SPLINT
reference

version 1.03

SPLINT

Alex Shibakov

July 21, 2014

1 Introduction

SPLinT [1]) (Simple Parsing and Lexing in TEX, or, following the great GNU tradition of creating recursive names, SPLinT Parses Languages in TEX) is a system (or rather a mélange of systems) designed to facilitate developing parsing macros in TEX and (to a lesser degree) documenting parsers written in other languages. As an application, a parser for bison input file syntax has been developed, along with a macro collection that makes it possible to design and pretty print bison grammars using CWEB.

Developing software in CWEB involves two programs. The first of these is CTANGLE that outputs the actual code, intended to be in C. In reality, CTANGLE cares very little about the language it produces. Exceptions are C comments and **#line** directives that might confuse lesser software, although bison is all too happy to swallow them (there are also some C specific constructs that CTANGLE tries to recognize). CTANGLE's main function is to rearrange the text of the program as written by the programmer (in a way that, hopefully, emphasizes the internal logic of the code) into an appropriate sequence (e.g. all variable declaration must textually precede their use). All that is required to adopt CTANGLE to produce bison output is some very rudimentary post- and pre-processing.

Our main concern is thus CWEAVE that not only pretty prints the program but also creates an index, cross-references all the sections, etc. Getting CWEAVE to pretty print a language other than C requires some additional attention. A true digital warrior would probably try to decipher CWEAVE's output 'in the raw' but, alas, my WebFu is not that strong. The loophole comes in the form of a rarely (for a good reason) used CWEB command: the verbatim (@=...@>) output. The material to be output by this construct undergoes minimal processing and is put inside \vb{...}. All that is needed now is a way to process this virtually straight text inside TEX.

2 Using the bison parser

The process of using SPLinT for writing parsing macros in TEX is treated in considerable detail later in this document. We begin, instead, by outlining how one such parser can be used to pretty print a bison grammar. Following the convention mentioned above and putting all non-C code inside CWEAVE's verbatim blocks, consider the following (meaningless) code fragment. The fragment contains a mixture of C and bison code, the former appears outside of the verbatim blocks.

```
@= non_terminal:                              @>
@=   term.1 term.2       {@> a = b;    @=}@>
@= | term.3 other_term   {@> $$ = $1;  @=}@>
@= | still more terms    {@> f($1);    @=}@>
@= ;                                          @>
```

The fragment above will appear as (the output of CTANGLE can be examined in sill.y)

⟨ A silly example 2 ⟩ =

non_terminal :

$term_1$ $term_2$	$a \Leftarrow b;$
$term_3$ *other_term*	$\Upsilon \Leftarrow \Upsilon_1;$
still more terms	$f(\Upsilon_1);$

See also sections 3, 5, and 8.

This code is used in section 11.

3 ... if the syntax is correct. In case it is a bit off, the parser will give up and you will see a different result. The code in the fragment below is easily recognizable, and some parts of it (all of C code, in fact) are still pretty printed in CWEAVE. Only the verbatim portion is left unprocessed.

⟨ A silly example 2 ⟩ +=

```
whoops
  term.1 term.2          { a ⇐ b; }
| term.3 other_term      { Υ ⇐ Υ1; }
```

[1]) I was tempted to call the package ParLALRgram which stands for Parsing LALR Grammars or PinT for 'Parsing in TEX' but both sounded too generic.

```
| still more terms      { f(ϒ1); }
;
```

4 The TEX header that makes such output possible is quite plain. In this example (i.e. this very file) it consists of

```
\input limbo.sty
\input frontmatter.sty
\input yy.sty
```
[more code . . .]

The first two lines are presented here merely for completeness: there is no parsing-relevant code in them. The line that follows loads the macros that implement the parsing and scanning machinery. This is enough to set up all the basic mechanisms used by the parsing and lexing macros. The rest of the header provides a few definitions to fine tune the typesetting of grammar productions. It starts with

```
\let\currentparsernamespace\parsernamespace
    \let\parsernamespace\mainnamespace
    \let\currenttokeneq\tokeneq
        \def\tokeneq#1#2{\prettytoken{#1}}
            \input bo.tok % re-use token equivalence table to set the
    \let\tokeneq\currenttokeneq
    \input btokenset.sty
```
[more code . . .]

We will have a chance to discuss all the `\...namespace` macros later, at this point it will suffice to say that the lines above are responsible for controlling the typesetting of term names. The file `bo.tok` consists of a number of lines like the ones below:

```
\tokeneq {STRING}{{34}{115}{116}{114}{105}{110}{103}{34}}
\tokeneq {PERCENT_TOKEN}{{34}{37}{116}{111}{107}{101}{110}{34}}
```
[more code . . .]

The cryptic looking sequences of integers above are strings of ASCII codes of the letters that form the name `bison` uses when it needs to refer to the corresponding token (thus, the second one is `"%token"` which might help explain why such an elaborate scheme has been chosen). The macro `\tokeneq` is defined in `yymisc.sty`, which in turn is input by `yy.sty` but what about the token names themselves? In this case they were extracted automatically from the `CWEB` source file by the parser during the `CWEAVE` processing stage. All of these definitions can be overwritten to get the desired output (say, one might want to typeset `ID` in a roman font, as 'identifier'; all that needs to be done is a macro that says `\prettywordpair{ID}{{\rm identifier}}`). The file `btokenset.sty` input above contains a number of such definitions.

5 To round off this short overview, I must mention a caveat associated with using the macros in this collection: while one of the greatest advantages of using `CWEB` is its ability to rearrange the code in a very flexible way, the parser will either give up or produce unintended output if this feature is abused while describing the grammar. For example, in the code below

⟨ A silly example 2 ⟩ +=
next_term :
stuff ⟨ Rest of line 7 ⟩$a \Leftarrow f(x)$;
⟨ A production 6 ⟩

6 the line titled ⟨ A production 6 ⟩ is intended to be a rule defined later. Notice that while it seems that the parser was able to recognize the first code fragment as a valid `bison` input, it misplaced the ⟨ Rest of line 7 ⟩, having erroneously assumed it to be a part of the action code for this grammar (later on we will go into the details of why it is necessary to collect all the non-verbatim output of `CWEAVE`, even the one that contains no

interesting C code; hint: it has something to do with money (`$`), also known as math and the way `CWEAVE` processes the 'gaps' between verbatim sections). The production line that follows did not fare as well: the parser gave up. There is simply no point in including such a small language fragment as a valid input for the grammar the parser uses to process the verbatim output.

⟨ A production 6 ⟩ =
    `more stuff in this line {`$b \Leftarrow g(y);$`}`

See also section 9.

This code is cited in section 6.

This code is used in sections 5 and 8.

7 Finally, if you forget that only the verbatim part of the output is looked at by the parser you might get something unrecognizable, such as

⟨ Rest of line 7 ⟩ =
  but $^{\text{not}}$ *all of it*

See also section 10.

This code is cited in section 6.

This code is used in sections 5 and 8.

8 To correct this, one can provide a more complete grammar fragment to allow the parser to complete its task successfully. In some cases, this imposes too strict a constraint on the programmer. Instead, the parser that pretty prints `bison` grammars allows one to add *hidden context* to the code fragments above. The context is added inside `\vb` sections using `CWEB`'s `@t...@>` facility. The `CTANGLE` output is not affected by this while the code above can now be typeset as:

⟨ A silly example 2 ⟩ +=
  next_term :
      stuff ⟨ *Rest of line* 7 ⟩ $a \Leftarrow f(x);$
  ⟨ A production 6 ⟩

9 ... even a single line can now be displayed properly.

⟨ A production 6 ⟩ +=
      more stuff in this line $b \Leftarrow g(y);$

10 With enough hidden context, even a small rule fragment can be typeset as intended. The 'action star' was inserted to reveal some of the context.

⟨ Rest of line 7 ⟩ +=
      but not all of it ⋆

11 What makes all of this even more confusing is that `CTANGLE` will have no trouble outputting this as a(n almost, due to the intentionally bad `whoops` production above) valid `bison` file (as can be checked by looking into `sill.y`). The author happens to think that one should not fragment the software into pieces that are too small: `bison` is not C so it makes sense to write `bison` code differently. However, if the logic behind your code organization demands such fine fragmentation, hidden context provides you with a tool to show it off. A look inside the source of this document shows that adding hidden context can be a bit ugly so it is not recommended for routine use. The short example above is output in the file below.

⟨ `sill.y` 11 ⟩ =
  ⟨ A silly example 2 ⟩

12 On debugging

This concludes a short introduction to the `bison` grammar pretty printing using this macro collection. It would be incomplete, however, without any reference to debugging [1]). There is a fair amount of debugging

[1]) Here we are talking about debugging the output produced by `CWEAVE` when the included `bison` parser is used, *not* debugging parsers written with the help of this software: the latter topic is covered in more detail later on

information that the macros can output, unfortunately, very little of it is tailored to the *use* of the macros in the `bison` parser. Most of it is designed to help *build* a new parser. If you find that the parser gives up too often or even crashes (the latter is most certainly a bug in the parser itself), the first approach is to make sure that your code *compiles* i.e. forget about the printed output and try to see if the 'real' `bison` accepts the code (just the syntax, no need to worry about conflicts and such).

If this does not shed any light on why the macros seem to fail, turn on the debugging output by saying `\trace...true` for various trace macros. This can produce *a lot* of output, even for small fragments, so turn it on only for a section at a time. If you need still *more* details of the inner workings of the parser and the lexer, various other debugging conditionals are available. For example, `\yyflexdebugtrue` turns on the debugging output for the scanner. There are a number of such conditionals that are discussed in the commentary for the appropriate TEX macros.

Remember, what you are seeing at this point is the parsing process of the `bison` input file, not the one for *your* grammar (which might not even be complete at this point). However, if this fails, you are on your own: drop me a line if you figure out how to fix any bugs you find.

13 Terminology

We now list a few definitions of the concepts used repeatedly in this documentation. Most of this terminology is rather standard. Formal precision is not the goal here, and intuitive explanations are substituted whenever possible.

- **bison parser**: while, strictly speaking, not a formally defined term, this combination will always stand for one of the parsers generated by this package designed to parse a subset of the 'official' grammar for `bison` input files. All of these parsers are described later in this documentation. The term *main parser* will be used as a substitute in example documentation for the same purpose.
- **driver**: a generic but poorly defined concept. In this documentation it is used predominantly to mean both the C code and the resulting executable that outputs the TEX macros that contain the parser tables, token values, etc., for the parsers built by the user. It is understood that the C code of the 'driver' is unchanged and the information about the parser itself is obtained by *including* the C file produced by `bison` in the 'driver' (see the examples supplied with the package).
- **lexer**: a synonym for *scanner*, a subroutine that performs the *lexical analysis* phase of the parsing process, i.e. groups various characters from the input stream into parser *tokens*.
- **namespace**: this is an overused bit of terminology meaning a set of names grouped together according to some relatively well defined principle. In a language without a well developed type system (such as TEX) it is usually accompanied by a specially designed naming scheme. *Parser namespaces* are commonly used in this documentation to mean a collection of all the data structures describing a parser and its state, including tables, stacks, etc., named by using the 'root' name (say `\yytable`) and adding the name of the parser (for example, `[main]`). To support this naming scheme, a number of macros work in unison to create and rename the 'data macros' accordingly.
- **symbolic switch**: a macro (or an associative array of macros) that let the TEX parser generated by the package associate *symbolic term names* with the terms. Unlike the 'real' parser, the parser created with this suite requires some extra setup as explained in the included examples (one can also consult the source for this documentation which creates but does not use a symbolic switch).
- **symbolic term name**: a (relatively new) way to refer to stack values in `bison`. In addition to using the 'positional' names such as `$`*n* to refer to term values, one can utilize the new syntax: `$[`*name*`]`. The '*name*' can be assigned by the user or can be the name of the nonterminal or token used in the productions.
- **term**: in a narrow sense, an 'element' of a grammar. Instead of a long winded definition, an example, such as «identifier» should suffice. Terms are further classified into *terminals* (tokens) and *nonterminals* (which can be intuitively thought of as composite terms).
- **token**: in short, an element of a set. Usually encoded as an integer by most parsers, an indivisible *term* produced for the parser by the scanner. TEX's scanner uses a more sophisticated token classification, for example, (character code, character category) pairs, etc.

14 Languages, scanners, parsers, and TEX

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)

In order to understand the parsing routines in this collection, it would help to gain some familiarity with the internals of the parsers produced by `bison` for its intended target: C. A person looking inside a parser delivered by `bison` would quickly discover that the parsing procedure itself (*yyparse*) occupies a rather small portion of the file. If (s)he were to further reduce the size of the file by removing all the preprocessor directives intended to anticipate every conceivable combination of the operating system, compiler, and C dialect, and various reporting and error logging functions it would become very clear that the most valuable product of `bison`'s labor is a collection of integer *tables* that control the actions of the parser routine. Moreover, the routine itself is an extremely concise and well-structured loop composed of **goto**'s and a number of numerical conditionals. If one were to think of a way of accessing arrays and processing conditionals in the language of one's choice, once the tables produced by `bison` have been converted into a form suitable for the consumption by the appropriate language engine, the parser implementation becomes straightforward. Or nearly so.

The *scanning* (or *lexing*) step of this process—a way to convert a stream of symbols into a stream of integers, also deserves some attention here. There are a number of excellent tools written to automate this step in much the same fashion as `bison` automates the generation of parsers. One such tool, `flex`, though (in the opinion of this author) slightly lacking in the simplicity and elegance as compared to `bison`, was used to implement the lexer for this software suite. Lexing in TEX will be discussed in considerable detail later in this manual.

The language of interest in our case is, of course, TEX, so our future discussion will revolve around the five elements mentioned above: (1)data structures (mainly arrays and stacks), (2)converting `bison`'s output into a form suitable for TEX's consumption, (3)processing raw streams of TEX's tokens and converting them into streams of parser tokens, (4)the implementation of `bison`'s *yyparse* in TEX, and, finally, (5)producing TEX output via *syntax-directed translation* (which requires an appropriate abstraction to represent `bison`'s actions inside TEX). We shall begin by discussing the parsing process itself.

15 Arrays, stacks and the parser

Let us briefly examine the programming environment offered by TEX. Designed for typesetting, TEX's remarkable language provides a layer of macro processing atop of a set of commands that produce the output fulfilling its primary mission: delivering page layouts. In The TEXbook, macro *expansion* is likened to mastication, whereas TEX's main product, the typographic output is the result of its 'digestion' process. Not everything that goes through TEX's digestive tract ends up leaving a trace on the final page: a file full of `\relax`'s will produce no output, even though `\relax` is not a macro, and thus would have to be processed by TEX at the lowest level.

It is time to describe the details of defining suitable data structures in TEX. At first glance, TEX provides rather standard means of organizing and using general memory. At the core of its generic programming environment is an array of `\count` *n registers*, which may be viewed as general purpose integer variables that are randomly accessible by their indices. The integer arithmetic machinery offered by TEX is spartan but is very adequate for the sort of operations a parser would perform: mostly additions and comparisons.

Is the `\count` array a good way to store tables in TEX? Probably not. The first factor is the *size* of this array: only 256 `\count` registers exist in a standard TEX (the actual number of such registers on a typical machine running TEX is significantly higher but this author is a great believer in standards, and to his knowledge, none of the standardization efforts in the TEX world has resulted in anything even close to the definitive masterpiece that is The TEXbook). The issue of size can be mitigated to some extent by using a number of other similar arrays used by TEX (`\catcode`, `\uccode`, `\dimen`, `\sfcode` and others can be used for this purpose as long as one takes care to restore the 'sane' values before control is handed off to TEX's

typesetting mechanisms). If a table has to span several such arrays, however, the complexity of accessing code would have to increase significantly, and the issue of size would still haunt the programmer.

The second factor is the use of several registers by TEX for special purposes (in addition, some of these registers can only store a limited range of values). Thus, the first 10 `\count` registers are used by plain TEX for (well, *intended* for, anyway) the purposes of page accounting: their values would have to be carefully saved and restored before and after each parsing call, respectively. Other registers (`\catcode` in particular) have even more disrupting effects on TEX's internal mechanisms. While all of this can be managed (after all, using TEX as an arithmetic engine such as a parser suspends the need for any typographic or other specialized functions controlled by these arrays), the added complexity of using several memory banks simultaneously and the speed penalty caused by the need to store and restore register values make this approach much less attractive.

What other means of storing arrays are provided by TEX? Essentially, only three options remain: `\token` registers, macros holding whole arrays, and associative arrays accessed through `\csname`...`\endcsname`. In the first two cases if care is taken to store such arrays in an appropriate form one can use TEX's `\ifcase` primitive to access individual elements. The trade-off is the speed of such access: it is *linear* in the size of the array for most operations, and worse than that for others, such as removing the last item of an array. Using clever ways of organizing such arrays, one can improve the linear access time to $O(\log n)$ by simply modifying the access macros but at the moment, a straightforward `\ifcase` is used after expanding a list macro or the contents of a `\token` n register in an *un*optimized parser. An *optimized* parser uses associative arrays.

The array discussion above is just as applicable to *stacks* (indeed, an array is the most common form of stack implementation). Since stacks pop up and disappear frequently (what else are stacks to do?), list macros are usually used to store them. The optimized parser uses a separate `\count` register to keep track of the top of the stack in the appropriate associative array.

Let us now switch our attention to the code that implements the parser and scanner *functions*. If one has spent some time writing TEX macros of any sophistication (or any macros, for that matter) (s)he must be familiar with the general feeling of frustration and the desire to 'just call a function here and move on'. Macros produce *tokens*, however, and tokens must either expand to nothing or stay and be contributed to your input, or worse, be out of place and produce an error. One way to sustain a stream of execution with macros is *tail recursion* (i.e. always expanding the *last token left standing*).

