O

1. Global definitions and files.

Basic preamble include files used by all others.

```
⟨globals.h 1⟩ ≡    /* file: globals.h */    /* prelude files using yacco2: for o2, o2linker utilities */
#ifndef globals_h__
#define globals_h__ 1
#include <stdarg.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "yacco2.h"
#include "yacco2_T_enumeration.h"
#include "yacco2_err_symbols.h"
#include "yacco2_characters.h"
#include "yacco2_k_symbols.h"
#include "yacco2_terminals.h"
  using namespace std;
  using namespace NS_yacco2_T_enum;
  using namespace NS_yacco2_k_symbols;
  using namespace NS_yacco2_terminals;
  using namespace yacco2;
#endif
```

2. "o2_types" header file of common set of definitions and structures.

"o2_types.h" file is a common set of definitions and structures used by "o2externs.w" external routines. Contains definitions and type-defs.

```
⟨o2_types.h 2⟩ ≡
#ifndef o2_types_
#define o2_types_ 1
  ⟨defines 3⟩;
  ⟨Type defs 4⟩;
  ⟨Structure defs 82⟩;
#endif
```

3. Definitions for O2 and my external routines — "o2externs.w".
As i'm writing directly out to a file, the use of the ctangle macro directive displays its displeasure so i'm using the direct c code route.

⟨ defines 3 ⟩ ≡

```
#define NO_BITS_PER_SET_PARTITION 8
#define CODE_PRESENCE_IN_ARBITRATOR_CODE "pp_accept_queue__"
#define ACCEPT_FILTER true
#define BYPASS_FILTER false
#define Success true
#define Failure false
#define Nested_file_cnt_limit 15
#define O2_library_file "yacco2.h"
#define Yacco2_holding_file "yacco2cmd.tmp"
#define Linker_holding_file "linkercmd.tmp"
#define Max_cweb_item_size 10 * 1024
#define Max_buf_size 2 * 1024
#define Max_no_subrules 8 * 1024
#define EPSILON_YES 0
#define EPSILON_NO 1
#define EPSILON_DONT_KNOW 2
#define EPSILON_MAYBE 3
#define SDC_user_imp_sym "#user-imp-sym"
#define SDC_user_imp_tbl "#user-imp-tbl"
#define SDC_user_prefix_declaration "#user-prefix-declaration"
#define SDC_user_suffix_declaration "#user-suffix-declaration"
#define SDC_user_declaration "#user-declaration"
#define SDC_constructor "#constructor"
#define SDC_destructor "#destructor"
#define SDC_op "#op"
#define SDC_failed "#failed"
#define SDC_user_implementation "#user-implementation"
#define SDC_user_imp_implementation "#user-imp-sym"
#define SDC_user_tbl_implementation "#user-tbl-tbl"
#define SDC_arbitrator_code "#arbitrator-code"
#define PP_thread_name "#parallel-thread-function"
#define Suffix_fsmheader ".h"
#define Suffix_fsmimp ".cpp"
#define Suffix_fsmsym "sym.cpp"
#define Suffix_fsmtbl "tbl.cpp"
#define Suffix_enumeration_hdr ".h"
#define Suffix_t_alphabet ".fsc"
#define Suffix_Errors_hdr ".h"
#define Suffix_Errors_imp ".cpp"
#define Suffix_T_hdr ".h"
#define Suffix_T_imp ".cpp"
#define Suffix_RC_hdr ".h"
#define Suffix_RC_imp ".cpp"
#define Suffix_LRK_hdr ".h"
#define Suffix_LRK_imp ".cpp"
#define Suffix_fsc ".fsc"
#define LR1_COMPATIBLE true
#define NOT_LR1_COMPATIBLE false
```

```
#define MERGED  true
#define NOT_MERGED  false
#define ABORT_GENING_STATES  2
#define START_STATE_ENUMERATE −1
#define LR1_QUESTIONABLE_SHIFT_OPERATOR  0
#define LR1_QUESTIONABLE_SHIFT_OPERATOR_LITERAL  "|?|"
#define LR1_EOG  1
#define LR1_EOLR  2
#define LR1_EOLR_LITERAL  "eolr"
#define LR1_PARALLEL_OPERATOR  3
#define LR1_REDUCE_OPERATOR  4
#define LR1_REDUCE_OPERATOR_LITERAL  "|r|"
#define LR1_INVISIBLE_SHIFT_OPERATOR  5
#define LR1_ALL_SHIFT_OPERATOR  6
#define LR1_FSET_TRANSIENCE_OPERATOR  7
#define LR1_FSET_TRANSIENCE_OPERATOR_LITERAL  "LR1_fset_transience_operator"
#define LR1_PROCEDURE_CALL_OPERATOR  7
#define LR1_PROCEDURE_CALL_OPERATOR_LITERAL  "|p|"
#define END_OF_LR1_DEFS  7
#define SMALL_BUFFER_4K1024 ∗ 4
#define BIG_BUFFER_32K1024 ∗ 32
#define THREAD_CALL  0
#define PROCEDURE_CALL  1
#define MICROSOFT_THREAD_LIBRARY  1
#define PTHREAD_LIBRARY  0
```

