1. License.

2. Summary of O_2 External parse routines.
These are the various procedures that parse Yac$_2$o$_2$'s grammar language and emit the grammar's c++ code and tex document with mpost generated diagrams. Each language construct has its appropriate external procedure that houses the monolithic grammar to start the parse. There is no namespace used to contain these routines as I felt that this was overkill. As this is a closed system, their grammars are not universal and cannot be re-cycled for others. Their only outside value are in teaching examples on "how to skin a cat" or is it "how to parse a lion?"...

External routines ratatouille:

o2externs.w - cweb generator file
o2_externs.h - header file
o2_externs.cpp - implementation

Dependency files from other Yacco2 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 meta terminal defs from Yacco2's grammar alphabet
yacco2_terminals.h - regular terminal definitions from Yacco2's grammar alphabet
$*$. *h* - assorted grammar definitions for Yacco2's parsing
yacco2_stbl.h - symbol table defnitions

External procedures and other globals:

`YACCO2_PARSE_CMD_LINE`
`PROCESS_INCLUDE_FILE`
`PROCESS_KEYWORD_FOR_SYNTAX_CODE`
O2_xxx phases — terminals per distinct parse phrase
`PRINT_RULES_TREE_STRUCTURE`
`WRT_CWEB_MARKER`
`GEN_FS_OF_RULE`
`PRT_RULE_S_FIRST_SET`
`OP_GRAMMAR_HEADER`
`OP_ERRORS_HEADER`
`OP_USER_T_HEADER`
`DATE_AND_TIME`
MAX_USE_CNT_RxSkeletion — rule recyling
XLATE_SYMBOLS_FOR_cweave

Internal procedures:

process_fsm_phrase
process_parallel_parser_phrase
process_T_enum_phrase
process_error_symbols_phrase
process_rc_phrase
process_lr1_k_phrase
process_terminals_phrase
process_rules_phrase
Print_dump_state

3. Global definitions, External parse routines for Yacco2.
Why `CWEB_MARKER` external? It holds the *T_cweb_marker* tree containing *cweb* comments. Why? It is a holding pointer that allows *cweb* comments to be processed before the appropriate parse phrases like *process_xxx_phrase* that are triggered by `PROCESS_KEYWORD_FOR_SYNTAX_CODE`.

4. Create header file.

```
⟨ o2_externs.h 4 ⟩ ≡
#ifndef o2_externs_
#define o2_externs_ 1
  ⟨ Preprocessor definitions ⟩
  ⟨ Files for header 5 ⟩;
#endif
```

5. Files for header.

⟨ Files for header 5 ⟩ ≡

```
#include "globals.h"
#include "o2_types.h"
#include "o2_lcl_opts.h"
#include "o2_lcl_opt.h"
#include "pass3.h"
#include "o2_err_hdlr.h"
#include "fsm_phrase.h"
#include "parallel_parser_phrase.h"
#include "T_enum_phrase.h"
#include "err_symbols_ph.h"
#include "rc_phrase.h"
#include "lr1_k_phrase.h"
#include "terminals_phrase.h"
#include "rules_phrase.h"
#include "yacco2_stbl.h"
#include "enumerate_T_alphabet.h"
#include "mpost_output.h"
#include "prt_xrefs_docs.h"
#include "cweb_put_k_into_ph.h"
  extern CYCLIC_USE_TBL_type CYCLIC_USE_TABLE;
  extern STBL_T_ITEMS_type STBL_T_ITEMS;
  extern int NO_LR1_STATES;
  extern T_fsm_phrase *O2_FSM_PHASE;
  extern T_parallel_parser_phrase *O2_PP_PHASE;
  extern T_enum_phrase *O2_T_ENUM_PHASE;
  extern T_lr1_k_phrase *O2_LRK_PHASE;
  extern T_rc_phrase *O2_RC_PHASE;
  extern T_error_symbols_phrase *O2_ERROR_PHASE;
  extern T_terminals_phrase *O2_T_PHASE;
  extern T_rules_phrase *O2_RULES_PHASE;
  extern STATES_type LR1_STATES;
  extern RULE_ENOSTART_OF_RULES_ENUM;
  extern STBL_T_ITEMS_type STBL_T_ITEMS;
  extern RULE_ENOSTART_OF_RULES_ENUM;
  extern yacco2::AST *GRAMMAR_TREE;
  extern yacco2::AST *CWEB_MARKER;
  extern void WRT_CWEB_MARKER(std::ofstream * Wfile, yacco2::AST * Cweb_marker);
  extern void LOAD_YACCO2_KEYWORDS_INTO_STBL( );
  extern void GET_CMD_LINE(int argc, char *argv[], const char *File, yacco2::TOKEN_GAGGLE & Errors);
  extern void DUMP_ERROR_QUEUE(yacco2::TOKEN_GAGGLE & Errors);
  extern void PRINT_RULES_TREE_STRUCTURE(AST * Node);
  extern const char *DATE_AND_TIME( );
  extern void YACCO2_PARSE_CMD_LINE
  (yacco2::CHAR & T_sw
  , yacco2::CHAR & ERR_sw, yacco2::CHAR & PRT_sw
  , std::string & Grammar_to_compile
  , yacco2::TOKEN_GAGGLE & Error_queue);
  extern bool PROCESS_INCLUDE_FILE
  (yacco2::Parser & Calling_parser
  , NS_yacco2_terminals::T_file_inclusion & File_include
```

```
, yacco2 :: token_container_type & T2);
extern bool PROCESS_KEYWORD_FOR_SYNTAX_CODE
(yacco2 :: Parser & Parser
, yacco2 :: CAbs_lr1_sym * Keyword
, yacco2 :: CAbs_lr1_sym * *Cont_tok
, yacco2 :: INT * Cont_pos);
extern void BUILD_GRAMMAR_TREE(yacco2 :: AST & Item);
extern void PRINT_GRAMMAR_TREE(AST * Node);
extern void GEN_FS_OF_RULE(NS_yacco2_terminals :: rule_def * Rule_def);
extern void GEN_CALLED_THREADS_FS_OF_RULE
(NS_yacco2_terminals :: rule_def * Start_rule);
extern int MAX_USE_CNT_RxR
(NS_yacco2_terminals :: rule_def * Rule_use, NS_yacco2_terminals :: rule_def * Against_rule);
extern void XLATE_SYMBOLS_FOR_cweave(const char *Sym_to_xlate, char *Xlated_sym);
extern void PRT_RULE_S_FIRST_SET(NS_yacco2_terminals :: rule_def * Rule_def);
extern void OP_GRAMMAR_HEADER(TOKEN_GAGGLE & Error_queue);
extern void OP_GRAMMAR_CPP(TOKEN_GAGGLE & Error_queue);
extern void OP_GRAMMAR_SYM(TOKEN_GAGGLE & Error_queue);
extern void OP_GRAMMAR_TBL(TOKEN_GAGGLE & Error_queue);
extern void OP_ENUMERATION_HEADER(TOKEN_GAGGLE & Error_queue);
extern void OP_T_Alphabet(TOKEN_GAGGLE & Error_queue);
extern void OP_ERRORS_HEADER(TOKEN_GAGGLE & Error_queue);
extern void OP_ERRORS_CPP(TOKEN_GAGGLE & Error_queue);
extern void OP_USER_T_HEADER(TOKEN_GAGGLE & Error_queue);
extern void OP_USER_T_CPP(TOKEN_GAGGLE & Error_queue);
extern void OP_FSC_FILE(TOKEN_GAGGLE & Error_queue);
extern void Print_dump_state(state * State);
```

This code is used in section 4.

6. Include Header file.

```
⟨ Include Header file 6 ⟩ ≡
#include "o2_externs.h"
```

This code is used in section 7.

7. Yacco2 external routines blueprint. Output of the code.

```
⟨ o2_externs.cpp 7 ⟩ ≡
  ⟨ Include Header file 6 ⟩;
  ⟨ accrue source for emit 8 ⟩;
```

8. Accrue source for emit.

⟨ accrue source for emit 8 ⟩ ≡

```
RULES_HAVING_AR_type RULES_HAVING_AR;
COMMON_LA_SETS_type COMMON_LA_SETS;
T_fsm_phrase * O2_FSM_PHASE(0);
T_parallel_parser_phrase * O2_PP_PHASE(0);
T_enum_phrase * O2_T_ENUM_PHASE(0);
T_lr1_k_phrase * O2_LRK_PHASE(0);
T_rc_phrase * O2_RC_PHASE(0);
T_error_symbols_phrase * O2_ERROR_PHASE(0);
T_terminals_phrase * O2_T_PHASE(0);
T_rules_phrase * O2_RULES_PHASE(0);
yacco2 :: AST * CWEB_MARKER(0);
yacco2 :: AST * GRAMMAR_TREE(0);
RULE_ENOSTART_OF_RULES_ENUM(−1);
extern STBL_T_ITEMS_type STBL_T_ITEMS;
```

See also sections 10, 11, 21, 22, 23, 24, 25, 26, 27, 28, 32, 34, 35, 37, 38, 39, 40, 41, 42, 44, 48, 49, 50, 51, 55, 56, 57, 58, 62, 70, 75, 81, 82, 83, 85, 86, 93, 98, 102, 104, 105, 106, 107, 109, 110, 111, 112, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 129, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 148, 150, 151, 152, 155, 157, 158, 163, 169, 170, 175, 179, 180, 184, 185, 186, 187, 197, 198, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 221, 222, 223, 224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 241, 242, 243, 244, 246, 248, 249, 250, 251, and 252.

This code is used in section 7.

9. Local Yacco2 routines.

10. Prescan literal names within *mpost*. Due to a bug in the "convertMPtoPDF" macro, spaces are edited.

```
⟨ accrue source for emit 8 ⟩ +≡
  void prescan_mpname_for_cweb(std::string & In, std::string & Out)
  {
    int len = In.length( );
    for (int x = 0; x < len; ++x) {
      switch (In[x]) {
      case '␣':
        {     /* substitute for epsilon */
          Out += '.';
          break;
        }
      case '\\':
        {     /* if esc seq on self just op */
          if (x + 1 < len) {      /* are there lookahead chars */
            if (In[x + 1] ≡ '\\') {      /* is it on self */
              Out += In[x];
              ++x;      /* bypass the 2nd esc */
              break;
            }
            Out += In[x];      /* nope so mpost the full expression */
            break;
          }
          else {
            Out += In[x];
            break;
          }
        }
      default:
        {
          Out += In[x];
          break;
        }
      }
    }
  }
```

11. *Print_dump_state*.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void Print_dump_state(state * State)
  {
    ⟨ print dump state 12 ⟩;
  }
```

12. ⟨ print dump state 12 ⟩ ≡

```
const char *literal_name;
yacco2::lrclog << std::endl;
yacco2::lrclog << "->State:␣";
state *cur_state = State;
literal_name = cur_state->entry_symbol_literal();
yacco2::lrclog << cur_state->state_no_ << "␣Entry:␣" << cur_state->vectored_into_by_elem_ <<
    "␣Symbol:␣" << literal_name;
if (cur_state->closure_state_birthing_it_ != 0) {
  yacco2::lrclog << "␣Birthing␣closure␣state:␣" << cur_state->closure_state_birthing_it_->state_no_;
}
yacco2::lrclog << std::endl;
yacco2::lrclog << "␣␣␣Follow␣set:" << std::endl;
S_FOLLOW_SETS_ITER_type sfi = cur_state->state_s_follow_set_map_.begin();
S_FOLLOW_SETS_ITER_type sfie = cur_state->state_s_follow_set_map_.end();
for ( ; sfi != sfie; ++sfi) {
  follow_element *fe = sfi->second;
  rule_def*␣rd␣=␣(rule_def*)AST::content(*fe->rule_def_t_);
  yacco2::lrclog << "␣␣␣␣Rule␣no:␣" << fe->rule_no_ << "␣Name:␣" << rd->rule_name()->c_str() <<
      std::endl;
  FOLLOW_SETS_ITER_type i = fe->follow_set_.begin();
  FOLLOW_SETS_ITER_type ie = fe->follow_set_.end();
  if (i != ie) {
    yacco2::lrclog << "␣␣␣␣follow␣set:" << endl;
    yacco2::lrclog << "␣␣␣␣␣␣";
  }
  int nos(1);
  for ( ; i != ie; ++i) {
    T_in_stbl *tit = *i;
    ++nos;
    if (nos > 15) {
      yacco2::lrclog << endl;
      yacco2::lrclog << "\t\t";
      nos = 1;
    }
    yacco2::lrclog << tit->t_def()->t_name()->c_str() << "␣";
  }
  TRANSITIONS_ITER_type j = fe->transitions_.begin();
  TRANSITIONS_ITER_type je = fe->transitions_.end();
  if (j != je) {
    yacco2::lrclog << "␣␣␣␣transitions" << endl;
  }
  for ( ; j != je; ++j) {
    follow_element *tfe = *j;
    rule_def*␣rd␣=␣(rule_def*)AST::content(*tfe->rule_def_t_);
    yacco2::lrclog << "␣␣␣␣␣␣S" << tfe->its_state_->state_no_ << "x" << rd->rule_name()->c_str() << endl;
  }
  MERGES_ITER_type k = fe->merges_.begin();
  MERGES_ITER_type ke = fe->merges_.end();
  if (k != ke) {
    yacco2::lrclog << "␣␣␣␣merges" << endl;
```

```
    }
    for ( ; k ≠ ke; ++k) {
      state * s = *k;
      yacco2 :: lrclog ≪ "␣␣␣␣␣␣S" ≪ s→state_no_ ≪ "␣";
    }
    yacco2 :: lrclog ≪ endl;
  }
  yacco2 :: lrclog ≪ "␣␣␣Vectors:" ≪ std :: endl;
  S_VECTORS_ITER_type svi = cur_state→state_s_vector_.begin( );
  S_VECTORS_ITER_type svie = cur_state→state_s_vector_.end( );
  for ( ; svi ≠ svie; ++svi) {
    yacco2 :: lrclog ≪ "␣␣␣␣Symbol␣no:␣" ≪ svi→first;
    int first_time(0);
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    S_VECTOR_ELEMS_ITER_type selie = svi→second.end( );
    for ( ; seli ≠ selie; ++seli) {
      state_element * se = *seli;
      if (first_time ≡ 0) {
        first_time = 1;
        CAbs_lr1_sym * sym = AST :: content(*se→sr_element_);
        yacco2 :: lrclog ≪ "␣Symbol:␣";
        switch (sym→enumerated_id__) {
        case T_Enum :: T_refered_rule_:
          {
            ⟨ get cast referenced rule 16 ⟩;
            rule_def * rd = rr→its_rule_def( );
            yacco2 :: lrclog ≪ rd→rule_name( )→c_str( ) ≪ endl;
            break;
          }
        case T_Enum :: T_refered_T_:
          {
            ⟨ get cast referenced T 17 ⟩;
            T_terminal_def * td = rt→its_t_def( );
            yacco2 :: lrclog ≪ td→t_name( )→c_str( ) ≪ endl;
            break;
          }
        default:
          {
            yacco2 :: lrclog ≪ sym→id( ) ≪ endl;
            break;
          }
        }
      }
      ⟨ dump se element 13 ⟩;
    }
  }
  if (cur_state→state_s_conflict_state_list_.empty( ) ≠ true) {
    yacco2 :: lrclog ≪ "␣␣===>Conflict␣states:" ≪ std :: endl;
    S_CONFLICT_STATES_ITER_type csi = cur_state→state_s_conflict_state_list_.begin( );
    S_CONFLICT_STATES_ITER_type csie = cur_state→state_s_conflict_state_list_.end( );
    for ( ; csi ≠ csie; ++csi) {
      state * cstate = *csi;
```

```
        yacco2 :: lrclog ≪ "␣␣␣␣␣␣State␣no:␣" ≪ cstate→state_no_ ≪ endl;
      }
    }
    yacco2 :: lrclog ≪ "End␣of␣state" ≪ std :: endl;
```

This code is used in section 11.

13.

```
⟨ dump se element 13 ⟩ ≡
    CAbs_lr1_sym ∗ sym = AST :: content(∗se→sr_element_);
    Voc_ENO id = sym→enumerated_id__;
    ⟨ dump sr element 14 ⟩;
```

This code is used in section 12.

14.

```
⟨ dump sr element 14 ⟩ ≡
  switch (id) {
  case T_Enum :: T_refered_rule_:
    {
      ⟨ get cast referenced rule 16 ⟩;
      yacco2 :: lrclog ≪ "␣␣␣␣␣␣RxSRxPos:␣" ≪ rr→grammar_s_enumerate( )→c_str( );
      rule_def * rd = rr→its_rule_def( );
      yacco2 :: lrclog ≪ "␣" ≪ rd→rule_name( )→c_str( );
      break;
    }
  case T_Enum :: T_T_eosubrule_:
    {
      ⟨ get cast referenced eosubrule 18 ⟩;
      if (se→la_set_ ≠ 0) {
        LA_SET_ITER_type i = se→la_set_→begin( );
        LA_SET_ITER_type ie = se→la_set_→end( );
        int nos(1);
        if (i ≠ ie) {
          yacco2 :: lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣" ≪ se→common_la_set_idx_ ≪ endl;
          yacco2 :: lrclog ≪ "␣␣␣␣␣␣␣␣";
          for ( ; i ≠ ie; ++i) {
            T_in_stbl * tit = *i;
            ++nos;
            if (nos > 15) {
              yacco2 :: lrclog ≪ endl;
              yacco2 :: lrclog ≪ "␣␣␣␣␣␣";
              nos = 1;
            }
            yacco2 :: lrclog ≪ tit→t_def( )→t_name( )→c_str( ) ≪ "␣";
          }
          yacco2 :: lrclog ≪ endl;
        }
        else {
          yacco2 :: lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣Empty␣see␣following␣transition" ≪ endl;
        }
      }
      yacco2 :: lrclog ≪ "␣␣␣␣␣␣RxSRxPos:␣" ≪ eos→grammar_s_enumerate( )→c_str( );
      rule_def * rd = eos→its_rule_def( );
      yacco2 :: lrclog ≪ "␣Eos␣of␣" ≪ rd→rule_name( )→c_str( );
      break;
    }
  case T_Enum :: T_T_null_call_thread_eosubrule_:
    {
      ⟨ get cast referenced null called thread eosubrule 19 ⟩;
      if (se→la_set_ ≠ 0) {
        LA_SET_ITER_type i = se→la_set_→begin( );
        LA_SET_ITER_type ie = se→la_set_→end( );
        int nos(1);
        if (i ≠ ie) {
          yacco2 :: lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣" ≪ se→common_la_set_idx_ ≪ endl;
```

```
        yacco2::lrclog ≪ "␣␣␣␣␣␣␣␣";
        for ( ; i ≠ ie; ++i) {
          T_in_stbl * tit = *i;
          ++nos;
          if (nos > 15) {
            yacco2::lrclog ≪ endl;
            yacco2::lrclog ≪ "␣␣␣␣␣␣";
            nos = 1;
          }
          yacco2::lrclog ≪ tit→t_def()→t_name()→c_str() ≪ "␣";
        }
        yacco2::lrclog ≪ endl;
      }
      else {
        yacco2::lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣Empty␣see␣following␣transition" ≪ endl;
      }
    }
    yacco2::lrclog ≪ "␣␣␣␣␣␣RxSRxPos:␣" ≪ eos→grammar_s_enumerate()→c_str();
    rule_def * rd = eos→its_rule_def();
    yacco2::lrclog ≪ "␣Eos␣of␣" ≪ rd→rule_name()→c_str();
    break;
  }
case T_Enum::T_T_called_thread_eosubrule_:
  {
    ⟨ get cast referenced called thread eosubrule 20 ⟩;
    if (se→la_set_ ≠ 0) {
      LA_SET_ITER_type i = se→la_set_→begin();
      LA_SET_ITER_type ie = se→la_set_→end();
      int nos(1);
      if (i ≠ ie) {
        yacco2::lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣" ≪ se→common_la_set_idx_ ≪ endl;
        yacco2::lrclog ≪ "␣␣␣␣␣␣␣␣";
        for ( ; i ≠ ie; ++i) {
          T_in_stbl * tit = *i;
          ++nos;
          if (nos > 15) {
            yacco2::lrclog ≪ endl;
            yacco2::lrclog ≪ "␣␣␣␣␣␣";
            nos = 1;
          }
          yacco2::lrclog ≪ tit→t_def()→t_name()→c_str() ≪ "␣";
        }
        yacco2::lrclog ≪ endl;
      }
      else {
        yacco2::lrclog ≪ "␣␣␣␣␣␣reduce␣set␣#:␣Empty␣see␣following␣transition" ≪ endl;
      }
    }
    yacco2::lrclog ≪ "␣␣␣␣␣␣RxSRxPos:␣" ≪ eos→grammar_s_enumerate()→c_str();
    rule_def * rd = eos→its_rule_def();
    yacco2::lrclog ≪ "␣Eos␣of␣" ≪ rd→rule_name()→c_str();
    break;
```

```
    }
  case T_Enum::T_refered_T_:
    {
      ⟨ get cast referenced T 17 ⟩;
      yacco2::lrclog << "␣␣␣␣␣␣RxSRxPos:␣" << rt→grammar_s_enumerate()→c_str();
      T_terminal_def * td = rt→its_t_def();
      yacco2::lrclog << "␣" << td→t_name()→c_str();
      break;
    }
  }
  yacco2::lrclog << "␣Closured␣S" << se→closure_state_→state_no_;
  yacco2::lrclog << "␣CS-gening-it␣S" << se→closured_state_gening_it_→state_no_;
  if (se→closure_state_→state_no_ ≠ se→closured_state_gening_it_→state_no_) {
    yacco2::lrclog << "␣due␣to␣rt␣bnd␣";
  }
  if (se→goto_state_ ≠ 0) {
    yacco2::lrclog << "␣goto␣S" << se→goto_state_→state_no_;
  }
  if (se→reduced_state_ ≠ 0) {
    yacco2::lrclog << "␣reduced␣S" << se→reduced_state_→state_no_;
    yacco2::lrclog << std::endl;
  }
  else {
    yacco2::lrclog << "␣reduced␣S" << "????";
    yacco2::lrclog << std::endl;
  }
```

This code is used in section 13.

15.

⟨ get subrule's referenced rule in follow string 15 ⟩ ≡

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

16.

⟨ get cast referenced rule 16 ⟩ ≡

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

This code is used in sections 12 and 14.

17.

⟨ get cast referenced T 17 ⟩ ≡

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

This code is used in sections 12 and 14.

18.

⟨ get cast referenced eosubrule 18 ⟩ ≡

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

This code is used in section 14.

19.

⟨ get cast referenced null called thread eosubrule 19 ⟩ ≡

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

This code is used in section 14.

20.

⟨ get cast referenced called thread eosubrule 20 ⟩ ≡

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

This code is used in section 14.

21. *process_fsm_phrase*.
This process demonstrates quasi recursive descent and bottom-up grammar parsing. There is no recursion taking place: just a single call to get things going. The procedure is a packaging agent that houses a monolithic grammar *Cfsm_phrase* which then uses parallel parsing threads to do its deeds. The grammars parse a stream of characters which get turned into the appropriate tokens. This is an inbetween style of parsing of the lexical and syntax phrases interweaving together. This is due to the c++ syntax directed code being just a stream of characters where a lookahead in the character stream for '***' is done to close the code directive. Of course, this code directive is parsed in parallel for literal and comment tokens which can subsume this. The reason for this skip across the characters to end the c++ code approach was laziness! I did not want to develop a c++ suite of grammars to deal with the syntax directed code. The risk in this current approach is to misprogram the end-of-code indicator '***'. This could lead to grammar stream overrun (end-of-file reached) while thinking that one is still in the syntax directed code stream. It also passes the buck to the c++ compiler to deal with any c++ syntax directed code errors. At least this run-away type error is guarded against in the grammars.

The other potential problem is the perverse pointer to a pointer to a pointer buried in the syntax directed code: "***". I thought of this situation and how to resolve it by use of a different character set: for example 3 pound signs. I nixed it as I felt it ulgy in aesthetics and as a highlighter of code boundaries. **So heed the warnin**.

The other interesting part to this family of grammars is in the use of the wild card facility to trap errors and how the subordinate grammars issue errors as normal tokens. This allows for a cleaner way to deal with error handling within a grammar's subrule phrase and associated syntax directed code. This is achieved by writing explicit production subrules using these error terminals or to program the wild card catch all facility. Take a look in the grammars at how |+| is used.

The last note is how the newly generated tokens are dealt with when an error occurs. This is rather easy, the holding token *T_fsm_phrase* is deleted. It contains all the important tokens created in its parse. Hence when there is a ruptured grammar sequence, the accumulated tokens are deleted by *T_fsm_phrase* destructor. Throw away tokens like white space and comments that are part of the fsm phrase are auto deleted when popped from the parse stack. Comments and white space within the syntax directed code are kept. This is why the output token container's contents are not deleted. There is no *yacco2*::*Delete_tokens*(...); statement in this procedure. The holding token is deposited in the passed *T_fsm* parameter. `O2_FSM_PHASE` holds the *T_fsm_phrase* terminal. Please see its family of grammars:

fsm_phrase.lex - monolithic grammar
fsm_phrase_th.lex - parse fsm phrase
fsm_attributes.lex - fsm keywords recognizer
fsm_class_directives.lex - fsm syntax code directives
fsm_class_phrase_th.lex - parse fsm-class phrase containing c++ code snippets

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_fsm_phrase
  (yacco2::Parser & Calling_parser
  , NS_yacco2_terminals::T_fsm & Phrase_to_parse
  , yacco2::CAbs_lr1_sym * *Cont_tok
  , yacco2::INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      TOKEN_GAGGLE op1;
      yacco2::INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      using namespace NS_fsm_phrase;
      Cfsm_phrase fsm;
      Parser p1(fsm, ip1, &op1, start_pos, er1);
```

```
    p1.parse( );
    if (er1→empty( ) ≡ YES) {
      *Cont_pos = p1.current_token_pos__;
      *Cont_tok = p1.current_token( );
      TOKEN_GAGGLE_ITERi = op1.begin( );
      CAbs_lr1_sym * sym = *i;
      T_fsm_phrase*␣t␣=␣(T_fsm_phrase*)sym;
      Phrase_to_parse.fsm_phrase(t);
      if (O2_FSM_PHASE ≠ 0) {
        CAbs_lr1_sym * sym = new Err_already_processed_fsm_phase;
        sym→set_rc(Phrase_to_parse);
        p1.add_token_to_error_queue(*sym);
        goto error_exit;
      }
      O2_FSM_PHASE = t;
      AST * gt = t→phrase_tree( );
      BUILD_GRAMMAR_TREE(*gt);
      return Success;
    }
  error_exit: lrclog ≪ "error␣in␣fsm-phrase" ≪ std::endl;
    return Failure;      /* error placed in error_queue by called thread */
    }
```

22. Output macros and banner.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_cweb_macros_and_banner(NS_mpost_output::Cmpost_output * Fsm, std::ofstream & Ofile,
          yacco2::KCHARP Banner, const char *Date, string * Grammar_name, string * Name_space, int
          No_t)
  {
    using namespace NS_mpost_output;
    KCHARP macros = "\\input%s␣\"supp-pdf\"\n"
    "\\input␣\"/usr/local/yacco2/diagrams/o2mac.tex\"\n";
    int x = sprintf(Fsm→big_buf_, macros, "␣");
    Ofile.write(Fsm→big_buf_, x);
    KCHARP banner = "\\DOCtitle{%s}{%s}%\n""{%s}{%i}\n";
    char fname[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Grammar_name→c_str( ), fname);
    char ns_name[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Name_space→c_str( ), ns_name);
    x = sprintf(Fsm→big_buf_, banner, Banner, fname, ns_name, No_t);
    Ofile.write(Fsm→big_buf_, x);
  }
```

23. *process_parallel_parser_phrase*.
This parses the parallel component of a grammar. It supplies the threading components and lookahead expression for a threaded grammar. One thing of note, the lookahead expression is held as a character stream which gets parsed after the complete grammar has been recognized. The reason for this is in the lookahead expression's potential use of productions (rules). Rules are packaging agents to haul in terminals as elements for the first set. As the production name used has not yet been defined, one must wait until the complete production section has been parsed before obtaining its FIRST SET of terminals for the thread's lookahead expression. Only then can the lookahead character stream be parsed in terms of "rule-in-stbl" and "T-in-stbl" tokens. From these tokens, the expression set will be calculated from their terminals. The production declaration is not restricted to use within the grammar's grammatical phrases though a check will be done with an appropriate warning if the production has not been used somewhere within the grammar or parallel parser lookahead expression.

The lookahead expression uses addition and subtraction of terminal terms to arrive at the lookahead set. The *eolr* terminal has the meaning of "use all terminals defined including self". The advantage is two fold: its supplies all the terminals thus shrinking substantially the lookahead set's size — only one terminal in its set, and two it eases the grammar writer's fingers in that the expression is much shorter to express. `O2_PP_PHASE` holds the *T_parallel_parser_phrase* terminal.

Its family of grammars are:

parallel_parser_phrase.lex - monolithic grammar
parallel_parser_phrase_th.lex - parse parallel parser construct
parallel_parser_attributes.lex - parallel parser keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_parallel_parser_phrase
  (yacco2::Parser & Calling_parser
  , NS_yacco2_terminals::T_parallel_parser & Phrase_to_parse
  , yacco2::CAbs_lr1_sym * *Cont_tok
  , yacco2::INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_parallel_parser_phrase;
      TOKEN_GAGGLE op1;
      yacco2::INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier(); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue();
      Cparallel_parser_phrase pparser;
      Parser p1(pparser, ip1, &op1, start_pos, er1);
      p1.parse();
      if (er1→empty() ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token();
        TOKEN_GAGGLE_ITER i = op1.begin();
        CAbs_lr1_sym * sym = *i;
        T_parallel_parser_phrase*␣t␣=␣(T_parallel_parser_phrase*)sym;
        Phrase_to_parse.parallel_parser_phrase(t);
        if (O2_PP_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_pp_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_PP_PHASE = t;
        AST * gt = t→phrase_tree();
```

```
    BUILD_GRAMMAR_TREE(*gt);
    return Success;
  }
error_exit: lrclog << "error in parallel-parser-phrase" << std::endl;
  return Failure;      /* error placed in error_queue by called thread */
  }
```

24. *process_T_enum_phrase*.
Its use is in housing all the enumerated terminals from the generated alphabets and possible user included constant definitions that are needed outside the grammars. These span the spectrums of errors, raw characters, LR constants, and finally the pedestrian terminals.

The enumeration is lr k starting at 0 < raw characters < error < regular terminals. As raw chars and "lr k" terminals are constant, the other two alphabets grow outwards in the right direction as new members are defined. `O2_T_ENUM_PHASE` holds the *T_enum_phrase* terminal.

Its family of grammars are:

T_enum_phrase.lex - monolithic grammar
T_enum_phrase_th.lex - parse *T_enumeration* construct
T_enum_attributes.lex - *T_enumeration* keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_T_enum_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_enumeration & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_T_enum_phrase;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      CT_enum_phrase enum_ph;
      Parser p1(enum_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym * sym = *i;
        T_enum_phrase* t = (T_enum_phrase*)sym;
        Phrase_to_parse.enum_phrase(t);
        if (O2_T_ENUM_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_T_enum_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_T_ENUM_PHASE = t;
        AST * gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(*gt);
        return Success;
      }
    error_exit: lrclog ≪ "error in t-enum-phrase" ≪ std :: endl;
      return Failure;      /* error placed in error_queue by called thread */
      }
```

25. *process_error_symbols_phrase*.
This handles the error alphabet introduced to the grammar. It is normally placed into its own file and brought into the grammar by way of the grammar's include file facility. It was designed this way to segregate the terminal symbols both in definition and own namespace enclosure. The same holds for the other terminal types: raw characters, LR constants, and the regular terminals. `O2_ERROR_PHASE` holds the *T_error_symbols_phrase* terminal.

Its family of grammars are:

err_symbols_ph.lex - monolithic grammar
err_symbols_ph_th.lex - parse error symbols construct
error_symbols_attributes.lex - error symbols keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_error_symbols_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_error_symbols & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_err_symbols_ph;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      Cerr_symbols_ph err_ph;
      Parser p1 (err_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym * sym = *i;
        T_error_symbols_phrase* t = (T_error_symbols_phrase*)sym;
        Phrase_to_parse.error_symbols_phrase(t);
        if (O2_ERROR_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_error_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_ERROR_PHASE = t;
        AST * gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(*gt);
        return Success;
      }
    error_exit: lrclog ≪ "error in error-symbols-phrase" ≪ std :: endl;
      return Failure;      /* error placed in error_queue by called thread */
      }
```

26. *process_rc_phrase*.
This handles the raw character terminals. This is the alphabet of raw character terminals introduced to the grammar. It is normally placed into its own file and brought into the grammar by way of the grammar's include file facility. It was designed this way to segregate the terminal symbols both in definition and own namespace enclosure. `O2_RC_PHASE` holds the *T_rc_phrase* terminal.

Its family of grammars are:

rc_phrase.lex - monolithic grammar
rc_phrase_th.lex - parse error symbols construct
rc_attributes.lex - raw character symbols keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_rc_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_raw_characters & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym ∗ ∗Cont_tok
  , yacco2 :: INT ∗ Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_rc_phrase;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type ∗ ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE ∗ er1 = ( TOKEN_GAGGLE
          ∗ ) Calling_parser.error_queue( );
      Crc_phrase rc_ph;
      Parser p1 (rc_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        ∗Cont_pos = p1.current_token_pos__;
        ∗Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym ∗ sym = ∗i;
        T_rc_phrase*␣t␣=␣(T_rc_phrase*)sym;
        Phrase_to_parse.rc_phrase(t);
        if (O2_RC_PHASE ≠ 0) {
          CAbs_lr1_sym ∗ sym = new Err_already_processed_rc_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(∗sym);
          goto error_exit;
        }
        O2_RC_PHASE = t;
        AST ∗ gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(∗gt);
        return Success;
      }
    error_exit: lrclog ≪ "error␣in␣rc-phrase" ≪ std :: endl;
      return Failure;      /∗ error placed in error_queue by called thread ∗/
      }
```

27. *process_lr1_k_phrase*.
This handles the *lr_k* of constants used throughout the grammars: for example, end-of-grammar, parallel operators and the like. It is normally placed into its own file and brought into the grammar by way of the grammar's include file facility. It was designed this way to segregate the terminal symbols both in definition and own namespace enclosure. `O2_LRK_PHASE` holds the *T_lr1_k_phrase* terminal.

Its family of grammars are:

lr1_k_phrase.lex - monolithic grammar
lr1_k_phrase_th.lex - parse lr1 k symbols construct
lr1_k_attributes.lex - lr1 k symbols keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_lr1_k_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_lr1_constant_symbols & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_lr1_k_phrase;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      Clr1_k_phrase lr_ph;
      Parser p1 (lr_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym * sym = *i;
        T_lr1_k_phrase* t = (T_lr1_k_phrase*)sym;
        Phrase_to_parse.lr1_k_phrase(t);
        if (O2_LRK_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_lrk_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_LRK_PHASE = t;
        AST * gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(*gt);
        return Success;
      }
    error_exit: lrclog ≪ "error in lr1-k-phrase" ≪ std :: endl;
      return Failure;      /* error placed in error_queue by called thread */
      }
```

28. *process_terminals_phrase*.
This handles the regular terminals alphabet introduced to the grammar. It is normally placed into its own file and brought into the grammar by way of the grammar's include file facility. It was designed this way to segregate the terminal symbols both in definition and own namespace enclosure. This is the grammar writer's stable of terminals used by the grammars to process the specific language. `O2_T_PHASE` holds the *T_terminals_phrase* terminal.

Its family of grammars are:

terminals_phrase.lex - monolithic grammar
terminals_phrase_th.lex - parse terminal symbols construct
terminals_attributes.lex - terminal symbols keywords recognizer

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_terminals_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_terminals & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
      { using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_terminals_phrase;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      Cterminals_phrase term_ph;
      Parser p1 (term_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym * sym = *i;
        T_terminals_phrase* t = (T_terminals_phrase*)sym;
        Phrase_to_parse.terminals_phrase(t);
        if (O2_T_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_T_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_T_PHASE = t;
        AST * gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(*gt);
        return Success;
      }
    error_exit: lrclog ≪ "error in terminals-phrase" ≪ std :: endl;
      return Failure;      /* error placed in error_queue by called thread */
      }
```

29. *process_rules_phrase*.
This parses the rule's alphabet of the grammar. In grammar parlance, this is the *productions* and *rules* rolled into one. The rule names are local to the grammar. They do not use global space allowing the grammar writer to assign meaningful rule names of reuse patterns expressed throughout each grammar.

The 'fsm' structure of the grammar has a namespace property that allows each grammar to be mixed in global space. `O2_RULES_PHASE` holds the *T_rules_phrase* terminal.

Why the enumeration of the Terminal alphabet before processing rules? I wanted it as a discrete process rather than distributed throughout pieces of grammars, and to improve rule parsing, and i reference the parallel operator terminal "|||" specifically for clarity in the sub-rule parsing. How so clarity? I felt dividing a grammar's productions into thread expressions or standard symbol strings made the subrule grammar more readable. It sharpened the distinction between the 2 types of grammars: monolithic and thread. Due to this, the |||was explicitly programmed and its presence tested for using a fixed enumeration value rather than its literal string name.

Its family of grammars are:

rules_phrase.lex - monolithic grammar
rules_phrase_th.lex - parse terminal symbols construct
terminals_attributes.lex - terminal symbols keywords recognizer
rule_def_phrase.lex - parse rule's definition construct
rule_lhs_phrase.lex - parse rule's lhs construct
parallel_monitor_phrase.lex - parse rule's arbitration construct
subrules_phrase.lex - parse subrule's definition construct
subrule_def .lex - parse a subrule's rhs expression

30. Enumerate Terminal alphabet.
Each terminal phase contains its mapped symbols and "create order" list. The implied enumeration starts with the lrk symbols followed by raw characters, errors, and finally the terminals. The rank order is 0..n-1 where n is the total number of terminals in the alphabet. The "create order" list provides the symbol order. As i rant on relative zero from previous writings regarding vector access etc, why not here regarding the rank assignment from 0? Modulo arithmetic is used on the register's number of bits to calculate a specific element in the first set: a divisional way of doing things.

To traverse the terminal symbols, *O2_xxx_PHASE* per phases. Each specific terminal phrase will be accessed and its symbols traversed by "create order" to assign their appropriate enumerate value. Each phase derives from the same base class :

0 - `O2_FSM_PHASE` : *T_fsm_phrase**
1 - `O2_PP_PHASE` : *T_parallel_parser_phrase**
2 - `O2_T_ENUM_PHASE` : *T_enum_phrase**
3 - `O2_LRK_PHASE` : *T_lr1_k_phrase**
4 - `O2_RC_PHASE` : *T_rc_phrase**
5 - `O2_ERROR_PHASE` : *T_error_symbols_phrase**
6 - `O2_T_PHASE` : *T_terminals_phrase**
7 - `O2_RULES_PHASE` : *T_rules_phrase**

Why use a grammar anyway when straight procedure calls would do the trick? Good question as the token stream is not even being consumed! This is my experiment in using grammars as a descriptor of software events — not as efficient as in procedure calls but the intentions are explicit. Is not the procedure call explicit in intent? Yes it is within a no token consumption context. So, does this grammar approach look like an inefficient double indirection to achieve the same result. Yup. But i also use a sentence approach within the grammar of right-hand-side left-hand-side recognition to calculate the overall total of symbols enumerated. This processing model is a mainstay to pre-post event evaluation. Also, my grammars have automatic flow control tracing facility that comes out-of-the-box.

Now consider when other events are surrounding a software process. Grammars are explicit finite state automata with ambiguity verification. I could have created behavioural terminals to explicitly express execution by the "shift operation" but this is overkill in this situation. For now this grammar has implicit behaviours using epsilon.

It is interesting to reflect on how epsilon with no terminals can be used as an operation sequencer. Only the "error" container is passed to the parser. The i/o token containers are not supplied.

⟨ enumerate Terminal alphabet 30 ⟩ ≡

```
lrclog ≪ "Enumerate␣T␣alphabet" ≪ endl;
using namespace NS_enumerate_T_alphabet;
Cenumerate_T_alphabet enum_syms_fsm;
Parser enum_syms(enum_syms_fsm, 0, 0, 0, Calling_parser.error_queue( ), 0, 0);
enum_syms.parse( );
if (Calling_parser.error_queue( )→empty( ) ≠ true) return Failure;
```

This code is used in section 32.

31. Enumerate Rule alphabet.
Enumeration starts after the errors vocabulary. The `START_OF_RULES_ENUM` global now registers the rules' enumerate boundary. To speed up the LR1 compatibilty check, this boundary becomes the cut off point in shift / reduce evaluations. *eosubrule* enumerate is excluded in the Terminals shift set of the state as it represents the "end-of-subrule" string condition: reduce.

To speed up the pushdown automata, rules are recyclesd for re-use. To do this each rule needs a rule no relative to its create order starting from 1.

⟨ enumerate Rule alphabet 31 ⟩ ≡
```
lrclog ≪ "Enumerate␣Rule␣alphabet" ≪ endl;
int Rules_enum = START_OF_RULES_ENUM;
RULE_DEFS_TBL_ITER_type ri = t→crt_order( )→begin( );
RULE_DEFS_TBL_ITER_type rie = t→crt_order( )→end( );
int rule_no(0);
for ( ; ri ≠ rie; ++ri, ++Rules_enum) {
  rule_def * rd = *ri;
  ++rule_no;
  rd→enum_id(Rules_enum);
  rd→rule_no(rule_no);
}
```
This code is used in section 32.

32. Driver of *process_rules_phrase*.

```
⟨ accrue source for emit 8 ⟩ +≡
  bool process_rules_phrase
  (yacco2 :: Parser & Calling_parser
  , NS_yacco2_terminals :: T_rules & Phrase_to_parse
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
      { ⟨ enumerate Terminal alphabet 30 ⟩;
      using namespace NS_yacco2_terminals;
      using namespace yacco2;
      using namespace NS_rules_phrase;
      TOKEN_GAGGLE op1;
      yacco2 :: INT start_pos = Calling_parser.current_token_pos__;
      token_container_type * ip1 = Calling_parser.token_supplier( ); TOKEN_GAGGLE * er1 = ( TOKEN_GAGGLE
          * ) Calling_parser.error_queue( );
      Crules_phrase rule_ph;
      Parser p1 (rule_ph, ip1, &op1, start_pos, er1);
      p1.parse( );
      if (er1→empty( ) ≡ YES) {
        *Cont_pos = p1.current_token_pos__;
        *Cont_tok = p1.current_token( );
        TOKEN_GAGGLE_ITER i = op1.begin( );
        CAbs_lr1_sym * sym = *i;
        T_rules_phrase* t = (T_rules_phrase*)sym;
        Phrase_to_parse.rules_phrase(t);
        if (O2_RULES_PHASE ≠ 0) {
          CAbs_lr1_sym * sym = new Err_already_processed_rule_phase;
          sym→set_rc(Phrase_to_parse);
          p1.add_token_to_error_queue(*sym);
          goto error_exit;
        }
        O2_RULES_PHASE = t;
        AST * gt = t→phrase_tree( );
        BUILD_GRAMMAR_TREE(*gt);
        ⟨ enumerate Rule alphabet 31 ⟩;
        return Success;
      }
    error_exit: lrclog ≪ "error in rules-phrase" ≪ std :: endl;
      return Failure;      /* error placed in error_queue by called thread */
      }
```

33. External routines.

34. Yacco2 Parse command line: `YACCO2_PARSE_CMD_LINE`.
The parameters have been extracted from the program run environment and placed into Yacco2's holding file which is hardwired by *Yacco2_holding_file* definition. Now it's time to parse them for kosherness.

Constraints:
ip1-4: switches used to compile grammar
ip8: To compile grammar name extracted from parameters
ip9: Error token container for generated error token

Errors:
1) bad filename
2) parameters errors

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void YACCO2_PARSE_CMD_LINE
  (yacco2::CHAR & T_sw, yacco2::CHAR & ERR_sw
  , yacco2::CHAR & PRT_sw
  , std::string & Grammar_to_compile
  , yacco2::TOKEN_GAGGLE & Error_queue)
  {
    using namespace NS_yacco2_err_symbols;
    using namespace yacco2;
    tok_can < std::ifstream > Cmd1_tokens(Yacco2_holding_file);
    if (Cmd1_tokens.file_ok() ≡ NO) {
      yacco2::Delete_tokens(Cmd1_tokens.container());
      CAbs_lr1_sym * sym = new Err_bad_filename(Yacco2_holding_file);
      sym→set_external_file_id(1);
      sym→set_line_no(1);
      sym→set_pos_in_line(1);
      Error_queue.push_back(*sym);
      return;
    }
    using namespace NS_o2_lcl_opts;
    TOKEN_GAGGLE lcl_options_tokens;
    Co2_lcl_opts opts_fsm;
    Parser options(opts_fsm, &Cmd1_tokens, &lcl_options_tokens, 0, &Error_queue, 0, 0);
    options.parse();
    yacco2::Delete_tokens(Cmd1_tokens.container());
    if (Error_queue.empty() ≠ YES) return;
    T_sw = opts_fsm.t_sw_;
    ERR_sw = opts_fsm.err_sw_;
    PRT_sw = opts_fsm.prt_sw_;
    Grammar_to_compile += opts_fsm.file_to_compile_;
  }
```

35. BUILD_GRAMMAR_TREE while parsing grammar.
Builds a single level tree having the following forests: all the parse phrases: for example FSM_PHASE etc. With the introduction of *cweb* comments, a *cweb* holding queue of comments is kept so that the comments can placed into the following phrase. Why? For example, a series of *cweb* comments could act as an introduction before the "fsm" phrase is detected by the lexical grammar *pass3* by its "fsm" keyword. So what do u do with the orphaned *cweb* comments recognized by *pass3* before the "fsm" phrase parse? Well call BUILD_GRAMMAR_TREE and discriminate what is passed to it. If it's a *cweb* comment put it into a holding list. When BUILD_GRAMMAR_TREE is called by a phrase process, only then graft it into the passed phrase terminal. The only exception to this is *cweb* comments after the end of the grammar's phrases have been parsed. This comment is then placed as the last node in the grammar tree.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void BUILD_GRAMMAR_TREE(yacco2::AST & Item)
  {
    using namespace NS_yacco2_T_enum;
    using namespace NS_yacco2_terminals;
    using namespace yacco2;
    CAbs_lr1_sym * sym = AST::content(Item);
    static AST*cur_node(0);
    static AST*holding_kcweb(0);
    static AST*end_holding_kcweb(0);
    static bool rules_phrase_seen(false);
    if (sym→enumerated_id__ ≡ T_Enum::T_T_cweb_marker_) {
      if (rules_phrase_seen ≡ true) {
        AST::join_sts(*cur_node, Item);
        cur_node = &Item;
        return;
      }
      if (holding_kcweb ≡ 0) {
        holding_kcweb = &Item;
      }
      else {
        if (end_holding_kcweb ≡ 0) {
          end_holding_kcweb = &Item;
          AST::join_sts(*holding_kcweb, *end_holding_kcweb);
        }
        else {
          AST::join_sts(*end_holding_kcweb, Item);
          end_holding_kcweb = &Item;
        }
      }
      return;
    }
    if (cur_node ≡ 0) {
      CAbs_lr1_sym * gp = new T_grammar_phrase( );
      gp→set_line_no_and_pos_in_line(*sym);
      gp→set_external_file_id(sym→tok_co_ords__.external_file_id__);
      GRAMMAR_TREE = new AST(*gp);
      AST::crt_tree_of_1son(*GRAMMAR_TREE, Item);
      cur_node = &Item;
    }
    else {
```

```
    AST::join_sts(*cur_node, Item);
    cur_node = &Item;
  }
  if (sym->enumerated_id__ ≡ T_Enum::T_T_rules_phrase_) {
    rules_phrase_seen = true;
  }
  ⟨ handle the cweb comments that prefix phrases 36 ⟩;
}
```

36. Handle the *cweb* comments that prefix phrases.
cweb comments are chained forests where the first comment will get placed into the calling phrase. i use the grammar to elegantly handle the the various phrase rather than casing them.