As we have already discussed, `bison`'s *yyparse*( ) is a well laid out loop organized as a sequence of **goto**'s (no reason to become religious about structured programming here). This fact, and the following well known trick, make C to TEX translation almost straightforward.

```
label A: ...
   [more code ...]
     if(condition)
          goto C;
   [more code ...]
label B: ...
   [more code ...]
          goto A;
   [more code ...]
label C: ...
   [more code ...]
```

Given the code on the left (where **goto**'s are the only means of branching but can appear inside conditionals), 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 which precedes `goto C;` can now be written in TEX as presented on the right, where (condition) is an appropriate translation of the corresponding condition in the code being translated (usually, one of '=' or '≠'). 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 [1]). This concludes an overview of the general approach, It is time to consider the way characters get consumed on the lower levels of the macro hierarchy and the interaction between the different layers of the package.

```
\if(condition)
    \let\next=\labelC
\else
    \let\next=\labelAtail
```

[1]) Running the risk of overloading the reader with details, the author would like to note that the actual implementation follows a *slightly* different route in order to avoid any `\let` assignments or changing the meaning of `\next`

16 TEX into tokens

Thus far we have covered the ideas behind items (1) and (4) on our list. It is time to discuss the lowest level of processing done by these macros: converting TEX's tokens into the tokens consumed by the parser, i.e. part(3) of the plan. Perhaps, it would be most appropriate to begin by defining the term *token*.

As commonly defined, a token is simply an element of a set. Depending on how much structure the said set possesses, a token can be represented by an integer or a more complicated data structure. In the discussion below, we will be dealing with two kinds of tokens: the tokens consumed by the parsers and the TEX tokens seen by the input routines. The latter play the role of *characters* that combine to become the former. `bison`'s internal representation for its tokens is non-negative integers so this is what a scanner must produce.

TEX's tokens are a good deal more sophisticated: they can be either pairs $(c_{\mathrm{ch}}, c_{\mathrm{cat}})$, where c_{ch} is the character code and c_{cat} is TEX's category code (1 and 2 for group characters, 5 for end of line, etc.), or *control sequences*, such as `\relax`. Some of these tokens (control sequences and *active*, i.e. category 13 characters) can have complicated internal structure (expansion). The situation is further complicated by TEX's `\let` facility, which can create 'character-like' control sequences, and the lack of conditionals to distinguish them from the 'real' characters. Finally, not all pairs can appear as part of the input (say, there is no $(n, 0)$ token for any n, in the terminology above).

The scanner expects to see *characters* in its input, which are represented by their ASCII codes, i.e. integers between 0 and 255 (actually, a more general notion of the Unicode character is supported but we will not discuss it further). Before character codes appear as the input to the scanner, however, and make its integer table-driven mechanism 'tick', a lot of work must be done to collect and process the stream of TEX tokens produced after `CWEAVE` is done with your input. This work becomes further complicated when the typesetting routines that interpret the parser's output must sneak outside of the parsed stream of text (which is structured by the parser) and insert the original TEX code produced by `CWEAVE` into the page.

`SPLinT` comes with a customizeable input routine of moderate complexity (`\yyinput`) that classifies all TEX tokens into seven categories: 'normal' spaces (i.e. category 10 tokens, skipped by TEX's parameter scanning mechanism), 'explicit' spaces (includes the control sequences `\let` to ␣, as well as \␣), groups (*avoid* using `\bgroup` and `\egroup` in your input but 'real', `{...}` groups are fine), active characters, normal characters (of all character categories that can appear in TEX input, including `$`, `^`, `#`, `a`–`Z`, etc.), single letter control sequences, and multi-letter control sequences. Each of these categories can be processed separately to 'fine-tune' the input routine to the problem at hand. The input routine is not very fast, instead, flexibility was the main goal. Therefore, if speed is desirable, a customized input routine is a great place to start. As an example, a minimalistic `\yyinputtrivial` macro is included.

When `\yyinput` 'returns' by calling `\yyreturn` (which is a macro you design), your lexing routines have access to three registers: `\yycp@`, that holds the character value of the character just consumed by `\yyinput`, `\yybyte`, that most of the time holds the token just removed from the input, and `\yybytepure`, that (again, with very few exceptions) holds a 'normalized' version of the read character (i.e. a character of the same character code as `\yycp@`, and category 11 (to be even more precise (and to use nested parentheses), 'normalized' characters have the same category code as the current category code of `@`)).

Most of the time it is the character code one needs (say, in the case of `\{`, `\}`, `\&` and so on) but under some circumstances the distinction is important (outside of `\vb{...}`, the sequence `\1` has nothing to do with the digit '`1`'). This mechanism makes it easy to examine the consumed token. It also forms the foundation of the 'hidden context' passing mechanism described later.

The remainder of this section discusses the internals of `\yyinput` and some of the design trade-offs one has to make while working on processing general TEX token streams. It is typeset in 'small print' and can be skipped if desired.

To examine every token in its path (including spaces that are easy to skip), the input routine uses one of the two well-known TEXnologies: `\futurelet\next\examinenext` or equally effective `\afterassignment\next\let=`␣. Recursively inserting one of these sequences, `\yyinput` can go through any list of tokens, as long as it knows where to stop (i.e. return an end of file character). The signal to stop is provided by the `\yyeof` primitive which should not appear in any 'ordinary' text presented for parsing, other than for the purpose of providing such a stop signal. Even the dependence on `\yyeof` can be eliminated if one is willing to invest the time in writing macros that juggle TEX's `\token` registers and only limit oneself to input from such registers (which is, aside from an obvious efficiency hit, a strain on TEX's memory, as you have to store multiple (3 in

the general case) copies of your input to be able to back up when the lexer makes a wrong choice). There does not seem to be a way of doing it unless the text has been stored in a `\token` register first (or storing the whole input as a *parameter* for the appropriate macro: this scheme is remarkably powerful and leads to *expandable* versions of very complicated macros, although the amount of effort required to write such macros grows at a frightening rate). All of these are non-issues for the text inside `\vb{...}` and the care that `\yyinput` takes in processing characters inside such lists is an overkill. In a more 'hostile' environment (such as the one encountered by the now obsolete `\Tex` macros), this extra attention to detail pays off in the form of a more robust input mechanism.

One subtlety deserves a special mention here, as it can be important to the designer of 'higher-level' scanning macros. Two types of tokens are extremely difficult to deal with whenever TEX's own lexing mechanisms are used: (implicit) spaces and even more so, braces. We will only discuss braces here, however, almost everything that follows applies equally well to spaces (category 10 tokens to be precise), with a few simplifications (or complications, in a couple of places). To understand the difficulty, let's consider one of the approaches above:

```
\futurelet\next\examinenext.
```

The macro `\examinenext` usually looks at `\next` and inserts another macro (usually also called `\next`) at the very end of its expansion list. This macro usually takes one parameter, to consume the next token. This mechanism works flawlessly, until the lexer encounters a {br,sp}ace. The `\next` sequence, seen by `\examinenext` contains a lot of information about the brace ahead: it knows its category code (left brace, so 1), its character code (in case there was, say a `\catcode`\[=1␣` earlier) but not whether it is a 'real' brace (i.e. a character {$_1$) or an implicit one (a `\bgroup`). There is no way to find that out until the control sequence 'launched' by `\examinenext` sees the token as a parameter.

If the next token is a 'real' brace, however, `\examinenext`'s successor will never see the token itself: the braces are stripped by TEX's scanning mechanism. Even if it finds a `\bgroup` as the parameter, there is no guarantee that the actual input was not `{\bgroup}`. One way to handle this is by using `\string` ahead of any consumption of the next token. If prior to expanding `\string` care has been taken to set the `\escapechar` appropriately (remember, we know the character code in advance), as soon as one sees a character with `\escapechar`'s character code, (s)he knows that an implicit brace has just been seen. One added complication to all this is that a very determined programmer can insert an *active* character (using, say, the `\uccode` mechanism) that has the *same* character code as the *brace* token that it has been `\let` to! Setting this possibility aside, the `\string` mechanism (or, its cousin, `\meaning`) is not perfect: both produce a sequence of category 12 and 10 tokens. If it is indeed a brace character that we just saw, we can consume the next token and move on but what if this was a control sequence? After all, just as easily as `\string` makes a sequence into characters, `\csname`...`\endcsname` pair will make any sequence of characters into a control sequence. Huh ...

What we need is a backup mechanism: if one has a copy of the token sequence ahead, one can use `\string` to see if it is a real brace first, and if it is, consume it and move on (the active character case can be handled as the implicit case below, with one extra backup to count how many tokens have been consumed). At this point one has to *reinsert* the brace in case, at some point, a future 'back up' requires that the rest of the tokens are removed from the output (to avoid '`Too many }'s`' complaints from TEX). This can be done by using the `\iftrue{\else}\fi` trick but of course, some bookkeeping is needed to keep track of how far inside the brace groups we are. If it is an implicit brace, more work is needed: read all the characters that `\string` produced (an maybe more), then remember the number of characters consumed. Remove the rest of the input using the method described above and restart the scanning from the same point knowing that the next token can be scanned as a parameter.

Another strategy is to design a general enough macro that counts tokens in a token register and simply recount the tokens after every brace was consumed.

Either way, it takes a lot of work. If anyone would like to pursue the counting strategy, simple counting macros are provided in `/examples/count/count.sty`. The macros in this example supply a very general counting mechanism that does not depend on `\yyeof` (or *any* other token) being 'special' and can count the tokens in any token register, as long as none of those tokens is an `\outer` control sequence. In other words, if the macro is used immediately after the assignment to the token register, it should always produce a correct count.

Needless to say, if such a general mechanism is desired, one has to look elsewhere. The added complications of treating spaces (TEX tends to ignore them most of the time) make this a torturous exercise in TEX's macro wizardry. The included `\yyinput` has two ways of dealing with braces: strip them or view the whole group as a token. Pick one or write a different `\yyinput`. Spaces, implicit or explicit are reported as a specially selected character code and consumed with a likeness of

```
\afterassignment\moveon\let\next=␣.
```

Now that a steady stream of character codes is arriving at `\yylex` after `\yyreturn` the job of converting it into numerical tokens is performed by the *scanner* (or *lexer*, or *tokenizer*, or even *tokener*), discussed in the next section.

17 Lexing in TEX

In a typical system that uses a parser to process text, the parsing pass is usually split into several stages: the raw input, the lexical analysis (or simply *lexing*), and the parsing proper. The *lexing* (also called *scanning*, we use these terms interchangeably) clumps various sequences of characters into *tokens* to facilitate the parsing stage. The reasons for this particular hierarchy are largely pragmatic and are partially historic (there is no reason that *parsing* cannot be done in multiple phases, as well, although it usually isn't).

If one remembers a few basic facts from the formal language theory, it becomes obvious that a lexer, that parses *regular* languages, can (theoretically) be replaced by an LALR parser, that parses *context-free* ones (or some subset thereof, which is still a super set of all regular languages). A common justification given for creating specialized lexers is efficiency and speed. The reality is somewhat more subtle. While we do care about the efficiency of parsing in TEX, having a specialized scanner is important for a number of different reasons.

The real advantage of having a dedicated scanner is the ease with which it can match incomplete inputs

and back up. A parser can, of course, *recognize* any valid input that is also acceptable to a lexer, as well as *reject* any input that does not form a valid token. Between those two extremes, however, lies a whole realm of options that a traditional parser will have great difficulty exploring. Thus, to mention just one example, it is relatively easy to set up a DFA [1]) so that the *longest* matching input is accepted. The only straightforward way to do this with a traditional parser is to parse longer and longer inputs again and again. While this process can be optimized to a certain degree, the fact that a parser has a *stack* to maintain limits its ability to back up.

As an aside, the mechanism by which `CWEB` assembles its 'scraps' into chunks of recognized code is essentially iterative lexing, very similar to what a human does to make sense of complicated texts. Instead of trying to match the longest running piece of text, `CWEB` simply looks for patterns to combine inputs into larger chunks, which can later be further combined. Note that this is not quite the same as the approach taken by, say GLR parsers, where the parser must match the *whole* input or declare a failure. Where a `CWEB`-type parser may settle for the first available match (or the longest available) a GLR parser must try *all* possible matches or use an algorithm to reject the majority of the ones that are bound to fail in the end.

This '`CWEB` way' is also different from a traditional 'strict' LR parser/scanner approach and certainly deserves serious consideration when the text to be parsed possesses some rigid structure but the parser is only allowed to process it one small fragment at a time.

Returning to the present macro suite, the lexer produced by `flex` uses integer tables similar to those employed by `bison` so the usual TEXniques used in implementing `\yyparse` are fully applicable to `\yylex`.

An additional advantage provided by having a `flex` scanner implemented as part of the suite is the availability of the original `bison` scanner written in C for the use by the macro package.

This said, the code generated by `flex` contains a few idiosyncrasies not present in the `bison` output. These 'quirks' mostly involve handling of end of input and error conditions. A quick glance at the `\yylex` implementation will reveal a rather extensive collection of macros designed to deal with end of input actions.

Another difficulty one has to face in translating `flex` output into TEX is a somewhat unstructured namespace delivered in the final output (this is partially due to the POSIX standard that `flex` strives to follow). One consequence of this 'messy' approach is that the writer of a `flex` scanner targeted to TEX has to declare `flex` 'states' (more properly called *subautomata*) twice: first for the benefit of `flex` itself, and then again, in the C *preamble* portion of the code to output the states to be used by the action code in the lexer. `Define_State(...)` macro is provided for this purpose. This macro can be used explicitly by the programmer or be inserted by a specially designed parser. Using `CWEB` helps to keep these declarations together.

The 'hand-off' 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 the user: the character sequence representing the token just matched (`\yytext`). This is not the whole story though. There are three more token sequences that 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* $(c_{\mathrm{ch}}, c_{\mathrm{cat}})$ pairs, a few simple conventions are followed, some of which will be explained below. This sequence is provided merely for convenience and its typical use is to generate a key for an associate 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 the mechanism itself and the motivation behind it are somewhat subtle, let me spend a few moments discussing the range of formatting options desirable in a generic pretty-printer.

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'). To provide a simple illustration, an isolated identifier, such as '`lg_integer`' can be a type name, a variable name, or a structure tag (in a

[1]) Which stands for Deterministic Finite Automaton, a common (and mathematically unique) way of implementing a scanner for regular languages. Incidentally LALR mentioned above is short for Look Ahead Left to Right.

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 a 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 (the original `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 (about 90%? it would be interesting to carry out a study in the spirit of the ones discussed in [Jo] to find out) 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 canceling 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 second construct (`@[...@]`), is used to create an *exp* scrap out of whatever happens to be inside the brackets.

This is a powerful tool at the author'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 grammar. 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 approach is similar to, although not quite the same one employed by the *General LR parsers*).

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 (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; it also 'bakes in' the pretty-printing conventions into the language structure (making 'hidden' context much less 'stealthy'). It does avoid the 'synchronicity problem' mentioned below.

A marginally better technique is to introduce a new regular expression recognizable by the scanner which will then do all the necessary bookkeeping upon matching the sequence. All the difficulties with altering the

grammar mentioned above apply in this case, as well, only at the 'lexical analysis level'. At a minimum, the set of tokens matched by the scanner would have to be changed.

A much better approach involves inserting the hints at the input stage and passing this information to the scanner and 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. This makes hints, in effect, invisible.

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 labeled 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).

Finally, while probably not directly related to the present discussion, this approach has one more interesting feature: after the parser is finished, the parser output and the streams exist 'statically', fully available for any last minute preprocessing or for debugging purposes, if necessary. Under most circumstances, the parser output is 'executed' and the macros in the output are the ones reading the various streams using the pointers supplied at the parsing stage (at least, this is the case for all the parsers supplied with the package).

18 Inside semantic actions: switch statements and 'functions' in TEX

Now you have a lexer for your input, and a grammar ready to be put into action (we will talk about actions a bit later). It is time to discuss how the tables produced by `bison` get converted into TEX *macros* that drive the parser in *TEX*.

The tables that drive the `bison` input parsers are collected in various `{b,d,f,g,n}yytab.tex` and `small_tab.tex`. Each one of these files contains the tables that implement a specific parser used during different stages of processing. Their exact function is well explained in the source file produced by `bison` (*how* this is done is explained elsewhere, see [Ah] for a good reference). It would suffice to mention here that there are three types of tables in this file: (1)numerical tables such as `\yytable` and `\yycheck` (both are either TEX's token registers in an unoptimized parser or associate arrays in an optimized version of such as discussed below), (2)a string array `\yytname`, and (3)an action switch. The action switch is what gets called when the parser does a *reduction*. It is easy to notice that the numerical tables come 'premade' whereas the string array consisting of token names is difficult to recognize. This is intentional: this form of initialization is designed to allow the widest range of characters to appear inside names. The macros that do this reside in `yymisc.sty`. The generated table files also contain constant and token declarations used by the parser.

The description of the process used to output `bison` tables in an appropriate form continues in the section about outputting TEX tables, we pick it up here with the description of the syntax-directed translation and

the actions. The line

```
\switchon\next\in\currentswitch
```

is responsible for calling an appropriate action in the current switch, as is easy to infer. A *switch* is also a macro that consists of strings of TEX tokens intermixed with TEX macros inside braces. Each group of macros gets executed whenever the character or the group of characters in `\next` matches a substring preceding the braced group. If there are two different substrings that match, only the earliest group of macros gets expanded. Before a state is used, a special control sequence, `\setspecialcharsfrom\switchname` can be used to put the TEX tokens in a form suitable for the consumption by `\switchon`'s. The most important step it performs is it *turns every token in the list into a character with the same character code and category 12*. Thus `\{` becomes $\texttt{\{}_{12}$. There are other ways of inserting tokens into a state: enclosing a token or a string of tokens in `\raw...\raw` adds it to the state macro unchanged. If you have a sequence of category 12 characters you want to add to the state, put it after `\classexpand` (such sequences are usually prepared by the `\setspecialchars` macro that uses the token tables generated by `bison` from your grammar).

