Parsers in TEX and using CWEB for general pretty-printing

Alexander Shibakov

In this article I describe a collection of TEX macros and a few simple C programs called SPLinT that enable the use of the standard parser and scanner generator tools, bison and flex, to produce very general parsers and scanners coded as TEX macros. SPLinT is freely available from http://ctan.org/pkg/splint and http://math.tntech.edu/alex.

Introduction

The need to process formally structured languages inside TEX documents is neither new nor uncommon. Several graphics extensions for TEX (and LATEX) have introduced a variety of small specialized languages for their purposes that depend on simple (and not so simple) interpreters coded as TEX macros. A number of *pretty-printing* macros take advantage of different parsing techniques to achieve their goals (see [Go], [Do], and [Wo]).

Efforts to create general and robust parsing frameworks inside TEX go back to the origins of TEX itself. A well-known BASIC subset interpreter, BASIX (see [Gr]) was written as a demonstration of the flexibility of TEX as a programming language and a showcase of TEX's ability to handle a variety of abstract data structures. On the other hand, a relatively recent addition to the LATEX toolbox, l3regex (see [La]), provides a powerful and very general way to perform regular expression matching in LATEX, which can be used (among other things) to design parsers and scanners.

Paper [Go] contains a very good overview of several approaches to parsing and tokenizing in TEX and outlines a universal framework for parser design using TEX macros. In an earlier article (see [Wo]), Marcin Woliński describes a parser creation suite paralleling the technique used by CWEB (CWEB's 'grammar' is hard-coded into CWEAVE, whereas Woliński's approach is more general). One commonality between these two methods is a highly customized tokenizer (or scanner) used as the input to the parser proper. Woliński's design uses a finite automaton as the scanner engine with a 'manually' designed set of states. No backing up mechanism was provided, so matching, say, the longest input would require some custom coding (it is, perhaps, worth mentioning here that a backup mechanism is all one needs to turn any regular language *scanner* into a general CWEB-type parser). The scanner in [Go] was designed mainly with efficiency in mind and thus relies on a number of very clever techniques that are highly language-specific (out of necessity).

Since TEX is a system well-suited for typesetting technical documents, *pretty-printing* texts written in formal languages is a common task and is also one of the primary reasons to consider a parser written in TEX.

The author's initial motivation for writing the software described in this article grew out of the desire to fully document a few embedded microcontroller projects that contain a mix of C code, Makefiles, linker scripts, etc. While the majority of code for such projects is written in C, superbly processed by CWEB itself, some crucial information resides in the kinds of files mentioned above and can only be handled by CWEB's verbatim output (with some minimal postprocessing, mainly to remove the #line directives left by CTANGLE).

Parsing with TEX vs. others

Naturally, using TEX in isolation is not the only way to produce pretty-printed output. The CWEB system for writing structured documentation uses TEX merely as a typesetting engine, while handing over the job of parsing and preprocessing the user's input to a program built specifically for that purpose. Sometimes, however, a paper or a book written in TEX contains a few short examples of programs written in another programming language. Using a system such as CWEB to process these fragments is certainly possible (although it may become somewhat involved) but a more natural approach would be to create a parser that can process such texts (with some minimal help from the user) entirely inside TEX itself. As an example, pascal (see [Go]) was created to pretty-print Pascal programs using TEX. It used a custom scanner for a subset of standard Pascal and a parser, generated from an LL(1) Pascal grammar by a parser generator, called parTEX.

Even if CWEB or a similar tool is used, there may still be a need to parse a formal language inside TEX. One example would be the use of CWEB to handle a language other than C.

Before I proceed with the description of the tool that is the main subject of this paper, allow me to pause for just a few moments to discuss the *wisdom* (or *lack* thereof) of laying the task of parsing formal texts entirely on TEX's shoulders. In addition to using an external program to preprocess a TEX document, some recent developments allow one to implement a parser in a language 'meant for such tasks' inside an extension of TEX. We are speaking of course about LuaTEX (see [Ha]) that essentially implements an entirely separate interface to TEX's typesetting mechanisms and data structures in Lua (see [Lu]), 'grafted' onto a TEX extension.

