;
  S_VECTOR_ELEMS_ITER_type j = elem_list.begin();
  S_VECTOR_ELEMS_ITER_type je = elem_list.end();
  bool rt_bnded = s→crt_core_items_of_state(j, je, For_gening_context);
  if (rt_bnded ≡ true) {
    associated_rt_bnded_cs = For_gening_context;
  }
  else {
    associated_rt_bnded_cs.for_closure_state_ = 0;
    associated_rt_bnded_cs.gen_vector_ = −1;
  }
  s→add_closure_rules_subrules_to_state(associated_rt_bnded_cs, *s);
  s→create_follow_sets_of_state();
```

This code is cited in section 146.

This code is used in section 82.

86. Can new state be merged into state network? Yes then exit.
Watch for indicator to stop gening the states caused by "not lr1 compatibile" while gening the closure state network.

```
⟨can new state be merged into state network? yes erase its existance and exit 86⟩ ≡
  int compatibility_result = find_2_states_compatible_and_merge(*s);
  switch (compatibility_result) {
  case MERGED:
    {
      delete s;
      ⟨Decrement Recursion counter 140⟩;
      return true;   /* keep gening */
    }
  case ABORT_GENING_STATES:
    {
      ⟨Decrement Recursion counter 140⟩;
      return NOT_LR1_COMPATIBLE;   /* stop gening */
    }
  case NOT_MERGED:
    {
      break;   /* continue the gen_a_state logic by fall through */
    }
  }
```

This code is used in section 82.

87. *determine_reduced_state_type*.
Determines the "lrness" of the state: no conflict — shift(s) or reduce(s) only, conflict: shift / reduce, multiple reduces, shift(s) with mutiple reduces.

⟨ Structure implementations 44 ⟩ +≡
```
int state::determine_reduced_state_type(state * S)
{   /* rtned 0 so, 1 ro, 2 s/r, 3 r2 , 4 s/r2 */
  using namespace NS_yacco2_T_enum;
  int no_reduces(0);
  int no_reduce_types(0);
  S_VECTORS_ITER_type svi = S→state_s_vector_.begin();
  S_VECTORS_ITER_type svie = S→state_s_vector_.end();
  S_VECTORS_ITER_type tvi = S→state_s_vector_.find(−T_Enum::T_T_eosubrule_);
  if (tvi ≠ svie) {
    no_reduces += tvi→second.size();
    ++no_reduce_types;
  }
  tvi = S→state_s_vector_.find(−T_Enum::T_T_called_thread_eosubrule_);
  if (tvi ≠ svie) {
    no_reduces += tvi→second.size();
    ++no_reduce_types;
  }
  tvi = S→state_s_vector_.find(−T_Enum::T_T_null_call_thread_eosubrule_);
  if (tvi ≠ svie) {
    no_reduces += tvi→second.size();
    ++no_reduce_types;
  }
  if (no_reduces > 1) no_reduces = 3;
  if (S→state_s_vector_.size() > no_reduce_types) {   /* shift present */
    if (no_reduce_types > 0) {   /* combo */
      ++no_reduces;   /* combo shift / reduce */
    }
  }
  return no_reduces;
}
```

88. *add_state_to_gbl_lr1_state_tbls*.

⟨ Structure implementations 44 ⟩ +≡
```
void state::add_state_to_gbl_lr1_state_tbls(state * State)
{
  ++NO_LR1_STATES;
  State→state_no_ = NO_LR1_STATES;
  LR1_STATES.push_back(State);
  Voc_ENO eno = State→vectored_into_by_elem_;
  LR1_STATES_ITER_type i = LR1_COMMON_STATES.find(eno);
  if (i ≡ LR1_COMMON_STATES.end()) {
    LR1_COMMON_STATES[eno] = STATES_type();
    i = LR1_COMMON_STATES.find(eno);
  }
  i→second.push_back(State);
}
```

89. *add_state_to_conflict_states_list_if* **.**
If newly created state has "eosubrule" in its vector map, there are 2 possibilities that make it a conflict state:
1) reduce / reduce — more than 1 production whose string is consumned
2) shift / reduce — one reduce with a shift

⟨Structure implementations 44⟩ +≡
```
void state::add_state_to_conflict_states_list_if(gen_context &Gening_context, state &State)
{
  S_VECTORS_ITER_type i = State.state_s_vector_.find(−T_Enum::T_T_eosubrule_);
  if (i ≠ State.state_s_vector_.end()) goto reduce_fnd;
  return;
reduce_fnd: ;
  if (i→second.size() > 1) {    /* reduce / reduce */
    Gening_context.for_closure_state_→state_s_conflict_state_list_.push_back(&State);
    return;
  }
  if (State.state_s_vector_.size() < 2) return;
  Gening_context.for_closure_state_→state_s_conflict_state_list_.push_back(&State);
}
```

90. General routines on state compatibilities.

91. Determining if 2 states are equivalent?.
As an aid to quickly determine whether 2 states are equal, each state except the "closure-state" has an element going into it: proxies "rule-ref" or "T-ref" are references to their definitions. They are general classifications of the items whose contents refer to the specific definition. "eosubrule" is excluded from this as it does not generate any states. Why use the definitions? I need a unique identifier and this only comes from the definition. The "closure-only" (start) state has no entry.

State equivalence is arrived at by:

1) the "entered into" element generating each state must be identical
2) State's A "to vector" map must be the same size as B
3) A's "to vector" 's keys must be the same as B
4) A's "to vector" 's state element list's contents must be the same as B

Point 4: why the state's element list ordered? One can have a state whereby its elements order are not the same but the "to vector" result is. Give me a real example as i'm a doubter.

Rule		*subrule's symbols*
Rab	→	a b
Rac	→	a c
Rad	→	Rab
	→	Rac
Rae	→	Rac
	→	Rab

Rad or Rae inside other productions should allow the merge of Rab, Rac.

Constraints:

1) state's vector is a map ordered by Voc_ENO
2) state elements list ordered by tree address

⟨ Structure implementations 44 ⟩ +≡

```
bool state::are_states_equivalent(state & Merge_into_state, state & To_merge_state)
{
  if (Merge_into_state.vectored_into_by_elem_ ≠ To_merge_state.vectored_into_by_elem_) {
    return false;
  }
  if (Merge_into_state.state_s_vector_.size() ≠ To_merge_state.state_s_vector_.size()) {
    return false;
  }
  S_VECTORS_ITER_type i = Merge_into_state.state_s_vector_.begin();
  S_VECTORS_ITER_type ie = Merge_into_state.state_s_vector_.end();
  S_VECTORS_ITER_type j = To_merge_state.state_s_vector_.begin();
  S_VECTORS_ITER_type je = To_merge_state.state_s_vector_.end();
  for ( ; i ≠ ie; ++i, ++j) {
    Voc_ENO ieno = i→first;
    Voc_ENO jeno = j→first;
    if (ieno ≠ jeno) {
      return false;
    }
    if (i→second.size() ≠ j→second.size()) {
      return false;
    }
    S_VECTOR_ELEMS_ITER_type l = i→second.begin();
    S_VECTOR_ELEMS_ITER_type le = i→second.end();
    S_VECTOR_ELEMS_ITER_type m = j→second.begin();
    S_VECTOR_ELEMS_ITER_type me = j→second.end();
```

```
    for ( ; l ≠ le; ++l, ++m) {
      state_element *ls = *l;
      state_element *ms = *m;
      if (ls→sr_element_ ≠ ms→sr_element_) {
        return false;
      }
    }
  }
  return true;
}
```

92. *is_state_lr1_compatible*.

0) check if it's a s/r or r/r type state no exit as compatible
1) Fill in its lookahead per reducing subrule
1.5) make sure that its reducing set is not empty! or not lr1 compatible
2) calculate state's T shift set
3) do a set intersection on all these sets

The resulting set must be empty to be LR1 compatible. To be more efficient i use a T count as i do not have to report on what type of non compatibility produced it nor between whom: reduce / reduce or shift /reduce. The cost is to read each set while being gened and add up each item's referenced count: not the combinatorics between each set.

Add check on eolr presence: Eg, eolr in la expression only, it is not exploded. This leads to the bug that a shift/reduce is incompatible. Somehow i must have optimized this out from *la_expr* grammar. So if there are 2 or more reduces taking place and eolr is present -¿ not lr1. Same goes for shift/reduce condition.

⟨Structure implementations 44⟩ +≡

```
bool state::is_state_lr1_compatible(state & State_to_eval)
{
  T_COUNT_type t_cnt(START_OF_RULES_ENUM);
  for (int x = 0; x < START_OF_RULES_ENUM; ++x) t_cnt[x] = 0;
  S_VECTORS_ITER_type i;
  i = State_to_eval.state_s_vector_.find(-T_Enum::T_T_eosubrule_);
  if (i ≠ State_to_eval.state_s_vector_.end()) {
    goto assess_state;
  }
  i = State_to_eval.state_s_vector_.find(-T_Enum::T_T_called_thread_eosubrule_);
  if (i ≠ State_to_eval.state_s_vector_.end()) {
    goto assess_state;
  }
  i = State_to_eval.state_s_vector_.find(-T_Enum::T_T_null_call_thread_eosubrule_);
  if (i ≠ State_to_eval.state_s_vector_.end()) {
    goto assess_state;
  }
  return LR1_COMPATIBLE;
assess_state:
  int no_reduces = 0;
  S_VECTOR_ELEMS_ITER_type j = i→second.begin();
  S_VECTOR_ELEMS_ITER_type je = i→second.end();
  bool T_not_meta(false);
  for ( ; j ≠ je; ++j) {      /* fill in lookahead per reducing subrule */
    ++no_reduces;
    state_element *se = *j;
    if (se→calc_la(*se) ≡ false) {
      return NOT_LR1_COMPATIBLE;
    }
    LA_SET_ITER_type k = se→la_set_→begin();
    LA_SET_ITER_type ke = se→la_set_→end();
    for ( ; k ≠ ke; ++k) {
      T_in_stbl *tintbl = *k;
      T_ENO teno = tintbl→t_def()→enum_id();      /* T of new alphabet */
      if (teno ≤ END_OF_LR1_DEFS) {
        if (teno ≡ LR1_EOG) {
          T_not_meta = true;
```

```
          }
        }
        else {
          T_not_meta = true;
        }
        ++t_cnt[teno];
        if (t_cnt[teno] > 1) return NOT_LR1_COMPATIBLE;
      }
    }
    if (no_reduces > 1) {      /* reduces ¿ 1 where eolr super set to others */
      if (T_not_meta ≡ true) {
        if (t_cnt[LR1_EOLR] > 0) return NOT_LR1_COMPATIBLE;
      }
    }     /* shift / reduce evaluation */      /* shift type not of meta Tes |?|, |+|etc */
    S_VECTORS_ITER_type l = State_to_eval.state_s_vector_.begin();
    S_VECTORS_ITER_type le = State_to_eval.state_s_vector_.end();
    for ( ; l ≠ le; ++l) {      /* T shift set of state */
      Voc_ENO t = l→first;
      if (t ≡ −T_Enum::T_T_eosubrule_) continue;
      if (t ≡ −T_Enum::T_T_null_call_thread_eosubrule_) continue;
      if (t ≡ −T_Enum::T_T_called_thread_eosubrule_) continue;
      if (t ≡ T_Enum::T_refered_rule_) continue;
      S_VECTOR_ELEMS_ITER_type m = l→second.begin();
      state_element *se = *m;
      CAbs_lr1_sym *sym = AST::content(*se→sr_element_);
      T_ENO tid = sym→enumerated_id__;
      switch (tid) {
      case T_Enum::T_refered_T_:
        {
          ⟨get cast referenced T 109⟩;
          T_in_stbl *tintbl = rt→t_in_stbl();
          T_ENO teno = tintbl→t_def()→enum_id();      /* T of new alphabet */
          if (teno ≤ END_OF_LR1_DEFS) {
            if (teno ≠ LR1_EOG) {
              continue;      /* bypass the meta Tes */
            }
          }
          ++t_cnt[teno];
          if (t_cnt[teno] > 1) return NOT_LR1_COMPATIBLE;
          if (teno > END_OF_LR1_DEFS) {      /* specific shift against eolr usuage */
            if (t_cnt[LR1_EOLR] > 0) return NOT_LR1_COMPATIBLE;
          }
        }
      }
    }
    return LR1_COMPATIBLE;
  }
```

93. *are_2_states_compatible_yes_merge*.
Is it LR1 compatible? If *To_merge_into_state*'s closured-generating state has no conflict states then it's straight sailing. To determine if the mariage is compatible, the conflict states must be checked with the new follow set info supplied by *State_for_merging* birthing "closured context". Let's review the LR1 conditions:

1) a conflict state has at least a reduce / reduce or reduce / shift condition
2) reduced lookaheads must be regened with the new closured follow set context

Point 2's "new closured follow set" context comes from the closured states of *State_for_merging*'s productions. To do the check i must merge the new follow set contexts of *To_merge_into_state*'s birth state per transitive production. Why? These are the potential follow set contexts where each production was born. Now generate the lookahead for each conflict state's reduces and then see whether all the conflict states are kosher. An incompatibility just requires rolling back the new follow set context from each of the proposed production's birthing states of *To_merge_into_state*.

A kosher merge requires that the spawning state must unlink each of its spawned productions to the new state's "goto" and "reduce" links and latch into the merged state's productions linkages.

A caveat: **infinite states gened when a merge into its own closure state network is not lr(1)**. Spector's **/usr/local/yacco2/qa/lr1_sp2.lex** grammar illustrates when the closure state 2's start gening vector:**x** produces its own states. Now state 2's brethern vector: **y** gets gened and has the potential to merge into one of x's states but the potential merge is not lr(1). Cuz state's 2 two closure state's vectors are different(x,y), this means that their gened states can be separate as long as the lr(1) constraint is respected. So y's states are separate from vectored:x and not merged into x's states due to lr(1) constraint. So continue gening closure state 2's y's states. If down the road gening y's states are not lr(1) then abort the generation and issue such a message. The not lr(1) condition is specific to the closure state's gening vector and not due to a merge into another's state network that is kosher.

What happens when a closure state's vector is gened and one of its states is not lr(1) whether being merged into or not within itsself states network? **/usr/local/yacc2/qa/knu1_sick.lex** grammar illustrates this situation. Have a read of its pdf document, as it talks about the situation and how it came about. This situation is a legitimate non lr(1) grammar and so put on the brakes to generating its states. Another example is: **lr1_sp6.lex** where the start rule has a subrule (production) and it is epsilonable. Ditto on the reading.

So the terminating condition is:
1) not lr(1) within its own closure state's vector generated states