You can give a case a readable label (say, `brackets`) and enclose this label in `\raw...\raw`. A word of caution: an 'a' inside of `\raw...\raw` (which is most likely an $\texttt{a}_{11}$ unless you played with category codes before loading the `\switchon` macros) and the one outside it are two different characters, as one is no longer a letter (category 11) in the eyes of TEX whereas the other one still is. For this reason one should not use characters other than letters in `h{is,er}` state names: the way a state picks an action does not distinguish between, say, a '`(`' in '`(letter)`' and a stand alone '`(`' and may pick an action that you did not intend. This applies even if '`(`' is not among the characters explicitly inserted in the state macro: if an action for a given character is not found in the state macro, the `\switchon` macro will insert a current `\default` action instead, which most often you would want to be `\yylex` or `\yyinput` (i.e. skip this token). If '`(`' or '`)`' matches the braced group that follows '`(letter)`' chaos may ensue (most likely TEX will keep reading past the `\end` or `\yyeof` that should have terminated the input). Make the names of character categories as unique as possible: the `\switchon` is simply a string matching mechanism, with the added distinction between characters of different categories.

Finally, the construct `\statecomment`*anything*`\statecoment` allows you to insert comments in the state sequence (note that the state *name* is put at the beginning of the state macro (by `\setspecialcharsfrom`) in the form of a special control sequence that expands to nothing: this elaborate scheme is needed because another control sequence can be `\let` to the state macro which makes the debugging information difficult to decipher). The debugging mode for the lexer implemented with these macros is activated by `\tracedfatrue`.

The functionality of the `\switchon` macros (for 'historical' reasons, one can also use `\action` as a synonym) has been implemented in a number of other macro packages (see [Fi] that discusses the well-known and widely used `\CASE` and `\FIND` macros). The macros in this collection have the additional property that the only assignments that persist after the `\switchon` completes are the ones performed by the user code inside the selected case.

This last property of the switch macros is implemented using another mechanism that is part of this macro suite: the 'subroutine-like' macros, `\begingroup...\tokreturn`. For examples, an interested reader can take a look at the macros included with the package. A typical use is `\begingroup...\tokreturn{}{\toks0 }{}` which will preserve all the changes to `\toks0` and have no other side effects (if, for example, in typical TEX vernacular, `\next` is used to implement tail recursion inside the group, after the `\tokreturn`, `\next` will still have the same value it had before the group was entered). This functionality comes at the expense of some computational efficiency.

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 a term in `bison`). Instead of going into the minutia 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]` but this requires a bit more effort on the user's part 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.

One last remark about the parser operation is worth making here: the parser automaton itself does not make any `\global` assignments. This (along with some careful semantic action writing) can be used to 'localize' the effects of the parser operation and, most importantly, to create 'reentrant' parsers that can, e.g. call *themselves* recursively.

19 'Optimization'

By default, the generated parser and scanner keep all of their tables in separate token registers. Each stack is kept in a single macro (this description is further complicated by the support for parser *namespaces* that exists even for unoptimized parsers but this subtlety will not be mentioned again—see the macros in the package for further details). 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 (which are immutable) in token registers does not have any better rationale than saving the control sequence memory (the most abundant memory in TEX), this way of storing *stacks* does have an advantage when multiple parsers get to 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', all these control sequenced are 'spread over' appropriate associative arrays. One caveat to be aware of: the action switches for both the parser and the scanner have to be output differently (a command line option is used to control this) for optimized and unoptimized parsers. 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.

20 *TEX* with a different *slant* or do you C an escape?

Some TEX productions below probably look like alien script. The authors of [Er] cite a number of reasons pretty printing of TEX in general is a nearly impossible task. The macros included with the package follow a very straightforward strategy and do not try to be very comprehensive. Instead, the burden of presenting TEX code in a readable form is placed on the programmer. Appropriate hints can be supplied by means of indenting the code, using assignments (=) where appropriate, etc. If you would rather look at straight TEX instead, the line `\def\texnspace{other}` at the beginning of this section can be uncommented and ${}^{\mathrm{nox}\bullet}(\,\Upsilon \leftarrow \langle \Upsilon_1 \rangle\,)$ becomes `\noexpand\inmath{\yy0{\yy1{}}}`. There is, however, more to this story. A look at the actual file will reveal that the line above was typed as

```
TeX_( "/noexpand/inmath{/yy0{/yy1{}}}" );
```

The 'escape character' is leaning the other way! The lore of TEX is uncompromising: '`\`' is *the* escape character. What is the reason to avoid it in this case?

The mystery is not very deep: '`/`' was chosen as an escape character by the parser macros (a quick glance at `?yytab.tex` will reveal as much). There is, of course, nothing sacred (other than tradition, which this author is trying his hardest to follow) about what character code the escape character has. The reason for this choice is straightforward: '`\`' is a special character in C, as well (also an 'escape' in fact). The line `TeX_( "..." );` is a *macro-call* but ... in C. This function simply prints out (almost 'as-is') the line in parenthesis. An attempt at `TeX_( "\noexpand" );` would result in

```
01                      01
02      oexpand         02
```

Other escape combinations [1]) are even worse: most are simply undefined. If anyone feels trapped without an escape, however, the same line can be typed as

```
TeX_( "\\noexpand\\inmath{\\yy0{\\yy1{}}}" );
```

Twice the escape!

If one were to look closer at the code, another oddity stands out: there are no `$`'s anywhere in sight. The big money, `$` is a beloved character in `bison`. It is used in action code to reference the values of the appropriate terms in a production. If mathematics pays your bills, use `\inmath` instead.

21 The `bison` parser(s)

Let's take a short break for a broad overview of the input file. The basic structure is that of an ordinary `bison` file that produces plain C output. The C actions, however, are programmed to output TEX.

⟨ `bg.yy` 21 ⟩ =

```
⟨ Grammar parser C preamble 95 ⟩
⟨ Grammar parser bison options 25 ⟩
⟨ union ⟩      ⟨ Union of grammar parser types 100 ⟩
⟨ Grammar parser C postamble 96 ⟩
⟨ Tokens and types for the grammar parser 26 ⟩

⟨ Fake start symbol for rules only grammar 31 ⟩
⟨ Parser common productions 44 ⟩
⟨ Parser grammar productions 60 ⟩
```

22 Bootstrap mode is next. The reason for a separate bootstrap parser is to collect the minimal amount of information to 'spool up' the 'production' parsers. To understand the mechanics and the reasons behind it, consider what happens following a declaration such as `%token TOKEN "token"` (or, as it would be typeset by the macros in this package '⟨`token`⟩ `TOKEN token`'; see the index entries for more details). The two names for the same token are treated very differently. `TOKEN` becomes an **enum** constant in the C parser generated by `bison`. Even when that parser becomes part of the 'driver' program that outputs the TEX version of the parser tables, there is no easy way to output the *names* of the appropriate **enum** constants. The other name (`"token"`) becomes an entry in the *yytname* array. These names can be output by either the 'driver' or TEX itself after the `\yytname` table has been input. The scanner, on the other hand, will use the first version (`TOKEN`). Therefore, it is important to establish an equivalence between the two versions of the name. In the 'real' parser, the token values are output in a special header file. Hence, one has to either parse the header file to establish the equivalences or find some other means to find out the numerical values of the tokens.

One approach is to parse the file containing the *declarations* and extract the equivalences between the names from it. This is the function of the bootstrap parser. Since the lexer is reused, some token values need to be known in advance (and the rest either ignored or replaced by some 'made up' values). These tokens are 'hard coded' into the parser file generated by `bison` and output using a special function. The switch '**#define** `BISON_BOOTSTRAP_MODE`' tells the 'driver' program to output the hard coded token values.

Note that the equivalence of the two versions of token names would have to be established every time a 'string version' of a token is declared in the `bison` file and the 'macro name version' of the token is used by the corresponding scanner. To establish this equivalence, however, the bootstrapping parser below is not

[1]) Here is a full list of *defined* escaped characters in C: `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v`, `\`[*octal digit*], `\'`, `\"`, `\?`, `\\`, `\x`, `\u`, `\U`. Note that the last three combinations must be followed by a specific string of characters to appear in the input without generating errors.

always necessary (see the xxpression example, specifically, the file xxpression.w in the examples directory for an example of using a different parser for this purpose). The reason it is necessary here is that a parser for an appropriate subset of the bison syntax is not yet available (indeed, *any* functional parser for a bison syntax subset would have to use the same scanner (unless you want to write a custom scanner for it), which would need to know how to output tokens, for which it would need a parser for a subset of bison syntax ... it is a 'chicken and egg'). Hence the name 'bootstrap'. Once a functional parser for a large enough subset of the bison input grammar is operational, *it* can be used to pair up the token names.

The second function of the bootstrap parser is to collect information about the scanner's states. The mechanism is slightly different for states. While the token equivalences are collected purely in 'TEX mode', the bootstrap parser collects all the state names into a special C header file. The reason is simple: unlike the token values, the numerical values of the scanner states are not passed to the 'driver' program in any data structure and are instead defined as ordinary macros. The header file is the information the 'driver' file needs to output the state values.

An additional subtlety in the case of state value output is that the main lexer for the bison grammar utilizes states extensively and thus cannot be easily used with the bootstrap parser before the state values are known. The solution is to substitute a very simple scanner barely capable of lexing state declarations. Such a scanner is implemented in ssffo.w (the somewhat cryptic name stands for 'simple scanner for flex options').

⟨ bb.yy 22 ⟩ =

```
⟨ Grammar parser C preamble 95 ⟩
#define BISON_BOOTSTRAP_MODE
⟨ Grammar parser bison options 25 ⟩
⟨union⟩ ⟨ Union of grammar parser types 100 ⟩
⟨ Bootstrap parser C postamble 97 ⟩
⟨ Tokens and types for the grammar parser 26 ⟩

⟨ Fake start symbol for bootstrap grammar 32 ⟩
⟨ Parser bootstrap productions 52 ⟩
⟨ flex options parser productions 40 ⟩
⟨ List of symbols 55 ⟩
⟨ Definition of symbol 84 ⟩
```

23 The prologue parser is responsible for parsing various grammar declarations as well as parser options.

⟨ bd.yy 23 ⟩ =

```
⟨ Grammar parser C preamble 95 ⟩
⟨ Grammar parser bison options 25 ⟩
⟨union⟩ ⟨ Union of grammar parser types 100 ⟩
⟨ Grammar parser C postamble 96 ⟩
⟨ Tokens and types for the grammar parser 26 ⟩

⟨ Fake start symbol for prologue grammar 34 ⟩
⟨ Parser common productions 44 ⟩
⟨ Parser prologue productions 35 ⟩
```

24 Full `bison` input parser is used when a complete `bison` file is expected. It is also capable of parsing a 'skeleton' of such a file, similar to the one that follows this paragraph.

⟨ bf.yy 24 ⟩ =
```
.....
⟨ Grammar parser C preamble 95 ⟩
.....
⟨ Grammar parser bison options 25 ⟩
⟨union⟩ ⟨ Union of grammar parser types 100 ⟩
.....
⟨ Grammar parser C postamble 96 ⟩
.....
⟨ Tokens and types for the grammar parser 26 ⟩

⟨ Parser common productions 44 ⟩
⟨ Parser prologue productions 35 ⟩
⟨ Parser grammar productions 60 ⟩
⟨ Parser full productions 29 ⟩
```

25 The first two options are essential for the parser operation. The start symbol can be set implicitly by listing the appropriate production first.

⟨ Grammar parser bison options 25 ⟩ =
```
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆   (set as ⟨debug⟩)
⟨start⟩           input
```
This code is used in sections 21, 22, 23, and 24.

26 Grammar rules

Most of the original comments present in the grammar file used by `bison` itself have been preserved and appear in *italics* at the beginning of each appropriate section.

To facilitate the *bootstrapping* of the parser (see above), some declarations have been separated into their own sections. Also, a number of new rules have been introduced to create a hierarchy of 'subparsers' that parse subsets of the grammar. We begin by listing most of the tokens used by the grammar. Only the string versions are kept in the *yytname* array, which, in part is the reason for a special bootstrapping parser as explained earlier.

⟨ Tokens and types for the grammar parser 26 ⟩ =
```
end of file   (GRAM_EOF):0
string        («string»)
⟨token⟩       (PERCENT_TOKEN)
⟨nterm⟩       (PERCENT_NTERM)
⟨type⟩        (PERCENT_TYPE)
⟨destructor⟩  (PERCENT_DESTRUCTOR)
⟨printer⟩     (PERCENT_PRINTER)
⟨left⟩        (PERCENT_LEFT)
⟨right⟩       (PERCENT_RIGHT)
⟨nonassoc⟩    (PERCENT_NONASSOC)
⟨precedence⟩  (PERCENT_PRECEDENCE)
⟨prec⟩        (PERCENT_PREC)
⟨dprec⟩       (PERCENT_DPREC)
⟨merge⟩       (PERCENT_MERGE)
⟨ Global Declarations 27 ⟩
```
See also sections 28, 46, and 70.

This code is used in sections 21, 22, 23, and 24.

27 We continue with the list of tokens below, following the layout of the original parser.

⟨ Global Declarations 27 ⟩ =

⟨code⟩	(PERCENT_CODE)	⟨token-table⟩	(PERCENT_TOKEN_TABLE)
⟨default-prec⟩	(PERCENT_DEFAULT_PREC)	⟨verbose⟩	(PERCENT_VERBOSE)
⟨define⟩	(PERCENT_DEFINE)	⟨yacc⟩	(PERCENT_YACC)
⟨defines⟩	(PERCENT_DEFINES)	{...}	(BRACED_CODE)
⟨error-verbose⟩	(PERCENT_ERROR_VERBOSE)	%?{...}	(BRACED_PREDICATE)
⟨expect⟩	(PERCENT_EXPECT)	[identifier]	(BRACKETED_ID)
⟨expect-rr⟩	(PERCENT_EXPECT_RR)	char	(**char**)
⟨⋆⟩	(PERCENT_FLAG)	epilogue	(EPILOGUE)
⟨file-prefix⟩	(PERCENT_FILE_PREFIX)	=	(EQUAL)
⟨glr-parser⟩	(PERCENT_GLR_PARSER)	identifier	(«identifier»)
⟨initial-action⟩	(PERCENT_INITIAL_ACTION)	identifier:	(«identifier: »)
⟨language⟩	(PERCENT_LANGUAGE)	⟨%⟩	(PERCENT_PERCENT)
⟨name-prefix⟩	(PERCENT_NAME_PREFIX)	|	(PIPE)
⟨no-default-prec⟩	(PERCENT_NO_DEFAULT_PREC)	%{...%}	(PROLOGUE)
⟨no-lines⟩	(PERCENT_NO_LINES)	;	(SEMICOLON)
⟨non...ic-parser⟩	(PER...NON...IC_PARSER)	<*tag*>	(<*tag*>)
⟨output⟩	(PERCENT_OUTPUT)	<*>	(TAG_ANY)
⟨require⟩	(PERCENT_REQUIRE)	<>	(TAG_NONE)
⟨skeleton⟩	(PERCENT_SKELETON)	integer	(**int**)
⟨start⟩	(PERCENT_START)	⟨param⟩	(PERCENT_PARAM: ⟨union⟩.*param*)

This code is used in section 26.

28 Extra tokens for typesetting `flex` state declarations and options are declared in addition to the ones that a standard `bison` parser recognizes.

⟨ Tokens and types for the grammar parser 26 ⟩ +=

⟨**option**⟩$_{\mathrm{f}}$ ⟨*auto*⟩ ⟨**state-x**⟩$_{\mathrm{f}}$ ⟨*auto*⟩

⟨**state-s**⟩$_{\mathrm{f}}$ ⟨*auto*⟩

29 We are ready to describe the top levels of the parse tree. The first 'sub parser' we consider is a 'full' parser, that is the parser that expects a full grammar file, complete with the prologue, declarations, etc. This parser can be used to extract information from the grammar that is otherwise absent from the executable code generated by `bison`. This includes, for example, the 'name' part of `$[name]`. This parser is therefore used to generate the 'symbolic switch' to provide support for symbolic term names similar to 'genuine' `bison`'s `$[...]` syntax.

⟨ Parser full productions 29 ⟩ =

input : *prologue_declarations* ⟨%⟩ *grammar* *epilogue*$_{\mathrm{opt}}$ ⟨ Finish the input setup 30 ⟩

This code is used in section 24.

30 The action of the parser in this case is simply to separate the accumulated 'parse tree' from the auxiliary information carried by the parser on the stack.

⟨ Finish the input setup 30 ⟩ =

$\pi_2(\Upsilon_3) \mapsto v_a$ ▷ extract grammar contents ◁

$\Upsilon \leftarrow \langle \llcorner v_a \lrcorner \rangle\ \Omega = \Upsilon_0$

This code is used in section 29.

31 Another subgrammar deals with the syntax of isolated `bison` rules. This is the most commonly used 'subparser' since a rules cluster is the most natural 'unit' to include in a `CWEB` file.

⟨ Fake start symbol for rules only grammar 31 ⟩ =

input : *grammar* *epilogue*$_{\mathrm{opt}}$ $\pi_2(\Upsilon_1) \mapsto \Omega$

This code is used in section 21.

32 The bootstrap parser has a very narrow set of goals: it is concerned with ⟨`token`⟩ and ⟨`nterm`⟩ declarations only in order to supply the token information to the lexer (since, as noted above, such information is not kept in the *yytname* array). It also extends the syntax of a *grammar_declaration* by allowing a declaration with or without semicolon at the end (the latter is only allowed in the prologue). This works since the token declarations have been carefully separated from the rest of the grammar in different `CWEB` sections. The range of tokens understood by the bootstrap parser is limited, hence most of the other rules are ignored.

⟨ Fake start symbol for bootstrap grammar 32 ⟩ =

input : *grammar_declarations*	$\Omega = \Upsilon_1$
grammar_declarations :	
symbol_declaration $;_{\text{opt}}$	⟨ Carry on 33 ⟩
flex_declaration $;_{\text{opt}}$	⟨ Carry on 33 ⟩
grammar_declarations symbol_declaration $;_{\text{opt}}$	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \text{val}\,\Upsilon_2 \rangle$
grammar_declarations flex_declaration $;_{\text{opt}}$	$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \text{val}\,\Upsilon_2 \rangle$
$;_{\text{opt}}$: ∘ | ;	

This code is used in section 22.