Although I feel nothing but admiration for the LuaTEX developers, and completely share their desire to empower TEX by providing a general purpose programming language on top of its internal mechanisms, I would like to present three reasons to avoid taking advantage of LuaTEX's impressive capabilities for this particular task.

First, I am unaware of any standard tools for generating parsers and scanners in Lua (of course, it would be just as easy to use the approach described here to create such tools). At this point in time, it is just as easy to coax standard parser generators into outputting parsers in TEX as it is to make them output Lua.

Second, I am a great believer in generating 'archival quality' documents: standard TEX has been around for almost three decades in a practically unchanged form, an eternity in the software world. The parser produced using the methods outlined in this paper uses standard (plain) TEX exclusively. Moreover, if the grammar is unchanged, the parser code itself (i.e. its semantic actions) is very readable, and can be easily modified without going through the whole pipeline (`bison`, `flex`, etc.) again. A full record of the grammar is left with the generated parser and scanner so even if the more 'volatile' tools, such as `bison` and `flex`, become incompatible with the package, the parser can still be utilized with TEX alone. Perhaps the following quote by D. Knuth (see [DEK2]) would help to reinforce this point of view: "*Of course I do not claim to have found the best solution to every problem. I simply claim that it is a great advantage to have a fixed point as a building block.*"

Finally, the idea that TEX is somehow unsuitable for such tasks may have been overstated. While it is true that TEX's macro language lacks some of the expressive ability of its 'general purpose' brethren, it does possess a few qualities that make it quite adept at processing text (it is a typesetting language after all!). Among these features are: a built-in hashing mechanism (accessible through `\csname...\endcsname` and `\string` primitives) for storing and accessing control sequence names and creating associative arrays, a number of tools and data structures for comparing and manipulating strings (token registers, the `\ifx` primitive, various expansion primitives: `\edef`, `\expandafter` and the like), and even string matching and replacement (using delimited parameters in macros). TEX notoriously lacks a good (i.e. efficient *and* easy to use) framework for storing and manipulating *arrays* and *lists* (see the discussion of list macros in Appendix D of *The TEXbook* and in [Gr]) but this limitation is readily overcome by putting some extra care into one's macros.

Languages, grammars, parsers, and TEX

Or . . .

Tokens and tables keep macros in check.
Make 'em with `bison`*, use* `WEAVE` *as a tool.*
Add TEX and `CTANGLE`*, and* C *to the pool.*
Reduce 'em with actions, look forward, not back.
Macros, productions, recursion and stack!

Computer generated (most likely)

The goal of the software described in this article, `SPLinT` (Simple Parsing and Lexing in TEX, or, in the tradition of GNU, `SPLinT` Parses Languages in TEX) is to give a macro writer the ability to use standard parser/scanner generator tools to produce TEX macros capable of parsing formal languages.

Let me begin by presenting a 'bird's eye view' of the setup and the workflow one would follow to create a new parser with this package. To take full advantage of this software, two external programs (three if one counts a C compiler) are required: `bison` and `flex` (see [Bi] and [Pa]), the parser and scanner generators, respectively. Both are freely available under the terms of the General Public License version 3 or higher and are standard tools included in practically every modern GNU/Linux distribution. Versions that run under a number of other operating systems exist as well.

While the software allows the creation of both parsers and scanners in TEX, the steps involved in making a scanner are very similar to those required to generate a parser, so only the parser generation will be described below.

Setting the semantic actions aside for the moment, one begins by preparing a generic `bison` input file, following some simple guidelines. Not all `bison` options are supported (the most glaring omission is the ability to generate a general LR (glr) parser but this may be added in the future) but in every other respect it is an ordinary `bison` grammar. In some cases, a `bison` grammar may already exist and can be turned into a TEX parser with just a few (or none!) modifications and a new set of semantic actions (written in TEX of course). As a matter of example, the grammar used to pretty-print `bison` grammars in `CWEB` that comes with this package was adopted (with very minor modifications, mainly to create a more logical presentation in `CWEB`) from the original grammar used by `bison` itself.