```
⟨Structure implementations 44⟩ +≡
  int state::are_2_states_compatible_yes_merge(state & To_merge_into_state, state & State_for_merging)
  {
    state * cs_To_merge_into = To_merge_into_state.closure_state_birthing_it_;
    state * cs_for_merging = State_for_merging.closure_state_birthing_it_;
    ⟨add potential follow set context per production 94⟩;
    bool compatible(false);
    if (cs_To_merge_into→state_s_conflict_state_list_.empty() ≡ true) goto merged;
  lr1_test:
    {
      S_CONFLICT_STATES_ITER_type k = cs_To_merge_into→state_s_conflict_state_list_.begin();
      S_CONFLICT_STATES_ITER_type ke = cs_To_merge_into→state_s_conflict_state_list_.end();
      for ( ; k ≠ ke; ++k) {    /* evaluate lr1 compatibility */
        state * s = *k;
        compatible = s→is_state_lr1_compatible(*s);
        if (compatible ≡ NOT_LR1_COMPATIBLE) goto unwind_merge;
      }
```

```
    }
  merged:
    {
      ⟨ relink spawning state of merged state 95 ⟩;
      return MERGED;
    }
  unwind_merge:
    {
    unwind: ;
      ⟨ unwind potential merge 97 ⟩;
      return NOT_MERGED;
    }
    return NOT_MERGED;
  }
```

94. Add potential follow set context per production.
Only work with the core items.

⟨ add potential follow set context per production 94 ⟩ ≡

```
  RULE_NOS_SET_type rules_to_add;
  S_VECTORS_ITER_type sfmi = To_merge_into_state.state_s_vector_.begin( );
  S_VECTORS_ITER_type sfmie = To_merge_into_state.state_s_vector_.end( );
  S_VECTORS_ITER_type msfmi = State_for_merging.state_s_vector_.begin( );
  for ( ; sfmi ≠ sfmie; ++sfmi, ++msfmi) {
    S_VECTOR_ELEMS_ITER_type rri = sfmi→second.begin( );
    S_VECTOR_ELEMS_ITER_type mrri = msfmi→second.begin( );
    S_VECTOR_ELEMS_ITER_type rrie = sfmi→second.end( );
    for ( ; rri ≠ rrie; ++rri, ++mrri) {
      state_element *re = *rri;
      state_element *mre = *mrri;
      if (re→closure_state_ ≡ re→self_state_) continue;    /* not a core item */
      RULE_ENO ruleno = re→find_state_element_s_rule_no( );
      S_FOLLOW_SETS_ITER_type j = re→closure_state_→state_s_follow_set_map_.find(ruleno);
      follow_element *fe = j→second;
      RULE_NOS_SET_ITER_type rni = rules_to_add.find(ruleno);
      if (rni ≡ rules_to_add.end( )) {
        fe→add_merge_closure_info(*mre→closure_state_);
        rules_to_add.insert(ruleno);
      }
    }
  }
```

This code is used in section 93.

95. Relink spawning state of merged state.

⟨relink spawning state of merged state 95⟩ ≡

```
S_VECTORS_ITER_type ri = State_for_merging.state_s_vector_.begin();
S_VECTORS_ITER_type rie = State_for_merging.state_s_vector_.end();
S_VECTORS_ITER_type si = To_merge_into_state.state_s_vector_.begin();
for ( ; ri ≠ rie; ++ri, ++si) {
  S_VECTOR_ELEMS_ITER_type rri = ri→second.begin();
  S_VECTOR_ELEMS_ITER_type rrie = ri→second.end();
  S_VECTOR_ELEMS_ITER_type ssi = si→second.begin();
  for ( ; rri ≠ rrie; ++rri, ++ssi) {
    state_element *re = *rri;
    state_element *se = *ssi;
    if (re→previous_state_element_ ≡ 0) continue;
    state_element *prev_re = re→previous_state_element_;
    prev_re→goto_state_ = se→self_state_;
    prev_re→reduced_state_ = se→reduced_state_;
    prev_re→next_state_element_ = se;    /* walk thru the rhs backwards laying those reduce eggs */
    for (prev_re = prev_re→previous_state_element_; prev_re ≠ 0; prev_re = prev_re→previous_state_element_)
      {    /* deposit reducing state */
      prev_re→reduced_state_ = se→reduced_state_;
    }
  }
}
⟨add conflict states to to merge network 96⟩;
```

This code is used in section 93.

96. Add conflict states to merge network.

For now add all the conflict states until i refine a map of states that derives each specific conflict state.

If the merged state is part of the current network being gened then bypass. Why? The conflict states are already registered that appliy to this merged state cuz its part of this closured generating state containing the conflist state list.

⟨add conflict states to to merge network 96⟩ ≡

```
if (cs_for_merging→cs_vector_combo_gening_it_.for_closure_state_ ≠
      cs_To_merge_into→cs_vector_combo_gening_it_.for_closure_state_) {
  S_CONFLICT_STATES_ITER_type k = cs_To_merge_into→state_s_conflict_state_list_.begin();
  S_CONFLICT_STATES_ITER_type ke = cs_To_merge_into→state_s_conflict_state_list_.end();
  for ( ; k ≠ ke; ++k) {
    state *s = *k;
    cs_for_merging→state_s_conflict_state_list_.push_back(s);
  }
}
```

This code is used in section 95.

97. Unwind potential merge.

⟨ unwind potential merge 97 ⟩ ≡

```
RULE_NOS_SET_type rules_to_add;
S_VECTORS_ITER_type sfmi = To_merge_into_state.state_s_vector_.begin();
S_VECTORS_ITER_type sfmie = To_merge_into_state.state_s_vector_.end();
for ( ; sfmi ≠ sfmie; ++sfmi) {
  S_VECTOR_ELEMS_ITER_type rri = sfmi→second.begin();
  S_VECTOR_ELEMS_ITER_type rrie = sfmi→second.end();
  for ( ; rri ≠ rrie; ++rri) {
    state_element *re = *rri;
    if (re→closure_state_ ≡ re→self_state_) continue;   /* not a core item */
    RULE_ENO ruleno = re→find_state_element_s_rule_no();
    S_FOLLOW_SETS_ITER_type j = re→closure_state_→state_s_follow_set_map_.find(ruleno);
    follow_element *fe = j→second;
    RULE_NOS_SET_ITER_type rni = rules_to_add.find(ruleno);
    if (rni ≡ rules_to_add.end()) {
      fe→remove_merge_closure_info();
      rules_to_add.insert(ruleno);
    }
  }
}
```

This code is used in section 93.

98. *find_2_states_compatible_and_merge*.

Read the LR1_COMMON_STATES table looking for states with the same enumerate and same state core items.

⟨ Structure implementations 44 ⟩ +≡

```
int state::find_2_states_compatible_and_merge(state &State_for_merging)
{
  Voc_ENO eno = State_for_merging.vectored_into_by_elem_;
  LR1_STATES_ITER_type i = LR1_COMMON_STATES.find(eno);
  if (i ≡ LR1_COMMON_STATES.end()) {
    return NOT_MERGED;
  }
  STATES_ITER_type j = i→second.begin();
  STATES_ITER_type je = i→second.end();
  for ( ; j ≠ je; ++j) {
    state *s = *j;
    bool equivalent = are_states_equivalent(State_for_merging, *s);
    if (equivalent ≡ false) continue;
    int compatible = are_2_states_compatible_yes_merge(*s, State_for_merging);
    if (compatible ≡ LR1_COMPATIBLE) return MERGED;
    if (compatible ≡ ABORT_GENING_STATES) return ABORT_GENING_STATES;
  }
  return NOT_MERGED;
}
```

99. *are_gened_states_lr1_compatible*.
Read gening state's conflict states for lr1 health check.

```
⟨ Structure implementations 44 ⟩ +≡
  bool state::are_gened_states_lr1_compatible()
  {
    S_CONFLICT_STATES_ITER_type i = state_s_conflict_state_list_.begin();
    S_CONFLICT_STATES_ITER_type ie = state_s_conflict_state_list_.end();
    if (i ≡ ie) return LR1_COMPATIBLE;
    for ( ; i ≠ ie; ++i) {
      state *s = *i;
      if (is_state_lr1_compatible(*s) ≡ NOT_LR1_COMPATIBLE) return NOT_LR1_COMPATIBLE;
    }
    return LR1_COMPATIBLE;
  }
```

100. *is_str_rt_bnded*.
Determine if the production's lookahead transitions thru other closured state's follow sets.
Read the production's string positioned by it's passed parameter. The right bounded condition is whether the last symbol in the string is a rule before the "eosubrule". This demands that the closure productions caused by this rule within the state must also be generated as part of the being gened closure state.

A subtle condition arises when the string's balance of symbols contains only rules that are all epsilonable. This moves the right bounded condition into the interior of the production currently positioned. Why? Epsilon is a window that allows one to see past its contents into the next adjacent symbol. This is inductive as it moves right thru all the epsilon rules to the end-of-string. Thus each rule's follow set strings transitizes rightward along the production's string to its end and then up the follow sets of its spawning closure rule's environment.

Please note, the thread call variants do not support right-bounded expressions. U'll never have a rule following the thread call prase — this is a parsing error.

```
⟨ Structure implementations 44 ⟩ +≡
  bool state::is_str_rt_bnded(AST *Str)
  {
    AST *rstr_t = AST::brother(*Str);     /* get next node */
    if (rstr_t ≡ 0) return false;     /* eosubule passed in for verification */
    CAbs_lr1_sym *sym = AST::content(*rstr_t);
    Voc_ENO id = sym→enumerated_id__;
    if (id ≡ T_Enum::T_T_eosubrule_) {
      CAbs_lr1_sym *rsym = AST::content(*Str);
      if (rsym→enumerated_id__ ≡ T_Enum::T_refered_rule_) {
        return true;
      }
    }
    return is_str_epsilonable(rstr_t);
  }
```

101. *is_str_epsilonable*.
The symbol string passed is one to the right of the current item within the state called the "follow string". From this point within the production string, the balance of the symbols must be composed of "refered-rule" having the epsilon attribute. Remember: the grammar is internally represented by a tree where each node of a production contains a "refered-rule", "refered-T", or an "eosubrule". Thread calls is not applicable. It is never epsilonable.

```
⟨Structure implementations 44⟩ +≡
  bool state::is_str_epsilonable(AST * Str)
  {
    AST * str_t = Str;
    for ( ; str_t ≠ 0; str_t = AST::brother(*str_t)) {
      CAbs_lr1_sym * sym = AST::content(*str_t);
      Voc_ENO id = sym→enumerated_id__;
      ⟨is str's element epsilon 102⟩;
    }
    return true;
  }
```

102. Is str's element epsilon?.
The tree node's symbol must be a rule or "eosubrule". "eosubrule" indicates either the empty string (epsilon) where it is the only symbol within the production or the end of the production's string of symbols.