33 The following is perhaps the most common action performed by the parser. It is done automatically by the parser code but this feature is undocumented so we supply an explicit action in each case.

⟨ Carry on 33 ⟩ =
$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \rangle$

This code is used in sections 32, 37, 39, 40, 44, 47, 53, 55, 56, 57, 58, 61, 69, 81, 87, 88, 89, 90, 91, 92, and 93.

34 Next, a subgrammar for processing prologue declarations. Finer differentiation is possible but the 'subparsers' described here work pretty well and impose a mild style on the grammar writer.

⟨ Fake start symbol for prologue grammar 34 ⟩ =

input : *prologue_declarations epilogue*$_{\text{opt}}$	$\pi_2(\Upsilon_1) \mapsto \Omega$
prologue_declarations ⟨%⟩ ⟨%⟩ `EPILOGUE`	$\pi_2(\Upsilon_1) \mapsto \Omega$
prologue_declarations ⟨%⟩ ⟨%⟩	$\pi_2(\Upsilon_1) \mapsto \Omega$

This code is used in section 23.

35 *Declarations: before the first* ⟨%⟩. We are now ready to deal with the specifics of the declarations themselves. The `\grammar` macro is a 'structure', whose first 'field' is the grammar itself, whereas the second carries the type of the last declaration added to the grammar.

⟨ Parser prologue productions 35 ⟩ =

prologue_declarations :	
∘	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash grammar\{}\,\texttt{\}\{}\,^{\text{nx}}\varnothing\,\texttt{\}}\rangle$
prologue_declarations prologue_declaration	⟨ Attach a prologue declaration 36 ⟩

See also sections 37, 39, and 93.

This code is used in sections 23 and 24.

36 ⟨ Attach a prologue declaration 36 ⟩ =
⟨ Attach a productions cluster 63 ⟩

This code is used in section 35.

37 Here is a list of most kinds of declarations that can appear in the prologue. The scanner returns the 'stream pointers' for all the keywords so the declaration 'structures' pass on those pointers to the grammar list. The original syntax has been left intact even though for the purposes of this parser some of the inline rules are unnecessary.

⟨ Parser prologue productions 35 ⟩ +=

prologue_declaration :	
grammar_declaration	⟨ Carry on 33 ⟩
`%{...%}`	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash prologuecode}\,\text{val}\,\Upsilon_1\rangle$
⟨⋆⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\text{val}\,\Upsilon_1\rangle$

⟨**define**⟩ *variable value*	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash vardef}\,\{\,\text{val}\,\Upsilon_2\,\}\{\,\text{val}\,\Upsilon_3\,\}\text{val}\,\Upsilon_1\rangle$
⟨**defines**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{defines}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**defines**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{defines}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**error-verbose**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{error verbose}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**expect**⟩ **int**	$v_a \leftarrow \langle\,\texttt{expect}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**expect-rr**⟩ **int**	$v_a \leftarrow \langle\,\texttt{expect-rr}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**file-prefix**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{file prefix}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**glr-parser**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{glr parser}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**initial-action**⟩ **{...}**	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash initaction}\,\text{val}\,\Upsilon_2\rangle$
⟨**language**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{language}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**name-prefix**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{name prefix}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**no-lines**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{no lines}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**non...ic-parser**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{nondet. parser}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**output**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{output}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**param**⟩ ◇ *params*	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash paramdef}\,\{\,\text{val}\,\Upsilon_3\,\}\text{val}\,\Upsilon_1\rangle$
⟨**require**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{require}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**skeleton**⟩ «**string**»	$v_a \leftarrow \langle\,\texttt{skeleton}\rangle\,\langle$ Prepare one parametric option 38 $\rangle$
⟨**token-table**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{token table}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**verbose**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{verbose}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
⟨**yacc**⟩	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash optionflag}\,\{\,\texttt{yacc}\,\}\{\,\}\text{val}\,\Upsilon_1\rangle$
;	$\Upsilon \leftarrow \langle^{\text{nx}}\varnothing\rangle$
params :	
params **{...}**	$\Upsilon \leftarrow \langle\text{val}\,\Upsilon_1\,{}^{\text{nx}}\texttt{\textbackslash braceit}\,\text{val}\,\Upsilon_2\rangle$
{...}	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash braceit}\,\text{val}\,\Upsilon_1\rangle$

38 This is a typical parser action: encapsulate the 'type' of the construct just parsed and attach some auxiliary info, in this case the stream pointers.

⟨ Prepare one parametric option 38 ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash oneparametricoption}\,\{\,\llcorner v_a \lrcorner\,\}\{\,\text{val}\,\Upsilon_2\,\}\text{val}\,\Upsilon_1\rangle$

This code is used in sections 37 and 44.

39 Some extra declarations to typeset `flex` options and declarations. These are not part of the `bison` syntax but their structure is similar enough that they can be included in the grammar.

⟨ Parser prologue productions 35 ⟩ +=

prologue_declaration :

flex_declaration ⟨ Carry on 33 ⟩

⟨ `flex` options parser productions 40 ⟩

40 The syntax of `flex` options was extracted from `flex` documentation so it is not guaranteed to be correct.

⟨ `flex` options parser productions 40 ⟩ =

flex_declaration :	
⟨**option**⟩$_{\text{f}}$ *flex_option_list*	⟨ Define `flex` option list 41 ⟩
flex_state symbols$_1$	⟨ Define `flex` states 42 ⟩
flex_state :	
⟨**state-x**⟩$_{\text{f}}$	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash flexxstatedecls}\,\text{val}\,\Upsilon_1\rangle$
⟨**state-s**⟩$_{\text{f}}$	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash flexsstatedecls}\,\text{val}\,\Upsilon_1\rangle$
flex_option_list :	
flex_option	⟨ Carry on 33 ⟩
flex_option_list flex_option	⟨ Add a `flex` option 43 ⟩
flex_option :	
«identifier»	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash flexoptionpair}\,\{\,\text{val}\,\Upsilon_1\,\}\{\,\}\rangle$
«identifier» **=** *symbol*	$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\textbackslash flexoptionpair}\,\{\,\text{val}\,\Upsilon_1\,\}\{\,\text{val}\,\Upsilon_3\,\}\rangle$

This code is used in sections 22 and 39.

41 ⟨ Define `flex` option list 41 ⟩ =
$\Upsilon \leftarrow \langle^{\text{nx}}$`\flexoptiondecls{` val Υ_2 `}`val $\Upsilon_1\rangle$
This code is used in section 40.

42 ⟨ Define `flex` states 42 ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\pi_3(\Upsilon_1) \mapsto v_c$
$\Upsilon \leftarrow \langle \llcorner v_a \lrcorner${ val Υ_2 }{$\llcorner v_b \lrcorner$}{$\llcorner v_c \lrcorner$}$\rangle$
This code is used in section 40.

43 ⟨ Add a `flex` option 43 ⟩ =
$\pi_2(\Upsilon_2) \mapsto v_a$ ▷ the identifier ◁
$\pi_4(v_a) \mapsto v_b$ ▷ the format pointer ◁
$\pi_5(v_a) \mapsto v_c$ ▷ the stash pointer ◁
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\text{nx}}{}_{\sqcup}${$\llcorner v_b \lrcorner$}{$\llcorner v_c \lrcorner$}val $\Upsilon_2\rangle$
This code is used in section 40.

44 *Grammar declarations.* These declarations can appear in both prologue and the rules sections. Their treatment is very similar to prologue-only options.
⟨ Parser common productions 44 ⟩ =
grammar_declaration :

precedence_declaration	⟨ Carry on 33 ⟩
symbol_declaration	⟨ Carry on 33 ⟩
⟨`start`⟩ *symbol*	$v_a \leftarrow$ ⟨`start`⟩ ⟨ Prepare one parametric option 38 ⟩
code_props_type `{...}` *generic_symlist*	⟨ Assign a code fragment to symbols 45 ⟩
⟨`default-prec`⟩	$\Upsilon \leftarrow \langle^{\text{nx}}$`\optionflag{default prec.}{ }`val $\Upsilon_1\rangle$
⟨`no-default-prec`⟩	$\Upsilon \leftarrow \langle^{\text{nx}}$`\optionflag{no default prec.}{ }`val $\Upsilon_1\rangle$
⟨`code`⟩ `{...}`	$\Upsilon \leftarrow \langle^{\text{nx}}$`\codeassoc{code}{ }`val $\Upsilon_2$val $\Upsilon_1\rangle$
⟨`code`⟩ «identifier» `{...}`	$\Upsilon \leftarrow \langle^{\text{nx}}$`\codeassoc{code}{`val Υ_2 `}`val $\Upsilon_3$val $\Upsilon_1\rangle$

code_props_type :

⟨`destructor`⟩	$\Upsilon \leftarrow \langle$`{destructor}`val $\Upsilon_1\rangle$
⟨`printer`⟩	$\Upsilon \leftarrow \langle$`{printer}`val $\Upsilon_1\rangle$

See also sections 47, 51, 53, 54, 56, 83, and 94.
This code is used in sections 21, 23, and 24.

45 ⟨ Assign a code fragment to symbols 45 ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$ ▷ name of the property ◁
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ contents of the braced code ◁
$\pi_2(\Upsilon_2) \mapsto v_c$ ▷ braced code format pointer ◁
$\pi_3(\Upsilon_2) \mapsto v_d$ ▷ braced code stash pointer ◁
$\pi_2(\Upsilon_1) \mapsto v_e$ ▷ code format pointer ◁
$\pi_3(\Upsilon_1) \mapsto v_f$ ▷ code stash pointer ◁
$\Upsilon \leftarrow \langle^{\text{nx}}$`\codepropstype{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}{` val Υ_3 `}{`$\llcorner v_c \lrcorner$`}{`$\llcorner v_d \lrcorner$`}{`$\llcorner v_e \lrcorner$`}{`$\llcorner v_f \lrcorner$`}`$\rangle$
This code is used in section 44.

46 ⟨ Tokens and types for the grammar parser 26 ⟩ +=
⟨`union`⟩ (`PERCENT_UNION`)

47 ⟨ Parser common productions 44 ⟩ +=

union_name : ∘ | «identifier»	⟨ Carry on 33 ⟩
grammar_declaration : ⟨`union`⟩ *union_name* `{...}`	⟨ Prepare union definition 48 ⟩
symbol_declaration : ⟨`type`⟩ `<`*tag*`>` *symbols*$_1$	⟨ Define symbol types 49 ⟩
precedence_declaration :	

precedence_declarator tag_{opt} *symbols.prec* ⟨ Define symbol precedences 50 ⟩

precedence_declarator :

⟨`left`⟩ | ⟨`right`⟩ | ⟨`nonassoc`⟩ | ⟨`precedence`⟩ $\Upsilon \leftarrow \langle^{\text{nx}}$`\preckind{precedence}`val $\Upsilon_1\rangle$

$\boldsymbol{tag}_{\text{opt}}$: ∘ | `<`*tag*`>` ⟨ Carry on 33 ⟩

48 ⟨ Prepare union definition 48 ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}$`\codeassoc{union}{`val Υ_2`}`val $\Upsilon_3$val $\Upsilon_1\rangle$

This code is used in section 47.

49 ⟨ Define symbol types 49 ⟩ =

$\Upsilon \leftarrow \langle^{\text{nx}}$`\typedecls{`val Υ_2`}{`val Υ_3`}`val $\Upsilon_1\rangle$

This code is used in section 47.

50 ⟨ Define symbol precedences 50 ⟩ =

$\pi_3(\Upsilon_1) \mapsto v_a$ ▷ format pointer ◁

$\pi_4(\Upsilon_1) \mapsto v_b$ ▷ stash pointer ◁

$\pi_2(\Upsilon_1) \mapsto v_c$ ▷ kind of precedence ◁

$\Upsilon \leftarrow \langle^{\text{nx}}$`\precdecls{`$\llcorner v_c \lrcorner$`}{`val Υ_2`}{`val Υ_3`}{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$

This code is used in section 47.

51 The bootstrap grammar forms the smallest subset of the full grammar.

⟨ Parser common productions 44 ⟩ +=

⟨ Parser bootstrap productions 52 ⟩

52 These are the two most important rules for the bootstrap parser.

⟨ Parser bootstrap productions 52 ⟩ =

symbol_declaration :

⟨`nterm`⟩ ◇ $symbol\_defs_1$ $\Upsilon \leftarrow \langle^{\text{nx}}$`\ntermdecls{`val Υ_3`}`val $\Upsilon_1\rangle$

⟨`token`⟩ ◇ $symbol\_defs_1$ $\Upsilon \leftarrow \langle^{\text{nx}}$`\tokendecls{`val Υ_3`}`val $\Upsilon_1\rangle$

See also sections 57, 58, 82, and 86.

This code is used in sections 22 and 51.

53 *Just like* $symbols_1$ *but accept* **int** *for the sake of* POSIX. Perhaps the only point worth mentioning here is the inserted separator (`\hspace`). Like any other separator, it takes two parameters, stream pointers. In this case, however, both pointers are null since there seems to be no other meaningful assignment. If any formatting or stash information is needed, it can be extracted by the symbols themselves.

⟨ Parser common productions 44 ⟩ +=

symbols.prec :

symbol.prec ⟨ Carry on 33 ⟩

symbols.prec symbol.prec $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\text{nx}}{}_{\sqcup}$`{0}{0}`val $\Upsilon_2\rangle$

symbol.prec :

symbol $\Upsilon \leftarrow \langle^{\text{nx}}$`\symbolprec{`val Υ_1`}{}`$\rangle$

symbol **int** $\Upsilon \leftarrow \langle^{\text{nx}}$`\symbolprec{`val Υ_1`}{`val Υ_2`}`$\rangle$

54 *One or more symbols to be* ⟨`type`⟩*'d.*

⟨ Parser common productions 44 ⟩ +=

⟨ List of symbols 55 ⟩

55 ⟨ List of symbols 55 ⟩ =

$\boldsymbol{symbols_1}$:

symbol ⟨ Carry on 33 ⟩

$symbols_1$ *symbol* $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\text{nx}}{}_{\sqcup}$`{0}{0}`val $\Upsilon_2\rangle$

This code is used in sections 22 and 54.

56 ⟨ Parser common productions 44 ⟩ +=

generic_symlist :

generic_symlist_item	⟨ Carry on 33 ⟩
generic_symlist generic_symlist_item	$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}{}_{\sqcup}\,\texttt{\{0\}\{0\}}\,\mathrm{val}\,\Upsilon_2 \rangle$

generic_symlist_item : *symbol* | *tag* ⟨ Carry on 33 ⟩

tag : `<`*tag*`>` | `<*>` | `<>` ⟨ Carry on 33 ⟩

57 *One token definition.*

⟨ Parser bootstrap productions 52 ⟩ +=

symbol_def :

`<`*tag*`>`	⟨ Carry on 33 ⟩
id | *id* **int** | *id string_as_id* | *id* **int** *string_as_id*	$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash onesymbol\{}\,\mathrm{val}\,\Upsilon_1\,\texttt{\}\{}\,\mathrm{val}\,\Upsilon_2\,\texttt{\}\{}\,\mathrm{val}\,\Upsilon_3\,\texttt{\}}\rangle$

58 *One or more symbol definitions.*

⟨ Parser bootstrap productions 52 ⟩ +=

***symbol_defs*$_1$** :

symbol_def	⟨ Carry on 33 ⟩
symbol_defs$_1$ *symbol_def*	⟨ Add a symbol definition 59 ⟩

59 ⟨ Add a symbol definition 59 ⟩ =

$\pi_2(\Upsilon_2) \mapsto v_a$ ▷ the identifier ◁

$\pi_4(v_a) \mapsto v_b$ ▷ the format pointer ◁

$\pi_5(v_a) \mapsto v_c$ ▷ the stash pointer ◁

$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1\,{}^{\mathrm{nx}}{}_{\sqcup}\,\texttt{\{}\llcorner v_b \lrcorner\texttt{\}\{}\llcorner v_c \lrcorner\texttt{\}}\,\mathrm{val}\,\Upsilon_2 \rangle$

This code is used in section 58.

60 *The grammar section: between the two* ⟨%⟩*'s.* Finally, the following few short sections define the syntax of `bison`'s rules.

⟨ Parser grammar productions 60 ⟩ =

grammar :

rules_or_grammar_declaration	⟨ Start with a production cluster 62 ⟩
grammar rules_or_grammar_declaration	⟨ Attach a productions cluster 63 ⟩

See also sections 61, 71, and 85.

This code is used in sections 21 and 24.

61 *As a* `bison` *extension, one can use the grammar declarations in the body of the grammar.* What follows is the syntax of the right hand side of a grammar rule.

⟨ Parser grammar productions 60 ⟩ +=

rules_or_grammar_declaration :

rules	⟨ Add a productions cluster 64 ⟩
grammar_declaration `;`	⟨ Carry on 33 ⟩
error `;`	`\errmessage{parsing error!}`

rules : *id_colon named_ref*$_{\mathrm{opt}}$ ◇ *rhses*$_1$ ⟨ Complete a production 65 ⟩

***rhses*$_1$** :

rhs	⟨ Start the right hand side 66 ⟩
rhses$_1$ `|`	⟨ Insert local formatting 67 ⟩
rhs	⟨ Add a right hand side to a production 68 ⟩
rhses$_1$ `;`	⟨ Add an optional semicolon 69 ⟩

62 The next few actions describe what happens when a left hand side is attached to a rule.

⟨ Start with a production cluster 62 ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a$

$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash grammar\{}\,\mathrm{val}\,\Upsilon_1\,\texttt{\}\{}\llcorner v_a \lrcorner\texttt{\}}\rangle$

This code is used in section 60.