```
⟨ handle the cweb comments that prefix phrases 36 ⟩ ≡
  if (holding_kcweb ≡ 0) return;
  tok_can_ast_functor just_walk_functr;
  ast_prefix_1forest element_walk(Item, &just_walk_functr);
  tok_can < AST *> phrase_can(element_walk);
  using namespace NS_cweb_put_k_into_ph;
  Ccweb_put_k_into_ph cweb_k_fsm;
  cweb_k_fsm.initialize(holding_kcweb, &rules_phrase_seen);
  Parser cweb_k(cweb_k_fsm, &phrase_can, 0);
  cweb_k.parse( );
  holding_kcweb = 0;
  end_holding_kcweb = 0;
```

This code is used in section 35.

37. LOAD_YACCO2_KEYWORDS_INTO_STBL.
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. Why cloning of keywords? Cuz of their containment properties — e.g., syntax code directives.

Why the kludge in *load_kw_into_tbl*? The problem is a keyword like "fsm" cannot be represented as a terminal with the same literal name "fsm". So i prefixed the terminal's literal name as "#fsm" allowing the T alphabet parse to go thru. Without this change, the symbol table would return the terminal's literal name as the keyword "fsm" which is a parse error. The symbol table contains the real literal name whilst the terminal has its symbolic name sake. To condense the loading of the table code, i use the terminal's literal name and guard against its symbolic name.

```
⟨ accrue source for emit 8 ⟩ +≡
  void load_kw_into_tbl(yacco2::CAbs_lr1_sym * Kw)
  {
    using namespace yacco2_stbl;
    T_sym_tbl_report_card report_card;
    const char *kwkey = Kw->id__;
    if (*kwkey == '#') ++kwkey;      /* kludge: bypass 1st char eg "#fsm" */
    kw_in_stbl * kw = new kw_in_stbl(Kw);
    add_sym_to_stbl(report_card, *kwkey, *kw, table_entry::defed, table_entry::keyword);
    kw->stbl_idx(report_card.pos_);
  }
  extern void LOAD_YACCO2_KEYWORDS_INTO_STBL()
  {
    using namespace yacco2_stbl;
    load_kw_into_tbl(new T_raw_characters);
    load_kw_into_tbl(new T_lr1_constant_symbols);
    load_kw_into_tbl(new T_error_symbols);
    load_kw_into_tbl(new T_eocode);
    load_kw_into_tbl(new T_AD);
    load_kw_into_tbl(new T_AB);
    load_kw_into_tbl(new T_parallel_la_boundary);
    load_kw_into_tbl(new T_arbitrator_code);
    load_kw_into_tbl(new T_parallel_parser);
    load_kw_into_tbl(new T_parallel_thread_function);
    load_kw_into_tbl(new T_parallel_control_monitor);
    load_kw_into_tbl(new T_fsm);
    load_kw_into_tbl(new T_fsm_id);
    load_kw_into_tbl(new T_fsm_filename);
    load_kw_into_tbl(new T_fsm_namespace);
    load_kw_into_tbl(new T_fsm_class);
    load_kw_into_tbl(new T_fsm_version);
    load_kw_into_tbl(new T_fsm_date);
    load_kw_into_tbl(new T_fsm_debug);
    load_kw_into_tbl(new T_fsm_comments);
    load_kw_into_tbl(new T_terminals);
    load_kw_into_tbl(new T_enumeration);
    load_kw_into_tbl(new T_file_name);
```

```
    load_kw_into_tbl(new T_name_space);
    load_kw_into_tbl(new T_sym_class);
    load_kw_into_tbl(new T_rules);
    load_kw_into_tbl(new T_lhs);
    load_kw_into_tbl(new T_user_declaration);
    load_kw_into_tbl(new T_user_prefix_declaration);
    load_kw_into_tbl(new T_user_suffix_declaration);
    load_kw_into_tbl(new T_constructor);
    load_kw_into_tbl(new T_destructor);
    load_kw_into_tbl(new T_op);
    load_kw_into_tbl(new T_failed);
    load_kw_into_tbl(new T_user_implementation);
    load_kw_into_tbl(new T_user_imp_tbl);
    load_kw_into_tbl(new T_user_imp_sym);
    load_kw_into_tbl(new T_constant_defs);
    load_kw_into_tbl(new T_terminals_refs);
    load_kw_into_tbl(new T_terminals_sufx);
    load_kw_into_tbl(new T_lrk_sufx);
    load_kw_into_tbl(new T_NULL);
  }
```

38. Beauty and the tree: PRINT_RULES_TREE_STRUCTURE.
Print out the tree's contents. It is a function that works with Yac$_2o_2$'s *prt_ast_functor* and a tree walker like *ast_prefix*. See O_2 documentation for an example of use and Yac$_2o_2$ documentation on functors and tree traversals — good stuff.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void PRINT_RULES_TREE_STRUCTURE(AST * Node)
  {
    using namespace NS_yacco2_T_enum;
    using namespace NS_yacco2_terminals;
    if (Node ≡ 0) return;
    CAbs_lr1_sym * sym = AST::content(*Node);
    if (sym ≡ 0) {
      yacco2::lrclog ≪ "␣SYMBOL␣IN␣AST␣CONTENT␣NULL" ≪ "␣Node*:␣" ≪ Node ≪ endl;
      return;
    }
    yacco2::lrclog ≪ sym→id__ ≪ '␣';
    switch (sym→enumerated_id__) {
    case T_Enum::T_refered_T_:
      {
        refered_T*␣sym1␣=␣(refered_T*)sym;
        T_in_stbl * T = sym1→t_in_stbl();
        yacco2::lrclog ≪ T→t_def()→t_name()→c_str();
        break;
      }
    case T_Enum::T_refered_rule_:
      {
        refered_rule*␣sym1␣=␣(refered_rule*)sym;
        rule_in_stbl * R = sym1→Rule_in_stbl();
        yacco2::lrclog ≪ R→r_def()→rule_name()→c_str();
        break;
      }
    case T_Enum::T_T_identifier_:
      {
        T_identifier*␣sym1␣=␣(T_identifier*)sym;
        yacco2::lrclog ≪ sym1→identifier()→c_str();
        break;
      }
    case T_Enum::T_rule_def_:
      {
        rule_def*␣sym1␣=␣(rule_def*)sym;
        yacco2::lrclog ≪ sym1→rule_name()→c_str();
        yacco2::lrclog ≪ "␣epsilon:␣";
        if (sym1→epsilon() ≡ true) {
          yacco2::lrclog ≪ "␣Y␣";
        }
        else {
          yacco2::lrclog ≪ "␣N␣";
        }
        break;
      }
    case T_Enum::T_T_terminal_def_:
      {
```

```
      T_terminal_def*␣sym1␣=␣(T_terminal_def*)sym;
      yacco2::lrclog ≪ sym1→t_name()→c_str();
      break;
    }
  case T_Enum::T_T_subrule_def_:
    {
      T_subrule_def*␣sym1␣=␣(T_subrule_def*)sym;
      yacco2::lrclog ≪ "␣epsilon:␣";
      if (sym1→epsilon() ≡ true) {
        yacco2::lrclog ≪ "␣Y␣";
      }
      else {
        yacco2::lrclog ≪ "␣N␣";
      }
      break;
    }
  case T_Enum::T_T_eosubrule_:
    {
      break;
    }
  case T_Enum::T_T_NULL_:
    {
      break;
    }
  case T_Enum::T_T_2colon_:
    {
      break;
    }
  default:
    {
      yacco2::lrclog ≪ sym→id__;
    }
  }
  yacco2::lrclog ≪ "␣file#␣" ≪ sym→tok_co_ords__.external_file_id__ ≪ ':' ≪
      sym→tok_co_ords__.rc_pos__ ≪ ':';
  yacco2::lrclog ≪ "␣line#␣" ≪ sym→tok_co_ords__.line_no__ ≪ ':' ≪ sym→tok_co_ords__.pos_in_line__ ≪
      ':';
  yacco2::lrclog ≪ "␣sym*:␣" ≪ sym ≪ '␣';
  yacco2::lrclog ≪ endl;
}
```

39. `WRT_CWEB_MARKER`.
Just write out to a cweave file the cweb comments.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void WRT_CWEB_MARKER(std::ofstream * Wfile, yacco2::AST * Cweb_marker){ using namespace
        NS_yacco2_T_enum;
    using namespace NS_yacco2_terminals;
    using namespace yacco2;
    INT_SET_type filter;
    filter.insert(T_Enum::T_T_cweb_comment_);
    tok_can_ast_functor just_walk_functr;
    ast_prefix_1forest rule_walk(*Cweb_marker, &just_walk_functr, &filter, ACCEPT_FILTER);
    tok_can < AST *> comments_can(rule_walk); for (int x(0);
        comments_can[x] ≠ yacco2::PTR_LR1_eog__; ++x) { T_cweb_comment * k = (
        T_cweb_comment * ) comments_can[x];
    (*Wfile) ≪ k→comment_data( )→c_str( ) ≪ endl; } }
```

40. ?Beauty and the tree: `PRINT_GRAMMAR_TREE`.
Print out the grammar's contents. This includes the *cweb* contents. It is a function that works with Yac$_2$o$_2$'s *prt_ast_functor* and a tree walker like *ast_prefix*. See O_2 documentation for an example of use and Yac$_2$o$_2$ documentation on functors and tree traversals — good stuff.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern void PRINT_GRAMMAR_TREE(AST * Node)
  {
    using namespace NS_yacco2_T_enum;
    using namespace NS_yacco2_terminals;
    if (Node ≡ 0) return;
    CAbs_lr1_sym * sym = AST::content(*Node);
    if (sym ≡ 0) {
      yacco2::lrclog ≪ "␣SYMBOL␣IN␣AST␣CONTENT␣NULL" ≪ "␣Node*:␣" ≪ Node ≪ "␣lt*:␣" ≪
          Node→lt_ ≪ "␣rt*:␣" ≪ Node→rt_ ≪ std::endl;
      return;
    }
    yacco2::lrclog ≪ sym→id__ ≪ '␣';
    switch (sym→enumerated_id__) {
    case T_Enum::T_T_cweb_comment_:
      {
        T_cweb_comment*␣sym1␣=␣(T_cweb_comment*)sym;
        yacco2::lrclog ≪ sym1→comment_data( )→c_str( );
        break;
      }
    }
    yacco2::lrclog ≪ "␣file#␣" ≪ sym→tok_co_ords__.external_file_id__ ≪ ':' ≪
        sym→tok_co_ords__.rc_pos__ ≪ ':';
    yacco2::lrclog ≪ "␣line#␣" ≪ sym→tok_co_ords__.line_no__ ≪ ':' ≪ sym→tok_co_ords__.pos_in_line__ ≪
        ':';
    yacco2::lrclog ≪ "␣sym*:␣" ≪ sym ≪ '␣';
    yacco2::lrclog ≪ "␣Node*:␣" ≪ Node ≪ "␣lt*:␣" ≪ Node→lt_ ≪ "␣rt*:␣" ≪ Node→rt_ ≪ std::endl;
  }
```

41. Process Nested include files: PROCESS_INCLUDE_FILE.
This routine gets called from the *pass3* grammar when Yacco2's include file expression has been parsed. It uses the hardwired definition *Nested_file_cnt_limit*. The one important point is how it appends the newly processed include tokens to the calling parser's producer container. *pass3* has a global variable to ensure that only 2 *PTR_LR1_eog__* tokens are emitted to the producer container from the first *pass3* invocation. All other recursively invoked *pass3* guard against these extra end-of-grammar tokens when they finish parsing.

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

Constraints:

ip1: Calling parser environment
ip2: include token to process
ip3: Token container to append to

Note: The filename to process has already had its existence check done by the *T_file_inclusion* token.

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

Error processing:

op1: nested file limit is exceeded
op2: bad file name

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

```
⟨ accrue source for emit 8 ⟩ +≡
  extern bool PROCESS_INCLUDE_FILE
  (yacco2::Parser & Calling_parser
  ,NS_yacco2_terminals::T_file_inclusion & File_include
  ,yacco2::token_container_type & T2)
  {
    using namespace NS_yacco2_terminals;
    using namespace NS_yacco2_err_symbols;
    using namespace yacco2;
    std::string * ps_fn = File_include.file_name()->c_string();
    lrclog << "Pre-process␣file:␣" << ps_fn->c_str() << std::endl;
    if (yacco2::STK_FILE_NOS__.size() ≥ Nested_file_cnt_limit) {
      CAbs_lr1_sym * s = new Err_nested_files_exceeded(Nested_file_cnt_limit, *ps_fn);
      s->set_line_no_and_pos_in_line(File_include);
      s->set_external_file_id(File_include.tok_co_ords__.external_file_id__);
      Calling_parser.add_token_to_error_queue(*s);
      Calling_parser.abort_parse__;
      return Failure;
    }
    tok_can < std::ifstream > p1_tokens(ps_fn->c_str());
    if (p1_tokens.file_ok() ≡ NO) {
      yacco2::Delete_tokens(p1_tokens.container());
      CAbs_lr1_sym * sym = new NS_yacco2_err_symbols::Err_bad_filename(*ps_fn);
      sym->set_external_file_id(File_include.tok_co_ords__.external_file_id__);
      sym->set_line_no_and_pos_in_line(File_include);
      Calling_parser.add_token_to_error_queue(*sym);
```

```
    Calling_parser.abort_parse__;
    return Failure;
  }
  using namespace NS_pass3;
  Cpass3 p3_fsm;
  Parser pass3(p3_fsm, &p1_tokens, &T2, 0, Calling_parser.error_queue( ), Calling_parser.recycle_bin__);
  pass3.parse( );
  yacco2::Delete_tokens(p1_tokens.container( ));
  if (pass3.error_queue( )→empty( ) ≠ YES) {
    Calling_parser.abort_parse__;
    return Failure;
  }
  yacco2::STK_FILE_NOS__.pop_back( );
  return Success;
}
```

42. Process syntax code: PROCESS_KEYWORD_FOR_SYNTAX_CODE.
This is the dispatch routine called from the *pass3* grammar per phase to parse. What you will see within the *pass3.lex* grammar is a subrule with the appropriate keyword. Reality is the grammar thread *identifier* returns a carrier terminal called *keyword*. Inside it is the individual keyword recognized that must be extracted by the syntax directed code. This was done to lower the amount of subrules within *pass3*. This is the partial truth; evolution in the wild card facility also lowers the explicit programming of subrules. The extracted keyword is passed to this routine for processing. If a success is returned, the extracted keyword is put into the output container while an error stops the parse with the error placed into the 'error queue'. The carrier terminal *keyword* has the 'AD' auto delete attribute defined so *keyword* is deleted when it is popped from the parse stack.

This is a simple way to deal with individual components. Within each of the to-be-parsed phases, their syntax is gramatically verified. As it comes out of a lexical grammar, down the road, the syntactic part must check for the proper sequencing of these phases. For example, the production component cannot come before the LRk phase etc. Due to a fixed ordering of T vocabulary components enumeration, T's components must be in lrk,rc,user, and error terminals parse order as i add their forests to the grammar tree while they are being build! Out of order will violate the T vocabulary's enumeration as i walk the tree as created when enumerating.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern bool PROCESS_KEYWORD_FOR_SYNTAX_CODE
  (yacco2 :: Parser & Parser
  , yacco2 :: CAbs_lr1_sym * Keyword
  , yacco2 :: CAbs_lr1_sym * *Cont_tok
  , yacco2 :: INT * Cont_pos)
  {
    using namespace NS_yacco2_T_enum;
    using namespace NS_yacco2_terminals;
    using namespace yacco2;
    switch (Keyword→enumerated_id__) {
    case T_Enum :: T_T_fsm_:
      {
        T_fsm*␣fsm␣=␣(T_fsm*)Keyword;
        return process_fsm_phrase(Parser, *fsm, Cont_tok, Cont_pos);
      }
    case T_Enum :: T_T_parallel_parser_:
      {
        T_parallel_parser*␣pparser␣=␣(T_parallel_parser*)Keyword;
        return process_parallel_parser_phrase(Parser, *pparser, Cont_tok, Cont_pos);
      }
    case T_Enum :: T_T_enumeration_:
      {
        T_enumeration*␣enumer␣=␣(T_enumeration*)Keyword;
        return process_T_enum_phrase(Parser, *enumer, Cont_tok, Cont_pos);
      }
    case T_Enum :: T_T_error_symbols_:
      {
        T_error_symbols*␣err␣=␣(T_error_symbols*)Keyword;
        return process_error_symbols_phrase(Parser, *err, Cont_tok, Cont_pos);
      }
    case T_Enum :: T_T_raw_characters_:
      {
        T_raw_characters*␣rc␣=␣(T_raw_characters*)Keyword;
        return process_rc_phrase(Parser, *rc, Cont_tok, Cont_pos);
```

```
    }
  case T_Enum :: T_T_lr1_constant_symbols_:
    {
      T_lr1_constant_symbols* lr = (T_lr1_constant_symbols*)Keyword;
      return process_lr1_k_phrase(Parser, *lr, Cont_tok, Cont_pos);
    }
  case T_Enum :: T_T_terminals_:
    {
      T_terminals* term = (T_terminals*)Keyword;
      return process_terminals_phrase(Parser, *term, Cont_tok, Cont_pos);
    }
  case T_Enum :: T_T_rules_:
    {
      T_rules* rule = (T_rules*)Keyword;
      return process_rules_phrase(Parser, *rule, Cont_tok, Cont_pos);
    }
  default:
    {
      CAbs_lr1_sym * sym = new Err_not_kw_defining_grammar_construct;
      sym->set_rc(*Keyword);
      Parser.add_token_to_error_queue(*sym);
      return Failure;
    }
  }
  return Success;
}
```

43. Mpost and Cweb Routines.
These external routines simplify the *cweave* code generated for the *mpost_output* grammar. As i do not parse the syntax code into appropriate structures but build up strings of characters, at least i can beautify the emitted *cweave* grammar code by use of external rountines that are under *cweb*'s generation.

44. `MPOST_CWEB_LOAD_XLATE_CHRS.`
These are the raw characters whose values are given a literal title. Why? Some raw characters will cause *cweave* to cough. So i'm attempting to avoid this clearing of the throat by character translation as I'm not wanting to parse formally the syntax code and to emit yet another properly formed language.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_LOAD_XLATE_CHRS
  (NS_mpost_output::Cmpost_output * Fsm)
  {
    Fsm→xlated_names_[T_Enum::T_raw_exclam_] = "exclamation␣mark";
    Fsm→xlated_names_[T_Enum::T_raw_dbl_quote_] = "dbl␣quote";
    Fsm→xlated_names_[T_Enum::T_raw_no_sign_] = "no␣sign";
    Fsm→xlated_names_[T_Enum::T_raw_dollar_sign_] = "dollar␣sign";
    Fsm→xlated_names_[T_Enum::T_raw_percent_] = "percent";
    Fsm→xlated_names_[T_Enum::T_raw_ampersign_] = "ampersand";
    Fsm→xlated_names_[T_Enum::T_raw_right_quote_] = "rt␣quote";
    Fsm→xlated_names_[T_Enum::T_raw_open_bracket_] = "open␣bracket";
    Fsm→xlated_names_[T_Enum::T_raw_close_bracket_] = "close␣bracket";
    Fsm→xlated_names_[T_Enum::T_raw_asteric_] = "asterisk";
    Fsm→xlated_names_[T_Enum::T_raw_plus_] = "plus";
    Fsm→xlated_names_[T_Enum::T_raw_comma_] = "comma";
    Fsm→xlated_names_[T_Enum::T_raw_minus_] = "minus";
    Fsm→xlated_names_[T_Enum::T_raw_period_] = "period";
    Fsm→xlated_names_[T_Enum::T_raw_slash_] = "slash";
    Fsm→xlated_names_[T_Enum::T_raw_colon_] = "colon";
    Fsm→xlated_names_[T_Enum::T_raw_semi_colon_] = "semi␣colon";
    Fsm→xlated_names_[T_Enum::T_raw_less_than_] = "less␣than";
    Fsm→xlated_names_[T_Enum::T_raw_eq_] = "eq";
    Fsm→xlated_names_[T_Enum::T_raw_gt_than_] = "gt␣than";
    Fsm→xlated_names_[T_Enum::T_raw_question_mark_] = "question␣mark";
    Fsm→xlated_names_[T_Enum::T_raw_at_sign_] = "at␣sign";
    Fsm→xlated_names_[T_Enum::T_raw_open_sq_bracket_] = "open␣sq␣bracket";
    Fsm→xlated_names_[T_Enum::T_raw_back_slash_] = "back␣slash";
    Fsm→xlated_names_[T_Enum::T_raw_close_sq_bracket_] = "close␣sq␣bracket";
    Fsm→xlated_names_[T_Enum::T_raw_up_arrow_] = "up␣arrow";
    Fsm→xlated_names_[T_Enum::T_raw_under_score_] = "under␣score";
    Fsm→xlated_names_[T_Enum::T_raw_left_quote_] = "left␣quote";
    Fsm→xlated_names_[T_Enum::T_raw_open_brace_] = "open␣brace";
    Fsm→xlated_names_[T_Enum::T_raw_vertical_line_] = "vertical␣line";
    Fsm→xlated_names_[T_Enum::T_raw_close_brace_] = "close␣brace";
    Fsm→xlated_names_[T_Enum::T_raw_tilde_] = "tilde";
    Fsm→xlated_names_[T_Enum::T_raw_del_] = "del";
    Fsm→xlated_names_[T_Enum::T_LR1_invisible_shift_operator_] = "\\invisibleshift";
    Fsm→xlated_names_[T_Enum::T_LR1_all_shift_operator_] = "\\allshift";
    Fsm→xlated_names_[T_Enum::T_LR1_reduce_operator_] = "\\reduceoperator";
    Fsm→xlated_names_[T_Enum::T_LR1_fset_transience_operator_] = "\\transienceoperator";
    Fsm→xlated_names_[T_Enum::T_LR1_questionable_shift_operator_] = "\\questionableoperator";
  }
```

45. `MPOST_CWEB_EMIT_PREFIX_CODE`.
Output prelude statements for *cweave* and *mpost* files. *cweave* contains basic packages and macros whilest *mpost* contains its grammar drawing macros and basic array of variables.

46. Setup *mpost* macros and variable arrays.

```
⟨ setup mpost macros and variable arrays 46 ⟩ ≡
  KCHARP mp_prefix_file = "% file: ";
  Fsm→omp_file_ ≪ mp_prefix_file;
  KCHARP mp_prefix_file_value = "%s - grammar railroad diagrams for mpost program\n";
  int x = sprintf(Fsm→big_buf_, mp_prefix_file_value, Fsm→mp_filename_.c_str());
  Fsm→omp_file_.write(Fsm→big_buf_, x);
  Fsm→omp_file_ ≪ endl;
  KCHARP mp_prefix_date = "% date: ";
  Fsm→omp_file_ ≪ mp_prefix_date;
  KCHARP mp_prefix_date_value =
  "%s\n"
  "input \"/usr/local/yacco2/diagrams/o2diag.mp\"\n"
  "numeric no_of_rules,Box_solid,Box_dotted,Circle_solid,Circle_dotted;\n"
  "Box_solid:=1;Box_dotted:=2;Circle_solid:=3;Circle_dotted:=4;\n"
  "string rule_names[].literal;\n"
  "string rule_names[].vname;\n"
  "numeric rule_s_no_rhs[];\n"
  "string rhs_elems[][][].literal;\n"
  "string rhs_elems[][][].vname;\n"
  "numeric rhs_elems[][][].Drw_how;\n"
  "numeric rule_s_subrule_no_elems[][];\n";
  x = sprintf(Fsm→big_buf_, mp_prefix_date_value, Fsm→gened_date_time_.c_str());
  Fsm→omp_file_.write(Fsm→big_buf_, x);
```

This code is used in section 48.

47. Setup *cweave* macros and odds and ends.

```
⟨ setup cweave macros and odds and ends 47 ⟩ ≡
  KCHARP w_prefix_file = "% file: ";
  Fsm→ow_file_ ≪ w_prefix_file;
  KCHARP w_prefix_filename_value = "%s - cweb grammar\n";
  x = sprintf(Fsm→big_buf_, w_prefix_filename_value, Fsm→w_filename_.c_str());
  Fsm→ow_file_.write(Fsm→big_buf_, x);
  KCHARP w_prefix_date = "%Date: ";
  Fsm→ow_file_ ≪ w_prefix_date;
  KCHARP w_prefix_date_value = "%s\n";
  x = sprintf(Fsm→big_buf_, w_prefix_date_value, Fsm→gened_date_time_.c_str());
  Fsm→ow_file_.write(Fsm→big_buf_, x);
  KCHARP macros = "\\input%s \"supp-pdf\"\n"
  "\\input \"/usr/local/yacco2/diagrams/o2mac.tex\"\n";
  x = sprintf(Fsm→big_buf_, macros, " ");
  Fsm→ow_file_.write(Fsm→big_buf_, x);
```

This code is used in section 48.

48. Driver of MPOST_CWEB_EMIT_PREFIX_CODE.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_EMIT_PREFIX_CODE
  (NS_mpost_output::Cmpost_output * Fsm)
  {
    ⟨ setup mpost macros and variable arrays 46 ⟩;
    ⟨ setup cweave macros and odds and ends 47 ⟩;
  }
```

49. *MPOST_CWEB_gen_dimension_name*.
Generates the string literal for a dimension — rule no, subrule no, or element no. It's out string is used to build up the *mpost*'s variable object literal name. The base number is 27 but as the elements are 1 and greater the outputed string is 26^2 of 2 digits. So rule 27, subrule 3, element 5 produces 3 strings of "ba", "ac", and "ae".

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_gen_dimension_name
  (Cmpost_output * Fsm, std::string & Mp_obj_name, int Dimension)
  {
    int R, Q;
    R = Dimension % 27;
    Q = Dimension/27;
    ++Q;
    Mp_obj_name += Fsm→mp_dimension_[Q];
    Mp_obj_name += Fsm→mp_dimension_[R];
  }
```

50. *MPOST_CWEB_calc_mp_obj_name*.
Build up the 3 dimension name for *mpost*'s object name.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_calc_mp_obj_name
  (Cmpost_output * Fsm, std::string & Mp_obj_name, int Elem_no)
  {
    Fsm→MPOST_CWEB_gen_dimension_name(Fsm, Mp_obj_name, Fsm→rule_no_);
    Fsm→MPOST_CWEB_gen_dimension_name(Fsm, Mp_obj_name, Fsm→subrule_no_);
    Fsm→MPOST_CWEB_gen_dimension_name(Fsm, Mp_obj_name, Elem_no);
  }
```

51. *MPOST_CWEB_wrt_mp_rhs_elem*.
Output *mpost*'s statements for a subrule's element. Due to the *convertMPtoPDF* marco having problems with embedded space within literal names, i do a substitution on spaces for "." to keep the TEXflowing.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_mp_rhs_elem
  (Cmpost_output * Fsm, std::string & Elem_name, std::string & Drw_how)
  {
    std::string mp_xlate_name;
    prescan_mpname_for_cweb(Elem_name, mp_xlate_name);
    std::string mp_obj_name;
    KCHARP mp_subrule_elems_literal = "rhs_elems[%i][%i][%i].literal␣:=␣\"%s\";";
    int x = sprintf(Fsm→big_buf_, mp_subrule_elems_literal, Fsm→rule_no_, Fsm→subrule_no_, Fsm→elem_no_,
        mp_xlate_name.c_str());
    Fsm→omp_file_.write(Fsm→big_buf_, x);
    Fsm→omp_file_ ≪ endl;
    KCHARP mp_subrule_elems_RorT = "rhs_elems[%i][%i][%i].Drw_how␣:=␣%s;";
    x = sprintf(Fsm→big_buf_, mp_subrule_elems_RorT, Fsm→rule_no_, Fsm→subrule_no_, Fsm→elem_no_,
        Drw_how.c_str());
    Fsm→omp_file_.write(Fsm→big_buf_, x);
    Fsm→omp_file_ ≪ endl;
    KCHARP mp_subrule_elems_vname = "rhs_elems[%i][%i][%i].vname␣:=␣\"%s\";";
    Fsm→MPOST_CWEB_calc_mp_obj_name(Fsm, mp_obj_name, Fsm→elem_no_);
    x = sprintf(Fsm→big_buf_, mp_subrule_elems_vname, Fsm→rule_no_, Fsm→subrule_no_, Fsm→elem_no_,
        mp_obj_name.c_str());
    Fsm→omp_file_.write(Fsm→big_buf_, x);
    Fsm→omp_file_ ≪ endl;
  }
```

52. *MPOST_CWEB_gen_sr_elem_xrefs*.
To aid the grammar writer, all vocubulary symbols are cross referencesd using *cweave*'s directive. Originally each subrule was redrawn having the cross reference specific to the subrule. As there was too much spam being generated for aireous subrules with no syntax directed code, i decided to redraw subrules only having code. Consequently the cross references go against the rule definition rather than its specific subrules.

cweave doesn't xref the 1 characters so i bypass them as i do not want to change the characteristics of it... for now ...

53. Establish tree container with appropriate element filters.

⟨ establish tree container with appropriate element filters 53 ⟩ ≡

```
using namespace yacco2;
using namespace NS_yacco2_T_enum;
using namespace NS_yacco2_terminals;
KCHARP xref = "@.%s@>\n";
INT_SET_type filter;
filter.insert(T_Enum::T_refered_T_);
filter.insert(T_Enum::T_refered_rule_);
filter.insert(T_Enum::T_T_eosubrule_);
filter.insert(T_Enum::T_T_called_thread_eosubrule_);
filter.insert(T_Enum::T_T_null_call_thread_eosubrule_);
filter.insert(T_Enum::T_LR1_parallel_operator_);
filter.insert(T_Enum::T_LR1_fset_transience_operator_);
tok_can_ast_functor just_walk_functr;
ast_prefix_1forest element_walk(*Subrule_tree, &just_walk_functr, &filter, ACCEPT_FILTER);
tok_can < AST *> elements_can(element_walk);
```

This code is used in section 55.

54. Walk the container and produce the elements' cross references.

```
⟨ walk the container and produce the elements' cross references 54 ⟩ ≡
  for (int x(0); elements_can[x] ≠ yacco2::PTR_LR1_eog__; ++x) {
    ++elem_cnt;
    CAbs_lr1_sym * sym = elements_can[x];
    switch (sym→enumerated_id__) {
    case T_Enum::T_LR1_parallel_operator_:
      {
        int x = sprintf(Fsm→big_buf_, xref, "\\paralleloperator");
        Fsm→ow_file_.write(Fsm→big_buf_, x);
        break;
      }
    case T_Enum::T_LR1_fset_transience_operator_:
      {
        int x = sprintf(Fsm→big_buf_, xref, "\\transienceoperator");
        Fsm→ow_file_.write(Fsm→big_buf_, x);
        break;
      }
    case T_Enum::T_refered_T_:
      {
        refered_T* rt = (refered_T*)sym;
        Fsm→MPOST_CWEB_xref_refered_T(Fsm, rt);
        break;
      }
    case T_Enum::T_refered_rule_:
      {
        refered_rule* rr = (refered_rule*)sym;
        Fsm→MPOST_CWEB_xref_refered_rule(Fsm, rr);
        break;
      }
    case T_Enum::T_T_eosubrule_:
      {
        if (elem_cnt ≡ 1) {
          int x = sprintf(Fsm→big_buf_, xref, "\\emptyrule");
          Fsm→ow_file_.write(Fsm→big_buf_, x);
        }
        break;
      }
    case T_Enum::T_T_called_thread_eosubrule_:
      {
        T_called_thread_eosubrule* id = (T_called_thread_eosubrule*)sym;
        if (id→ns() ≠ 0) {      /* TRAshift call has no namespace :: qualifiers */
          th_name += id→ns()→identifier()→c_str();
          th_name += "::";
        }
        th_name += id→called_thread_name()→identifier()→c_str();
        string name_with_bkslash;
        int len = th_name.length();
        for (int x = 0; x < len; ++x) {
          char c = th_name[x];
```

```
        if (c ≡ '_') {
          name_with_bkslash += '\\';
        }
        name_with_bkslash += c;
      }
      int x = sprintf(Fsm→big_buf_, xref, name_with_bkslash.c_str());
      Fsm→ow_file_.write(Fsm→big_buf_, x);
      break;
    }
  case T_Enum :: T_T_null_call_thread_eosubrule_:
    {
      int x = sprintf(Fsm→big_buf_, xref, "NULL");
      Fsm→ow_file_.write(Fsm→big_buf_, x);
      break;
    }
  }
}
```

This code is used in section 55.

55. Driver of *MPOST_CWEB_gen_sr_elem_xrefs*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output :: Cmpost_output :: MPOST_CWEB_gen_sr_elem_xrefs
  (Cmpost_output * Fsm, AST * Subrule_tree)
  {
    ⟨ establish tree container with appropriate element filters 53 ⟩;
    int elem_cnt(0);
    string th_name;
    ⟨ walk the container and produce the elements' cross references 54 ⟩;
  }
```

56. *MPOST_CWEB_xlated_symbol*.

```
⟨ accrue source for emit 8 ⟩ +≡
  extern int MPOST_CWEB_xlated_symbol(AST * Sym_t, char *Xlated_sym)
  {
    CAbs_lr1_sym * Sym = AST::content(*Sym_t);
    int Enum = Sym→enumerated_id( );
    Xlated_sym[0] = (char) 0;
    const char *xsym = "%s";
    switch (Enum) {
    case T_Enum::T_LR1_parallel_operator_:
      {
        sprintf(Xlated_sym, xsym, "\\parallelоperator");
        return 1;
      }
    case T_Enum::T_LR1_fset_transience_operator_:
      {
        sprintf(Xlated_sym, xsym, "\\transienceoperator");
        return 1;
      }
    case T_Enum::T_refered_T_:
      {
        refered_T* rt = (refered_T*)Sym;
        XLATE_SYMBOLS_FOR_cweave(rt→its_t_def( )→t_name( )→c_str( ), Xlated_sym);
        return rt→element_pos( );
      }
    case T_Enum::T_refered_rule_:
      {
        refered_rule* rr = (refered_rule*)Sym;
        XLATE_SYMBOLS_FOR_cweave(rr→its_rule_def( )→rule_name( )→c_str( ), Xlated_sym);
        return rr→element_pos( );
      }
    case T_Enum::T_T_eosubrule_:
      {    /* if parallel expr then use eosurule of thread */
        T_eosubrule* eos = (T_eosubrule*)Sym;
        AST * prev_t = Sym_t→pr_;
        if (eos→element_pos( ) ≡ 1) {
          sprintf(Xlated_sym, xsym, "\\emptyrule");
          return eos→element_pos( );
        }
        else {
          sprintf(Xlated_sym, xsym, "");
        }
        CAbs_lr1_sym * sym = AST::content(*prev_t);
        switch (sym→enumerated_id( )) {
        case T_Enum::T_T_called_thread_eosubrule_:
          {
            return eos→element_pos( ) − 1;
          }
        case T_Enum::T_T_null_call_thread_eosubrule_:
          {
```

```
            return eos→element_pos( ) − 1;
          }
        default:
          {
            return eos→element_pos( );
          }
        }
      }
    case T_Enum :: T_T_called_thread_eosubrule_:
      {
        T_called_thread_eosubrule* id = (T_called_thread_eosubrule*)Sym;
        string th_name;
        if (id→ns( ) ≠ 0) {      /* TRAshift call has no namespace :: qualifiers */
          th_name += id→ns( )→identifier( )→c_str( );
          th_name += "::";
        }
        th_name += id→called_thread_name( )→identifier( )→c_str( );
        XLATE_SYMBOLS_FOR_cweave(th_name.c_str( ), Xlated_sym);
        return id→element_pos( );
      }
    case T_Enum :: T_T_null_call_thread_eosubrule_:
      {
        T_null_call_thread_eosubrule* id = (T_null_call_thread_eosubrule*)Sym;
        sprintf(Xlated_sym, xsym, "NULL");
        return id→element_pos( );
      }
    }
    return 0;      /* fake it: cuz apple's latest compiler: symantic error-never reached! */
}
```

57. *MPOST_CWEB_crt_rhs_sym_str* **- follow set contributing symbols.**
Generate the state's subrule symbols in for cweave to digest. If the first symbol in the string is a rule, this rule's follow set string gets highlighted by underlining the symbols that contribute to it. It will transient walk though these follow symbols if they are epsilonabe rules. This underline string of symbols will use the "..."' symbol to indicate more symbols contribute but due to the report's space constrains are not reported.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_crt_rhs_sym_str
  (state_element * se, std::string * Xlated_str){ char a[BUFFER_SIZE];
      char cur_sym[Max_cweb_item_size];
      char nxtsym_1[Max_cweb_item_size];
      char nxtsym_2[Max_cweb_item_size];
      KCHARP thread_expr_string_template = "{%s}␣{%s}␣{%s}";      /* 3 symbols: current and 2 la */
      KCHARP underline_symbol = "{$\\underline{%s}$}␣\n";
      KCHARP underline_symbol_wepsilon = "{$\\underline{%s^{\\emptyrule}}$}␣\n";
      KCHARP not_underline_symbol = "{%s}␣\n";
      state_element * nxt1_se = 0;
      state_element * nxt2_se = 0;
      state_element * nxt3_se = 0;
      int pp = MPOST_CWEB_xlated_symbol(se→sr_element_, cur_sym);
      if (      /* chained or thread call expression */
      (se→its_enum_id_ ≡ T_Enum::T_LR1_parallel_operator_) ∨ (se→its_enum_id_ ≡
            T_Enum::T_LR1_fset_transience_operator_)) {
        nxt1_se = se→next_state_element_;
        AST * nxt2_se = se→sr_element_→rt_→rt_;      /* thread eos for called thd nm */
        pp = MPOST_CWEB_xlated_symbol(nxt1_se→sr_element_, nxtsym_1);
        pp = MPOST_CWEB_xlated_symbol(nxt2_se, nxtsym_2);
        sprintf(a, thread_expr_string_template, cur_sym, nxtsym_1, nxtsym_2);
        (*Xlated_str) += a;
        return;
      }
      sprintf(a, not_underline_symbol, cur_sym);
      (*Xlated_str) += a;
      if (se→sr_def_element_→enumerated_id__ ≠ T_Enum::T_rule_def_) {      /* T */
        return;
      }
      nxt1_se = se→next_state_element_;
      if (nxt1_se ≡ 0) return;
      if (nxt1_se→its_enum_id_ ≡ T_Enum::T_T_eosubrule_) return;
      pp = MPOST_CWEB_xlated_symbol(nxt1_se→sr_element_, nxtsym_1);
      if (nxt1_se→sr_def_element_→enumerated_id__ ≠ T_Enum::T_rule_def_) {
        sprintf(a, underline_symbol, nxtsym_1);
      }
      else { rule_def * rd = ( rule_def * ) nxt1_se→sr_def_element_;
      if (rd→epsilon( ) ≡ YES) {
        sprintf(a, underline_symbol_wepsilon, nxtsym_1);
        nxt2_se = nxt1_se→next_state_element_;
      }
      else {
        sprintf(a, underline_symbol, nxtsym_1);
        nxt2_se = 0;
      }
```

```
} (*Xlated_str) += a; if (nxt2_se ≠ 0) {      /* 2nd follow set sym due to epsilon rule */
if (nxt2_se→its_enum_id_ ≡ T_Enum::T_T_eosubrule_) return;
pp = MPOST_CWEB_xlated_symbol(nxt2_se→sr_element_, nxtsym_2);
if (nxt2_se→sr_def_element_→enumerated_id__ ≠ T_Enum::T_rule_def_) {      /* T */
  sprintf(a, underline_symbol, nxtsym_2);
}
else { rule_def * rd = ( rule_def * ) nxt2_se→sr_def_element_;
if (rd→epsilon( ) ≡ YES) {
  sprintf(a, underline_symbol_wepsilon, nxtsym_2);
  nxt3_se = nxt1_se→next_state_element_;
  if (nxt3_se→its_enum_id_ ≡ T_Enum::T_T_eosubrule_) {
    nxt3_se = 0;
  }
}
else {
  sprintf(a, underline_symbol, nxtsym_2);
  nxt3_se = 0;
}
} (*Xlated_str) += a; }
if (nxt3_se ≠ 0) {      /* ... all others due to 2nd epsilon follow symbol rule */
  sprintf(a, not_underline_symbol, "...");
  (*Xlated_str) += a;
}
}
```

58. *MPOST_CWEB_woutput_sr_sdcode*.
cweave's statements for a subrule having syntax directed code. Note: the use of a grammar to generate it.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_woutput_sr_sdcode
  (Cmpost_output * Fsm, T_subrule_def * Subrule_def)
  {
    SDC_MAP_type * sr_dirs = Subrule_def→subrule_directives( );
    SDC_MAP_ITER_type i = sr_dirs→begin( );
    SDC_MAP_ITER_type ie = sr_dirs→end( );
    TOKEN_GAGGLE dirs_tokens;
    for ( ; i ≠ ie; ++i) {
      CAbs_lr1_sym * sym = i→second;
      dirs_tokens.push_back(*sym);
    }
    using namespace NS_cweave_sdc;
    Ccweave_sdc sdc_fsm;
    sdc_fsm.initialize(&Fsm→ow_file_, Fsm→subrule_def_, Fsm→subrule_no_);
    Parser sdc_emit(sdc_fsm, &dirs_tokens, 0);
    sdc_emit.parse( );
  }
```

59. *MPOST_CWEB_wrt_fsm*.
cweave's *fsm* part of the grammar. Not much but there's that grammar use to generate the output.

60. Output *fsm*'s class section.

⟨ output *fsm*'s class section 60 ⟩ ≡

```
char xa[Max_cweb_item_size];
XLATE_SYMBOLS_FOR_cweave(Fsm_phrase→fsm_class_phrase( )→identifier( )→identifier( )→c_str( ), xa);
KCHARP fsm_class = "@*2␣Fsm␣%s␣class.\n";
int x = sprintf(Fsm→big_buf_, fsm_class, xa);
Fsm→ow_file_.write(Fsm→big_buf_, x);
Fsm→ow_file_ ≪ endl;
```

This code is used in section 62.

61. Ready those *fsm* syntax directed code jelly beans for consumption.

⟨ ready those *fsm* sdcode class jelly beans for consumption 61 ⟩ ≡

```
T_fsm_class_phrase * fcp = Fsm_phrase→fsm_class_phrase( );
SDC_MAP_type * sr_dirs = fcp→directives_map( );
SDC_MAP_ITER_type i = sr_dirs→begin( );
SDC_MAP_ITER_type ie = sr_dirs→end( );
TOKEN_GAGGLE dirs_tokens;
for ( ; i ≠ ie; ++i) {
  CAbs_lr1_sym * sym = i→second;
  dirs_tokens.push_back(*sym);
}
```

This code is used in section 62.

62. Driver of *MPOST_CWEB_wrt_fs*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_fsm
  (Cmpost_output * Fsm, T_fsm_phrase * Fsm_phrase)
  {
    KCHARP banner = "\\GRAMMARtitle{%s}\n{%s}{%s}%\n""{%s}{%s}%\n""{%s}\n";
    KCHARP banner_of_thread = "\\THREADtitle{%s}%\n""{%s}{%s}%\n""{%s}{%s}{%s}{%i}%\n\
        ""{%s}\n";
    char fname[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Fsm→grammar_filename_prefix_.c_str(), fname);
    char ns_name[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Fsm_phrase→namespace_id()→identifier()→c_str(), ns_name);
    char fsm_name[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Fsm_phrase→fsm_id()→c_string()→c_str(), fsm_name);
    char k_cweb[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(Fsm_phrase→comment()→c_string()→c_str(), k_cweb);
    if (O2_PP_PHASE ≠ 0) {
      T_parallel_parser_phrase * la_ph = O2_PP_PHASE;
      T_parallel_la_boundary * la = la_ph→la_bndry();      /* build srce expr */
      char expr_prescan_cweb[Max_cweb_item_size];
      int x = sprintf(Fsm→big_buf_, banner_of_thread, fname, fsm_name, ns_name,
          Fsm_phrase→version()→c_string()→c_str(), Fsm_phrase→debug()→c_string()→c_str(), k_cweb,
          la→la_first_set()→size()      /*,expr_prescan_cweb */
      , la→cweb_la_srce_expr()→c_str());
      Fsm→ow_file_.write(Fsm→big_buf_, x);
      Fsm→ow_file_ ≪ endl;
    }
    else {
      int x = sprintf(Fsm→big_buf_, banner, fname, fsm_name, ns_name,
          Fsm_phrase→version()→c_string()→c_str(), Fsm_phrase→debug()→c_string()→c_str(), k_cweb);
      Fsm→ow_file_.write(Fsm→big_buf_, x);
      Fsm→ow_file_ ≪ endl;
    }
    if (Fsm_phrase→cweb_marker() ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_file_, Fsm_phrase→cweb_marker());
    }
    ⟨ output fsm's class section 60 ⟩;
    ⟨ ready those fsm sdcode class jelly beans for consumption 61 ⟩;
    using namespace NS_cweave_fsm_sdc;
    Ccweave_fsm_sdc sdc_fsm;
    sdc_fsm.initialize(&Fsm→ow_file_, Fsm_phrase);
    Parser sdc_emit(sdc_fsm, &dirs_tokens, 0);
    sdc_emit.parse();
  }
```

63. *MPOST_CWEB_wrt_T*.

cweave's T portion of grammar vocabulary.

64. Output T macros and files.

⟨ output T macros and files 64 ⟩ ≡

```
output_cweb_macros_and_banner(Fsm, Fsm→ow_t_file_, "Terminal␣Vocabulary",
    Fsm→gened_date_time_.c_str( ), T_phrase→filename_id( )→identifier( ),
    T_phrase→namespace_id( )→identifier( ), T_phrase→alphabet( )→size( ));
```

This code is used in section 70.

65. hrule to end T entry.

⟨ rule-it 65 ⟩ ≡

```
KCHARP rule_it = "\\fbreak%s\n"
"\\hrule\n";
x = sprintf(Fsm→big_buf_, rule_it, "␣");
Fsm→ow_t_file_.write(Fsm→big_buf_, x);
```

This code is used in section 66.

66. Output T's entry.

For now i hardwire the enumeration label here. It will be deposited in the enumeration grammar for me to fetch.