This code is used in section 2.

4. Typedef definitions.

⟨ Type defs 4 ⟩ ≡

```
  typedef int Voc_ENO;
  typedef int RULE_ENO;
  struct state;
  struct state_element;
  struct follow_element;
```

See also sections 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, and 80.

This code is used in section 2.

5. ⟨ Type defs 4 ⟩ +≡

```
  typedef std::map < std::string, yacco2::CAbs_lr1_sym ∗> SDC_MAP_type;
```

6. ⟨ Type defs 4 ⟩ +≡

```
  typedef SDC_MAP_type ::iterator SDC_MAP_ITER_type;
```

7. ⟨ Type defs 4 ⟩ +≡

```
  typedef std::vector < NS_yacco2_terminals::T_in_stbl ∗> STBL_T_ITEMS_type;
```

8. ⟨ Type defs 4 ⟩ +≡

```
  typedef std::set < NS_yacco2_terminals::T_in_stbl ∗> T_IN_STBL_SET_type;
```

9. ⟨ Type defs 4 ⟩ +≡
typedef T_IN_STBL_SET_type ::iterator T_IN_STBL_SET_ITER_type;

10. ⟨ Type defs 4 ⟩ +≡
typedef std::vector ⟨ **NS_yacco2_terminals**::*T_in_stbl* ∗> *T_IN_STBL_SORTED_SET_type*;

11. ⟨ Type defs 4 ⟩ +≡
typedef T_IN_STBL_SORTED_SET_type ::iterator T_IN_STBL_SORTED_SET_ITER_type;

12. ⟨ Type defs 4 ⟩ +≡
typedef std::set ⟨ **NS_yacco2_terminals**::*rule_in_stbl* ∗> *RULE_IN_STBL_SET_type*;

13. ⟨ Type defs 4 ⟩ +≡
typedef RULE_IN_STBL_SET_type ::iterator RULE_IN_STBL_SET_ITER_type;

14. ⟨ Type defs 4 ⟩ +≡
typedef std::vector ⟨ **yacco2**::*CAbs_lr1_sym* ∗> *O2_PHRASE_TBL_type*;

15. ⟨ Type defs 4 ⟩ +≡
typedef O2_PHRASE_TBL_type ::iterator O2_PHRASE_TBL_ITER_type;

16. ⟨ Type defs 4 ⟩ +≡
typedef std::set ⟨ **NS_yacco2_terminals**::*rule_in_stbl* ∗> *RULES_IN_FS_SET_type*;

17. ⟨ Type defs 4 ⟩ +≡
typedef RULES_IN_FS_SET_type ::iterator RULES_IN_FS_SET_ITER_type;

18. ⟨ Type defs 4 ⟩ +≡
typedef std::vector ⟨ **NS_yacco2_terminals**::*rule_def* ∗> *RULE_DEFS_TBL_type*;

19. ⟨ Type defs 4 ⟩ +≡
typedef RULE_DEFS_TBL_type ::iterator RULE_DEFS_TBL_ITER_type;

20. ⟨ Type defs 4 ⟩ +≡
typedef std::vector ⟨ **NS_yacco2_terminals**::*T_subrule_def* ∗> *SUBRULE_DEFS_type*;

21. ⟨ Type defs 4 ⟩ +≡
typedef SUBRULE_DEFS_type ::iterator SUBRULE_DEFS_ITER_type;

22. ⟨ Type defs 4 ⟩ +≡
typedef std::map ⟨**std**::*string*, **NS_yacco2_terminals**::*T_terminal_def* ∗> *T_DEF_MAP_type*;