63 ⟨ Attach a productions cluster 63 ⟩ =
$\pi_3(\Upsilon_1) \mapsto v_a$ ▷ type of the last rule ◁
$\pi_2(\Upsilon_1) \mapsto v_c$ ▷ accumulated rules ◁
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ type of the new rule ◁
let `default \positionswitchdefault`
switch $(\llcorner v_b \lrcorner)$ ε `\positionswitch` ▷ determine the position of the first token in the group ◁
def$_{\mathrm{x}}$ **next** `{`$\llcorner v_a \lrcorner$`}`
def$_{\mathrm{x}}$ `default {`$\llcorner v_b \lrcorner$`}` ▷ reuse `\default` ◁
if$_{\mathrm{x}}$ **next** `default`
  let `default \separatorswitchdefaulteq`
  switch $(\llcorner v_a \lrcorner)$ ε `\separatorswitcheq`
else
  $v_a \leftarrow v_a +_{\mathrm{s}} v_b$
  let `default \separatorswitchdefaultneq`
  switch $(\llcorner v_a \lrcorner)$ ε `\separatorswitchneq`
fi
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\grammar {`$\llcorner v_c \lrcorner$val `\postoks`$\llcorner v_d \lrcorner$val Υ_2 `}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in sections 36 and 60.

64 ⟨ Add a productions cluster 64 ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$ ▷ `\prodheader` ◁
$\pi_2(v_a) \mapsto v_b$ ▷ `\idit` ◁
$\pi_4(v_b) \mapsto v_c$ ▷ format stream pointer ◁
$\pi_5(v_b) \mapsto v_d$ ▷ stash stream pointer ◁
$\pi_3(\Upsilon_1) \mapsto v_b$ ▷ `\rules` ◁
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\oneproduction {`$\llcorner v_a \lrcorner \llcorner v_b \lrcorner$`}{`$\llcorner v_c \lrcorner$`}{`$\llcorner v_d \lrcorner$`}`$\rangle$
This code is used in section 61.

65 ⟨ Complete a production 65 ⟩ =
$\pi_4(\Upsilon_1) \mapsto v_a$ ▷ format stream pointer ◁
$\pi_5(\Upsilon_1) \mapsto v_b$ ▷ stash stream pointer ◁
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\pcluster {`$^{\mathrm{nx}}$`\prodheader {`val Υ_1 `}{`val Υ_2 `}`
`{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}}{`val Υ_4 `}}`
This code is used in section 61.

66 It is important to format the right hand side properly, since we would like to indicate that an action is inlined by an indentation. The 'format' of the `\rhs` 'structure' includes the stash pointers and a 'boolean' to indicate whether the right hand side ends with an action. Since the action can be implicit, this decision has to be postponed until, say, a semicolon is seen. No formatting or stash pointers are added for such implicit action.

⟨ Start the right hand side 66 ⟩ =
$\pi_{\vdash}(\Upsilon_1) \mapsto v_a$ $\llcorner v_a \lrcorner$
$\pi_3(\Upsilon_1) \mapsto v_b$ ▷ the format pointer ◁
$\pi_4(\Upsilon_1) \mapsto v_c$ ▷ the stash pointer ◁
if (rhs = full)
  $\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\rules {`val Υ_1 `}{`$\llcorner v_b \lrcorner$`}{`$\llcorner v_c \lrcorner$`}`$\rangle$
else ▷ it does not end with an action, fake one ◁
  $\pi_{\{\}}(\Upsilon_1) \mapsto v_a$ ▷ rules ◁
  def$_{\mathrm{x}}$ **next** `{`$\llcorner v_a \lrcorner$`}`
  if$_{\mathrm{x}}$ **next** $\varnothing$
    $v_a \leftarrow \langle \ulcorner \ldots \urcorner \rangle$
  fi
  $\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\rules {`$^{\mathrm{nx}}$`\rhs {`$\llcorner v_a \lrcorner^{\mathrm{nx}}$`\rarhssep {0}{0}`
    $^{\mathrm{nx}}$`\actbraces { }{ }{0}{0}`$^{\mathrm{nx}}$`\bdend }{ }{`$^{\mathrm{nx}}$rhs = full `}}{`$\llcorner v_b \lrcorner$`}{`$\llcorner v_c \lrcorner$`}}`
fi
This code is used in section 61.

67 ⟨ Insert local formatting 67 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto$ { Υ_0 }
$\Upsilon \leftarrow \langle \mathrm{val}\ \Upsilon_0\ ^{\mathrm{nx}}$\midf $\mathrm{val}\ \Upsilon_2 \rangle$
This code is used in section 61.

68 No pointers are provided for an *implicit* action.
⟨ Add a right hand side to a production 68 ⟩ =
$\pi_{\vdash}(\Upsilon_4) \mapsto v_a$ $\llcorner v_a \lrcorner$
if (rhs = full)
$\Upsilon \leftarrow \langle ^{\mathrm{nx}}$\rules { $\mathrm{val}\ \Upsilon_3\ ^{\mathrm{nx}}$\rrhssep $\mathrm{val}\ \Upsilon_2 \mathrm{val}\ \Upsilon_4$ } $\mathrm{val}\ \Upsilon_2 \rangle$
else
$\pi_{\{\}}(\Upsilon_4) \mapsto v_a$
def$_{\mathrm{x}}$ **next** { $\llcorner v_a \lrcorner$ }
if$_{\mathrm{x}}$ **next** ∅
$v_a \leftarrow \langle \ulcorner \ldots \urcorner \rangle$
fi
$\Upsilon \leftarrow \langle ^{\mathrm{nx}}$\rules { $\mathrm{val}\ \Upsilon_3\ ^{\mathrm{nx}}$\rrhssep $\mathrm{val}\ \Upsilon_2$
$^{\mathrm{nx}}$\rhs { $\llcorner v_a \lrcorner\ ^{\mathrm{nx}}$\rarhssep {0}{0} ▷ streams have already been grabbed ◁
$^{\mathrm{nx}}$\actbraces { }{ }{0}{0}$^{\mathrm{nx}}$\bdend }{ }{ $^{\mathrm{nx}}$rhs = full } } $\mathrm{val}\ \Upsilon_2$ }
fi
This code is used in section 61.

69 ⟨ Add an optional semicolon 69 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 61.

70 ⟨ Tokens and types for the grammar parser 26 ⟩ +=
⟨empty⟩ (PERCENT_EMPTY)

71 The centerpiece of the grammar is the syntax of the right hand side of a production. Various 'precedence hints' must be attached to an appropriate portion of the rule, just before an action (which can be inline, implicit or both in this case).
⟨ Parser grammar productions 60 ⟩ +=
rhs :

∘	⟨ Make an empty right hand side 72 ⟩
rhs symbol named_ref$_{\mathrm{opt}}$	⟨ Add a term to the right hand side 73 ⟩
rhs {...} *named_ref*$_{\mathrm{opt}}$	⟨ Add an action to the right hand side 74 ⟩
rhs %?{...}	⟨ Add a predicate to the right hand side 75 ⟩
rhs ⟨empty⟩	⟨ Add ⟨empty⟩ to the right hand side 76 ⟩
rhs ⟨prec⟩ *symbol*	⟨ Add a precedence directive to the right hand side 77 ⟩
rhs ⟨dprec⟩ **int**	⟨ Add a ⟨dprec⟩ directive to the right hand side 78 ⟩
rhs ⟨merge⟩ <*tag*>	⟨ Add a ⟨merge⟩ directive to the right hand side 79 ⟩

named_ref$_{\mathrm{opt}}$:

∘	⟨ Create an empty named reference 80 ⟩
BRACKETED_ID	⟨ Create a named reference 81 ⟩

72 ⟨ Make an empty right hand side 72 ⟩ =
$\Upsilon \leftarrow \langle ^{\mathrm{nx}}$\rhs { }{ }{ $^{\mathrm{nx}}$rhs = not full } $\rangle$
This code is used in section 71.

73 ⟨ Add a term to the right hand side 73 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
def$_{\mathrm{x}}$ **next** { $\llcorner v_b \lrcorner$ }
if$_{\mathrm{x}}$ **next** ∅

else
$\pi_4(\Upsilon_2) \mapsto v_c$
$\pi_5(\Upsilon_2) \mapsto v_d$
$v_b \leftarrow v_b +_{\mathrm{sx}} [\texttt{\{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}]$
fi
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash rhs \{}\llcorner v_a \lrcorner\llcorner v_b \lrcorner$
${}^{\mathrm{nx}}\texttt{\textbackslash termname \{}\,\mathrm{val}\,\Upsilon_2\,\texttt{\}\{}\,\mathrm{val}\,\Upsilon_3\,\texttt{\}\}\{}\,{}^{\mathrm{nx}}\sqcup\,\texttt{\}\{}\,{}^{\mathrm{nx}}\mathrm{rhs} = \mathrm{not\ full}\,\texttt{\}\}}$

This code is used in section 71.

74 ⟨ Add an action to the right hand side 74 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_b\ \llcorner v_b \lrcorner$
if (rhs = full) ▷ the first half ends with an action ◁
$v_a \leftarrow v_a +_{\mathrm{sx}} [{}^{\mathrm{nx}}\texttt{\textbackslash arhssep \{0\}\{0\}}^{\mathrm{nx}}\ulcorner \ldots \urcorner]$ ▷ no pointers to streams ◁
fi
def$_{\mathrm{x}}$ **next** $\texttt{\{}\llcorner v_a \lrcorner\texttt{\}}$
if$_{\mathrm{x}}$ **next** $\varnothing$
$v_a \leftarrow \langle \ulcorner \ldots \urcorner \rangle$
fi
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ the contents of the braced code ◁
$\pi_2(\Upsilon_2) \mapsto v_c$ ▷ the format stream pointer ◁
$\pi_3(\Upsilon_2) \mapsto v_d$ ▷ the stash stream pointer ◁
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash rhs \{}\llcorner v_a \lrcorner{}^{\mathrm{nx}}\texttt{\textbackslash rarhssep \{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}$
${}^{\mathrm{nx}}\texttt{\textbackslash actbraces \{}\llcorner v_b \lrcorner\texttt{\}\{}\,\mathrm{val}\,\Upsilon_3\,\texttt{\}\{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}{}^{\mathrm{nx}}\texttt{\textbackslash bdend }$
$\texttt{\{}\,{}^{\mathrm{nx}}\texttt{\textbackslash arhssep \}\{}\,{}^{\mathrm{nx}}\mathrm{rhs} = \mathrm{full}\,\texttt{\}\}}$

This code is used in section 71.

75 ⟨ Add a predicate to the right hand side 75 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_b\ \llcorner v_b \lrcorner$
if (rhs = full) ▷ the first half ends with an action ◁
$v_a \leftarrow v_a +_{\mathrm{sx}} [{}^{\mathrm{nx}}\texttt{\textbackslash arhssep \{0\}\{0\}}^{\mathrm{nx}}\ulcorner \ldots \urcorner]$ ▷ no pointers to streams ◁
fi
def$_{\mathrm{x}}$ **next** $\texttt{\{}\llcorner v_a \lrcorner\texttt{\}}$
if$_{\mathrm{x}}$ **next** $\varnothing$
$v_a \leftarrow \langle \ulcorner \ldots \urcorner \rangle$
fi
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ the contents of the braced code ◁
$\pi_2(\Upsilon_2) \mapsto v_c$ ▷ the format stream pointer ◁
$\pi_3(\Upsilon_2) \mapsto v_d$ ▷ the stash stream pointer ◁
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash rhs \{}\llcorner v_a \lrcorner{}^{\mathrm{nx}}\texttt{\textbackslash rarhssep \{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}$
${}^{\mathrm{nx}}\texttt{\textbackslash bpredicate \{}\llcorner v_b \lrcorner\texttt{\}\{ \}\{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}{}^{\mathrm{nx}}\texttt{\textbackslash bdend }$
$\texttt{\{}\,{}^{\mathrm{nx}}\texttt{\textbackslash arhssep \}\{}\,{}^{\mathrm{nx}}\mathrm{rhs} = \mathrm{full}\,\texttt{\}\}}$

This code is used in section 71.

76 ⟨ Add ⟨empty⟩ to the right hand side 76 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
def$_{\mathrm{x}}$ **next** $\texttt{\{}\llcorner v_b \lrcorner\texttt{\}}$
if$_{\mathrm{x}}$ **next** $\varnothing$
else
$\pi_4(\Upsilon_2) \mapsto v_c$
$\pi_5(\Upsilon_2) \mapsto v_d$
$v_b \leftarrow v_b +_{\mathrm{sx}} [\texttt{\{}\llcorner v_c \lrcorner\texttt{\}\{}\llcorner v_d \lrcorner\texttt{\}}]$
fi
$\Upsilon \leftarrow \langle {}^{\mathrm{nx}}\texttt{\textbackslash rhs \{}\llcorner v_a \lrcorner\llcorner v_b \lrcorner$
${}^{\mathrm{nx}}\ulcorner \ldots \urcorner\texttt{\}\{}\,{}^{\mathrm{nx}}\sqcup\,\texttt{\}\{}\,{}^{\mathrm{nx}}\mathrm{rhs} = \mathrm{not\ full}\,\texttt{\}\}}$

This code is used in section 71.

77 ⟨ Add a precedence directive to the right hand side 77 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c \ \llcorner v_c \lrcorner$
if (rhs = full)
$\Upsilon \leftarrow \langle^{\rm nx}$`\sprecop{` val Υ_3 `}`val $\Upsilon_2\rangle$ ▷ reuse `\yyval` ◁
`\supplybdirective` $v_a \Upsilon$ ▷ the directive is 'absorbed' by the action ◁
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = full `}`$\rangle$
else
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$
$^{\rm nx}$`\sprecop{` val Υ_3 `}`val Υ_2 `}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = not full `}}`
fi

This code is used in section 71.

78 ⟨ Add a ⟨`dprec`⟩ directive to the right hand side 78 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c \ \llcorner v_c \lrcorner$
if (rhs = full)
$\Upsilon \leftarrow \langle^{\rm nx}$`\dprecop{` val Υ_3 `}`val $\Upsilon_2\rangle$ ▷ reuse `\yyval` ◁
`\supplybdirective` $v_a \Upsilon$ ▷ the directive is 'absorbed' by the action ◁
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = full `}`$\rangle$
else
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$
$^{\rm nx}$`\dprecop{` val Υ_3 `}`val Υ_2 `}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = not full `}}`
fi

This code is used in section 71.

79 ⟨ Add a ⟨`merge`⟩ directive to the right hand side 79 ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c \ \llcorner v_c \lrcorner$
if (rhs = full)
$\Upsilon \leftarrow \langle^{\rm nx}$`\mergeop{` val Υ_3 `}`val $\Upsilon_2\rangle$ ▷ reuse `\yyval` ◁
`\supplybdirective` $v_a \Upsilon$ ▷ the directive is 'absorbed' by the action ◁
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = full `}`$\rangle$
else
$\Upsilon \leftarrow \langle^{\rm nx}$`\rhs{`$\llcorner v_a \lrcorner$
$^{\rm nx}$`\mergeop{` val Υ_3 `}`val Υ_2 `}{`$\llcorner v_b \lrcorner$`}{`$^{\rm nx}$rhs = not full `}}`
fi

This code is used in section 71.

80 ⟨ Create an empty named reference 80 ⟩ =
$\Upsilon \leftarrow \langle\rangle$

This code is used in section 71.

81 ⟨ Create a named reference 81 ⟩ =
⟨ Carry on 33 ⟩

This code is used in section 71.

82 Identifiers. *Identifiers are returned as uniqstr values by the scanner. Depending on their use, we may need to make them genuine symbols.* We, on the other hand simply copy the values returned by the scanner.

⟨ Parser bootstrap productions 52 ⟩ +=
id :

«identifier» ⟨ Turn an identifier into a term 87 ⟩
char ⟨ Turn a character into a term 88 ⟩

83 ⟨ Parser common productions 44 ⟩ +=
⟨ Definition of *symbol* 84 ⟩

84 ⟨ Definition of *symbol* 84 ⟩ =
symbol :
id ⟨ Turn an identifier into a symbol 89 ⟩
string_as_id ⟨ Turn a string into a symbol 90 ⟩
This code is used in sections 22 and 83.

85 ⟨ Parser grammar productions 60 ⟩ +=
id_colon : «identifier: » ⟨ Prepare the left hand side 91 ⟩

86 A string used as an «identifier».
⟨ Parser bootstrap productions 52 ⟩ +=
string_as_id : «`string`» ⟨ Prepare a string for use 92 ⟩

87 The remainder of the action code is trivial but we reserved the placeholders for the appropriate actions in case the parser gains some sophistication in processing low level types (or starts expecting different types from the scanner).
⟨ Turn an identifier into a term 87 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 82.

88 ⟨ Turn a character into a term 88 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 82.

89 ⟨ Turn an identifier into a symbol 89 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 84.

90 ⟨ Turn a string into a symbol 90 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 84.

91 ⟨ Prepare the left hand side 91 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 85.

92 ⟨ Prepare a string for use 92 ⟩ =
⟨ Carry on 33 ⟩
This code is used in section 86.

93 *Variable and value. The* «`string`» *form of variable is deprecated and is not* `M4`*-friendly. For example,* `M4` *fails for* `%define "[" "value"`.
⟨ Parser prologue productions 35 ⟩ +=
variable : «identifier» | «`string`» ⟨ Carry on 33 ⟩
value : ∘ | «identifier» | «`string`» | `{...}` $\Upsilon \leftarrow \langle^{\text{nx}}$`\bracedvalue` val $\Upsilon_1\rangle$

94 ⟨ Parser common productions 44 ⟩ +=
$\textit{epilogue}_{\text{opt}}$: ∘ | ⟨`%`⟩ `EPILOGUE`

95 C preamble for the grammar parser. In this case, there are no 'real' actions that our grammar performs, only TEX output, so this section is empty.

⟨ Grammar parser C preamble 95 ⟩ =

This code is used in sections 21, 22, 23, and 24.

96 C postamble for the grammar parser. It is tricky to insert function definitions that use `bison`'s internal types, as they have to be inserted in a place that is aware of the internal definitions but before said definitions are used.

⟨ Grammar parser C postamble 96 ⟩ =

```
#define YYPRINT(file, type, value) yyprint (file, type, value)
  static void yyprint(FILE *file, int type, YYSTYPE value)
  { }
```

This code is used in sections 21, 23, 24, and 97.

97 ⟨ Bootstrap parser C postamble 97 ⟩ =

```
  ⟨ Grammar parser C postamble 96 ⟩
  ⟨ Bootstrap token output 98 ⟩
```

This code is used in section 22.

