1. Copyright.

2. K symbols vocabulary.

Ahh the "Constant grammar" symbols used throughout all grammars. Depending on the command line options O_2 can generate the grammar and possibly the various flavours of the Terminal vocabulary. Under normal development, the grammar writer compiles and emits just the grammar. Flavours of the terminal vocabulary are not usually generated unless there have been changes to either **error** or **terminal** with accompaning command line option. At the initial "big bang" of bootstrapping O_2's library and compiler / compiler, both **raw characters** and **lr k** terminals were generated using their command line options /rc and /lrk. The command line option now uses an Unix style -t, -err and these 2 terminal types "lrk" and "raw characters" are now cast in cement: u cannot regen them from their "*.T" file definitions but u can change their "big bang" generated "c++" modules. These terminals are now read-only: they will never be changed by a user of O_2 and who'd want to anyway?.

The hardwired "k" terminals are used by O_2's library for internal parsing situations. Apart from **eog** who represents the end-of-grammar and end-of-file conditions, all other definitions are not part of the token source stream being parsed. I call them meta terminals as they are never in the token stream but represent internal parsing conditions within the emitted finite-state table that triggers the O_2's library routines. For example, the presence of `|||` within a parse state indicates the potential "to run" threads. If u look carefully, their file definitions and implementations reside in O_2's "../yacco2/library/grammars" folder. Their definition files are "yacco2_k_symbols.T" and "yacco2_characters.T" with their "c++" variants having the ".h" and ".cpp" extensions.

3. eog.

Enum: T_LR1_eog_
Class: LR1_eog AB: N AD: N

Used to indicate an end-of-grammar or an end-of-file condition. When the token container is reached, calls for another terminal will always return the *eog*. It's your door bouncer before hell.

---

4. eog user-declaration directive.

⟨ eog user-declaration directive 4 ⟩ ≡
  LR1_eog( );

5. eog user-implementation directive.

⟨ eog user-implementation directive 5 ⟩ ≡
  LR1_eog :: *LR1_eog*( )`T_CTOR("eog"`, *T_LR1_eog_*, 0, *false*, *false*)
  { }
  LR1_eog *LR1_eog__*;
  yacco2 :: *CAbs_lr1_sym* ∗ *yacco2* :: *PTR_LR1_eog__* = &*LR1_eog__*;

6. eolr.

Enum: T_LR1_eolr_
Class: LR1_eolr AB: N AD: N

Used to indicate all-terminals of the terminal vocabulary including itself. It saves finger blisters by not having to be explicit in the thread's lookahead expression. Dieting hasn't been this effective to code bloat.

---

7. eolr user-declaration directive.

⟨ eolr user-declaration directive 7 ⟩ ≡
  LR1_eolr( );

8. eolr user-implementation directive.

⟨ eolr user-implementation directive 8 ⟩ ≡
```
  LR1_eolr :: LR1_eolr( )T_CTOR("eolr", T_LR1_eolr_, 0, false, false)
  { }
  LR1_eolr LR1_eolr__;
  yacco2 :: CAbs_lr1_sym * NS_yacco2_k_symbols :: PTR_LR1_eolr__ = &LR1_eolr__;
```

9. `|+|`.

Enum: T_LR1_all_shift_operator_
Class: LR1_all_shift_operator AB: N AD: N

Represents the wild token situation. Lowers the specific shifts of the finite-state-table and allows the grammar writer to field the unexpected from returned threads. Good stuff.

Caveat: One should use the `|?|`to field unknow return Tes if they are to be interpreted as errors.

---

10. `|+|{} user-declaration directive.`

⟨ `|+|{} user-declaration directive` 10 ⟩ ≡
```
  LR1_all_shift_operator( );
```

11. `|+|{} user-implementation directive.`

⟨ `|+|{} user-implementation directive` 11 ⟩ ≡
```
  LR1_all_shift_operator :: LR1_all_shift_operator( )T_CTOR("|+|", T_LR1_all_shift_operator_, 0, false, false)
  { }
  LR1_all_shift_operator LR1_all_shift_operator__;
  yacco2 :: CAbs_lr1_sym * NS_yacco2_k_symbols :: PTR_LR1_all_shift_operator__ = &LR1_all_shift_operator__;
```

12. `|.|`.

Enum: T_LR1_invisible_shift_operator_
Class: LR1_invisible_shift_operator AB: N AD: N

It's a nice way to program out of an ambiguous grammar. It can also lower the code bloat of a thread's first set.

---

13. `|.|{} user-declaration directive.`

⟨ `|.|{} user-declaration directive` 13 ⟩ ≡
```
  LR1_invisible_shift_operator( );
```

14. `|.|{} user-implementation directive.`

⟨ `|.|{} user-implementation directive` 14 ⟩ ≡
```
  LR1_invisible_shift_operator :: LR1_invisible_shift_operator( )T_CTOR("|.|",
          T_LR1_invisible_shift_operator_, 0, false, false)
  { }
  LR1_invisible_shift_operator LR1_invisible_shift_operator__;
  yacco2 :: CAbs_lr1_sym * NS_yacco2_k_symbols :: PTR_LR1_invisible_shift_operator__ =
      &LR1_invisible_shift_operator__;
```

15. `|?|.`
Enum: T_LR1_questionable_shift_operator_
Class: LR1_questionable_shift_operator AB: N AD: N
Represents a questionable grammar situation. It pinpoints programmed error points within the grammar. The subrule using this symbol has a lr(0) reduction as the lookahead is not kosher and so would probably not reduce in the lr(1) context. It can be used both in the following grammar expressions:

1) →`|?|`
2) →`|||` `|?|` NULL

Point 1 covers the state where the current token being parsed is improper. Point 2 is more interesting as it captures a returned terminal that the thread passes back as an error.

