Literate programming applied to my compiler / compiler's output documents

Dave Bone

Abstract

I describe in a light hearted way a variant on "literate programming" [1] as applied to my compiler / compiler's output grammar documents. Emitted grammar tables are plain boring. How do you spice them up such that they become a treat to proof-read one's logic? The question is double-ended: the input grammar file with its logic should be written in a way to convey understanding and intent of material whilst the derived output tables should convey their meaning in a higher level rather then bits and bytes and cpp code. This article hopefully will show to the reader how the "marriage of TEX with Metapost and cweb's tools" provides a decent framework to document grammars. I leave the read and conclusion to you the reader. Let the telling begin...

1 Introduction — the ugly ducking[4].

Let's look at a grammar file of plain characters. Below is an example of a raw grammar file with prefixed line numbers that I'll reference them with my running comments. It is created by some form of text editor. I ask you now to view it as if you were visiting an art gallery. To each their own in interpretation, thoughts, questions, about this object before you. I'm not trying to place this object into the sphere of art but ask you just to view it as a canvas and apply your own aesthetic. The content is not important nor is an expertise in compiler theory required.

```
 1  /*
 2  FILE:             eol.lex
 3  Dates:            17 Juin 2003
 4  Purpose:          end-of-line recognizer
 5  */
 6  /@
 7  @** Eol Thread.\fbreak
 8  Claim-to-fame is its end-of-line matching
 9  for Unix, Mac, or Windows variants.
10     ...
11  @/
12  fsm (fsm-id "eol.lex",fsm-filename eol
13  ,fsm-namespace NS_eol,fsm-class Ceol
14  ,fsm-version "1.0"
15  ,fsm-date "6/2003",fsm-debug "false"
16  ,fsm-comments "end-of-line matcher.")
17  parallel-parser
18  (
19  parallel-thread-function TH_eol ***
20  parallel-la-boundary eolr ***
21  )
22  @"/yacco2/compiler/grammars/yacco2_T_inc.T"
23  /@
24  @** Rules.\fbreak
25  @/
26  rules{
27  Reol (){
28    -> Rdelimiters {
29   /@
30   Return the |eol| token back to the caller.
31   @/
32      op
33   CAbs_lr1_sym* sym = NS_o2_terms::PTR_eol__;
34   sym->set_rc(*parser()->start_token__
35   ,rule_info__.parser__,__FILE__,__LINE__);
36   sym->set_line_no_and_pos_in_line
37   (*parser()->start_token__);
38   RSVP(sym);
39     ***   }
40  }
41  /@
42  @** |Rdelimiters| rule.\fbreak
43  @/
44
45  Rdelimiters
46  /@Eol Variants.
47  \invisibleshift escapes from shift/reduce
48  conflict caused by x0d common prefix.
49  As it's not in the token stream, it deals
50  effectively with S/R conflict type.
51  @/
52  ()
53  {
54    ->
55  /@ lf: Unix\fbreak@/
56  "x0a"
57    ->
58    /@ cr: Mac.
59    Note \invisibleshift removes
60    shift/reduce conflict caused by the
61    lookahead boundary of |eolr|.
62    \fbreak
63    @/
64
65  "x0d" |.|
66    ->  /@ cr:lf Windows\fbreak@/ "x0d" "x0a"
67  }
68  }// end of rules
```

Figure 1: A raw grammar file

Visually it's unstructured and plain scrappy as paper being wind pushed along a street; it's just not eye catching. To paraphrase "the media is the message?". Well some message and what is it? Try your own data-mining on the file with conjectures but it still leaves you dissatisfied giving way to your brisk dismissal of "lets look elsewhere" for entertainment.

Please bear with me and let's pick out some clues to content. Lines 1 – 5 are basic "c++" comments of the ordinary. For people familiar with cweb you will see a play on *cweave*'s directives and c++ comments at lines 6 – 11: again is this for my eyes only? Following this is the grammar's logic which is another language from lines 12 – 68 with syntax directed cpp code[3] mixed in as exampled by lines 32 – 39. All of this is like some score needing interpretation by some artist. The same type of comments can be levelled at a LaTeX's text file. But as you well know, what savoury delights are produced by LaTeX. Well I better achieve some facsimile or that white cane will remove me off this dancing stage.

2 What should the output documents look like?

When evaluating this question, it was rather easy to list my wants: the information had to convey intent succinctly, with high visualisation density, and be open to other visual aids to improve readership. Is this a Madison Avenue comment on cosmetics? No just my take as I'm not trying to replace content with visual effects for monetary rewards. To the best of my sensitiveness, effective props are aids to help me and others understand "what the code should be doing" in the development and maintenance cycles.