98 ⟨ Bootstrap token output 98 ⟩ =

```
      void bootstrap_tokens(char *bootstrap_token_format){
#define _register_token_d(name) fprintf(tables_out, bootstrap_token_format, #name, name, #name);
          ⟨ Bootstrap token list 99 ⟩
#undef _register_token_d
          }
```

This code is used in section 97.

99 Here is the minimal list of tokens needed to make the lexer operational just enough to extract the rest of the token information from the grammar.

⟨ Bootstrap token list 99 ⟩ =

```
  _register_token_d(INT)
  _register_token_d(ID)
  _register_token_d(CHAR)
  _register_token_d(STRING)
  _register_token_d(TAG)
  _register_token_d(SEMICOLON)
  _register_token_d(PERCENT_TOKEN)
  _register_token_d(PERCENT_NTERM)
  _register_token_d(FLEX_STATE_X)
  _register_token_d(FLEX_STATE_S)
```

This code is used in section 98.

100 Union of types.

⟨ Union of grammar parser types 100 ⟩ =

This code is used in sections 21, 22, 23, and 24.

101 The scanner for grammar syntax

The fact that `bison` has a relatively straightforward grammar is due to the sophistication of its scanner. The primary reason for this increased complexity is `bison`'s awareness of syntax variations in its input files. In addition to the grammar syntax, the parser has to be able to deal with extended C syntax inside `bison`'s actions.

Since the names of the scanner *states* reside in the common namespace with other variables, in order to make the TEX version of the scanner aware of the numerical values of the states, a special procedure is

required. It is executed as part of `flex`'s user initialization code but the data for it has to be collected separately. The procedure is declared in the preamble section of the scanner.

Below, we follow the same convention (of italicizing the original comments) as in the code for the parser.

⟨ `lo.ll` 101 ⟩ =

```
⟨ Grammar lexer definitions 102 ⟩
.....
⟨ Grammar lexer C preamble 114 ⟩
.....
⟨ Grammar lexer options 115 ⟩

⟨ Grammar token regular expressions 116 ⟩
void define_all_states(void)
{
        ⟨ Collect state definitions for the grammar lexer 105 ⟩
}
```

102 It is convenient to abbreviate some commonly used subexpressions.

⟨ Grammar lexer definitions 102 ⟩ =

```
⟨ Grammar lexer states 106 ⟩
letter    [.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_]
notletter [^.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_]{-}[%\{]
id        {letter}({letter}|[-0-9])*
int       [0-9]+
```

See also sections 103 and 104.

This code is used in section 101.

103 *Zero or more instances of backslash-newline. Following GCC, allow white space between the backslash and the newline.*

⟨ Grammar lexer definitions 102 ⟩ +=

```
splice   (\\[ \f\t\v]*\n)*
```

104 *An equal sign, with optional leading whitespaces. This is used in some deprecated constructs.*

⟨ Grammar lexer definitions 102 ⟩ +=

```
eqopt    ([[:space:]]*=)?
```

105 This is how the code for state value output is put inside the routine mentioned above. The state information is collected by a special small scanner that is coupled with the bootstrap parser. This way, all the necessary token information comes 'hardwired' in the bootstrap parser, and the small scanner itself does not use any state manipulation and thus can get away without any state setup. It can, however, scan just enough of the `flex` syntax to extract the state information from it (only the state *names* are needed) and output it in the form of a header file for the 'real' lexer output 'driver' to use.

⟨ Collect state definitions for the grammar lexer 105 ⟩ =

```
#define _register_name(name) Define_State(#name, name)
#include "lo_states.h"
#undef _register_name
```

This code is used in section 101.

106 *A C-like comment in directives/rules.*

⟨ Grammar lexer states 106 ⟩ =

```
⟨states-x⟩f:  SC_YACC_COMMENT
```

See also sections 107, 108, 109, 110, 111, 112, and 113.

This code is used in section 102.

107 *Strings and characters in directives/rules.*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_ESCAPED_STRING SC_ESCAPED_CHARACTER`

108 *A identifier was just read in directives/rules. Special state to capture the sequence '*`identifier:`*'.*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_AFTER_IDENTIFIER`

109 POSIX *says that a tag must be both an id and a* C *union member, but historically almost any character is allowed in a tag. We disallow* Λ*, as this simplifies our implementation. We match angle brackets in nested pairs: several languages use them for generics/template types.*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_TAG`

110 *Four types of user code:*

- *prologue (code between* `%{ %}` *in the first section, before* ⟨`%`⟩*);*
- *actions, printers, union, etc, (between braced in the middle section);*
- *epilogue (everything after the second* ⟨`%`⟩*).*
- *predicate (code between* `%?{` *and* `}` *in middle section);*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_PROLOGUE SC_BRACED_CODE SC_EPILOGUE SC_PREDICATE`

111 C *and* C++ *comments in code.*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_COMMENT SC_LINE_COMMENT`

112 *Strings and characters in code.*

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_STRING SC_CHARACTER`

113 Bracketed identifiers support.

⟨ Grammar lexer states 106 ⟩ +=
⟨`states-x`⟩$_{\mathrm{f}}$: `SC_BRACKETED_ID SC_RETURN_BRACKETED_ID`

114 ⟨ Grammar lexer C preamble 114 ⟩ =

```
#include <stdint.h>
#include <stdbool.h>
```

This code is used in section 101.

115 The code for the generated scanner is highly dependent on the options supplied. Most of the options below are essential for the scheme adopted in this package to work.

⟨ Grammar lexer options 115 ⟩ =

```
⟨bison-bridge⟩f ⋆
⟨noyywrap⟩f ⋆
⟨nounput⟩f ⋆
⟨noinput⟩f ⋆
⟨reentrant⟩f ⋆
⟨noyy_top_state⟩f ⋆
⟨debug⟩f ⋆
⟨stack⟩f ⋆
⟨outfile⟩f              "lo.c"
```

This code is used in section 101.

116 Tokenizing with regular expressions

Here is a full collection of regular expressions employed by the scanner.

⟨ Grammar token regular expressions 116 ⟩ =
⟨ Scan grammar white space 117 ⟩
⟨ Scan `flex` directives and options 119 ⟩
⟨ Scan `bison` directives 118 ⟩
⟨ Do not support zero characters 131 ⟩
⟨ Scan after an identifier, check whether a colon is next 132 ⟩
⟨ Scan bracketed identifiers 137 ⟩
⟨ Scan a Yacc comment 144 ⟩
⟨ Scan a C comment 145 ⟩
⟨ Scan a line comment 146 ⟩
⟨ Scan a `bison` string 147 ⟩
⟨ Scan a character literal 149 ⟩
⟨ Scan a tag 151 ⟩
⟨ Decode escaped characters 154 ⟩
⟨ Scan user-code characters and strings 155 ⟩
⟨ Strings, comments etc. found in user code 156 ⟩
⟨ Scan code in braces 157 ⟩
⟨ Scan prologue 160 ⟩
⟨ Scan the epilogue 162 ⟩
⟨ Add the scanned symbol to the current string 164 ⟩

This code is used in section 101.

117 ⟨ Scan grammar white space 117 ⟩ =

```
<INITIAL,SC_AFTER_IDENTIFIER,SC_BRACKETED_ID,SC_RETURN_BRACKETED_ID>
{
    ▷ Comments and white space. ◁
  ","                {\yycomplain{stray ‘,’ treated as white space}\yylexnext}
  [ \f\n\t\v] |
    "//".*           {\yylexnext}
  "/*"               {\YYSTART \contextstate = t_a \yyBEGIN{SC_YACC_COMMENT}\yylexnext}
    ▷ #line directives are not documented, and may be withdrawn or modified in future versions of bison. ◁
  ^"#line "{int}(" \"".*"\"")?"\n" {\yylexnext}
}
```

This code is used in section 116.

118 *For directives that are also command line options, the regex must be* `"%..."` *after* `"[-_]"`*'s are removed, and the directive must match the* `--long` *option name, with a single string argument. Otherwise, add exceptions to* `../build-aux/cross-options.pl`. For most options the scanner returns a pair of pointers as the value.

⟨ Scan `bison` directives 118 ⟩ =

```
<INITIAL>
{
  "%binary"                 {\yylexreturnptr{PERCENT_NONASSOC}}
  "%code"                   {\yylexreturnptr{PERCENT_CODE}}
  "%debug"                  {⟨ Set ⟨debug⟩ flag 121 ⟩}
  "%default-prec"           {\yylexreturnptr{PERCENT_DEFAULT_PREC}}
  "%define"                {\yylexreturnptr{PERCENT_DEFINE}}
  "%defines"                {\yylexreturnptr{PERCENT_DEFINES}}
  "%destructor"             {\yylexreturnptr{PERCENT_DESTRUCTOR}}
  "%dprec"                  {\yylexreturnptr{PERCENT_DPREC}}
  "%empty"                  {\yylexreturnptr{PERCENT_EMPTY}}
  "%error-verbose"          {\yylexreturnptr{PERCENT_ERROR_VERBOSE}}
  "%expect"                 {\yylexreturnptr{PERCENT_EXPECT}}
  "%expect-rr"              {\yylexreturnptr{PERCENT_EXPECT_RR}}
  "%file-prefix"            {\yylexreturnptr{PERCENT_FILE_PREFIX}}
```

```
"%fixed-output-files"             {\yylexreturnptr{PERCENT_YACC}}
"%initial-action"                 {\yylexreturnptr{PERCENT_INITIAL_ACTION}}
"%glr-parser"                     {\yylexreturnptr{PERCENT_GLR_PARSER}}
"%language"                       {\yylexreturnptr{PERCENT_LANGUAGE}}
"%left"                           {\yylexreturnptr{PERCENT_LEFT}}
"%lex-param"                      {⟨ Return lexer parameters 122 ⟩}
"%locations"                      {⟨ Set ⟨locations⟩ flag 123 ⟩}
"%merge"                          {\yylexreturnptr{PERCENT_MERGE}}
"%name-prefix"                    {\yylexreturnptr{PERCENT_NAME_PREFIX}}
"%no-default-prec"                {\yylexreturnptr{PERCENT_NO_DEFAULT_PREC}}
"%no-lines"                       {\yylexreturnptr{PERCENT_NO_LINES}}
"%nonassoc"                       {\yylexreturnptr{PERCENT_NONASSOC}}
"%nondeterministic-parser"        {\yylexreturnptr{PERCENT_NONDETERMINISTIC_PARSER}}
"%nterm"                          {\yylexreturnptr{PERCENT_NTERM}}
"%output"                         {\yylexreturnptr{PERCENT_OUTPUT}}
"%param"                          {⟨ Return lexer and parser parameters 124 ⟩}
"%parse-param"                    {⟨ Return parser parameters 125 ⟩}
"%prec"                           {\yylexreturnptr{PERCENT_PREC}}
"%precedence"                     {\yylexreturnptr{PERCENT_PRECEDENCE}}
"%printer"                        {\yylexreturnptr{PERCENT_PRINTER}}
"%pure-parser"                    {⟨ Set ⟨pure-parser⟩ flag 126 ⟩}
"%require"                        {\yylexreturnptr{PERCENT_REQUIRE}}
"%right"                          {\yylexreturnptr{PERCENT_RIGHT}}
"%skeleton"                       {\yylexreturnptr{PERCENT_SKELETON}}
"%start"                          {\yylexreturnptr{PERCENT_START}}
"%term"                           {\yylexreturnptr{PERCENT_TOKEN}}
"%token"                          {\yylexreturnptr{PERCENT_TOKEN}}
"%token-table"                    {\yylexreturnptr{PERCENT_TOKEN_TABLE}}
"%type"                           {\yylexreturnptr{PERCENT_TYPE}}
"%union"                          {\yylexreturnptr{PERCENT_UNION}}
"%verbose"                        {\yylexreturnptr{PERCENT_VERBOSE}}
"%yacc"                           {\yylexreturnptr{PERCENT_YACC}}

  ▷ deprecated ◁
"%default"[-_]"prec"              {\yypdeprecated{\%default-prec}}
"%error"[-_]"verbose"             {\yypdeprecated{\%define parse.error verbose}}
"%expect"[-_]"rr"                 {\yypdeprecated{\%expect-rr}}
"%file-prefix"{eqopt}             {\yypdeprecated{\%file-prefix}}
"%fixed"[-_]"output"[-_]"files"   {\yypdeprecated{\%fixed-output-files}}
"%name"[-_]"prefix"{eqopt}        {\yypdeprecated{\%name-prefix}}
"%no"[-_]"default"[-_]"prec"      {\yypdeprecated{\%no-default-prec}}
"%no"[-_]"lines"                  {\yypdeprecated{\%no-lines}}
"%output"{eqopt}                  {\yypdeprecated{\%output}}
"%pure"[-_]"parser"               {\yypdeprecated{\%pure-parser}}
"%token"[-_]"table"               {\yypdeprecated{\%token-table}}

  ▷ Semantic predicate. ◁
"%?"[ \f\n\t\v]*"{"               {\yyBEGIN{SC_PREDICATE}\yylexnext}

"%"{id}|"%"{notletter}([[:graph:]])+ {⟨ Possbly complain about a bad directive 127 ⟩}

"="                               {\yylexreturnptr{EQUAL}}
"|"                               {\yylexreturnptr{PIPE}}
";"                               {\yylexreturnptr{SEMICOLON}}

{id}                              {⟨ Prepare an identifier 128 ⟩}
{int}                             {\textbf{def}$_{\text{x}}$ \textbf{next}{\yylval{}^{\text{nx}}\anint{val\yytext}}
```

```
                                        {val\yyfmark}{val\yysmark}}next
                                     \yylexreturn{INT}}
  0[xX][0-9abcdefABCDEF]+            {def_x next{\yylval{nx\hexint{val\yytext}
                                        {val\yyfmark}{val\yysmark}}next
                                     \yylexreturn{INT}}

    ▷ Identifiers may not start with a digit. Yet, don't silently accept 1FOO as 1 FOO. ◁
  {int}{id}                          {\yycomplain{invalid identifier: val\yytext}
                                     \yyerrterminate}

    ▷ Characters. ◁
  "'"                                {\yyBEGIN{SC_ESCAPED_CHARACTER}\yylexnext}

    ▷ Strings. ◁
  "\""                               {\yyBEGIN{SC_ESCAPED_STRING}\yylexnext}

    ▷ Prologue. ◁
  "%{"                               {⟨Start assembling prologue code 130⟩}

    ▷ Code in between braces. Originally preceded by \STRINGGROW but it is omitted here. ◁
  "{"                                {\lonesting 0_R \yyBEGIN{SC_BRACED_CODE}\yylexnext}

    ▷ A type. ◁
  "<*>"                              {\yylexreturnptr{TAG_ANY}}
  "<>"                               {\yylexreturnptr{TAG_NONE}}
  "<"                                {\lonesting = 0_R \yyBEGIN{SC_TAG}\yylexnext}

  "%%"                               {⟨Switch sections 129⟩}
  "["                                {let \bracketedidstr = ∅ \YYSTART
                                     \bracketedidcontextstate = t_a
                                     \yyBEGIN{SC_BRACKETED_ID}\yylexnext}

  <<EOF>>                            {\yyterminate% EOF in INITIAL}

  [^\[%A-Za-z0-9_<>{}\"\'*;|=/, \f\n\t\v]+|. {⟨Process a bad character 120⟩}
}
```

This code is used in section 116.

119 Some additional constructs needed to typeset simple `flex` declarations. This is not part of the original `bison` scanner.

⟨Scan `flex` directives and options 119⟩ =

```
<INITIAL>
{
  "%option"                          {\yylexreturnptr{FLEX_OPTION}}
  "%x"                               {\yylexreturnptr{FLEX_STATE_X}}
  "%s"                               {\yylexreturnptr{FLEX_STATE_S}}
}
```

This code is used in section 116.

120 We present the 'bad character' code first, before going into the details of the character matching by the rest of the lexer.

⟨Process a bad character 120⟩ =

```
def_x next{nx\csname val\yytextpure nx\endcsname}
\expandafter v_a \expandafter \expandafter \expandafter{next}
\expandafter if_x ⌞v_a⌟○
    if_t [bad char]
```

```
        \yycomplain{invalid character(s): val\yytext}
    fi
    \yylexreturn{$undefined}
  else
    \expandafter\lexspecialchar\expandafter{␣$v_a$␣}{val\yyfmark}{val\yysmark}\yylexnext
  fi
```

This code is used in section 118.

121 ⟨ Set ⟨debug⟩ flag 121 ⟩ =

```
  def_x next{\yylval{{parse.trace}{debug}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_FLAG}
```

This code is used in section 118.

122 ⟨ Return lexer parameters 122 ⟩ =

```
  def_x next{\yylval{{lex-param}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_PARAM}
```

This code is used in section 118.

123 ⟨ Set ⟨locations⟩ flag 123 ⟩ =

```
  def_x next{\yylval{{locations}{}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_FLAG}
```

This code is used in section 118.

124 ⟨ Return lexer and parser parameters 124 ⟩ =

```
  def_x next{\yylval{{both-param}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_PARAM}
```

This code is used in section 118.

125 ⟨ Return parser parameters 125 ⟩ =

```
  def_x next{\yylval{{parse-param}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_PARAM}
```

This code is used in section 118.

126 ⟨ Set ⟨pure-parser⟩ flag 126 ⟩ =

```
  def_x next{\yylval{{api.pure}{pure-parser}{val\yyfmark}{val\yysmark}}}next
  \yylexreturn{PERCENT_FLAG}
```

This code is used in section 118.

127 ⟨ Possbly complain about a bad directive 127 ⟩ =

```
  if_t [bad char]
      \yycomplain{invalid directive: val\yytext}
  fi
  \yylexnext
```

This code is used in section 118.

128 ⟨ Prepare an identifier 128 ⟩ =

```
  def_x next{\yylval{^nx\idit{val\yytextpure}{val\yytext}
      {val\yyfmark}{val\yysmark}}}next
  let \bracketedidstr = ∅
  \yyBEGIN{SC_AFTER_IDENTIFIER}\yylexnext
```

This code is used in section 118.