Once the grammar has been debugged (using a combination of bison's own impressive debugging facilities and the debugging features supported by the macros in the package), it is time to write the semantic actions for the *syntax-directed translation* (see [Ah]). These are ordinary TEX macros written using a few simple conventions listed below. First, the actions themselves will be executed inside a large `\ifcase` statement (this is not *always* the case, see the discussion of 'optimization' below, but it would be better to assume that it is); thus, care must be taken to write the macros so that they can be 'skipped' by TEX's scanning mechanism. Second, instead of using bison's $\$n$ syntax to access the value stack, a variety of `\yy`*p* macros are provided. Finally, the 'driver' (a small C program, see below) provided with the package merely cycles through the actions to output TEX macros, so one has to use one of the C macros provided with the package to output TEX in a proper form. One such macro is `TeX_`, used as `TeX_("{`TEX tokens`}");`.

The next step is the most technical, and the one most in need of automation. A `Makefile` provided with the package shows how such automation can be achieved. The newly generated parser (the '`.c`-file' produced by bison) is `#include`'d in (yes, included, not merely linked to) a special 'driver' file. No modifications to the driver file or the bison produced parser are necessary; all one has to do is call a C compiler with an appropriately defined macro (see the `Makefile` for details). The resulting executable is then run which produces a `.tex` file that contains the macros necessary to use the freshly-minted parser in TEX. This short brush with a C compiler is the only time one ventures outside of the world of pure TEX to build a parser with this software (not counting the one needed to create the accompanying scanner if one is desired). It is possible to add a 'plugin' to bison to create a 'TEX output mode' but at the moment the 'lazy' version seems to be sufficient.

Now `\input` this file into your TEX document along with the macros that come with the package and voilà! You have a brand new parser in TEX! A full-featured parser for the bison input file format is included, and can be used as a template. For smaller projects, it might help to take a look at the examples portion of the package.

The discussion above glosses over a few important details that anybody who has experience writing 'ordinary' (i.e. non-TEX) parsers in bison would be eager to find out. Let us now discuss some of these details.

Data structures for parsing

A surprisingly modest amount of machinery is required to make a bison-generated parser 'tick'. In addition to the standard arithmetic 'bag of tricks' (integer addition, multiplication and conditionals), some basic integer and string array (or closely related list and stack) manipulation is all that is needed.

Parser tables and stack access 'in the raw' are normally hidden from the parser designer but creating lists and arrays is standard fare for most semantic actions. The bison parser supplied with the package does not use any techniques that are more sophisticated than simple token register operations. Representing and accessing arrays this way (see Appendix D of *The TEXbook* or the `\concat` macro in the package) is simple and intuitive but computationally expensive. The computational costs are not prohibitive though, as long as the arrays are kept short. In the case of large arrays that are read often, it pays to use a different mechanism. One such technique (used also in [Go], [Gr], and [Wo]) is to 'split' the array into a number of control sequences (creating an *associative array* of token sequences called, for example `\array[`n`]`, where n is an index value). This approach is used with the parser and scanner tables (which tend to be quite large) when the parser is 'optimized' (more about this later). Once again, it is possible to write the parser semantic actions without this (slightly unintuitive and cumbersome to implement) machinery.

This covers most of the routine computations inside semantic actions; all that is left is a way to 'tap' into the stack automaton built by bison using an interface similar to the special $\$n$ variables utilized by the 'genuine' bison parsers (i.e. written in C or any other target language supported by bison).