23. ⟨ Type defs 4 ⟩ +≡
typedef T_DEF_MAP_type ::iterator T_DEF_MAP_ITER_type;

24. ⟨ Type defs 4 ⟩ +≡
typedef std::set⟨**int**⟩ **INT_SET_type**;

25. ⟨ Type defs 4 ⟩ +≡
typedef INT_SET_type::iterator INT_SET_ITER_type;

26. ⟨ Type defs 4 ⟩ +≡
typedef std::map⟨**int**, **std**::*string*⟩ **INT_STR_MAP_type**;

27. ⟨ Type defs 4 ⟩ +≡
typedef INT_STR_MAP_type :: **iterator INT_STR_MAP_ITER_type**;

28. ⟨ Type defs 4 ⟩ +≡
typedef std :: **set** ⟨ **std** :: *string* ∗> *STR_SET_type*;

29. ⟨ Type defs 4 ⟩ +≡
typedef STR_SET_type :: **iterator STR_SET_ITER_type**;

30. ⟨ Type defs 4 ⟩ +≡
typedef std :: **map** ⟨ **NS_yacco2_terminals** :: *T_in_stbl* ∗ , **STR_SET_type** >
T_IN_STBL_SET_STR_MAP_type;

31. ⟨ Type defs 4 ⟩ +≡
typedef T_IN_STBL_SET_STR_MAP_type :: **iterator**
T_IN_STBL_SET_STR_MAP_ITER_type;

32. ⟨ Type defs 4 ⟩ +≡
typedef std :: **list** < **state** ∗ > *STATES_type*;

33. ⟨ Type defs 4 ⟩ +≡
typedef STATES_type :: **iterator STATES_ITER_type**;

34. ⟨ Type defs 4 ⟩ +≡
typedef int Voc_ENO;

35. ⟨ Type defs 4 ⟩ +≡
typedef int RULE_ENO;

36. ⟨ Type defs 4 ⟩ +≡
typedef int T_ENO;

37.
⟨ Type defs 4 ⟩ +≡
typedef std :: **set**⟨**RULE_ENO**⟩ **RULE_NOS_SET_type**;

38.
⟨ Type defs 4 ⟩ +≡
typedef RULE_NOS_SET_type :: **iterator RULE_NOS_SET_ITER_type**;

39.
⟨ Type defs 4 ⟩ +≡
typedef std :: **set**⟨**state** ∗⟩ **STATES_SET_type**;

40.
⟨ Type defs 4 ⟩ +≡
typedef STATES_SET_type :: **iterator STATES_SET_ITER_type**;

41. Map of lr1 states used for potential state merge.
⟨ Type defs 4 ⟩ +≡
typedef std :: **map**⟨**Voc_ENO**, **STATES_type**⟩ **LR1_STATES_type**;

42.
⟨ Type defs 4 ⟩ +≡
  typedef LR1_STATES_type :: **iterator LR1_STATES_ITER_type**;

43.
⟨ Type defs 4 ⟩ +≡
  typedef std :: **list**⟨**state_element** ∗⟩ **S_VECTOR_ELEMS_type**;

44.
⟨ Type defs 4 ⟩ +≡
  typedef S_VECTOR_ELEMS_type :: **iterator S_VECTOR_ELEMS_ITER_type**;

45. This supplies all GPS positions within each subrule string of symbols to generate the lr states for the state and it also supplies all the follow set GPS contributors for each referenced rules. All strings to the right of these referenced rules become the follow set strings of symbols to calculate the follow set.
⟨ Type defs 4 ⟩ +≡
  typedef std :: **map**⟨**Voc_ENO**, **S_VECTOR_ELEMS_type**⟩ **S_VECTORS_type**;

46. ⟨ Type defs 4 ⟩ +≡
  typedef S_VECTORS_type :: **iterator S_VECTORS_ITER_type**;

47. ⟨ Type defs 4 ⟩ +≡
  typedef std :: **set** ⟨ *T_in_stbl* ∗> *FOLLOW_SETS_type*;

48. ⟨ Type defs 4 ⟩ +≡
  typedef FOLLOW_SETS_type :: **iterator FOLLOW_SETS_ITER_type**;

49. ⟨ Type defs 4 ⟩ +≡
  typedef std :: **set** ⟨ *T_in_stbl* ∗> *LA_SET_type*;