```
⟨is str's element epsilon 102⟩ ≡
  switch (id) {
  case T_Enum::T_refered_rule_:
    {
      ⟨get cast referenced rule 108⟩;
      rule_def * rd = rr→its_rule_def( );
      if (rd→epsilon( ) ≠ true) return false;
      break;
    }
  case T_Enum::T_T_eosubrule_:
    {
      return true;
    }
  case T_Enum::T_refered_T_:
    {
      return false;
    }
  }
```

This code is used in section 101.

103. Follow set routines.

104. *add_rule_to_follow_list*.

⟨ Structure implementations 44 ⟩ +≡

```
void state::add_rule_to_follow_list(RULE_ENO Refered_rule)
{
  FOLLOW_RULES_ITER_type i = follow_rule_list_.find(Refered_rule);
  if (i ≡ follow_rule_list_.end()) {
    follow_rule_list_.insert(Refered_rule);
  }
}
```

105. *create_follow_sets_of_state*.

The list contains the rule's enumerate value. This is used to get the *state_s_vector_*[*rule no*] context that makes up the state. It supplies the context in the subrule(s) where it resides. The follow set is calculated from the symbol string(s) to the right of these subrule context.

⟨ Structure implementations 44 ⟩ +≡

```
void state::create_follow_sets_of_state()
{
  if (follow_rule_list_.empty() ≡ true) return;
  RULE_ENO eno(−1);
  FOLLOW_RULES_ITER_type i = follow_rule_list_.begin();
  FOLLOW_RULES_ITER_type ie = follow_rule_list_.end();
  for ( ; i ≠ ie; ++i) {    /* create initial follow set map entries */
    follow_element *fe = new follow_element(this);
    eno = *i;
    fe→rule_no_ = eno;
    state_s_follow_set_map_[eno] = fe;
  }
  i = follow_rule_list_.begin();
  for ( ; i ≠ ie; ++i) {    /* read rules */
    eno = *i;
    S_VECTORS_ITER_type j = state_s_vector_.find(eno);
    S_FOLLOW_SETS_ITER_type fsi = state_s_follow_set_map_.find(eno);
    follow_element *fe = fsi→second;
    S_VECTOR_ELEMS_ITER_type k = j→second.begin();
    S_VECTOR_ELEMS_ITER_type ke = j→second.end();
    for ( ; k ≠ ke; ++k) {    /* read subrule context */
      state_element *se = *k;
      ⟨ get subrule's referenced rule in follow string 107 ⟩;
      AST *follow_str_t = AST::brother(*se→sr_element_);
    deal_with_follow_str_sym: ;
      fe→rule_def_t_ = rr→its_rule_def()→rule_s_tree();
      fe→add_follow_set_contributor(follow_str_t);
      CAbs_lr1_sym *sym = AST::content(*follow_str_t);
      Voc_ENO id = sym→enumerated_id__;
      ⟨ deal with current follow string element 106 ⟩;
    }
  }
}
```

106.

⟨ deal with current follow string element 106 ⟩ ≡

```
switch (id) {
case T_Enum::T_refered_rule_:
  {
    ⟨ get cast referenced rule 108 ⟩;
    rule_def *rd = rr→its_rule_def();
    if (rd→first_set()→empty() ≠ true) {
      FIRST_SET_ITER_type a = rd→first_set()→begin();
      FIRST_SET_ITER_type ae = rd→first_set()→end();
      for ( ; a ≠ ae; ++a) {
        T_in_stbl *t = *a;
        if (fe→follow_set_.find(t) ≡ fe→follow_set_.end()) {
          fe→follow_set_.insert(t);
        }
      }
    }
    if (rd→epsilon() ≡ true) {    /* next follow str symbol */
      follow_str_t = AST::brother(*follow_str_t);
      goto deal_with_follow_str_sym;
    }
    break;
  }
case T_Enum::T_T_eosubrule_:
  {
    ⟨ get cast referenced eosubrule 110 ⟩;
    fe→add_follow_set_transition(*se, *eos);
    break;
  }
case T_Enum::T_T_null_call_thread_eosubrule_:
  {
    break;
  }
case T_Enum::T_T_called_thread_eosubrule_:
  {
    break;
  }
case T_Enum::T_refered_T_:
  {
    ⟨ get cast referenced T 109 ⟩;
    fe→follow_set_.insert(rt→t_in_stbl());
    break;
  }
}
```

This code is used in section 105.

107.

⟨ get subrule's referenced rule in follow string 107 ⟩ ≡

```
refered_rule*␣rr␣=␣(refered_rule*)AST::content(*se->sr_element_);
```

This code is used in section 105.

108.

⟨ get cast referenced rule 108 ⟩ ≡

```
refered_rule*␣rr␣=␣(refered_rule*)sym;
```

This code is used in sections 47, 102, and 106.

109.

⟨ get cast referenced T 109 ⟩ ≡

```
refered_T*␣rt␣=␣(refered_T*)sym;
```

This code is used in sections 47, 92, and 106.

110.

⟨ get cast referenced eosubrule 110 ⟩ ≡

```
T_eosubrule*␣eos␣=␣(T_eosubrule*)sym;
```

This code is used in sections 47, 52, and 106.

111.

⟨ get cast referenced null called thread eosubrule 111 ⟩ ≡

```
T_null_call_thread_eosubrule*␣eos␣=␣(T_null_call_thread_eosubrule*)sym;
```

This code is used in section 47.

112.

⟨ get cast referenced called thread eosubrule 112 ⟩ ≡

```
T_called_thread_eosubrule*␣eos␣=␣(T_called_thread_eosubrule*)sym;
```

This code is used in section 47.

113. *crt_start_rule_s_follow_set*.

What is the follow set for the Start rule and should there be one anyway? There is only the empty string so what's it follow string? Empty? Here's the scoop. A monolithic grammar force an "eolr" terminal. This covers the end-of-the grammar process and homogenizes it in the same manner as threaded grammars. For the thread grammar use its calculated "parallel-la-boundary" expression. This is its follow set. The second question raised allows the Start rule's productions to reduce as accepted.

⟨ Structure implementations 44 ⟩ +≡

```
void state::crt_start_rule_s_follow_set(AST & Start_rule)
{
  CAbs_lr1_sym * sym = AST::content(Start_rule);
  rule_def* rd = (rule_def*)sym; ;
  RULE_ENO rno = rd→enum_id();
  follow_element * fe = new follow_element(this);
  fe→rule_no_ = rno;
  state_s_follow_set_map_[rno] = fe;
  fe→rule_def_t_ = rd→rule_s_tree();
  if (O2_PP_PHASE ≡ 0) {    /* monolithic: use eolr not eog */
    fe→follow_set_.insert(STBL_T_ITEMS[LR1_EOLR]);
  }
  else {    /* thread: use calculate lookahead expression */
    T_parallel_parser_phrase * pp_ph = O2_PP_PHASE;
    T_parallel_la_boundary * la = pp_ph→la_bndry();
    LA_SET_ITER_type i = la→la_first_set()→begin();
    LA_SET_ITER_type ie = la→la_first_set()→end();
    for ( ; i ≠ ie; ++i) {
      T_in_stbl * t = *i;
      FOLLOW_SETS_ITER_type j = fe→follow_set_.find(t);
      if (j ≡ fe→follow_set_.end()) {
        fe→follow_set_.insert(t);
      }
    }
  }
}
```

114. For tracing facilities.

115. *entry_symbol_literal*.

```
⟨ Structure implementations 44 ⟩ +≡
  const char *state::entry_symbol_literal()
  {
    if (vectored_into_by_elem_sym_ ≡ 0) return "␣No␣symbol";
    ⟨ return entry symbol literal 116 ⟩;
  }
```

116. Return entry symbol literal.
Eases tracing of lr states easier instead of just displaying its enumerated value.

```
⟨ return entry symbol literal 116 ⟩ ≡
  Voc_ENO id = vectored_into_by_elem_sym_→enumerated_id__;
  CAbs_lr1_sym * sym = vectored_into_by_elem_sym_;
  switch (id) {
  case T_Enum::T_rule_def_:
    {
      rule_def*␣rd␣=␣(rule_def*)sym;
      return rd→rule_name()→c_str();
    }
  case T_Enum::T_T_eosubrule_:
    {
      return "eos";
      break;
    }
  case T_Enum::T_T_null_call_thread_eosubrule_:
    {
      return "null-eos";
      break;
    }
  case T_Enum::T_T_called_thread_eosubrule_:
    {
      return "called-thd-eos";
      break;
    }
  case T_Enum::T_T_terminal_def_:
    {
      T_terminal_def*␣td␣=␣(T_terminal_def*)sym;
      return td→t_name()→c_str();
    }
  default:
    {   /* cuz apple's latest compiler: symantic analysis error-never reached */
      return "null-eos";
      break;
    }
  }
```

This code is used in section 115.

117. Emit FSA, FSC, and Documents of grammar.
Finally. This is a hodge/podge of routines to emit the "cpp" code, "first set control" file for O_2linker, and various documents to be run thru the "mpost, cweave, and pdftex' stages. A caution, under my Sun Solaris system, the "Adobe 3rd party Readers" like "gpdf and xpdf" occassionally throw tantrums. I'll look into these after finishing O_2.

118. Output enumeration header file.

```
⟨output enumeration header file 118⟩ ≡
  lrclog ≪ "Output␣enumeration␣header␣file␣" ≪ endl;
  OP_ENUMERATION_HEADER(Error_queue);
  ⟨if error queue not empty then deal with posted errors 21⟩;
```

This code is used in section 130.

119. Output Errors file.

```
⟨output Errors files 119⟩ ≡
  if (ERR_SW ≡ 'y') {
    lrclog ≪ "Output␣Errors␣vocabulary␣files␣" ≪ endl;
    OP_ERRORS_HEADER(Error_queue);
    OP_ERRORS_CPP(Error_queue);
    ⟨if error queue not empty then deal with posted errors 21⟩;
  }
```

This code is used in section 130.

120. Output User Terminals files.

```
⟨output User Terminals files 120⟩ ≡
  if (T_SW ≡ 'y') {
    lrclog ≪ "Output␣User␣Terminal␣vocabulary␣files␣" ≪ endl;
    OP_USER_T_HEADER(Error_queue);
    OP_USER_T_CPP(Error_queue);
    ⟨if error queue not empty then deal with posted errors 21⟩;
  }
```

This code is used in section 130.

121. Output grammar header file.

```
⟨output grammar header file 121⟩ ≡
  lrclog ≪ "Output␣grammar␣header␣file␣" ≪ endl;
  OP_GRAMMAR_HEADER(Error_queue);
  ⟨if error queue not empty then deal with posted errors 21⟩;
```

This code is used in section 130.

122. Output grammar cpp file.

```
⟨output grammar cpp file 122⟩ ≡
  lrclog ≪ "Output␣grammar␣cpp␣file␣" ≪ endl;
  OP_GRAMMAR_CPP(Error_queue);
  ⟨if error queue not empty then deal with posted errors 21⟩;
```

This code is used in section 130.

123. Output grammar sym file.

```
⟨ output grammar sym file 123 ⟩ ≡
  lrclog ≪ "Output␣grammar␣sym␣file␣" ≪ endl;
  OP_GRAMMAR_SYM(Error_queue);
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
```

This code is used in section 130.

124. Output grammar tbl file.

```
⟨ output grammar tbl file 124 ⟩ ≡
  lrclog ≪ "Output␣grammar␣tbl␣file␣" ≪ endl;
  OP_GRAMMAR_TBL(Error_queue);
  ⟨ if error queue not empty then deal with posted errors 21 ⟩;
```

This code is used in section 130.

125. Output T-alphabet file.

```
⟨ output T-alphabet file 125 ⟩ ≡
  if (T_SW ≡ 'y' ∨ ERR_SW ≡ 'y') {
    lrclog ≪ "Output␣User␣Terminal␣vocabulary␣files␣" ≪ endl;
    OP_T_Alphabet(Error_queue);
    ⟨ if error queue not empty then deal with posted errors 21 ⟩;
  }
```

This code is used in section 130.

126. Documents — Grammar's Cweb and Mpost diagrams, and Cross references.

```
⟨ emit documents 126 ⟩ ≡
  if (PRT_SW ≡ 'y') {⟨ emit Cweb and Mpost files 127 ⟩⟨ print out xref docs 128 ⟩}
```

This code is used in section 130.

127. Emit Cweb and Mpost diagrams.
This is my attempt at producing grammar reports by *mpost* diagrams and *cweave*. Of course, this is recursive in that cweb uses *cweave* to generate code with a simple attempt at grammar's railroad diagrams. To do this, a new comment type was created — "/@" – "@/" to include the *cweb* directives declaring how the document's formatting will look like with its table of contents. The comment is a play on *cweave*'s directives. They can be sprinkled thru out the grammar except within the syntax-directed code constructs as u know, this is a character at a time lex crawler with little knowledge of c++ syntax apart from comments, literals, and strings. Once u become familiar with *cweave*, u'll never go back to base comments — aka c++. The files emitted take on the file naming format of:

1) grammar name without its extension ".lex" for the filename body
2) ".mp" mpost extension
3) ".w"cweb extension

As an example, a grammar of "eol.lex" would produce the *mpost* file 'eol.mp' and *cweb* file 'eol.w'. These files are run thru *mpost* and *cweave* followed by *pdftex* to produce the 'pdf' grammar document. *mpost* generates from 'eol.mp' its figure files with numeric extensions like 'eol.1' that are referenced in the *cweb* file 'eol.w' using the *convertMPtoPDF* macro.