⟨ output T's entry 66 ⟩ ≡

```
char ab[ ] = "Y";
if (tdef→autoabort( ) ≡ true) ab[0] = 'Y';
else ab[0] = 'N';
char ad[ ] = "Y";
if (tdef→autodelete( ) ≡ true) ad[0] = 'Y';
else ad[0] = 'N';
char name[Max_cweb_item_size];
XLATE_SYMBOLS_FOR_cweave(tdef→t_name( )→c_str( ), name);
char class_name[Max_cweb_item_size];
XLATE_SYMBOLS_FOR_cweave(tdef→classsym( )→c_str( ), class_name);
const char *enum_name_for_format = "T\\_%s\\_";
char a[SMALL_BUFFER_4K];
sprintf(a, enum_name_for_format, class_name);
KCHARP t_entry = "@*2␣{\\bf␣%s}.\\fbreak\n"
"Enum:␣%s\n\\fbreak\n"
"\\line{Class:␣%s␣\\hfil␣AB:␣%s␣\\hfil␣AD:␣%s}\n";
int x = sprintf(Fsm→big_buf_, t_entry, name, a, class_name, ab, ad);
Fsm→ow_t_file_.write(Fsm→big_buf_, x);
Fsm→ow_t_file_ ≪ endl;
if (tdef→cweb_marker( ) ≠ 0) {
  WRT_CWEB_MARKER(&Fsm→ow_t_file_, tdef→cweb_marker( ));
}
⟨ rule-it 65 ⟩;
```

This code is used in section 70.

67. Output prefix code if present.

```
⟨ output prefix code if present 67 ⟩ ≡
  if (T_phrase→terminals_refs_code() ≠ 0) {
    KCHARP tref = "@*2␣{\\bf␣Terminals-refs}␣directive.\\fbreak\n";
    Fsm→ow_t_file_ ≪ tref;
    T_syntax_code * stc = T_phrase→terminals_refs_code();
    if (stc→cweb_marker() ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_t_file_, stc→cweb_marker());
    }
    KCHARP tref_code =
    "@<terminals-refs␣directive@>=\n""%s"
    "\n";
    int x = sprintf(Fsm→big_buf_, tref_code, T_phrase→terminals_refs_code()→syntax_code()→c_str());
    Fsm→ow_t_file_.write(Fsm→big_buf_, x);
    Fsm→ow_t_file_ ≪ endl;
  }
```

This code is used in section 70.

68. Output suffix code if present.

```
⟨ output suffix code if present 68 ⟩ ≡
  if (T_phrase→terminals_sufx_code() ≠ 0) {
    KCHARP tsuf = "@*1␣{\\bf␣Terminals-sufx}␣directive.\\fbreak\n";
    Fsm→ow_t_file_ ≪ tsuf;
    T_syntax_code * stc = T_phrase→terminals_sufx_code();
    if (stc→cweb_marker() ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_t_file_, stc→cweb_marker());
    }
    KCHARP tsuf_code = "@<terminals-sufx␣directive@>=\n"
    "%s"
    "\n";
    int x = sprintf(Fsm→big_buf_, tsuf_code, T_phrase→terminals_sufx_code()→syntax_code()→c_str());
    Fsm→ow_t_file_.write(Fsm→big_buf_, x);
    Fsm→ow_t_file_ ≪ endl;
  }
```

This code is used in section 70.

69. Get those T syntax directed code jelly beans ready for consumption.

```
⟨ get those T sdcode class jelly beans ready for consumption 69 ⟩ ≡
  SDC_MAP_type * sr_dirs = tdef→directives_map();
  SDC_MAP_ITER_type i = sr_dirs→begin();
  SDC_MAP_ITER_type ie = sr_dirs→end();
  TOKEN_GAGGLE dirs_tokens;
  for ( ; i ≠ ie; ++i) {
    CAbs_lr1_sym * sym = i→second;
    dirs_tokens.push_back(*sym);
  }
```

This code is used in sections 70, 75, and 81.

70. Driver of *MPOST_CWEB_wrt_T*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_T
  (Cmpost_output * Fsm, T_terminals_phrase * T_phrase)
  {
    ⟨ output T macros and files 64 ⟩;
    if (T_phrase→cweb_marker() ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_t_file_, T_phrase→cweb_marker());
    }
    ⟨ output prefix code if present 67 ⟩;
    T_DEF_MAP_ITER_type a = T_phrase→alphabet()→begin();
    T_DEF_MAP_ITER_type ae = T_phrase→alphabet()→end();
    char term_name[Max_cweb_item_size];
    for (; a ≠ ae; ++a) {
      T_terminal_def * tdef = a→second;
      ⟨ output T's entry 66 ⟩;
      ⟨ get those T sdcode class jelly beans ready for consumption 69 ⟩;
      using namespace NS_cweave_T_sdc;
      Ccweave_T_sdc sdc_fsm;
      XLATE_SYMBOLS_FOR_cweave(tdef→t_name()→c_str(), term_name);
      sdc_fsm.initialize(&Fsm→ow_t_file_, term_name);
      Parser sdc_emit(sdc_fsm, &dirs_tokens, 0);
      sdc_emit.parse();
    }
    ⟨ output suffix code if present 68 ⟩;
    Fsm→ow_t_file_ ≪ "@**␣Index.\n";
  }
```

71. *MPOST_CWEB_wrt_Err*.
cweave's *Err* portion of grammar vocabulary.

72. Output *err* macros and files.

```
⟨ output err macros and files 72 ⟩ ≡
  output_cweb_macros_and_banner(Fsm, Fsm→ow_err_file_, "Error␣Vocabulary",
      Fsm→gened_date_time_.c_str(), Err_phrase→filename_id()→identifier(),
      Err_phrase→namespace_id()→identifier(), Err_phrase→alphabet()→size());
```

This code is used in section 75.

73. Err hrule to end T entry.

```
⟨ err rule-it 73 ⟩ ≡
  KCHAR rule_it = "\\fbreak%s\n"
  "\\hrule\n";
  x = sprintf(Fsm→big_buf_, rule_it, "␣");
  Fsm→ow_err_file_.write(Fsm→big_buf_, x);
```

This code is used in section 74.

74. Output *err*'s entry.
Due to my verbosity and the black slug outputted from *cweave* / TEX, i removed the emitting of the enumeration symbol as part of the index. For now i have not tested against the enumeration symbol. i explicitly reference it by its key from a returned thread call or i use the `|+|` to catch it. Time may squelch my decision.

```
⟨ output err's entry 74 ⟩ ≡
  char ab[ ] = "Y";
  if (tdef→autoabort( ) ≡ true) ab[0] = 'Y';
  else ab[0] = 'N';
  char ad[ ] = "Y";
  if (tdef→autodelete( ) ≡ true) ad[0] = 'Y';
  else ad[0] = 'N';
  char name[Max_cweb_item_size];
  XLATE_SYMBOLS_FOR_cweave(tdef→t_name( )→c_str( ), name);
  char class_name[Max_cweb_item_size];
  XLATE_SYMBOLS_FOR_cweave(tdef→classsym( )→c_str( ), class_name);
  const char *enum_name_for_format = "T\\_%s\\_";
  char enum_name[Max_cweb_item_size];
  XLATE_SYMBOLS_FOR_cweave(tdef→classsym( )→c_str( ), enum_name);
  char a[SMALL_BUFFER_4K];
  sprintf(a, enum_name_for_format, enum_name);
  KCHARP t_entry = "@*2␣{\\bf␣%s}.\\fbreak\n"
  "Enum:␣%s\n\\fbreak\n"
  "\\line{Class:␣%s␣\\hfil␣AB:␣%s␣\\hfil␣AD:␣%s}\n";
  int x = sprintf(Fsm→big_buf_, t_entry, name, a, class_name, ab, ad);
  Fsm→ow_err_file_.write(Fsm→big_buf_, x);
  Fsm→ow_err_file_ ≪ endl;
  if (tdef→cweb_marker( ) ≠ 0) {
    WRT_CWEB_MARKER(&Fsm→ow_err_file_, tdef→cweb_marker( ));
  }
  ⟨ err rule-it 73 ⟩;
```

This code is used in section 75.

75. Driver of *MPOST_CWEB_wrt_Err*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_Err
  (Cmpost_output * Fsm, T_error_symbols_phrase * Err_phrase)
  {
    ⟨ output err macros and files 72 ⟩;
    if (Err_phrase→cweb_marker() ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_err_file_, Err_phrase→cweb_marker());
    }
    T_DEF_MAP_ITER_type a = Err_phrase→alphabet()→begin();
    T_DEF_MAP_ITER_type ae = Err_phrase→alphabet()→end();
    char term_name[Max_cweb_item_size];
    for (; a ≠ ae; ++a) {
      T_terminal_def * tdef = a→second;
      ⟨ output err's entry 74 ⟩;
      ⟨ get those T sdcode class jelly beans ready for consumption 69 ⟩;
      using namespace NS_cweave_T_sdc;
      Ccweave_T_sdc sdc_fsm;
      XLATE_SYMBOLS_FOR_cweave(tdef→t_name()→c_str(), term_name);
      sdc_fsm.initialize(&Fsm→ow_err_file_, term_name);
      Parser sdc_emit(sdc_fsm, &dirs_tokens, 0);
      sdc_emit.parse();
    }
    Fsm→ow_err_file_ ≪ "@**␣Index.\n";
  }
```

76. *MPOST_CWEB_wrt_lrk*.
cweave's *lrk* portion of grammar vocabulary.

77. Output *lrk* macros and files.

```
⟨ output lrk macros and files 77 ⟩ ≡
  output_cweb_macros_and_banner(Fsm, Fsm→ow_lrk_file_, "Lr␣K␣Vocabulary",
      Fsm→gened_date_time_.c_str(), Lrk_phrase→filename_id()→identifier(),
      Lrk_phrase→namespace_id()→identifier(), Lrk_phrase→alphabet()→size());
```

This code is used in section 81.

78. Lrk hrule to end T entry.

```
⟨ lrk rule-it 78 ⟩ ≡
  KCHARP rule_it = "\\fbreak%s\n"
  "\\hrule\n";
  x = sprintf(Fsm→big_buf_, rule_it, "␣");
  Fsm→ow_lrk_file_.write(Fsm→big_buf_, x);
```

This code is used in section 79.

79. Output *lrk*'s entry.
For now i hardwire the enumeration label here. It will be deposited in the enumeration grammar for me to fetch.

Note: why the problem with *reduce***operator**? The index part of pdftex honks when it uses the TEXmacro *reduce_operator* so i renamed it to *rrrr* to give the proper effect `|r|`.

```
⟨ output lrk's entry 79 ⟩ ≡
  char ab[ ] = "Y";
  if (tdef→autoabort( ) ≡ true) ab[0] = 'Y';
  else ab[0] = 'N';
  char ad[ ] = "Y";
  if (tdef→autodelete( ) ≡ true) ad[0] = 'Y';
  else ad[0] = 'N';
  switch (tdef→enumerated_id__) {
  case T_Enum :: T_LR1_parallel_operator_:
    {
      strcpy(name, "\\paralleloperator");
      strcpy(term_name, "|parallel_operator|");
      break;
    }
  case T_Enum :: T_LR1_all_shift_operator_:
    {
      strcpy(name, "\\allshift");
      strcpy(term_name, "|all_shift|");
      break;
    }
  case T_Enum :: T_LR1_invisible_shift_operator_:
    {
      strcpy(name, "\\invisibleshift");
      strcpy(term_name, "|invisible_shift|");
      break;
    }
  case T_Enum :: T_LR1_fset_transience_operator_:
    {
      strcpy(name, "\\transienceoperator");
      strcpy(term_name, "|transience_operator|");
      break;
    }
  case T_Enum :: T_LR1_questionable_shift_operator_:
    {
      strcpy(name, "\\questionableoperator");
      strcpy(term_name, "|questionable_operator|");
      break;
    }
  default:
    {
      XLATE_SYMBOLS_FOR_cweave(tdef→t_name( )→c_str( ), name);
      XLATE_SYMBOLS_FOR_cweave(tdef→t_name( )→c_str( ), term_name);
      break;
    }
  }
  const char *enum_class_name_format = "T\\_%s\\_";
```

```
char class_name[Max_cweb_item_size];
XLATE_SYMBOLS_FOR_cweave(tdef→classsym( )→c_str( ), class_name);
char a[SMALL_BUFFER_4K];
sprintf(a, enum_class_name_format, class_name);
KCHARP t_entry = "@*2␣{\\bf␣%s}.\\fbreak\n"
"Enum:␣%s\n\\fbreak\n"
"\\line{Class:␣%s␣\\hfil␣AB:␣%s␣\\hfil␣AD:␣%s}\n";
int x = sprintf(Fsm→big_buf_, t_entry, name, a, class_name, ab, ad);
Fsm→ow_lrk_file_.write(Fsm→big_buf_, x);
Fsm→ow_lrk_file_ ≪ endl;
if (tdef→cweb_marker( ) ≠ 0) {
  WRT_CWEB_MARKER(&Fsm→ow_lrk_file_, tdef→cweb_marker( ));
}
⟨ lrk rule-it 78 ⟩;
```

This code is used in section 81.

80. Output lrk suffix code if present.

```
⟨ output lrk suffix code if present 80 ⟩ ≡
  if (Lrk_phrase→lrk_sufx_code( ) ≠ 0) {
    KCHARP tsuf = "@*1␣{\\bf␣lrk-sufx}␣directive.\\fbreak\n";
    Fsm→ow_lrk_file_ ≪ tsuf;
    T_syntax_code * stc = Lrk_phrase→lrk_sufx_code( );
    if (stc→cweb_marker( ) ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_lrk_file_, stc→cweb_marker( ));
    }
    KCHARP tsuf_code = "@<lrk-sufx␣directive@>=\n""%s""\n";
    int x = sprintf(Fsm→big_buf_, tsuf_code, Lrk_phrase→lrk_sufx_code( )→syntax_code( )→c_str( ));
    Fsm→ow_lrk_file_.write(Fsm→big_buf_, x);
    Fsm→ow_lrk_file_ ≪ endl;
  }
```

This code is used in section 81.

81. Driver of *MPOST_CWEB_wrt_Lrk*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_Lrk
  (Cmpost_output * Fsm, T_lr1_k_phrase * Lrk_phrase)
  {
    ⟨ output lrk macros and files 77 ⟩;
    if (Lrk_phrase→cweb_marker( ) ≠ 0) {
      WRT_CWEB_MARKER(&Fsm→ow_lrk_file_, Lrk_phrase→cweb_marker( ));
    }
    char term_name[Max_cweb_item_size];
    char name[Max_cweb_item_size];
    T_DEF_MAP_ITER_type a = Lrk_phrase→alphabet( )→begin( );
    T_DEF_MAP_ITER_type ae = Lrk_phrase→alphabet( )→end( );
    for ( ; a ≠ ae; ++a) {
      T_terminal_def * tdef = a→second;
      ⟨ output lrk's entry 79 ⟩;
      ⟨ get those T sdcode class jelly beans ready for consumption 69 ⟩;
      using namespace NS_cweave_T_sdc;
      Ccweave_T_sdc sdc_fsm;
      sdc_fsm.initialize(&Fsm→ow_lrk_file_, term_name);
      Parser sdc_emit(sdc_fsm, &dirs_tokens, 0);
      sdc_emit.parse( );
    }
    ⟨ output lrk suffix code if present 80 ⟩;
    Fsm→ow_lrk_file_ ≪ "@**␣Index.\n";
  }
```

82. *MPOST_CWEB_wrt_rule_s_lhs_sdc*.
cweave's statements for a rule's "lhs" syntax directed code. Go to it grammar.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_wrt_rule_s_lhs_sdc
  (Cmpost_output * Fsm, rule_def * Rule_def)
  {
    T_rule_lhs_phrase * rlp = Rule_def→rule_lhs( );
    if (rlp ≡ 0) return;
    SDC_MAP_type * sr_dirs = rlp→lhs_directives_map( );
    SDC_MAP_ITER_type i = sr_dirs→begin( );
    SDC_MAP_ITER_type ie = sr_dirs→end( );
    TOKEN_GAGGLE dirs_tokens;
    for ( ; i ≠ ie; ++i) {
      CAbs_lr1_sym * sym = i→second;
      dirs_tokens.push_back(*sym);
    }
    T_parallel_monitor_phrase * ppm = Rule_def→parallel_mntr( );
    if (ppm ≠ 0) {
      sr_dirs = ppm→mntr_directives_map( );
      i = sr_dirs→begin( );
      ie = sr_dirs→end( );
      for ( ; i ≠ ie; ++i) {
        CAbs_lr1_sym * sym = i→second;
        dirs_tokens.push_back(*sym);
      }
    }
    using namespace NS_cweave_lhs_sdc;
    Ccweave_lhs_sdc lhs_sdc_fsm;
    lhs_sdc_fsm.initialize(&Fsm→ow_file_, Rule_def→rule_name( )→c_str( ));
    Parser lhs_sdc_emit(lhs_sdc_fsm, &dirs_tokens, 0);
    lhs_sdc_emit.parse( );
  }
```

83. *MPOST_CWEB_should_subrule_be_printed*.
Check whether the subrule should be *cweave*ed. If there is no syntax directed code, then bypass diagramming it as its rule already does the big picture — forest.

```
⟨ accrue source for emit 8 ⟩ +≡
  bool NS_mpost_output::Cmpost_output::MPOST_CWEB_should_subrule_be_printed
  (Cmpost_output * Fsm, T_subrule_def * Subrule_def)
  {
    if (Subrule_def→cweb_marker( ) ≠ 0) return true;
    SDC_MAP_type * sr_dirs = Subrule_def→subrule_directives( );
    if (sr_dirs→empty( ) ≡ true) return false;
    return true;
  }
```

84. Deal with T's classifications. To do: emit single character references; adjust *cweave*?;

```
⟨ deal with T's classifications 84 ⟩ ≡
  switch (td→classification()) {
  case T_terminal_def::err:
    {
      int x = sprintf(Fsm→big_buf_, xref, name);
      Fsm→ow_file_.write(Fsm→big_buf_, x);
      break;
    }
  case T_terminal_def::rc:
    {
      break;
      INT_STR_MAP_ITER_type i = Fsm→xlated_names_.find(td→enumerated_id__);
      if (i ≠ Fsm→xlated_names_.end()) {
        int x = sprintf(Fsm→big_buf_, xref, i→second.c_str());
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      else {
        int x = sprintf(Fsm→big_buf_, xref, name);
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      break;
    }
  case T_terminal_def::lrk:
    {
      INT_STR_MAP_ITER_type i = Fsm→xlated_names_.find(td→enumerated_id__);
      if (i ≠ Fsm→xlated_names_.end()) {
        int x = sprintf(Fsm→big_buf_, xref, i→second.c_str());
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      else {
        int x = sprintf(Fsm→big_buf_, xref, name);
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      break;
    }
  case T_terminal_def::t:
    {
      INT_STR_MAP_ITER_type i = Fsm→xlated_names_.find(td→enumerated_id__);
      if (i ≠ Fsm→xlated_names_.end()) {
        int x = sprintf(Fsm→big_buf_, xref, i→second.c_str());
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      else {
        int x = sprintf(Fsm→big_buf_, xref, name);
        Fsm→ow_file_.write(Fsm→big_buf_, x);
      }
      break;
    }
  }
```

This code is used in section 85.

85. *MPOST_CWEB_xref_refered_T*.
Don't emit single characters as *pdftex* honks in the index section as this was a decision to not clutter temporary variables. Until i get more intimate with its code i check to this limitation and my ignorance.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_xref_refered_T
  (Cmpost_output * Fsm, refered_T * R_T)
  {
    T_in_stbl * t = R_T→t_in_stbl( );
    T_terminal_def * td = t→t_def( );
    KCHARP xref = "@.%s@>\n";
    char name[Max_cweb_item_size];
    XLATE_SYMBOLS_FOR_cweave(td→t_name( )→c_str( ), name);
    ⟨ deal with T's classifications 84 ⟩;
  }
```

86. *MPOST_CWEB_xref_refered_rule*.
Watch for underscore in rule's name. So makesure that *cweave*'s directive backslash underscore is part of the name.

```
⟨ accrue source for emit 8 ⟩ +≡
  void NS_mpost_output::Cmpost_output::MPOST_CWEB_xref_refered_rule
  (Cmpost_output * Fsm, refered_rule * R_rule)
  {
    rule_in_stbl * r = R_rule→Rule_in_stbl( );
    rule_def * rd = r→r_def( );
    KCHARP xref = "@.%s@>\n";
    string name_with_bkslash;
    int len = rd→rule_name( )→length( );
    for (int x = 0; x < len; ++x) {
      char c = (*rd→rule_name( ))[x];
      if (c ≡ '_') {
        name_with_bkslash += '\\';
      }
      name_with_bkslash += c;
    }
    int x = sprintf(Fsm→big_buf_, xref, name_with_bkslash.c_str( ));
    Fsm→ow_file_.write(Fsm→big_buf_, x);
  }
```

87. Grammar's "Called threads" First Set Calculation.
What is it? Similar to a terminal first set definition, it is the list of "called thread" expressions in the "Start Rule" arrived at from rule closures. Why is it needed? It is the content of the "list-of-transitive-threads" construct emitted in the grammar's "fsc" file for O_2^{linker} to resolve into T first sets per thread. These first sets are what determines whether a thread should be called.

The Algorithm:
add start rule's subrules to the element space
walk the element space
Element assessment:
a) eosubrule: advance to next point in space
b) referenced rule:
determine if referenced rule already contributed
no — add rule to contributed closure rules set
add its subrules to the element space
is "referenced rule" epsilonable?
yes — overlay current element with its right neighbour
no — advance to next subrule in element space
c) |||rtn-T NSa::THa type expression?
no — advance to next subrule point in space
yes — add called thread to "called thread first set'
advance to next subrule in element space
until all entries in element space stomped on

The algorithm uses a 1 dimensional space of subrules. Each point represents a subrule's element position within its rhs. As the subrule's current element is being evaluated, if the next element in the subrule is needed due to the current element being an epsilonable rule it overlays the being assessed element. Effectively the subrule being processed is a fishing line that reels in its next element ontop of the current element. It tap dances ontop of the element point until the subrule's elements assessment is exhausted before advancing to the next subrule. After, the subrules residues are their last element that exhausted its assessment. The element space is fixed in size: just the "total number of subrules".

Rule		*subrule's symbols*	residue sym
St	→	S ⊥	S
S	→	R_a^ϵ R_b^ϵ R_c	R_c
	→	R_a^ϵ R_d^ϵ R_e	R_e
R_a^ϵ	→		
	→	||| *a* NSa::THa	|||
R_b^ϵ	→		
	→	||| *b* NSb::THb	|||
R_c	→	c	c
R_d^ϵ	→		
	→	||| *d* NSd::THd	|||
R_e	→	e	e

Above grammar's called thread's first set: FS(Tha)+FS(Thb)+FS(THd)
"/usr/local/yacco2/qa/test_called_thd_fs.lex" grammar exercises algorithm

88. Add subrule's 1st element to element space.

```
⟨ add subrule's 1st element to element space 88 ⟩ ≡
  vector < AST *> ::iterator ti = subrules_can.nodes_visited()→begin();
  vector < AST *> ::iterator tie = subrules_can.nodes_visited()→end();
  for ( ; ti ≠ tie; ++ti) {
    AST * srdef_t = *ti;
    T_subrule_def*␣sr_def␣=␣(T_subrule_def*)AST::content(*srdef_t);
    AST * sr_t = sr_def→subrule_s_tree();
    AST * et = AST::get_spec_child(*sr_t, 1);
    ++nc;
    elem_space[nc] = et;
  }
```

This code is used in sections 89, 90, and 95.

89. Add Start rule's subrules to element space.

```
⟨ add Start rule's subrules to element space 89 ⟩ ≡
  {
    AST * rtree = Start_rule→rule_s_tree();
    ast_prefix_1forest subrule_walk(*rtree, &just_walk_functr, &sr_filter, ACCEPT_FILTER);
    tok_can < AST *> subrules_can(subrule_walk);
    yacco2::UINT xxx(0);
    for ( ; subrules_can.operator[](xxx) ≠ yacco2::PTR_LR1_eog__; ++xxx) ;
    ⟨ add subrule's 1st element to element space 88 ⟩;
  }
```

This code is used in section 93.

90. Add referenced rule to element space.

```
⟨ add referenced rule to element space 90 ⟩ ≡
  {
    i = rules_in_elem_space.find(rdef);
    if (i ≡ rules_in_elem_space.end()) {      /* rule not dealt with so add it */
      rules_in_elem_space.insert(rdef);
      AST * rtree = rdef→rule_s_tree();
      ast_prefix_1forest subrule_walk(*rtree, &just_walk_functr, &sr_filter, ACCEPT_FILTER);
      tok_can < AST *> subrules_can(subrule_walk);
      yacco2::UINT xxx(0);
      for ( ; subrules_can.operator[](xxx) ≠ yacco2::PTR_LR1_eog__; ++xxx) ;
      ⟨ add subrule's 1st element to element space 88 ⟩;
    }
  }
```

This code is used in sections 91 and 96.

91. Assess element type.
Handles "called thread" expression. All other types are thrown out.

```
⟨ deal with thread called element type 91 ⟩ ≡
  switch (cur_elem→enumerated_id__) {
  case T_Enum :: T_T_eosubrule_:
    {
      ++cc;      /* finished with subrule, so adv to next subrule */
      continue;
    }
  case T_Enum :: T_refered_T_:
    {
      refered_T* rt = (refered_T*)cur_elem;
      T_terminal_def * td = rt→its_t_def( );
      if (td→enum_id( ) ≠ LR1_PARALLEL_OPERATOR) {
        ++cc;      /* finished with subrule, so adv to next subrule */
        continue;
      }
      ⟨ deal with thread call expression 92 ⟩;
      ++cc;      /* finished with called expr, so adv to next subrule */
      continue;
    }
  case T_Enum :: T_refered_rule_:
    {
      refered_rule* rr = (refered_rule*)cur_elem;
      rule_def * rdef = rr→its_rule_def( );
      ⟨ add referenced rule to element space 90 ⟩;
      if (rdef→epsilon( ) ≡ YES) {
        AST * et = AST :: brother(*cur_elem_t);
        elem_space[cc] = et;      /* overlay */
        continue;
      }
      else {      /* finished with subrule */
        ++cc;      /* adv to next subrule */
        continue;
      }
    }
  }
```

This code is used in section 93.

92. Deal with "thread call" expression.
Get the "called-thread-eosubrule" terminal in the 3rd tree position. Only add it and not the "null-called-thread-eosubrule" as it indicates that one of the called threads will return its stated "return T".

```
⟨ deal with thread call expression 92 ⟩ ≡
  AST * first_el_t = cur_elem_t;
  AST * third_el_t = AST :: get_younger_sibling(*first_el_t, 2);
  CAbs_lr1_sym * sym = AST :: content(*third_el_t); if (sym→enumerated_id__ ≡
      T_Enum :: T_T_called_thread_eosubrule_) { T_called_thread_eosubrule * ct = ( T_called_thread_eosubrule
      * ) sym;
  Start_rule→add_to_called_thread_first_set(ct); }
```

This code is used in section 91.

93. GEN_CALLED_THREADS_FS_OF_RULE.
Make *elem_space* local to the translation unit instead of the local to the function for efficiecy reasons.

```
⟨ accrue source for emit 8 ⟩ +≡
  AST * elem_space[Max_no_subrules];
  void GEN_CALLED_THREADS_FS_OF_RULE(rule_def * Start_rule)
  {
    using namespace NS_yacco2_T_enum;
    using namespace std;
    std::set < rule_def *> rules_in_elem_space;
    std::set < rule_def *> ::iterator i;
    INT_SET_type sr_filter;
    sr_filter.insert(T_Enum::T_T_subrule_def_);
    tok_can_ast_functor just_walk_functr;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    int cc(0);      /* cur cell */
    int nc(-1);     /* next cell */
    AST * cur_elem_t(0);
    CAbs_lr1_sym * cur_elem(0);
    ⟨ add Start rule's subrules to element space 89 ⟩;
    while (cc ≤ nc) {
      cur_elem_t = elem_space[cc];
      cur_elem = AST::content(*cur_elem_t);
      ⟨ deal with thread called element type 91 ⟩;
    }
  }
```

94. First set calculations.
Raison d'être: ugly formatted code by cweave. This code should reside in *first_set_rules* grammar. There are 2 things being done. Calculate the first set and record the closure only rules for the state used to generate the follow sets.

95. Add Defined rule's subrules to element space.

```
⟨ add Defined rule's subrules to element space 95 ⟩ ≡
  {
    AST * rtree = Rule_def→rule_s_tree( );
    ast_prefix_1forest subrule_walk(*rtree, &just_walk_functr, &sr_filter, ACCEPT_FILTER);
    tok_can < AST *> subrules_can(subrule_walk);
    yacco2::UINT xxx(0);
    for ( ; subrules_can.operator[](xxx) ≠ yacco2::PTR_LR1_eog__; ++xxx) ;
    ⟨ add subrule's 1st element to element space 88 ⟩;
  }
```

This code is used in section 98.

96. Assess element type.

```
⟨ deal with element type 96 ⟩ ≡
  switch (cur_elem→enumerated_id__) {
  case T_Enum::T_T_eosubrule_:
    {
      ++cc;      /* finished with subrule, so adv to next subrule */
      continue;
    }
  case T_Enum::T_refered_T_:
    {
      refered_T* rt = (refered_T*)cur_elem;
      T_in_stbl * tstbl = rt→t_in_stbl( );
      Rule_def→add_to_first_set(*tstbl);
      Rule_def→add_rule_adding_T_in_first_set(tstbl, rt→grammar_s_enumerate( ));
      ++cc;      /* finished subrule, so adv to next subrule */
      continue;
    }
  case T_Enum::T_refered_rule_:
    {
      refered_rule* rr = (refered_rule*)cur_elem;
      rule_def * rdef = rr→its_rule_def( );
      ⟨ add referenced rule to element space 90 ⟩;
      if (rdef→epsilon( ) ≡ YES) {
        AST * et = AST::brother(*cur_elem_t);
        elem_space[cc] = et;      /* overlay */
        continue;
      }
      else {      /* finished with subrule */
        ++cc;      /* adv to next subrule */
        continue;
      }
    }
  }
```

This code is used in section 98.

97. Check Start rule for ϵ condition.
This deals with thread only grammars. Their first sets determine whether they get called. So why the check? What does it mean when the Start rule is ϵ for a thread? The backstop is its lookahead that normally becomes the reducing T set to accept the grammar. I choose to call it just in case the thread is a logic sequencer of ϵ rules. The consequence is just an abort blip by the called thread. If the grammar writer doesn't like it, remove the Start rule ϵ condition.

```
⟨ is Start rule ϵ if yes add LA expression to first set 97 ⟩ ≡
  if (Rule_def→rule_no( ) ≠ 1) goto not_start_rule;
  if (O2_PP_PHASE ≡ 0) goto not_start_rule;      /* not a thread grammar */
  if (Rule_def→epsilon( ) ≡ NO) goto not_start_rule;      /* LA not needed */
  {
    T_parallel_la_boundary * la = O2_PP_PHASE→la_bndry( );
    set < T_in_stbl *> ::iterator lai = la→la_first_set( )→begin( );
    set < T_in_stbl *> ::iterator laie = la→la_first_set( )→end( );
    for ( ; lai ≠ laie; ++lai) {
      T_in_stbl * tstbl = *lai;
      Rule_def→add_to_first_set(*tstbl);
    }
  }
not_start_rule: ;
```

This code is used in section 98.

98. GEN_FS_OF_RULE.
The only wrinkle is the Start rule. What do u do when the grammar is a thread and the Start rule is ϵ ? Well add the "lookahead expression" to its first set.

```
⟨ accrue source for emit 8 ⟩ +≡
  void GEN_FS_OF_RULE(rule_def * Rule_def)
  {
    using namespace NS_yacco2_T_enum;
    using namespace std;
    std::set < rule_def *> rules_in_elem_space;
    std::set < rule_def *> ::iterator i;
    INT_SET_type sr_filter;
    sr_filter.insert(T_Enum::T_T_subrule_def_);
    tok_can_ast_functor just_walk_functr;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    int cc(0);      /* cur cell */
    int nc(-1);      /* next cell */
    AST * cur_elem_t(0);
    CAbs_lr1_sym * cur_elem(0);
    ⟨ add Defined rule's subrules to element space 95 ⟩;
    ⟨ is Start rule ϵ if yes add LA expression to first set 97 ⟩;
    while (cc ≤ nc) {
      cur_elem_t = elem_space[cc];
      cur_elem = AST::content(*cur_elem_t);
      ⟨ deal with element type 96 ⟩;
    }
  }
```

99. Print out Rule's First Set.
The rule's first set elements are sorted in lexicographical order.

100. Set up first set sort.
Why your own copy into a "random iterator" container? Glad u asked dave, MS "copy" template algorithm aborts and "stable_sort" requires a "random iterator" or it will give u a compile time error on non Microsoft platforms. *copy*(*fs*→*begin*( ), *fs*→*end*( ), *ran_array*.*begin*( )); works properly on Sun.

```
⟨ set up first set sort 100 ⟩ ≡
  T_IN_STBL_SET_type * fs = Rule_def→first_set( );
  Ofile ≪ "@*2␣|" ≪ Rule_def→rule_name( )→c_str( ) ≪ "|";
  if (Rule_def→epsilon( ) ≡ true) {
    Ofile ≪ "{$^\\emptyrule$}";
  }
  Ofile ≪ "␣\\#␣in␣set:␣" ≪ fs→size( ) ≪ ".\\fbreak" ≪ std :: endl;
  T_IN_STBL_SORTED_SET_type ran_array;
  ran_array.reserve(fs→size( ));
  T_IN_STBL_SET_ITER_type ii = fs→begin( );
  T_IN_STBL_SET_ITER_type iie = fs→end( );
  for ( ; ii ≠ iie; ++ii) {
    ran_array.push_back(*ii);
  }
  stable_sort(ran_array.begin( ), ran_array.end( ), fs_sort_criteria);
```

This code is used in section 102.

101. Print out the sorted first set elements.

```
⟨ print out the sorted first set elements 101 ⟩ ≡
  char xlate[Max_cweb_item_size];
  T_IN_STBL_SORTED_SET_ITER_type i = ran_array.begin( );
  T_IN_STBL_SORTED_SET_ITER_type ie = ran_array.end( );
  for ( ; i ≠ ie; ++i) {
    T_in_stbl * el = *i;
    Ofile ≪ "␣";
    XLATE_SYMBOLS_FOR_cweave(el→t_def( )→t_name( )→c_str( ), xlate);
    Ofile ≪ xlate ≪ "\\␣\\␣" ≪ std :: endl;
  }
```

This code is used in section 102.

102. Driver of PRT_RULE_S_FIRST_SET.

```
⟨ accrue source for emit 8 ⟩ +≡
  bool fs_sort_criteria(const NS_yacco2_terminals::T_in_stbl*E1,
          const NS_yacco2_terminals::T_in_stbl*E2)
  {
    T_in_stbl*␣e1␣=␣(T_in_stbl*)E1;
    T_in_stbl*␣e2␣=␣(T_in_stbl*)E2;
    T_terminal_def * e1_t_def = e1→t_def( );
    T_terminal_def * e2_t_def = e2→t_def( );
    string * e1s = e1_t_def→t_name( );
    string * e2s = e2_t_def→t_name( );
    if (*e1s < *e2s) return true;
    return false;
  }
  void PRT_RULE_S_FIRST_SET(std::ofstream & Ofile, NS_yacco2_terminals::rule_def * Rule_def)
  {
    ⟨ set up first set sort 100 ⟩;
    ⟨ print out the sorted first set elements 101 ⟩;
  }
```

103. Commonize LA sets.

104. *find_common_la_set_idx*.
Reduce the number of lookahead sets by keeping a global registry. The index returned is the vector's index. This value is deposited in the state element's reducing configuration. It is is used in manufacturing the common la set's name.

Some optimizations:
If *eolr* is in set, as it represents all terminals, just keep it as the other terminals are ch ch chatter.

```
⟨ accrue source for emit 8 ⟩ +≡
  LA_SET_type * make_copy_of_la_set(LA_SET_type * Set1)
  {
    LA_SET_type * copy_set = new LA_SET_type;
    using namespace yacco2_stbl;
    T_sym_tbl_report_card report_card;
    find_sym_in_stbl(report_card, *LR1_EOLR_LITERAL);
    T_in_stbl* td = (T_in_stbl*)report_card.tbl_entry_->symbol_;
    LA_SET_ITER_type f = Set1->find(td);
    if (f ≠ Set1->end()) {
      copy_set->insert(td);
      return copy_set;
    }
    LA_SET_ITER_type i = Set1->begin();
    LA_SET_ITER_type ie = Set1->end();
    for ( ; i ≠ ie; ++i) {
      copy_set->insert(*i);
    }
    return copy_set;
  }
```

105. *are_2_la_sets_equal*.
See bugs as to why i roll my own.

```
⟨ accrue source for emit 8 ⟩ +≡
  bool are_2_la_sets_equal(LA_SET_type * Common_set1, LA_SET_type * Set2)
  {
    LA_SET_type * la_set_to_compare_against = Set2;
    using namespace yacco2_stbl;
    T_sym_tbl_report_card report_card;
    find_sym_in_stbl(report_card, *LR1_EOLR_LITERAL);
    T_in_stbl* td = (T_in_stbl*)report_card.tbl_entry_->symbol_;
    LA_SET_type * eolr_set(0);
    LA_SET_ITER_type f = Set2->find(td);
    if (f ≠ Set2->end()) {
      eolr_set = new LA_SET_type;
      eolr_set->insert(td);
      la_set_to_compare_against = eolr_set;
    }
    if (Common_set1->size() ≠ la_set_to_compare_against->size()) {
      if (eolr_set ≠ 0) delete eolr_set;
      return false;
    }
    LA_SET_ITER_type set1i = Common_set1->begin();
    LA_SET_ITER_type set1ie = Common_set1->end();
    LA_SET_ITER_type set2i = la_set_to_compare_against->begin();
    LA_SET_ITER_type set2ie = la_set_to_compare_against->end();
    for ( ; set1i ≠ set1ie; ) {
      T_in_stbl * set1T = *set1i;
      T_in_stbl * set2T = *set2i;
      if (set1T ≠ set2T) return false;
      ++set1i;
      ++set2i;
    }
    if (eolr_set ≠ 0) delete eolr_set;
    return true;
  }
```

106. *find_common_la_set_idx*.

```
⟨ accrue source for emit 8 ⟩ +≡
  int find_common_la_set_idx(LA_SET_type * La_)
  {
    if (COMMON_LA_SETS.empty( ) ≡ true) {
      LA_SET_type * copy_set = make_copy_of_la_set(La_);
      COMMON_LA_SETS.push_back(copy_set);
      return 0;
    }
    COMMON_LA_SETS_ITER_typei = COMMON_LA_SETS.begin( );
    COMMON_LA_SETS_ITER_typeie = COMMON_LA_SETS.end( );
    int idx(0);
    for ( ; i ≠ ie; ++i, ++idx) {
      LA_SET_type * st1 = *i;
      bool result = are_2_la_sets_equal(st1 , La_);
      if (result ≡ true) {
        return idx;
      }
    }
    LA_SET_type * copy_set = make_copy_of_la_set(La_);
    COMMON_LA_SETS.push_back(copy_set);
    return idx;
  }
```

107. COMMONIZE_LA_SETS.
Reduce the number of lookahead sets by keeping a global registry. The index returned is the COMMON_LA_SETS index. This value is deposited in the state element's reducing configuration. It is used in the manufacturing of the common la set's name referenced from the emitted tables.

```
⟨ accrue source for emit 8 ⟩ +≡
  void COMMONIZE_LA_SETS( )
  {
    STATES_ITER_type si = LR1_STATES.begin( );
    STATES_ITER_type sie = LR1_STATES.end( );
    for ( ; si ≠ sie; ++si) {
      state * cur_state = *si;
      S_VECTORS_ITER_type svi = cur_state→state_s_vector_.begin( );
      S_VECTORS_ITER_type svie = cur_state→state_s_vector_.end( );
      for ( ; svi ≠ svie; ++svi) {
        S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
        S_VECTOR_ELEMS_ITER_type selie = svi→second.end( );
        for ( ; seli ≠ selie; ++seli) {
          state_element * se = *seli;
          if (se→goto_state_ ≠ 0) continue;
          bool set_ok = se→calc_la(*se);      /* make sure it is filled directly */
          if (set_ok ≡ false) {      /* la set is empty so get out */
            return;
          }
          se→common_la_set_idx_ = find_common_la_set_idx(se→la_set_);
        }
      }
    }
  }
```

108. Calculate recycling rule's use count.
This is external as my cweave documentation approach to grammars is rudimentary. It tag teams with *rules_use_cnt.lex* grammar who drives the overall evaluation and use count posting per defined rule.

1) if RxSkeleton exists in cyclic table then return its use value
2) add RxSkeleton to cyclic table
3) evaluate each skeleton's rhs for rule's use cnt
   a) determine indirect use count for rhs
   b) determine rhs's max use cnt from direct and indirect use cnts

109. *calc_cyclic_key*.
Compound keys for a c++ map template are a nuisance due to the support operators required for key ordering etc. So i'll use a unique key of int and let the map deal well with an integer key.

The 2 parts of the key are the rule defs using their enumerated values. So i'll make the 2 parts of the key as a number system of base "number of rules". Part 1 is the 2nd digit position and part 2 is the units digit. As the rule's enumeration starts after the T vocabulary, i must remove the relative ranking of the T vocabulary. This way i'm only dealing with base number of rules.

```
⟨ accrue source for emit 8 ⟩ +≡
  int calc_cyclic_key
  (NS_yacco2_terminals::rule_def * Rule_use, NS_yacco2_terminals::rule_def * Against_rule)
  {
    using namespace NS_yacco2_T_enum;
    static int no_of_rules(0);
    static int start_of_rules(0);
    if (no_of_rules ≡ 0) {
      start_of_rules = O2_T_ENUM_PHASE→total_enumerate( );
      no_of_rules = O2_RULES_PHASE→rules_alphabet( )→size( );
    }
    int rule_use_enum_rel0 = Rule_use→enum_id( ) − start_of_rules;
    int part1_key_of_rule_use = rule_use_enum_rel0 * no_of_rules;
    int against_rule_enum_rel0 = Against_rule→enum_id( ) − start_of_rules;
    int part2_key_of_against_rule = against_rule_enum_rel0;
    int key = part1_key_of_rule_use + part2_key_of_against_rule;
    return key;
  }
```

110. *determine_rhs_indirect_use_cnt*.
The subrule forest is passed in. Just walk its rhs's elements. Indirect count comes from rules who reference the "use cnt" rule. The assessment is from its maximum derives count determined against all its rhses. Note the recursion on a referenced rule within its rhs that can derive another rule that can reference the "use cnt" rule.

```
⟨ accrue source for emit 8 ⟩ +≡
  void determine_rhs_indirect_use_cnt
  (NS_yacco2_terminals::rule_def * Use_for_rule, yacco2::AST * SRule_t,
        NS_yacco2_terminals::rule_def * Against_rule)
  {
    using namespace NS_yacco2_T_enum;
    SET_FILTER_type filter;
    filter.insert(T_Enum::T_refered_rule_);
    tok_can_ast_functor walk_functr;
    ast_prefix_1forest walk(*SRule_t, &walk_functr, &filter, ACCEPT_FILTER);
    TOK_CAN_TREE_type rules_use_can(walk);
    int indirect_use_cnt(0);
    int direct_use_cnt(0);
    int x(0);
    CAbs_lr1_sym * sym = rules_use_can[x];
    for ( ; sym→enumerated_id( ) ≠ LR1_Eog; ++x, sym = rules_use_can[x]) {
      refered_rule* rr = (refered_rule*)sym;
      rule_def * its_rd = rr→its_rule_def( );
      if (its_rd ≡ Use_for_rule) {
        continue;      /* direct cnt */
      }
      int idc = MAX_USE_CNT_RxR(Use_for_rule, its_rd);
    }
  }
```

111. *determine_rhs_max_use_cnt*.
The interesting part is to determine the the use count from direct and derived rules. A derived rule in the rhs must exercise its rhs elements before it reduces. This is like a lookahead of "used rules". Once it reduces to self, it frees up its "used rules".

```
⟨ accrue source for emit 8 ⟩ +≡
  int determine_rhs_max_use_cnt
  (NS_yacco2_terminals::rule_def * Use_for_rule, yacco2::AST * SRule_t,
        NS_yacco2_terminals::rule_def * Against_rule)
  {
    using namespace NS_yacco2_T_enum;
    SET_FILTER_type filter;
    filter.insert(T_Enum::T_refered_rule_);
    tok_can_ast_functor walk_functr;
    ast_prefix_1forest walk(*SRule_t, &walk_functr, &filter, ACCEPT_FILTER);
    TOK_CAN_TREE_type rules_use_can(walk);
    int running_indirect_use_cnt(0);
    int running_direct_use_cnt(0);
    int x(0);
    CAbs_lr1_sym * sym = rules_use_can[x];
    rule_def * its_rd(0);
    for ( ; sym→enumerated_id( ) ≠ LR1_Eog; ++x, sym = rules_use_can[x]) {
      refered_rule* rr=(refered_rule*)sym;
      its_rd = rr→its_rule_def( );
      if (its_rd ≡ Against_rule) Against_rule→recursive(YES);
      if (its_rd ≡ Use_for_rule) {
        ++running_direct_use_cnt;
        continue;      /* direct cnt */
      }
      else {
        use_cnt_type rule_use_key(Use_for_rule, its_rd);
        int use_cnt_key = calc_cyclic_key(Use_for_rule, its_rd);
        CYCLIC_USE_TBL_ITER_type i = CYCLIC_USE_TABLE.find(use_cnt_key);
        use_cnt_type & uc = i→second;
        int idc = uc.use_cnt_;      /* use cnt */
        int potential_idc_cnt = idc + running_direct_use_cnt;
        if (potential_idc_cnt > running_indirect_use_cnt) {
          running_indirect_use_cnt = potential_idc_cnt;
        }
      }
    }
    if (running_direct_use_cnt > running_indirect_use_cnt) {
      return running_direct_use_cnt;
    }
    return running_indirect_use_cnt;
  }
```

112. ***MAX_USE_CNT_RxR*: called by *rules_use_cnt.lex* grammar.**
Watch for cycling. It is initially called from *rules_use_cnt.lex* grammar.