50. ⟨ Type defs 4 ⟩ +≡
  typedef LA_SET_type :: **iterator LA_SET_ITER_type**;

51. ⟨ Type defs 4 ⟩ +≡
  typedef std :: **set** ⟨ *T_in_stbl* ∗> *SHIFT_SET_type*;

52. ⟨ Type defs 4 ⟩ +≡
  typedef SHIFT_SET_type :: **iterator SHIFT_SET_ITER_type**;

53. ⟨ Type defs 4 ⟩ +≡
  typedef std :: **list**⟨**follow_element** ∗⟩ **TRANSITIONS_type**;

54. ⟨ Type defs 4 ⟩ +≡
  typedef TRANSITIONS_type :: **iterator TRANSITIONS_ITER_type**;

55. ⟨ Type defs 4 ⟩ +≡
  typedef std :: **list**⟨**state** ∗⟩ **MERGES_type**;

56. ⟨ Type defs 4 ⟩ +≡
  typedef MERGES_type :: **iterator MERGES_ITER_type**;

57. List of subrules elements that require follow set calculation. This is the contributors list that allows me to debug my code. A contributor is the grammar node of the subrule element.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **list** ⟨ AST ∗> *SR_ELEMENTS_type*;

58. State's follow sets of referenced rules making up its "closure-only" core. The rule definition enumerate value is the map's key.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **map**⟨**RULE_ENO**, **follow_element** ∗⟩ **S_FOLLOW_SETS_type**;

59.

⟨ Type defs 4 ⟩ +≡
  typedef S_FOLLOW_SETS_type :: **iterator S_FOLLOW_SETS_ITER_type**;

60. Closure state's conflict list from its "closure-part" gened state network. These are the lr1 states that have a reduce configuration (subrule's string of symbols consumed or epsilon) with other reduce or shift subrule configurations that are checked for lr1 compatibility. The list allows one to pretest the merge potential of a newly being gened state into an already gened lr1 state network.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **list**⟨**state** ∗⟩ **S_CONFLICT_STATES_type**;

61.

⟨ Type defs 4 ⟩ +≡
  typedef S_CONFLICT_STATES_type :: **iterator S_CONFLICT_STATES_ITER_type**;

62.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **set** ⟨ *rule_in_stbl* ∗> *CLOSURE_RULES_type*;

63.

⟨ Type defs 4 ⟩ +≡
  typedef CLOSURE_RULES_type :: **iterator CLOSURE_RULES_ITER_type**;

64.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **set**⟨**T_ENO**⟩ **FOLLOW_RULES_type**;

65.

⟨ Type defs 4 ⟩ +≡
  typedef FOLLOW_RULES_type :: **iterator FOLLOW_RULES_ITER_type**;

66.

⟨ Type defs 4 ⟩ +≡
  typedef std :: **set** ⟨ *T_in_stbl* ∗> *FIRST_SET_type*;

67.

⟨ Type defs 4 ⟩ +≡
  typedef FIRST_SET_type :: **iterator FIRST_SET_ITER_type**;

68. ⟨ Type defs 4 ⟩ +≡
```
typedef vector⟨T_ENO⟩ T_COUNT_type;
```

69. ⟨ Type defs 4 ⟩ +≡
```
typedef T_COUNT_type::iterator T_COUNT_ITER_type;
```

70. ⟨ Type defs 4 ⟩ +≡
```
typedef vector ⟨ T_in_stbl *> STBL_T_ITEMS_type;
```

71. ⟨ Type defs 4 ⟩ +≡
```
typedef STBL_T_ITEMS_type ::iterator STBL_T_ITEMS_ITER_type;
```

72. ⟨ Type defs 4 ⟩ +≡
```
typedef std::set ⟨ rule_def *> RULES_HAVING_AR_type;
```

73. ⟨ Type defs 4 ⟩ +≡
```
typedef RULES_HAVING_AR_type ::iterator RULES_HAVING_AR_ITER_type;
```

74. ⟨ Type defs 4 ⟩ +≡
```
typedef std::vector⟨LA_SET_type *⟩ COMMON_LA_SETS_type;
```

75. ⟨ Type defs 4 ⟩ +≡
```
typedef COMMON_LA_SETS_type::iterator COMMON_LA_SETS_ITER_type;
```