The `|?|`was not one of the original "k" terminals. It replaced the "eof" terminal which was marginal in intent. I felt the `|?|`symbol drew the reader's eye of the grammar where "faulty" points where captured and to force lr(0) context processing to reduce its subrule. Why lr(0) context? Glad u asked, the lookahead terminal — the current terminal being parsed, is in error and so "how is the subrule with the `|?|`to reduce after its shifted T?". It must be divorced of any lookahead and just acted upon.

Now another question arises: "how is this condition detected in a parsing state of mixed conditions — threading, shifting, reducing"? There is a pecking order on the conditions tried by the parser:

- ○ threading
  - if tried and unsuccessful the balance of conditions are attempted
- ○ shifts pecking order by their presence in current parse state:
  - can the current token be shifted?
  - `|?|`— error condition
  - `|.|`— explicit ϵ
  - `|+|`— any terminial
- ○ reduce
  - note shifting is favoured over reducing

---

16. `|?|{} user-declaration directive.`

⟨ `|?|{} user-declaration directive` 16 ⟩ ≡
LR1_questionable_shift_operator( );

17. `|?|{} user-implementation directive.`

⟨ `|?|{} user-implementation directive` 17 ⟩ ≡
LR1_questionable_shift_operator :: *LR1_questionable_shift_operator*( )`T_CTOR("|?|",`
*T_LR1_all_shift_operator_*, 0, *false*, *false*)
{ }
LR1_questionable_shift_operator LR1_questionable_shift_operator__;
yacco2 :: *CAbs_lr1_sym* ∗ *NS_yacco2_k_symbols* :: *PTR_LR1_questionable_shift_operator__* =
&*LR1_questionable_shift_operator__*;

18. `|r|.`
Enum: T_LR1_reduce_operator_
Class: LR1_reduce_operator AB: Y AD: Y
Its presence within the individual state of the "fsm" table is to force a reduce operation. Why? it's a back-to-back situation within the state table whereby a thread should reduce while its reducing lookahead is the `|||`indicating to run a thread.

---

19. `|t|`.

Enum: T_LR1_fset_transience_operator_
Class: LR1_fset_transience_operator AB: Y AD: Y

`|t|` has dual purposes: used in O_2linker to process the transient first sets generated by threads, and used within a grammar's "chained call procedure" expression to lower thread overhead by calling a procedure with explicit intent on double use of its "first set" token. I'll give an example of a "chained procedure call" expression drawn from the "pass3.lex" grammar handling the grammar's file include expression:

→"@" Rprefile_inc_dispatcher

The "Rprefile_inc_dispatcher" grammar rule has the following subrule:

→`|t|`"file-inclusion" NS_prefile_include::PROC_TH_prefile_include

The "chained" part is in the duplicating of "@"; that is, the parsing mechanism does not get a new terminal when shifted but passes this T onto the called procedure. The called PROC_TH_prefile_include procedure / thread has its start rule as:

→"@" Rpossible_ws Rfile_string Reof

The repeated use of "@" was to reenforce the idea that the procedure called cuz of "@": there's that "first set" again. Well time will pass its comments on this thought process.

---

20. `|t|{} user-declaration directive`.

⟨ `|t|{} user-declaration directive` 20 ⟩ ≡
LR1_fset_transience_operator( );

21. `|t|{} user-implementation directive`.

⟨ `|t|{} user-implementation directive` 21 ⟩ ≡
LR1_fset_transience_operator :: *LR1_fset_transience_operator*( )`T_CTOR("|t|",`
*T_LR1_fset_transience_operator_*, 0, *false*, *false*)
{ }
LR1_fset_transience_operator LR1_fset_transience_operator__;
yacco2 :: *CAbs_lr1_sym* ∗ *NS_yacco2_k_symbols* :: *PTR_LR1_fset_transience_operator__* =
&*LR1_fset_transience_operator__*;

22. `|||`.

Enum: T_LR1_parallel_operator_
Class: LR1_parallel_operator AB: N AD: N

Its presence within the individual state of the "fsm" table dictates potential threads to run. You see it sprinkled throughout my grammars to call threads. This is part of O_2's raison d'être.

---

23. `|||{} user-declaration directive`.

⟨ `|||{} user-declaration directive` 23 ⟩ ≡
LR1_parallel_operator( );

24. `|||{} user-implementation directive.`

⟨ `|||{} user-implementation directive` 24 ⟩ ≡

```
LR1_parallel_operator::LR1_parallel_operator()T_CTOR("|||", T_LR1_parallel_operator_, 0, false, false)
{ }
LR1_parallel_operator LR1_parallel_operator__;
yacco2::CAbs_lr1_sym *NS_yacco2_k_symbols::PTR_LR1_parallel_operator__ = &LR1_parallel_operator__;
```

25. lrk-sufx directive.

As they are constants, they are defined globally to save space / overhead in the typical new create / delete cycle of terminals. Thar's recycling going on in this green space.

⟨ lrk-sufx directive 25 ⟩ ≡

```
extern yacco2::CAbs_lr1_sym*PTR_LR1_parallel_operator__;
extern yacco2::CAbs_lr1_sym*PTR_LR1_fset_transience_operator__;
extern yacco2::CAbs_lr1_sym*PTR_LR1_invisible_shift_operator__;
extern yacco2::CAbs_lr1_sym*PTR_LR1_questionable_shift_operator__;
extern yacco2::CAbs_lr1_sym*PTR_LR1_all_shift_operator__;
extern yacco2::CAbs_lr1_sym*PTR_LR1_eolr__;
```

26. Index.

Lr K Vocabulary

Date: January 2, 2015 at 16:31

File: yacco2_k_symbols Namespace: NS_yacco2_k_symbols

Number of terminals: 8