1. Copyright.

2. *epsilon_rules* **grammar.**
Determine and check for the following:

1: epsilon rules
2: rules not deriving any string
3: referenced rules are defined
4: recursion cycling deriving only epsilon
5: ambiguous subrules caused by multiple recursion cycling

Epsilon definition:
A rule having a subrule with no grammar symbols present: ie, it derives an empty symbol string. This is explicit epsilon. A second condition is when a rule's subrule has all its symbols as rules and they are all epsiloned. I call this transient epsilon. It doesn't matter how the condition is arrived at. Now a third condition is the "invisible-shift" operator usuage. Though it's not part of the token stream per say it is normally used to get out of ambigous situations and so is part of the lookahead set. Once apon a time i thought this was an implicit epsilon: IT IS NOT.

Some epsilon terminalogy:
Noun: epsilon represents the empty string of symbols. The Greek flavour (ϵ) represents it visually.
Adjective: epsilonable - has the empty symbol string condition and non-epsilonable generates terminal strings.
Verb: shake and bake the tenses.
Verbal: epsiloned.

This grammar demonstrates the elegance of flow control. Review of a grammar's flow control:
This depends on how the grammar tree is walked. This grammar receives its tokens in prefix formation: a "rule-def" token is processed before its "subrule-def". Using sentences, a subrule's expression will be processed before its left-handside rule. This means the dependencies are in bottom-up left-to-right order of token recognition. One can use the prefix order to initialize the various conditions, and postfix the results from the bottom-up order. A grammar of no tokens can be programmed using all epsiloned rules whereby the rules act as logic triggers. In this paricular example, the grammar was originally written to detect "epsilon" rules. Now it detects whether all rules derive T. The grammar recognition just loads up into a 2 dimensional list the rule x subrule x element. Once the language is accepted, it then traverses the list to derive the terminal strings per subrule.

The Algorithm.
Pass one: fill up the derives list
Fill up the triplet list of all subrules: rule $\otimes$ subrule $\otimes$ element. One can view it as a 2 dimensional array where the row is subrule dominent with its subordinent element columns. The rule just tags along providing the parental anchor. The grammar tree is walked top-down (prefix order) to load up the derives list. Only the "rule-def" and "subrule-def" come in as tokens while the elements are filtered out. This was done for efficiency reasons. Within the grammar at the "subrule-def" logic point, the "subrule-def" start element is fetched by syntax directed code.

Pass 2: derive T for each subrule
Try to derive terminal strings for each indeterminate subrule. This stage does multilpe passes thru the derives list whereby each pass is predicated on the fact that the previous pass has a change in the status of a subrule — advance to the next element or the subrule's elements have been consumed. How is this a 2 dimensional walk? The subrules in the list are the dominant axis while its elements are the secondary axis. This is a dynamic grid of points being consumned. The final outcome is either the grammar's subrules all derive terminal strings and the derives list is empty or the remainder of items are the pathological subrules.

Examples of Pathological conditions:

Rule **A** calls itself or a variation of **A** calls **B** calls **A** without any terminal strings in their subrules.

Please look at the following test grammars that exercise the pathos...

1: *ts_path1.lex*
2: *ts_path1a.lex*
3: *ts_path2.lex*
4: *ts_path3.lex*
5: *ts_path4.lex*
6: *ts_path5.lex*
7: *ts_path6.lex*
8: *ts_path7.lex*

They are exercised by *qa_alltests.bat* batch command file.

3. Recursion and derives a string.

Derives:
Greek symbols are used to denote a mixed string of symbol(s) drawn from the nonterminal, terminal vocabularies. The empty string is represented by ϵ. Derives is a relationship that starts with a string of symbols and substitutes equivalent strings of symbols until eventually nomore substitutions can take place. The resulting string is either empty or a string of terminals. The order of substitution will be from left to right. The $\longrightarrow$ symbol represents the derives relation. Sometimes i will subscript the greek symbol with ϵ denoting that it derives the empty string. The $\not\longrightarrow$ symbol represents "no derives" takes place. I use this where a specific subrule pattern is being discussed and the greek symbol is not present.

Left recursion: A $\rightarrow$ A β
As shown, the rule **A** calls itself from the left part of its subrule. β represents strings of Rules and Terminals

Right recursion: A $\rightarrow \alpha$ A
The rule **A** calls itself from the end of the subrule as shown. α represents strings of Rules and Terminals.

4. Pathological Grammars.

The below pictures grids the good the bad and the ugly.
Note: the emitted code contains some imperfections caused by *cweave*. When i have time i'll deal with *cweave*. Reader beware.