There are many takes on "how to" improve software development and its resultant code. It is a many dimensional universe to which some of my axis are "literate programming" [1] and "quality assurance" — system testing a system. I do not advocate one approach over another as each to their own devices in producing robust code. But software is maintained and without an accurate blueprint leads one to continually investigate as in "this old house" approach before renovations can be done. Cryptic comments in margins allows one to muse on musical scores and author's intent but this can just reinforce software's fragility.

2.1 Shopping list of wants

Now to documenting, I didn't want to write "yet another documenting" system: time told me to recycle some other tool's abilities for my own purposes. Cweb's tool kit that I use to write software has *cweave* to generate documents. So why not investigate its features such that a *cweave* file of grammar content could be constructed. Please see [10] for a detailed description of *ctangle* and *cweave* the tag team to "literate programming". Below is a summary of some of its features that I wanted to use:

- Title and Table of contents facility
- Index
- Variations to cross references entries within the Index
- Graphic inclusion from *MetaPost* [5]
- Full TEX coding capability — create and use of macros
- A c++ code beautifier with cross references to its variables and procedures in the Index

This is quite a list of capabilities that all come out of the "cracker jack box" so to speak. Now it became a simple process to ferret out the commands needed to generate the grammar into a *cweave* file. To which its output generates a TEX file to be digested by some program like "pdftex" to produce a pdf document. In Unix terms a pipelining of output from one program as input for another program to eventually assemble a result. Finally this document is rendered to a printer or screen by a "pdf" reader like "xpdf". Just think, the mundane example of Figure 1 now has the potential to become svelte.

2.2 Some benefits to openness

As my project progressed, it became more clear how "open ended" TEX, *MetaPost*, and cweb were. Here we take it for granted but TEX is a visual compiler. Even more it has its own dynamic language called macros to program your own visual effects. As a system TEX has many levels of languages being digested, created, and digested... The trick is to plug into the stage that you want.

From a grammar developer, I wanted to apply this openness to its generated files of *cweave* and *MetaPost* for graphic effects. As each tool interprets its own programming language, I would build a simple set of library routines for each program to compose documentary effects. Now these subroutines can be refined by rewriting them outside of the c++ compiler / compiler recompile and link cycle; a leaning to allow each author liberties to express their own tastes.

3 Compiler / compiler system overview

Before discussing these output documents, let's see how this system looks like and what documents are generated. Figure 2 shows how a grammar file is compiled with various outputs. One of its outputs is another language that becomes input to its companion program the "grammar linker".

3.1 Where's the beef?

You can skip this subsection if it's too technical. I digress a bit as to why there are individual grammars making up a language whole. As a comment, designing languages for computers be it for PC-to-human interaction or computer-to-computer interaction (like Internet protocols) takes many parts to describe and recognise the language. One of my beefs in writing a grammar to describe a language is that Yacc and variants [8] never applied the principle that many parts should make up a whole. If you ever looked at a formal grammar describing a language like Java [6], it is one monolithic (big — stop being polite call it as it is: horribly huge and marginally readable) grammar. Where are its parts and their dividing lines? Its rules "u say". Well try dealing with grammars describing the "c" or "c++" languages (if you can find

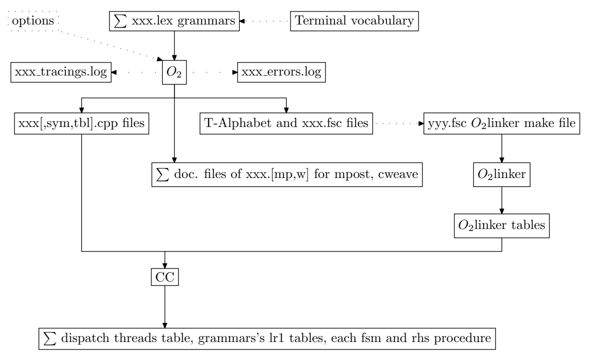
options
∑ xxx.lex grammars
Terminal vocabulary
xxx_tracings.log
O_2
xxx_errors.log
xxx[,sym,tbl].cpp files
T-Alphabet and xxx.fsc files
yyy.fsc O_2linker make file
∑ doc. files of xxx.[mp,w] for mpost, cweave
O_2linker
O_2linker tables
CC
∑ dispatch threads table, grammars's lr1 tables, each fsm and rhs procedure

Figure 2: Overview of compiler /compiler run environment

them), their precedence rules and even more Yacc's extensions dealing with the grammar's ambiguities. Well there's beef number one.

Other questions like "How easy is it to add or extend language features?" or "How easy is it to disambiguate the grammar?" are tests of bulldog tenacity (euphemism for trial and error) in trying to get it right. Does any one really understand their logic expressions without debugging them to build up an understanding? Why aren't grammar snippets reusable for other languages? Enough of your rantings Dave! There's beef number two from the reader.