This role is played by the several varieties of `\yy`*p* command sequences (for the sake of completeness, p stands for one of `(`n`)`, `[`name`]`, `]`name`[` or n; here n is a string of digits; and a 'name' is any name acceptable as a symbolic name for the term in bison). Instead of going into the minutiae of various flavors of `\yy`-macros, let me just mention that one can get by with only two 'idioms' and still be able to write parsers of arbitrary sophistication: `\yy(`n`)` can be treated as a token register containing the value of the n-th term of the rule's right hand side, $n > 0$. The left hand side of a production is accessed through `\yyval`. A convenient shortcut is `\yy0{`⟨*TEX material*⟩`}` which will expand the ⟨*TEX material*⟩ inside the braces. Thus, a simple way to concatenate the values of the first two production terms is `\yy0{\the\yy(1)\the\yy(2)}`. The included `bison` parser can also be used to provide support for 'symbolic names', analogous to `bison`'s `$[name]` syntax but this requires a bit more effort on the user's part in order to initialize such support. It could make the parser more readable and maintainable, however.

Naturally, a parser writer may need a number of other data abstractions to complete the task. Since these are highly dependent on the nature of the processing the parser is supposed to provide, we refer the interested reader to the parsers included in the package as a source of examples of such specialized data structures.

Pretty-printing support with formatting hints

The scanner 'engine' is propelled by the same set of data structures and operations that drive the parser automaton: stacks, lists and the like. Table manipulation happens 'behind the scenes' just as in the case of the parser. There is also a stack of 'states' (more properly called *subautomata*) that is manipulated by the user directly, where the access to the stack is coded as a set of macros very similar to the corresponding C functions in the 'real' `flex` scanners. The 'handoff' from the scanner to the parser is implemented through a pair of registers: `\yylval`, a token register containing the value of the returned token and `\yychar`, a `\count` register that contains the numerical value of the token to be returned.

Upon matching a token, the scanner passes one crucial piece of information to its user: the character sequence representing the token just matched (`\yytext`). This is not the whole story, though: three more token sequences are made available to the parser writer whenever a token is matched.

The first of these is simply a 'normalized' version of `\yytext` (called `\yytextpure`). In most cases it is a sequence of TeX tokens with the same character codes as the one in `\yytext` but with their category codes set to 11. In cases when the tokens in `\yytext` are *not* (character code, category code) pairs, a few simple conventions are followed, explained elsewhere. This sequence is provided merely for convenience and its typical use is to generate a key for an associative array.

The other two sequences are special 'stream pointers' that provide access to the extended scanner mechanism in order to implement passing of 'formatting hints' to the parser without introducing any changes to the original grammar, as explained below.

Unlike strict parsers employed by most compilers, a parser designed for pretty-printing cannot afford being too picky about the structure of its input ([Go] calls such parsers 'loose'). As a way of simple illustration, an isolated identifier, such as '`lg_integer`' can be a type name, a variable name, or a structure tag (in a language like C for example). If one expects the pretty-printer to typeset this identifier in a correct style, some context must be supplied, as well. There are several strategies a pretty-printer can employ to get hold of the necessary context. Perhaps the simplest way to handle this, and to reduce the complexity of the pretty-printing algorithm, is to insist on the user providing enough context for the parser to do its job. For short examples like the one above, this is an acceptable strategy. Unfortunately, it is easy to come up with longer snippets of grammatically deficient text that a pretty-printer should be expected to handle. Some pretty-printers, such as the one employed by `CWEB` and its ilk (`WEB`, `FWEB`), use a very flexible bottom-up technique that tries to make sense of as large a portion of the text as it can before outputting the result (see also [Wo], which implements a similar algorithm in LaTeX).

The expectation is that this algorithm will handle the majority of the cases with the remaining few left for the author to correct. The question is, how can such a correction be applied?

`CWEB` itself provides two rather different mechanisms for handling these exceptions. The first uses direct typesetting commands (for example, `@+` and `@*` for cancelling and introducing a line break, resp.) to change the typographic output.

The second (preferred) way is to supply *hidden context* to the pretty-printer. Two commands, `@;` and `@[...@]` are used for this purpose. The former introduces a 'virtual semicolon' that acts in every way like a real one except it is not typeset (it is not output in the source file generated by `CTANGLE`, either but this has nothing to do with pretty-printing, so I will not mention `CTANGLE` anymore). For instance, from the parser's point of view, if the preceding text was parsed as a 'scrap' of type *exp*, the addition of `@;` will make it into a 'scrap' of type *stmt* in `CWEB`'s parlance. The latter construct (`@[...@]`), is used to create an *exp* scrap out of whatever happens to be inside the brackets.