76. ⟨ Type defs 4 ⟩ +≡
```
typedef std::map⟨int, int⟩ BIT_MAP_type;
```

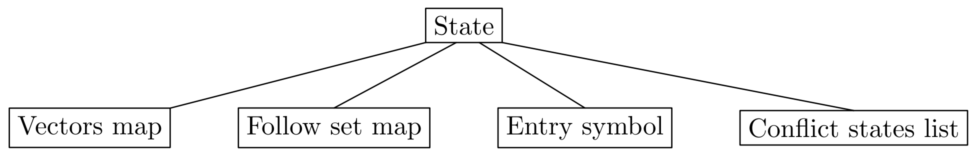
77. ⟨ Type defs 4 ⟩ +≡
```
typedef BIT_MAP_type::iterator BIT_MAP_ITER_type;
```

78. ⟨ Type defs 4 ⟩ +≡
```
struct use_cnt_type {
  rule_def * for_rule_use_cnt_;
  rule_def * against_rule_;
  int use_cnt_;
use_cnt_type(rule_def * R, rule_def * S): for_rule_use_cnt_(R), against_rule_(S), use_cnt_(0)
  { }
  ;
  use_cnt_type(const use_cnt_type &C)
  : for_rule_use_cnt_(C.for_rule_use_cnt_), against_rule_(C.against_rule_), use_cnt_(C.use_cnt_) { }
  ;
};
typedef std::map⟨int, use_cnt_type⟩ CYCLIC_USE_TBL_type;
typedef CYCLIC_USE_TBL_type::iterator CYCLIC_USE_TBL_ITER_type;
```

79. ⟨ Type defs 4 ⟩ +≡
```
typedef std::set⟨int⟩ SET_FILTER_type;
```

80. ⟨ Type defs 4 ⟩ +≡
```
typedef tok_can < AST *> TOK_CAN_TREE_type;
```

81. LR1 definitions.



vectors_map[eno]: symbol's enumerate

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

follow_set_map[eno]: rule's enumerate

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

Let's review what makes up a state:
1) subrules's specific element — state's to gen vectors
2) rules's follow set
3) state's entry symbol — Start state has no entry symbol
4) state's list of conflict states

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

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

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

Parts of a state:
1) Closured
2) Transitive

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

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

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

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

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

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

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

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

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

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

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

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

Building a state core:
There are only 2 contexts that provide the generation fodder for a state:

1) Start (closure-only) state — Start rule’s grammar tree definition
2) Transitive states — generated from a closured state

The “closured-part” of a state generates all its transitive states from its closure subrules regardless of the type of state — Start or transitive. Point 1 starts things off. It generates all its transitive states. Point 2 deals with transitive states from point 1 that have closured-only residues that need generating. Of course these newly generated transitive states could be merged into the existing lr1 state network if they meet the lr1 compatibility criteria. Eventually the newly added transitive states will be assessed for their “closured-part” generation.

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

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

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

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

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

```
⟨ Structure defs 82 ⟩ ≡
  struct gen_context {
    state *for_closure_state_;
    Voc_ENO gen_vector_;
    gen_context(state *S, Voc_ENO Ve);
  };
```

See also sections 83, 84, and 85.

This code is used in section 2.

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

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

```
⟨ Structure defs 82 ⟩ +≡
  struct state_element {
    gen_context cs_vector_combo_gening_it_;
    LA_SET_type *la_set_;
    state *closure_state_;
    state *goto_state_;
    state *reduced_state_;
    state *previous_state_;
    state *self_state_;
    state *closured_state_gening_it_;
    state_element *previous_state_element_;
    state_element *next_state_element_;
    AST * sr_element_;     /* grammar tree node */
    CAbs_lr1_sym * sr_def_element_;     /* 1 of terminal-def or rule-def */
    T_subrule_def * subrule_def_;
    int common_la_set_idx_;
    int its_enum_id_;     /* grammar's enumerated value of terminal-def or rule-def */
    state_element(AST * Elem);
    ~state_element( );
    void fill_la_from_merge(state_element &La_to_fill_in, MERGES_type &Merge, RULE_ENO
        Rule_no);
    void fill_la_from_transition(state_element &La_to_fill_in, TRANSITIONS_type &Transition);
    bool calc_la(state_element &La_to_fill_in);
    void add_fs_setA_to_LA(follow_element &Fe, LA_SET_type &La_to_fill_in);
    RULE_ENO find_state_element_s_rule_no( );
  };
```