```
⟨ emit Cweb and Mpost files 127 ⟩ ≡
  using namespace yacco2;
  using namespace NS_yacco2_k_symbols;
  using namespace NS_mpost_output;
  lrclog << "Emit␣grammar's␣railroad␣diagrams␣for␣Mpost" << endl;
  set < int > cweb_k_filter;
  cweb_k_filter.insert(T_Enum::T_T_cweb_comment_);
  tok_can_ast_functor mpost_just_walk_functr;
  ast_prefix mpost_rule_walk(*GRAMMAR_TREE, &mpost_just_walk_functr, &cweb_k_filter, BYPASS_FILTER);
  tok_can < AST *> mpost_rules_can(mpost_rule_walk);
  Cmpost_output mpost_fsm;
  T_fsm_phrase *fsm_ph = O2_FSM_PHASE;
  mpost_fsm.grammar_filename_prefix_ += fsm_ph→filename_id()→identifier()→c_str();
  mpost_fsm.fq_filename_noext_ += o2_fq_fn_noext.c_str();
  Parser mpost_rules(mpost_fsm, &mpost_rules_can, 0, 0, &Error_queue);
  mpost_rules.parse();
```

This code is used in section 126.

128. Print out xref docs.
1) cross reference of used symbols: grammar's vocabulary
2) grammar's rules first sets

```
⟨ print out xref docs 128 ⟩ ≡
  using namespace NS_prt_xrefs_docs;
  yacco2::lrclog << "-----␣Print␣xref␣docs␣-----" << std::endl;
  set<int>␣prt_xrefs_docs_filter;
  prt_xrefs_docs_filter.insert(T_Enum::T_rule_def_);
  prt_xrefs_docs_filter.insert(T_Enum::T_T_subrule_def_);
  prt_xrefs_docs_filter.insert(T_Enum::T_refered_T_);
  prt_xrefs_docs_filter.insert(T_Enum::T_refered_rule_);
  prt_xrefs_docs_filter.insert(T_Enum::T_T_called_thread_eosubrule_);
  prt_xrefs_docs_filter.insert(T_Enum::T_T_null_call_thread_eosubrule_);
  prt_xrefs_docs_filter.insert(T_Enum::T_T_eosubrule_);
  tok_can_ast_functor xrefs_docs_walk_functr;
  ast_prefix prt_xrefs_docs_walk(*rules_tree, &xrefs_docs_walk_functr, &prt_xrefs_docs_filter, ACCEPT_FILTER);
  tok_can < AST *> prt_xrefs_docs_can(prt_xrefs_docs_walk);
  Cprt_xrefs_docs prt_xrefs_docs_fsm;
  prt_xrefs_docs_fsm.grammar_filename_prefix_ += mpost_fsm.grammar_filename_prefix_;
  prt_xrefs_docs_fsm.fq_filename_noext_ += o2_fq_fn_noext.c_str();
  Parser prt_xrefs_docs(prt_xrefs_docs_fsm, &prt_xrefs_docs_can, 0, 0, &Error_queue);
  prt_xrefs_docs.parse();
```

This code is used in section 126.

129. Emit FSC file.
Let the linker know what's happening with the grammar's first set for the linker.

```
⟨ emit fsc file 129 ⟩ ≡
  OP_FSC_FILE(Error_queue);
```

This code is used in section 130.

130. Driver to emit FSA, FSC, and Documents of grammar.

```
⟨ emit FSA, FSC, and Documents of grammar 130 ⟩ ≡    /* ⟨ print grammar tree 133 ⟩; */
  ⟨ output enumeration header file 118 ⟩;
  ⟨ output Errors files 119 ⟩;
  ⟨ output User Terminals files 120 ⟩;
  ⟨ output T-alphabet file 125 ⟩;
  ⟨ output grammar header file 121 ⟩;
  ⟨ output grammar cpp file 122 ⟩;
  ⟨ output grammar sym file 123 ⟩;
  ⟨ output grammar tbl file 124 ⟩;
  ⟨ emit fsc file 129 ⟩;
  ⟨ emit documents 126 ⟩;
```

This code is used in section 14.

131. Print routines.

132. Print tree structure of rules.

⟨ print tree 132 ⟩ ≡

```
yacco2::lrclog << "Tree␣dump␣of␣grammar" << std::endl;
prt_ast_functor prt_functr(&PRINT_RULES_TREE_STRUCTURE);
ast_prefix pre(*GRAMMAR_TREE, &prt_functr);
while (pre.base_stk_.cur_stk_rec() ≠ 0) {
  pre.exec();
}
```

This code is cited in sections 14 and 29.

133. Print grammar tree.

⟨ print grammar tree 133 ⟩ ≡

```
yacco2::lrclog << "Grammar␣Tree␣dump" << std::endl;
prt_ast_functor prt_grammar_functr(&PRINT_GRAMMAR_TREE);
ast_prefix prefix_grammar(*GRAMMAR_TREE, &prt_grammar_functr);
tok_can < AST *> pt_can(prefix_grammar);
int n(−1);
for ( ; pt_can[++n] ≠ NS_yacco2_k_symbols::PTR_LR1_eog__; ) ;
```

This code is cited in section 130.

134. Print dump common states.

```
⟨ print dump common states 134 ⟩ ≡
  LR1_STATES_ITER_type ci = LR1_COMMON_STATES.begin( );
  LR1_STATES_ITER_type cie = LR1_COMMON_STATES.end( );
  yacco2::lrclog ≪ "Common␣States␣dump" ≪ std::endl;
  int cstate_no(0);
  for ( ; ci ≠ cie; ++ci) {
    ++cstate_no;
    STATES_ITER_type si = ci→second.begin( );
    STATES_ITER_type sie = ci→second.end( );
    bool pre = false;
    for ( ; si ≠ sie; ++si) {
      state ∗ cstate = ∗si;
      if (pre ≡ false) {
        pre = true;
        yacco2::lrclog ≪ cstate_no ≪ "::Common␣State:␣" ≪ cstate→vectored_into_by_elem_;
        if (cstate→vectored_into_by_elem_sym_ ≡ 0) {
          yacco2::lrclog ≪ "␣Entry␣Symbol:␣No␣symbol" ≪ std::endl;
        }
        else {
          yacco2::lrclog ≪ "␣Entry␣Symbol:␣" ≪ cstate→entry_symbol_literal( ) ≪ std::endl;
        }
      }
      yacco2::lrclog ≪ "\tstate␣no:␣" ≪ cstate→state_no_ ≪ std::endl;
    }
    yacco2::lrclog ≪ std::endl;
  }
```

This code is cited in section 39.

This code is used in section 40.

135. Print dump state.

```
⟨ print dump state 135 ⟩ ≡
  const char ∗literal_name;
  yacco2::lrclog ≪ std::endl;
  yacco2::lrclog ≪ "State␣dump" ≪ std::endl;
  si = LR1_STATES.begin( );
  sie = LR1_STATES.end( );
  for ( ; si ≠ sie; ++si) {
    state ∗ cur_state = ∗si;
    Print_dump_state(cur_state);
  }
```

This code is cited in section 39.

This code is used in section 40.

136. Some tracing facilities.

137. Increment and printout Recursion counter.

⟨ Increment and printout Recursion counter 137 ⟩ ≡
  ⟨ Increment Recursion counter 138 ⟩;
  ⟨ Printout Recursion counter 139 ⟩;

This code is used in sections 79, 81, and 82.

138. Increment Recursion counter.

⟨ Increment Recursion counter 138 ⟩ ≡
  ++RECURSION_INDEX__;

This code is used in section 137.

139. Printout Recursion counter.

⟨ Printout Recursion counter 139 ⟩ ≡
  for (**int** $y__(1)$; $y__ \le$ RECURSION_INDEX__; $++y__$) *lrclog* ≪ '.';

This code is used in sections 79, 82, 83, 84, and 137.

140. Decrement Recursion counter.

⟨ Decrement Recursion counter 140 ⟩ ≡
  −−RECURSION_INDEX__;

This code is used in sections 79, 81, 82, 84, and 86.

141. Write out *o2_defs.cpp* Structure implementations.

```
⟨ o2_defs.cpp 141 ⟩ ≡
#include "o2.h"
  ⟨ Structure implementations 44 ⟩;
```

142. Bugs — ugh.
What are they good for? Abolutely nothing! Though i paraphrase the song, my entomology teaches and exposes my myopic shortcomings. Ahh... the forest versus the trees of relativist processes attending to correctness.

143. *Microsoft compiler* **Corrupted memory heap.**
Well here's the scoop: IT IS THEIR TEMPLATE HANDLING of a globally defined set: The following code shows the bug:

```
1:  std::set<state*> VISITED_MERGE_STATES_IN_LA_CALC;
2:  int main(int argc,char*argv[]){
3:    VISITED_MERGE_STATES_IN_LA_CALC.clear();}
4:    if(VISITED_MERGE_STATES_IN_LA_CALC.empty() == true){
5:      cout << "===========its empty" << endl;
6:    }
7:    return 0;
8:  }
9:
```

Debug displays the dtor winddown of the tree when the program is exiting that gives its access violation. It is walking the tree when it thinks there is something to delete — bad initialization values for their iterators within their container. There are other heap variants but ... Now the real stupidity: screen shots for MS to deal with and trying to send the "feedback" to MS out of Visual Studio 2005 caused another Visual Studio bug: it experience technical difficulties ... did not register the problem with them — but i had their acknowledgement reference number that i emailed with their assurance that they would within 3 days (whose days business?) acknowledge receipt of the to-be-investigated problem. Well u got it i'm still waiting. From the above code i have no pre-initialization code before it enters the mainline but i'll not display their screen shot as this is wasting my time.

Again it's a sad statement regarding how they deal with problems and lag time before product fixes: just read their approach to voting on bugs before fixing them, all other software companies call it a "Bug Report" or a variant there of but not MS — their euphemism is Customer Feedback ... stop gagging Dave, and why is it 2 years between compiler releases to correct bugs, and have u tried obtaining their Beta compiler release? There is no fix until MS fixes it in possibly their next release. Until then at least it reguritates at the end of the program and my patience for the deliverance day.

Early July 2006.

144. Comment no 2: Apple bug – Problem Id: 4403453 unresolved linkages.
Open source has it's warts whereby i was testing how Apple deals with problems when they go thru a middle man like gnu.org for their C++ compiler and linker. At least they have a "Bug Reporter" and dialogue of receipt and "how's it going" followups. Unfortunately they flunked in their Californian way to getting the problem fixed and in not answering how i could get their lodged reference number when they passed the problem to the gnu.org. My beef is how i can track my problem(s) within the open source community thru the middleman and see how fast they correct the problem? Voting does not count here. This problem is still being sat on by Apple when their last followup was "how's it going?" (only 10 months o/s). It only confirmed my skepticism about this community and my middleman and how they pass on the problem. I have not reached the next step in how gnu.org deals with the fielded problem. Going to gnu.org shows a genuine honesty towards publishing problems. Microsoft is not of the same ilk. If they were GM would people buy their cars particularly when there are the Naders who keep them accountable? What would it cost in litigation towards tarnished goods / services by MS?

Another beef:
Proof of bug demonstrated with logged outputs etc does not seem to be adequate in Apple's engineers' minds? Boy i can't wait for the day when the software manufacturer can under restricted access enter my computer and see / diagnose problems without the crap of whittle down code size, demonstrate the bug in at most 1 line of code platitudes. At least Apple is a bit more earnest in frontline manners... but i'm still waiting on this problem particularly when their engineer flubbed the initial response back to me.

Alas what can i do? I'm currently looking at Sun and their development team to deliver a better comnpiler with faster logistics in dealing with bugs. I'm more optimistic as my people used the "Oregon Pascal" compiler for 16 years. Even though the company folded in the early nineties some of their excellent compiler engineers are working for Sun. So let's see when my Sun AMD desktop running Solaris 10 and 3 years software support with Sun comes in so that i can peddle-to-the-metal my program against this new backdrop. So far all portings have drawn problems be it interpretations of the C++ compiler standard by the compiler writers with their implementations, past failings by C++ compilers to handle properly class itor lists with 1998 work arounds, template tantrums to this day and i'm not meta templating, and threading issues. I guess i'm just stubborn in what i think will be good for the software world: as i just keep on hanging on as the song goes... I'm slowly regressing towards C only cuz C++ has that ++ complexity with ++ bugs. The beguiling part of C++ is templates dealing with common containers that makes the withdrawal process more painful...

9 Jan 2006.

145. *Sun Microsystems compiler* **.** Excellent!.
Well so far the O_2 port is going well. Their performance tools are great. Their compiler works as it should. Their development IDE is close to Microsoft in facilities. The only speed bump is in the "file mount" when it occassionally gets lost in not finding things even when the mounted account has been refreshed. Also their text editor can't handle very large text files like tracings. The odd time "Sunstudio" goes in tail-chasing: must exit out and sign back in but... Overall rating between a 1 - 10 dada 11 that binary for well done even with the droppings as what is probably happening (my musings) in their messaging / interupt facility drops those claques — that french for hits.

23 March 2007.
PS: won't be going back to Microsoft. They are too arrogent without the personalized client support. Sun's problem reporting / case followup is just excellent. With just posted black quarter results and the flop of Vista buy-in i hope this bodes well for Sun who are trying very hard to be Open and frank. Keep it going Sun!

146. Incomplete gening of state's core items due to common prefix.
The state being gened is determined by the elements associated to the closure state being formed. If there is a common prefix making up the state whereby the elements come from different closure states, still all the elements must go into the goto state being gened. As i walked the vector list, the elements that were of different closure state were skipped. This was the filtering that took place when the State was being assessed for closure items generation. The first item found for generation also became the starting point within the list to gen from. Remember the "goto symbol" vector is a list of elements to gen. Solution: any element in the "goto symbol" vector meets the filter criteria, then the complete list must be used: not from the point found within the list. Like any mistake — just dumb! It was not conceptual but an implementation slippage. ⟨ create a new state 85 ⟩ adjusted as the element list's "begin and end" j and je variables are present due to Apple's compiler's indigestion. Again quality assurance on Open Source can be exasperating at the compiler level! Sun's compiler is greeeeat!

July 2007.

147. Missing reduced state deposit on prefix elements leading up to the merged state.
Only went back 1 element to deposit the reduced state: Missing reduce deposits for the balance of elements thru to the beginning of the subrule. Man there are times when u missed the boat Dave. No its links!

July 2007.

148. 3rd party AdobeReader annoyances.
xpdf and *gpdf* programs ain't doing too well on my Sun Solaris Operating system. As i output pdf documents and these programs either core dump or display partial info, this ain't good. So i have split up the documents generated so that one can take what is wanted, and hopefully i'll be able to get a more recent version that works as Adobe seems to be avoiding proprietary Ops — particularly Sun. BTW, the generated documents work properly on my Apple iBook laptop.

Sept. 2007.

149. Missing T(s) in lookahead set due to merges.
The problem was i use a list of closure states of merges that are state contexts for the follow set calculation. Remember, closure states provide the follow set contexts. If 2 subrules of a rule are merged into the same "closure state" context: u now have duplicate states in the "merged into" list. So i test on membership before adding to the list. Here's the melted down rogue grammar where "Ra" has the 2 subrules:

```
 1: /*        FILE:                 1test.lex
 2:         dates:                17 Apr 2001
 3:         Purpose:        see why merge does not work
 4: */
 5: fsm
 6: (fsm-id "test.lex",fsm-filename test,fsm-namespace NS_test
 7: ,fsm-class Cpas_keyword
 8: ,fsm-version "1.0",fsm-date "17 Apr 2001",fsm-debug "true"
 9: ,fsm-comments        "Merge proplempascal Keyword recognizer")
10: parallel-parser
11: (
12:   parallel-thread-function
13:     TH_test
14:   ***
15:   parallel-la-boundary
16:     eolr
17:   ***
18: )
19: @"/pascalxlator/pas_include_files.T"
20: rules{
21: Rtest AD AB(){
22:   -> Rtest_indiv
23: }
24:
25: Rtest_indiv AD AB(){
26:   -> Ra Rc
27:   -> Rr Ra Rm
28:   -> RE Ra Rt
29: }
30:
31: Ra AD AB(){-> "a" -> "A"}  Rc AD AB(){-> "c" } Rr AD AB(){-> "r" }
32: RE AD AB(){-> "e" } Rt AD AB(){-> "t" } Rm AD AB(){-> "m" }
33: }// end of rules
34:
```

Ra's lookahead set is "c,m,t" but it was missing "m" caused by the duplicate closure state. Why u ask does duplicates cause a problem? i tested visited states to guard agaist cycles so stop going thru the balance of the list!

Nov. 2007.

150. Not testing Start State for LRness.
The symtoms: the quality assurance non lr1 grammar caused O_2 to loop when gening its states and more states ... The cause came from the epsilon rule in the state state not having its reduce_state set to self. When gening the *calc_la* for the reduced subrule, it exited due to the sanity check on state address must be the same as the reduce state address. So there was no lookahead set for the reducing and consequently no proper santity checks of outcome. Well why did u not thrown an error? Don't know what my thought process was at time so enough of the conjecture. The *TS_path7.lex* grammar did it. So let's hear it for QA.

Jul. 2008.

151. Run run run run run away.
The symtoms: gening of states keeps going into a right recursion loop that never ends until "death do us part" with memory. The current closure state network being gened did not detect on self one of its states being "not lr1" compatible. Please see / **yacco2** / *qa/knu1_sick.lex* grammar to test this out. This bug came about when i was gening up some of the grammars from Knuth's paper — "On the translation of laguages from left to right" In "Information and Control", volume 8 of 6, pages 607–639, 1965. The first grammar got mistyped where RB was replaced with RA which made it not lr1. Not too swift Dave — ahh the Occam of self reflections. But the detection of it strengthens O_2. O_2 stops gening the lr states, announces the bad news, and reguritates its states configurations to show where the nonlr1ness was found.

Oct. 2008.

152. Ditto to above — random monkeys throwing coconuts.
The symtoms: occassionally not completing the gening of states for a specific closure network. Due to the introduction of *gening_a_state* returning a status, i forgot to return "a default true" when the vector loop was completed. This loop is normally not completed "the normal for loop" way but a "bypass this vector for the gening" as it is associated with another closure state network continues with the next vector in the for loop. So random results returned was caused by whatever was in the memory at the time. So it's not the sky falling only Dave's memory lapses, myopic vission, and not paying attention to the simple code.

Oct. 2008.

153. Random VMS monkeys throwing spurious end-of-lines into streaming text.
The symtoms: in VMS the outputted C++ source files contains spurious carriage returns. Man u should see what it does to the C++ compiler when trying to compile these grammar programs where variable names are sliced-and-diced, struct defintions malformed, and keywords being morphed. When the VMS C++ compiler hits the streaming text's buffer-end it takes the initiative to insert an end-of-line (carrage return). How calvalier!
The work-around: Use lots and lots of "std::endl()" to flush out the text buffer instead of text ending with "*n*". Not efficient but hey do u think HP will fix this when their C++ compiler is probably contracted out? Researching this problem they suggested this work-around as the smell is still lingering and the latest compiler dittos the same tune. Ditto workaround to O_2^{linker}'s output that manages the global thread list and their "first sets".

Dec. 2008.

154. Take 2: Random VMS monkeys throwing spurious end-of-lines.
The problem is how VMS opens a file. If it is transfered using "ftp" onto the Alpha as default text, then the file attributes are variable length with carriage return. This is the fault. By getting "o2" working and deleting all the "cpp" type files, they get gened as "stream-lf" file and not as a file of variable length with carriage return. This was discovered when "o2linker" was taking 60 to 100 x longer to output the "yacco2_fsc.fsc" file from its input: "yacco2.fsc" file. When the old "yacco2_fsc.cpp" file was removed, VMS operating system did not default to the previous version with its attributes but created it as a "stream-lf" type file provided by "ofstream" type. This stopped the heavy seeking on writing out the file as it was probably doing this per character. Now the file is generated in less that a second rather than 60 plus seconds.

This led to looking at the last problem of random "end-of-line" inserts being injected into the source file when on its file buffer boundary: when it flushes the buffer out to the disk. Low and behold, both generating of Tes and Error definitions now work with "stream-lf" file attributes and **no suprious end-of-line** injections. Yipee.

Dec. 2009.

155. Other takes on VMS port: cxxlink.
When linking the last problem was unresolved symbols: *cxxdemangle* on these symbols showed that they were all template based. Required was making sure the various respositories were also present per library: "yacco2", and "o2grammars". Without this, some of the templates from "o2" were undefined due to incomplete instantiation. For the 2 libraries, it is their type definitions used within the templates that are required. Here is *cxxlink*'s **/reportitory** parameter for "o2" link to resolve their references:

```
/repository=([yacco2.compiler.o2.cxx_repository]-
    ,[yacco2.compiler.grammars.cxx_repository]-
    ,[yacco2.library.cxx_repository])-
```

This allows VMS's *cxxlink* to recompile these partially instantiated templates. The 1st repository is read/write while the others are read-only: u must follow the order as "o2" is still being compiled and its repository will be completed by the *cxxlink* reguritation of source templates back to *cxx* compiler.

The order of the libraries inputted are not multipassed by *cxxlink* for unresolved symbols. This occurred with yacco2's symbol table "add_sym_to_stbl" routine that is in the "o2grammars" library. Originally the inputted libraries was "yacco2" followed by "o2grammars". I needed to have the "o2grammars/lib" first in the list of libraries instead of "yacco2/lib".

A final adjustment was "how much stack space" to reserve both per thread and for the process that the threads ran under. Parameter "stack=1024" for the process like "o2" and "o2linker". "pthread" allocation is 10^6 in bytes in the "yacco2_compile_symbols.h" file:

```
VMS_PTHREAD_STACK_SIZE__ 1024*4*256
```

It was a bit of a hit-and-trial-by-fire to get it going. All told now "o2" is ported onto VMS and it works.

Dec. 2009.

156. Evil con numbra those optimizations: lr1 compatibility check.
Background to my stupidities in *is_state_lr1_compatible*:
A thread grammar's la expression has only "eolr" in it. So it is not exploded into all the other Tes if alone. Now the lr1 compatibility check expected that the reducing la had exploded it. So ... Before it was working but with the explosion or is my implosion?

To respect the optimization, 2 checks are done:
If there are 2 reduces taking place and "eolr" is present then not lr1. On the shift / reduce check, if "eolr" is present then ca boom.

Jan. 2010.

157. `|+|` versus "eolr" Evil con numbra duo: lr1 compatibility check.
See previous stupidity. With above eolr vs `|+|` which also means all-the-terminals. So u are shifting on all Tes and trying to reduce on all Tes. I lean to the shift-in-the-wild until the use `|+|` facility is turned off. This is a non lr(1) extension. Leave alone and default to `|+|` over "eolr".

2 lr(1) compatability checks are done:
If there are 2 reduces taking place and "eolr" is present then not lr1. Shift / reduce check: if shift not one of the "lr1 k terminals" and the eolr is present its ca boom.

Feb. 2010.

158. Closure state/vector context infinitely gening states when 1 of its own states not lr(1).
Originally detected improperly which made a legitimate grammar declared as not lr1.
The short of it is to test the states while being gened for lr1 compatibility. The loop was caused when the gening closure state/vector was within itself and not lr1 which kept adding to the states to be gened. *gen_a_state* and *gen_transitive_states_for_closure_part_of_state* now handle the situation properly. The gening closure context is passed to them with these functions returning "continue gening" or "stop" status back to their callers. The bad state is added to the gening states network so that the appropriate state reducing context is reported to the user as not lr1 for fixing. Also the gening of the grammar's start state loop within the main program handles the compatibility issue while gening rather than do a post lr1 evaluation.

Rewrote the "lr states driver" and included tracing to **lrclog** with the lr state decisions made using the closure/vector gening contexts regarding states its gens, merging of states, and non lr1ness discovery be it against proposed state merges, and ugh a non lr1 grammar. This log allows one to review decisions made by O_2 to discover faulty grammars, or faulty gened grammars. Of course its a faster way for me the author to correct implementation foibles. It also helps u when the grammar is wonkie and u want to see where the ambiguous context is for the correcting.

Oct. 2014.

159. Improvement: arbitration-code procedure name added to grammar doc.
Where did i indicate the arbritration code used or not by each ||| state? Nada. So *mpost_output.lex* grammar now emits such indicator when present to the grammar's document within the lr state network, *o2_types.w* and *o2_defs.w* were adjusted to add the *arbitration_name_* string in the state definition. *o2_externs.w* sets the state's *arbitration_name_* when found in the state where one of its threading subrule(s) has arbitration-code. At least now the grammar writer can post assess when arbitration is happening.

Oct. 2014

160. Cweb's writing out of O_2 program.

161. Include files.
To start things off, there are the Standard Template Library (STL) containers, Yac_2o_2's parse library definitions, and the specific grammar definitions to parse the grammar.

⟨ Include files 161 ⟩ ≡
```
#include "globals.h"
#include "yacco2_stbl.h"
  using namespace yacco2_stbl;
#include "o2_lcl_opts.h"
#include "o2_lcl_opt.h"
#include "pass3.h"
#include "o2_types.h"
#include "la_expr_lexical.h"
#include "la_expr.h"
#include "enumerate_T_alphabet.h"
#include "epsilon_rules.h"
#include "mpost_output.h"
#include "prt_xrefs_docs.h"
#include "eval_phrases.h"
#include "enumerate_grammar.h"
#include "rules_use_cnt.h"
```
This code is used in section 163.

162. Accrue O_2 code.

⟨ accrue O_2 code 14 ⟩ +≡ /* accrue code */

163. Create header file for O_2 environment.
Note, the "include search" directories for the c++ compiler has to be supplied to the compiler environment used. This must include Yac_2o_2's library.

⟨ o2.h 163 ⟩ ≡
```
  ⟨ Preprocessor definitions ⟩
#ifndef o2__
#define o2__ 1
  ⟨ Include files 161 ⟩;
  ⟨ External rtns and variables 13 ⟩;
#endif
```

164. O_2 implementation.
Start the code output to *o2.cpp* by appending its include file.

⟨ `o2.cpp` 164 ⟩ ≡
#include `"o2.h"`
  ⟨ accrue O_2 code 14 ⟩;

165. Index.

ABORT_GENING_STATES: 86, 98.
ACCEPT_FILTER: 28, 29, 31, 32, 34, 37, 128.
add_closure_rules_subrules_to_state: 70, 73, 78, 85.
add_element_to_state_vector: 72, 75, 77.
add_follow_set_contributor: 57, 105.
add_follow_set_transition: 58, 106.
add_fs_setA_to_LA: 50, 52, 53, 54.
add_merge_closure_info: 63, 94.
add_rule_s_subrules_to_state: 70, 73, 75, 78.
add_rule_to_closure_list: 74, 75, 77.
add_rule_to_follow_list: 75, 77, 104.
add_state_to_conflict_states_list_if: 82, 84, 89.
add_state_to_gbl_lr1_state_tbls: 67, 82, 88.
add_T_to_follow_set: 60.
ae: 106.
*arbitration_name_*: 159.
*arbitrator_name_*: 67, 68, 69.
are_gened_states_lr1_compatible: 99.
are_states_equivalent: 91, 98.
are_2_states_compatible_yes_merge: 93, 98.
argc: 14, 20.
argv: 14, 20.
assess_state: 92.
associated_rt_bnded_cs: 82, 85.
AST: 28, 29, 31, 32, 34, 37, 39, 46, 47, 52, 57, 60, 67, 69, 70, 73, 75, 76, 78, 92, 100, 101, 105, 106, 113, 127, 128, 133.
ast_prefix: 28, 29, 31, 32, 37, 127, 128, 132, 133.
ast_prefix_wbreadth_only: 34.
*base_stk_*: 132.
begin: 27, 39, 50, 53, 54, 63, 72, 73, 74, 79, 80, 81, 82, 85, 87, 91, 92, 93, 94, 95, 96, 97, 98, 99, 105, 106, 113, 134, 135.
Big_buf: 14.
BIG_BUFFER_32K: 14.
brother: 75, 76, 100, 101, 105, 106.
bug: 10.
BYPASS_FILTER: 127.
c_str: 24, 26, 40, 116, 127, 128.
CAbs_lr1_sym: 27, 36, 47, 52, 69, 75, 76, 92, 100, 101, 105, 113, 116.
calc_la: 52, 92, 150.
Cenumerate_grammar: 28.
Cepsilon_rules: 29.
Ceval_phrases: 34.
CHAR: 14.
ci: 134.
cie: 134.
Cla_expr: 36.
Cla_expr_lexical: 36.
clear: 30, 52.
clog: 31.
close: 24.
closure_only_derives: 64, 70.
*closure_rule_list_*: 64, 73, 74.
CLOSURE_RULES_ITER_type: 73, 74.
closure_rules_making_up_first_set: 74.
CLOSURE_RULES_type: 73, 74.
Closure_state: 73.
*closure_state_*: 46, 52, 57, 58, 59, 75, 76, 94, 97.
Closure_state_associate_with: 75.
*closure_state_birthing_it_*: 67, 68, 69, 85, 93.
*closured_state_gening_it_*: 75, 76.
cmd_line: 26.
Cmpost_output: 127.
*common_la_set_idx_*: 46.
COMMON_LA_SETS: 41.
COMMONIZE_LA_SETS: 13, 41.
compatibility_result: 86.
compatible: 82, 93, 98.
compiler: 143, 145.
content: 47, 52, 75, 76, 92, 100, 101, 105, 113.
continue_gening: 79, 81.
convertMPtoPDF: 127.
cout: 14, 40.
Cpass3: 26.
cpp: 10, 141, 164.
Cprt_xrefs_docs: 128.
cr: 74.
create_follow_sets_of_state: 78, 85, 105.
create_start_state: 67, 78.
crt_core_items_of_state: 76, 85.
crt_start_rule_s_follow_set: 78, 113.
Crules_use_cnt: 37.
cs: 52, 53.
cs_for_merging: 93, 96.
cs_id: 75.
cs_To_merge_into: 93, 96.
*cs_vector_combo_gening_it_*: 46, 67, 68, 69, 75, 76, 83, 84, 85, 96.
cstate: 134.
cstate_no: 134.
cur_state: 135.
cur_stk_rec: 132.
cweave: 127.
cweb: 10, 20, 23, 127.
cweb_k_filter: 127.
CYCLIC_USE_TABLE: 13, 14.
CYCLIC_USE_TBL_type: 13, 14.
deal_with_follow_str_sym: 105, 106.
defs: 10.
*derives_closure_rule_list_*: 74.

determine_reduced_state_type: 78, 82, 87.
DUMP_ERROR_QUEUE: 21, 30.
el: 72.
Elem: 46, 47, 72.
Elem_id: 72.
Elem_iter: 82.
elem_list: 80, 82, 85.
elem_space: 31.
empty: 21, 30, 52, 53, 54, 72, 73, 74, 93, 105, 106.
end: 27, 39, 50, 53, 54, 58, 63, 72, 73, 74, 79, 80, 81, 82, 85, 87, 88, 89, 91, 92, 93, 94, 95, 96, 97, 98, 99, 104, 105, 106, 113, 134, 135.
END_OF_LR1_DEFS: 92.
endl: 14, 16, 25, 27, 31, 37, 39, 40, 79, 81, 82, 83, 84, 118, 119, 120, 121, 122, 123, 124, 125, 127, 128, 132, 133, 134, 135.
Eno: 69.
eno: 58, 59, 79, 80, 81, 82, 85, 88, 98, 105.
Entry_sym: 69.
entry_symbol_literal: 115, 134.
enum_id: 47, 51, 52, 55, 58, 75, 77, 92, 113.
enumerate_filter: 28.
enumerate_grammar: 10, 28.
enumerate_grammar_can: 28.
enumerate_grammar_fsm: 28.
enumerate_grammar_walk: 28.
enumerate_T_alphabet: 10.
enumerated_id__: 47, 52, 75, 76, 92, 100, 101, 105, 116.
Eos: 58.
eos: 47, 52, 106.
eosubrule: 65.
epsilon: 102, 106.
epsilon_fsm: 29.
epsilon_rules: 10, 29.
equivalent: 98.
erase: 73.
Err_pp_la_boundary_attribute_not_fnd: 36.
ERR_SW: 14, 22, 25, 119, 125.
error_logging: 24.
Error_queue: 13, 14, 20, 21, 22, 26, 28, 29, 30, 34, 36, 37, 118, 119, 120, 121, 122, 123, 124, 125, 127, 128, 129.
eval_fsm: 34.
eval_phrases: 34.
evaluate_phase_order: 34.
exec: 132.
exit: 14.
external_file_id__: 27.
false: 52, 76, 82, 91, 92, 93, 98, 100, 102, 134.
fe: 52, 53, 54, 58, 94, 97, 105, 106, 113.
Fe: 50.
filename_id: 127.
fill_la_from_merge: 52, 53, 54.
fill_la_from_transition: 52, 53, 54.
filter: 29.
find: 50, 52, 53, 58, 72, 73, 74, 87, 88, 89, 92, 94, 97, 98, 104, 105, 106, 113.
find_state_element_s_rule_no: 55, 94, 97.
find_sym_in_stbl: 51.
find_2_states_compatible_and_merge: 86, 98.
first: 79, 81, 82, 91, 92.
first_element: 75.
first_element_t: 75.
first_set: 10, 106.
FIRST_SET_ITER_type: 106.
first_set_rules: 9, 32.
follow_element: 50, 52, 53, 54, 57, 58, 60, 62, 63, 94, 97, 105, 113.
follow_rule_list_: 64, 104, 105.
FOLLOW_RULES_ITER_type: 104, 105.
follow_set_: 50, 60, 106, 113.
FOLLOW_SETS_ITER_type: 50, 113.
follow_str_t: 105, 106.
For_closure_state: 79, 81, 82, 84.
for_closure_state_: 39, 44, 75, 76, 79, 81, 82, 83, 84, 85, 89, 96.
For_gening_context: 79, 82, 83, 84, 85.
fq_filename_noext_: 127, 128.
from_se: 76.
fs_filter: 32.
fs_rule_walk: 32.
fs_rules_can: 32, 33.
fsc: 10.
fsi: 58, 105.
fsm_ph: 127.
gen_a_state: 79, 81, 82, 86, 158.
GEN_CALLED_THREADS_FS_OF_RULE: 33.
gen_context: 39, 44, 70, 73, 75, 76, 78, 79, 81, 82, 89.
Gen_context: 81.
GEN_FS_OF_RULE: 32.
gen_ok: 82, 84.
gen_transitive_states_balance_for_closure_vector: 81, 82, 84.
gen_transitive_states_for_closure_context: 39, 79.
gen_transitive_states_for_closure_part_of_state: 158.
gen_vector_: 39, 44, 75, 79, 81, 82, 83, 84, 85.
gening_a_state: 152.
Gening_context: 76, 89.
gening_context: 39, 70, 78.
gening_state: 39, 40.
GET_CMD_LINE: 20.
get_1st_son: 39, 75.

Goto_state: 81.
*goto_state_*: 46, 76, 84, 95.
gpdf: 148.
*grammar_filename_prefix_*: 127, 128.
GRAMMAR_TREE: 34, 37, 127, 132, 133.
id: 47, 52, 75, 76, 77, 100, 101, 102, 105, 106, 116.
id__: 27.
identifier: 127.
ie: 27, 50, 53, 54, 63, 79, 81, 82, 91, 99, 105, 113.
ieno: 91.