This is a powerful tool at one's disposal. Stylistically, this is the right way to handle exceptions as it forces the writer to emphasize the *logical* structure of the formal text. If the pretty-printing style is changed extensively later, the texts with such hidden contexts should be able to survive intact in the final document (as an example, using a break after every statement in C may no longer be considered appropriate, so any forced break introduced to support this convention would now have to be removed, whereas `@;`'s would simply quietly disappear into the background).

The same hidden context idea has another important advantage: with careful grammar fragmenting (facilitated by `CWEB`'s or any other literate programming tool's 'hypertext' structure) and a more diverse hidden context (or even arbitrary hidden text) mechanism, it is possible to use a strict parser to parse incomplete language fragments. For example, the productions that are needed to parse C's expressions form a complete subset of the parser. If the grammar's 'start' symbol is changed to *expression* (instead of the *translation-unit* as it is in the full C grammar), a variety of incomplete C fragments can now be parsed and pretty-printed. Whenever such granularity is still too 'coarse', carefully supplied hidden context will give the pretty-printer enough information to adequately process each fragment. A number of such *sub*-parsers can be tried on each fragment (this may sound computationally expensive, however, in practice, a carefully chosen hierarchy of parsers will finish the job rather quickly) until a correct parser produced the desired output.

This somewhat lengthy discussion brings us to the question directly related to the tools described in this article: how does one provide typographical hints or hidden context to the parser?

One obvious solution is to build such hints directly into the grammar. The parser designer can, for instance, add new tokens (terminals, say, `BREAK_LINE`) to the grammar and extend the production set to incorporate the new additions. The risk of introducing new conflicts into the grammar is low (although not entirely non-existent, due to the lookahead limitations of LR(1) grammars) and the changes required are easy, although very tedious, to incorporate.

In addition to being labor intensive, this solution has two other significant shortcomings: it alters the original grammar and hides its logical structure, and it 'bakes in' the pretty-printing conventions into the language structure (making 'hidden' context much less 'stealthy').

A much better approach involves inserting the hints at the lexing stage and passing this information to the parser as part of the token 'values'. The hints themselves can masquerade as characters ignored by the scanner (white space, for example) and preprocessed by a specially designed input routine. The scanner then simply passes on the values to the parser.

The difficulty lies in synchronizing the token production with the parser. This subtle complication is very familiar to anyone who has designed TeX's output routines: the parser and the lexer are not synchronous, in the sense that the scanner might be reading several (in the case of the general LR(n) parsers) tokens ahead of the parser before deciding on how to proceed (the same way TeX can consume a whole paragraph's worth of text before exercising its page builder).

If we simple-mindedly let the scanner return every hint it has encountered so far, we may end up feeding the parser the hints meant for the token that appears *after* the fragment the parser is currently working on. In other words, when the scanner 'backs up' it must correctly back up the hints as well.

This is exactly what the scanner produced by the tools in this package does: along with the main stream of tokens meant for the parser, it produces two hidden streams (called the `\format` stream and the `\stash` stream) and provides the parser with two strings (currently only strings of digits are used although arbitrary sequences of TeX tokens can be used as pointers) with the promise that *all the 'hints' between the beginning of the corresponding stream and the point labelled by the current stream pointer appeared among the characters up to and, possibly, including the ones matched as the current token.* The macros to extract the relevant parts of the streams (`\yyreadfifo` and its cousins) are provided for the convenience of the parser designer. The interested reader can consult the input routine macros for the details of the internal representation of the streams.

In the interest of full disclosure, let me point out that this simple technique introduces a significant strain on TeX's computational resources: the lowest level macros, the ones that handle character input and are thus executed (sometimes multiple times), for *every* character in the input stream are rather complicated and therefore, slow. Whenever the use of such streams is not desired a simpler input routine can be written to speed up the process (see `\yyinputtrivial` for a working example of such macro).