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

```
⟨Structure defs 82⟩ +≡
  struct follow_element {
    RULE_ENO rule_no_;
    AST * rule_def_t_;
    state *its_state_;
    FOLLOW_SETS_type follow_set_;
    TRANSITIONS_type transitions_;
    MERGES_type merges_;
    SR_ELEMENTS_type sr_elements_;
    void add_T_to_follow_set(AST * T_element);
    void add_follow_set_contributor(AST * SR_element);
    void add_follow_set_transition(state_element &State_elem, T_eosubrule & Eos);
    void add_follow_set_transition(state_element &State_elem, T_called_thread_eosubrule & Eos);
    void add_follow_set_transition(state_element &State_elem, T_null_call_thread_eosubrule & Eos);
    AST * rule_def_t( );
    state *its_state( );
    follow_element(RULE_ENO Rule_no, state_element &State_elem, AST & Rule_def_t);
    follow_element(state *State);
    void remove_merge_closure_info( );
    void add_merge_closure_info(state &To_merge_closure_state);
  };
```

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

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

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

```
⟨Structure defs 82⟩ +≡
  struct state {
    gen_context cs_vector_combo_gening_it_;
    Voc_ENO vectored_into_by_elem_;
      /* enumerate from N or T vocabulary: -1= closure-only state */
    CAbs_lr1_sym *vectored_into_by_elem_sym_;    /* rule_def, T_def, 0=closure-only state */
    int state_no_;
    CLOSURE_RULES_type closure_rule_list_;
    CLOSURE_RULES_type derives_closure_rule_list_;    /* only for: rule recycling optimization */
    FOLLOW_RULES_type follow_rule_list_;
    void add_closure_rules_subrules_to_state(gen_context &Possible_gen_context, state &Closure_state);
    void add_rule_s_subrules_to_state(AST &Start_Rule_def_t, gen_context &Possible_gen_context, state
        &Closure_state_associate_with);
    bool crt_core_items_of_state(S_VECTOR_ELEMS_ITER_type &Iter_begin,
        S_VECTOR_ELEMS_ITER_type &Iter_end, gen_context &Gening_context);
    S_VECTORS_type state_s_vector_;
    S_FOLLOW_SETS_type state_s_follow_set_map_;
    S_CONFLICT_STATES_type state_s_conflict_state_list_;
    void add_element_to_state_vector(Voc_ENO Elem_id, state_element &Elem);
    void add_rule_to_closure_list(rule_in_stbl *Rule_in_stbl);
    void add_rule_to_follow_list(RULE_ENO Refered_rule);
    void create_follow_sets_of_state();
    void create_start_state(AST &Start_rule_t);
    bool gen_transitive_states_for_closure_context(gen_context &For_gening_context, state
        &For_closure_state, state &State);
    bool gen_transitive_states_balance_for_closure_vector(gen_context &Gen_context, state
        &For_closure_state, state &Goto_state);

    state(Voc_ENO Eno, CAbs_lr1_sym *Entry_sym);
    state(AST *Start_rule_t);

    bool gen_a_state(gen_context &For_gening_context, state &For_closure_state, state
        &Requesting_state, S_VECTORS_ITER_type &Elem_iter);
    const char *entry_symbol_literal();
    void add_state_to_gbl_lr1_state_tbls(state *State);
    void add_state_to_conflict_states_list_if(gen_context &Gening_context, state &State);
    state *closure_state_birthing_it_;
    bool is_state_lr1_compatible(state &State_to_eval);
    int are_2_states_compatible_yes_merge(state &To_merge_into_state, state &State_for_merging);
    void merge_state(state &To_merge_into_state, state &State_for_merging);
    int find_2_states_compatible_and_merge(state &State_for_merging);
    bool are_states_equivalent(state &Merge_into_state, state &To_merge_state);
    bool are_gened_states_lr1_compatible();
    bool is_str_rt_bnded(AST * Str);
    bool is_str_epsilonable(AST * Str);
    void crt_start_rule_s_follow_set(AST & Str);
    state();      /* for closure only derives */
    void closure_only_derives(AST * Rule_tree);
    int determine_reduced_state_type(state *S);
    int state_type_;
    string * arbitrator_name_;
  };
```

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

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

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

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

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

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

O2˙TYPES