5. Fsm Cepsilon_rules class.

6. Cepsilon_rules op directive.

⟨ Cepsilon_rules op directive 6 ⟩ ≡
```
rule_def_ = 0;
subrule_def_ = 0;
elem_t_ = 0; ip_can_ = ( tok_can < AST *> * ) parser__→token_supplier__;
```

7. Cepsilon_rules user-declaration directive.

⟨ Cepsilon_rules user-declaration directive 7 ⟩ ≡

```
public: std::list < elem_list_type > derives_list_;
  rule_def * rule_def_;
  T_subrule_def * subrule_def_;
  AST * elem_t_;
  tok_can < AST *> *ip_can_;
  void deal_with_derives_list( );
  void deal_with_undecided_derives_list( );
  AST * advance_element(AST * Elemt);
```

8. Cepsilon_rules user-implementation directive.

```
⟨ Cepsilon_rules user-implementation directive 8 ⟩ ≡
  AST * Cepsilon_rules::advance_element(AST * Elemt)
  {
    AST * nxt_t = AST::brother(*Elemt);      /* next element */
    for ( ; ; ) {      /* bypass thread expression: go to jail ... eosubrule */
      using namespace NS_yacco2_T_enum;
      CAbs_lr1_sym * sym = AST::content(*nxt_t);
      switch (sym→enumerated_id__) {
      case T_Enum::T_T_identifier_: break;
      case T_Enum::T_T_2colon_: break;
      case T_Enum::T_T_NULL_: break;
      case T_Enum::T_T_cweb_marker_: break;
      case T_Enum::T_T_cweb_comment_: break;
      default: return nxt_t;
      }
      nxt_t = AST::brother(*nxt_t);      /* skip thread expr: next element */
    }
  }
  void Cepsilon_rules::deal_with_derives_list(){      /* force cweave to */
      using namespace NS_yacco2_T_enum;
      bool has_list_chged(false);
      std::list < elem_list_type > ::iterator i;
    Read_list: ;
      i = derives_list_.begin( );
      if (i ≡ derives_list_.end( )) return;
      for ( ; i ≠ derives_list_.end( ); ) {      /* force cweave */
      elem_list_type & el = *i;
      CAbs_lr1_sym * rteos = AST::content(*el.elem_t_);      /* force cweave */
      switch (rteos→enumerated_id__) {      /* force cweave */
      case T_Enum::T_refered_rule_: { refered_rule * rr = ( refered_rule * ) rteos;
      rule_in_stbl * rstbl = rr→Rule_in_stbl( );
      using namespace yacco2_stbl;
      T_sym_tbl_report_card report_card;
      get_sym_entry_by_sub(report_card, rstbl→stbl_idx( ));
      if (report_card.status_ ≡ T_sym_tbl_report_card::failure) {
        report_card.err_entry_→set_rc(*rr, __FILE__, __LINE__);
        parser__→add_token_to_error_queue(*report_card.err_entry_);
        return;      /* caused by problem with sym. tbl */
      }
      if (report_card.tbl_entry_→defined_ ≡ false) {
        CAbs_lr1_sym * sym = new Err_used_rule_but_undefined;
        sym→set_rc(*rr, __FILE__, __LINE__);
        parser__→add_token_to_error_queue(*sym);      /* keep pouring them into the error queue */
        has_list_chged = true;      /* go to next element */
        el.elem_t_ = advance_element(el.elem_t_);
        ++i;      /* next sr in derive list */
        break;
      }
      rule_def * r = rr→Rule_in_stbl( )→r_def( );
      if (r→epsilon( ) ≡ YES) {
```

```
      has_list_chged = true;
      el.elem_t_ = advance_element(el.elem_t_);
      ++i;      /* next sr in derive list */
      if (el.gen_epsilon_ ≠ EPSILON_NO) {
        el.gen_epsilon_ = EPSILON_MAYBE;
      }
      break;
    }
    if (r→derive_t( ) ≡ YES) {
      has_list_chged = true;
      el.elem_t_ = advance_element(el.elem_t_);
      el.gen_epsilon_ = EPSILON_NO;
      ++i;      /* next sr in derive list */
      break;
    }
    ++i;      /* indeterminate spot: next pass might free it up so next sr */
    break; }
  case T_Enum :: T_T_eosubrule_:
    {
      if (el.gen_epsilon_ ≠ EPSILON_NO) {
        el.rule_→epsilon(YES);
        el.subrule_→epsilon(YES);
      }
      else {
        el.subrule_→epsilon(NO);
        el.rule_→derive_t(YES);
      }
      has_list_chged = true;
      i = derives_list_.erase(i);      /* finished with sr */
      break;
    }
  case T_Enum :: T_T_null_call_thread_eosubrule_:
    {
      el.gen_epsilon_ = EPSILON_NO;
      el.subrule_→epsilon(NO);
      el.rule_→derive_t(YES);
      has_list_chged = true;
      i = derives_list_.erase(i);      /* finished with sr */
      break;
    }
  case T_Enum :: T_T_called_thread_eosubrule_:
    {
      el.gen_epsilon_ = EPSILON_NO;
      el.subrule_→epsilon(NO);
      el.rule_→derive_t(YES);
      has_list_chged = true;
      i = derives_list_.erase(i);      /* finished with sr */
      break;
    }
    case T_Enum :: T_refered_T_: { el.gen_epsilon_ = EPSILON_NO;
    el.subrule_→epsilon(NO);
    has_list_chged = true;
```

```
  el.elem_t_ = advance_element(el.elem_t_);
  ++i;      /* next sr in derive list */
  refered_T * rt = ( refered_T * ) rteos;
  break; }
default:
  {     /* lr k */
    el.gen_epsilon_ = EPSILON_NO;
    el.subrule_→epsilon(NO);
    has_list_chged = true;
    el.elem_t_ = advance_element(el.elem_t_);
    ++i;      /* next sr in derive list */
    break;
  }
  } }
  if (derives_list_.empty( ) ≡ true) return;      /* kosher grammar */
  if (has_list_chged ≡ true) {      /* still o/s but going forward */
    has_list_chged = false;
    goto Read_list;
  }
  } void Cepsilon_rules :: deal_with_undecided_derives_list( )
  {     /* chk if undefined rules were found: yes out damn spot */
    if (parser__→error_queue( )→empty( ) ≠ true) return;      /* errors: pathological grammar */
    std :: list < elem_list_type > :: iterator i = derives_list_.begin( );
    if (i ≡ derives_list_.end( )) return;
    for ( ; i ≠ derives_list_.end( ); ++i) {
      elem_list_type & el = *i;
      CAbs_lr1_sym * elem = AST :: content(*el.elem_t_);
      CAbs_lr1_sym * sym = new ERR_sick_grammar;
      sym→set_rc(*elem, __FILE__, __LINE__);
      parser__→add_token_to_error_queue(*sym);
    }
  }
```