The parser function

To achieve such a tight integration with `bison`, its parser template, `yyparse()` was simply translated into TeX using the following well known method.

Given the code (where `goto`'s are the only means of branching but can appear inside conditionals):

```
label A: ...
        [more code ...]
            goto C;
        [more code ...]
label B: ...
        [more code ...]
            goto A;
        [more code ...]
label C: ...
        [more code ...]
```

one way to translate it into TeX is to define a set of macros (call them `\labelA`, `\labelAtail` and so forth for clarity) that end in `\next` (a common name for this purpose). Now, `\labelA` will implement the code that comes between `label A:` and `goto C;`, whereas `\labelAtail` is responsible for the code after `goto C;` and before `label B:` (provided no other `goto`'s intervene which can always be arranged). The conditional preceding `goto C;` can now be written in TeX as

```
\if⟨condition⟩
    \let\next=\labelC
\else
    \let\next=\labelAtail
```

where ⟨condition⟩ is an appropriate translation of the corresponding condition in the code being translated (usually, one of '$=$' or '$\neq$'). Further details can be extracted from the TeX code that implements these functions where the corresponding C code is presented alongside the macros that mimic its functionality.

Debugging

If the tools in the package are used to create medium to high complexity parsers, the question of debugging will come up sooner or later. The grammar design stage of this process can utilize all the excellent debugging facilities provided by `bison` and `flex` (reporting of conflicts, output of the automaton, etc.). The `Makefile`s supplied with the package will automatically output all the debugging information the corresponding tool can provide. Eventually, when all the conflicts are ironed out and the parser begins to process input without performing any actions, it becomes important to have a way to see 'inside' the parsing process. Since the processing performed by the generated parser is done in several stages, the debugging may become rather involved.

All the debugging features are activated by using various `\iftrace`... conditionals, as well as `\ifyyinputdebug` and `\ifyyflexdebug` (for example, to look at the parser stack, one would set `\tracestackstrue`). When all of the conditionals are activated, *a lot* of output is produced. At this point it is important to narrow down the problem area and only activate the debugging features relevant to any errant behaviour exhibited by the parser. Most of the debugging features built into ordinary `bison` parsers (and `flex` scanners) are available.

In general, debugging parsers and scanners (and debugging in general) is a very deep topic that may require a separate paper (or maybe a book!) all by itself, so I will simply leave it here and encourage the reader to experiment with the included parsers to learn the general operational principles behind the parsing automaton. One needs to be aware that, unlike the 'real' C parsers, the TeX parser has to deal with more than simply straight text. So if it looks like the parser (or the scanner) absolutely has to accept the (rejected) input displayed on the screen, just remember that an 'a' with a category code 11 and an 'a' with a category code 12 look the same on the terminal while TeX and the parser/scanner may treat them as completely different characters (this behavior itself can be fine tuned by changing `\yyinput`).

Speeding up the parser

By default, the generated parser and scanner keep all of their tables in separate token registers. Each stack is kept in a single macro. Thus, every time a table is accessed, it has to be expanded making the table access latency linear in *the size of the table*. The same holds for stacks and the action 'switches', of course. While keeping the parser tables (that are constant) in token registers does not have any better rationale than saving control sequence memory (the most abundant memory in TeX), this way of storing *stacks* does have an advantage when multiple parsers come into play simultaneously. All one has to do to switch from one parser to another is to save the state by renaming the stack control sequences accordingly.

When the parser and scanner are 'optimized' (by saying `\def\optimization{5}`, for example), all these control sequences are 'spread over' the appropriate associative arrays (by creating a number of control sequences that look like `\array[`n`]`, where *n* is the index, as explained above). While it is certainly possible to optimize only some of the parsers (if your document uses multiple) or even only some *parts* of a given parser (or scanner), the details of how to do this are rather technical and are left for the reader to discover by reading the examples supplied with the package. At least at the beginning it is easier to simply set the highest optimization level and use it consistently throughout the document.