```
⟨ accrue source for emit 8 ⟩ +≡
  int MAX_USE_CNT_RxR
  (NS_yacco2_terminals::rule_def * Rule_use, NS_yacco2_terminals::rule_def * Against_rule)
  {
    use_cnt_type rule_use_key(Rule_use, Against_rule);
    int use_cnt_key = calc_cyclic_key(Rule_use, Against_rule);
    CYCLIC_USE_TBL_ITER_type i = CYCLIC_USE_TABLE.find(use_cnt_key);
    if (i ≠ CYCLIC_USE_TABLE.end()) {
      return i→second.use_cnt_;
    }
    CYCLIC_USE_TABLE.insert(make_pair(use_cnt_key, rule_use_key));
    using namespace NS_yacco2_T_enum;
    if (Against_rule→rule_use_skeleton() ≡ 0) {      /* no rules used in its rhses */
      i = CYCLIC_USE_TABLE.find(use_cnt_key);
      use_cnt_type & uc = i→second;
      uc.use_cnt_ = 0;
      return 0;
    }
    SET_FILTER_type filter;
    filter.insert(T_Enum::T_T_subrule_def_);
    tok_can_ast_functor walk_functr;
    ast_prefix walk(*Against_rule→rule_use_skeleton(), &walk_functr, &filter, ACCEPT_FILTER);
    TOK_CAN_TREE_type rules_use_can(walk);
    int use_cnt(0);
    int x(0);
    CAbs_lr1_sym * sym = rules_use_can[x];
    for ( ; sym→enumerated_id() ≠ LR1_Eog; ++x, sym = rules_use_can[x]) {      /* walk rhses */
      AST * sr_t = rules_use_can.ast(x);
      determine_rhs_indirect_use_cnt(Rule_use, sr_t, Against_rule);
      int rhs_uc = determine_rhs_max_use_cnt(Rule_use, sr_t, Against_rule);
      if (use_cnt < rhs_uc) {
        use_cnt = rhs_uc;
      }
    }
    i = CYCLIC_USE_TABLE.find(use_cnt_key);
    use_cnt_type & uc = i→second;
    if ((Against_rule→recursive() ≡ YES)      /* eliminate self recursion */
    ∧ (Rule_use ≠ Against_rule)) use_cnt = 2 * use_cnt;
    uc.use_cnt_ = use_cnt;
    return use_cnt;
  }
```

113. Emit Grammar by External procedures.
I choose this route to make life easy for me. A table of contents outlines the activities. Each specificly outputted component will be done by an external routine. Some calls could be nested within other external procedures. Local scope uses lower case letters for procedure names whilst global scope uses capital letters.

This approach isolates the variable scoping per routine where software responsibilities are self contained per module and not distributed! It would be nice if c++ could have nested procedure definitions but...

State name convention:
fsm class name both for monolithic and thread grammars. An example:
Format: Sxxx_yyy where xxx is the state number starting from 1 and yyy is the fsm class name : Ceol.
S1_Ceol is the starting state for the Ceol grammar.

Thread convention:
There are 2 thread name contexts — the thread name to call and the thread entry reference:
Thread name procedure:
Name supplied by the "parallel-thread-function" construct.
Eample of a thread procedure from the "eol.lex" grammar:
yacco2::THR _YACCO2_CALL_TYPE NS_eol::TH_eol(yacco2::Parser* To_judge);
Notice that it is caccooned by its namespace supplied by its "parallel-parser" grammar contruct.

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

Thread entry Format: Iyyy where yyy is the thread entry name gened from O_2linker: Ixxx where xxx is thread name supplied by the "parallel-thread-function" construct.
Ex: ITH_eol where eol grammar has "parallel-thread-function" as TH_eol

Thread dispatch tables:
These are arrays sprinkled thru out the LR1 states. They provide the list of thread entries within the state of the threads to be dispatched. Their referenced names use the thread entry "ITH_eol" format.
Within the lr1 state, any called threads will contain a dispatch array of thread procedure addresses.

Arbitrators name convention:
Format: AR_yyy where yyy is the rule name containing thread use. A lr state calling threads could have an arbitrator. This depends whether a rule that closured the called threads into the state has c++ code inside the "arbitrator-code" contruct. This contruct can be empty signifying that the called threads are deterministic: possibly only one of the called threads will return a terminal and no arbritation required to choose between 2 or more returned terminals.

114. Template of Grammar's header file declaration: OP_GRAMMAR_HEADER.

1) Comments — file name, time and date info
2) Includes of O_2 library and grammar vocabulary
3) User-prefix-declaration directive from grammar
4) grammar include guard declaration
  4.1) namespace declaration of grammar
    4.1.0) fsm's rules reuse table extern
    4.1.1) Possible thread entry extern if it's thread grammar
    4.1.2) fsm's 1st state extern reference
    4.1.3) Using O_2 and enumerate namespaces
    4.1.4) Fsm class declaration
      4.1.4.1) Enumeration of grammar's rules and subrules
      4.1.4.2) comments about number of la sets and states
      4.1.4.3) ctor of fsm — constructor directive is implementation only
      4.1.4.4) dtor of fsm — destructor directive is implementation only
      4.1.4.5) op and failed directives — directives are implementation only
      4.1.4.6) user-declaration
      4.1.4.7) Create subrules to rules mapping
    4.1.5) grammar's rules declarations for forward references
    4.1.6) grammar's rules class definitions
5) User-suffix-declaration directive from grammar

115. *intro_comment* — **Intro comments for emitted files.**

A fixed format C++ comment describing the outputted file name with date / time stamp occurance. Used for all grammar's emitted files:

1) xxx.h fsm header file with rule declarations
2) xxx.cpp fsm class and rules implementation
3) xxxsym.cpp reduce_rhs_of_rule procedure and perchance to thread imps
4) xxxtbl.cpp finite automaton — the lr1 state network implementation

Ip:

1) output string
2) file name suffix : 1 of .h, .cpp, sym.cpp, tbl.cpp

⟨ accrue source for emit 8 ⟩ +≡

```
void intro_comment(std::ofstream & Op_str, const char *File_name)
{
  char big_buf_[BIG_BUFFER_32K];
  KCHARP op_template = "/*\n"
  "␣File:␣%s\n"
  "␣Date␣and␣Time:␣%s\n"
  "*/";
  int x = sprintf(big_buf_, op_template, File_name, DATE_AND_TIME());
  Op_str.write(big_buf_, x);
  Op_str << endl;
}
```

116. *grammar_header_includes*.
Get those include files from O_2 library — the canned O_2 library header, LRk and Rc vocabularies normally come from the O_2 library, along with the grammar's local enumeration header, Error and Terminal vocabularies. The libraries order is important as the Errors vocabulary can be used by the Terminals and Rc vocabularies.

```
⟨ accrue source for emit 8 ⟩ +≡
  void grammar_header_includes(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lrk_ph = O2_LRK_PHASE;
    T_rc_phrase * rc_ph = O2_RC_PHASE;
    T_error_symbols_phrase * err_ph = O2_ERROR_PHASE;
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_list = "#include␣\"%s\"\n"      /* yacco2 library HWired */
    "#include␣\"%s.h\"\n"      /* enumeration */
    "#include␣\"%s.h\"\n"      /* lrk symbols vocabulary */
    "#include␣\"%s.h\"\n"      /* Errors vocabulary */
    "#include␣\"%s.h\"\n"      /* Terminal vocabulary */
    "#include␣\"%s.h\"";      /* raw characters vocabulary */
    int x = sprintf(big_buf_
    , include_list
    , O2_library_file
    , enum_ph→filename_id()→identifier()→c_str()
    , lrk_ph→filename_id()→identifier()→c_str()
    , err_ph→filename_id()→identifier()→c_str()
    , t_ph→filename_id()→identifier()→c_str()
    , rc_ph→filename_id()→identifier()→c_str()
    );
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

117. *user_prefix_declaration_for_header*.
Grammar writer supplied code thru the "user-prefix-declaration" directive to be added to the grammar's header file.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_prefix_declaration_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP upd_code = "%s";
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_user_prefix_declaration);
    if (i ≡ dir_map→end( )) return;
    T_user_prefix_declaration* upd = (T_user_prefix_declaration*)i->second;
      int x = sprintf(big_buf_, upd_code, upd→syntax_code( )→syntax_code( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

118. *thread_entry_extern_for_header*.
If the grammar is a thread then its extern reference must be declared in the header. I fetch "parallel-thread-function"'s thread name.

```
⟨ accrue source for emit 8 ⟩ +≡
  void thread_entry_extern_for_header(std::ofstream & Op_str)
  {
    T_parallel_parser_phrase * pp_ph = O2_PP_PHASE;
    if (pp_ph ≡ 0) {
      Op_str ≪ "// monolithic grammar: no thread" ≪ endl;
      return;
    }
    char big_buf_[BIG_BUFFER_32K];
    KCHARP th_extern = "extern yacco2::Thread_entry I%s;\n";
    int x = sprintf(big_buf_, th_extern, pp_ph→pp_funct( )→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

119. *using_ns_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void using_ns_for_header(std::ofstream & Op_str)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP default_ns_use =
    "using␣namespace␣%s;//␣enumerate\n"
    "using␣namespace␣yacco2;";
    int x = sprintf(big_buf_, default_ns_use, enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

120. *fsm_enum_rules_subrules_for_header*.
Enumerates used by *reduce_rhs_of_rule*. The enumerate values for the rules begin after the grammar's terminal vocabularies. The subrules range is 1..n where n is the total numder of all subrules supplied by the rules.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_enum_rules_subrules_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enumeration_of_rules_sbrules =
    "␣␣enum␣rules_and_subrules{\n"
    "␣␣␣␣start_of_rule_list␣=␣%s::T_Enum::sum_total_T";
    int x = sprintf(big_buf_, enumeration_of_rules_sbrules, enum_ph→namespace_id( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    T_rules_phrase * rules_ph = O2_RULES_PHASE;      /* loop thru the rules */
    RULE_DEFS_TBL_ITER_typei = rules_ph→crt_order( )→begin( );
    RULE_DEFS_TBL_ITER_typeie = rules_ph→crt_order( )→end( );
    int overall_subrules_nos(0);
    int current_rule_no(−1);      /* rel to zero */
    int no_rules = rules_ph→rules_alphabet( )→size( );
    KCHARP rule_enum =
    "␣␣␣␣,R_%s_␣=␣%i//start_of_rule_list␣+␣%i";
    for ( ; i ≠ ie; ++i) {
      ++current_rule_no;
      rule_def * rd = *i;
      x = sprintf(big_buf_, rule_enum, rd→rule_name( )→c_str( ), rd→enum_id( ), current_rule_no++);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      T_subrules_phrase * sr_ph = rd→subrules( );
      SUBRULE_DEFS_ITER_typej = sr_ph→subrules( )→begin( );
      SUBRULE_DEFS_ITER_typeje = sr_ph→subrules( )→end( );
      KCHARP subrule_enum =
      "␣␣␣␣␣,rhs%i_%s_␣=␣%i";
      for ( ; j ≠ je; ++j) {
        T_subrule_def * srd = *j;
        ++overall_subrules_nos;
        int x = sprintf(big_buf_, subrule_enum, srd→subrule_no_of_rule( ), rd→rule_name( )→c_str( ),
            overall_subrules_nos);
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
    }
    KCHARP end_enumeration_of_rules_sbrules =
    "␣␣};";
    Op_str ≪ end_enumeration_of_rules_sbrules ≪ endl;
  }
```

121. *fsm_map_subrules_to_rules_for_header* — **Optimization.**
Create the array to map the subrules into their rules. This is used to optimize rule crete-run-delete cylce by recycling. Note, the subrules enumerate is 1..x and the rule's enumerate must be remapped to 0..number of rules-1. To handle array accessing to 0, an extra entry for 0 is generated.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_map_subrules_to_rules_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP array_of_subrules_to_rules_mapping =
    "␣␣fsm_rules_reuse_table_type␣fsm_rules_reuse_table;\n""␣␣static␣int␣rhs_to\
        _rules_mapping_[%i];";
    T_rules_phrase * rules_ph = O2_RULES_PHASE;      /* loop thru the rules */
    RULE_DEFS_TBL_ITER_typei = rules_ph→crt_order( )→begin( );
    RULE_DEFS_TBL_ITER_typeie = rules_ph→crt_order( )→end( );
    int overall_subrules_nos(0);
    for ( ; i ≠ ie; ++i) {
      rule_def * rd = *i;
      T_subrules_phrase * sr_ph = rd→subrules( );
      SUBRULE_DEFS_ITER_typej = sr_ph→subrules( )→begin( );
      SUBRULE_DEFS_ITER_typeje = sr_ph→subrules( )→end( );
      for ( ; j ≠ je; ++j) {
        ++overall_subrules_nos;
      }
    }
    ++overall_subrules_nos;      /* extra subrule due to [0] needed */
    int x = sprintf(big_buf_, array_of_subrules_to_rules_mapping, overall_subrules_nos);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

122. *fsm_comments_about_la_sets_and_states*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_comments_about_la_sets_and_states(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP no_of_la_sets_and_states =
    "␣␣//no␣of␣la␣sets␣=␣%i\n"
    "␣␣//no␣of␣states␣=␣%i";
    int x = sprintf(big_buf_, no_of_la_sets_and_states, COMMON_LA_SETS.size( ), NO_LR1_STATES);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

123. *fsm_class_declaration_for_header*.
The dtor, "op", and "user-declaration" procedures are gened when their directives present.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_class_declaration_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP class_fsm =
    "class␣%s:␣public␣yacco2::CAbs_fsm␣{\n"
    "␣public:";
    int x = sprintf(big_buf_, class_fsm, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    fsm_enum_rules_subrules_for_header(Op_str);
    fsm_comments_about_la_sets_and_states(Op_str);
    KCHARP ctor_fsm =
    "␣␣%s();";
    x = sprintf(big_buf_, ctor_fsm, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP dtor_fsm =
    "␣␣~%s();";
    x = sprintf(big_buf_, dtor_fsm, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP op_failed =
    "␣␣void␣op();\n"
    "␣␣bool␣failed();";
    Op_str ≪ op_failed ≪ endl;
    KCHARP reduce_rhs =
    "␣␣void␣reduce_rhs_of_rule\n"
    "␣␣␣␣␣␣␣(yacco2::UINT␣Sub_rule_no,yacco2::Rule_s_reuse_entry**␣Recycled_rule);";
    Op_str ≪ reduce_rhs ≪ endl;
    fsm_map_subrules_to_rules_for_header(Op_str);
    KCHARP user_declaration =
    "␣␣%s";
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_user_declaration);
    if (i ≠ dir_map→end( )) {
      T_user_declaration*␣ud␣=␣(T_user_declaration*)i->second;
        int x = sprintf(big_buf_, user_declaration, ud→syntax_code( )→syntax_code( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    KCHARP end_class_fsm =
    "␣};";
    Op_str ≪ end_class_fsm ≪ endl;
  }
```

124. *forward_refs_of_rules_declarations_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void forward_refs_of_rules_declarations_for_header(std::ofstream & Op_str)
  {
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP forward_ref =
    "struct␣%s;";      /* loop thru the rules */
    RULE_DEFS_TBL_ITER_type i = rules_ph->crt_order()->begin();
    RULE_DEFS_TBL_ITER_type ie = rules_ph->crt_order()->end();
    for ( ; i ≠ ie; ++i) {
      rule_def * rd = *i;
      int x = sprintf(big_buf_, forward_ref, rd->rule_name()->c_str());
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

125. *rules_class_defs_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void rules_class_defs_for_header(std::ofstream & Op_str)
  {
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP end_rule_def =
    "};\n";
    ⟨ gen fsm rules declarations 126 ⟩;
  }
```

126.

```
⟨ gen fsm rules declarations 126 ⟩ ≡
  RULE_DEFS_TBL_ITER_type i = rules_ph→crt_order()→begin();
  RULE_DEFS_TBL_ITER_type ie = rules_ph→crt_order()→end();
  KCHARP start_rule_def =
  "struct␣%s:public␣yacco2::CAbs_lr1_sym␣{\n"
  "␣␣%s(yacco2::Parser*␣P);";
  for ( ; i ≠ ie; ++i) {
    rule_def * rd = *i;
    ⟨ gen rule's arbitrator procedure declaration 127 ⟩;
    int x = sprintf(big_buf_, start_rule_def, rd→rule_name()→c_str(), rd→rule_name()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    T_rule_lhs_phrase * rlhs = rd→rule_lhs();
    if (rlhs ≠ 0) {
      SDC_MAP_type * dir_map = rlhs→lhs_directives_map();
      SDC_MAP_ITER_type i = dir_map→find(SDC_destructor);
      if (i ≠ dir_map→end()) {
        KCHARP rule_def_dtor =
        "␣␣static␣void␣dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P);";
        int x = sprintf(big_buf_, rule_def_dtor, rd→rule_name()→c_str());
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      i = dir_map→find(SDC_op);
      if (i ≠ dir_map→end()) {
        KCHARP rule_def_op =
        "␣␣void␣op();";
        Op_str ≪ rule_def_op ≪ endl;
      }
      i = dir_map→find(SDC_constructor);
      if (i ≠ dir_map→end()) {
        KCHARP rule_def_ctor =
        "␣␣void␣ctor();";
        Op_str ≪ rule_def_ctor ≪ endl;
      }
      i = dir_map→find(SDC_user_declaration);
      if (i ≠ dir_map→end()) {
        T_user_declaration*␣ud␣=␣(T_user_declaration*)i->second;
        Op_str ≪ ud→syntax_code()→syntax_code()→c_str() ≪ endl;
      }
    }
    ⟨ gen fsm subrules declarations 128 ⟩;
  }
```

This code is used in section 125.

127. Gen rule's arbitrator procedure declaration.
It is declared outside of the rule making its call easier from the generated lr1 tables.

```
⟨ gen rule's arbitrator procedure declaration 127 ⟩ ≡
  if (rd→parallel_mntr() ≠ 0) {
    T_parallel_monitor_phrase * pp_phrase = rd→parallel_mntr();
    if (rd→parallel_mntr()→mntr_directives_map()→empty() ≠ true) {
      SDC_MAP_type * dir_map = pp_phrase→mntr_directives_map();
      SDC_MAP_ITER_type i = dir_map→find(SDC_arbitrator_code);
      if (i ≠ dir_map→end()) {
        T_arbitrator_code* ac = (T_arbitrator_code*)i->second;
        string::size_type idx =
            ac→syntax_code()→syntax_code()→find(CODE_PRESENCE_IN_ARBITRATOR_CODE);
        if (idx ≠ string::npos) {
          KCHARP rule_s_arbitrator =
          "yacco2::THR _YACCO2_CALL_TYPE\n"
          "AR_%s(yacco2::Parser* Caller_pp);// rule's arbitrator";
          int x = sprintf(big_buf_, rule_s_arbitrator, rd→rule_name()→c_str());
          Op_str.write(big_buf_, x);
          Op_str ≪ endl;
          RULES_HAVING_AR.insert(rd);
        }
      }
    }
  }
```

This code is used in section 126.

128. Gen fsm subrules declarations.
Optimization: only gen definition if there is syntax directed code.

```
⟨ gen fsm subrules declarations 128 ⟩ ≡
  KCHARP sub_rule_def_public =
  "  public:";
  Op_str ≪ sub_rule_def_public ≪ endl;
  KCHARP sub_rule_def =
  "  void sr%i();";
  T_subrules_phrase * sr_ph = rd→subrules();
  SUBRULE_DEFS_ITER_type j = sr_ph→subrules()→begin();
  SUBRULE_DEFS_ITER_type je = sr_ph→subrules()→end();
  for (int xx = 1; j ≠ je; ++j, ++xx) {
    T_subrule_def * srd = *j;
    SDC_MAP_type * sdrmap = srd→subrule_directives();
    if (¬sdrmap→empty()) {      /* any syntax directed code, call it */
      int x = sprintf(big_buf_, sub_rule_def, xx);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
  Op_str ≪ end_rule_def ≪ endl;
```

This code is used in section 126.

129. *possible_thread_procedure_declaration*.
2 parts:
1) the thread declaration
2) the procedure twin to thread declaration: an experiment in trying to improve threading overhead with all its mutex paraphernalia. Only happens when only 1 thread can run to call it as a local procedure. The part needed improving is in the ctor/use/dtor action caused by the localness of Parser and the fsm table. Now it will be newed when first called.

```
⟨ accrue source for emit 8 ⟩ +≡
  void possible_thread_procedure_declaration(std::ofstream & Op_str)
  {
    if (O2_PP_PHASE ≡ 0) return;
    T_parallel_parser_phrase * pp_ph = O2_PP_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP thd_proc =
    "yacco2::THR␣_YACCO2_CALL_TYPE\n"
    "%s(yacco2::Parser*␣Caller);//␣called␣thread\n""yacco2::THR_result␣_YACCO2_CALL_TYPE\n"
    "PROC_%s(yacco2::Parser*␣Caller);//␣called␣thread's␣twin␣the␣procedure";
    int x = sprintf(big_buf_, thd_proc, pp_ph→pp_funct()→identifier()→identifier()→c_str(),
        pp_ph→pp_funct()→identifier()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

130. *rules_reuse_table_declaration* — **Optimization.**
Table definition supporting recycled rules. Gen specifically per rule from the findings determined by *rules_use_cnt.lex* grammar;

```
⟨ accrue source for emit 8 ⟩ +≡
  void rules_reuse_table_declaration(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase();
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    int no_rules = rules_ph→rules_alphabet()→size();
    char big_buf_[BIG_BUFFER_32K];
    KCHARP rules_recycled_table_type =
    "struct␣fsm_rules_reuse_table_type{\n"
    "␣fsm_rules_reuse_table_type();\n"
    "␣int␣no_rules_entries_;\n"
    "␣Per_rule_s_reuse_table*␣per_rule_s_table_[%i];\n"
    "};";
    int x = sprintf(big_buf_, rules_recycled_table_type, no_rules);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

131. *namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void namespace_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP ns_of_grammar_start =
    "namespace␣%s␣{";
    int x = sprintf(big_buf_, ns_of_grammar_start, fsm_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    possible_thread_procedure_declaration(Op_str);
    using_ns_for_header(Op_str);
    rules_reuse_table_declaration(Op_str);
    fsm_class_declaration_for_header(Op_str);
    forward_refs_of_rules_declarations_for_header(Op_str);
    rules_class_defs_for_header(Op_str);
    KCHARP ns_of_grammar_end =
    "}␣//␣end␣of␣namespace\n";
    Op_str ≪ ns_of_grammar_end ≪ endl;
  }
```

132. *state_1_extern*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void state_1_extern(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase();
    char big_buf_[BIG_BUFFER_32K];
    KCHARP state_extern =
    "extern␣yacco2::State␣S1_%s;";
    int x = sprintf(big_buf_, state_extern, fsm_class→identifier()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

133. *grammar_include_guard_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void grammar_include_guard_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP signal_guard_start =
    "#ifndef␣__%s_h__\n"
    "#define␣__%s_h__␣1";
    int x = sprintf(big_buf_, signal_guard_start, fsm_ph→filename_id( )→identifier( )→c_str( ),
        fsm_ph→filename_id( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    grammar_header_includes(Op_str);
    user_prefix_declaration_for_header(Op_str);
    thread_entry_extern_for_header(Op_str);
    state_1_extern(Op_str);
    namespace_for_header(Op_str);
    KCHARP signal_guard_end =
    "#endif";
    Op_str ≪ signal_guard_end ≪ endl;
  }
```

134. *user_suffix_declaration_for_header*.
Grammar writer supplied code thru the "user-suffix-declaration" directive to be added to the grammar's header file.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_suffix_declaration_for_header(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_user_suffix_declaration);
    if (i ≡ dir_map→end( )) return;
    T_user_suffix_declaration*␣upd␣=␣(T_user_suffix_declaration*)i->second;
    KCHARP upd_code = "%s";
    int x = sprintf(big_buf_, upd_code, upd→syntax_code( )→syntax_code( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

135. OP_GRAMMAR_HEADER implementation.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_GRAMMAR_HEADER(TOKEN_GAGGLE & Error_queue)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    string fn(fsm_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_fsmheader;
    std :: ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_fsmheader_filename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*fsm_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_GRAMMAR_HEADER", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    grammar_include_guard_for_header(Op_file);
    user_suffix_declaration_for_header(Op_file);
    Op_file.close( );
  }
```

136. Template of Grammar's fsm file implementation: `OP_GRAMMAR_CPP`.

1) Comments — file name, time and date info
2) Include of grammar's header
3) using namespaces
4) fill in rules reuse table
5) Fsm class implementation
6) grammar's rules / subrules implementations

137. *fsm_cpp_includes*.
Get grammar's include file.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_cpp_includes(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_list = "#include␣\"%s.h\"";      /* grammar's header */
    int x = sprintf(big_buf_, include_list, fsm_ph→filename_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
```

138. *using_ns_for_fsm_cpp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void using_ns_for_fsm_cpp(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lrk_ph = O2_LRK_PHASE;
    T_rc_phrase * rc_ph = O2_RC_PHASE;
    T_error_symbols_phrase * err_ph = O2_ERROR_PHASE;
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP default_ns_use =
    "using␣namespace␣%s;//␣enumerate\n"
    "using␣namespace␣%s;//␣error␣symbols\n"
    "using␣namespace␣%s;//␣lrk␣\n"
    "using␣namespace␣%s;//␣terminals\n"
    "using␣namespace␣%s;//␣rc␣\n"
    "using␣namespace␣%s;//␣yacco2␣library\n"
    "using␣namespace␣%s;//␣grammar's␣ns\n"
    "//␣first␣set␣terminals";
    int x = sprintf(big_buf_, default_ns_use, enum_ph→namespace_id( )→identifier( )→c_str( ),
        err_ph→namespace_id( )→identifier( )→c_str( )
    , lrk_ph→namespace_id( )→identifier( )→c_str( )
    , t_ph→namespace_id( )→identifier( )→c_str( )
    , rc_ph→namespace_id( )→identifier( )→c_str( )
    , "yacco2"
    , fsm_ph→namespace_id( )→identifier( )→c_str( )
    );
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

139. *rules_reuse_table_implementation* — **Optimization.**
Table definition supporting recycled rules. Use count per rule garnered from the *rules_use_cnt.lex* grammar.

```
⟨ accrue source for emit 8 ⟩ +≡
  void rules_reuse_table_implementation(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    int no_rules = rules_ph→rules_alphabet()→size();
    char big_buf_[BIG_BUFFER_32K];
    KCHARP rules_recycled_table_type =
    "fsm_rules_reuse_table_type::fsm_rules_reuse_table_type(){\n""␣no_rules_entries_␣\
        =␣%i;";
    int x = sprintf(big_buf_, rules_recycled_table_type, no_rules);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP per_rule_s_table_ref_type =
    "␣per_rule_s_table_[%i]␣=␣␣new␣Per_rule_s_reuse_table();";
    for (int xx = 1; xx ≤ no_rules; ++xx) {
      int x0 = xx − 1;
      x = sprintf(big_buf_, per_rule_s_table_ref_type, x0);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    Op_str ≪ "}" ≪ endl;
  }
```

140. *fsm_map_subrules_to_rules_table_imp* — **Optimization.**
Load up the table to map subrules into their rules numeration.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_map_subrules_to_rules_table_imp(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    RULE_DEFS_TBL_ITER_typei = rules_ph→crt_order( )→begin( );
    RULE_DEFS_TBL_ITER_typeie = rules_ph→crt_order( )→end( );
    int overall_subrules_nos(0);
    for ( ; i ≠ ie; ++i) {
      rule_def * rd = *i;
      T_subrules_phrase * sr_ph = rd→subrules( );
      SUBRULE_DEFS_ITER_typej = sr_ph→subrules( )→begin( );
      SUBRULE_DEFS_ITER_typeje = sr_ph→subrules( )→end( );
      for ( ; j ≠ je; ++j) {
        ++overall_subrules_nos;
      }
    }
    ++overall_subrules_nos;      /* extra subrule due to [0] needed */
    KCHARP array_of_subrules_to_rules_mapping_start =
    "int␣%s::rhs_to_rules_mapping_[%i]␣=␣{\n"
    "␣-1";      /* make room for false [0] entry */
    int x = sprintf(big_buf_, array_of_subrules_to_rules_mapping_start,
        fsm_class→identifier( )→identifier( )→c_str( ), overall_subrules_nos);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP subrules_to_rule_mapping =
    "␣,%i␣//␣subrule␣%i␣for␣rule␣%i";
    KCHARP array_of_subrules_to_rules_mapping_end =
    "};%s";      /* loop thru the rules */
    i = rules_ph→crt_order( )→begin( );
    ie = rules_ph→crt_order( )→end( );
    overall_subrules_nos = 0;
    for ( ; i ≠ ie; ++i) {
      rule_def * rd = *i;
      T_subrules_phrase * sr_ph = rd→subrules( );
      SUBRULE_DEFS_ITER_typej = sr_ph→subrules( )→begin( );
      SUBRULE_DEFS_ITER_typeje = sr_ph→subrules( )→end( );
      for ( ; j ≠ je; ++j) {
        ++overall_subrules_nos;
        int x = sprintf(big_buf_, subrules_to_rule_mapping, rd→rule_no( ) − 1, overall_subrules_nos,
            rd→rule_no( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
    }
```

```
    x = sprintf(big_buf_, array_of_subrules_to_rules_mapping_end, "␣");
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    enum_ph→total_no_subrules(overall_subrules_nos);
  }
```

141. *fsm_class_implementation*.
Directives: — see *fsm_class_phrase_th* grammar
1) constructor
2) destructor
3) failed
4) op
5) load up subrles mapping into rules
6) user-implementation roll your own code if present
7) user-imp-tbl — injection code for xxxtbl.cpp where xxx is grammar name
8) user-imp-sym — injection code for xxxsym.cpp where xxx is grammar name

Points 6 and 7 will be dealt with in their own "cpp" implementations. Note: added deletion of each "recycling table" per rule within the dtor.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_class_implementation(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP ctor_fsm =
    "␣␣%s::\n"
    "␣␣%s()\n"
    "␣␣␣␣:yacco2::CAbs_fsm\n"
    "␣␣␣␣␣␣(\"%s\"\n"
    "␣␣␣␣␣␣,\"%s\"\n"
    "␣␣␣␣␣␣,\"%s\"\n"
    "␣␣␣␣␣␣,%s\n"
    "␣␣␣␣␣␣,\"%s\"\n"
    "␣␣␣␣␣␣,\"%s\"\n"
    "␣␣␣␣␣␣,S1_%s){\n"
    "␣␣␣␣%s\n"
    "␣␣}";
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_constructor);
    const char *cc_str(0);
    if (i ≠ dir_map→end( )) {
      T_constructor*␣cc␣=␣(T_constructor*)i->second;
      cc_str = cc→syntax_code( )→syntax_code( )→c_str( );
    }
    else {
      cc_str = "";
    }
    int x = sprintf(big_buf_, ctor_fsm, fsm_class→identifier( )→identifier( )→c_str( ),
        fsm_class→identifier( )→identifier( )→c_str( ), fsm_ph→fsm_id( )→c_string( )→c_str( ),
        fsm_ph→version( )→c_string( )→c_str( ), fsm_ph→date( )→c_string( )→c_str( ),
        fsm_ph→debug( )→c_string( )→c_str( ), fsm_ph→comment( )→c_string( )→c_str( ), DATE_AND_TIME( ),
        fsm_class→identifier( )→identifier( )→c_str( ), cc_str);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
```

```
KCHARP dtor_rtn_start =
"␣\n"
"%s::~%s(){\n"
"%s";
i = dir_map→find(SDC_destructor);
const char *dc_str(0);
if (i ≠ dir_map→end()) {
  T_destructor*␣dc␣=␣(T_destructor*)i->second;
  dc_str = dc→syntax_code()→syntax_code()→c_str();
}
else {
  dc_str = "";
}
x = sprintf(big_buf_, dtor_rtn_start, fsm_class→identifier()→identifier()→c_str(),
    fsm_class→identifier()→identifier()→c_str(), dc_str);
Op_str.write(big_buf_, x);
Op_str ≪ endl;      /* delete those recycled rules */
T_rules_phrase * rules_ph = O2_RULES_PHASE;
int no_rules = rules_ph→rules_alphabet()→size();
KCHARP delete_recycled_rules =
"␣␣for(int␣x␣=␣0;x␣<␣%i;++x){\n"
"␣␣␣///delete␣fsm_rules_reuse_table.per_rule_s_table_[x];\n"
"␣␣}";
x = sprintf(big_buf_, delete_recycled_rules, no_rules);
Op_str.write(big_buf_, x);
Op_str ≪ endl;
KCHARP dtor_rtn_end =
"}%s\n""";
x = sprintf(big_buf_, dtor_rtn_end, "␣");
Op_str.write(big_buf_, x);
Op_str ≪ endl;
KCHARP failed_rtn =
"␣␣bool␣%s::failed(){\n"
"␣␣␣␣␣%s\n"
"␣␣}";
i = dir_map→find(SDC_failed);
const char *fc_str(0);
if (i ≠ dir_map→end()) {
  T_failed*␣fc␣=␣(T_failed*)i->second;
  fc_str = fc→syntax_code()→syntax_code()→c_str();
}
else {
  fc_str = "␣␣return␣false;";
}
x = sprintf(big_buf_, failed_rtn, fsm_class→identifier()→identifier()→c_str(), fc_str);
Op_str.write(big_buf_, x);
Op_str ≪ endl;
KCHARP op_rtn =
"␣␣void␣%s::op(){%s\n}";
i = dir_map→find(SDC_op);
```

```
    const char *oc_str(0);
    if (i ≠ dir_map→end( )) {
      T_op* oc = (T_op*)i->second;
      oc_str = oc→syntax_code( )→syntax_code( )→c_str( );
    }
    else {
      oc_str = "";
    }
    x = sprintf(big_buf_, op_rtn, fsm_class→identifier( )→identifier( )→c_str( ), oc_str);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    fsm_map_subrules_to_rules_table_imp(Op_str);
    dir_map = fsm_class→directives_map( );
    i = dir_map→find(SDC_user_implementation);
    if (i ≠ dir_map→end( )) {
      T_user_implementation* ui = (T_user_implementation*)i->second;
      Op_str ≪ ui→syntax_code( )→syntax_code( )→c_str( ) ≪ endl;
    }
  }
```

142. *rules_subrules_implementations*.
Rule directives: See *rule_lhs_phrase* grammar
1) constructor 2) destructor
3) op 4) user-implementation

Subrule directive: See *subrule_def* grammar
1) op

```
⟨ accrue source for emit 8 ⟩ +≡
  void rules_subrules_implementations(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    ⟨ loop thru the rules 143 ⟩;
  }
```

143. Go thru rules.

```
⟨ loop thru the rules 143 ⟩ ≡
  RULE_DEFS_TBL_ITER_type i = rules_ph→crt_order( )→begin( );
  RULE_DEFS_TBL_ITER_type ie = rules_ph→crt_order( )→end( );
  for ( ; i ≠ ie; ++i) {
    rule_def * rd = *i;
    ⟨ deal with arbitrator code 144 ⟩;
    T_rule_lhs_phrase * rlhs = rd→rule_lhs( );
    const char *ctor_code(0);
    const char *dtor_code(0);
    string dtor_name("0");
    const char *op_code(0);
    const char *user_imp_code(0);
    T_destructor * dtor(0);
    if (rlhs ≠ 0) {
      SDC_MAP_type * dir_map = rlhs→lhs_directives_map( );
      SDC_MAP_ITER_type i = dir_map→find(SDC_constructor);
      if (i ≠ dir_map→end( )) {
        T_constructor* ctor = (T_constructor*)i->second;
        ctor_code = ctor→syntax_code( )→syntax_code( )→c_str( );
      }
      i = dir_map→find(SDC_destructor);
      if (i ≠ dir_map→end( )) {
        dtor = (T_destructor*)i->second;
        dtor_code = dtor→syntax_code( )→syntax_code( )→c_str( );
        KCHARP dtor_nm =
        "&dtor_%s";
        sprintf(big_buf_, dtor_nm, rd→rule_name( )→c_str( ));
        dtor_name.clear( );
        dtor_name += big_buf_;
      }
      i = dir_map→find(SDC_op);
      if (i ≠ dir_map→end( )) {
        T_op* op = (T_op*)i->second;
        op_code = op→syntax_code( )→syntax_code( )→c_str( );
      }
      i = dir_map→find(SDC_user_implementation);
      if (i ≠ dir_map→end( )) {
        T_user_implementation* uimp = (T_user_implementation*)i->second;
        user_imp_code = uimp→syntax_code( )→syntax_code( )→c_str( );
      }
    }
    KCHARP rule_def_imp =
    "%s::%s(yacco2::Parser* P)\n"
    " :CAbs_lr1_sym\n"
    "  (\"%s\",%s,%s::R_%s_,P,%s,%s){\n"
    "}\n";
    string ad;
    if (rd→autodelete( ) ≡ true) {
      ad += "true";
    }
```

```
else {
  ad += "false";
}
string ab;
if (rd→autoabort() ≡ true) {
  ab += "true";
}
else {
  ab += "false";
}
int x = sprintf(big_buf_, rule_def_imp, rd→rule_name()→c_str(), rd→rule_name()→c_str(),
    rd→rule_name()→c_str(), dtor_name.c_str(), fsm_class→identifier()→identifier()→c_str(),
    rd→rule_name()→c_str(), ad.c_str(), ab.c_str());
Op_str.write(big_buf_, x);
Op_str ≪ endl;
if (dtor_code ≠ 0) {
  std::string::size_type r = dtor→syntax_code()→syntax_code()→find("ABORT_STATUS");
  KCHARP rule_def_dtor =
  "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
  "␣bool␣ABORT_STATUS␣=␣((yacco2::Parser*)P)->top_stack_record()->aborted__;\n"
  "␣%s*␣R␣=␣(%s*)(This);\n"
  "␣%s\n"
  "}";
  KCHARP rule_def_dtor_noabort =
  "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
  "␣%s*␣R␣=␣(%s*)(This);\n"
  "␣%s\n"
  "}";
  int x;
  if (r ≠ std::string::npos) {      /* using abort status */
    x = sprintf(big_buf_, rule_def_dtor, rd→rule_name()→c_str(), rd→rule_name()→c_str(),
        rd→rule_name()→c_str(), rd→rule_name()→c_str(), dtor_code);
  }
  else {
    x = sprintf(big_buf_, rule_def_dtor_noabort, rd→rule_name()→c_str(), rd→rule_name()→c_str(),
        rd→rule_name()→c_str(), rd→rule_name()→c_str(), dtor_code);
  }
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
}
if (op_code ≠ 0) {
  KCHARP rule_def_op =
  "void␣%s::op(){\n"
  "␣␣sstrace_rulesss\n"
  "␣%s\n"
  "}";
  int x = sprintf(big_buf_, rule_def_op, rd→rule_name()→c_str(), op_code);
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
}
if (ctor_code ≠ 0) {
```

```
    KCHARP rule_def_ctor =
    "void␣%s::ctor(){\n"
    "␣%s\n"
    "}";
    int x = sprintf(big_buf_, rule_def_ctor, rd→rule_name()→c_str(), ctor_code);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  if (user_imp_code ≠ 0) {
    Op_str ≪ user_imp_code ≪ endl;
  }
  ⟨ loop thru the subrules 146 ⟩;
}
```

This code is used in section 142.

144. Deal with arbitrator code.

```
⟨ deal with arbitrator code 144 ⟩ ≡
  if (rd→parallel_mntr() ≠ 0) { T_parallel_monitor_phrase*␣pp_phrase␣=␣rd->parallel_mntr();
  ⟨ is there arbitration code? — yes output 145 ⟩;
  }
```

This code is used in section 143.

145.

```
⟨ is there arbitration code? — yes output 145 ⟩ ≡
  if (pp_phrase→mntr_directives_map()→empty() ≠ true) {
    SDC_MAP_type * pp_map = pp_phrase→mntr_directives_map();
    SDC_MAP_ITER_type p = pp_map→find(SDC_arbitrator_code);
    if (p ≠ pp_map→end()) {
      T_arbitrator_code*␣ac␣=␣(T_arbitrator_code*)p->second;
      string::size_type idx = ac→syntax_code()→syntax_code()→find(CODE_PRESENCE_IN_ARBITRATOR_CODE);
      if (idx ≠ string::npos) {
        KCHARP rule_s_arbitrator_imp =
        "yacco2::THR␣_YACCO2_CALL_TYPE\n"
        "%s::AR_%s(yacco2::Parser*␣Caller_pp){\n"
        "␣yacco2::KCHARP␣ar_name␣=␣\"AR_%s\";\n"
        "#include␣\"war_begin_code.h\"\n"
        "␣%s\n"
        "#include␣\"war_end_code.h\"\n"
        "}\n";
        int x = sprintf(big_buf_, rule_s_arbitrator_imp, fsm_ph→namespace_id()→identifier()→c_str(),
            rd→rule_name()→c_str(), rd→rule_name()→c_str(), ac→syntax_code()→syntax_code()→c_str());
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
    }
  }
```

This code is used in section 144.

146. Emit the syntax directed code(sdc).
2 optimizations are done: gen the definition if there is syntax directed code, and the second optimization gens the stack frame when it is referenced within the syntax directed code. It is crude in determining whether the sdc uses it: see if there is a reference to the local stack frame variable "sf" searched within the code string. The stack frame variable contains the subrule's items having each parameter start with "p". To reference a subrule's item, one uses the *sf→px__* whereby the stacked item is x starting from 1. *p10__* would reference the tenth item on the stack. The parameter count starts from the left of the subrule. An epsilon subrule has no parameters. There is no c++ sdc validity check done here. It leaves it to the c++ compiler to digest O_2's code.

```
⟨ loop thru the subrules 146 ⟩ ≡
  T_subrules_phrase * sr_ph = rd→subrules( );
  SUBRULE_DEFS_ITER_type j = sr_ph→subrules( )→begin( );
  SUBRULE_DEFS_ITER_type je = sr_ph→subrules( )→end( );
  int nosrs = sr_ph→no_subrules( );
  for (int xx = 1; j ≠ je; ++j, ++xx) {
    T_subrule_def * srd = *j;
    AST * sr_t = AST::get_spec_child(*srd→subrule_s_tree( ), 1);
    int no_parms = srd→no_of_elems( ) − 1;      /* remove eos */
    KCHARP sub_rule_def_imp =
    "void␣%s::sr%i(){";
    SDC_MAP_type * dir_map = srd→subrule_directives( );
    if (dir_map→empty( ) ≠ true) {      /* gen if there is syntax directed code */
      int x = sprintf(big_buf_, sub_rule_def_imp, rd→rule_name( )→c_str( ), xx);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      SDC_MAP_ITER_type i = dir_map→find(SDC_op);
      if (i ≠ dir_map→end( )) {
        T_op*␣rhsc␣=␣(T_op*)i->second;
        std::string::size_type idx;
        idx = rhsc→syntax_code( )→syntax_code( )→find("sf->p");
          /* any reference to the stack frame within the sdc? */
        if (idx ≡ std::string::npos) goto write_out_op_code;
        ⟨ gen stack frame def and equate it to the parse stack 147 ⟩;
      write_out_op_code: ;
        KCHARP sub_rule_def_op_code =
        "␣␣%s";      /* rhs op */
        int x = sprintf(big_buf_, sub_rule_def_op_code, rhsc→syntax_code( )→syntax_code( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      KCHARP sub_rule_def_end =
      "}\n";
      Op_str ≪ sub_rule_def_end ≪ endl;
    }
  }
```

This code is used in section 143.

147. Gen stack frame definition and equate it to the parse stack.
There are 3 conditions that need explanation:
1) LR1_ALL_SHIFT_OPERATOR — `|+|`
2) LR1_INVISIBLE_SHIFT_OPERATOR — `|.|`
2) LR1_QUESTIONABLE_SHIFT_OPERATOR — `|?|`

The "all shift" operator is a proxy that represents a wild terminal on the stack. It has no specific representation! This is why the terminal has to be represented by the abstract terminal "CAbs_lr1_sym". To get its specifics, testing of its enumerate coupled with the cast statement now turns it into a somebody.

The "invisible shift" operator is a proxy epsilon. It does not need to reference the parse stack frame so optimize it out. The "questionable shift" operator is handles error setection points within the grammar. It does not need to reference the parse stack frame so optimize it out.

To introduce chained-called-procedures, the "eosubrule" was attached to the thread call expressions and their variants. To note the called thread or procedure is not part of the parse stack so drop it and re-align the number of subrule parameters.

```
⟨ gen stack frame def and equate it to the parse stack 147 ⟩ ≡
  for (int y = 1; y ≤ no_parms; ++y) {      /* gen stack frame items */
    bool eos_thd_or_proc(false);
    CAbs_lr1_sym * sym = AST::content(*sr_t);
    const char *parm_name(0);
    switch (sym→enumerated_id__) {
    case T_Enum::T_refered_rule_:
      {
        refered_rule* rr = (refered_rule*)sym;
        parm_name = rr→its_rule_def()→rule_name()→c_str();
        break;
      }
    case T_Enum::T_refered_T_:
      {
        refered_T* rt = (refered_T*)sym;
        T_terminal_def * td = rt→its_t_def();
        if (td→enum_id() ≡ LR1_ALL_SHIFT_OPERATOR) {
          parm_name = "CAbs_lr1_sym";
          break;
        }
        if (td→enum_id() ≡ LR1_QUESTIONABLE_SHIFT_OPERATOR) {
          parm_name = "CAbs_lr1_sym";
          break;
        }
        if (td→enum_id() ≡ LR1_INVISIBLE_SHIFT_OPERATOR) {
          parm_name = "CAbs_lr1_sym";
          break;
        }
        parm_name = td→classsym()→c_str();
        break;
      }
    case T_Enum::T_T_identifier_:
      {
        parm_name = "T_identifier";
        break;
      }
```

```
    case T_Enum::T_T_NULL_:
      {
        parm_name = "T_NULL";
        break;
      }
    case T_Enum::T_T_2colon_:
      {
        parm_name = "T_2colon";
        break;
      }
    case T_Enum::T_T_called_thread_eosubrule_:
      {
        eos_thd_or_proc = true;
        --no_parms;
        --y;
        break;
      }
    case T_Enum::T_T_null_call_thread_eosubrule_:
      {
        eos_thd_or_proc = true;
        --no_parms;
        --y;
        break;
      }
    }
    if (y ≡ 1) {      /* stk frame */
      KCHARP sub_rule_stk_parms_begin =
      "␣␣struct␣SF{";
      Op_str ≪ sub_rule_stk_parms_begin ≪ endl;
    }
    KCHARP sub_rule_stk_parm =
    "␣␣␣%s*␣p%i__;\n"
    "␣␣␣State*␣s%i__;\n"
    "␣␣␣bool␣abort%i__;\n""␣␣␣Rule_s_reuse_entry*␣rule_s_reuse_entry%i__;";
    if (eos_thd_or_proc ≡ false) {
      int x = sprintf(big_buf_, sub_rule_stk_parm, parm_name, y, y, y, y);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    sr_t = AST::brother(*sr_t);
  }
  if (no_parms > 0) {
    KCHARP sub_rule_stk_parms_end =
    "␣␣};";
    Op_str ≪ sub_rule_stk_parms_end ≪ endl;
    KCHARP stk_frame_equate =
    "␣␣SF*␣sf␣=␣(SF*)rule_info__.parser__->parse_stack__.sf_by_top(%i);";
    int x = sprintf(big_buf_, stk_frame_equate, no_parms);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

This code is used in section 146.

148. `OP_GRAMMAR_CPP` implementation.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_GRAMMAR_CPP(TOKEN_GAGGLE & Error_queue)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    string fn(fsm_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_fsmimp;
    std::ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base::out | ios::trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_fsmcpp_filename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*fsm_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_GRAMMAR_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    fsm_cpp_includes(Op_file);
    using_ns_for_fsm_cpp(Op_file);
    rules_reuse_table_implementation(Op_file);
    fsm_class_implementation(Op_file);
    rules_subrules_implementations(Op_file);
    Op_file.close( );
  }
```

149. Template of fsm sym file implementation: OP_GRAMMAR_SYM.
reduce_rhs_of_rule implementation and possible thread procedure. Back in time when c++ compilers were aborting i needed to split the size of the emitted grammar's source file so that digestion instead of indigestion took place. *reduce_rhs_of_rule* should be part of the "fsm" cpp file. I leave it as is cuz i'm lazy and makes reading of the outfiles easier.