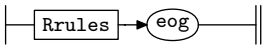
9. Cepsilon_rules user-prefix-declaration directive.

⟨ Cepsilon_rules user-prefix-declaration directive 9 ⟩ ≡

```
#include "yacco2_stbl.h"
  using namespace NS_yacco2_terminals;
#define EPSILON_YES  0
#define EPSILON_NO  1
#define EPSILON_DONT_KNOW  2
#define EPSILON_MAYBE  3
  struct elem_list_type {
    rule_def * rule_;
    T_subrule_def * subrule_;
    AST * elem_t_;
    int gen_epsilon_;
    int gen_t_;
    elem_list_type(rule_def * Rule, T_subrule_def * Subrule, AST * Elem_t)
    {
      rule_ = Rule;
      subrule_ = Subrule;
      elem_t_ = Elem_t;
      gen_epsilon_ = EPSILON_DONT_KNOW;
      gen_t_ = false;
    }
    ;
    elem_list_type( )
    {
      rule_ = 0;
      subrule_ = 0;
      elem_t_ = 0;
      gen_epsilon_ = false;
      gen_t_ = false;
    }
    ;
  };
```

10. *Repsilon_rules* **rule.**

```
Repsilon_rules
```



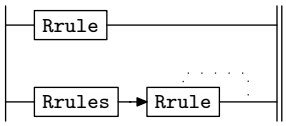
Pass 2 and greater.

⟨ Repsilon_rules subrule 1 op directive 10 ⟩ ≡

```
  Cepsilon_rules * fsm = ( Cepsilon_rules * ) rule_info__.parser__->fsm_tbl__;
  fsm->deal_with_derives_list( );
  if (fsm->derives_list_.empty( ) ≠ true) {
    fsm->deal_with_undecided_derives_list( );
  }
```

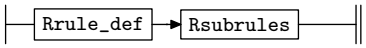
11. *Rrules* **rule.**

Rrules



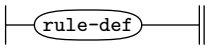
12. *Rrule* **rule.**

Rrule



13. *Rrule_def* **rule.**

Rrule_def



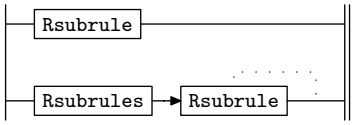
Initialize for its subrule findings.

⟨ Rrule_def subrule 1 op directive 13 ⟩ ≡

```
Cepsilon_rules * fsm = ( Cepsilon_rules * ) rule_info__.parser__->fsm_tbl__;
fsm->rule_def_ = sf->p1__;
```

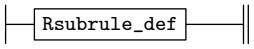
14. *Rsubrules* **rule.**

Rsubrules



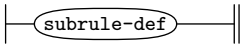
15. *Rsubrule* **rule.**

Rsubrule



16. *Rsubrule_def* **rule.**

Rsubrule_def



Create the entry within the *derives_list_*. This will be walked until either all the subrules' elements are derived or they are pathological.