Grammars can be monolithic but they should also be small and reusable, be developed separately, tested separately, and then brought into an assembly line to make up the compiler as one whole. So for you people who are familiar with compiler/compilers like Yacc, my approach is off the wall and completely different. All this to say that there are many grammars like subroutines written and then assembled by the "grammar linker" described in Section 5.3. One of the side benefits to this linker is a generated document of encapsulated comments drawn from each grammar.

3.2 A little thing called ... threads

My compiler/compiler uses the best of both worlds to recognise and compile a language. It uses a Lr(1) algorithm to compile the grammars [9] supporting syntax-directed code that can call top/down recursive descent procedures that themselves use bottom-up parsing techniques. (This is a mouthful: bring out the "soap and brush" and teach the child some etiquette.) Nothing like a bit of everything. All this to make it simpler and easier in developing language recognisers by the compiler writer.

Grammar modularity is packaged in its own thread [2]. Threads are controlled by a "Pthread" environment. To make things more expressive, a thread can call other threads that could possibly turn onto self; this is equivalent to recursion using a different runtime technique. I call this nested threading.

3.3 Let the hoards run amok?

Nondeterminism [1] allows threads to compete against one another. Their outcomes use "context sensitive arbitration" to mediate the outcome. If needed, all this mediation is controlled by the grammar writer through syntax-directed code. From a bird's eye perspective a thread calls its workers while waiting for a result: in computer terms it's known as a master/slave relationship. These workers can themselves become a master waiting for an outcome. The net result is the original master eventually gets its requested outcome after all the workers have tried doing their tasks.

How do you know when to let these hoards loose? Bottom-up parsing uses tables describing the execution states. Within each state are contexts as to what parsing action can take place. The controlling program (automaton) uses a stack mechanism to keep track of its parse states. To answer the question, depending on the current parse state there could be a list of

[1] a term for tasks being done in parallel

potential threads in its table to be run. To distinguish various contexts within the grammar, I use graphic symbols to highlight their conditions:

- thread call — | | | symbol
- error point within the grammar — | ? | symbol
- accept anything — |+| symbol
- get out of an ambiguous situation — | . | symbol
- epsilon symbol ϵ to indicate the empty symbol string

Left recursion is graphically drawn in the same manner as Pascal's railroad diagrams [7].

4 Grammar Linker — the gossip columnist 5.3

Before describing the linker, let me pose a question: "What are the things that determine when the grammar thread should be called?". In grammar terms there is a vocabulary of terminals: a term describing the basic unit to play with. It is these terminals that the linker needs to figure out for each thread before the parser calls it from its current parse state. It's a little more complicated then this as a thread's starting point could call another thread ... All this to say that the linker figures this out from all the grammars' output linker summaries: it's the party line operator to "who-calls-who" in the thread world. If a thread demands execution but it calling terminal(s) is not present then the thread call is skipped. So much for another very quick overview of my parsing automaton. Here is the Linker's run environment. Figure 3

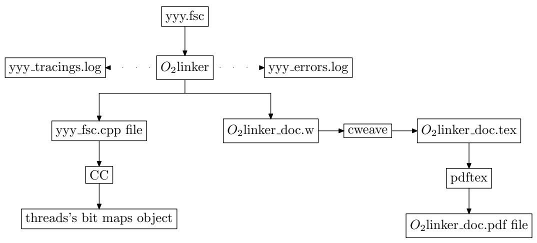

Figure 3: The Gossiper — linker run environment

depicts what feeds the linker and what it manufactures as cpp code and its "bible of grammars" document. This compendium sure helps in overall debugging with each grammar commenting about itself. Each comment is drawn from the grammar's construct "fsm-comments". You can peek again at the "ugly duckling" or go directly to "not jail but" a finished example. Each grammar's thread calling dependencies are also expressed within this document. As an aid dealing with a problem or researching to extend a language construct, use of *cweave*'s cross referencing facility pre-googles the now web search.

5 Enough already show me the goods

To end this article I'll summarise the content of each document. I apologise as some of the material to be appreciated requires knowledge about compiler theory of languages. Please view these documents as a TEXnolgist and connoisseur of documents without concern for their content — are they effective in use of underlining, sub and superscripting etc? The three documents are:

- the grammar (one document per grammar)
- the companion grammar's "lr state network" supporting material
- the linker's compendium of all compiled grammars

The grammar's vocabulary also uses "literate programming" alias *cweave* in the same manner described to generate their documents. As an aside the vocabulary is broken down into 4 classes: special constants, raw characters, errors, and lastly the user terminals. Each has its own document with a "Table of contents" being the jewel in their document's cap. I left them out due to space but mention them to complete the review.