129 ⟨ Switch sections 129 ⟩ =

```
add \percentpercentcount 1R
if_ω \percentpercentcount = 2R
    \yyBEGIN{SC_EPILOGUE}
fi
\yylexreturnptr{PERCENT_PERCENT}
```

This code is used in section 118.

130 ⟨ Start assembling prologue code 130 ⟩ =

```
def_x next {\postoks{{val\yyfmark}{val\yysmark}}}next
\yyBEGIN{SC_PROLOGUE}\yylexnext
```

This code is used in section 118.

131 *Supporting* `\0` *complexifies our implementation for no expected added value.*

⟨ Do not support zero characters 131 ⟩ =

```
<SC_ESCAPED_CHARACTER,SC_ESCAPED_STRING,SC_TAG>
{
  \0                                  {\yycomplain{invalid null character}\yylexnext}
}
```

This code is used in section 116.

132 ⟨ Scan after an identifier, check whether a colon is next 132 ⟩ =

```
<SC_AFTER_IDENTIFIER>
{
  "["                                 {⟨ Process the bracketed part of an identifier 133 ⟩}
  ":"                                 {⟨ Process a colon after an identifier 134 ⟩}
  <<EOF>>                             {⟨ End the scan with an identifier 136 ⟩}
  .                                   {⟨ Process a character after an identifier 135 ⟩}
}
```

This code is used in section 116.

133 ⟨ Process the bracketed part of an identifier 133 ⟩ =

```
if_x \bracketedidstr∅
    \YYSTART \bracketedidcontextstate t_a \yyBEGIN{SC_BRACKETED_ID}
    let next = \yylexnext
else
    \ROLLBACKCURRENTTOKEN
    \yyBEGIN{SC_RETURN_BRACKETED_ID}
    def next {\yylexreturn{ID}}
fi
next
```

This code is used in section 132.

134 ⟨ Process a colon after an identifier 134 ⟩ =

```
if_x \bracketedidstr∅
    \yyBEGIN{INITIAL}
else
    \yyBEGIN{SC_RETURN_BRACKETED_ID}
fi
\yylexreturn{ID_COLON}
```

This code is used in section 132.

135 ⟨ Process a character after an identifier 135 ⟩ =

```
\ROLLBACKCURRENTTOKEN
if_x \bracketedidstr∅
    \yyBEGIN{INITIAL}
```

```
else
    \yyBEGIN{SC_RETURN_BRACKETED_ID}
fi
\yylexreturn{ID}
```

This code is used in section 132.

136 ⟨ End the scan with an identifier 136 ⟩ =

```
if_x \bracketedidstr ∅
    \yyBEGIN{INITIAL}
else
    \yyBEGIN{SC_RETURN_BRACKETED_ID}
fi
\ROLLBACKCURRENTTOKEN
\yylexreturn{ID}
```

This code is used in section 132.

137 ⟨ Scan bracketed identifiers 137 ⟩ =

```
<SC_BRACKETED_ID>
{
  <<EOF>>                          {⟨ Complain about unexpected end of file inside brackets 141 ⟩}
  {id}                             {⟨ Process bracketed identifier 138 ⟩}
  "]"                              {⟨ Finish processing bracketed identifier 139 ⟩}
  [^\].A-Za-z0-9_/ \f\n\t\v]+|.    {⟨ Complain about improper identifier characters 140 ⟩}
}
```

See also section 142.

This code is used in section 116.

138 ⟨ Process bracketed identifier 138 ⟩ =

```
if_x \bracketedidstr ∅
    def_x \bracketedidstr{nx\idit{val\yytextpure }
        {val\yytext }{val\yyfmark }{val\yysmark }}
    let next = \yylexnext
else
    def next {\yycomplain{unexpected
        identifier in bracketed name: val\yytext }\yylexnext }
fi
next
```

This code is used in section 137.

139 ⟨ Finish processing bracketed identifier 139 ⟩ =

```
\yyBEGINr \bracketedidcontextstate
if_x \bracketedidstr ∅
    def next {\yycomplain{an identifier expected}\yylexnext }
else
    if_ω \bracketedidcontextstate = \yylexstate{INITIAL}∘
        \expandafter\yylval \expandafter{\bracketedidstr }
        let \bracketedidstr = ∅
        def next {\yylexreturn{BRACKETED_ID}}
    else
        let next = \yylexnext
    fi
fi
next
```

This code is used in section 137.

140 ⟨ Complain about improper identifier characters 140 ⟩ =

```
\yycomplain{invalid character(s) in bracketed name: val\yytext}\yyerrterminate
```

This code is used in section 137.

141 ⟨ Complain about unexpected end of file inside brackets 141 ⟩ =

```
\yyBEGINr\bracketedidcontextstate
\yycomplain{unexpected end of file inside brackets}\yyerrterminate
```

This code is used in section 137.

142 ⟨ Scan bracketed identifiers 137 ⟩ +=

```
<SC_RETURN_BRACKETED_ID>
{
  .                                   {⟨ Return a bracketed identifier 143 ⟩}
}
```

143 ⟨ Return a bracketed identifier 143 ⟩ =

```
\ROLLBACKCURRENTTOKEN
\expandafter\yylval\expandafter{\bracketedidstr}
let \bracketedidstr = ∅
\yyBEGIN{INITIAL}
\yylexreturn{BRACKETED_ID}
```

This code is used in section 142.

144 *Scanning a Yacc comment. The initial* `/*` *is already eaten.*

⟨ Scan a Yacc comment 144 ⟩ =

```
<SC_YACC_COMMENT>
{
  <<EOF>>                             {\yycomplain{unexpected end of file in
                                          a comment}\yyerrterminate}
  "*/"                                {\yyBEGINr{\contextstate}\yylexnext}
  .|\n                                {\yylexnext}
}
```

This code is used in section 116.

145 *Scanning a* C *comment. The initial* `/*` *is already eaten.*

⟨ Scan a C comment 145 ⟩ =

```
<SC_COMMENT>
{
  <<EOF>>                             {\yycomplain{unexpected end of file in
                                          a comment}\yyerrterminate}
  "*"{splice}"/"                      {\STRINGGROW\yyBEGINr\contextstate\yylexnext}
}
```

This code is used in section 116.

146 *Scanning a line comment. The initial* `//` *is already eaten.*

⟨ Scan a line comment 146 ⟩ =

```
<SC_LINE_COMMENT>
{
  <<EOF>>                             {\yyBEGINr\contextstate \ROLLBACKCURRENTTOKEN
                                          \yylexnext}
  "\n"                                {\STRINGGROW\yyBEGINr\contextstate \yylexnext}
  {splice}                            {\STRINGGROW\yylexnext}
}
```

This code is used in section 116.

147 *Scanning a* `bison` *string, including its escapes. The initial quote is already eaten.*

⟨ Scan a `bison` string 147 ⟩ =

```
<SC_ESCAPED_STRING>
{
  <<EOF>>                          {\yycomplain{unexpected end of file in
                                       a string}\yyerrterminate}
  "\""                             {⟨ Finish a bison string 148 ⟩}
  "\n"                             {\yycomplain{unexpected end of line in
                                       a string}\yyerrterminate}
}
```

This code is used in section 116.

148 ⟨ Finish a `bison` string 148 ⟩ =

```
\STRINGFINISH
def_x next {\yylval{^nx\stringify{val\laststring}
{val\laststringraw}{val\yyfmark}{val\yysmark}}}next
\yyBEGIN{INITIAL}
\yylexreturn{STRING}
```

This code is used in section 147.

149 *Scanning a* `bison` *character literal, decoding its escapes. The initial quote is already eaten.*

⟨ Scan a character literal 149 ⟩ =

```
<SC_ESCAPED_CHARACTER>
{
  <<EOF>>                          {\yycomplain{unexpected end of file in
                                       a literal}\yyerrterminate}
  "'"                              {⟨ Return an escaped character 150 ⟩}
  "\n"                             {\yycomplain{unexpected end of line in
                                       a literal}\yyerrterminate}
}
```

This code is used in section 116.

150 ⟨ Return an escaped character 150 ⟩ =

```
\STRINGFINISH
def_x next {\yylval{^nx\charit{val\laststring}{val\laststringraw}
    {val\yyfmark}{val\yysmark}}}next
\STRINGFREE
\yyBEGIN{INITIAL}
\yylexreturn{CHAR}
```

This code is used in section 149.

151 *Scanning a tag. The initial angle bracket is already eaten.*

⟨ Scan a tag 151 ⟩ =

```
<SC_TAG>
{
  ">"                              {⟨ Finish a tag 152 ⟩}
  ([^<>]|->)+                      {\STRINGGROW\yylexnext}
  "<"                              {⟨ Raise nesting level 153 ⟩}
  <<EOF>>                          {\yycomplain{unexpected end of file in
                                       a literal}\yyerrterminate}
}
```

This code is used in section 116.

152 ⟨ Finish a tag 152 ⟩ =

```
add \lonesting −1R
if_ω \lonesting < 0R
    \STRINGFINISH
    def_x next { \yylval { nx\tagit { val \laststring }{ val \laststringraw }
        { val \yyfmark }{ val \yysmark } } }next
    \STRINGFREE
    \yyBEGIN { INITIAL }
    def next { \yylexreturn { TAG } }
else
    \STRINGGROW let next = \yylexnext
fi
next
```

This code is used in section 151.

153 This is a slightly different rule from the original scanner. We do not perform *yyleng* computations, so it makes sense to raise the nesting level one by one.

⟨ Raise nesting level 153 ⟩ =

```
\STRINGGROW
add \lonesting 1R
\yylexnext
```

This code is used in section 151.

154 ⟨ Decode escaped characters 154 ⟩ =

```
<SC_ESCAPED_STRING,SC_ESCAPED_CHARACTER>
{
  \\[0-7]{1,3}                      {\STRINGGROW \yylexnext }
  \\x[0-9abcdefABCDEF]+             {\STRINGGROW \yylexnext }
  \\a                               {\STRINGGROW \yylexnext }
  \\b                               {\STRINGGROW \yylexnext }
  \\f                               {\STRINGGROW \yylexnext }
  \\n                               {\STRINGGROW \yylexnext }
  \\r                               {\STRINGGROW \yylexnext }
  \\t                               {\STRINGGROW \yylexnext }
  \\v                               {\STRINGGROW \yylexnext }

    ▷ \\[\"\'?\\] would be shorter, but it confuses xgettext. ◁
  \\("\""|"'"|"?"|"\\")             {\STRINGGROW \yylexnext }

  \\(u|U[0-9abcdefABCDEF]{4})[0-9abcdefABCDEF]{4} {\STRINGGROW \yylexnext }
  \\(.|\n)                          {\yycomplain { invalid character after
                                        \\-escape: val \yytext }\yylexnext }
}
```

This code is used in section 116.

155 ⟨ Scan user-code characters and strings 155 ⟩ =

```
<SC_CHARACTER,SC_STRING>
{
  {splice}|\\{splice}[^\n\[\]]      {\STRINGGROW \yylexnext }
}

<SC_CHARACTER>
{
  "'"                               {\STRINGGROW  \yyBEGINr { \contextstate }\yylexnext }
  \n                                {\yycomplain { unexpected end of line instead of
                                        a character }\yyerrterminate }
```

```
  <<EOF>>                          {\yycomplain{unexpected end of file instead of
                                       a character}\yyerrterminate}
}

<SC_STRING>
{
  "\""                             {\STRINGGROW \yyBEGINr{\contextstate}\yylexnext}
  \n                               {\yycomplain{unexpected end of line instead of
                                       a character}\yyerrterminate}
  <<EOF>>                          {\yycomplain{unexpected end of file instead of
                                       a character}\yyerrterminate}
}
```

This code is used in section 116.

156 ⟨ Strings, comments etc. found in user code 156 ⟩ =

```
<SC_BRACED_CODE,SC_PROLOGUE,SC_EPILOGUE,SC_PREDICATE>
{
  "'"                              {\STRINGGROW \YYSTART \contextstate t_a
                                       \yyBEGIN{SC_CHARACTER}\yylexnext}
  "\""                             {\STRINGGROW \YYSTART \contextstate t_a
                                       \yyBEGIN{SC_STRING}\yylexnext}
  "/"{splice}"*"                   {\STRINGGROW \YYSTART \contextstate t_a
                                       \yyBEGIN{SC_COMMENT}\yylexnext}
  "/"{splice}"/"                   {\STRINGGROW \YYSTART \contextstate t_a
                                       \yyBEGIN{SC_LINE_COMMENT}\yylexnext}
}
```

This code is used in section 116.

157 *Scanning some code in braces (actions, predicates). The initial* `{` *is already eaten.*

⟨ Scan code in braces 157 ⟩ =

```
<SC_BRACED_CODE,SC_PREDICATE>
{
  "{"|"<"{splice}"%"               {\STRINGGROW  add \lonesting 1_R  \yylexnext}
  "%"{splice}">"                   {\STRINGGROW  add \lonesting −1_R  \yylexnext}

    ▷ Tokenize <<% correctly (as << %) rather than incorrectly (as < <%). ◁
  "<"{splice}"<"                   {\STRINGGROW \yylexnext}
  <<EOF>>                          {\yycomplain{unexpected end of line
                                       inside braced code}\yyerrterminate}
}

<SC_BRACED_CODE>
{
  "}"                              {⟨ Add closing brace to the braced code 158 ⟩}
}

<SC_PREDICATE>
{
  "}"                              {⟨ Add closing brace to a predicate 159 ⟩}
}
```

This code is used in section 116.

158 Unlike the original lexer, we do not return the closing brace as part of the braced code.

⟨ Add closing brace to the braced code 158 ⟩ =

```
add \lonesting −1_R
if_ω \lonesting < 0_R
```

```
    \STRINGFINISH
    def_x next {\yylval{{val \laststring }{val \yyfmark }{val \yysmark }}}next
    def next {\yylexreturn{BRACED_CODE}}
    \yyBEGIN{INITIAL}
  else
    \STRINGGROW
    let next =\yylexnext
  fi
  next
```

This code is used in section 157.

159 ⟨ Add closing brace to a predicate 159 ⟩ =

```
  add \lonesting -1_R
  if_ω \lonesting<0_R
    \STRINGFINISH
    def_x next {\yylval{{val \laststring }{val \yyfmark }{val \yysmark }}}next
    \yyBEGIN{INITIAL}
    def next {\yylexreturn{BRACED_PREDICATE}}
  else
    \STRINGGROW
    let next =\yylexnext
  fi
  next
```

This code is used in section 157.

160 *Scanning some prologue: from* `%{` *(already scanned) to* `%}`.

⟨ Scan prologue 160 ⟩ =

```
  <SC_PROLOGUE>
  {
    "%}"                          {⟨ Finish braced code 161 ⟩}
    <<EOF>>                       {\yycomplain{unexpected end of file
                                       inside prologue}\yyerrterminate}
  }
```

This code is used in section 116.

161 ⟨ Finish braced code 161 ⟩ =

```
  \STRINGFINISH
  def_x next {\yylval{{val \laststring }val \postoks{val \yyfmark }{val \yysmark }}}next
  \yyBEGIN{INITIAL}
  \yylexreturn{PROLOGUE}
```

This code is used in section 160.

162 *Scanning the epilogue (everything after the second ⟨*`%`*⟩, which has already been eaten).*

⟨ Scan the epilogue 162 ⟩ =

```
  <SC_EPILOGUE>
  {
    <<EOF>>                       {⟨ Handle end of file in the epilogue 163 ⟩}
  }
```

This code is used in section 116.

163 ⟨ Handle end of file in the epilogue 163 ⟩ =

```
  \ROLLBACKCURRENTTOKEN
  \STRINGFINISH
  \yylval =\laststring
  \yyBEGIN{INITIAL}
  \yylexreturn{EPILOGUE}
```

This code is used in section 162.

164 *By default, grow the string obstack with the input.*

⟨ Add the scanned symbol to the current string 164 ⟩ =

```
<SC_COMMENT,SC_LINE_COMMENT,SC_BRACED_CODE,SC_PREDICATE,SC_PROLOGUE,SC_EPILOGUE,
 SC_STRING,SC_CHARACTER,SC_ESCAPED_STRING,SC_ESCAPED_CHARACTER>. |
 <SC_COMMENT,SC_LINE_COMMENT,SC_BRACED_CODE,SC_PREDICATE,
 SC_PROLOGUE,SC_EPILOGUE>\n          {\STRINGGROW \yylexnext}
```

This code is used in section 116.

165 The name parser

What follows is an example parser for the name processing. This approach (i.e. using a 'full blown' parser/scanner combination) is probably not the best way to implement such machinery but its main purpose is to demonstrate a way to create a separate parser for local purposes.

⟨ `small_parser.yy` 165 ⟩ =

```
.....
⟨ Name parser C preamble 186 ⟩
.....
⟨ Bison options 166 ⟩
⟨union⟩      ⟨ Union of parser types 188 ⟩
.....
⟨ Name parser C postamble 187 ⟩
.....
⟨ Token and types declarations 167 ⟩

⟨ Parser productions 168 ⟩
```

166 ⟨ Bison options 166 ⟩ =

```
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆   (set as ⟨debug⟩)
⟨start⟩           full_name
```

This code is used in section 165.

167 ⟨ Token and types declarations 167 ⟩ =

```
%[a...Z0...9]*   ⟨auto⟩
[a...Z0...9]*    ⟨auto⟩
opt              ⟨auto⟩
na               ⟨auto⟩
[0...9]*         ⟨auto⟩
ext              ⟨auto⟩
```

This code is used in section 165.