Use with CWEB

Since the macros in the package were designed to support pretty-printing of languages other than C in CWEB it makes sense to spend a few paragraphs on this particular application. The CWEB system consists of two weakly related programs: CWEAVE and CTANGLE. The latter extracts the C portion of the users input, and outputs a C file after an appropriate rearrangement of the various sections of the code. The task of CWEAVE is very different and arguably more complex: not only does it have to be aware of the general 'hierarchy' of various subsections of the program to create cross references, an index, etc., it also has to understand enough of the C code in order to pretty-print it. Whereas CTANGLE simply separates the program code from the programmer's documentation, rearranges it and outputs the original program text (with added #line directives and simple C comments that can be easily removed in postprocessing if necessary), the output of CWEAVE bears very little resemblance to the original program. It might sound a bit exaggerated but CWEAVE's processing is 'one-way': it would be difficult or even impossible to write software that 'undoes' the pretty-printing performed by CWEAVE.

There is, however, a loophole that allows one to use CWEB with practically any language, *and* pretty-print the results, if an appropriate 'filter' is available. The saving grace comes in the form of CWEB's *verbatim output*: any text inside @= and @> undergoes some minimal processing (mainly to 'escape' dangerous TEX characters such as '$') and is put inside \vb{...} by CWEAVE.

The macros in the package take advantage of this feature by collecting all the text inside \vb groups and trying to parse it. If the parsing pass is successful, pretty-printed output is produced, if not, the text is output in 'typewriter' style.

With languages such as bison's input script, an additional complication has to be dealt with: most of the time the actions are written in C so it makes sense to use CWEAVE's C pretty-printer to typeset the action code. Most material outside of \vb groups is therefore assumed to be C code and is carefully collected and 'cleaned up' by the macros included in the package.

For the purposes of documenting the TEX parser, one additional feature of CWEAVE is taken advantage of: the text inside double quotes, "..." is treated similarly to the verbatim portion of the input (this can be viewed as a 'local' version of the verbatim sections). Moreover, CWEAVE allows one to treat a function name (or nearly any identifier) as a TEX macro. These two features are enough to implement pretty-printing of semantic actions in TEX. The macros will turn an input string such as, e.g. 'TeX_("\\relax");' into '∘' (for the sake of convenience, the string above would actually be written as 'TeX_("/relax");' as explained in the manual for the package). See the documentation that comes with the package and the bison language pretty-printer implementation for any additional details.

An example

As an example, let us walk through the development process of a simple parser. Since the language itself is not of any particular importance, a simple grammar for additive expressions was chosen. The example, with a detailed description, and all the necessary files, is included in the examples directory. The Makefile there allows one to type, say, make step1 to produce all the files needed in the first step of this short walk-through. Finally, make docs will produce a pretty-printed version of the grammar, the regular expressions, and the test TEX file along with detailed explanations of every stage.

As the first step, one creates a bison input file (expp.y) and a similar input for flex (expl.l). A typical fragment of expp.y looks like the following:

```
value:
  expression {TeX_("/yy0{/the/yy(1)}");}
;
```

The scanner's regular expression section, in its entirety is:

```
[ \f\n\t\v]  {TeX_("/yylexnext");}
{id}         {
    TeX_("/yylexreturnval{IDENTIFIER}");}
{int}        {
    TeX_("/yylexreturnval{INTEGER}");}
[+*()]       {TeX_("/yylexreturnchar");}
.            {
TeX_("/iftracebadchars");
TeX_("    /yycomplain{%%");
TeX_("      invalid character(s): %%");
TeX_("      /the/yytext}");
TeX_("/fi");
TeX_("/yylexreturn{$undefined}");
}
```

Once the files have been prepared and debugged, the next step is to generate the 'driver' files, `ptabout` and `ltabout`. For the parser 'driver', this is done with

```
bison expp.y -o expp.c
gcc -DPARSER_FILE=\
    \"examples/expression/expp.c\" \
    -o ptabout ../../mkeparser.c
```

The first line generates the parser from the `bison` input file that was prepared in the first step and the next line uses this file to produce a 'driver'. If the included `Makefile` is used, the file `mkeparser.c` is generated automatically, otherwise one has to make sure that it exists and resides in the appropriate directory first. It has no external dependencies and can be freely moved to any place that is convenient.