5.1 Main document

The main document is the compiled grammar. It contains the grammar writer's *cweave* comments, its "literate program" components with *MetaPost* drawn Pascal's railroad like diagrams of the individual grammar's rules, the thread material for the companion *linker*, and finally the emitted "lr state network" table. Not to be too technical, the "lr state network" is very dense in information and specific to bottom-up parsing. I looked at various graphic programs to draw the network but they were too sparse in enriching their content within each state's node. So I choose to use a 2 document approach and to save trees: the second document is usually used when there are problems or when one is reviewing efficiency.

The "lr state network" is expressed in the form similar to a spreadsheet with enhancements like superscripting of the state's entry symbol, underlining to emphasise conditions within the rule's string of symbols, graphic symbols to draw the reader's eye to intent: for example, error points within the grammar, thread calls, wild terminals, epsilon sub-rules. Of course we have cross referencing of symbols for the curious and the human debugger.

5.2 The shadow of Main

The two document approach allows the grammar writer to make sense out of the cpp emitted "lr state network" tables. It provides various cross referencing material about the grammar's rules and their used symbols, follow set and lookahead sets of terminals (how they are derived and where they are used), reducing states and their state types. There's a lot more density of information that only the compiler writer would appreciate.

5.3 A tattler of tales?

As to the "linker" compendium, it lists all the grammars compiled with a nice "Table of content" of comments per grammar. The rest of its contents pertain to called threads and terminals that cause the thread to be called[2]. There's not too much content except that it summarises all those hoards. To give a feel to individual grammar numbers developed by me, a Structured Pascal translator had approximately 90 grammars. My compiler/compiler has 78 grammars. Experience writing them had this compendium act as a GPS to problem solving when "where to search when things went afoul" arose. Given that a specific grammar is causing a problem, this document cross references what grammar called whom in the chain of calls.

Now let's place this in its current run environment of parallel multi-threaded parsing with simultaneous multi-core activities. Sequential processing pales in debugging when many sequential processes are simultaneously active. Execution trace order of a thread is interleaved quasi-randomly with other tracing threads as they all log to a common trace file. There are different types of tracing classes but one of them is trace the grammar if its "debugging" option is on. If you turn on all grammars for tracing, the log file is rather big and can blow up an editor due to volume when browsing through it. So how do you fine-tune what grammars you would like to see traced? This document steers the grammar writer into choosing what grammars to turn on so to speak in their chain of calls. This way only executing grammars with debugging turned on will selectively log their runtime stacks etc. It becomes just another member in the set of diagnostic aids in getting things right.

5.4 Comments in unsavoury places?

In keeping with "literate programming" tradition of emitting cpp comments, my emitted code also contains running c++ grammar comments to pinpoint to the *cweave* source. This gives the programmer a reference point while reviewing the code or debugging it. From my experience it is extremely rare that one needs to look at the cpp code even if you are debugging a grammar. There are other tracing facilities from the compiler/compiler to orient the programmer to the grammar's errant activities. So in the spirit of "literate programming", "do all roads lead to TeX and its progeny"?

[2] In compiler terms the first set

5.5 Amen the closing remarks.

In closing please have a look at the attached documents to see if I might merit an encore. The attached documents are: grammar "pass3.pdf", its lr1 tables: "pass3_idx.pdf", and "o2linker.pdf": table-of-contents of all grammars. It's a new approach to using all the power of a typesetting system to generate grammar related documents. The easy part is in the generating of the file for typesetting; the harder part is in the typesetting of it.

Enjoyment from reading these documents tells me that this is a better way to communicating information to the human. So how do I integrate audio into "literate programming"? Just kidding but the remark holds on blending in as many sensory media as possible to entertain, teach, and document software. Though my project has not yet been released publicly, this is my slanted opinion from experience. All this said my goal was use "literate programming" effectively towards all parts of my project such that one could navigate intelligently the outcomes through documentation and not through scraps of code! So what is your take after viewing the sampled documents attached to this article? Is it 2 thumbs up or throw me to the TeX lions?

References

[1] http://en.wikipedia.org/wiki/
Literate_programming.

[2] http://en.wikipedia.org/wiki/POSIX_Threads.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design.* Addison-Wesley Publishing Company, 1977.

[4] Hans Christian Andersen. *New Fairy Tales.* 1844.

[5] John D. Hobby. *A User's Manual for MetaPost.* AT&T Bell Laboratories, Murray Hill, NJ 07974.

[6] Guy Steele James Gosling, Bill Joy. *The JAVA(TM) Language Specification.* Addison-Wesley Publishing Company, 1996.

[7] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report, Second Edition.* Springer-Verlag, 1974.

[8] S. C. Johnson. Yacc – yet another compiler-compiler. Technical Report 32, Murray Hill, NJ 07974, 1975.

[9] Donald E. Knuth. On the translation of languages from left to right. In *Information and Control*, volume 8 of *6*, pages 607–639, 1965.

[10] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation, Version 3.0.* Addison-Wesley Publishing Company, Reading, MA, USA, 1993.