1) Comments — file name, time and date info
2) Include of grammar's header
3) possible "user-imp-sym" code
4) using namespaces
5) Possible thread procedure implementation
6) Fsm class *reduce_rhs_of_rule* implementation
7) grammar's rules / subrules implementations

150. *user_imp_sym*.
There are times when circularity hits u in include definitions by the compiler's preprocessor. This allows the grammar writer to roll-his-own by injection points.
Grammar Directive: "user-imp-sym" for the emitted "xxxsym.cpp" code module.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_imp_sym(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_user_imp_sym);
    if (i ≠ dir_map→end( )) {
      T_user_imp_sym* ui = (T_user_imp_sym*)i->second;
      Op_str ≪ ui→syntax_code( )→syntax_code( )→c_str( );
      Op_str ≪ endl;
    }
  }
```

151. *thread_implementation*.
Some naming conventions:
ssPARSE_TABLE is referenced out of the "wpp_core.h" file. This was my way to import info into a canned thread code body.

```
⟨ accrue source for emit 8 ⟩ +≡
  void thread_implementation(std::ofstream & Op_str)
  {
    T_parallel_parser_phrase * pp_ph = O2_PP_PHASE;
    if (pp_ph ≡ 0) {
      Op_str ≪ "//␣monolithic␣grammar␣---␣no␣thread" ≪ endl;
      return;
    }
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP thread_definition =
    "yacco2::THR␣_YACCO2_CALL_TYPE\n"
    "%s::%s(yacco2::Parser*␣Caller_pp){\n"
    "␣␣yacco2::Thread_entry&␣pp_thread_entry␣=␣I%s;\n"
    "␣␣%s::%s␣%s_;//␣parallel-parser's␣parse␣table\n""#define␣ssPARSE_TABLE␣%s_\n"
    "#include␣\"wpp_core.h\"\n"
    "}";
    int x = sprintf(big_buf_, thread_definition, fsm_ph→namespace_id( )→identifier( )→c_str( ),
        pp_ph→pp_funct( )→identifier( )→identifier( )→c_str( ),
        pp_ph→pp_funct( )→identifier( )→identifier( )→c_str( ), fsm_ph→namespace_id( )→identifier( )→c_str( ),
        fsm_class→identifier( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
        fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP proc_thread_definition =
    "THR_result␣_YACCO2_CALL_TYPE\n"
    "%s::PROC_%s(yacco2::Parser*␣Caller_pp){\n"
    "␣␣char␣called_proc_name[]␣=␣\"PROC_%s\";\n""␣␣static␣bool␣one_time(false);\n"
    "␣␣static␣%s::%s*␣%s_%s_(0);//␣parallel-parser's␣fsm␣table\n"
    "␣␣static␣Parser*␣%s_%s_parser(0);\n"
    "␣␣Parser*␣proc_parser(0);\n"
    "␣␣if(one_time␣==␣false){\n""␣␣␣␣one_time␣=␣true;\n"
    "␣␣␣␣%s_%s_␣=␣new␣%s::%s();//␣parallel-parser's␣fsm␣table\n""␣␣␣␣%s_%s_parser␣=␣\
        new␣Parser(*%s_%s_,Caller_pp);\n"
    "␣␣}\n"
    "␣proc_parser␣=␣%s_%s_parser;\n"
    "#include␣\"wproc_pp_core.h\"\n"
    "}";
    x = sprintf(big_buf_, proc_thread_definition, fsm_ph→namespace_id( )→identifier( )→c_str( ),
        pp_ph→pp_funct( )→identifier( )→identifier( )→c_str( ),
        pp_ph→pp_funct( )→identifier( )→identifier( )→c_str( )      /* called proc name */
```

```
    , fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
        fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( )
      /* static fsm */
    , fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( )
      /* static parser */
    , fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
        fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( )
      /* new fsm */
    , fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
        fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( )
      /* new parser */
    , fsm_ph→namespace_id( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( )
      /* set as proc_parser */
    );
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

152. *fsm_reduce_rhs_of_rule_implementation*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsm_reduce_rhs_of_rule_implementation(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP reduce_rhs_of_rule_hdr =
    "void␣\n"
    "%s::reduce_rhs_of_rule\n"
    "␣␣␣␣␣(yacco2::UINT␣Sub_rule_no,yacco2::Rule_s_reuse_entry**␣Recycled_rule){\n"
    "␣␣␣␣int␣reducing_rule␣=␣rhs_to_rules_mapping_[Sub_rule_no];\n"
    "␣␣␣␣Per_rule_s_reuse_table*␣rule_reuse_tbl_ptr␣=␣\n"
    "␣␣␣␣␣␣␣␣␣fsm_rules_reuse_table.per_rule_s_table_[reducing_rule];\n"
    "␣␣␣␣Rule_s_reuse_entry*␣re(0);\n"
    "␣␣␣␣find_a_recycled_rule(rule_reuse_tbl_ptr,&re);\n""␣␣␣(*Recycled_rule)\
        ␣=␣re;\n""␣␣␣fnd_re:␣switch␣(Sub_rule_no){";
    int x = sprintf(big_buf_, reduce_rhs_of_rule_hdr, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    ⟨ gen rules's subrules case stmts 153 ⟩;
  }
```

153. Gen rules's subrules case statements.

```
⟨ gen rules's subrules case stmts 153 ⟩ ≡
  RULE_DEFS_TBL_ITER_type i = rules_ph→crt_order( )→begin( );
  RULE_DEFS_TBL_ITER_type ie = rules_ph→crt_order( )→end( );
  for ( ; i ≠ ie; ++i) {
    rule_def * rd = *i;
    ⟨ gen subrule case stmt 154 ⟩;
  }
  KCHARP default_case_and_end_sw =
  "␣␣␣␣default:␣return;\n"
  "␣␣␣}";
  Op_str ≪ default_case_and_end_sw ≪ endl;
  KCHARP end_rtn =
  "}";
  Op_str ≪ end_rtn ≪ endl;
```

This code is used in section 152.

154. Gen subrule case statement.
Optimization: Place the subrule's number of elements within the "xxxsym" module instead of within its subrule. This allows the saving of a procedure call if there is no syntax directed code for the subrule procedure. Now for the rule optimization:
1) the case statement determines whether the rule should be created or use the recycled one from a previous invocation. The wrinkle is to determine whether a ctor type call is needed to re-initialize the recycled rule. This depends on whether a "constructor" directive is present in the rule's definition. If so, then call a pseudo ctor initialization procedure or just include the "constructor code" in the commented "if ctor present commented below".

Review of Rule's emit code:
The parser sees a string of symbols and determines that a reduce is to take place. So the subrule to be executed within the rule determines its rule. So the rule is either created by newing for the first time or drawn from its recycled pool.

Now the interesting sequence:
1) Create/Recycled Rule for the subrule to run within
2) Execute the Rule's contructor directive if present in the rule definition
3) Execute the subrule's code
4) Execute the Rule's op directive if present int its definition

The constructor-op-destructor directives sequence of a fsm has a different meaning within a rule's sequence. This is due to a rule can be created many times thru left recursion so what does the constructor directive mean — run only once? It takes on more of an intialization role to the newly created rule or its recycled ego. The neat thing is the rhs aka subrule to be reduced now becomes the op directive to the rhs of a rule and the op directive of the rule now follows it in execution. This allows one to post evaluate what the subrule was and to react accordingly.

```
⟨ gen subrule case stmt 154 ⟩ ≡
  const char *rule_s_ctor = "//␣no␣rule's␣constructor␣directive";
  T_rule_lhs_phrase * rlhs = rd→rule_lhs( );
  if (rlhs ≠ 0) {
    SDC_MAP_type * rdlhs_map = rlhs→lhs_directives_map( );
    if (rdlhs_map ≠ 0) {
      SDC_MAP_ITER_type kkk = rdlhs_map→find(SDC_constructor);
      if (kkk ≠ rdlhs_map→end( )) {
        T_constructor*␣cc␣=␣(T_constructor*)kkk->second;
        rule_s_ctor = "sym->ctor();\n";
      }
    }
  }
  T_subrules_phrase * sr_ph = rd→subrules( );
  SUBRULE_DEFS_ITER_type j = sr_ph→subrules( )→begin( );
  SUBRULE_DEFS_ITER_type je = sr_ph→subrules( )→end( );
  KCHARP case_stmt_parta =
  "␣␣␣␣case␣rhs%i_%s_:{\n"
  "␣␣␣␣␣␣␣␣%s*␣sym;\n"
  "␣␣␣␣␣if(re->rule_␣==␣0){\n"
  "␣␣␣␣␣␣␣␣sym␣=␣new␣%s(parser__);\n"
  "␣␣␣␣␣␣␣␣re->rule_␣=␣sym;\n"
  "␣␣␣␣␣}else{\n"
  "␣␣␣␣␣␣␣␣sym␣=␣(%s*)re->rule_;\n"
  "␣␣␣␣␣}\n"
  "␣␣␣␣␣%s\n"
  "␣␣␣␣␣(*Recycled_rule)->rule_␣=␣sym;\n"
```

```
  "␣␣␣␣␣␣sym->rule_info__.rhs_no_of_parms__␣=␣%i;\n"
  "␣␣␣␣␣\n";
  KCHARP case_stmt_partb = "␣␣␣␣␣␣sym->sr%i();";
  KCHARP case_stmt_partc = "␣␣␣␣␣␣return;}";
  for ( ; j ≠ je; ++j) {
    T_subrule_def * srd = *j;
    int no_parms = srd→no_of_elems( ) − 1;      /* remove eos */
      /* see if subrule is a thread or proc call so also remove its eos */
    AST * sr_t = srd→subrule_s_tree( );
    set<int>␣eos_filter;
    eos_filter.insert(T_Enum :: T_T_called_thread_eosubrule_);
    eos_filter.insert(T_Enum :: T_T_null_call_thread_eosubrule_);
    tok_can_ast_functor walk_the_plank_mate;
    ast_prefix_1forest eos_walk(*sr_t, &walk_the_plank_mate, &eos_filter, ACCEPT_FILTER);
    tok_can < AST *> eos_can(eos_walk);
    for (int xxx = 0; eos_can.operator[](xxx) ≠ yacco2 :: PTR_LR1_eog__; ++xxx) ;
    if (eos_can.empty( ) ≡ false) --no_parms;
    int x = sprintf(big_buf_, case_stmt_parta, srd→subrule_no_of_rule( ), rd→rule_name( )→c_str( ),
        rd→rule_name( )→c_str( ), rd→rule_name( )→c_str( ), rd→rule_name( )→c_str( ), rule_s_ctor, no_parms);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    SDC_MAP_type * sdrmap = srd→subrule_directives( );
    if (¬sdrmap→empty( )) {      /* any syntax directed code, call it */
      int x = sprintf(big_buf_, case_stmt_partb, srd→subrule_no_of_rule( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    if (rd→rule_lhs( ) ≠ 0) {
      SDC_MAP_type * xxrdmap = rd→rule_lhs( )→lhs_directives_map( );
      if (xxrdmap ≠ 0) {
        SDC_MAP_ITER_type kkk = xxrdmap→find(SDC_op);
        if (kkk ≠ xxrdmap→end( )) {
          Op_str ≪ "␣␣␣␣␣␣sym->op();" ≪ endl;
        }
      }
    }
    Op_str ≪ case_stmt_partc ≪ endl;
  }
```

This code is used in section 153.

155. OP_GRAMMAR_SYM implementation.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_GRAMMAR_SYM(TOKEN_GAGGLE & Error_queue)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    string fn(fsm_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_fsmsym;
    std::ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base::out | ios::trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_fsmsym_filename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*fsm_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_GRAMMAR_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    user_imp_sym(Op_file);
    fsm_cpp_includes(Op_file);
    using_ns_for_fsm_cpp(Op_file);
    thread_implementation(Op_file);
    fsm_reduce_rhs_of_rule_implementation(Op_file);
    Op_file.close( );
  }
```

156. Template of fsm tbl file implementation: OP_GRAMMAR_TBL.
Emit the grammar's lr1 states tables, and threads for dispatch tables along with their arbitrators.
1) Comments — file name, time and date info
2) Include of grammar's header
3) possible "user-imp-tbl" code
4) using namespaces
5) lookahead sets
6) lr states externs and thread table definitions / implementations
7) lr states definitions / implementations

Please see O_2's documentation describing the following tables and their layouts:
1) State
2) Shift_entry
3) Reduce_tbl
4) State_s_thread_tbl
5) Thread_entry

157. *user_imp_tbl*.
There are times when circularity hits u in include definitions by the compiler's preprocessor. This allows the grammar writer to roll-his-own by injection points.
Grammar Directive: "user-imp-tbl" for the emitted "xxxtbl.cpp" code module.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_imp_tbl(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    SDC_MAP_type * dir_map = fsm_class→directives_map( );
    SDC_MAP_ITER_type i = dir_map→find(SDC_user_imp_tbl);
    if (i ≠ dir_map→end( )) {
      T_user_imp_tbl*␣ui␣=␣(T_user_imp_tbl*)i->second;
      Op_str ≪ ui→syntax_code( )→syntax_code( )→c_str( );
      Op_str ≪ endl;
    }
  }
```

158. *output_la_sets*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_la_sets(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    COMMON_LA_SETS_ITER_type i = COMMON_LA_SETS.begin( );
    COMMON_LA_SETS_ITER_type ie = COMMON_LA_SETS.end( );
    for (int idx = 0; i ≠ ie; ++i, ++idx) {
      LA_SET_type * la_set = *i;
      ⟨ list literal entries of la set 159 ⟩;
      ⟨ emit la set 160 ⟩;
    }
  }
```

159. List literal entries of la set.
Leave T traces of what makes up the compressed bit table.

```
⟨ list literal entries of la set 159 ⟩ ≡
  KCHARP la_set_entry_literal =
  "//␣%s";
  LA_SET_ITER_type j = la_set→begin( );
  LA_SET_ITER_type je = la_set→end( );
  for ( ; j ≠ je; ++j) {
    T_in_stbl * tsym = *j;
    int x = sprintf(big_buf_, la_set_entry_literal, tsym→t_def( )→classsym( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

This code is used in section 158.

160. Emit la set.
Sets are structured as a balanced table in bit partitions ascending order. Each record is a paired structure of partition number with its bits sized by `NO_BITS_PER_SET_PARTITION`. The overall table contains the number of entries so that a "bsearch' can take place. Remember each T is ordered by its enumerate from 0..x and due to modulo arithmetic each bit entry is positioned in right-to-left order within the bit entry.

```
⟨ emit la set 160 ⟩ ≡
  KCHARP la_set_hdr =
  "yacco2::UCHAR␣LA%i_%s[]␣={";
  int x = sprintf(big_buf_, la_set_hdr, idx + 1      /* name given to la set uses the set no relative to 1 */
  , fsm_class→identifier( )→identifier( )→c_str( ));
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
  ⟨ build la set's bit table 161 ⟩;
  ⟨ output the la set's compressed bits 162 ⟩;
```

This code is used in section 158.

161. Build la set's bit table.

```
⟨ build la set's bit table 161 ⟩ ≡
  BIT_MAP_type paired_set;
  LA_SET_ITER_type k = la_set→begin( );
  LA_SET_ITER_type ke = la_set→end( );
  int en_no(−1);
  for ( ; k ≠ ke; ++k) {      /* walk the items in set */
    T_in_stbl * T = *k;
    en_no = T→t_def( )→enum_id( );      /* calculate partition no and its set of elements */
    int Q = en_no/NO_BITS_PER_SET_PARTITION;
    int R = en_no % NO_BITS_PER_SET_PARTITION;
    int One(1);
    int E_v = One ≪ R;
    BIT_MAP_ITER_type e = paired_set.find(Q);
    if (e ≡ paired_set.end( )) {
      paired_set[Q] = E_v;
    }
    else {
      int se = e→second;
      int v = se + E_v;
      e→second = v;
    }
  }
```

This code is used in section 160.

162. Output the la set's compressed bits.

```
⟨ output the la set's compressed bits 162 ⟩ ≡
  KCHARP no_la_set_entries =
  "%i";
  x = sprintf(big_buf_, no_la_set_entries, paired_set.size( ));
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
  BIT_MAP_ITER_type kl = paired_set.begin( );
  BIT_MAP_ITER_type kle = paired_set.end( );
  for ( ; kl ≠ kle; ++kl) {
    int part_no = kl→first;
    int e_v = kl→second;
    KCHARP la_set_entry =
    ",%i,%i";
    x = sprintf(big_buf_, la_set_entry, part_no, e_v);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  KCHARP la_set_hdr_end =
  "};";
  Op_str ≪ la_set_hdr_end ≪ endl;
```

This code is used in section 160.

163. *externs_and_thread_tbl_defs*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void externs_and_thread_tbl_defs(std::ofstream & Op_str)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    STATES_ITER_type i = LR1_STATES.begin( );
    STATES_ITER_type ie = LR1_STATES.end( );
    int state_cnt(0);
    for ( ; i ≠ ie; ++i) {
      state * s = *i;
      ++state_cnt;
      KCHARP state_extern =
      "extern␣yacco2::State␣S%i_%s;";
      int x = sprintf(big_buf_, state_extern, state_cnt, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      ⟨ deal with threads in state 164 ⟩;
    }
  }
```

164. Deal with threads in state.
Emit their thread table definitions along with arbitrator code.

```
⟨ deal with threads in state 164 ⟩ ≡
  using namespace NS_yacco2_T_enum;
  if (s→vectored_into_by_elem_sym_ ≡ 0) continue;      /* next state */
  switch (s→vectored_into_by_elem_) {
  case LR1_PARALLEL_OPERATOR:
    {
      ⟨ determine number of threads to call 165 ⟩;
      ⟨ output called threads table def / imp 166 ⟩;
      break;
    }
  default: continue;      /* next state */
  }
```

This code is used in section 163.

165. Determine number of threads to call.
Read the core items of the state. At the same time find out the rule that could have arbitration. Watch out for ||| being a returned T from a called thread. If so this is not your average bear call.

```
⟨ determine number of threads to call 165 ⟩ ≡
  string ar_name;
  rule_def * ar_s_rule_s_name(0);
  int no_of_threads(0);
  S_VECTORS_ITER_type j = s→state_s_vector_.begin( );      /* rtn T's from threads */
  S_VECTORS_ITER_type je = s→state_s_vector_.end( );
  for ( ; j ≠ je; ++j) {      /* read subrules of returned Ts from called threads */
    S_VECTOR_ELEMS_type & elem_list = j→second;
    S_VECTOR_ELEMS_ITER_type k = elem_list.begin( );
    S_VECTOR_ELEMS_ITER_type ke = elem_list.end( );
    for ( ; k ≠ ke; ++k) {      /* walk along the call sequence 1st item is T */
      state_element * se = *k;
      if (se→next_state_element_ ≡ 0) continue;      /* not your thread call but rtned T */
      CAbs_lr1_sym * sym = AST::content(*AST::brother(*se→sr_element_));
        /* not part of state: bypassed */
      if (sym→enumerated_id__ ≠ T_Enum::T_T_called_thread_eosubrule_) continue;
      rule_def * rd = se→subrule_def_→its_rule_def( );
      RULES_HAVING_AR_ITER_type ai = RULES_HAVING_AR.find(rd);
      if (ai ≠ RULES_HAVING_AR.end( )) {
        ar_s_rule_s_name = rd;
      }
      ++no_of_threads;
    }
  }
  if (ar_s_rule_s_name ≠ 0) {
    ar_name += "AR_";
    ar_name += ar_s_rule_s_name→rule_name( )→c_str( );
    s→arbitrator_name_ = new string(ar_name);      /* add arbitration code to the state */
  }
```

This code is used in section 164.

166. Output called threads table def / imp.
To get the called threads for the dispatch table, one must look 1 state ahead cuz this state is the returned T from a called thread expression. Remember there are 2 types of thread call expressions:
1) ||| T NS_xxx::TH_eol — real called thread
2) ||| T NULL — expr to field a returned T from 1)

```
⟨ output called threads table def / imp 166 ⟩ ≡
  if (no_of_threads > 0) {
    KCHARP state_s_thread_tbl_def =
    "struct␣S%ittd_%s{\n"
    "␣yacco2::USINT␣no_entries_;\n"
    "␣yacco2::Type_pp_fnct_ptr␣ar_fnct_ptr_;\n"
    "␣yacco2::ULINT␣(*thd_id_bit_map_ptr__)[];\n"
    "␣yacco2::Thread_entry*␣thread_entries_[%i];\n"
    "};";
    int x = sprintf(big_buf_, state_s_thread_tbl_def, state_cnt, fsm_class→identifier()→identifier()→c_str(),
        no_of_threads);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    ⟨ determine if there is a rule's arbitrator to call 168 ⟩;
    KCHARP state_s_thread_tbl_imp =
    "S%ittd_%s␣S%itt_%s␣=␣{\n"
    "␣␣%i␣//␣no␣of␣threads\n"
    "␣,%s␣//AR_rulename␣or␣0\n"
    "␣,0//␣ptr␣to␣thread␣id␣bit␣map";
    x = sprintf(big_buf_, state_s_thread_tbl_imp, state_cnt, fsm_class→identifier()→identifier()→c_str(),
        state_cnt, fsm_class→identifier()→identifier()→c_str(), no_of_threads, ar_name.c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    j = s→state_s_vector_.begin();      /* read core items of state */
    je = s→state_s_vector_.end();
    for ( ; j ≠ je; ++j) {      /* read subrules of returned Ts from called threads */
      S_VECTOR_ELEMS_type & elem_list = j→second;
      S_VECTOR_ELEMS_ITER_type k = elem_list.begin();
      S_VECTOR_ELEMS_ITER_type ke = elem_list.end();
      for ( ; k ≠ ke; ++k) {      /* walk along call sequence 1st item is T */
        state_element * se = *k;
        if (se→next_state_element_ ≡ 0) continue;      /* not thd call but */
        CAbs_lr1_sym * sym = AST::content(*AST::brother(*se→sr_element_));
          /* not part of state: bypassed */
        if (sym→enumerated_id__ ≡ T_Enum::T_T_called_thread_eosubrule_) {
          T_called_thread_eosubrule*␣called_th␣=␣(T_called_thread_eosubrule*)sym;
          KCHARP state_s_thread_tbl_entry =
          "␣,(yacco2::Thread_entry*)&I%s";
          int x = sprintf(big_buf_, state_s_thread_tbl_entry,
              called_th→called_thread_name()→identifier()→c_str());
          Op_str.write(big_buf_, x);
          Op_str ≪ endl;
        }
      }
    }
    KCHARP state_s_thread_tbl_end =
```

```
    "};";
    Op_str ≪ state_s_thread_tbl_end ≪ endl;
  }
```

This code is used in section 164.

167. Output called procedure table def / imp.
To get the called procedure for the dispatch table, one must look 2 states ahead cuz this state is the `|t|` operator. So go to the returned T state and read its core items. Remember there are 2 types of thread call expressions:

1) `|t|` T PROC_TH_eol — called procedure
2) `|t|` T NULL — expr to field a returned T from 1)

```
⟨ output called procedure table def / imp 167 ⟩ ≡
  S_VECTORS_ITER_type j = se→goto_state_→state_s_vector_.begin( );      /* rtn T's from threads */
  S_VECTORS_ITER_type je = se→goto_state_→state_s_vector_.end( );
  for ( ; j ≠ je; ++j) {      /* read subrules of returned Ts from called procedure */
    S_VECTOR_ELEMS_type & elem_list = j→second;
    S_VECTOR_ELEMS_ITER_type k = elem_list.begin( );
    S_VECTOR_ELEMS_ITER_type ke = elem_list.end( );
    for ( ; k ≠ ke; ++k) {      /* walk along call sequence 1st item is T */
      state_element * se = *k;
      if (se→next_state_element_ ≡ 0) continue;      /* not thd call but */
      AST * bypassed_thd_eos_t = AST::brother(*se→sr_element_);
      CAbs_lr1_sym * sym = AST::content(*bypassed_thd_eos_t);
      if (sym→enumerated_id__ ≡ T_Enum::T_T_called_thread_eosubrule_) {
        T_called_thread_eosubrule* called_th = (T_called_thread_eosubrule*)sym;
        KCHARP state_s_thread_tbl_entry =
        ",(yacco2::Type_pc_fnct_ptr)&%s::%s";
        int x = sprintf(big_buf_, state_s_thread_tbl_entry, called_th→ns( )→identifier( )→c_str( ),
            called_th→called_thread_name( )→identifier( )→c_str( ));
        Op_str.write(big_buf_, x);
      }
    }
  }
```

This code is used in section 196.

168. determine if there is a rule's arbitrator to call.
This conditions on whether arbritrator code is present in some spawning rules. Backtrack to the calling state to figure out whether there is a rule that contains code. If not then set the rule name to "0" meaning nada to arbitrate. If found, the arbitrator's procedure's name is AR_rulename.

```
⟨ determine if there is a rule's arbitrator to call 168 ⟩ ≡
  if (ar_name.empty( ) ≡ true) ar_name += "0";
```

This code is used in section 166.

169. *determine_shift_element_name*.
For your eyes only...

```
⟨ accrue source for emit 8 ⟩ +≡
  const char *determine_shift_element_name(CAbs_lr1_sym * Sym)
  {
    switch (Sym→enumerated_id__) {
    case T_Enum :: T_T_terminal_def_:
      {
        T_terminal_def* td = (T_terminal_def*)Sym;
        return td→classsym( )→c_str( );
      }
    case T_Enum :: T_rule_def_:
      {
        rule_def* rd = (rule_def*)Sym;
        return rd→rule_name( )→c_str( );
      }
    }
    return "element not found";
  }
```

170. Output 1st state's shift table.
Hardwire "start rule" accept shift. The entries are in ascending order of their enumerates. Make sure the hardwired "start rule" shift entry comes right after the T vocabulary cuz of balanced table requirement. One wrinkle is if the 1st state does not contain any rules — only terminals. So check at the end of the generation for this situation to gen the shifted start rule.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_1st_state_s_shift_table(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    rule_def * s_rule = (*rules_ph→crt_order( ))[0];
    int s_rule_enum_no = s_rule→enum_id( );
    char big_buf_[BIG_BUFFER_32K];
    S_VECTORS_ITER_type svi;
    S_VECTORS_ITER_type svie;
    ⟨ determine number of shifts 176 ⟩;
    ++no_of_shift_items;      /* add start rule shift */
    bool first_or_2nd_prt(false);
    bool accept_prted(false);
    ⟨ output start of shift definition 173 ⟩;
    svi = State.state_s_vector_.begin( );
    svie = State.state_s_vector_.end( );
    for ( ; svi ≠ svie; ++svi) {      /* walk the vectors */
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int cur_se_enum_no = svi→first;
      ⟨ bypass reduces 178 ⟩;
      int goto_state_no = se→goto_state_→state_no_;
      const char *shift_elem_literal = determine_shift_element_name(se→sr_def_element_);
      if (cur_se_enum_no < s_rule_enum_no) {
        ⟨ print 1st or 2nd shift entry 171 ⟩;
      }
      else {
        if (accept_prted ≡ false) {
          accept_prted = true;
          ⟨ print shift accept entry 172 ⟩;
        }
        ⟨ print 1st or 2nd shift entry 171 ⟩;      /* balance of rules shifted */
      }
    }
    if (accept_prted ≡ false) {
      accept_prted = true;
      ⟨ print shift accept entry 172 ⟩;
    }
    ⟨ output end of shift table entries 174 ⟩;
  }
```

171. Print 1st or 2nd shift entry.

```
⟨ print 1st or 2nd shift entry 171 ⟩ ≡
  if (first_or_2nd_prt ≡ false) {
    first_or_2nd_prt = true;
    KCHARP imp_of_state_s_shift_1st_entry =
    "␣␣␣{%i,(State*)&S%i_%s}␣//␣shift␣sym:␣%s";
    int x = sprintf(big_buf_, imp_of_state_s_shift_1st_entry, cur_se_enum_no, goto_state_no,
        fsm_class→identifier( )→identifier( )→c_str( ), shift_elem_literal);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  else {
    KCHARP imp_of_state_s_shift_2nd_entry =
    "␣␣␣,{%i,(State*)&S%i_%s}␣//␣shift␣sym:␣%s";
    int x = sprintf(big_buf_, imp_of_state_s_shift_2nd_entry, cur_se_enum_no, goto_state_no,
        fsm_class→identifier( )→identifier( )→c_str( ), shift_elem_literal);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

This code is used in sections 170 and 175.

172. Print shift accept entry.

The *first_or_2nd_prt* makes sure the following comma is printed properly.

```
⟨ print shift accept entry 172 ⟩ ≡
  if (first_or_2nd_prt ≡ false) {
    first_or_2nd_prt = true;
    KCHARP imp_of_state_s_shift_1st_entry =
    "␣␣␣{%i,(State*)&S%i_%s}␣//␣accept␣sym:␣%s";
    int x = sprintf(big_buf_, imp_of_state_s_shift_1st_entry, s_rule_enum_no, 1,
        fsm_class→identifier( )→identifier( )→c_str( ), s_rule→rule_name( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  else {
    KCHARP imp_of_state_s_shift_1st_entry =
    "␣␣␣,{%i,(State*)&S%i_%s}␣//␣accept␣sym:␣%s";
    int x = sprintf(big_buf_, imp_of_state_s_shift_1st_entry, s_rule_enum_no, 1,
        fsm_class→identifier( )→identifier( )→c_str( ), s_rule→rule_name( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

This code is used in section 170.

173. Output start of shift definition.

```
⟨ output start of shift definition 173 ⟩ ≡
  KCHARP def_of_state_s_shift_entries =
  "struct␣S%istd_%s{\n"
  "␣␣yacco2::USINT␣no_entries_;\n"
  "␣␣yacco2::Shift_entry␣shift_entries_[%i];\n"
  "};";
  int x = sprintf(big_buf_, def_of_state_s_shift_entries, State.state_no_,
      fsm_class→identifier()→identifier()→c_str(), no_of_shift_items);
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
  KCHARP imp_of_state_s_shift_entries_begin =
  "S%istd_%s␣S%ist_%s␣=␣{\n"
  "␣%i\n"      /* no of entries */
  "␣,\n"
  "␣␣{//␣start␣of␣table";
  x = sprintf(big_buf_, imp_of_state_s_shift_entries_begin, State.state_no_,
      fsm_class→identifier()→identifier()→c_str(), State.state_no_,
      fsm_class→identifier()→identifier()→c_str(), no_of_shift_items);
  Op_str.write(big_buf_, x);
  Op_str ≪ endl;
```

This code is used in sections 170 and 175.

174. Output end of shift table entries.

```
⟨ output end of shift table entries 174 ⟩ ≡
  KCHARP imp_of_shift_entries_end =
  "␣␣}//␣end␣of␣shift␣table\n"
  "};";
  Op_str ≪ imp_of_shift_entries_end ≪ endl;
```

This code is used in sections 170 and 175.

175. Output all others state's shift table.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_all_others_state_s_shift_table(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    bool first_or_2nd_prt(false);
    S_VECTORS_ITER_type svi;
    S_VECTORS_ITER_type svie;
    ⟨ determine number of shifts 176 ⟩;
    if (no_of_shift_items) {
      ⟨ output start of shift definition 173 ⟩;
      S_VECTORS_ITER_type svi = State.state_s_vector_.begin( );
      S_VECTORS_ITER_type svie = State.state_s_vector_.end( );
      for ( ; svi ≠ svie; ++svi) {      /* walk the vectors */
        S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
        state_element * se = *seli;
        int cur_se_enum_no = svi→first;
        ⟨ bypass reduces 178 ⟩;
        int goto_state_no = se→goto_state_→state_no_;
        const char *shift_elem_literal = determine_shift_element_name(se→sr_def_element_);
        ⟨ print 1st or 2nd shift entry 171 ⟩;
      }
      ⟨ output end of shift table entries 174 ⟩;
    }
  }
```

176. Determine number of shifts.

```
⟨ determine number of shifts 176 ⟩ ≡
  int no_of_shift_items(0);
  svi = State.state_s_vector_.begin( );
  svie = State.state_s_vector_.end( );
  for ( ; svi ≠ svie; ++svi) {      /* walk the vectors */
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int cur_se_enum_no = svi→first;
    ⟨ bypass reduces 178 ⟩;
    ++no_of_shift_items;
  }
```

This code is used in sections 170, 175, and 193.

177. Bypass meta T.
The meta T are bypassed as each are implemented separately as individual entries within the state table. The real truth is the individual entries in the state are for a fast determination of their presence to sequentially attempt the fsm's order of operations: parallelism — thread calls by ||| expressions, specific shift on current terminal token, invisible shift – |.| implicit espilon, all-shift | + |, and reduce. Their presence in the shift table is REQUIRED for the shift operator. On 1 hand lookup speed by separate state's entries is fast but when it comes to shifting it's slow to sequentially check for their individual presence.

So below is my optimized mistake. THIS CODE IS NOW VOID! Only here for removal justifications. Meta T returned from a called thread are exempt and are part of the shift table. Why? They are treated like any other T returned from a called thread. The returned T is shifted into its reducing state that reduces the called-thread expression.

```
⟨ bypass meta T 177 ⟩ ≡
  switch (cur_se_enum_no) {
  case LR1_PARALLEL_OPERATOR:
    {
      if (se→previous_state_element_ ≡ 0) continue;      /* bypass start of call thd expr */
      if (      /* is ||| returned from thread */
      se→previous_state_element_→its_enum_id_ ≡ LR1_PARALLEL_OPERATOR) break;
          /* do not bypass: part of shift tbl */
      continue;      /* bypass */
    }
  case LR1_FSET_TRANSIENCE_OPERATOR:
    {
      if (se→previous_state_element_ ≡ 0) continue;      /* bypass start of call proc expr */
      if (      /* is TRAshift returned from thread */
      se→previous_state_element_→its_enum_id_ ≡ LR1_PARALLEL_OPERATOR) break;
          /* do not bypass: part of shift tbl */
      continue;      /* bypass */
    }
  case LR1_INVISIBLE_SHIFT_OPERATOR:
    {
      if (se→previous_state_element_ ≡ 0) continue;      /* bypass dealt a separate state entry */
      if (      /* is |.| returned from thread */
      se→previous_state_element_→its_enum_id_ ≡ LR1_PARALLEL_OPERATOR) break;
          /* do not bypass: part of shift tbl */
      continue;      /* bypass */
    }
  case LR1_ALL_SHIFT_OPERATOR:
    {
      if (se→previous_state_element_ ≡ 0) continue;
      if (      /* is | + | returned from thread */
      se→previous_state_element_→its_enum_id_ ≡ LR1_PARALLEL_OPERATOR) break;
          /* do not bypass: part of shift tbl */
      continue;      /* bypass */
    }
  case LR1_QUESTIONABLE_SHIFT_OPERATOR:
    {
      if (se→previous_state_element_ ≡ 0) continue;
      if (      /* is |?| returned from thread */
      se→previous_state_element_→its_enum_id_ ≡ LR1_PARALLEL_OPERATOR) break;
          /* do not bypass: part of shift tbl */
      continue;      /* bypass */
```

```
    }
  }
```

178. Bypass reduces.

```
⟨ bypass reduces 178 ⟩ ≡
  if (cur_se_enum_no ≥ 0) {      /* not a reduce T */
  }
  else {
    switch (cur_se_enum_no) {
    case −T_Enum :: T_T_eosubrule_:
      {
        if (se→next_state_element_ ≡ 0) continue;      /* real reduce: epsilon so bypass */
        if (se→previous_state_element_ ≡ 0) continue;      /* real reduce: epsilon so bypass */
        break;
      }
    default:
      {
        break;      /*not a reduce */
      }
    }
  }
```

This code is used in sections 170, 175, and 176.

179. Output meta shifts separately.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_meta_shifts_separately(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    S_VECTORS_ITER_type svi = State.state_s_vector_.find(LR1_PARALLEL_OPERATOR);
    S_VECTORS_ITER_type svie = State.state_s_vector_.end( );
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP parallel_entry =
      "yacco2::Shift_entry␣S%ipse_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, parallel_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ),
          LR1_PARALLEL_OPERATOR, goto_state_no, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_INVISIBLE_SHIFT_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP invisible_shift_entry =
      "yacco2::Shift_entry␣S%iise_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, invisible_shift_entry, State.state_no_,
          fsm_class→identifier( )→identifier( )→c_str( ), LR1_INVISIBLE_SHIFT_OPERATOR, goto_state_no,
          fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_ALL_SHIFT_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP all_shift_entry =
      "yacco2::Shift_entry␣S%iase_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, all_shift_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ),
          LR1_ALL_SHIFT_OPERATOR, goto_state_no, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_FSET_TRANSIENCE_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
```

```
    state_element * se = *seli;
    int goto_state_no = se->goto_state_->state_no_;
    KCHARP procedure_call_entry =
    "yacco2::Shift_entry␣S%ipcse_%s␣=␣{%i,(State*)&S%i_%s};";
    int x = sprintf(big_buf_, procedure_call_entry, State.state_no_,
        fsm_class->identifier()->identifier()->c_str(), LR1_FSET_TRANSIENCE_OPERATOR, goto_state_no,
        fsm_class->identifier()->identifier()->c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
  svi = State.state_s_vector_.find(LR1_QUESTIONABLE_SHIFT_OPERATOR);
  if (svi != svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi->second.begin();
    state_element * se = *seli;
    int goto_state_no = se->goto_state_->state_no_;
    KCHARP questionable_shift_entry =
    "yacco2::Shift_entry␣S%iqse_%s␣=␣{%i,(State*)&S%i_%s};";
    int x = sprintf(big_buf_, questionable_shift_entry, State.state_no_,
        fsm_class->identifier()->identifier()->c_str(), LR1_QUESTIONABLE_SHIFT_OPERATOR, goto_state_no,
        fsm_class->identifier()->identifier()->c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
}
```

180. Output state's reduced table.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_state_s_reduced_table(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph->fsm_class_phrase();
    char big_buf_[BIG_BUFFER_32K];
    ⟨ determine number of reduces 181 ⟩;
    ⟨ output reduced table def / imp 182 ⟩;
  }
```

181. Determine number of reduces.
Each shift / reduce entry in the state contains a list of subrules partaking in the activity. For the reduce activity this is why the number of entries in the list is accumulated. More than 1 entry indicates a reduce / reduce type conflict. As reduced vectors are < 0 and the state's vector is a map ordered on the enumerate value, sequentially reading the map is not an inefficient way to deal with these items as they are the first to be read. The only wrinkle is when a "eosubrule" is returned from a called thread. This is not an end-of-subrule.

```
⟨ determine number of reduces 181 ⟩ ≡
  int no_of_reduces(0);
  S_VECTORS_ITER_type j = State.state_s_vector_.begin( );
  S_VECTORS_ITER_type je = State.state_s_vector_.end( );
  for ( ; j ≠ je; ++j) {      /* read list of subrules per vector */
    int cur_se_enum_no = j→first;
    S_VECTOR_ELEMS_ITER_type seli = j→second.begin( );
    state_element * se = *seli;
    if (cur_se_enum_no ≥ 0) break;      /* not a reduce T */
    switch (cur_se_enum_no) {
    case -T_Enum :: T_T_eosubrule_:
      {
        if (se→next_state_element_ ≡ 0) {      /* real eos */
          no_of_reduces += j→second.size( );
          continue;
        }
      }
    }
  }
```

This code is used in sections 180 and 194.

182. Output reduced table def / imp.

```
⟨ output reduced table def / imp 182 ⟩ ≡
  if (no_of_reduces > 0) {
    KCHARP reduce_def =
    "struct␣S%irtd_%s{\n"
    "␣yacco2::USINT␣no_entries_;\n"
    "␣yacco2::Reduce_entry␣reduce_entries_[%i];\n"
    "};";
    int x = sprintf(big_buf_, reduce_def, State.state_no_, fsm_class→identifier()→identifier()→c_str(),
        no_of_reduces);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    KCHARP reduce_imp_begin =
    "S%irtd_%s␣S%irt_%s␣=␣{\n"
    "␣%i\n"
    "␣,\n"
    "␣{//␣start␣of␣table";
    x = sprintf(big_buf_, reduce_imp_begin, State.state_no_, fsm_class→identifier()→identifier()→c_str(),
        State.state_no_, fsm_class→identifier()→identifier()→c_str(), no_of_reduces);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    ⟨ output reduce entries 183 ⟩;
    KCHARP reduce_imp_end =
    "␣}//␣end␣of␣reduce␣table\n"
    "};";
    Op_str ≪ reduce_imp_end ≪ endl;
  }
```

This code is used in section 180.

183. Output reduce entries.

```
⟨ output reduce entries 183 ⟩ ≡
  j = State.state_s_vector_.begin( );      /* read core items of state */
  je = State.state_s_vector_.end( );
  int reduce_no(−1);
  for ( ; j ≠ je; ++j) {
    int cur_se_enum_no = j→first;
    if (cur_se_enum_no ≥ 0) break;
    S_VECTOR_ELEMS_type & elem_list = j→second;
    S_VECTOR_ELEMS_ITER_type k = elem_list.begin( );
    S_VECTOR_ELEMS_ITER_type ke = elem_list.end( );
    for ( ; k ≠ ke; ++k) {      /* la sets */
      ++reduce_no;
      state_element * se = *k;
      if (reduce_no ≡ 0) {
        KCHARP reduce_imp_1st_entry =
        "␣␣{(Set_tbl*)&LA%i_%s,%s::rhs%i_%s_}";
        int x = sprintf(big_buf_, reduce_imp_1st_entry, se→common_la_set_idx_ + 1,
            fsm_class→identifier( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
            se→subrule_def_→subrule_no_of_rule( ), se→subrule_def_→its_rule_def( )→rule_name( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      else {
        KCHARP reduce_imp_2nd_entry =
        "␣␣,{(Set_tbl*)&LA%i_%s,%s::rhs%i_%s_}";
        int x = sprintf(big_buf_, reduce_imp_2nd_entry, se→common_la_set_idx_ + 1,
            fsm_class→identifier( )→identifier( )→c_str( ), fsm_class→identifier( )→identifier( )→c_str( ),
            se→subrule_def_→subrule_no_of_rule( ), se→subrule_def_→its_rule_def( )→rule_name( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
    }
  }
```

This code is used in section 182.

184. Output state's called threads table.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_state_s_called_threads_table(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP call_thread_table_imp =
    "yacco2::Shift_entry␣S%ipse_%s␣=␣\n"
    "{%i,(State*)&S%i_%s};\n";
  }
```

185. Output state's called procedure table.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_state_s_called_procedure_table(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph->fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP call_thread_table_imp =
    "yacco2::Shift_entry␣S%ipcse_%s␣=␣\n"
    "{%i,(State*)&S%i_%s};\n";
  }
```

186. Output meta shifts separately.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_questionable_shift(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    S_VECTORS_ITER_type svi = State.state_s_vector_.find(LR1_PARALLEL_OPERATOR);
    S_VECTORS_ITER_type svie = State.state_s_vector_.end( );
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP parallel_entry =
      "yacco2::Shift_entry␣S%ipse_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, parallel_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ),
          LR1_PARALLEL_OPERATOR, goto_state_no, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_INVISIBLE_SHIFT_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP invisible_shift_entry =
      "yacco2::Shift_entry␣S%iise_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, invisible_shift_entry, State.state_no_,
          fsm_class→identifier( )→identifier( )→c_str( ), LR1_INVISIBLE_SHIFT_OPERATOR, goto_state_no,
          fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_ALL_SHIFT_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
      state_element * se = *seli;
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP all_shift_entry =
      "yacco2::Shift_entry␣S%iase_%s␣=␣{%i,(State*)&S%i_%s};";
      int x = sprintf(big_buf_, all_shift_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ),
          LR1_ALL_SHIFT_OPERATOR, goto_state_no, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    svi = State.state_s_vector_.find(LR1_FSET_TRANSIENCE_OPERATOR);
    if (svi ≠ svie) {
      S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
```

```
    state_element * se = *seli;
    int goto_state_no = se->goto_state_->state_no_;
    KCHARP procedure_call_entry =
    "yacco2::Shift_entry␣S%ipcse_%s␣=␣{%i,(State*)&S%i_%s};";
    int x = sprintf(big_buf_, procedure_call_entry, State.state_no_,
        fsm_class->identifier()->identifier()->c_str(), LR1_FSET_TRANSIENCE_OPERATOR, goto_state_no,
        fsm_class->identifier()->identifier()->c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
  svi = State.state_s_vector_.find(LR1_QUESTIONABLE_SHIFT_OPERATOR);
  if (svi != svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi->second.begin();
    state_element * se = *seli;
    int goto_state_no = se->goto_state_->state_no_;
    KCHARP questionable_shift_entry =
    "yacco2::Shift_entry␣S%iise_%s␣=␣{%i,(State*)&S%i_%s};";
    int x = sprintf(big_buf_, questionable_shift_entry, State.state_no_,
        fsm_class->identifier()->identifier()->c_str(), LR1_QUESTIONABLE_SHIFT_OPERATOR, goto_state_no,
        fsm_class->identifier()->identifier()->c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
}
```

187. Output lr state.
Note, the quotes are needed after the "State's vectored into symbol: ...' cuz the escape ' ' symbol is taken as a macro extension and causes havic with the Sun's compiler (corrected with a following space but Apple's compile... excuse me gnu's compiler has fits with the correct so let quote it.

```
⟨ accrue source for emit 8 ⟩ +≡
  void output_lr_state(std::ofstream & Op_str, state & State)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    T_fsm_class_phrase * fsm_class = fsm_ph→fsm_class_phrase( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP state_entry =
    "yacco2::State␣S%i_%s␣=␣//State's␣vectored␣into␣symbol:␣\"%s\"␣\n""{%i";
    int x = sprintf(big_buf_, state_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ),
        State.entry_symbol_literal( ), State.state_no_);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    ⟨ parallel shift entry 188 ⟩;
    ⟨ all shift entry 189 ⟩;
    ⟨ invisible shift entry 190 ⟩;
    ⟨ procedure call shift entry 191 ⟩;
    ⟨ shift entry 193 ⟩;
    ⟨ reduce entry 194 ⟩;
    ⟨ thread call entry 195 ⟩;
    ⟨ procedure call function entry 196 ⟩;
    ⟨ questionable shift entry 192 ⟩;
    Op_str ≪ "};" ≪ endl;
  }
```

188. Parallel shift entry.

```
⟨ parallel shift entry 188 ⟩ ≡
  S_VECTORS_ITER_type svi = State.state_s_vector_.find(LR1_PARALLEL_OPERATOR);
  S_VECTORS_ITER_type svie = State.state_s_vector_.end( );
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int goto_state_no = se→goto_state_→state_no_;
    KCHARP parallel_entry =
    ",(Shift_entry*)&S%ipse_%s";
    int x = sprintf(big_buf_, parallel_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

189. All shift entry.

```
⟨ all shift entry 189 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_ALL_SHIFT_OPERATOR);
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int goto_state_no = se→goto_state_→state_no_;
    KCHARP all_shift_entry =
    ",(Shift_entry*)&S%iase_%s";
    int x = sprintf(big_buf_, all_shift_entry, State.state_no_, fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

190. Invisible shift entry.