Next, run `ptabout` and `ltabout` to produce the automata tables:

```
ptabout --optimize-actions ptab.tex
ltabout --optimize-actions ltab.tex
```

Now, look inside `expression.sty` for a way to include the parser in your own documents, or simply `\input` it in your own TeX file. Executing `make test.tex` will produce a test file for the new parser. This is it!

Acknowledgment

The author would like to thank the editors, Barbara Beeton and Karl Berry, for a number of valuable suggestions and improvements to this article.

References

[Ah] Alfred V. Aho et al., *Compilers: Principles, Techniques, and Tools*, Pearson Education, 2006.

[Bi] Charles Donnelly and Richard Stallman, *Bison, The Yacc-compatible Parser Generator*, The Free Software Foundation, 2013. `http://www.gnu.org/software/bison/`

[DEK1] Donald E. Knuth, *The TeXbook*, Addison-Wesley Reading, Massachusetts, 1984.

[DEK2] Donald E. Knuth *The future of TeX and METAFONT*, *TUGboat* **11** (4), p. 489, 1990. `http://tug.org/TUGboat/tb11-4/tb30futuretex.pdf`

[Do] Jean-luc Doumont, *Pascal pretty-printing: An example of "preprocessing with TeX"*, *TUGboat* **15** (3), 1994 — Proceedings of the 1994 TUG Annual Meeting. `http://tug.org/TUGboat/tb15-3/tb44doumont.pdf`

[Er] Sebastian Thore Erdweg and Klaus Ostermann, *Featherweight TeX and Parser Correctness*, Proceedings of the Third International Conference on Software Language Engineering, pp. 397–416, Springer-Verlag Berlin, Heidelberg, 2011.

[Fi] Jonathan Fine, *The* `\CASE` *and* `\FIND` *macros*, TUGboat **14** (1), pp. 35–39, 1993. `http://tug.org/TUGboat/tb14-1/tb38fine.pdf`

[Go] Pedro Palao Gostanza, *Fast scanners and self-parsing in TeX*, *TUGboat* **21** (3), 2000 — Proceedings of the 2000 Annual Meeting. `http://tug.org/TUGboat/tb21-3/tb68gost.pdf`

[Gr] Andrew Marc Greene, *BASIX — An interpreter written in TeX*, *TUGboat* **11** (3), 1990 — Proceedings of the 1990 TUG Annual Meeting. `http://tug.org/TUGboat/tb11-3/tb29greene.pdf`

[Ha] Hans Hagen, *LuaTeX: Halfway to version 1*, *TUGboat* **30** (2), pp. 183–186, 2009. `http://tug.org/TUGboat/tb30-2/tb95hagen-luatex.pdf`

[Ie] R. Ierusalimschy et al., *Lua 5.1 Reference Manual*, `Lua.org`, August 2006. `http://www.lua.org/manual/5.1/`

[La] *The* `l3regex` *package: Regular expressions in TeX*, The LaTeX3 Project. `http://www.ctan.org/pkg/l3regex`

[Pa] Vern Paxson et al., *Lexical Analysis With Flex, for Flex 2.5.37*, July 2012. `http://flex.sourceforge.net/manual/`

[Wo] Marcin Woliński, `Pretprin` — *A LaTeX2ε package for pretty-printing texts in formal languages*, *TUGboat* **19** (3), 1998 — Proceedings of the 1998 TUG Annual Meeting. `http://tug.org/TUGboat/tb19-3/tb60wolin.pdf`

◇ Alexander Shibakov
Dept. of Mathematics
Tennessee Tech. University
Cookeville, TN
`http://math.tntech.edu/alex`