ifstream: 26.
ii: 53.
includes: 10.
insert: 28, 29, 31, 32, 34, 37, 51, 53, 60, 72, 73, 74, 94, 97, 104, 106, 113, 127, 128.
intro: 10.
is_state_lr1_compatible: 82, 92, 93, 99, 156.
is_str_epsilonable: 100, 101.
is_str_rt_bnded: 76, 100.
Iter_begin: 76.
Iter_end: 76.
*its_enum_id_*: 46, 47.
its_rule_def: 47, 52, 55, 58, 75, 77, 102, 105, 106.
its_state: 57.
*its_state_*: 57, 59.
its_t_def: 47, 75, 77.
je: 72, 74, 80, 85, 91, 92, 98.
jeno: 91.
JUNK_tokens: 14, 26, 36.
just_walk_functr: 29, 31, 32.
ke: 92, 93, 96, 105.
knu1_sick: 151.
la: 113.
la_bndry: 36, 113.
la_expr: 10, 92.
la_expr_fsm: 36.
la_expr_lex_fsm: 36.
la_expr_lex_parse: 36.
la_expr_parse: 36.
la_expr_source: 10.
la_expr_tok_can: 36.
la_express: 49.
la_first_set: 113.
*la_set_*: 45, 46, 47, 48, 52, 53, 54, 92.
LA_SET_ITER_type: 50, 92, 113.
LA_SET_type: 47, 50.
la_srce_tok_can: 36.
la_supplier: 36.
La_to_fill_in: 50, 51, 52, 53, 54.
la_tok_can_lex: 36.
le: 91, 92.
lex: 9, 10, 13, 26, 70, 150, 151, 159.
line_no__: 27.
literal_name: 135.
LOAD_YACCO2_KEYWORDS_INTO_STBL: 18.
loop_until_empty: 73.
lrclog: 14, 16, 24, 25, 27, 31, 37, 39, 40, 79, 81, 82, 83, 84, 118, 119, 120, 121, 122, 123, 124, 125, 127, 128, 132, 133, 134, 135, 139.
lrerrors: 24.
LR1_COMMON_STATES: 10, 13, 14, 88, 98, 134.
LR1_COMPATIBLE: 14, 79, 81, 82, 92, 98, 99.
LR1_EOG: 92.
LR1_EOLR: 92, 113.
LR1_HEALTH: 14, 38, 39, 40.
LR1_PARALLEL_OPERATOR: 51.
LR1_REDUCE_OPERATOR_LITERAL: 51.
LR1_STATES: 10, 14, 38, 39, 88, 135.
LR1_STATES_ITER_type: 88, 98, 134.
LR1_STATES_type: 13, 14.
lr1_test: 93.
Lr1_VERSION: 14.
ls: 91.
main: 14.
Max_no_subrules: 31.
me: 91.
Merge: 53.
Merge_into_state: 91.
merged: 93.
MERGED: 86, 93, 98.
*merges_*: 52, 53, 54, 62, 63.
MERGES_ITER_type: 53, 63.
MERGES_type: 53.
Microsoft: 143.
Microsystems: 145.
mpost: 127.
mpost_fsm: 127, 128.
mpost_just_walk_functr: 127.
mpost_output: 159.
mpost_rule_walk: 127.
mpost_rules: 127.
mpost_rules_can: 127.
mre: 94.
mrri: 94.
ms: 91.
msfmi: 94.
n: 133.
*next_state_element_*: 46, 76, 95.
no: 105.
NO_LR1_STATES: 13, 14, 88.
no_reduce_types: 87.
no_reduces: 87, 92.
normal_tracing: 24, 40.

NOT_LR1_COMPATIBLE: 39, 40, 79, 81, 82, 86, 92, 93, 99.
NOT_MERGED: 86, 93, 98.
npos: 23.
NS_enumerate_grammar: 28.
NS_epsilon_rules: 29.
NS_eval_phrases: 34.
NS_la_expr: 36.
NS_la_expr_lexical: 36.
NS_mpost_output: 127.
NS_pass3: 26.
NS_prt_xrefs_docs: 128.
NS_rules_use_cnt: 37.
NS_yacco2_k_symbols: 127, 133.
NS_yacco2_T_enum: 87.
OP_ENUMERATION_HEADER: 118.
OP_ERRORS_CPP: 119.
OP_ERRORS_HEADER: 119.
OP_FSC_FILE: 129.
OP_GRAMMAR_CPP: 122.
OP_GRAMMAR_HEADER: 121.
OP_GRAMMAR_SYM: 123.
OP_GRAMMAR_TBL: 124.
OP_T_Alphabet: 125.
OP_USER_T_CPP: 120.
OP_USER_T_HEADER: 120.
open: 24.
orderly_walk: 34.
o2: 10, 164.
o2__: 163.
o2_defs: 10, 141, 159.
o2_externs: 10, 13, 20, 159.
o2_file_to_compile: 14, 22, 23, 26.
o2_fq_fn_noext: 14, 23, 24, 127, 128.
O2_FSM_PHASE: 127.
o2_lcl_opt: 24.
o2_lcl_opts: 24.
O2_PHRASE_TBL: 34.
O2_PP_PHASE: 35, 36, 113.
O2_RULES_PHASE: 29.
O2_T_ENUM_PHASE: 31.
o2_types: 159.
O2_xxx: 34.
o2externs: 10.
Parallel_threads_shutdown: 16.
parse: 26, 28, 29, 34, 36, 37, 127, 128.
Parser: 26, 28, 29, 34, 36, 37, 127, 128.
pass3: 10, 13, 16, 26.
pdftex: 127.
phase_order_filter: 34.
phrase_tree: 29.
phrases_can: 34.
pop_front: 62.
pos_in_line__: 27.
Possible_gen_context: 73, 75.
pp: 23.
pp_ph: 36, 113.
pre: 132, 134.
prefix_grammar: 133.
prev_re: 95.
*previous_state_*: 46, 76.
*previous_state_element_*: 46, 76, 80, 95.
Print_dump_state: 13, 78, 82, 135.
PRINT_GRAMMAR_TREE: 133.
PRINT_RULES_TREE_STRUCTURE: 132.
processed_rules_set: 73.
prog: 10.
prt_ast_functor: 132, 133.
prt_fs_of_rules: 10.
prt_functr: 132.
prt_grammar_functr: 133.
PRT_SW: 14, 22, 25, 126.
prt_xrefs_docs: 128.
prt_xrefs_docs_can: 128.
prt_xrefs_docs_filter: 128.
prt_xrefs_docs_fsm: 128.
prt_xrefs_docs_walk: 128.
pt_can: 133.
PTR_LR1_eog__: 27, 31, 32, 133.
push_back: 36, 57, 58, 72, 88, 89, 96.
push_front: 63.
p3_fsm: 26.
P3_tokens: 14, 26, 27.
qa: 151.
r_def: 73, 74.
r_id: 52, 75, 77.
rd: 32, 47, 52, 58, 73, 74, 75, 77, 102, 106, 113, 116.
re: 94, 95, 97.
Recursion_count: 14.
RECURSION_INDEX__: 13, 14, 138, 139, 140.
reduce_fnd: 89.
reduced_state: 80.
*reduced_state_*: 46, 52, 75, 80, 95.
Refered_rule: 104.
Refered_T: 60.
remove_merge_closure_info: 62, 97.
report_card: 14, 51.
Requesting_state: 82.
rfind: 23.
ri: 95.
rie: 95.
ris: 74, 75, 77.
rni: 94, 97.
rno: 113.

rr: 47, 75, 77, 102, 105, 106.
rri: 94, 95, 97.
rrie: 94, 95, 97.
rstr_t: 100.
rsym: 100.
rt: 47, 75, 77, 92, 106.
rt_bnded: 76, 85.
rule: 105.
rule_def: 32, 33, 47, 52, 58, 73, 74, 75, 77, 102, 106.
Rule_def_t: 57.
rule_def_t: 57.
*rule_def_t_*: 57, 105, 113.
RULE_ENO: 52, 53, 55, 57, 58, 75, 77, 94, 97, 104, 105, 113.
Rule_in_stbl: 74, 75, 77.
rule_in_stbl: 74, 75, 77.
rule_name: 116.
Rule_no: 53, 57.
*rule_no_*: 53, 54, 57, 59, 105, 113.
RULE_NOS_SET_ITER_type: 94, 97.
RULE_NOS_SET_type: 94, 97.
rule_s_tree: 73, 105, 113.
Rule_tree: 70.
rule_walk: 29.
ruleno: 94, 97.
rules_can: 29.
rules_ph: 29.
rules_to_add: 94, 97.
rules_tree: 28, 29, 31, 32, 39, 128.
rules_use_can: 37.
rules_use_cnt: 37, 70.
rules_use_cnt_filter: 37.
rules_use_cnt_fsm: 37.
rules_use_walk: 37.
rules_use_walk_functr: 37.
S_CONFLICT_STATES_ITER_type: 93, 96, 99.
S_FOLLOW_SETS_ITER_type: 52, 53, 58, 94, 97, 105.
S_VECTOR_ELEMS_ITER_type: 72, 76, 80, 82, 85, 91, 92, 94, 95, 97, 105.
S_VECTOR_ELEMS_type: 72, 80, 82.
S_VECTORS_ITER_type: 72, 79, 81, 82, 87, 89, 91, 92, 94, 95, 97, 105.
S_VECTORS_type: 56, 65.
se: 72, 75, 76, 77, 80, 82, 83, 84, 85, 92, 95, 105, 106.
se_rt_bnded_condition: 76.
second: 52, 53, 58, 72, 80, 82, 87, 88, 89, 91, 92, 94, 95, 97, 98, 105, 134.
*self_state_*: 46, 52, 75, 76, 80, 94, 95, 97.
set: 127.
set_rc: 36.
sfmi: 94, 97.
sfmie: 94, 97.
si: 39, 95, 134, 135.
sie: 39, 134, 135.
size: 31, 87, 89, 91.
size_type: 23.
sr_can: 31.
*sr_def_element_*: 45, 46, 47, 85.
SR_element: 57.
*sr_element_*: 46, 52, 72, 76, 91, 92, 105.
*sr_elements_*: 57.
sr_filter: 31.
sr_walk: 31.
srd: 75.
ssi: 95.
START_OF_RULES_ENUM: 10, 92.
Start_rule: 113.
start_rule_def: 33.
Start_Rule_def_t: 75.
start_rule_def_t: 39.
Start_rule_t: 67, 78.
START_STATE_ENUMERATE: 64, 67, 68, 75, 76.
State: 13, 57, 79, 88, 89.
state: 13, 39, 44, 52, 53, 57, 63, 67, 68, 69, 70, 72, 73, 74, 75, 76, 78, 79, 80, 81, 82, 85, 87, 88, 89, 91, 92, 93, 96, 98, 99, 100, 101, 104, 105, 113, 115, 134, 135.
State_elem: 57, 58, 59.
state_element: 44, 45, 46, 48, 50, 52, 53, 54, 55, 56, 57, 58, 72, 75, 76, 80, 82, 91, 92, 94, 95, 97, 105.
State_for_merging: 93, 94, 95, 98.
*state_no_*: 39, 40, 59, 67, 68, 69, 79, 81, 82, 83, 84, 88, 134.
*state_s_conflict_state_list_*: 89, 93, 96, 99.
*state_s_follow_set_map_*: 52, 53, 58, 94, 97, 105, 113.
state_s_to_vector: 64.
*state_s_vector_*: 72, 79, 81, 87, 89, 91, 92, 94, 95, 97, 105.
State_to_eval: 92.
*state_type_*: 67, 68, 69, 78, 82.
STATES_ITER_type: 39, 98, 134.
STATES_SET_ITER_type: 53.
STATES_SET_type: 13, 14.
STATES_type: 14, 88.
STBL_T_ITEMS: 14, 113.
STBL_T_ITEMS_type: 14.
std: 14, 16, 23, 24, 25, 26, 31, 40, 128, 132, 133, 134, 135.
Str: 100, 101.
str_t: 101.
string: 14, 23, 24.

subrule_def_: 46, 55, 75, 76.
subrules_t: 75.
substr: 23.
Sun: 145.
svi: 87.
svie: 87.
sym: 27, 36, 47, 52, 92, 100, 101, 105, 113, 116.
t_cnt: 92.
T_COUNT_type: 92.
t_def: 51, 92.
T_ENO: 52, 75, 77, 92.
T_Enum: 28, 29, 31, 32, 34, 37, 47, 52, 75, 76, 77, 80, 87, 89, 92, 100, 102, 106, 116, 127, 128.
T_eosubrule: 58.
T_fsm_phrase: 127.
t_id: 75, 77.
t_in_stbl: 60, 92, 106.
T_in_stbl: 50, 92, 106, 113.
t_name: 116.
T_not_meta: 92.
T_parallel_la_boundary: 36, 113.
T_parallel_parser_phrase: 36, 113.
T_refered_rule_: 28, 37, 47, 75, 77, 92, 100, 102, 106, 128.
T_refered_T_: 28, 47, 75, 77, 92, 102, 106, 128.
T_rule_def_: 28, 29, 32, 37, 116, 128.
T_rules_phrase: 29.
T_SW: 14, 22, 25, 120, 125.
t_sym: 50, 51.
T_sym_tbl_report_card: 14, 51.
T_T_called_thread_eosubrule_: 28, 47, 52, 75, 76, 77, 87, 92, 106, 116, 128.
T_T_cweb_comment_: 127.
T_T_enum_phrase_: 34.
T_T_eosubrule_: 28, 47, 52, 75, 77, 80, 87, 89, 92, 100, 102, 106, 116, 128.
T_T_error_symbols_phrase_: 34.
T_T_fsm_phrase_: 34.
T_T_lr1_k_phrase_: 34.
T_T_null_call_thread_eosubrule_: 28, 47, 52, 75, 76, 77, 87, 92, 106, 116, 128.
T_T_rc_phrase_: 34.
T_T_rules_phrase_: 34.
T_T_subrule_def_: 28, 29, 31, 37, 128.
T_T_terminal_def_: 116.
T_T_terminals_phrase_: 34.
T_terminal_def: 47, 75, 77.
td: 47, 51, 75, 77, 116.
teno: 92.
tid: 92.
tintbl: 92.
tm: 63.
to_element: 76.
to_element_t: 76.
To_merge_closure_state: 63.
To_merge_into_state: 93, 94, 95, 97.
To_merge_state: 91.
tok_can: 26, 28, 29, 31, 32, 34, 37, 127, 128, 133.
tok_can_ast_functor: 28, 29, 34, 37, 127, 128.
tok_co_ords__: 27.
TOKEN_GAGGLE: 14, 36.
TOKEN_GAGGLE_ITER: 27.
total_no_subrules: 31.
Transition: 54.
transitions_: 52, 53, 54, 58.
TRANSITIONS_ITER_type: 54.
TRANSITIONS_type: 54.
true: 21, 30, 52, 53, 54, 72, 73, 74, 76, 83, 85, 86, 91, 92, 93, 100, 101, 102, 105, 106, 134.
TS_path7: 150.
tvi: 87.
unwind: 93.
unwind_merge: 93.
Ve: 44.
vectored_into_by_elem_: 64, 67, 68, 69, 88, 91, 98, 134.
vectored_into_by_elem_sym_: 64, 67, 68, 69, 115, 116, 134.
Vectored_into_id_t: 69.
VISITED_MERGE_STATES_IN_LA_CALC: 13, 14, 52, 53.
Voc_ENO: 44, 47, 69, 72, 75, 76, 79, 81, 82, 88, 91, 92, 98, 100, 101, 105, 116.
walk_the_plank_mate: 28.
x: 92.
xpdf: 148.
xrefs_docs_walk_functr: 128.
xx: 31, 32.
xxx: 10.
xxxsym: 10.
xxxtbl: 10.
y__: 139.
yacco2: 10, 12, 14, 16, 24, 26, 27, 31, 32, 36, 40, 127, 128, 132, 133, 134, 135, 151.
YACCO2_AR__: 12.
yacco2_characters: 10.
YACCO2_define_trace_variables: 12, 14.
yacco2_err_symbols: 10.
Yacco2_holding_file: 20.
yacco2_k_symbols: 10.
YACCO2_MSG__: 12, 26.
YACCO2_MU_GRAMMAR__: 12.
YACCO2_MU_TH_TBL__: 12.
YACCO2_MU_TRACING__: 12.

YACCO2_PARSE_CMD_LINE: 22.
yacco2_stbl: 51, 161.
YACCO2_T__: 12.
yacco2_T_enumeration: 10.
yacco2_terminals: 10.
YACCO2_TH__: 12, 26.
YACCO2_THP__: 12.
YACCO2_TLEX__: 12.
yyy: 27.

⟨ Decrement Recursion counter 140 ⟩ Used in sections 79, 81, 82, 84, and 86.
⟨ External rtns and variables 13 ⟩ Used in section 163.
⟨ Include files 161 ⟩ Used in section 163.
⟨ Increment Recursion counter 138 ⟩ Used in section 137.
⟨ Increment and printout Recursion counter 137 ⟩ Used in sections 79, 81, and 82.
⟨ Print pathological symptoms but continue 30 ⟩ Used in section 29.
⟨ Printout Recursion counter 139 ⟩ Used in sections 79, 82, 83, 84, and 137.
⟨ Structure implementations 44, 46, 48, 50, 52, 53, 54, 55, 57, 58, 60, 62, 63, 66, 67, 68, 69, 70, 72, 73, 74, 75, 76, 78, 79, 81, 82, 87, 88, 89, 91, 92, 93, 98, 99, 100, 101, 104, 105, 113, 115 ⟩ Used in section 141.
⟨ accrue O_2 code 14, 162 ⟩ Used in section 164.
⟨ add conflict states to to merge network 96 ⟩ Used in section 95.
⟨ add potential follow set context per production 94 ⟩ Used in section 93.