⟨ Rsubrule_def subrule 1 op directive 16 ⟩ ≡

```
Cepsilon_rules * fsm = ( Cepsilon_rules * ) rule_info__.parser__->fsm_tbl__;
fsm->subrule_def_ = sf->p1__;
AST * sr_t = fsm->subrule_def_->subrule_s_tree( );
AST * et = AST::get_spec_child(*sr_t, 1);
fsm->derives_list_.push_back(elem_list_type(fsm->rule_def_, fsm->subrule_def_, et));
```

17. First Set Language for O_2^{linker}.

```
/*
 File: epsilon_rules.fsc
 Date and Time: Fri Jan  2 15:33:33 2015
*/
transitive   n
grammar-name "epsilon_rules"
name-space   "NS_epsilon_rules"
thread-name  "Cepsilon_rules"
monolithic   y
file-name    "epsilon_rules.fsc"
no-of-T      569
list-of-native-first-set-terminals 1
   rule_def
end-list-of-native-first-set-terminals
list-of-transitive-threads 0
end-list-of-transitive-threads
list-of-used-threads 0
end-list-of-used-threads
fsm-comments
"Determine whether rules are epsilon, derive T, or are pathological."
```

18. Lr1 State Network.

⇒ State: 1 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
c	Rrule_def		4	1	1	rule-def		1	2	2	
c	Rrules		2	2	1	Rrules *Rrule*		1	3	5	
c	Repsilon_rules		1	1	1	Rrules *eog*		1	3	4	
c	Rrules		2	1	1	Rrule		1	12	12	
c	Rrule		3	1	1	Rrule_def *Rsubrules*		1	6	8	

$\Rightarrow^{rule-def}$ State: 2 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rrule_def		4	1	2			1	0	2	1

$\Rightarrow^{Rrules}$ State: 3 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Repsilon_rules		1	1	2	eog		1	4	4	
c	Rrule_def		4	1	1	rule-def		3	2	2	
t	Rrules		2	2	2	Rrule		1	5	5	
c	Rrule		3	1	1	Rrule_def *Rsubrules*		3	6	8	

$\Rightarrow^{eog}$ State: 4 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Repsilon_rules		1	1	3			1	0	4	2

$\Rightarrow^{Rrule}$ State: 5 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rrules		2	2	3			1	0	5	3

$\Rightarrow^{Rrule\_def}$ State: 6 state type: ^{s}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
c	Rsubrule_def		7	1	1	subrule-def		6	7	7	
t	Rrule		3	1	2	Rsubrules		3	8	8	
c	Rsubrules		5	2	1	Rsubrules *Rsubrule*		6	8	9	
c	Rsubrules		5	1	1	Rsubrule		6	11	11	
c	Rsubrule		6	1	1	Rsubrule_def		6	10	10	

$\Rightarrow^{subrule-def}$ State: 7 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrule_def		7	1	2			6	0	7	4

$\Rightarrow^{Rsubrules}$ State: 8 state type: $^{s/r}$

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rrule		3	1	3			3	0	8	3
c	Rsubrule_def		7	1	1	subrule-def		8	7	7	
t	Rsubrules		5	2	2	Rsubrule		6	9	9	
c	Rsubrule		6	1	1	Rsubrule_def		8	10	10	

$\Rightarrow^{Rsubrule}$ State: 9 state type: ^{r}

	← rule	→	R#	sr#	Po	← subrule element	→	Brn	Gto	Red	LA
t	Rsubrules		5	2	3			6	0	9	4

$\Rightarrow^{Rsubrule\_def}$

	←	rule	→	R#	sr#	Po	←
t	Rsubrule			6	1	2	

State: 10 state type: ^{r}

	subrule element	→	Brn	Gto	Red	LA
			8	0	10	4

$\Rightarrow^{Rsubrule}$

	←	rule	→	R#	sr#	Po	←
t	Rsubrules			5	1	2	

State: 11 state type: ^{r}

	subrule element	→	Brn	Gto	Red	LA
			6	0	11	4

$\Rightarrow^{Rrule}$

	←	rule	→	R#	sr#	Po	←
t	Rrules			2	1	2	

State: 12 state type: ^{r}

	subrule element	→	Brn	Gto	Red	LA
			1	0	12	3

19. Index.

epsilon_rules Grammar

Date: January 2, 2015 at 15:35

File: epsilon_rules.lex

Ns: NS_epsilon_rules

Version: 1.0 Debug: false

Grammar Comments: Type: Monolithic

Determine whether rules are epsilon, derive T, or are pathological.