```
⟨ invisible shift entry 190 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_INVISIBLE_SHIFT_OPERATOR);
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int goto_state_no = se→goto_state_→state_no_;
    KCHARP invisible_shift_entry =
    ",(Shift_entry*)&S%iise_%s";
    int x = sprintf(big_buf_, invisible_shift_entry, State.state_no_,
        fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

191. Procedure call shift entry.

```
⟨ procedure call shift entry 191 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_FSET_TRANSIENCE_OPERATOR);
  svie = State.state_s_vector_.end( );
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int goto_state_no = se→goto_state_→state_no_;
    KCHARP procedure_call_entry =
    ",(Shift_entry*)&S%ipcse_%s";
    int x = sprintf(big_buf_, procedure_call_entry, State.state_no_,
        fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

192. Questionable shift entry.

```
⟨ questionable shift entry 192 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_QUESTIONABLE_SHIFT_OPERATOR);
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    int goto_state_no = se→goto_state_→state_no_;
    KCHARP questionable_shift_entry =
    ",(Shift_entry*)&S%iqse_%s";
    int x = sprintf(big_buf_, questionable_shift_entry, State.state_no_,
        fsm_class→identifier( )→identifier( )→c_str( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

193. Shift entry.

```
⟨ shift entry 193 ⟩ ≡
  ⟨ determine number of shifts 176 ⟩;
  if (no_of_shift_items > 0) {
    KCHARP shift_entry =
    ",(Shift_tbl*)&S%ist_%s";
    int x = sprintf(big_buf_, shift_entry, State.state_no_, fsm_class→identifier()→identifier()→c_str());
    Op_str.write(big_buf_, x);
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

194. Reduce entry.

```
⟨ reduce entry 194 ⟩ ≡
  ⟨ determine number of reduces 181 ⟩;
  if (no_of_reduces > 0) {
    KCHARP reduce_entry =
    ",(Reduce_tbl*)&S%irt_%s";
    int x = sprintf(big_buf_, reduce_entry, State.state_no_, fsm_class→identifier()→identifier()→c_str());
    Op_str.write(big_buf_, x);
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

195. Thread call entry.
Watch out for |||being a returned T from a called thread. If so this is not your average bear call.

```
⟨ thread call entry 195 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_PARALLEL_OPERATOR);
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    if (se→next_state_element_→goto_state_ ≡ 0) {
      Op_str ≪ ",0";
    }
    else {
      int goto_state_no = se→goto_state_→state_no_;
      KCHARP thread_calls_entry =
      ",(State_s_thread_tbl*)&S%itt_%s";
      int x = sprintf(big_buf_, thread_calls_entry, goto_state_no, fsm_class→identifier( )→identifier( )→c_str( ));
      Op_str.write(big_buf_, x);
    }
  }
  else {
    Op_str ≪ ",0";
  }
```

This code is used in section 187.

196. Procedure call function entry.
Watch out for |t|being a returned T from a called thread. If so this is not your average bear call.

```
⟨ procedure call function entry 196 ⟩ ≡
  svi = State.state_s_vector_.find(LR1_FSET_TRANSIENCE_OPERATOR);
  if (svi ≠ svie) {
    S_VECTOR_ELEMS_ITER_type seli = svi→second.begin( );
    state_element * se = *seli;
    if (se→next_state_element_→goto_state_ ≡ 0) {
      Op_str ≪ ",0";      /* rtned TRAshift from a PARshift */
    }
    else {
      ⟨ output called procedure table def / imp 167 ⟩;
    }
  }
  else {
    Op_str ≪ ",0";      /* no procedure call */
  }
```

This code is used in section 187.

197. *emit_each_lr_state_s_tables*.
Actions that can occur within a state:
1) shift
2) reduce
3) accept
4) called threads
5) called procedure

⟨ accrue source for emit 8 ⟩ +≡

```
void emit_each_lr_state_s_tables(std::ofstream & Op_str)
{
  STATES_ITER_type si = LR1_STATES.begin( );
  STATES_ITER_type sie = LR1_STATES.end( );
  for ( ; si ≠ sie; ++si) {      /* walk the states */
    state * cur_state = *si;
    if (cur_state→state_no_ ≡ 1) {
      output_1st_state_s_shift_table(Op_str, *cur_state);
    }
    else {
      output_all_others_state_s_shift_table(Op_str, *cur_state);
    }
    output_meta_shifts_separately(Op_str, *cur_state);
    output_state_s_reduced_table(Op_str, *cur_state);
    output_state_s_called_threads_table(Op_str, *cur_state);
    output_state_s_called_procedure_table(Op_str, *cur_state);
    output_lr_state(Op_str, *cur_state);
  }
}
```

198. OP_GRAMMAR_TBL implementation.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_GRAMMAR_TBL(TOKEN_GAGGLE & Error_queue)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    string fn(fsm_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_fsmtbl;
    std::ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base::out | ios::trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_fsmtbl_filename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*fsm_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_GRAMMAR_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    user_imp_tbl(Op_file);
    fsm_cpp_includes(Op_file);
    using_ns_for_fsm_cpp(Op_file);
    output_la_sets(Op_file);
    externs_and_thread_tbl_defs(Op_file);
    emit_each_lr_state_s_tables(Op_file);
    Op_file.close( );
  }
```

199. Template of enumeration header: OP_ENUMERATION_HEADER.
Count those terminals.
1) Comments — file name, time and date info
2) enumeration include guard declaration
2.1) namespace declaration of enumeration scheme
2.1.1) start definition of T_enum structure
2.1.1.1) begin declaration of c enum list
2.1.1.1.1) summary of vocabulary classes
2.1.1.1.2) enumerate lrk
2.1.1.1.3) enumerate raw characters
2.1.1.1.4) enumerate meta terminals
2.1.1.1.5) enumerate errors
2.1.2) close off enum list
2.1.2) close off T_enum structure
2.2) close off namespace definition
3) close off include guard declaration

200. Enumerate the LR constants classification: *enumerate_lrk*.
Note that the manufactured enumerate name is composed of a prefix "T_", its class name, and suffixed with "_".

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumerate_lrk(std::ofstream & Op_str)
  {
    T_lr1_k_phrase * ph = O2_LRK_PHASE;
    std::vector < T_terminal_def *> *dictionary = ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enumerate_item =
    "␣␣␣,T_%s_␣=␣%i";
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      int x = sprintf(big_buf_, enumerate_item, td→classsym( )→c_str( ), td→enum_id( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

201. Enumerate the Raw Characters classification: *enumerate_rc*.
Note that the manufactured enumerate name is composed of a prefix "T_", its class name, and suffixed with "_".

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumerate_rc(std::ofstream & Op_str)
  {
    T_rc_phrase * ph = O2_RC_PHASE;
    std::vector < T_terminal_def *> *dictionary = ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enumerate_item =
    "␣␣␣,T_%s_␣=␣%i";
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      int x = sprintf(big_buf_, enumerate_item, td→classsym( )→c_str( ), td→enum_id( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

202. Enumerate the Errors classification: *enumerate_errors*.
Note that the manufactured enumerate name is composed of a prefix "T_", its class name, and suffixed with "_".

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumerate_errors(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * ph = O2_ERROR_PHASE;
    std::vector < T_terminal_def *> *dictionary = ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enumerate_item =
    "␣␣␣,T_%s_␣=␣%i";
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      int x = sprintf(big_buf_, enumerate_item, td→classsym( )→c_str( ), td→enum_id( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

203. Enumerate the T classification: *enumerate_T*.
Note that the manufactured enumerate name is composed of a prefix "T_", its class name, and suffixed with "_".

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumerate_T(std::ofstream & Op_str)
  {
    T_terminals_phrase * ph = O2_T_PHASE;
    std::vector < T_terminal_def *> *dictionary = ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enumerate_item =
    "␣␣␣,T_%s_␣=␣%i";
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      int x = sprintf(big_buf_, enumerate_item, td→classsym( )→c_str( ), td→enum_id( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

204. Terminals Enumeration summary: *enumeration_summary_for_struct*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumeration_summary_for_struct(std::ofstream & Op_str)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP summary =
    "␣␣␣sum_total_T␣=␣%i\n"
    "␣␣␣,no_of_terminals␣=␣%i\n"
    "␣␣␣,no_of_raw_chars␣=␣%i\n"
    "␣␣␣,no_of_lr1_constants␣=␣%i\n"
    "␣␣␣,no_of_error_terminals␣=␣%i\n"
    "␣␣␣,start_LRK=%i,end_LRK=%i\n"
    "␣␣␣,start_RC=%i,end_RC=%i\n"
    "␣␣␣,start_T=%i,end_T=%i\n"
    "␣␣␣,start_ERR=%i,end_ERR=%i\n"
    "␣␣␣,start_R=%i";
    int x = sprintf(big_buf_, summary, enum_ph→total_enumerate( ), enum_ph→total_T_enumerate( ),
        enum_ph→total_rc_enumerate( ), enum_ph→total_lrk_enumerate( ), enum_ph→total_err_enumerate( ),
        enum_ph→start_lrk_enumerate( ), enum_ph→stop_lrk_enumerate( ), enum_ph→start_rc_enumerate( ),
        enum_ph→stop_rc_enumerate( ), enum_ph→start_T_enumerate( ), enum_ph→stop_T_enumerate( ),
        enum_ph→start_err_enumerate( ), enum_ph→stop_err_enumerate( ), enum_ph→total_enumerate( ));
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

205. *enumeration_define_list*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumeration_define_list(std::ofstream & Op_str)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP enum_list_start =
    "␣␣enum␣enumerated_terminals%s{";
    int x = sprintf(big_buf_, enum_list_start, "␣");
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    enumeration_summary_for_struct(Op_str);
    enumerate_lrk(Op_str);
    enumerate_rc(Op_str);
    enumerate_T(Op_str);
    enumerate_errors(Op_str);
    KCHARP enum_list_end =
    "␣␣};␣//␣close␣defining␣enum␣list";
    Op_str ≪ enum_list_end ≪ endl;
  }
```

206. *enumeration_define_structure*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumeration_define_structure(std::ofstream & Op_str)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP struct_start =
    "␣struct␣T_Enum%s{";
    int x = sprintf(big_buf_, struct_start, "␣");
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    enumeration_define_list(Op_str);
    KCHARP struct_end =
    "␣};␣//␣close␣defining␣T_enum";
    Op_str ≪ struct_end ≪ endl;
  }
```

207. *enumeration_namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumeration_namespace_for_header(std::ofstream & Op_str)
  {
    using namespace NS_yacco2_terminals;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP ns_of_enum_start =
    "namespace␣%s␣{";
    int x = sprintf(big_buf_, ns_of_enum_start, enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    enumeration_define_structure(Op_str);
    KCHARP ns_of_enum_end =
    "}␣//␣end␣of␣namespace";
    Op_str ≪ ns_of_enum_end ≪ endl;
  }
```

208. *enumeration_include_guard_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void enumeration_include_guard_for_header(std::ofstream & Op_str)
  {
    using namespace NS_yacco2_terminals;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP signal_guard_start =
    "#ifndef␣__%s_h__\n"
    "#define␣__%s_h__␣1";
    int x = sprintf(big_buf_, signal_guard_start, enum_ph→filename_id()→identifier()→c_str(),
        enum_ph→filename_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    enumeration_namespace_for_header(Op_str);
    KCHARP signal_guard_end =
    "#endif";
    Op_str ≪ signal_guard_end ≪ endl;
  }
```

209. Gen Tes's literal names for specific Tes.

```
⟨ accrue source for emit 8 ⟩ +≡
  void gen_Tes_literals_per_spec_voc(std::ofstream & Op_str
  , std::vector < T_terminal_def *> ::iterator I, std::vector < T_terminal_def *> ::iterator IE)
  {
    char big_buf_[BIG_BUFFER_32K];
    KCHARP literal =
    "%s";
    for ( ; I ≠ IE; ++I) {
      T_terminal_def * td = *I;
      int x = sprintf(big_buf_, literal, td→classsym( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

210. Gen Tes's literal names for O_2^{linker}.

```
⟨ accrue source for emit 8 ⟩ +≡
  void gen_Tes_literals(std::ofstream & Op_str)
  {
    T_terminals_phrase * pht = O2_T_PHASE;
    T_rc_phrase * phrc = O2_RC_PHASE;
    T_error_symbols_phrase * phe = O2_ERROR_PHASE;
    T_lr1_k_phrase * phlr = O2_LRK_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP t_alphabet =
    "T-alphabet%s";
    int x = sprintf(big_buf_, t_alphabet, "␣");
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    gen_Tes_literals_per_spec_voc(Op_str, phlr→crt_order( )→begin( ), phlr→crt_order( )→end( ));
    gen_Tes_literals_per_spec_voc(Op_str, phrc→crt_order( )→begin( ), phrc→crt_order( )→end( ));
    gen_Tes_literals_per_spec_voc(Op_str, pht→crt_order( )→begin( ), pht→crt_order( )→end( ));
    gen_Tes_literals_per_spec_voc(Op_str, phe→crt_order( )→begin( ), phe→crt_order( )→end( ));
    KCHARP end_t_alphabet =
    "end-T-alphabet%s";
    x = sprintf(big_buf_, end_t_alphabet, "␣");
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

211. OP_ENUMERATION_HEADER **implementation.**
This is the counting scheme for the grammar's Terminal Vocabulary for all their classifications: LRk, raw characters, errors, and meta-terminals. The terminal count starts from 0 going outwards on the positive axis. The Terminals are ordered by "LRk" classification and its items with the same count pattern for "raw characters", "error terminals", and finally the meta "terminals". The "LRk" and "RC" classifications are constant and fixed. Once upon a time an approximate enumeration header and their c++ code was generated for the Yac_2o_2 library placed in "/yacco2/library" for the library code to compile. The library uses some error terminals. Please see Yac_2o_2's library documentation.

This header file is the glue for all other generated tables as the enumeration number per terminal is rooted in the token symbol used by the parsing tables and error reporting facilities. To note, each grammar builds on this counting scheme for their production rules ordered after the meta-terminals. The rules ranking is by order of appearance within the grammar.

I also included the "T-alphabet" file containing the literal names of the Tes for all vocabularies in create order. This is used by $O_2^{linker}$to sprinkle their referenced names throughout the lookahead sets gened by it from the "xxx.fsc" file. Why here, cus i use the enumeration file name prefix concatentaed with the ".fsc" extension. It only gets gened when meta-terminals or error Tes are to be gened.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_ENUMERATION_HEADER(TOKEN_GAGGLE & Error_queue)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    string fn(enum_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_enumeration_hdr;
    std :: ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_enum_filename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*enum_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_ENUMERATION_HEADER_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    enumeration_include_guard_for_header(Op_file);
    Op_file.close( );
  }
```

212. *OP_T_Alphabet* **implementation.**
I also included the "T-alphabet" file containing the literal names of the Tes for all vocabularies in create order. This is used by O_2^{linker} to sprinkle their referenced names throughout the lookahead sets gened by it from the "xxx.fsc" file. Why here, cus i use the enumeration file name prefix concatentaed with the ".fsc" extension. It only gets gened when meta-terminals or error Tes are to be gened.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_T_Alphabet(TOKEN_GAGGLE & Error_queue)
  {
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    string fn_literal(enum_ph→filename_id( )→identifier( )→c_str( ));
    fn_literal += Suffix_t_alphabet;
    std :: ofstream Op_file;
    Op_file.open(fn_literal.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_enum_filename(fn_literal.c_str( ));
      sym→set_line_no_and_pos_in_line(*enum_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_ENUMERATION_HEADER_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn_literal.c_str( ));
    gen_Tes_literals(Op_file);
    Op_file.close( );
  }
```

213. Template of Errors vocabulary header: `OP_ERRORS_HEADER`.
Errors vocabulary header generation.
1) Comments — file name, time and date info
2) include files
3) Errors include guard declaration
  3.1) namespace declaration of Errors
    3.1.1) use terminals enumeration namespace
    3.1.2) loop thru the Errors list to generate their declarations
    3.1.2.1) generate the specific Error terminal definition
    3.1.3) close off namespace definition
4) close off Errors include guard declaration

214. *errors_loop_thru_and_gen_defs_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_loop_thru_and_gen_defs_for_header(std::ofstream & Op_str){
      T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
      std::vector < T_terminal_def *> *dictionary = errors_ph→crt_order( );
      char big_buf_[BIG_BUFFER_32K];
      KCHARP def_item =
      "␣␣struct␣%s:public␣yacco2::CAbs_lr1_sym{\n""%s";
      std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
      std::vector < T_terminal_def *> ::iterator ie = dictionary→end( ); for ( ; i ≠ ie; ++i) {
          T_terminal_def * td = *i;
      SDC_MAP_type * sdc_map = td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_user_declaration);
      if (j ≡ sdc_map→end( )) {      /* shell so gen default */
        KCHARP shell_of_def_item =
        "␣␣struct␣%s:public␣yacco2::CAbs_lr1_sym{\n""␣␣␣␣␣%s();\n""␣␣};";
        int x = sprintf(big_buf_, shell_of_def_item, td→classsym( )→c_str( ), td→classsym( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      else {      /* grammar writer defined */
      T_user_declaration * gw_sdc = ( T_user_declaration * ) j→second;
      int x = sprintf(big_buf_, def_item, td→classsym( )→c_str( ),
          gw_sdc→syntax_code( )→syntax_code( )→c_str( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      j = sdc_map→find(SDC_op);
      if (j ≠ sdc_map→end( )) {      /* grammar writer code */
        Op_str ≪ "␣␣op();" ≪ endl;
      }
      j = sdc_map→find(SDC_destructor);
      if (j ≠ sdc_map→end( )) {      /* gw code */
        KCHARP dtor =
        "␣␣static␣void␣dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P);\n";
        int x = sprintf(big_buf_, dtor, td→classsym( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      Op_str ≪ "␣␣};" ≪ endl;      /* close off definition */
      } } }
```

215. *errors_use_enum_namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_use_enum_namespace_for_header(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP using_T_namespace =
    "␣␣using␣namespace␣%s;";
    int x = sprintf(big_buf_, using_T_namespace, enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

216. *errors_namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_namespace_for_header(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP namespace_start =
    "namespace␣%s{";
    int x = sprintf(big_buf_, namespace_start, errors_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    errors_use_enum_namespace_for_header(Op_str);
    errors_loop_thru_and_gen_defs_for_header(Op_str);
    KCHARP namespace_end =
    "}//namespace";
    Op_str ≪ namespace_end ≪ endl;
  }
```

217. *errors_include_guard_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_include_guard_for_header(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP signal_guard_start =
    "#ifndef␣__%s_h__\n"
    "#define␣__%s_h__␣1";
    int x = sprintf(big_buf_, signal_guard_start, errors_ph→filename_id()→identifier()→c_str(),
        errors_ph→filename_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    errors_namespace_for_header(Op_str);
    KCHARP signal_guard_end =
    "#endif";
    Op_str ≪ signal_guard_end ≪ endl;
  }
```

218. *errors_include_files_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_include_files_for_header(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lr_ph = O2_LRK_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_files =
    "#include␣\"%s\"\n"
    "#include␣\"%s%s\"\n"      /* T enumeration */
    "#include␣\"%s%s\"";       /* lr constants */
    int x = sprintf(big_buf_, include_files, O2_library_file, enum_ph→filename_id()→identifier()→c_str(),
        Suffix_enumeration_hdr, lr_ph→filename_id()→identifier()→c_str(), Suffix_LRK_hdr);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

219. OP_ERRORS_HEADER **implementation.**
Bang out those error definitions.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_ERRORS_HEADER(TOKEN_GAGGLE & Error_queue)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    string fn(errors_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_Errors_hdr;
    std :: ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_errors_hdrfilename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*errors_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_ERRORS_HEADER_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    errors_include_files_for_header(Op_file);
    errors_include_guard_for_header(Op_file);
    Op_file.close( );
  }
```

220. Template of User Errors vocabulary header: OP_ERRORS_HEADER.
Users Errors vocabulary header generation.
1) Comments — file name, time and date info
2) include files
3) use terminals enumeration namespace
4) loop thru the T list to generate their implementations
4.1) generate the specific T's terminal definition

221. *errors_imp_dtor*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_imp_dtor(std::ofstream & Op_str, T_terminal_def * Td, SDC_MAP_ITER_type & J){
      char big_buf_[BIG_BUFFER_32K]; T_destructor * dtor_t = ( T_destructor * ) J→second;
      KCHARP dtor_code =
      "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
      "␣bool␣ABORT_STATUS␣=␣((yacco2::Parser*)P)->top_stack_record()->aborted__;\n"
      "␣%s*␣R␣=␣(%s*)(This);\n"
      "␣%s\n"
      "}";
      KCHARP dtor_code_noabort =
      "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
      "␣%s*␣R␣=␣(%s*)(This);\n"
      "␣%s\n"
      "}";
      std::string::size_type r = dtor_t→syntax_code( )→syntax_code( )→find("ABORT_STATUS");
      int x(0);
      if (r ≠ std::string::npos) {      /* using abort status */
        x = sprintf(big_buf_, dtor_code, Td→classsym( )→c_str( ), Td→classsym( )→c_str( ),
            Td→classsym( )→c_str( ), Td→classsym( )→c_str( ), dtor_t→syntax_code( )→syntax_code( )→c_str( ));
      }
      else {
        x = sprintf(big_buf_, dtor_code_noabort, Td→classsym( )→c_str( ), Td→classsym( )→c_str( ),
            Td→classsym( )→c_str( ), Td→classsym( )→c_str( ), dtor_t→syntax_code( )→syntax_code( )→c_str( ));
      }
      Op_str.write(big_buf_, x);
      Op_str ≪ endl; }
```

222. *errors_imp_implementation*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_imp_implementation(std::ofstream & Op_str, T_terminal_def * Td, SDC_MAP_ITER_type & J){
      char big_buf_[BIG_BUFFER_32K]; T_user_implementation * imp_t = ( T_user_implementation * )
          J→second;
      Op_str ≪ imp_t→syntax_code( )→syntax_code( )→c_str( );
      SDC_MAP_type * sdc_map = Td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_destructor);
      if (j ≠ sdc_map→end( )) {
        errors_imp_dtor(Op_str, Td, j);
      }
      }
```

223. *errors_imp_shellimplementation*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_imp_shellimplementation(std::ofstream & Op_str, T_terminal_def * Td,
          std::string & Autodelete, std::string & Autoabort)
  {
    char big_buf_[BIG_BUFFER_32K];
    KCHARP shell_of_def_item =
    "%s::\n"
    "%s()\n"
    "␣T_CTOR(\"%s\",T_Enum::T_%s_,%s,%s,%s){}";
    int x = sprintf(big_buf_, shell_of_def_item, Td→classsym()→c_str(), Td→classsym()→c_str(),
        Td→t_name()→c_str()      /* literal */
    , Td→classsym()→c_str()      /* enum */
    ,"0"      /* no dtor */
    , Autodelete.c_str(), Autoabort.c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

224. *errors_loop_thru_and_gen_defs_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_loop_thru_and_gen_defs_for_imp(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    std::vector < T_terminal_def *> *dictionary = errors_ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    string auto_delete;
    string auto_abort;
    string dtor_adr;
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      SDC_MAP_type * sdc_map = td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_user_implementation);
      if (td→autodelete( ) ≡ false) {
        auto_delete = "false";
      }
      else {
        auto_delete = "true";
      }
      if (td→autoabort( ) ≡ false) {
        auto_abort = "false";
      }
      else {
        auto_abort = "true";
      }
      if (j ≡ sdc_map→end( )) {      /* shell so gen default */
        errors_imp_shellimplementation(Op_str, td, auto_delete, auto_abort);
      }
      else {      /* grammar writer defined */
        errors_imp_implementation(Op_str, td, j);
      }
    }
  }
```

225. *errors_use_enum_namespace_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_use_enum_namespace_for_imp(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP using_namespace =
    "␣␣using␣namespace␣%s;\n""␣␣using␣namespace␣%s;";      /* T_enum */
    int x = sprintf(big_buf_, using_namespace, errors_ph→namespace_id()→identifier()→c_str(),
        enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

226. *errors_include_files_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void errors_include_files_for_imp(std::ofstream & Op_str)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lr_ph = O2_LRK_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_files =
    "#include␣\"%s%s\"";
    int x = sprintf(big_buf_, include_files, errors_ph→filename_id()→identifier()→c_str(), Suffix_Errors_hdr);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

227. `OP_ERRORS_CPP` **implementation.**
Bang out thee implementation of those error definitions.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_ERRORS_CPP(TOKEN_GAGGLE & Error_queue)
  {
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    string fn(errors_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_Errors_imp;
    std :: ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_errors_impfilename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*errors_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_ERRORS_HEADER_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    errors_include_files_for_imp(Op_file);
    errors_use_enum_namespace_for_imp(Op_file);
    errors_loop_thru_and_gen_defs_for_imp(Op_file);
    Op_file.close( );
  }
```

228. Template of USER T vocabulary header: `OP_USER_T_HEADER`**.**
User terminals vocabulary header generation.
1) Comments — file name, time and date info
2) include files
3) User T include guard declaration
3.1) namespace declaration of Errors
3.1.1) use terminals enumeration namespace
3.1.2) loop thru the User T list to generate their declarations
3.1.2.1) generate the specific User T terminal definition
3.1.3) close off namespace definition
4) close off User T include guard declaration

229. *user_t_loop_thru_and_gen_defs_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_loop_thru_and_gen_defs_for_header(std::ofstream & Op_str){
      T_terminals_phrase * t_ph = O2_T_PHASE;
      std::vector < T_terminal_def *> *dictionary = t_ph→crt_order( );
      char big_buf_[BIG_BUFFER_32K];
      KCHARP def_item =
      "␣␣struct␣%s:public␣yacco2::CAbs_lr1_sym{\n""%s";
      std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
      std::vector < T_terminal_def *> ::iterator ie = dictionary→end( ); for ( ; i ≠ ie; ++i) {
          T_terminal_def * td = *i;
      SDC_MAP_type * sdc_map = td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_user_declaration);
      if (j ≡ sdc_map→end( )) {      /* shell so gen default */
        KCHARP shell_of_def_item =
        "␣␣struct␣%s:public␣yacco2::CAbs_lr1_sym{\n""␣␣␣␣%s();\n""␣␣};";
        int x = sprintf(big_buf_, shell_of_def_item, td→classsym( )→c_str( ), td→classsym( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      else {      /* grammar writer defined */
      T_user_declaration * gw_sdc = ( T_user_declaration * ) j→second;
      int x = sprintf(big_buf_, def_item, td→classsym( )→c_str( ),
          gw_sdc→syntax_code( )→syntax_code( )→c_str( ));
      Op_str.write(big_buf_, x);
      j = sdc_map→find(SDC_op);
      if (j ≠ sdc_map→end( )) {      /* grammar writer code */
        Op_str ≪ "␣␣op();" ≪ endl;
      }
      j = sdc_map→find(SDC_destructor);
      if (j ≠ sdc_map→end( )) {      /* gw code */
        KCHARP dtor =
        "␣␣static␣void␣dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P);";
        int x = sprintf(big_buf_, dtor, td→classsym( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      Op_str ≪ "␣␣};" ≪ endl;      /* close off definition */
      } } }
```

230. *user_t_use_enum_namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_use_enum_namespace_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP using_T_namespace =
    "␣␣using␣namespace␣%s;";
    int x = sprintf(big_buf_, using_T_namespace, enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

231. *user_t_terminals_refs_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_terminals_refs_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    if (t_ph→terminals_refs_code() ≠ 0) {
      KCHARP t_ref_code =
      "␣␣%s";
      int x = sprintf(big_buf_, t_ref_code, t_ph→terminals_refs_code()→syntax_code()→c_str());
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

232. *user_t_terminals_sufx_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_terminals_sufx_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    if (t_ph→terminals_sufx_code() ≠ 0) {
      KCHARP t_sufx_code =
      "␣␣%s";
      int x = sprintf(big_buf_, t_sufx_code, t_ph→terminals_sufx_code()→syntax_code()→c_str());
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
  }
```

233. *user_t_namespace_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_namespace_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP namespace_start =
    "namespace␣%s{";
    int x = sprintf(big_buf_, namespace_start, t_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    user_t_use_enum_namespace_for_header(Op_str);
    user_t_terminals_refs_for_header(Op_str);
    user_t_loop_thru_and_gen_defs_for_header(Op_str);
    user_t_terminals_sufx_for_header(Op_str);
    KCHARP namespace_end =
    "}//namespace";
    Op_str ≪ namespace_end ≪ endl;
  }
```

234. *user_t_include_guard_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_include_guard_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP signal_guard_start =
    "#ifndef␣__%s_h__\n"
    "#define␣__%s_h__␣1";
    int x = sprintf(big_buf_, signal_guard_start, t_ph→filename_id()→identifier()→c_str(),
        t_ph→filename_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    user_t_namespace_for_header(Op_str);
    KCHARP signal_guard_end =
    "#endif";
    Op_str ≪ signal_guard_end ≪ endl;
  }
```

235. *user_t_include_files_for_header*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_include_files_for_header(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lr_ph = O2_LRK_PHASE;
    T_error_symbols_phrase * errors_ph = O2_ERROR_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_files =
    "#include␣\"%s\"\n"
    "#include␣\"%s%s\"\n"      /* T enumeration */
    "#include␣\"%s%s\"\n"      /* lr constants */
    "#include␣\"%s%s\"";       /* errors */
    int x = sprintf(big_buf_, include_files, O2_library_file, enum_ph→filename_id()→identifier()→c_str(),
        Suffix_enumeration_hdr, lr_ph→filename_id()→identifier()→c_str(), Suffix_LRK_hdr,
        errors_ph→filename_id()→identifier()→c_str(), Suffix_Errors_hdr);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

236. `OP_USER_T_HEADER` **implementation.**
Bang out those error definitions.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_USER_T_HEADER(TOKEN_GAGGLE & Error_queue)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    string fn(t_ph→filename_id()→identifier()→c_str());
    fn += Suffix_T_hdr;
    std::ofstream Op_file;
    Op_file.open(fn.c_str(), ios_base::out | ios::trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_errors_hdrfilename(fn.c_str());
      sym→set_line_no_and_pos_in_line(*t_ph→filename_id());
      sym→set_who_created("o2externs.w␣-␣OP_USER_T_HEADER_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str());
    user_t_include_files_for_header(Op_file);
    user_t_include_guard_for_header(Op_file);
    Op_file.close();
  }
```

237. Template of USER T vocabulary header: OP_USER_T_HEADER.
Errors vocabulary header generation.
1) Comments — file name, time and date info
2) include files
3) use terminals enumeration namespace
4) terminals-refs code
5) loop thru the USer T list to generate their implementations
5.1) generate the specific User T terminal definition
6) terminals-sufx code

238. *user_t_imp_dtor*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_imp_dtor(std::ofstream & Op_str, T_terminal_def * Td, SDC_MAP_ITER_type & J){
      char big_buf_[BIG_BUFFER_32K]; T_destructor * dtor_t = ( T_destructor * ) J→second;
      KCHARP dtor_code =
      "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
      "␣bool␣ABORT_STATUS␣=␣((yacco2::Parser*)P)->top_stack_record()->aborted__;\n"
      "␣%s*␣R␣=␣(%s*)(This);\n"
      "␣%s\n"
      "}";
      KCHARP dtor_code_noabort =
      "void␣%s::dtor_%s(yacco2::VOIDP␣This,yacco2::VOIDP␣P){\n"
      "␣%s*␣R␣=␣(%s*)(This);\n"
      "␣%s\n"
      "}";
      std::string::size_type r = dtor_t→syntax_code( )→syntax_code( )→find("ABORT_STATUS");
      int x(0);
      if (r ≠ std::string::npos) {      /* using abort status */
        x = sprintf(big_buf_, dtor_code, Td→classsym( )→c_str( ), Td→classsym( )→c_str( ),
            Td→classsym( )→c_str( ), Td→classsym( )→c_str( ), dtor_t→syntax_code( )→syntax_code( )→c_str( ));
      }
      else {
        x = sprintf(big_buf_, dtor_code_noabort, Td→classsym( )→c_str( ), Td→classsym( )→c_str( ),
            Td→classsym( )→c_str( ), Td→classsym( )→c_str( ), dtor_t→syntax_code( )→syntax_code( )→c_str( ));
      }
      Op_str.write(big_buf_, x);
      Op_str ≪ endl; }
```

239. *user_t_imp_implementation*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_imp_implementation(std::ofstream & Op_str, T_terminal_def * Td, SDC_MAP_ITER_type & J){
      char big_buf_[BIG_BUFFER_32K]; T_user_implementation * imp_t = ( T_user_implementation * )
          J→second;
      Op_str ≪ imp_t→syntax_code( )→syntax_code( )→c_str( );
      SDC_MAP_type * sdc_map = Td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_destructor);
      if (j ≠ sdc_map→end( )) {
        errors_imp_dtor(Op_str, Td, j);
      }
      }
```

240. *user_t_imp_shellimplementation*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_imp_shellimplementation(std::ofstream & Op_str, T_terminal_def * Td,
          std::string & Autodelete, std::string & Autoabort)
  {
    char big_buf_[BIG_BUFFER_32K];
    KCHARP shell_of_def_item =
    "%s::\n"
    "%s()\n"
    "␣T_CTOR(\"%s\",T_Enum::T_%s_,%s,%s,%s){}";
    int x = sprintf(big_buf_, shell_of_def_item, Td→classsym()→c_str(), Td→classsym()→c_str(),
        Td→t_name()→c_str()      /* literal */
    , Td→classsym()→c_str()      /* enum */
    ,"0"      /* no dtor */
    , Autodelete.c_str(), Autoabort.c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

241. *user_t_loop_thru_and_gen_defs_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_loop_thru_and_gen_defs_for_imp(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    std::vector < T_terminal_def *> *dictionary = t_ph→crt_order( );
    char big_buf_[BIG_BUFFER_32K];
    string auto_delete;
    string auto_abort;
    string dtor_adr;
    std::vector < T_terminal_def *> ::iterator i = dictionary→begin( );
    std::vector < T_terminal_def *> ::iterator ie = dictionary→end( );
    for ( ; i ≠ ie; ++i) {
      T_terminal_def * td = *i;
      SDC_MAP_type * sdc_map = td→directives_map( );
      SDC_MAP_ITER_type j = sdc_map→find(SDC_user_implementation);
      if (td→autodelete( ) ≡ false) {
        auto_delete = "false";
      }
      else {
        auto_delete = "true";
      }
      if (td→autoabort( ) ≡ false) {
        auto_abort = "false";
      }
      else {
        auto_abort = "true";
      }
      if (j ≡ sdc_map→end( )) {      /* shell so gen default */
        user_t_imp_shellimplementation(Op_str, td, auto_delete, auto_abort);
      }
      else {      /* grammar writer defined */
        user_t_imp_implementation(Op_str, td, j);
      }
    }
  }
```

242. *user_t_use_enum_namespace_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_use_enum_namespace_for_imp(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP using_namespace =
    "␣␣using␣namespace␣%s;\n""␣␣using␣namespace␣%s;";      /* T_enum */
    int x = sprintf(big_buf_, using_namespace, t_ph→namespace_id()→identifier()→c_str(),
        enum_ph→namespace_id()→identifier()→c_str());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

243. *user_t_include_files_for_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void user_t_include_files_for_imp(std::ofstream & Op_str)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_lr1_k_phrase * lr_ph = O2_LRK_PHASE;
    char big_buf_[BIG_BUFFER_32K];
    KCHARP include_files =
    "#include␣\"%s%s\"";
    int x = sprintf(big_buf_, include_files, t_ph→filename_id()→identifier()→c_str(), Suffix_T_hdr);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

244. `OP_USER_T_CPP` **implementation.**
Bang out thee implementation of those error definitions.

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_USER_T_CPP(TOKEN_GAGGLE & Error_queue)
  {
    T_terminals_phrase * t_ph = O2_T_PHASE;
    string fn(t_ph→filename_id( )→identifier( )→c_str( ));
    fn += Suffix_Errors_imp;
    std :: ofstream Op_file;
    Op_file.open(fn.c_str( ), ios_base :: out | ios :: trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_errors_impfilename(fn.c_str( ));
      sym→set_line_no_and_pos_in_line(*t_ph→filename_id( ));
      sym→set_who_created("o2externs.w␣-␣OP_USER_T_CPP", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str( ));
    user_t_include_files_for_imp(Op_file);
    user_t_use_enum_namespace_for_imp(Op_file);
    user_t_loop_thru_and_gen_defs_for_imp(Op_file);
    Op_file.close( );
  }
```

245. Template of FSC File: OP_FSC_FILE.
File of grammar's first threads info for O_2linker.
1) Comments — file name, time and date info
2) fsc prelude
3) list of native Tes in grammar's first set
4) list of directly called threads in its first set
5) complete list of used threads in grammar for linker document
6) grammar's comments for linker document

246. *fsc_prelude_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void fsc_prelude_imp(std::ofstream & Op_str, std::string & Fn)
  {
    KCHARP pp_thd_nm(0);
    string transitive("n");
    string mono;
    char big_buf_[BIG_BUFFER_32K];
    T_fsm_phrase * fcp = O2_FSM_PHASE;
    T_parallel_parser_phrase * ppp = O2_PP_PHASE;
    T_enum_phrase * enum_ph = O2_T_ENUM_PHASE;
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    RULE_DEFS_TBL_ITER_type i = rules_ph→crt_order()→begin();
    RULE_DEFS_TBL_ITER_type ie = rules_ph→crt_order()→end();
    rule_def * rd = *i;
    FIRST_SET_type * fs = rd→first_set();
    FIRST_SET_ITER_type j = fs→begin();
    FIRST_SET_ITER_type je = fs→end();
    for ( ; j ≠ je; ++j) {
      T_in_stbl * tintbl = *j;
      T_terminal_def * tdef = tintbl→t_def();
      if (tdef→enum_id() ≡ LR1_PARALLEL_OPERATOR) {
        transitive[0] = 'y';
        break;
      }
    }
    KCHARP fsc_prelude = "transitive␣␣␣%s\n""grammar-name␣\"%s\"\n""name-space␣␣␣\"%s\"\
        \n""thread-name␣␣\"%s\"\n""monolithic␣␣␣%s\n""file-name␣␣␣␣\"%s\"\
        \n""no-of-T␣␣␣␣␣␣%i";
    if (ppp ≡ 0) {
      pp_thd_nm = fcp→fsm_class_phrase()→identifier()→identifier()→c_str();
      mono += 'y';
    }
    else {
      pp_thd_nm = ppp→pp_funct()→identifier()→identifier()→c_str();
      mono += 'n';
    }
    int x = sprintf(big_buf_, fsc_prelude, transitive.c_str(), fcp→filename_id()→identifier()→c_str(),
        fcp→namespace_id()→identifier()→c_str(), pp_thd_nm, mono.c_str(), Fn.c_str(),
        enum_ph→total_enumerate());
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

247. *list_of_native_Tes_in_fs_imp*.
Create a thread's first set by building a closure-only state. There are 2 conditions that are different to a regular first set calculation:

1) an ϵ start-rule
2) rules on the rhs of the start rule that have `|.|`.

Point 1 requires the first set to be the Lookahead expression for the thread. Now what does the `|.|` symbol represent within the first set as it is not part of the input token stream? `|.|` acts like an epsilon rule and therefore u must go through the goto states from the closure-only state deriving their concrete first sets. This is a recursive situation when `|.|` is present in one of these shifted states so u keep on being devine... If the start-rule has been completely derived (accepted) then the thread's "parallel-la-boundary" expression adds its booty to the first set. If the "eolr" is present in this booty, then the thread's first set is only this symbol as it represents the "all terminals" situation.

A cheap way would have been to just make the first set contain the "eolr" symbol if the `|.|` symbol was present. This would work but would be inefficient due to the false starts in firing up the thread to have it misfire and shut down: a bit of thread farting. So lets be efficient by alculating what truely represents its thread's first set.

An example of the above situation is a thread whose Start-rule contains only rules that are all explicitly or implicitely epsilonable (`|.|` is present).

Now for the most efficient and simpliest solution - a bird's view:
As an optimization the current token is used to determine whether a thread should fire, why not include in this check against the meta `|.|` symbol? This way its saves some space as its contains just the threads that have it in their first set rather than distributing the thread-id-to-run against all its thread first set's concrete Tes. Consider the following: if the first set has only the "all terminal — eolr" then this is a 1 to 1 relationship but if the lookahead set is volumious this becomes a 1 to many situation. The 1:m situation becomes less efficient in its set search as the current token contains all the possible threads that can run. When the running grammar detects threads to-be-run against its specific parse state, the first search is against the current token, then the secondary search for its specific threads takes place. So out damn *first_set_for_threads*.

The reason for the remapping of `|.|` into `|t|` for the thread's first set is the double meaning on `|.|`:

1) `|.|` is a conditional meta-terminal used to shift out of ambiguous situations. It is checked for its existence after parallelism was unsuccessful: no threads to run or an unsuccessful threading parse.
2) What does `|.|` mean in thread launching by use of the global table of threads per T? As the contexts are exclusive, i felt i should still not overload its meaning for the 2nd context: ie, only use it in the conditional-shift context. So `|.|` is renamed to `|t|` in the global T table for wild card thread launches.

Under O_2 there are only 16 threads of this type against 2 32 bit words makes for an efficient search against the local fsa's state of local threads to possiblely run.

248. Driver of *list_of_native_Tes_in_fs_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void list_of_native_Tes_in_fs_imp(std::ofstream & Op_str)
  {
    char big_buf_[BIG_BUFFER_32K];
    int no_Tes_in_list(0);
    string Tes_in_list;
    STATES_ITER_type si = LR1_STATES.begin( );
    state * cur_state = *si;
    int no_of_Tes_in_fs(0);
    std::set < CAbs_lr1_sym *> fs_set;
    T_rules_phrase * rules_ph = O2_RULES_PHASE;
    RULE_DEFS_TBL_ITER_type i = rules_ph→crt_order( )→begin( );
    RULE_DEFS_TBL_ITER_type ie = rules_ph→crt_order( )→end( );
    rule_def * rd = *i;
    FIRST_SET_type * fs = rd→first_set( );
    FIRST_SET_ITER_type j = fs→begin( );
    FIRST_SET_ITER_type je = fs→end( );
    for ( ; j ≠ je; ++j) {
      T_in_stbl * tintbl = *j;
      T_terminal_def * tdef = tintbl→t_def( );
      if (tdef→enum_id( ) ≡ LR1_PARALLEL_OPERATOR) {
        continue;
      }
      ++no_Tes_in_list;
    }
  done: ;
    KCHARP fs_list_of_Tes = "list-of-native-first-set-terminals␣%i";
    KCHARP fs_T_in_list = "␣␣␣%s";
    KCHARP end_fs_list_of_Tes = "end-list-of-native-first-set-terminals%s";
    int x = sprintf(big_buf_, fs_list_of_Tes, no_Tes_in_list);
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
    j = fs→begin( );
    for ( ; j ≠ je; ++j) {
      T_in_stbl * tintbl = *j;
      T_terminal_def * tdef = tintbl→t_def( );
      if (tdef→enum_id( ) ≡ LR1_PARALLEL_OPERATOR) {
        continue;
      }
      if (tdef→enum_id( ) ≡ LR1_INVISIBLE_SHIFT_OPERATOR) {      /* handled by O_2^linker */
        x = sprintf(big_buf_, fs_T_in_list, LR1_FSET_TRANSIENCE_OPERATOR_LITERAL);
      }
      else {
        x = sprintf(big_buf_, fs_T_in_list, tdef→classsym( )→c_str( ));
      }
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
    }
    x = sprintf(big_buf_, end_fs_list_of_Tes, "␣");
```

```
    Op_str.write(big_buf_, x);
    Op_str ≪ endl;
  }
```

249. *transitive_list_of_threads_in_fs_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void transitive_list_of_threads_in_fs_imp(std::ofstream & Op_str){
      char big_buf_[BIG_BUFFER_32K];
      T_rules_phrase * rules_ph = O2_RULES_PHASE;
      AST * rules_tree = rules_ph→phrase_tree( );
      AST * start_rule_def_t = AST::get_1st_son(*rules_tree); rule_def * rd = ( rule_def * )
          AST::content(*start_rule_def_t);
      int no_threads_in_list = rd→called_thread_first_set( )→size( );
      KCHARP transitive_thread_list = "list-of-transitive-threads␣%i";
      KCHARP transitive_thread_in_list = "␣␣%s::%s";
      KCHARP end_transitive_thread_list = "end-list-of-transitive-threads%s";
      int x = sprintf(big_buf_, transitive_thread_list, no_threads_in_list);
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      std::set < T_called_thread_eosubrule *> ::iterator e = rd→called_thread_first_set( )→begin( );
      std::set < T_called_thread_eosubrule *> ::iterator ee = rd→called_thread_first_set( )→end( );
      for ( ; e ≠ ee; ++e) {
        T_called_thread_eosubrule * called_thd = *e;
        x = sprintf(big_buf_, transitive_thread_in_list, called_thd→ns( )→identifier( )→c_str( ),
            called_thd→called_thread_name( )→identifier( )→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      x = sprintf(big_buf_, end_transitive_thread_list, "␣");
      Op_str.write(big_buf_, x);
      Op_str ≪ endl; }
```

250. *used_list_of_threads_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void used_list_of_threads_imp(std::ofstream & Op_str){
      char big_buf_[BIG_BUFFER_32K];
      std::set < string > used_threads;
      STATES_ITER_type i = LR1_STATES.begin( );
      STATES_ITER_type ie = LR1_STATES.end( ); for ( ; i ≠ ie; ++i) {
        /* extract all the used threads */
      state * start_state = *i;
      S_VECTOR_ELEMS_type * state_elems_wparallelism(0);
      S_VECTORS_type & vect = start_state→state_s_vector_;
      S_VECTORS_ITER_type vi = vect.find(LR1_PARALLEL_OPERATOR);
      if (vi ≡ vect.end( )) continue;      /* no parallelism in state? */
      state_elems_wparallelism = &vi→second;
      S_VECTOR_ELEMS_ITER_type k = state_elems_wparallelism→begin( );
      S_VECTOR_ELEMS_ITER_type ke = state_elems_wparallelism→end( ); for ( ; k ≠ ke; ++k) {
          state_element * thread_1st_elem = *k;      /* parallel operator */
      AST * rtned_T_t = thread_1st_elem→next_state_element_→sr_element_;
      AST * bypassed_thd_eos_t = AST::brother(*rtned_T_t);
      if (bypassed_thd_eos_t ≡ 0) continue;
      CAbs_lr1_sym * sym = AST::content(*bypassed_thd_eos_t);
      if (sym→enumerated_id__ ≠ T_Enum::T_T_called_thread_eosubrule_) {
        continue;
      }
      T_called_thread_eosubrule * called_thd = ( T_called_thread_eosubrule * ) sym;
        /* add to used thread if first time */
      string fqtnm;
      fqtnm += called_thd→ns( )→identifier( )→c_str( );
      fqtnm += "::";
      fqtnm += called_thd→called_thread_name( )→identifier( )→c_str( );
      if (used_threads.find(fqtnm) ≡ used_threads.end( )) used_threads.insert(fqtnm);
      } } KCHARP used_thread_list = "list-of-used-threads␣%i";
      KCHARP used_thread_in_list = "␣␣%s";
      KCHARP end_used_thread_list = "end-list-of-used-threads%s";
      int x = sprintf(big_buf_, used_thread_list, used_threads.size( ));
      Op_str.write(big_buf_, x);
      Op_str ≪ endl;
      std::set < string > ::iterator si = used_threads.begin( );
      std::set < string > ::iterator sie = used_threads.end( );
      for ( ; si ≠ sie; ++si) {
        x = sprintf(big_buf_, used_thread_in_list, si→c_str( ));
        Op_str.write(big_buf_, x);
        Op_str ≪ endl;
      }
      x = sprintf(big_buf_, end_used_thread_list, "␣");
      Op_str.write(big_buf_, x);
      Op_str ≪ endl; }
```

251. *grammar_s_comments_for_linker_doc_imp*.