⟨ add subrule's element to the being gened state's vector 77 ⟩ Used in section 76.
⟨ are all phases parsed? 34 ⟩ Used in section 14.
⟨ calculate Start rule called threads first sets 33 ⟩ Used in section 14.
⟨ calculate rules first sets 32 ⟩ Used in section 14.
⟨ can new state be merged into state network? yes erase its existance and exit 86 ⟩ Used in section 82.
⟨ common prefix gened goto state? yes deal with its goto state 84 ⟩ Used in section 82.
⟨ commonize la sets 41 ⟩ Used in section 39.
⟨ create a new state 85 ⟩ Cited in section 146. Used in section 82.
⟨ deal with current follow string element 106 ⟩ Used in section 105.
⟨ determine each rule use count 37 ⟩ Used in section 14.
⟨ determine entry symbol 47 ⟩ Used in section 46.
⟨ determine if la expression present. Yes parse it 35 ⟩ Used in section 14.
⟨ display to user options selected 25 ⟩ Used in section 22.
⟨ dump aid: enumerate grammar's components 28 ⟩ Used in section 14.
⟨ dump lexical and syntactic's outputted tokens 27 ⟩ Used in section 26.
⟨ emit Cweb and Mpost files 127 ⟩ Used in section 126.
⟨ emit FSA, FSC, and Documents of grammar 130 ⟩ Used in section 14.
⟨ emit documents 126 ⟩ Used in section 130.
⟨ emit fsc file 129 ⟩ Used in section 130.
⟨ epsilon and pathological assessment of Rules 29 ⟩ Used in section 14.
⟨ extract fq name without extension 23 ⟩ Used in section 22.
⟨ fetch command line info and parse the 3 languages 19 ⟩ Used in section 14.
⟨ generate grammar's LR1 states 39 ⟩ Used in section 14.
⟨ get cast referenced T 109 ⟩ Used in sections 47, 92, and 106.
⟨ get cast referenced called thread eosubrule 112 ⟩ Used in section 47.
⟨ get cast referenced eosubrule 110 ⟩ Used in sections 47, 52, and 106.
⟨ get cast referenced null called thread eosubrule 111 ⟩ Used in section 47.
⟨ get cast referenced rule 108 ⟩ Used in sections 47, 102, and 106.
⟨ get command line, parse it, and place contents into a holding file 20 ⟩ Used in section 19.
⟨ get refered-t 61 ⟩ Used in section 60.
⟨ get subrule's referenced rule in follow string 107 ⟩ Used in section 105.
⟨ get total number of subrules for *elem_space* size check 31 ⟩ Used in section 14.
⟨ if error queue not empty then deal with posted errors 21 ⟩ Used in sections 20, 22, 26, 34, 36, 118, 119, 120, 121, 122, 123, 124, and 125.
⟨ is state's element associated with gened closure state? no bypass 83 ⟩ Used in section 82.
⟨ is str's element epsilon 102 ⟩ Used in section 101.
⟨ is the grammar unhealthy? yes report the details and exit 40 ⟩ Used in sections 14 and 39.
⟨ is there back to back thread calls? 51 ⟩ Used in section 50.
⟨ left recursion on rule check — out damn spot 59 ⟩ Used in section 58.
⟨ load O_2's keywords into symbol table 18 ⟩ Used in section 17.

⟨ o2.cpp 164 ⟩
⟨ o2.h 163 ⟩
⟨ o2_defs.cpp 141 ⟩
⟨ output Errors files 119 ⟩ Used in section 130.
⟨ output T-alphabet file 125 ⟩ Used in section 130.
⟨ output User Terminals files 120 ⟩ Used in section 130.
⟨ output enumeration header file 118 ⟩ Used in section 130.
⟨ output grammar cpp file 122 ⟩ Used in section 130.
⟨ output grammar header file 121 ⟩ Used in section 130.
⟨ output grammar sym file 123 ⟩ Used in section 130.
⟨ output grammar tbl file 124 ⟩ Used in section 130.
⟨ parse command line data placed in holding file 22 ⟩ Used in section 19.
⟨ parse la expression and calculate its first set 36 ⟩ Used in section 35.
⟨ parse the grammar 26 ⟩ Used in section 19.
⟨ print dump common states 134 ⟩ Cited in section 39. Used in section 40.
⟨ print dump state 135 ⟩ Cited in section 39. Used in section 40.
⟨ print grammar tree 133 ⟩ Cited in section 130.
⟨ print out xref docs 128 ⟩ Used in section 126.
⟨ print tree 132 ⟩ Cited in sections 14 and 29.
⟨ relink spawning state of merged state 95 ⟩ Used in section 93.
⟨ return entry symbol literal 116 ⟩ Used in section 115.
⟨ set up logging files 24 ⟩ Used in section 22.
⟨ setup O_2 for parsing 17 ⟩ Used in section 14.
⟨ shutdown 16 ⟩ Cited in section 14.
⟨ unchain my reduce states if end-of-subrule and continue to next item 80 ⟩ Used in sections 79 and 81.
⟨ unwind potential merge 97 ⟩ Used in section 93.

O2

	Section	Page
License	1	1
Summary of O_2 — Yacco$_2$'s nickname	2	2
Component overview running O_2	3	2
Tracing facilities	4	3
Grammar anatomy	5	3
Terminal vocabulary	6	3
Overview of generating the grammar's pdf and postscript(ps) documents	7	4
A sample O_2 script where the options are described	8	5
Some definitions	9	6
Catalogue of O_2's files	10	8
O_2's language	11	10
C macros	12	11
External routines and globals	13	12
Main line of O_2	14	13
Some Programming sections	15	14
Shutdown	16	14
Setup O_2 for parsing	17	14
Load O_2's keywords into symbol table	18	14
Fetch command line info and parse the 3 languages	19	14
Get command line, parse it, and place contents into a holding file	20	14
Do we have errors?	21	15
Parse command line data placed in holding file	22	15
Extract fully qualified file name to compile without its extension	23	15
Set up O_2's logging files local to the parsed grammar	24	15
Display to user options selected	25	16
Parse the grammar	26	16
Dump lexical and syntactic's outputted tokens	27	16
Dump aid — Enumerate grammar's components	28	17
Epsilon and Pathological assessment of Rules	29	18
Get the total number of subrules	31	19
Calculate each rule's first set	32	19
Calculate Start rule's called threads first set list	33	19
Are all Grammar phases parsed?	34	20
Thread's end-of-token stream: Lookahead expression post evaluation	35	21
Parse the la expression and calculate its first set	36	22
Determine rule use count: Optimization	37	22
Generate grammar's LR1 states	38	23
Driver generating lr1 states	39	23
Is the grammar unhealthy? yes report the details and exit	40	24
Commonize LA Sets — Combine common sets as a space saver	41	24

Overview of O_2's state generated components 42 25
LR1 definitions 43 26
gen_context definition/implementation 44 29
state_element definition/implementation 45 29
state_element implementation 46 29
∼*state_element* 48 31
Lookahead Comments 49 31
add_fs_setA_to_LA 50 31
calc_la — fill the reduced element's la set by walking follow set graph 52 32
fill_la_from_merge 53 33
fill_la_from_transition 54 33
find_state_element_s_rule_no 55 33
Follow set definition for a rule 56 34
Follow set implementation 57 34
add_follow_set_transition 58 34
remove_merge_closure_info 62 35
add_merge_closure_info 63 35
State definition/implementation 64 36
State's map of "to vector" elements 65 36
State implementation 66 37
state(`AST` ∗ *Start_rule_t*) 67 37
state( ): for closure only state of derives 68 37
state(`AST` & *Vectored_into_id_t*) — Create transitive state 69 37
closure_only_derives — Create a closure only derives state 70 37
Generate states 71 38
add_element_to_state_vector 72 38
add_closure_rules_subrules_to_state 73 39
add_rule_to_closure_list 74 39
add_rule_s_subrules_to_state 75 40
crt_core_items_of_state 76 42
Add subrule's element to the being gened state's vector 77 44
create_start_state — Create start state 78 45
gen_transitive_states_for_closure_context 79 45
Unchain my reduced states if end-of-subrule and continue to next item 80 46
gen_transitive_states_balance_for_closure_vector 81 46
gen_a_state 82 47
Is element vector associated with the current closure state being gened? 83 48
Is element gened from common prefix of an earlier closure state gen? 84 48
Create a new state 85 49
Can new state be merged into state network? Yes then exit 86 49
determine_reduced_state_type 87 50
add_state_to_gbl_lr1_state_tbls 88 50
add_state_to_conflict_states_list_if 89 51
General routines on state compatibilities 90 52
Determining if 2 states are equivalent? 91 53
is_state_lr1_compatible 92 55
are_2_states_compatible_yes_merge 93 57
Add potential follow set context per production 94 58
Relink spawning state of merged state 95 59
Add conflict states to merge network 96 59
Unwind potential merge 97 60
find_2_states_compatible_and_merge 98 60

are_gened_states_lr1_compatible . . . 99 61
is_str_rt_bnded . . . 100 61
is_str_epsilonable . . . 101 62
Is str's element epsilon? . . . 102 62
Follow set routines . . . 103 63
add_rule_to_follow_list . . . 104 63
create_follow_sets_of_state . . . 105 63
crt_start_rule_s_follow_set . . . 113 66
For tracing facilities . . . 114 67
entry_symbol_literal . . . 115 67
Return entry symbol literal . . . 116 67

Emit FSA, FSC, and Documents of grammar . . . 117 68
Output enumeration header file . . . 118 68
Output Errors file . . . 119 68
Output User Terminals files . . . 120 68
Output grammar header file . . . 121 68
Output grammar cpp file . . . 122 68
Output grammar sym file . . . 123 69
Output grammar tbl file . . . 124 69
Output T-alphabet file . . . 125 69
Documents — Grammar's Cweb and Mpost diagrams, and Cross references . . . 126 69
Emit Cweb and Mpost diagrams . . . 127 70
Print out xref docs . . . 128 71
Emit FSC file . . . 129 71
Driver to emit FSA, FSC, and Documents of grammar . . . 130 71

Print routines . . . 131 72
Print tree structure of rules . . . 132 72
Print grammar tree . . . 133 72
Print dump common states . . . 134 73
Print dump state . . . 135 73
Some tracing facilities . . . 136 73
Increment and printout Recursion counter . . . 137 74
Increment Recursion counter . . . 138 74
Printout Recursion counter . . . 139 74
Decrement Recursion counter . . . 140 74

Write out *o2_defs.cpp* **Structure implementations** . . . 141 75

Bugs — ugh . . . 142 76
Microsoft compiler Corrupted memory heap . . . 143 76
Comment no 2: Apple bug – Problem Id: 4403453 unresolved linkages . . . 144 77
Sun Microsystems compiler . . . 145 77
Incomplete gening of state's core items due to common prefix . . . 146 78
Missing reduced state deposit on prefix elements leading up to the merged state . . . 147 78
3rd party AdobeReader annoyances . . . 148 78
Missing T(s) in lookahead set due to merges . . . 149 79
Not testing Start State for LRness . . . 150 80
Run run run run run away . . . 151 80
Ditto to above — random monkeys throwing coconuts . . . 152 80
Random VMS monkeys throwing spurious end-of-lines into streaming text . . . 153 80
Take 2: Random VMS monkeys throwing spurious end-of-lines . . . 154 81
Other takes on VMS port: cxxlink . . . 155 81
Evil con numbra those optimizations: lr1 compatibility check . . . 156 82

|+| versus "eolr" Evil con numbra duo: lr1 compatibility check . . . 157 82
Closure state/vector context infinitely gening states when 1 of its own states not lr(1) 158 82
Improvement: arbitration-code procedure name added to grammar doc . . . 159 83
Cweb's writing out of O_2 program . . . 160 84
Include files . . . 161 84
Create header file for O_2 environment . . . 163 84
O_2 implementation . . . 164 85
Index . . . 165 86