```
⟨ accrue source for emit 8 ⟩ +≡
  void grammar_s_comments_for_linker_doc_imp(std::ofstream & Op_str)
  {
    char big_buf_[BIG_BUFFER_32K];
    T_fsm_phrase * fcp = O2_FSM_PHASE;
    KCHARP fsm_comments = "fsm-comments\n""\"%s\"";
    int x = sprintf(big_buf_, fsm_comments, fcp->comment()->c_string()->c_str());
    Op_str.write(big_buf_, x);
    Op_str << endl;
  }
```

252. `OP_FSC_FILE` **implementation.**
Okay last of the Moh...

```
⟨ accrue source for emit 8 ⟩ +≡
  void OP_FSC_FILE(TOKEN_GAGGLE & Error_queue)
  {
    T_fsm_phrase * fsm_ph = O2_FSM_PHASE;
    stringfn(fsm_ph->filename_id()->identifier()->c_str());
    fn += Suffix_fsc;
    std::ofstream Op_file;
    Op_file.open(fn.c_str(), ios_base::out | ios::trunc);
    if (¬Op_file) {
      CAbs_lr1_sym * sym = new Err_bad_fsmtbl_filename(fn.c_str());
      sym->set_line_no_and_pos_in_line(*fsm_ph->filename_id());
      sym->set_who_created("o2externs.w␣-␣OP_FSC_FILE", __LINE__);
      Error_queue.push_back(*sym);
      return;
    }
    intro_comment(Op_file, fn.c_str());
    fsc_prelude_imp(Op_file, fn);
    list_of_native_Tes_in_fs_imp(Op_file);
    transitive_list_of_threads_in_fs_imp(Op_file);
    used_list_of_threads_imp(Op_file);
    grammar_s_comments_for_linker_doc_imp(Op_file);
    Op_file.close();
  }
```

253. Bric-a-brac.

254. Better coordination between the coordinates.
To improve processing of nested files, a global **yacco2**::STK_FILE_NOS__ was introduced to swing-and-sway with its partner **yacco2**::FILE_CNT__. **yacco2**::FILE_CNT__ gets incremented by the *tok_can* < **std**::*ifstream* > container while **yacco2**::STK_FILE_NOS__ pushes and pops **yacco2**::FILE_CNT__. The global **yacco2**::FILE_TBL__ keeps an array of files processed. The importance of **yacco2**::STK_FILE_NOS__ and **yacco2**::FILE_CNT__ is in the assignment of the source file coordinates to the manufactured tokens. This allows one to correlate the error token back to the source file.

As **yacco2**::FILE_CNT__ is an expanding number, the grammar writer must properly coordinate when nested files are occurring, and to ensure when the file gets closed that the calling file number becomes **yacco2**::STK_FILE_NOS__ current value. A little stacking sir...

Improvement: 8 Mar. 2005

255. Blending of Yacco2 and Linker externals.
General house cleaning. A one-stop-lollipop-stall of recycled ideas: your marche puce. Look at the stalls ... folders. Externals folder is now equal to Linker, Yacco2, Library. Ahem...

Improvement: 9 Mar. 2005

256. Remove option /s:xxxx number of source lines to generate.
This was a limitation of Microsoft's c++ compiler circa release 5.xx. It honked. Now I'm more optimistic of their current products. So I removed the /s:xxxx option from the *Yacco2_lcl_options*(*s*).*lex* grammars and as a passed in parameter to YACCO2_PARSE_CMD_LINE. Ah the signs of spring cleaning...

Improvement: 2 June 2005

257. Bug in MS compiler 2005.
To make my *mpost_output* grammar more print presentable when emitting itself for *mpost*, *cweave*, and *pdftex* consumption, i moved most of its class code into static class procedures whereby their implementations are within this document. Part of the improvement was the calling of the static member MPOST_CWEB_EMIT_PREFIX_CODE after *ctor* creation from the class's "op" directive to do basic table initialization and to establish the output files to be written to. The class object's "this" was passed as a parameter to MPOST_CWEB_EMIT_PREFIX_CODE.

Well the bug, ⟨ initialize the *Cmpost_output* class variables 258 ⟩ within MPOST_CWEB_EMIT_PREFIX_CODE established opening the files. After verifying that the file's "ofstream" was healthy, upon the 1st attempt to write to it MS c++ emitted code honked from its mutex handling routine. Well at least they are improving in their trappings.

Work around, extract from the MPOST_CWEB_EMIT_PREFIX_CODE procedure the initialization of the class's variables and place it in the "op" directive of the class before the call to MPOST_CWEB_EMIT_PREFIX_CODE to complete the balance of work. Now MS's emitted code doesn't honk when the initialization context is within the class code.

Grrr: 26 Jan. 2006

258. Initialize the *Cmpost_output* class variables. The initial *push_back* of "0" is to register a boggy 0 element as i count from 1.

```
⟨ initialize the Cmpost_output class variables 258 ⟩ ≡
  Fsm→no_subrules_per_rule_.push_back(0);
  Fsm→gened_date_time_ += DATE_AND_TIME( );
  Fsm→mp_dimension_ += "␣abcdefghijklmnopqrstuvwxyz";
  Fsm→w_fig_no_ = 0;
  Fsm→rule_def_ = 0;
  Fsm→subrule_def_ = 0;
  Fsm→rule_no_ = 0;
  Fsm→subrule_no_ = 0;
  Fsm→elem_no_ = 0;
  Fsm→no_of_rules_ = 0;
  Fsm→no_of_subrules_ = 0;
  Fsm→mp_filename_ += Fsm→grammar_filename_prefix_.c_str( );
  Fsm→mp_filename_ += ".mp";
  Fsm→omp_file_.open(Fsm→mp_filename_.c_str( ), ios_base::out | ios::trunc);
  if (¬Fsm→omp_file_) {
    CAbs_lr1_sym * sym = new Err_bad_filename(Fsm→mp_filename_.c_str( ));
    Fsm→parser__→add_token_to_error_queue(*sym);
    Fsm→parser__→set_stop_parse(true);
    return;
  }
  Fsm→w_filename_ += Fsm→grammar_filename_prefix_.c_str( );
  Fsm→w_filename_ += ".w";
  Fsm→ow_file_.open(Fsm→w_filename_.c_str( ), ios_base::out | ios::trunc);
  if (¬Fsm→ow_file_) {
    CAbs_lr1_sym * sym = new Err_bad_filename(Fsm→w_filename_.c_str( ));
    Fsm→parser__→add_token_to_error_queue(*sym);
    Fsm→parser__→set_stop_parse(true);
    return;
  }
```

This code is cited in section 257.

259. *convertMPtoPDF* macro bug.

Well here's the scoop: my grammar diagrams are smoking but when there is a blank (space) in a terminal being displayed, the *convertMPtoPDF* macro halts saying that the *empty* macro is illegally terminated.

Alas, for the moment i substitute a "." in place of the space to keep the pictures flowing.

3 Feb. 2006

260. *cweave* emits bad cross-reference command in o2externs.scn file.
It emits:
\I\X8, xx , yy, 73:accrue source for emit\X
\U7.
instead of:
\I\X8:accrue source for emit\X
\Us10, 11, ..., 70\ETs72. This works. But checking with other modules like "wlibrary" the code is right so what's making it cough?

Until i fix *cweave* either remove "*i*nd" line at the end of "o2externs.tex" file or "remove or add" the offending commands in "o2externs.scn" file.

14 June 2006

261. *cweave* emits tab in place of "open brace".
The "ctangle" emitted cpp code is okay. Shows up when i emit "cpp" source code implementation procedures for the grammar.

12 Nov 2006

262. Language cleanup.
Removed "entry and exit" routines from fsm class definition. Fsm's ctor and dtor allow the same points to be tweaked. In O_2's creation days, i was over zealous on trigger points — fsm, push and pop action off the parse stack, and subrules being separate classes. all this to leave tracks when needed.
Eliminated the subrule directives as they were needed only when a subrule was designed as a separate class — now a subrule definition is part of the rule definition: srx() where x is the subrule number. Separate subrule definitions really had marginal utility — euphemism for useless and never exploited.

1) rhs-op replaced by op directive
2) rhs-xxx elinimated directives
   where xxx is constructor, destructor, user-declaration or implementation

This my take on gently waff where whiffes linger...

13 Nov 2006

263. Templates and their tantrums.
Due to MS's equal algorithm throwing up, i roll my own set compare — *find_common_la_set_idx*.

20 Nov 2006

264. Rule use count off by nested recursion.
The rule use count algorithm didn't handle the following boiled down example properly:

T → Ra T Rb
T → Ra ...

The nested T was recognized but not post evaluated on Ra use. It should double its maximun calculated use count after direct and indirect rule calculations.

Nov. 2007

265. Index.

error_symbols_attributes.lex: 25.
error_symbols_phrase.lex: 25.
error_symbols_phrase_th.lex: 25.
fsm_attributes.lex: 21.
fsm_class_directives.lex: 21.
fsm_class_phrase_th.lex: 21.
fsm_phrase.lex: 21.
fsm_phrase_th.lex: 21.
lr1_k_attributes.lex: 27.
lr1_k_phrase.lex: 27.
lr1_k_phrase_th.lex: 27.
parallel_parser_attributes.lex: 23.
parallel_parser_phrase.lex: 23.
parallel_parser_phrase_th.lex: 23.
rc_attributes.lex: 26.
rc_phrase_th.lex: 26.
rc_symbols_phrase.lex: 26.
T_enum_attributes.lex: 24.
T_enum_phrase.lex: 24.
T_enum_phrase_th.lex: 24.
terminals_attributes.lex: 28, 29.
terminals_phrase.lex: 28, 29.
terminals_phrase_th.lex: 28, 29.
`__LINE__`: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
a: 57, 66, 74, 79.
ab: 66, 74, 79, 143.
abort_parse__: 41.
ac: 127, 145.
`ACCEPT_FILTER`: 39, 53, 89, 90, 95, 110, 111, 112, 154.
accept_prted: 170.
ad: 66, 74, 79, 143.
add_rule_adding_T_in_first_set: 96.
add_sym_to_stbl: 37.
add_to_called_thread_first_set: 92.
add_to_first_set: 96, 97.
add_token_to_error_queue: 21, 23, 24, 25, 26, 27, 28, 32, 41, 42, 258.
ae: 70, 75, 81.
Against_rule: 5, 109, 110, 111, 112.
against_rule_enum_rel0: 109.
ai: 165.
all_shift_entry: 179, 186, 189.
alphabet: 64, 70, 72, 75, 77, 81.
ar_name: 165, 166, 168.
ar_s_rule_s_name: 165.
*arbitrator_name_*: 165.
are_2_la_sets_equal: 105, 106.
argc: 5.
argv: 5.
array_of_subrules_to_rules_mapping: 121.
array_of_subrules_to_rules_mapping_end: 140.
array_of_subrules_to_rules_mapping_start: 140.
`AST`: 5, 8, 12, 13, 21, 23, 24, 25, 26, 27, 28, 32, 35, 36, 38, 39, 40, 53, 55, 56, 57, 88, 89, 90, 91, 92, 93, 95, 96, 98, 110, 111, 112, 146, 147, 154, 165, 166, 167, 249, 250.
ast: 112.
ast_prefix: 38, 40, 112.
ast_prefix_1forest: 36, 39, 53, 89, 90, 95, 110, 111, 154.
auto_abort: 224, 241.
auto_delete: 224, 241.
autoabort: 66, 74, 79, 143, 224, 241.
Autoabort: 223, 240.
Autodelete: 223, 240.
autodelete: 66, 74, 79, 143, 224, 241.
banner: 22, 62.
Banner: 22.
banner_of_thread: 62.
begin: 12, 14, 21, 23, 24, 25, 26, 27, 28, 31, 32, 58, 61, 69, 70, 75, 81, 82, 88, 97, 100, 101, 104, 105, 106, 107, 120, 121, 124, 126, 128, 140, 143, 146, 153, 154, 158, 159, 161, 162, 163, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196, 197, 200, 201, 202, 203, 210, 214, 224, 229, 241, 246, 248, 249, 250.
*big_buf_*: 22, 46, 47, 51, 54, 60, 62, 65, 66, 67, 68, 73, 74, 78, 79, 80, 84, 86, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 142, 143, 145, 146, 147, 151, 152, 154, 158, 159, 160, 162, 163, 166, 167, 170, 171, 172, 173, 175, 179, 180, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 222, 223, 224, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 239, 240, 241, 242, 243, 246, 248, 249, 250, 251.
`BIG_BUFFER_32K`: 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 142, 151, 152, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 222, 223, 224, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 239, 240, 241, 242, 243, 246, 248, 249, 250, 251.
BIT_MAP_ITER_type: 161, 162.
BIT_MAP_type: 161.
brother: 91, 96, 147, 165, 166, 167, 250.
`BUFFER_SIZE`: 57.

BUILD_GRAMMAR_TREE: 5, 21, 23, 24, 25, 26, 27, 28, 32, 35.
bypassed_thd_eos_t: 167, 250.
c: 54, 86.
c_str: 12, 14, 22, 38, 39, 40, 41, 46, 47, 51, 54, 56, 60, 62, 64, 66, 67, 68, 70, 72, 74, 75, 77, 79, 80, 82, 84, 85, 86, 100, 101, 116, 117, 118, 119, 120, 123, 124, 126, 127, 129, 131, 132, 133, 134, 135, 137, 138, 140, 141, 143, 145, 146, 147, 148, 150, 151, 152, 154, 155, 157, 159, 160, 163, 165, 166, 167, 169, 171, 172, 173, 179, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 198, 200, 201, 202, 203, 207, 208, 209, 211, 212, 214, 215, 216, 217, 218, 219, 221, 222, 223, 225, 226, 227, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 242, 243, 244, 246, 248, 249, 250, 251, 252, 258.
c_string: 41, 62, 141, 251.
CAbs_lr1_sym: 5, 12, 13, 21, 23, 24, 25, 26, 27, 28, 32, 34, 35, 37, 38, 40, 41, 42, 54, 56, 58, 61, 69, 82, 92, 93, 98, 110, 111, 112, 135, 147, 148, 155, 165, 166, 167, 169, 198, 211, 212, 219, 227, 236, 244, 248, 250, 252, 258.
calc_cyclic_key: 109, 111, 112.
calc_la: 107.
call_thread_table_imp: 184, 185.
called_th: 166, 167.
called_thd: 249, 250.
called_thread_first_set: 249.
called_thread_name: 54, 56, 166, 167, 249, 250.
Calling_parser: 5, 21, 23, 24, 25, 26, 27, 28, 30, 32, 41.
case_stmt_parta: 154.
case_stmt_partb: 154.
case_stmt_partc: 154.
cc: 91, 93, 96, 98, 141.
cc_str: 141.
Ccweave_fsm_sdc: 62.
Ccweave_lhs_sdc: 82.
Ccweave_sdc: 58.
Ccweave_T_sdc: 70, 75, 81.
Ccweb_put_k_into_ph: 36.
Cenumerate_T_alphabet: 30.
Cerr_symbols_ph: 25.
Cfsm_phrase: 21.
CHAR: 5, 34.
class_fsm: 123.
class_name: 66, 74, 79.
classification: 84.
classsym: 66, 74, 79, 147, 159, 169, 200, 201, 202, 203, 209, 214, 221, 223, 229, 238, 240, 248.
clear: 143.
close: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
*closure_state_*: 14.
*closure_state_birthing_it_*: 12.
*closured_state_gening_it_*: 14.
Clr1_k_phrase: 27.
Cmd1_tokens: 34.
Cmpost_output: 22, 44, 48, 49, 50, 51, 55, 57, 58, 62, 70, 75, 81, 82, 83, 85, 86, 258.
CODE_PRESENCE_IN_ARBITRATOR_CODE: 127, 145.
comment: 62, 141, 251.
comment_data: 39, 40.
comments_can: 39.
*common_la_set_idx_*: 14, 107, 183.
COMMON_LA_SETS: 8, 106, 122, 158.
COMMON_LA_SETS_ITER_type: 106, 158.
COMMON_LA_SETS_type: 8.
Common_set1: 105.
COMMONIZE_LA_SETS: 107.
Cont_pos: 5, 21, 23, 24, 25, 26, 27, 28, 32, 42.
Cont_tok: 5, 21, 23, 24, 25, 26, 27, 28, 32, 42.
container: 34, 41.
content: 12, 13, 35, 38, 40, 56, 92, 93, 98, 147, 165, 166, 167, 249, 250.
convertMPtoPDF: 51, 259.
copy: 100.
copy_set: 104, 106.
Co2_lcl_opts: 34.
Cparallel_parser_phrase: 23.
Cpass3: 41.
cpp: 2.
Crc_phrase: 26.
crt_order: 31, 120, 121, 124, 126, 140, 143, 153, 170, 200, 201, 202, 203, 210, 214, 224, 229, 241, 246, 248.
crt_tree_of_1son: 35.
Crules_phrase: 32.
csi: 12.
csie: 12.
cstate: 12.
ct: 92.
CT_enum_phrase: 24.
Cterminals_phrase: 28.
ctor: 143, 257.
ctor_code: 143.
ctor_fsm: 123, 141.
cur_elem: 91, 93, 96, 98.
cur_elem_t: 91, 92, 93, 96, 98.
cur_node: 35.
cur_se_enum_no: 170, 171, 175, 176, 177, 178, 181, 183.
cur_state: 12, 107, 197, 248.

cur_sym: 57.
current_rule_no: 120.
current_token: 21, 23, 24, 25, 26, 27, 28, 32.
current_token_pos__: 21, 23, 24, 25, 26, 27, 28, 32.
cweave: 43, 44, 45, 47, 52, 58, 59, 63, 71, 74, 76, 82, 83, 84, 86, 257, 260, 261.
cweb: 3, 35, 36, 40, 43.
cweb_k: 36.
cweb_k_fsm: 36.
cweb_la_srce_expr: 62.
cweb_marker: 62, 66, 67, 68, 70, 74, 75, 79, 80, 81, 83.
Cweb_marker: 5, 39.
CWEB_MARKER: 3, 5, 8.
CYCLIC_USE_TABLE: 5, 111, 112.
CYCLIC_USE_TBL_ITER_type: 111, 112.
CYCLIC_USE_TBL_type: 5.
date: 141.
Date: 22.
DATE_AND_TIME: 2, 5, 115, 141, 258.
dc: 141.
dc_str: 141.
debug: 62, 141.
def_item: 214, 229.
def_of_state_s_shift_entries: 173.
default_case_and_end_sw: 153.
default_ns_use: 119, 138.
defed: 37.
delete_recycled_rules: 141.
Delete_tokens: 21, 34, 41.
determine_rhs_indirect_use_cnt: 110, 112.
determine_rhs_max_use_cnt: 111, 112.
determine_shift_element_name: 169, 170, 175.
dictionary: 200, 201, 202, 203, 214, 224, 229, 241.
Dimension: 49.
dir_map: 117, 123, 126, 127, 134, 141, 143, 146, 150, 157.
direct_use_cnt: 110.
directives_map: 61, 69, 117, 123, 134, 141, 150, 157, 214, 222, 224, 229, 239, 241.
dirs_tokens: 58, 61, 62, 69, 70, 75, 81, 82.
done: 248.
Drw_how: 51.
dtor: 143, 214, 229.
dtor_adr: 224, 241.
dtor_code: 143, 221, 238.
dtor_code_noabort: 221, 238.
dtor_fsm: 123.
dtor_name: 143.
dtor_nm: 143.
dtor_rtn_end: 141.
dtor_rtn_start: 141.
dtor_t: 221, 238.
DUMP_ERROR_QUEUE: 5.
E_v: 161.
e_v: 162.
ee: 249.
el: 101.
elem_cnt: 54, 55.
elem_list: 165, 166, 167, 183.
Elem_name: 51.
Elem_no: 50.
*elem_no_*: 51, 258.
elem_space: 88, 91, 93, 96, 98.
element_pos: 56.
element_walk: 36, 53.
elements_can: 53, 54.
emit_each_lr_state_s_tables: 197, 198.
empty: 12, 21, 23, 24, 25, 26, 27, 28, 30, 32, 34, 41, 83, 106, 127, 128, 145, 146, 154, 168, 259.
en_no: 161.
end: 12, 14, 31, 58, 61, 69, 70, 75, 81, 82, 84, 88, 90, 97, 100, 101, 104, 105, 106, 107, 112, 117, 120, 121, 123, 124, 126, 127, 128, 134, 140, 141, 143, 145, 146, 150, 153, 154, 157, 158, 159, 161, 162, 163, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 191, 197, 200, 201, 202, 203, 210, 214, 222, 224, 229, 239, 241, 246, 248, 249, 250.
end_class_fsm: 123.
end_enumeration_of_rules_sbrules: 120.
end_fs_list_of_Tes: 248.
end_holding_kcweb: 35, 36.
end_rtn: 153.
end_rule_def: 125, 128.
end_t_alphabet: 210.
end_transitive_thread_list: 249.
end_used_thread_list: 250.
endl: 12, 14, 21, 23, 24, 25, 26, 27, 28, 30, 31, 32, 38, 39, 40, 41, 46, 51, 60, 62, 66, 67, 68, 74, 79, 80, 100, 101, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 143, 145, 146, 147, 150, 151, 152, 153, 154, 157, 159, 160, 162, 163, 166, 171, 172, 173, 174, 179, 182, 183, 186, 187, 189, 190, 192, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 223, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 240, 242, 243, 246, 248, 249, 250, 251.
entry_symbol_literal: 12, 187.
Enum: 56.
enum_class_name_format: 79.
enum_id: 31, 91, 109, 120, 147, 161, 170, 200, 201, 202, 203, 246, 248.
enum_list_end: 205.

enum_list_start: 205.
enum_name: 74.
enum_name_for_format: 66, 74.
enum_ph: 24, 93, 98, 116, 119, 120, 121, 122, 123, 138, 140, 141, 151, 204, 205, 206, 207, 208, 211, 212, 215, 218, 225, 226, 230, 235, 242, 243, 246.
enum_phrase: 24.
enum_syms: 30.
enum_syms_fsm: 30.
enumer: 42.
enumerate_errors: 202, 205.
enumerate_item: 200, 201, 202, 203.
enumerate_lrk: 200, 205.
enumerate_rc: 201, 205.
enumerate_T: 203, 205.
enumerated_id: 56, 110, 111, 112.
enumerated_id__: 12, 13, 35, 38, 40, 42, 54, 57, 79, 84, 91, 92, 96, 147, 165, 166, 167, 169, 250.
enumeration_define_list: 205, 206.
enumeration_define_structure: 206, 207.
enumeration_include_guard_for_header: 208, 211.
enumeration_namespace_for_header: 207, 208.
enumeration_of_rules_sbrules: 120.
enumeration_summary_for_struct: 204, 205.
eolr: 23, 104.
eolr_set: 105.
eos: 14, 56.
eos_can: 154.
eos_filter: 154.
eos_thd_or_proc: 147.
eos_walk: 154.
eosubrule: 31.
epsilon: 38, 57, 91, 96, 97, 100.
Err: 71.
err: 42, 72, 74, 84.
Err_already_processed_error_phase: 25.
Err_already_processed_fsm_phase: 21.
Err_already_processed_lrk_phase: 27.
Err_already_processed_pp_phase: 23.
Err_already_processed_rc_phase: 26.
Err_already_processed_rule_phase: 32.
Err_already_processed_T_enum_phase: 24.
Err_already_processed_T_phase: 28.
Err_bad_enum_filename: 211, 212.
Err_bad_errors_hdrfilename: 219, 236.
Err_bad_errors_impfilename: 227, 244.
Err_bad_filename: 34, 41, 258.
Err_bad_fsmcpp_filename: 148.
Err_bad_fsmheader_filename: 135.
Err_bad_fsmsym_filename: 155.
Err_bad_fsmtbl_filename: 198, 252.
Err_nested_files_exceeded: 41.
Err_not_kw_defining_grammar_construct: 42.
err_ph: 25, 116, 138.
Err_phrase: 72, 75.
ERR_sw: 5, 34.
*err_sw_*: 34.
err_symbols_ph: 25.
err_symbols_ph_th: 25.
error_exit: 21, 23, 24, 25, 26, 27, 28, 32.
error_queue: 21, 23, 24, 25, 26, 27, 28, 30, 32, 41.
Error_queue: 5, 34, 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
error_symbols_attributes: 25.
error_symbols_phrase: 25.
Errors: 5.
errors_imp_dtor: 221, 222, 239.
errors_imp_implementation: 222, 224.
errors_imp_shellimplementation: 223, 224.
errors_include_files_for_header: 218, 219.
errors_include_files_for_imp: 226, 227.
errors_include_guard_for_header: 217, 219.
errors_loop_thru_and_gen_defs_for_header: 214, 216.
errors_loop_thru_and_gen_defs_for_imp: 224, 227.
errors_namespace_for_header: 216, 217.
errors_ph: 214, 215, 216, 217, 218, 219, 224, 225, 226, 227, 235.
errors_use_enum_namespace_for_header: 215, 216.
errors_use_enum_namespace_for_imp: 225, 227.
er1: 21, 23, 24, 25, 26, 27, 28, 32.
et: 88, 91, 96.
expr_prescan_cweb: 62.
external_file_id__: 35, 38, 40, 41.
externs_and_thread_tbl_defs: 163, 198.
e1: 102.
E1: 102.
e1_t_def: 102.
e1s: 102.
e2: 102.
E2: 102.
e2_t_def: 102.
e2s: 102.
failed_rtn: 141.
Failure: 21, 23, 24, 25, 26, 27, 28, 30, 32, 41, 42.
false: 35, 83, 102, 105, 107, 147, 154, 170, 171, 172, 175, 224, 241.
fc: 141.
fc_str: 141.
fcp: 61, 246, 251.
fe: 12.
File: 5.
FILE_CNT__: 41, 254.
File_include: 5, 41.

file_name: 41.
File_name: 115.
file_ok: 34, 41.
FILE_TBL__: 254.
*file_to_compile_*: 34.
filename_id: 64, 72, 77, 116, 133, 135, 137, 148, 155, 198, 208, 211, 212, 217, 218, 219, 226, 227, 234, 235, 236, 243, 244, 246, 252.
filter: 39, 53, 110, 111, 112.
find: 84, 90, 104, 105, 111, 112, 117, 123, 126, 127, 134, 141, 143, 145, 146, 150, 154, 157, 161, 165, 179, 186, 188, 189, 190, 191, 192, 195, 196, 214, 221, 222, 224, 229, 238, 239, 241, 250.
find_common_la_set_idx: 104, 106, 107, 263.
find_sym_in_stbl: 104, 105.
first: 12, 162, 170, 175, 176, 181, 183.
first_el_t: 92.
first_or_2nd_prt: 170, 171, 172, 175.
first_set: 100, 246, 248.
first_set_for_threads: 247.
FIRST_SET_ITER_type: 246, 248.
first_set_rules: 94.
FIRST_SET_type: 246, 248.
first_time: 12.
fn: 135, 148, 155, 198, 211, 219, 227, 236, 244, 252.
Fn: 246.
fn_literal: 212.
fname: 22, 62.
follow_element: 12.
*follow_set_*: 12.
FOLLOW_SETS_ITER_type: 12.
forward_ref: 124.
forward_refs_of_rules_declarations_for_header: 124, 131.
fqtnm: 250.
fs: 100, 246, 248.
fs_list_of_Tes: 248.
fs_set: 248.
fs_sort_criteria: 100, 102.
fs_T_in_list: 248.
fsc_prelude: 246.
fsc_prelude_imp: 246, 252.
Fsm: 22, 44, 46, 47, 48, 49, 50, 51, 54, 55, 58, 60, 62, 64, 65, 66, 67, 68, 70, 72, 73, 74, 75, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 258.
fsm: 21, 42, 59, 60, 61.
fsm_attributes: 21.
fsm_class: 60, 117, 120, 121, 122, 123, 130, 132, 133, 134, 140, 141, 142, 143, 150, 151, 152, 157, 158, 160, 163, 166, 170, 171, 172, 173, 175, 179, 180, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195.
fsm_class_declaration_for_header: 123, 131.
fsm_class_directives: 21.
fsm_class_implementation: 141, 148.
fsm_class_phrase: 60, 61, 117, 120, 121, 122, 123, 130, 132, 133, 134, 140, 141, 142, 150, 151, 152, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 246.
fsm_class_phrase_th: 21, 141.
fsm_comments: 251.
fsm_comments_about_la_sets_and_states: 122, 123.
fsm_cpp_includes: 137, 148, 155, 198.
fsm_enum_rules_subrules_for_header: 120, 123.
fsm_id: 62, 141.
fsm_map_subrules_to_rules_for_header: 121, 123.
fsm_map_subrules_to_rules_table_imp: 140, 141.
fsm_name: 62.
fsm_ph: 116, 117, 120, 121, 122, 123, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 145, 148, 150, 151, 152, 155, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 198, 252.
FSM_PHASE: 35.
Fsm_phrase: 60, 61, 62.
fsm_phrase: 21.
fsm_phrase_th: 21.
fsm_reduce_rhs_of_rule_implementation: 152, 155.
GEN_CALLED_THREADS_FS_OF_RULE: 5, 93.
GEN_FS_OF_RULE: 2, 5, 98.
gen_Tes_literals: 210, 212.
gen_Tes_literals_per_spec_voc: 209, 210.
*gened_date_time_*: 46, 47, 64, 72, 77, 258.
GET_CMD_LINE: 5.
get_spec_child: 88, 146.
get_younger_sibling: 92.
get_1st_son: 249.
*goto_state_*: 14, 107, 167, 170, 175, 179, 186, 188, 189, 190, 191, 192, 195, 196.
goto_state_no: 170, 171, 175, 179, 186, 188, 189, 190, 191, 192, 195.
gp: 35.
*grammar_filename_prefix_*: 62, 258.
grammar_header_includes: 116, 133.
grammar_include_guard_for_header: 133, 135.
Grammar_name: 22.
grammar_s_comments_for_linker_doc_imp: 251, 252.
grammar_s_enumerate: 14, 96.
Grammar_to_compile: 5, 34.
GRAMMAR_TREE: 5, 8, 35.
gt: 21, 23, 24, 25, 26, 27, 28, 32.
gw_sdc: 214, 229.
holding_kcweb: 35, 36.
id: 12, 13, 14, 54, 56.

id__: 37, 38, 40.
idc: 110, 111.
identifier: 38, 42, 54, 56, 60, 62, 64, 72, 77, 116, 118, 119, 120, 123, 129, 131, 132, 133, 135, 137, 138, 140, 141, 143, 145, 148, 151, 152, 155, 160, 163, 166, 167, 171, 172, 173, 179, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 198, 207, 208, 211, 212, 215, 216, 217, 218, 219, 225, 226, 227, 230, 233, 234, 235, 236, 242, 243, 244, 246, 249, 250, 252.
idx: 106, 127, 145, 146, 158, 160.
ie: 12, 14, 58, 61, 69, 82, 101, 104, 106, 120, 121, 124, 126, 140, 143, 153, 158, 163, 200, 201, 202, 203, 214, 224, 229, 241, 246, 248, 250.
IE: 209.
ifstream: 34, 41, 254.
ii: 100.
iie: 100.
imp_of_shift_entries_end: 174.
imp_of_state_s_shift_entries_begin: 173.
imp_of_state_s_shift_1st_entry: 171, 172.
imp_of_state_s_shift_2nd_entry: 171.
imp_t: 222, 239.
In: 10.
include_files: 218, 226, 235, 243.
include_list: 116, 137.
indirect_use_cnt: 110.
initialize: 36, 58, 62, 70, 75, 81, 82.
insert: 39, 53, 90, 93, 98, 104, 105, 110, 111, 112, 127, 154, 250.
INT: 5, 21, 23, 24, 25, 26, 27, 28, 32, 42.
INT_SET_type: 39, 53, 93, 98.
INT_STR_MAP_ITER_type: 84.
intro_comment: 115, 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
invisible_shift_entry: 179, 186, 190.
ios: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
ios_base: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
ip1: 21, 23, 24, 25, 26, 27, 28, 32.
Item: 5, 35, 36.
iterator: 88, 93, 97, 98, 200, 201, 202, 203, 209, 214, 224, 229, 241, 249, 250.
*its_enum_id_*: 57, 177.
its_rd: 110, 111.
its_rule_def: 12, 14, 56, 91, 96, 110, 111, 147, 165, 183.
*its_state_*: 12.
its_t_def: 12, 14, 56, 91, 147.
je: 12, 120, 121, 128, 140, 146, 154, 159, 165, 166, 167, 181, 183, 246, 248.
join_sts: 35.
just_walk_functr: 36, 39, 53, 89, 90, 93, 95, 98.
k_cweb: 62.
KCHARP: 22, 46, 47, 51, 53, 57, 60, 62, 65, 66, 67, 68, 73, 74, 78, 79, 80, 85, 86, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 143, 145, 146, 147, 151, 152, 153, 154, 159, 160, 162, 163, 166, 167, 171, 172, 173, 174, 179, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 223, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 240, 242, 243, 246, 248, 249, 250, 251.
ke: 12, 161, 165, 166, 167, 183, 250.
key: 109.
Keyword: 5, 42.
keyword: 37, 42.
kkk: 154.
kl: 162.
kle: 162.
kw: 37.
Kw: 37.
kw_in_stbl: 37.
kwkey: 37.
la: 62, 97.
*La_*: 106.
la_bndry: 62, 97.
la_first_set: 62, 97.
la_ph: 62.
la_set: 158, 159, 161.
*la_set_*: 14, 107.
la_set_entry: 162.
la_set_entry_literal: 159.
la_set_hdr: 160.
la_set_hdr_end: 162.
LA_SET_ITER_type: 14, 104, 105, 159, 161.
la_set_to_compare_against: 105.
LA_SET_type: 104, 105, 106, 158.
lai: 97.
laie: 97.
lcl_options_tokens: 34.
len: 10, 54, 86.
length: 10, 54, 86.
lex: 42, 108, 112, 130, 139, 256.
lhs_directives_map: 82, 126, 143, 154.
lhs_sdc_emit: 82.
lhs_sdc_fsm: 82.
line_no__: 38, 40.
list_of_native_Tes_in_fs_imp: 247, 248, 252.
literal: 209.
literal_name: 12.

load_kw_into_tbl: 37.
LOAD_YACCO2_KEYWORDS_INTO_STBL: 5, 37.
lr: 42.
lr_k: 27.
lr_ph: 27, 218, 226, 235, 243.
lrclog: 12, 14, 21, 23, 24, 25, 26, 27, 28, 30, 31, 32, 38, 40, 41.
lrk: 76, 77, 79, 84.
lrk_ph: 116, 138.
Lrk_phrase: 77, 80, 81.
lrk_sufx_code: 80.
LR1_ALL_SHIFT_OPERATOR: 147, 177, 179, 186, 189.
LR1_Eog: 110, 111, 112.
LR1_EOLR_LITERAL: 104, 105.
LR1_FSET_TRANSIENCE_OPERATOR: 177, 179, 186, 191, 196.
LR1_FSET_TRANSIENCE_OPERATOR_LITERAL: 248.
LR1_INVISIBLE_SHIFT_OPERATOR: 147, 177, 179, 186, 190, 248.
lr1_k_attributes: 27.
lr1_k_phrase: 27.
lr1_k_phrase_th: 27.
LR1_PARALLEL_OPERATOR: 91, 164, 177, 179, 186, 188, 195, 246, 248, 250.
LR1_QUESTIONABLE_SHIFT_OPERATOR: 147, 177, 179, 186, 192.
LR1_STATES: 5, 107, 163, 197, 248, 250.
*lt_*: 40.
macros: 22, 47.
make_copy_of_la_set: 104, 106.
make_pair: 112.
Max_cweb_item_size: 22, 57, 60, 62, 66, 70, 74, 75, 79, 81, 85, 101.
Max_no_subrules: 93.
MAX_USE_CNT_RxR: 5, 110, 112.
MAX_USE_CNT_RxSkeletion: 2.
*merges_*: 12.
MERGES_ITER_type: 12.
mntr_directives_map: 82, 127, 145.
mono: 246.
*mp_dimension_*: 49, 258.
*mp_filename_*: 46, 258.
Mp_obj_name: 49, 50.
mp_obj_name: 51.
mp_prefix_date: 46.
mp_prefix_date_value: 46.
mp_prefix_file: 46.
mp_prefix_file_value: 46.
mp_subrule_elems_literal: 51.
mp_subrule_elems_RorT: 51.
mp_subrule_elems_vname: 51.
mp_xlate_name: 51.
mpost: 10, 45, 46, 49, 50, 51, 257.
MPOST_CWEB_calc_mp_obj_name: 50, 51.
MPOST_CWEB_crt_rhs_sym_str: 57.
MPOST_CWEB_EMIT_PREFIX_CODE: 45, 48, 257.
MPOST_CWEB_gen_dimension_name: 49, 50.
MPOST_CWEB_gen_sr_elem_xrefs: 52, 55.
MPOST_CWEB_LOAD_XLATE_CHRS: 44.
MPOST_CWEB_should_subrule_be_printed: 83.
MPOST_CWEB_woutput_sr_sdcode: 58.
MPOST_CWEB_wrt_Err: 71, 75.
MPOST_CWEB_wrt_fs: 62.
MPOST_CWEB_wrt_fsm: 59, 62.
MPOST_CWEB_wrt_lrk: 76.
MPOST_CWEB_wrt_Lrk: 81.
MPOST_CWEB_wrt_mp_rhs_elem: 51.
MPOST_CWEB_wrt_rule_s_lhs_sdc: 82.
MPOST_CWEB_wrt_T: 63, 70.
MPOST_CWEB_xlated_symbol: 56, 57.
MPOST_CWEB_xref_refered_rule: 54, 86.
MPOST_CWEB_xref_refered_T: 54, 85.
mpost_output: 43, 257.
name: 66, 74, 79, 81, 84, 85.
Name_space: 22.
name_with_bkslash: 54, 86.
namespace_end: 216, 233.
namespace_for_header: 131, 133.
namespace_id: 62, 64, 72, 77, 119, 120, 131, 138, 145, 151, 207, 215, 216, 225, 230, 233, 242, 246.
namespace_start: 216, 233.
nc: 88, 93, 98.
Nested_file_cnt_limit: 41.
*next_state_element_*: 57, 165, 166, 167, 178, 181, 195, 196, 250.
NO: 34, 41, 97.
NO_BITS_PER_SET_PARTITION: 160, 161.
no_la_set_entries: 162.
NO_LR1_STATES: 5, 122.
no_of_elems: 146, 154.
no_of_la_sets_and_states: 122.
no_of_reduces: 181, 182, 194.
no_of_rules: 109.
*no_of_rules_*: 258.
no_of_shift_items: 170, 173, 175, 176, 193.
*no_of_subrules_*: 258.
no_of_Tes_in_fs: 248.
no_of_threads: 165, 166.
no_parms: 146, 147, 154.
no_rules: 120, 130, 139, 141.
no_subrules: 146.
*no_subrules_per_rule_*: 258.
No_t: 22.

no_Tes_in_list: 248.
no_threads_in_list: 249.
Node: 5, 38, 40.
nodes_visited: 88.
nos: 12, 14.
nosrs: 146.
not_start_rule: 97.
not_underline_symbol: 57.
npos: 127, 143, 145, 146, 221, 238.
ns: 54, 56, 167, 249, 250.
NS_cweave_fsm_sdc: 62.
NS_cweave_lhs_sdc: 82.
NS_cweave_sdc: 58.
NS_cweave_T_sdc: 70, 75, 81.
NS_cweb_put_k_into_ph: 36.
NS_enumerate_T_alphabet: 30.
NS_err_symbols_ph: 25.
NS_fsm_phrase: 21.
NS_lr1_k_phrase: 27.
NS_mpost_output: 22, 44, 48, 49, 50, 51, 55, 57, 58, 62, 70, 75, 81, 82, 83, 85, 86.
ns_name: 22, 62.
ns_of_enum_end: 207.
ns_of_enum_start: 207.
ns_of_grammar_end: 131.
ns_of_grammar_start: 131.
NS_o2_lcl_opts: 34.
NS_parallel_parser_phrase: 23.
NS_pass3: 41.
NS_rc_phrase: 26.
NS_rules_phrase: 32.
NS_T_enum_phrase: 24.
NS_terminals_phrase: 28.
NS_yacco2_err_symbols: 34, 41.
NS_yacco2_T_enum: 35, 38, 39, 40, 42, 53, 93, 98, 109, 110, 111, 112, 164.
NS_yacco2_terminals: 5, 21, 23, 24, 25, 26, 27, 28, 32, 35, 38, 39, 40, 41, 42, 53, 102, 109, 110, 111, 112, 207, 208.
nxtsym_1: 57.
nxtsym_2: 57.
nxt1_se: 57.
nxt2_se: 57.
nxt3_se: 57.
oc: 141.
oc_str: 141.
Ofile: 22, 100, 101, 102.
ofstream: 5, 22, 39, 102, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 129, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 148, 150, 151, 152, 155, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 197, 198, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 221, 222, 223, 224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 241, 242, 243, 244, 246, 248, 249, 250, 251, 252.
*omp_file_*: 46, 51, 258.
One: 161.
op: 143.
op_code: 143.
`OP_ENUMERATION_HEADER`: 5, 199, 211.
`OP_ERRORS_CPP`: 5, 227.
`OP_ERRORS_HEADER`: 2, 5, 213, 219, 220.
op_failed: 123.
Op_file: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
`OP_FSC_FILE`: 5, 245, 252.
`OP_GRAMMAR_CPP`: 5, 136, 148.
`OP_GRAMMAR_HEADER`: 2, 5, 114, 135.
`OP_GRAMMAR_SYM`: 5, 149, 155.
`OP_GRAMMAR_TBL`: 5, 156, 198.
op_rtn: 141.
Op_str: 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 142, 143, 145, 146, 147, 150, 151, 152, 153, 154, 157, 158, 159, 160, 162, 163, 166, 167, 170, 171, 172, 173, 174, 175, 179, 180, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 222, 223, 224, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 239, 240, 241, 242, 243, 246, 248, 249, 250, 251.
OP_T_Alphabet: 5, 212.
op_template: 115.
`OP_USER_T_CPP`: 5, 244.
`OP_USER_T_HEADER`: 2, 5, 228, 236, 237.
open: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
options: 34.
opts_fsm: 34.
op1: 21, 23, 24, 25, 26, 27, 28, 32.
Out: 10.
out: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
output_all_others_state_s_shift_table: 175, 197.
output_cweb_macros_and_banner: 22, 64, 72, 77.
output_la_sets: 158, 198.
output_lr_state: 187, 197.
output_meta_shifts_separately: 179, 197.
output_questionable_shift: 186.
output_state_s_called_procedure_table: 185, 197.
output_state_s_called_threads_table: 184, 197.

output_state_s_reduced_table: 180, 197.
output_1st_state_s_shift_table: 170, 197.
overall_subrules_nos: 120, 121, 140.
*ow_err_file_*: 72, 73, 74, 75.
*ow_file_*: 47, 54, 58, 60, 62, 82, 84, 86, 258.
*ow_lrk_file_*: 77, 78, 79, 80, 81.
*ow_t_file_*: 64, 65, 66, 67, 68, 70.
O2_ERROR_PHASE: 5, 8, 25, 30, 116, 138, 202, 210, 214, 215, 216, 217, 218, 219, 224, 225, 226, 227, 235.
o2_externs: 2.
*o2_externs_*: 4.
O2_FSM_PHASE: 5, 8, 21, 30, 116, 117, 120, 121, 122, 123, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 148, 150, 151, 152, 155, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 198, 246, 251, 252.
O2_library_file: 116, 218, 235.
O2_LRK_PHASE: 5, 8, 27, 30, 116, 138, 200, 210, 218, 226, 235, 243.
O2_PP_PHASE: 5, 8, 23, 30, 62, 97, 118, 129, 151, 246.
O2_RC_PHASE: 5, 8, 26, 30, 116, 138, 201, 210.
O2_RULES_PHASE: 5, 8, 29, 30, 32, 109, 120, 121, 124, 125, 130, 139, 140, 141, 142, 151, 152, 170, 246, 248, 249.
O2_T_ENUM_PHASE: 5, 8, 24, 30, 93, 98, 109, 116, 119, 120, 121, 122, 123, 138, 140, 141, 151, 204, 205, 206, 207, 208, 211, 212, 215, 218, 225, 226, 230, 235, 242, 243, 246.
O2_T_PHASE: 5, 8, 28, 30, 116, 138, 203, 210, 229, 230, 231, 232, 233, 234, 235, 236, 241, 242, 243, 244.
O2_xxx: 2.
O2_xxx_PHASE: 30.
o2externs: 2.
paired_set: 161, 162.
parallel_entry: 179, 186, 188.
parallel_mntr: 82, 127, 144.
parallel_monitor_phrase: 29.
parallel_parser_attributes: 23.
parallel_parser_phrase: 23.
parallel_parser_phrase_th: 23.
parm_name: 147.
parse: 21, 23, 24, 25, 26, 27, 28, 30, 32, 34, 36, 41, 58, 62, 70, 75, 81, 82.
Parser: 5, 21, 23, 24, 25, 26, 27, 28, 30, 32, 34, 36, 41, 42, 58, 62, 70, 75, 81, 82.
parser__: 258.
part_no: 162.
part1_key_of_rule_use: 109.
part2_key_of_against_rule: 109.
pass3: 35, 41, 42.
pdftex: 85, 257.
per_rule_s_table_ref_type: 139.
ph: 200, 201, 202, 203.
phe: 210.
phlr: 210.
phrase_can: 36.
Phrase_to_parse: 21, 23, 24, 25, 26, 27, 28, 32.
phrase_tree: 21, 23, 24, 25, 26, 27, 28, 32, 249.
phrc: 210.
pht: 210.
pop_back: 41.
*pos_*: 37.
pos_in_line__: 38, 40.
possible_thread_procedure_declaration: 129, 131.
potential_idc_cnt: 111.
pp: 57.
pp_funct: 118, 129, 151, 246.
pp_map: 145.
pp_ph: 118, 129, 151.
pp_phrase: 127, 145.
pp_thd_nm: 246.
pparser: 23, 42.
ppm: 82.
ppp: 246.
*pr_*: 56.
prescan_mpname_for_cweb: 10, 51.
prev_t: 56.
*previous_state_element_*: 177, 178.
Print_dump_state: 2, 5, 11.
PRINT_GRAMMAR_TREE: 5, 40.
PRINT_RULES_TREE_STRUCTURE: 2, 5, 38.
proc_thread_definition: 151.
procedure_call_entry: 179, 186, 191.
process_error_symbols_phrase: 2, 25, 42.
process_fsm_phrase: 2, 21, 42.
PROCESS_INCLUDE_FILE: 2, 5, 41.
PROCESS_KEYWORD_FOR_SYNTAX_CODE: 2, 3, 5, 42.
process_lr1_k_phrase: 2, 27, 42.
process_parallel_parser_phrase: 2, 23, 42.
process_rc_phrase: 2, 26, 42.
process_rules_phrase: 2, 29, 32, 42.
process_T_enum_phrase: 2, 24, 42.
process_terminals_phrase: 2, 28, 42.
process_xxx_phrase: 3.
productions: 29.
prt_ast_functor: 38, 40.
PRT_RULE_S_FIRST_SET: 2, 5, 102.
PRT_sw: 5, 34.
*prt_sw_*: 34.
ps_fn: 41.
PTR_LR1_eog__: 39, 41, 54, 89, 90, 95, 154.

push_back: 34, 58, 61, 69, 82, 100, 106, 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
px__: 146.
p1: 21, 23, 24, 25, 26, 27, 28, 32.
p1_tokens: 41.
p10__: 146.
p3_fsm: 41.
Q: 49, 161.
questionable_shift_entry: 179, 186, 192.
R: 49, 161.
r_def: 38, 86.
R_rule: 86.
R_T: 85.
ran_array: 100, 101.
rc: 42, 84.
rc_attributes: 26.
rc_ph: 26, 116, 138.
rc_phrase: 26.
rc_phrase_th: 26.
rc_pos__: 38, 40.
rd: 12, 14, 31, 57, 86, 120, 121, 124, 126, 127, 128, 140, 143, 144, 145, 146, 153, 154, 165, 169, 246, 248, 249.
rdef: 90, 91, 96.
rdlhs_map: 154.
recursive: 111, 112.
recycle_bin__: 41.
reduce: 79.
reduce_def: 182.
reduce_entry: 194.
reduce_imp_begin: 182.
reduce_imp_end: 182.
reduce_imp_1st_entry: 183.
reduce_imp_2nd_entry: 183.
reduce_no: 183.
reduce_operator: 79.
reduce_rhs: 123.
reduce_rhs_of_rule: 120, 149.
reduce_rhs_of_rule_hdr: 152.
*reduced_state_*: 14.
refered_rule: 86.
refered_T: 85.
report_card: 37, 104, 105.
reserve: 100.
result: 106.
rhs_uc: 112.
rhsc: 146.
ri: 31.
rie: 31.
rlhs: 126, 143, 154.
rlp: 82.
rr: 12, 14, 54, 56, 91, 96, 110, 111, 147.
rrrr: 79.
rt: 12, 14, 54, 56, 91, 96, 147.
*rt_*: 40, 57.
rtned_T_t: 250.
rtree: 89, 90, 95.
rule: 42.
Rule_def: 5, 82, 95, 96, 97, 98, 100, 102.
rule_def: 5, 12, 14, 31, 57, 82, 86, 91, 93, 96, 98, 102, 109, 110, 111, 112, 120, 121, 124, 126, 140, 143, 153, 165, 170, 246, 248, 249.
*rule_def_*: 258.
rule_def_ctor: 126, 143.
rule_def_dtor: 126, 143.
rule_def_dtor_noabort: 143.
rule_def_imp: 143.
rule_def_op: 126, 143.
rule_def_phrase: 29.
RULE_DEFS_TBL_ITER_type: 31, 120, 121, 124, 126, 140, 143, 153, 246, 248.
RULE_ENO: 5, 8.
rule_enum: 120.
Rule_in_stbl: 38, 86.
rule_in_stbl: 38, 86.
rule_it: 65, 73, 78.
rule_lhs: 82, 126, 143, 154.
rule_lhs_phrase: 29, 142.
rule_name: 12, 14, 38, 56, 82, 86, 100, 120, 124, 126, 127, 143, 145, 146, 147, 154, 165, 169, 172, 183.
rule_no: 31, 97, 140.
*rule_no_*: 12, 50, 51, 258.
rule_ph: 32.
rule_s_arbitrator: 127.
rule_s_arbitrator_imp: 145.
rule_s_ctor: 154.
rule_s_tree: 89, 90, 95.
Rule_use: 5, 109, 112.
rule_use_enum_rel0: 109.
rule_use_key: 111, 112.
rule_use_skeleton: 112.
rule_walk: 39.
rules: 29.
rules_alphabet: 109, 120, 130, 139, 141.
rules_class_defs_for_header: 125, 131.
Rules_enum: 31.
RULES_HAVING_AR: 8, 127, 165.
RULES_HAVING_AR_ITER_type: 165.
RULES_HAVING_AR_type: 8.
rules_in_elem_space: 90, 93, 98.
rules_ph: 120, 121, 124, 125, 126, 130, 139, 140, 141, 142, 143, 151, 152, 153, 170, 246, 248, 249.
rules_phrase: 29, 32.

rules_phrase_seen: 35, 36.
rules_phrase_th: 29.
rules_recycled_table_type: 130, 139.
rules_reuse_table_declaration: 130, 131.
rules_reuse_table_implementation: 139, 148.
rules_subrules_implementations: 142, 148.
rules_tree: 249.
rules_use_can: 110, 111, 112.
rules_use_cnt: 108, 112, 130, 139.
running_direct_use_cnt: 111.
running_indirect_use_cnt: 111.
S_CONFLICT_STATES_ITER_type: 12.
S_FOLLOW_SETS_ITER_type: 12.
s_rule: 170, 172.
s_rule_enum_no: 170, 172.
S_VECTOR_ELEMS_ITER_type: 12, 107, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196, 250.
S_VECTOR_ELEMS_type: 165, 166, 167, 183, 250.
S_VECTORS_ITER_type: 12, 107, 165, 167, 170, 175, 179, 181, 186, 188, 250.
S_VECTORS_type: 250.
SDC_arbitrator_code: 127, 145.
SDC_constructor: 126, 141, 143, 154.
SDC_destructor: 126, 141, 143, 214, 222, 229, 239.
sdc_emit: 58, 62, 70, 75, 81.
SDC_failed: 141.
sdc_fsm: 58, 62, 70, 75, 81.
sdc_map: 214, 222, 224, 229, 239, 241.
SDC_MAP_ITER_type: 58, 61, 69, 82, 117, 123, 126, 127, 134, 141, 143, 145, 146, 150, 154, 157, 214, 221, 222, 224, 229, 238, 239, 241.
SDC_MAP_type: 58, 61, 69, 82, 83, 117, 123, 126, 127, 128, 134, 141, 143, 145, 146, 150, 154, 157, 214, 222, 224, 229, 239, 241.
SDC_op: 126, 141, 143, 146, 154, 214, 229.
SDC_user_declaration: 123, 126, 214, 229.
SDC_user_imp_sym: 150.
SDC_user_imp_tbl: 157.
SDC_user_implementation: 141, 143, 224, 241.
SDC_user_prefix_declaration: 117.
SDC_user_suffix_declaration: 134.
sdrmap: 128, 154.
se: 12, 13, 14, 57, 107, 161, 165, 166, 167, 170, 175, 176, 177, 178, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196.
second: 12, 58, 61, 69, 70, 75, 81, 82, 84, 107, 111, 112, 161, 162, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196, 214, 221, 222, 229, 238, 239, 250.
seli: 12, 107, 170, 175, 176, 179, 181, 186, 188, 189, 190, 191, 192, 195, 196.
selie: 12, 107.
set: 93, 97, 98, 248, 249, 250.
set_external_file_id: 34, 35, 41.
SET_FILTER_type: 110, 111, 112.
set_line_no: 34.
set_line_no_and_pos_in_line: 35, 41, 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
set_ok: 107.
set_pos_in_line: 34.
set_rc: 21, 23, 24, 25, 26, 27, 28, 32, 42.
set_stop_parse: 258.
set_who_created: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
Set1: 104.
set1i: 105.
set1ie: 105.
set1T: 105.
Set2: 105.
set2i: 105.
set2ie: 105.
set2T: 105.
sf: 146.
sfi: 12.
sfie: 12.
shell_of_def_item: 214, 223, 229, 240.
shift_elem_literal: 170, 171, 175.
shift_entry: 193.
si: 107, 197, 248, 250.
sie: 107, 197, 250.
signal_guard_end: 133, 208, 217, 234.
signal_guard_start: 133, 208, 217, 234.
size: 41, 62, 64, 72, 77, 100, 105, 109, 120, 122, 130, 139, 141, 162, 181, 249, 250.
size_type: 127, 143, 145, 146, 221, 238.
`SMALL_BUFFER_4K`: 66, 74, 79.
sprintf: 22, 46, 47, 51, 54, 56, 57, 60, 62, 65, 66, 67, 68, 73, 74, 78, 79, 80, 84, 86, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 143, 145, 146, 147, 151, 152, 154, 159, 160, 162, 163, 166, 167, 171, 172, 173, 179, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 223, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 240, 242, 243, 246, 248, 249, 250, 251.
sr_def: 88.
*sr_def_element_*: 57, 170, 175.
sr_dirs: 58, 61, 69, 82, 83.
*sr_element_*: 12, 13, 57, 165, 166, 167, 250.
sr_filter: 89, 90, 93, 95, 98.
sr_ph: 120, 121, 128, 140, 146, 154.

sr_t: 88, 112, 146, 147, 154.
srd: 120, 128, 146, 154.
srdef_t: 88.
SRule_t: 110, 111.
stable_sort: 100.
start_err_enumerate: 204.
start_lrk_enumerate: 204.
start_of_rules: 109.
START_OF_RULES_ENUM: 5, 8, 31.
start_pos: 21, 23, 24, 25, 26, 27, 28, 32.
start_rc_enumerate: 204.
Start_rule: 5, 89, 92, 93.
start_rule_def: 126.
start_rule_def_t: 249.
start_state: 250.
start_T_enumerate: 204.
State: 5, 11, 12, 170, 173, 175, 176, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196.
state: 5, 11, 12, 107, 163, 170, 175, 179, 180, 184, 185, 186, 187, 197, 248, 250.
state_cnt: 163, 166.
state_element: 12, 57, 107, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196, 250.
state_elems_wparallelism: 250.
state_entry: 187.
state_extern: 132, 163.
*state_no_*: 12, 14, 170, 173, 175, 179, 182, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 197.
*state_s_conflict_state_list_*: 12.
*state_s_follow_set_map_*: 12.
state_s_thread_tbl_def: 166.
state_s_thread_tbl_end: 166.
state_s_thread_tbl_entry: 166, 167.
state_s_thread_tbl_imp: 166.
*state_s_vector_*: 12, 107, 165, 166, 167, 170, 175, 176, 179, 181, 183, 186, 188, 189, 190, 191, 192, 195, 196, 250.
state_1_extern: 132, 133.
STATES_ITER_type: 107, 163, 197, 248, 250.
STATES_type: 5.
stbl_idx: 37.
STBL_T_ITEMS: 5, 8.
STBL_T_ITEMS_type: 5, 8.
stc: 67, 68, 80.
std: 5, 10, 12, 14, 21, 22, 23, 24, 25, 26, 27, 28, 32, 34, 39, 40, 41, 49, 50, 51, 57, 93, 98, 100, 101, 102, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 129, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 143, 146, 148, 150, 151, 152, 155, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 197, 198, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 221, 222, 223, 224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 241, 242, 243, 244, 246, 248, 249, 250, 251, 252, 254.
STK_FILE_NOS__: 41, 254.
stk_frame_equate: 147.
stop_err_enumerate: 204.
stop_lrk_enumerate: 204.
stop_rc_enumerate: 204.
stop_T_enumerate: 204.
strcpy: 79.
string: 5, 10, 22, 34, 41, 49, 50, 51, 54, 55, 56, 57, 86, 102, 127, 135, 143, 145, 146, 148, 155, 165, 198, 211, 212, 219, 221, 223, 224, 227, 236, 238, 240, 241, 244, 246, 248, 250, 252.
struct_end: 206.
struct_start: 206.
st1: 106.
sub_rule_def: 128.
sub_rule_def_end: 146.
sub_rule_def_imp: 146.
sub_rule_def_op_code: 146.
sub_rule_def_public: 128.
sub_rule_stk_parm: 147.
sub_rule_stk_parms_begin: 147.
sub_rule_stk_parms_end: 147.
subrule_def: 29, 142.
Subrule_def: 58, 83.
*subrule_def_*: 58, 165, 183, 258.
SUBRULE_DEFS_ITER_type: 120, 121, 128, 140, 146, 154.
subrule_directives: 58, 83, 128, 146, 154.
subrule_enum: 120.
*subrule_no_*: 50, 51, 58, 258.
subrule_no_of_rule: 120, 154, 183.
subrule_s_tree: 88, 146, 154.
Subrule_tree: 53, 55.
subrule_walk: 89, 90, 95.
subrules: 120, 121, 128, 140, 146, 154.
subrules_can: 88, 89, 90, 95.
subrules_phrase: 29.
subrules_to_rule_mapping: 140.
Success: 21, 23, 24, 25, 26, 27, 28, 32, 41, 42.
Suffix_enumeration_hdr: 211, 218, 235.
Suffix_Errors_hdr: 219, 226, 235.
Suffix_Errors_imp: 227, 244.
Suffix_fsc: 252.
Suffix_fsmheader: 135.
Suffix_fsmimp: 148.
Suffix_fsmsym: 155.

Suffix_fsmtbl: 198.
Suffix_LRK_hdr: 218, 235.
Suffix_t_alphabet: 212.
Suffix_T_hdr: 236, 243.
summary: 204.
svi: 12, 107, 170, 175, 176, 179, 186, 188, 189, 190, 191, 192, 195, 196.
svie: 12, 107, 170, 175, 176, 179, 186, 188, 189, 190, 191, 192, 195, 196.
Sym: 56, 169.
sym: 12, 13, 21, 23, 24, 25, 26, 27, 28, 32, 34, 35, 38, 40, 41, 42, 54, 56, 58, 61, 69, 82, 92, 110, 111, 112, 135, 147, 148, 155, 165, 166, 167, 198, 211, 212, 219, 227, 236, 244, 250, 252, 258.
Sym_t: 56.
Sym_to_xlate: 5.
sym1: 38, 40.
syntax_code: 67, 68, 80, 117, 123, 126, 127, 134, 141, 143, 145, 146, 150, 157, 214, 221, 222, 229, 231, 232, 238, 239.
`T_AB`: 37.
`T_AD`: 37.
t_alphabet: 210.
T_arbitrator_code: 37.
T_called_thread_eosubrule: 92, 249, 250.
T_constant_defs: 37.
T_constructor: 37.
T_cweb_comment: 39.
T_cweb_marker: 3.
t_def: 12, 14, 38, 85, 101, 102, 159, 161, 246, 248.
T_DEF_MAP_ITER_type: 70, 75, 81.
T_destructor: 37, 143, 221, 238.
t_entry: 66, 74, 79.
T_enum: 225, 242.
T_Enum: 12, 14, 35, 38, 39, 40, 42, 44, 53, 54, 56, 57, 79, 91, 92, 93, 96, 98, 110, 111, 112, 147, 154, 165, 166, 167, 169, 178, 181, 250.
T_enum_attributes: 24.
T_enum_phrase: 5, 8, 24, 30, 93, 98, 116, 119, 120, 121, 122, 123, 138, 140, 141, 151, 204, 205, 206, 207, 208, 211, 212, 215, 218, 225, 226, 230, 235, 242, 243, 246.
T_enum_phrase_th: 24.
T_enumeration: 24, 37.
T_eocode: 37.
T_error_symbols: 25, 37.
T_error_symbols_phrase: 5, 8, 25, 30, 75, 116, 138, 202, 210, 214, 215, 216, 217, 218, 219, 224, 225, 226, 227, 235.
T_failed: 37.
T_file_inclusion: 5, 41.
T_file_name: 37.
T_fsm: 21, 37.
T_fsm_class: 37.
T_fsm_class_phrase: 61, 117, 120, 121, 122, 123, 130, 132, 133, 134, 140, 141, 142, 150, 151, 152, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187.
T_fsm_comments: 37.
T_fsm_date: 37.
T_fsm_debug: 37.
T_fsm_filename: 37.
T_fsm_id: 37.
T_fsm_namespace: 37.
T_fsm_phrase: 5, 8, 21, 30, 62, 116, 117, 120, 121, 122, 123, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 148, 150, 151, 152, 155, 157, 158, 163, 170, 175, 179, 180, 184, 185, 186, 187, 198, 246, 251, 252.
T_fsm_version: 37.
T_grammar_phrase: 35.
t_in_stbl: 38, 85, 96.
T_in_stbl: 12, 14, 38, 85, 96, 97, 101, 102, 105, 159, 161, 246, 248.
T_IN_STBL_SET_ITER_type: 100.
T_IN_STBL_SET_type: 100.
T_IN_STBL_SORTED_SET_ITER_type: 101.
T_IN_STBL_SORTED_SET_type: 100.
T_lhs: 37.
T_lrk_sufx: 37.
*T_LR1_all_shift_operator_*: 44, 79.
T_lr1_constant_symbols: 27, 37.
*T_LR1_fset_transience_operator_*: 44, 53, 54, 56, 57, 79.
*T_LR1_invisible_shift_operator_*: 44, 79.
T_lr1_k_phrase: 5, 8, 27, 30, 81, 116, 138, 200, 210, 218, 226, 235, 243.
*T_LR1_parallel_operator_*: 53, 54, 56, 57, 79.
*T_LR1_questionable_shift_operator_*: 44, 79.
*T_LR1_reduce_operator_*: 44.
t_name: 12, 14, 38, 56, 66, 70, 74, 75, 79, 85, 101, 102, 223, 240.
T_name_space: 37.
`T_NULL`: 37.
T_op: 37.
T_parallel_control_monitor: 37.
T_parallel_la_boundary: 37, 62, 97.
T_parallel_monitor_phrase: 82, 127.
T_parallel_parser: 23, 37.
T_parallel_parser_phrase: 5, 8, 23, 30, 62, 118, 129, 151, 246.
T_parallel_thread_function: 37.
t_ph: 116, 138, 229, 230, 231, 232, 233, 234, 235, 236, 241, 242, 243, 244.

T_phrase: 64, 67, 68, 70.
*T_raw_ampersign_*: 44.
*T_raw_asteric_*: 44.
*T_raw_at_sign_*: 44.
*T_raw_back_slash_*: 44.
T_raw_characters: 26, 37.
*T_raw_close_brace_*: 44.
*T_raw_close_bracket_*: 44.
*T_raw_close_sq_bracket_*: 44.
*T_raw_colon_*: 44.
*T_raw_comma_*: 44.
*T_raw_dbl_quote_*: 44.
*T_raw_del_*: 44.
*T_raw_dollar_sign_*: 44.
*T_raw_eq_*: 44.
*T_raw_exclam_*: 44.
*T_raw_gt_than_*: 44.
*T_raw_left_quote_*: 44.
*T_raw_less_than_*: 44.
*T_raw_minus_*: 44.
*T_raw_no_sign_*: 44.
*T_raw_open_brace_*: 44.
*T_raw_open_bracket_*: 44.
*T_raw_open_sq_bracket_*: 44.
*T_raw_percent_*: 44.
*T_raw_period_*: 44.
*T_raw_plus_*: 44.
*T_raw_question_mark_*: 44.
*T_raw_right_quote_*: 44.
*T_raw_semi_colon_*: 44.
*T_raw_slash_*: 44.
*T_raw_tilde_*: 44.
*T_raw_under_score_*: 44.
*T_raw_up_arrow_*: 44.
*T_raw_vertical_line_*: 44.
T_rc_phrase: 5, 8, 26, 30, 116, 138, 201, 210.
t_ref_code: 231.
*T_refered_rule_*: 12, 14, 38, 53, 54, 56, 91, 96, 110, 111, 147.
*T_refered_T_*: 12, 14, 38, 53, 54, 56, 91, 96, 147.
*T_rule_def_*: 38, 57, 169.
T_rule_lhs_phrase: 82, 126, 143, 154.
T_rules: 32, 37.
T_rules_phrase: 5, 8, 29, 30, 120, 121, 124, 125, 130, 139, 140, 141, 142, 151, 152, 170, 246, 248, 249.
T_subrule_def: 58, 83, 120, 128, 146, 154.
T_subrules_phrase: 120, 121, 128, 140, 146, 154.
t_sufx_code: 232.
T_sw: 5, 34.
*t_sw_*: 34.
T_sym_class: 37.
T_sym_tbl_report_card: 37, 104, 105.
T_syntax_code: 67, 68, 80.
*T_T_called_thread_eosubrule_*: 14, 53, 54, 56, 92, 147, 154, 165, 166, 167, 250.
*T_T_cweb_comment_*: 39, 40.
*T_T_cweb_marker_*: 35.
*T_T_enumeration_*: 42.
*T_T_eosubrule_*: 14, 38, 53, 54, 56, 57, 91, 96, 178, 181.
*T_T_error_symbols_*: 42.
*T_T_fsm_*: 42.
*T_T_identifier_*: 38, 147.
*T_T_lr1_constant_symbols_*: 42.
`T_T_NULL_`: 38, 147.
*T_T_null_call_thread_eosubrule_*: 14, 53, 54, 56, 147, 154.
*T_T_parallel_parser_*: 42.
*T_T_raw_characters_*: 42.
*T_T_rules_*: 42.
*T_T_rules_phrase_*: 35.
*T_T_subrule_def_*: 38, 93, 98, 112.
*T_T_terminal_def_*: 38, 169.
*T_T_terminals_*: 42.
*T_T_2colon_*: 38, 147.
T_terminal_def: 12, 14, 70, 75, 81, 84, 85, 91, 102, 147, 200, 201, 202, 203, 209, 214, 221, 222, 223, 224, 229, 238, 239, 240, 241, 246, 248.
T_terminals: 28, 37.
T_terminals_phrase: 5, 8, 28, 30, 70, 116, 138, 203, 210, 229, 230, 231, 232, 233, 234, 235, 236, 241, 242, 243, 244.
T_terminals_refs: 37.
T_terminals_sufx: 37.
T_user_declaration: 37, 214, 229.
T_user_imp_sym: 37.
T_user_imp_tbl: 37.
T_user_implementation: 37, 222, 239.
T_user_prefix_declaration: 37.
T_user_suffix_declaration: 37.
table_entry: 37.
td: 12, 14, 84, 85, 91, 104, 105, 147, 169, 200, 201, 202, 203, 209, 214, 224, 229, 241.
Td: 221, 222, 223, 238, 239, 240.
tdef: 66, 69, 70, 74, 75, 79, 81, 246, 248.
term: 42.
term_name: 70, 75, 79, 81.
term_ph: 28.
terminals_attributes: 28, 29.
terminals_phrase: 28.
terminals_phrase_th: 28.
terminals_refs_code: 67, 231.
terminals_sufx_code: 68, 232.

Tes_in_list: 248.
tfe: 12.
th_extern: 118.
th_name: 54, 55, 56.
thd_proc: 129.
third_el_t: 92.
thread_calls_entry: 195.
thread_definition: 151.
thread_entry_extern_for_header: 118, 133.
thread_expr_string_template: 57.
thread_implementation: 151, 155.
thread_1st_elem: 250.
ti: 88.
tie: 88.
tintbl: 246, 248.
tit: 12, 14.
tok_can: 34, 36, 39, 41, 53, 89, 90, 95, 154, 254.
tok_can_ast_functor: 36, 39, 53, 93, 98, 110, 111, 112, 154.
TOK_CAN_TREE_type: 110, 111, 112.
tok_co_ords__: 35, 38, 40, 41.
token_container_type: 5, 21, 23, 24, 25, 26, 27, 28, 32, 41.
TOKEN_GAGGLE: 5, 21, 23, 24, 25, 26, 27, 28, 32, 34, 58, 61, 69, 82, 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252.
TOKEN_GAGGLE_ITER: 21, 23, 24, 25, 26, 27, 28, 32.
token_supplier: 21, 23, 24, 25, 26, 27, 28, 32.
total_enumerate: 109, 204, 246.
total_err_enumerate: 204.
total_lrk_enumerate: 204.
total_no_subrules: 140.
total_rc_enumerate: 204.
total_T_enumerate: 204.
*transitions_*: 12.
TRANSITIONS_ITER_type: 12.
transitive: 246.
transitive_list_of_threads_in_fs_imp: 249, 252.
transitive_thread_in_list: 249.
transitive_thread_list: 249.
tref: 67.
tref_code: 67.
true: 12, 30, 35, 38, 66, 74, 79, 83, 100, 102, 105, 106, 127, 143, 145, 146, 147, 168, 170, 171, 172, 258.
trunc: 135, 148, 155, 198, 211, 212, 219, 227, 236, 244, 252, 258.
tstbl: 96, 97.
tsuf: 68, 80.
tsuf_code: 68, 80.
tsym: 159.
T2: 5, 41.
uc: 111, 112.
ud: 123, 126.
ui: 141, 150, 157.
uimp: 143.
UINT: 89, 90, 95.
underline_symbol: 57.
underline_symbol_wepsilon: 57.
upd: 117, 134.
upd_code: 117, 134.
use_cnt: 112.
*use_cnt_*: 111, 112.
use_cnt_key: 111, 112.
use_cnt_type: 111, 112.
Use_for_rule: 110, 111.
used_list_of_threads_imp: 250, 252.
used_thread_in_list: 250.
used_thread_list: 250.
used_threads: 250.
user_declaration: 123.
user_imp_code: 143.
user_imp_sym: 150, 155.
user_imp_tbl: 157, 198.
user_prefix_declaration_for_header: 117, 133.
user_suffix_declaration_for_header: 134, 135.
user_t_imp_dtor: 238.
user_t_imp_implementation: 239, 241.
user_t_imp_shellimplementation: 240, 241.
user_t_include_files_for_header: 235, 236.
user_t_include_files_for_imp: 243, 244.
user_t_include_guard_for_header: 234, 236.
user_t_loop_thru_and_gen_defs_for_header: 229, 233.
user_t_loop_thru_and_gen_defs_for_imp: 241, 244.
user_t_namespace_for_header: 233, 234.
user_t_terminals_refs_for_header: 231, 233.
user_t_terminals_sufx_for_header: 232, 233.
user_t_use_enum_namespace_for_header: 230, 233.
user_t_use_enum_namespace_for_imp: 242, 244.
using_namespace: 225, 242.
using_ns_for_fsm_cpp: 138, 148, 155, 198.
using_ns_for_header: 119, 131.
using_T_namespace: 215, 230.
v: 161.
vect: 250.
vector: 88, 200, 201, 202, 203, 209, 214, 224, 229, 241.
*vectored_into_by_elem_*: 12, 164.
*vectored_into_by_elem_sym_*: 164.
version: 62, 141.
vi: 250.
Voc_ENO: 13.
*w_fig_no_*: 258.

*w_filename_*: 47, 258.
w_prefix_date: 47.
w_prefix_date_value: 47.
w_prefix_file: 47.
w_prefix_filename_value: 47.
walk: 110, 111, 112.
walk_functr: 110, 111, 112.
walk_the_plank_mate: 154.
Wfile: 5, 39.
write: 22, 46, 47, 51, 54, 60, 62, 65, 66, 67, 68, 73, 74, 78, 79, 80, 84, 86, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 143, 145, 146, 147, 151, 152, 154, 159, 160, 162, 163, 166, 167, 171, 172, 173, 179, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 223, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 240, 242, 243, 246, 248, 249, 250, 251.
write_out_op_code: 146.
WRT_CWEB_MARKER: 2, 5, 39, 62, 66, 67, 68, 70, 74, 75, 79, 80, 81.
x: 10, 22, 39, 46, 51, 54, 60, 62, 66, 67, 68, 74, 79, 80, 84, 86, 110, 111, 112, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 126, 127, 128, 129, 130, 131, 132, 133, 134, 137, 138, 139, 140, 141, 143, 145, 146, 147, 151, 152, 154, 159, 160, 163, 166, 167, 171, 172, 173, 179, 182, 183, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 214, 215, 216, 217, 218, 221, 223, 225, 226, 229, 230, 231, 232, 233, 234, 235, 238, 240, 242, 243, 246, 248, 249, 250, 251.
xa: 60.
xlate: 101.
XLATE_SYMBOLS_FOR_cweave: 2, 5, 22, 56, 60, 62, 66, 70, 74, 75, 79, 85, 101.
*xlated_names_*: 44, 84.
Xlated_str: 57.
Xlated_sym: 5, 56.
xref: 53, 54, 84, 85, 86.
xsym: 56.
xx: 128, 139, 146.
xxrdmap: 154.
xxx: 89, 90, 95, 154.
x0: 139.
y: 147.
yacco2: 2, 5, 8, 12, 14, 21, 22, 23, 24, 25, 26, 27, 28, 32, 34, 35, 37, 38, 39, 40, 41, 42, 53, 54, 89, 90, 95, 110, 111, 154, 254.
yacco2_characters: 2.
yacco2_err_symbols: 2.
Yacco2_holding_file: 34.
yacco2_k_symbols: 2.
Yacco2_lcl_options: 256.
YACCO2_PARSE_CMD_LINE: 2, 5, 34, 256.
yacco2_stbl: 2, 37, 104, 105.
yacco2_T_enumeration: 2.
yacco2_terminals: 2.
YES: 21, 23, 24, 25, 26, 27, 28, 32, 34, 41, 57, 91, 96, 111, 112.

⟨ Files for header 5 ⟩ Used in section 4.
⟨ Include Header file 6 ⟩ Used in section 7.
⟨ accrue source for emit 8, 10, 11, 21, 22, 23, 24, 25, 26, 27, 28, 32, 34, 35, 37, 38, 39, 40, 41, 42, 44, 48, 49, 50, 51, 55, 56, 57, 58, 62, 70, 75, 81, 82, 83, 85, 86, 93, 98, 102, 104, 105, 106, 107, 109, 110, 111, 112, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 129, 130, 131, 132, 133, 134, 135, 137, 138, 139, 140, 141, 142, 148, 150, 151, 152, 155, 157, 158, 163, 169, 170, 175, 179, 180, 184, 185, 186, 187, 197, 198, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 214, 215, 216, 217, 218, 219, 221, 222, 223, 224, 225, 226, 227, 229, 230, 231, 232, 233, 234, 235, 236, 238, 239, 240, 241, 242, 243, 244, 246, 248, 249, 250, 251, 252 ⟩ Used in section 7.
⟨ add Defined rule's subrules to element space 95 ⟩ Used in section 98.
⟨ add Start rule's subrules to element space 89 ⟩ Used in section 93.
⟨ add referenced rule to element space 90 ⟩ Used in sections 91 and 96.
⟨ add subrule's 1st element to element space 88 ⟩ Used in sections 89, 90, and 95.
⟨ all shift entry 189 ⟩ Used in section 187.
⟨ build la set's bit table 161 ⟩ Used in section 160.
⟨ bypass meta T 177 ⟩
⟨ bypass reduces 178 ⟩ Used in sections 170, 175, and 176.
⟨ deal with T's classifications 84 ⟩ Used in section 85.
⟨ deal with arbitrator code 144 ⟩ Used in section 143.
⟨ deal with element type 96 ⟩ Used in section 98.
⟨ deal with thread call expression 92 ⟩ Used in section 91.
⟨ deal with thread called element type 91 ⟩ Used in section 93.
⟨ deal with threads in state 164 ⟩ Used in section 163.
⟨ determine if there is a rule's arbitrator to call 168 ⟩ Used in section 166.
⟨ determine number of reduces 181 ⟩ Used in sections 180 and 194.
⟨ determine number of shifts 176 ⟩ Used in sections 170, 175, and 193.
⟨ determine number of threads to call 165 ⟩ Used in section 164.
⟨ dump se element 13 ⟩ Used in section 12.
⟨ dump sr element 14 ⟩ Used in section 13.
⟨ emit la set 160 ⟩ Used in section 158.
⟨ enumerate Rule alphabet 31 ⟩ Used in section 32.
⟨ enumerate Terminal alphabet 30 ⟩ Used in section 32.
⟨ err rule-it 73 ⟩ Used in section 74.
⟨ establish tree container with appropriate element filters 53 ⟩ Used in section 55.
⟨ gen fsm rules declarations 126 ⟩ Used in section 125.
⟨ gen fsm subrules declarations 128 ⟩ Used in section 126.
⟨ gen rule's arbitrator procedure declaration 127 ⟩ Used in section 126.
⟨ gen rules's subrules case stmts 153 ⟩ Used in section 152.
⟨ gen stack frame def and equate it to the parse stack 147 ⟩ Used in section 146.
⟨ gen subrule case stmt 154 ⟩ Used in section 153.
⟨ get cast referenced T 17 ⟩ Used in sections 12 and 14.
⟨ get cast referenced called thread eosubrule 20 ⟩ Used in section 14.
⟨ get cast referenced eosubrule 18 ⟩ Used in section 14.
⟨ get cast referenced null called thread eosubrule 19 ⟩ Used in section 14.
⟨ get cast referenced rule 16 ⟩ Used in sections 12 and 14.
⟨ get subrule's referenced rule in follow string 15 ⟩
⟨ get those T sdcode class jelly beans ready for consumption 69 ⟩ Used in sections 70, 75, and 81.
⟨ handle the *cweb* comments that prefix phrases 36 ⟩ Used in section 35.
⟨ initialize the *Cmpost_output* class variables 258 ⟩ Cited in section 257.
⟨ invisible shift entry 190 ⟩ Used in section 187.
⟨ is Start rule ϵ if yes add LA expression to first set 97 ⟩ Used in section 98.
⟨ is there arbitration code? — yes output 145 ⟩ Used in section 144.
⟨ list literal entries of la set 159 ⟩ Used in section 158.

⟨ loop thru the rules 143 ⟩ Used in section 142.
⟨ loop thru the subrules 146 ⟩ Used in section 143.
⟨ lrk rule-it 78 ⟩ Used in section 79.
⟨ o2_externs.cpp 7 ⟩
⟨ o2_externs.h 4 ⟩
⟨ output called procedure table def / imp 167 ⟩ Used in section 196.
⟨ output called threads table def / imp 166 ⟩ Used in section 164.
⟨ output end of shift table entries 174 ⟩ Used in sections 170 and 175.
⟨ output lrk suffix code if present 80 ⟩ Used in section 81.
⟨ output prefix code if present 67 ⟩ Used in section 70.
⟨ output reduce entries 183 ⟩ Used in section 182.
⟨ output reduced table def / imp 182 ⟩ Used in section 180.
⟨ output start of shift definition 173 ⟩ Used in sections 170 and 175.
⟨ output suffix code if present 68 ⟩ Used in section 70.
⟨ output the la set's compressed bits 162 ⟩ Used in section 160.
⟨ output T macros and files 64 ⟩ Used in section 70.
⟨ output T's entry 66 ⟩ Used in section 70.
⟨ output *err* macros and files 72 ⟩ Used in section 75.
⟨ output *err*'s entry 74 ⟩ Used in section 75.
⟨ output *fsm*'s class section 60 ⟩ Used in section 62.
⟨ output *lrk* macros and files 77 ⟩ Used in section 81.
⟨ output *lrk*'s entry 79 ⟩ Used in section 81.
⟨ parallel shift entry 188 ⟩ Used in section 187.
⟨ print 1st or 2nd shift entry 171 ⟩ Used in sections 170 and 175.
⟨ print dump state 12 ⟩ Used in section 11.
⟨ print out the sorted first set elements 101 ⟩ Used in section 102.
⟨ print shift accept entry 172 ⟩ Used in section 170.
⟨ procedure call function entry 196 ⟩ Used in section 187.
⟨ procedure call shift entry 191 ⟩ Used in section 187.
⟨ questionable shift entry 192 ⟩ Used in section 187.
⟨ ready those *fsm* sdcode class jelly beans for consumption 61 ⟩ Used in section 62.
⟨ reduce entry 194 ⟩ Used in section 187.
⟨ rule-it 65 ⟩ Used in section 66.
⟨ set up first set sort 100 ⟩ Used in section 102.
⟨ setup *cweave* macros and odds and ends 47 ⟩ Used in section 48.
⟨ setup *mpost* macros and variable arrays 46 ⟩ Used in section 48.
⟨ shift entry 193 ⟩ Used in section 187.
⟨ thread call entry 195 ⟩ Used in section 187.
⟨ walk the container and produce the elements' cross references 54 ⟩ Used in section 55.

O2EXTERNS

	Section	Page
License	1	1
Summary of O_2 External parse routines	2	2
Global definitions, External parse routines for Yacco2	3	3
Files for header	5	4
Local Yacco2 routines	9	7
Print_dump_state	11	7
process_fsm_phrase	21	15
Output macros and banner	22	16
process_parallel_parser_phrase	23	17
process_T_enum_phrase	24	19
process_error_symbols_phrase	25	20
process_rc_phrase	26	21
process_lr1_k_phrase	27	22
process_terminals_phrase	28	23
process_rules_phrase	29	24
Enumerate Terminal alphabet	30	25
Enumerate Rule alphabet	31	26
Driver of *process_rules_phrase*	32	27
External routines	33	28
Yacco2 Parse command line: `YACCO2_PARSE_CMD_LINE`	34	28
`BUILD_GRAMMAR_TREE` while parsing grammar	35	29
`LOAD_YACCO2_KEYWORDS_INTO_STBL`	37	31
Beauty and the tree: `PRINT_RULES_TREE_STRUCTURE`	38	33
`WRT_CWEB_MARKER`	39	35
?Beauty and the tree: `PRINT_GRAMMAR_TREE`	40	35
Process Nested include files: `PROCESS_INCLUDE_FILE`	41	36
Process syntax code: `PROCESS_KEYWORD_FOR_SYNTAX_CODE`	42	38

Mpost and Cweb Routines 43 40
MPOST_CWEB_LOAD_XLATE_CHRS 44 40
MPOST_CWEB_EMIT_PREFIX_CODE 45 41
Setup *mpost* macros and variable arrays 46 41
Setup *cweave* macros and odds and ends 47 41
Driver of MPOST_CWEB_EMIT_PREFIX_CODE 48 42
MPOST_CWEB_gen_dimension_name 49 42
MPOST_CWEB_calc_mp_obj_name 50 42
MPOST_CWEB_wrt_mp_rhs_elem 51 43
MPOST_CWEB_gen_sr_elem_xrefs 52 43
Establish tree container with appropriate element filters 53 44
Walk the container and produce the elements' cross references 54 45
Driver of *MPOST_CWEB_gen_sr_elem_xrefs* 55 46
MPOST_CWEB_xlated_symbol 56 47
MPOST_CWEB_crt_rhs_sym_str - follow set contributing symbols 57 49
MPOST_CWEB_woutput_sr_sdcode 58 50
MPOST_CWEB_wrt_fsm 59 50
Output *fsm*'s class section 60 51
Ready those *fsm* syntax directed code jelly beans for consumption 61 51
Driver of *MPOST_CWEB_wrt_fs* 62 52
MPOST_CWEB_wrt_T 63 52
Output *T* macros and files 64 53
hrule to end T entry 65 53
Output *T*'s entry 66 53
Output prefix code if present 67 54
Output suffix code if present 68 54
Get those *T* syntax directed code jelly beans ready for consumption 69 54
Driver of *MPOST_CWEB_wrt_T* 70 55
MPOST_CWEB_wrt_Err 71 55
Output *err* macros and files 72 55
Err hrule to end T entry 73 55
Output *err*'s entry 74 56
Driver of *MPOST_CWEB_wrt_Err* 75 57
MPOST_CWEB_wrt_lrk 76 57
Output *lrk* macros and files 77 57
Lrk hrule to end T entry 78 57
Output *lrk*'s entry 79 58
Output lrk suffix code if present 80 59
Driver of *MPOST_CWEB_wrt_Lrk* 81 60
MPOST_CWEB_wrt_rule_s_lhs_sdc 82 61
MPOST_CWEB_should_subrule_be_printed 83 61
Deal with T's classifications 84 62
MPOST_CWEB_xref_refered_T 85 63
MPOST_CWEB_xref_refered_rule 86 63
Grammar's "Called threads" First Set Calculation 87 64
Add subrule's 1st element to element space 88 65
Add Start rule's subrules to element space 89 65
Add referenced rule to element space 90 65
Assess element type 91 66
Deal with "thread call" expression 92 66
GEN_CALLED_THREADS_FS_OF_RULE 93 67
First set calculations 94 68

Add Defined rule's subrules to element space 95 68
Assess element type 96 68
Check Start rule for ϵ condition 97 69
GEN_FS_OF_RULE 98 69

Print out Rule's First Set 99 70
Set up first set sort 100 70
Print out the sorted first set elements 101 70
Driver of PRT_RULE_S_FIRST_SET 102 71

Commonize LA sets 103 72
find_common_la_set_idx 104 72
are_2_la_sets_equal 105 73
find_common_la_set_idx 106 74
COMMONIZE_LA_SETS 107 75

Calculate recycling rule's use count 108 76
calc_cyclic_key 109 76
determine_rhs_indirect_use_cnt 110 77
determine_rhs_max_use_cnt 111 78
MAX_USE_CNT_RxR: called by *rules_use_cnt.lex* grammar 112 79

Emit Grammar by External procedures 113 80
Template of Grammar's header file declaration: OP_GRAMMAR_HEADER 114 81
intro_comment — Intro comments for emitted files 115 81
grammar_header_includes 116 82
user_prefix_declaration_for_header 117 83
thread_entry_extern_for_header 118 83
using_ns_for_header 119 84
fsm_enum_rules_subrules_for_header 120 85
fsm_map_subrules_to_rules_for_header — Optimization 121 86
fsm_comments_about_la_sets_and_states 122 86
fsm_class_declaration_for_header 123 87
forward_refs_of_rules_declarations_for_header 124 88
rules_class_defs_for_header 125 88
Gen rule's arbitrator procedure declaration 127 90
Gen fsm subrules declarations 128 90
possible_thread_procedure_declaration 129 91
rules_reuse_table_declaration — Optimization 130 91
namespace_for_header 131 92
state_1_extern 132 92
grammar_include_guard_for_header 133 93
user_suffix_declaration_for_header 134 93
OP_GRAMMAR_HEADER implementation 135 94
Template of Grammar's fsm file implementation: OP_GRAMMAR_CPP 136 95
fsm_cpp_includes 137 95
using_ns_for_fsm_cpp 138 96
rules_reuse_table_implementation — Optimization 139 97
fsm_map_subrules_to_rules_table_imp — Optimization 140 98
fsm_class_implementation 141 100
rules_subrules_implementations 142 102
Go thru rules 143 103
Deal with arbitrator code 144 105
Emit the syntax directed code(sdc) 146 106
Gen stack frame definition and equate it to the parse stack 147 107

OP_GRAMMAR_CPP implementation ... 148 109
Template of fsm sym file implementation: OP_GRAMMAR_SYM ... 149 110
user_imp_sym ... 150 110
thread_implementation ... 151 111
fsm_reduce_rhs_of_rule_implementation ... 152 112
Gen rules's subrules case statements ... 153 113
Gen subrule case statement ... 154 114
OP_GRAMMAR_SYM implementation ... 155 116
Template of fsm tbl file implementation: OP_GRAMMAR_TBL ... 156 117
user_imp_tbl ... 157 117
output_la_sets ... 158 117
List literal entries of la set ... 159 118
Emit la set ... 160 118
Build la set's bit table ... 161 119
Output the la set's compressed bits ... 162 119
externs_and_thread_tbl_defs ... 163 120
Deal with threads in state ... 164 120
Determine number of threads to call ... 165 121
Output called threads table def / imp ... 166 122
Output called procedure table def / imp ... 167 123
determine if there is a rule's arbitrator to call ... 168 123
determine_shift_element_name ... 169 124
Output 1st state's shift table ... 170 125
Print 1st or 2nd shift entry ... 171 126
Print shift accept entry ... 172 126
Output start of shift definition ... 173 127
Output end of shift table entries ... 174 127
Output all others state's shift table ... 175 128
Determine number of shifts ... 176 128
Bypass meta T ... 177 129
Bypass reduces ... 178 130
Output meta shifts separately ... 179 131
Output state's reduced table ... 180 132
Determine number of reduces ... 181 133
Output reduced table def / imp ... 182 134
Output reduce entries ... 183 135
Output state's called threads table ... 184 135
Output state's called procedure table ... 185 136
Output meta shifts separately ... 186 137
Output lr state ... 187 139
Parallel shift entry ... 188 139
All shift entry ... 189 140
Invisible shift entry ... 190 140
Procedure call shift entry ... 191 141
Questionable shift entry ... 192 141
Shift entry ... 193 142
Reduce entry ... 194 142
Thread call entry ... 195 143
Procedure call function entry ... 196 143
emit_each_lr_state_s_tables ... 197 144
OP_GRAMMAR_TBL implementation ... 198 145
Template of enumeration header: OP_ENUMERATION_HEADER ... 199 146

Enumerate the LR constants classification: *enumerate_lrk* . . . 200 146
Enumerate the Raw Characters classification: *enumerate_rc* . . . 201 147
Enumerate the Errors classification: *enumerate_errors* . . . 202 147
Enumerate the T classification: *enumerate_T* . . . 203 148
Terminals Enumeration summary: *enumeration_summary_for_struct* . . . 204 148
enumeration_define_list . . . 205 149
enumeration_define_structure . . . 206 149
enumeration_namespace_for_header . . . 207 150
enumeration_include_guard_for_header . . . 208 150
Gen Tes's literal names for specific Tes . . . 209 151
Gen Tes's literal names for O_2^{linker} . . . 210 151
`OP_ENUMERATION_HEADER` implementation . . . 211 152
OP_T_Alphabet implementation . . . 212 153
Template of Errors vocabulary header: `OP_ERRORS_HEADER` . . . 213 154
errors_loop_thru_and_gen_defs_for_header . . . 214 155
errors_use_enum_namespace_for_header . . . 215 156
errors_namespace_for_header . . . 216 156
errors_include_guard_for_header . . . 217 157
errors_include_files_for_header . . . 218 157
`OP_ERRORS_HEADER` implementation . . . 219 158
Template of User Errors vocabulary header: `OP_ERRORS_HEADER` . . . 220 159
errors_imp_dtor . . . 221 159
errors_imp_implementation . . . 222 159
errors_imp_shellimplementation . . . 223 160
errors_loop_thru_and_gen_defs_for_imp . . . 224 161
errors_use_enum_namespace_for_imp . . . 225 162
errors_include_files_for_imp . . . 226 162
`OP_ERRORS_CPP` implementation . . . 227 163
Template of USER T vocabulary header: `OP_USER_T_HEADER` . . . 228 164
user_t_loop_thru_and_gen_defs_for_header . . . 229 165
user_t_use_enum_namespace_for_header . . . 230 166
user_t_terminals_refs_for_header . . . 231 166
user_t_terminals_sufx_for_header . . . 232 166
user_t_namespace_for_header . . . 233 167
user_t_include_guard_for_header . . . 234 167
user_t_include_files_for_header . . . 235 168
`OP_USER_T_HEADER` implementation . . . 236 168
Template of USER T vocabulary header: `OP_USER_T_HEADER` . . . 237 169
user_t_imp_dtor . . . 238 169
user_t_imp_implementation . . . 239 169
user_t_imp_shellimplementation . . . 240 170
user_t_loop_thru_and_gen_defs_for_imp . . . 241 171
user_t_use_enum_namespace_for_imp . . . 242 172
user_t_include_files_for_imp . . . 243 172
`OP_USER_T_CPP` implementation . . . 244 173
Template of FSC File: `OP_FSC_FILE` . . . 245 174
fsc_prelude_imp . . . 246 175
list_of_native_Tes_in_fs_imp . . . 247 176
Driver of *list_of_native_Tes_in_fs_imp* . . . 248 177
transitive_list_of_threads_in_fs_imp . . . 249 178
used_list_of_threads_imp . . . 250 179
grammar_s_comments_for_linker_doc_imp . . . 251 180

`OP_FSC_FILE` implementation 252 180
Bric-a-brac 253 181
Index 265 184