168 ⟨ Parser productions 168 ⟩ =

```
full_name :
    identifier_string suffixes_opt          ⟨ Compose the full name 169 ⟩
identifier_string :
    %[a...Z0...9]*                          ⟨ Attach option name 170 ⟩
    [a...Z0...9]*                           ⟨ Start with an identifier 171 ⟩
    < [a...Z0...9]* >                       ⟨ Start with a tag 172 ⟩
    qualifier                               ⟨ Turn a qualifier into an identifier 173 ⟩
    identifier_string [a...Z0...9]*         ⟨ Attach an identifier 174 ⟩
    identifier_string qualifier             ⟨ Attach qualifier to a name 175 ⟩
    identifier_string [0...9]*              ⟨ Attach an integer 176 ⟩
suffixes_opt :
    ○                                       ϒ ← ⟨⟩
    .                                       ϒ ← ⟨nx\dotsp nx\sfxnone⟩
    . suffixes                              ⟨ Attach suffixes 177 ⟩
    . qualified_suffixes                    ⟨ Attach qualified suffixes 178 ⟩
suffixes :
```

[a...Z0...9]*
[0...9]*
suffixes .
suffixes [a...Z0...9]*
suffixes [0...9]*
qualifier .
suffixes qualifier .

qualified_suffixes :
suffixes qualifier
qualifier

qualifier :
`opt`
`na`
`ext`

This code is used in section 165.

169 ⟨ Compose the full name 169 ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \mathrm{val}\,\Upsilon_2 \rangle$ `\namechars` Υ
This code is used in section 168.

170 ⟨ Attach option name 170 ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\optstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in section 168.

171 ⟨ Start with an identifier 171 ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$`\idstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in sections 168 and 173.

172 ⟨ Start with a tag 172 ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle$`\idstr{<`$\llcorner v_a \lrcorner$`>}{<`$\llcorner v_b \lrcorner$`>}`$\rangle$
This code is used in section 168.

173 ⟨ Turn a qualifier into an identifier 173 ⟩ =
⟨ Start with an identifier 171 ⟩
This code is used in section 168.

174 ⟨ Attach an identifier 174 ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$
$v_a \leftarrow v_a +_{\mathrm{sx}} [\ \sqcup\ ]$
$\pi_1(\Upsilon_2) \mapsto v_b$
$v_a \leftarrow v_a +_{\mathrm{s}} v_b$
$\pi_3(\Upsilon_1) \mapsto v_b$
$v_b \leftarrow v_b +_{\mathrm{sx}} [\ \sqcup\ ]$
$\pi_2(\Upsilon_2) \mapsto v_c$
$v_b \leftarrow v_b +_{\mathrm{s}} v_c$
$\Upsilon \leftarrow \langle$`\idstr{`$\llcorner v_a \lrcorner$`}{`$\llcorner v_b \lrcorner$`}`$\rangle$
This code is used in sections 168, 175, and 176.

⟨ Start with a named suffix 179 ⟩
⟨ Start with a numeric suffix 180 ⟩
⟨ Add a dot separator 181 ⟩
⟨ Attach a named suffix 183 ⟩
⟨ Attach integer suffix 182 ⟩
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\sfxn` $\mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\dotsp`$\rangle$
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\sfxn` $\mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\dotsp`$\rangle$

⟨ Attach a qualifier 184 ⟩
⟨ Start suffixes with a qualifier 185 ⟩

$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1 \rangle$

175 ⟨ Attach qualifier to a name 175 ⟩ =
⟨ Attach an identifier 174 ⟩
This code is used in section 168.

176 ⟨ Attach an integer 176 ⟩ =
⟨ Attach an identifier 174 ⟩
This code is used in section 168.

177 ⟨ Attach suffixes 177 ⟩ =
$\Upsilon \leftarrow \langle {}^{\text{nx}}$`\dotsp` val $\Upsilon_2 \rangle$
This code is used in sections 168 and 178.

178 ⟨ Attach qualified suffixes 178 ⟩ =
⟨ Attach suffixes 177 ⟩
This code is used in section 168.

179 ⟨ Start with a named suffix 179 ⟩ =
$\Upsilon \leftarrow \langle {}^{\text{nx}}$`\sfxn` val $\Upsilon_1 \rangle$
This code is used in section 168.

180 ⟨ Start with a numeric suffix 180 ⟩ =
$\Upsilon \leftarrow \langle {}^{\text{nx}}$`\sfxi` val $\Upsilon_1 \rangle$
This code is used in section 168.

181 ⟨ Add a dot separator 181 ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1 {}^{\text{nx}}$`\dotsp` $\rangle$
This code is used in section 168.

182 ⟨ Attach integer suffix 182 ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1 {}^{\text{nx}}$`\sfxi` val $\Upsilon_2 \rangle$
This code is used in section 168.

183 ⟨ Attach a named suffix 183 ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1 {}^{\text{nx}}$`\sfxn` val $\Upsilon_2 \rangle$
This code is used in section 168.

184 ⟨ Attach a qualifier 184 ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1 {}^{\text{nx}}$`\qual` val $\Upsilon_2 \rangle$
This code is used in section 168.

185 ⟨ Start suffixes with a qualifier 185 ⟩ =
$\Upsilon \leftarrow \langle {}^{\text{nx}}$`\qual` val $\Upsilon_1 \rangle$
This code is used in section 168.

186 C preamble. In this case, there are no 'real' actions that our grammar performs, only TEX output, so this section is empty.

⟨ Name parser C preamble 186 ⟩ =
This code is used in section 165.

187 C postamble. It is tricky to insert function definitions that use `bison`'s internal types, as they have to be inserted in a place that is aware of the internal definitions but before said definitions are used.

⟨ Name parser C postamble 187 ⟩ =

```
#define YYPRINT(file, type, value) yyprint(file, type, value)
  static void yyprint(FILE *file, int type, YYSTYPE value)
  { }
```

This code is used in section 165.

188 Union of types.

⟨ Union of parser types 188 ⟩ =

This code is used in section 165.

189 The name scanner

```
⟨ small_lexer.ll 189 ⟩ =
  ⟨ Lexer definitions 190 ⟩
  ..............................
  ⟨ Lexer C preamble 193 ⟩
  ..............................
  ⟨ Lexer options 194 ⟩

  ⟨ Regular expressions 195 ⟩
   void define_all_states(void)
  {
        ⟨ Collect all state definitions 191 ⟩
  }
```

190 ⟨ Lexer definitions 190 ⟩ =

```
  ⟨ Lexer states 192 ⟩
  letter     [_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
  id         {letter}({letter}|[-0-9])*
  int        [0-9]+
```

This code is used in section 189.

191 ⟨ Collect all state definitions 191 ⟩ =

```
#define _register_name(name) Define_State(#name, name)    ▷ nothing for now ◁
#undef _register_name
```

This code is used in section 189.

192 Strings and characters in directives/rules.

⟨ Lexer states 192 ⟩ =

```
  ⟨states-x⟩f:  SC_ESCAPED_STRING SC_ESCAPED_CHARACTER
```

This code is used in section 190.

193 ⟨ Lexer C preamble 193 ⟩ =

```
#include <stdint.h>
#include <stdbool.h>
```

This code is used in section 189.

194 ⟨ Lexer options 194 ⟩ =

```
  ⟨bison-bridge⟩f ⋆
  ⟨noyywrap⟩f ⋆
  ⟨nounput⟩f ⋆
  ⟨noinput⟩f ⋆
  ⟨reentrant⟩f ⋆
  ⟨noyy_top_state⟩f ⋆
  ⟨debug⟩f ⋆
  ⟨stack⟩f ⋆
  ⟨outfile⟩f              "small_lexer.c"
```

This code is used in section 189.

195 ⟨ Regular expressions 195 ⟩ =

```
  ⟨ Scan white space 196 ⟩
  ⟨ Scan identifiers 197 ⟩
```

This code is used in section 189.

196 White space skipping.

⟨ Scan white space 196 ⟩ =

```
[ \f\n\t\v]                    {\yylexnext}
```

This code is used in section 195.

197 This collection of regular expressions might seem redundant, and in its present state, it certainly is. However, if later on the typesetting style for some of the keywords would need to be adjusted, such changes would be easy to implement, since the template is already here.

⟨ Scan identifiers 197 ⟩ =

```
"%binary"                      {\yylexreturnval{PERCENT_IDENTIFIER}}
"%code"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%debug"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%default-prec"                {\yylexreturnval{PERCENT_IDENTIFIER}}
"%define"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%defines"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%destructor"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%dprec"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%empty"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%error-verbose"               {\yylexreturnval{PERCENT_IDENTIFIER}}
"%expect"                      {\yylexreturnval{PERCENT_IDENTIFIER}}
"%expect-rr"                   {\yylexreturnval{PERCENT_IDENTIFIER}}
"%file-prefix"                 {\yylexreturnval{PERCENT_IDENTIFIER}}
"%fixed-output-files"          {\yylexreturnval{PERCENT_IDENTIFIER}}
"%initial-action"              {\yylexreturnval{PERCENT_IDENTIFIER}}
"%glr-parser"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%language"                    {\yylexreturnval{PERCENT_IDENTIFIER}}
"%left"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%lex-param"                   {\yylexreturnval{PERCENT_IDENTIFIER}}
"%locations"                   {\yylexreturnval{PERCENT_IDENTIFIER}}
"%merge"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%name-prefix"                 {\yylexreturnval{PERCENT_IDENTIFIER}}
"%no-default-prec"             {\yylexreturnval{PERCENT_IDENTIFIER}}
"%no-lines"                    {\yylexreturnval{PERCENT_IDENTIFIER}}
"%nonassoc"                    {\yylexreturnval{PERCENT_IDENTIFIER}}
"%nondeterministic-parser"     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%nterm"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%output"                      {\yylexreturnval{PERCENT_IDENTIFIER}}
"%param"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%parse-param"                 {\yylexreturnval{PERCENT_IDENTIFIER}}
"%prec"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%precedence"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%printer"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%pure-parser"                 {\yylexreturnval{PERCENT_IDENTIFIER}}
"%require"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%right"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%skeleton"                    {\yylexreturnval{PERCENT_IDENTIFIER}}
"%start"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%term"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%token"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%token-table"                 {\yylexreturnval{PERCENT_IDENTIFIER}}
"%type"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%union"                       {\yylexreturnval{PERCENT_IDENTIFIER}}
"%verbose"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%yacc"                        {\yylexreturnval{PERCENT_IDENTIFIER}}
"%default"[-_]"prec"           {\yylexreturnval{PERCENT_IDENTIFIER}}
"%error"[-_]"verbose"          {\yylexreturnval{PERCENT_IDENTIFIER}}
```

```
"%expect"[-_]"rr"                    {\yylexreturnval{PERCENT_IDENTIFIER}}
"%fixed"[-_]"output"[-_]"files"      {\yylexreturnval{PERCENT_IDENTIFIER}}
"%name"[-_]"prefix"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%no"[-_]"default"[-_]"prec"         {\yylexreturnval{PERCENT_IDENTIFIER}}
"%no"[-_]"lines"                     {\yylexreturnval{PERCENT_IDENTIFIER}}
"%pure"[-_]"parser"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%token"[-_]"table"                  {\yylexreturnval{PERCENT_IDENTIFIER}}
"%"({letter}|[0-9]|[-_]|"%"|[<>])+   {\yylexreturnval{PERCENT_IDENTIFIER}}

"opt"                                {\yylexreturnval{OPTIONAL}}
"na"                                 {\yylexreturnval{NO_ATTR}}
"ext"                                {\yylexreturnval{EXTENDED}}

[<>._]                               {\yylexreturnchar}

{id}                                 {⟨ Prepare to process an identifier 198 ⟩}
{int}                                {\yylexreturnval{INTEGER}}

.                                    {⟨ React to a bad character 199 ⟩}
```

This code is used in section 195.

198 ⟨ Prepare to process an identifier 198 ⟩ =

```
\yylexreturnval{IDENTIFIER}
```

This code is used in section 197.

199 ⟨ React to a bad character 199 ⟩ =

```
if_t [bad char]
    \yycomplain{invalid character(s): val\yytext }
fi
\yylexreturn{$undefined}
```

This code is used in section 197.

200 Forcing bison and flex to output TEX

Instead of implementing a `bison` (or `flex`) 'plugin' for outputting TEX parser, the code that follows produces a separate executable that outputs all the required tables after the inclusion of an ordinary C parser produced by `bison` (or a scanner produced by `flex`). The actions in both `bison` parser and `flex` scanner are assumed to be merely *printf*( ) statements that output the 'real' TEX actions. The code below simply cycles through all such actions to output an 'action switch' appropriate for use with TEX. In every other respect, the included parser or scanner can use any features allowed in 'real' parsers and scanners.

201 Common routines

The 'top' level of the scanner and parser 'drivers' is very similar, and is therefore separated into a few sections that are common to both drivers. The layout is fairly typical and follows a standard 'initialize-input-process-output-clean up' scheme. The logic behind each section of the program will be explained in detail below.

The section below is called ⟨ C postamble 201 ⟩ because the output of the tables can happen only after the `bison` (or `flex`) generated `.c` file is included and all the data structures are known.

The actual 'assembly' of each driver has to be done separately due to some 'singularities' of the `CWEB` system and the design of this software. All the essential routines are presented in the sections below, though.

```
⟨ C postamble 201 ⟩ =
  ⟨ Outer definitions 203 ⟩;
  ⟨ Global variables and types 211 ⟩
  ⟨ Auxiliary function declarations 235 ⟩
  ⟨ Auxiliary function definitions 236 ⟩
  int main(int argc, char **argv)
  {
    ⟨ Local variable and type declarations 207 ⟩
    ⟨ Establish defaults 239 ⟩
    ⟨ Command line processing variables 242 ⟩
    ⟨ Process command line options 243 ⟩
    switch (mode) {
      ⟨ Various output modes 202 ⟩
    default: break;
    }
    if (tables_out) {
      ⟨ Perform output 213 ⟩
      ⟨ Output action switch, if any 232 ⟩
    }
    else {
      fprintf(stderr, "No output, exiting\n");
      exit(0);
    }
    ⟨ Clean up 206 ⟩
    return 0;
  }
```

This code is cited in section 201.

202 Not all the code can be supplied at this stage (most of the routines here are at the 'top' level so the specifics have to be 'filled-in' by each driver), so many of the sections above are placeholders for the code provided by a specific driver. However, we still need to supply a trivial definition here to placate `CWEAVE` whenever this portion of the code is used isolated in documentation.

```
⟨ Various output modes 202 ⟩ =
```

This code is used in section 201.

203 Standard library declarations for memory management routines, some syntactic sugar, command line processing, and variadic functions are all that is needed.

⟨ Outer definitions 203 ⟩ =
```
#include <stdlib.h>
#include <stdbool.h>
#include <stdarg.h>
#include <assert.h>
#include <string.h>
```
See also section 240.

This code is used in section 201.

204 This code snippet is a payment for some poor (in my view) philosophy on the part of the `bison` and `flex` developers. There used to be an option in `bison` to output just the tables and the action code but it had never worked correctly and it was simply dropped in the latest version. Instead, one can only get access to `bison`'s goodies as part of a tangled mess of **#define**'s and error processing code. Had the tables and the parser function itself been considered separate, well isolated sections of `bison`'s output, there would simply be no reason for dirty tricks like the one below, one would be able to write custom error processing functions, unicorns would roam the Earth and pixies would hand open sourced tablets to everyone. At a minimum, it would have been a much cleaner, modular approach. There is also strange reluctance on the part of the `gcc` team to output any intermediate code other than the results of preprocessing and assembly. I have seen an argument that involves some sort of appeal to making the code difficult to close source but the logic of it escaped me completely.

Ideally, there should be no such thing as a parser generator, or a compiler, for that matter: all of these are just basic table driven rewriting routines. Tables are hard but table driven code should not be. If one had access to the tables themselves, and some canonical examples of code driven by such tables, like *yyparse*( ) and *yylex*( ), the flexibility of these tools would improve tremendously. Barring that, this is what we have to do *now*.

There are several ways to gain write access to the data declared **const** in C, like passing its address to a function with no prototype. All these methods have one drawback: loopholes that make them possible have been steadily getting on the chopping block of the C standards committee. Indeed, **const** data should be constant. Even if one succeeds in getting access, there is no reason to believe that the data is not allocated in a write-only region of the memory. The cleanest way to get write access then is to eliminate **const** altogether. The code should have the same semantics after that, and the trick is only marginally bad.

The last two definitions are less innocent (and, at least the second one, are prohibited by the ISO standard (clause 6.10.8(2), see [ISO/C11])) but `gcc` does not seem to mind, and it gets rid of warnings about dropping a **const** qualifier whenever an *assert* is encountered. Since the macro is not recursively expanded, this will only work if . . .`FUNCTION__` is treated as a pseudo-variable, as it is in `gcc`, not a macro.

```
#define const
#define __PRETTY_FUNCTION__ (char *) __PRETTY_FUNCTION__
#define __FUNCTION__ (char *) __FUNCTION__
```

205 The output file has to be known to both parts of the code, so it is declared at the very beginning of the program. We also add some syntactic sugar for loops.

```
#define forever for ( ; ; )
```
⟨ Common code for C preamble 205 ⟩ =
```
#include <stdio.h>
  FILE *tables_out;
```

206 The clean-up portion of the code can be left empty, as all it does is close the output file, which can be left to the operating system but we take care of it ourselves to keep out code 'clean' [1]).

[1]) In case the reader has not noticed yet, this is a weak attempt at humor to break the monotony of going through the lines of `CTANGLE`'d code

⟨ Clean up 206 ⟩ =
fclose(*tables_out*);
This code is used in section 201.

207 There is a descriptor controlling the output of the program as a whole. The code below is an example of a literate programming technique that will be used repeatedly to maintain large structures that can grow during the course of the program design. Note that the name of each table is only mentioned once, the rest of the code is generic.

Technically speaking, all of this can be done with C preprocessor macros of moderate complexity, taking advantage of its expansion rules but it is not nearly as transparent as the `CWEB` approach.

⟨ Local variable and type declarations 207 ⟩ =
struct output_d {
⟨ Output descriptor fields 208 ⟩
};
struct output_d *output_desc* ⇐ {⟨ Default outputs 209 ⟩};
See also sections 210, 221, 225, 237, and 241.
This code is used in section 201.

208 To declare each table field in the global output descriptor, all one has to do is to provide a general pattern.

⟨ Output descriptor fields 208 ⟩ =
#define *_register_table_d*(*name*) **bool** *output_##name*:1;
⟨ Table names 215 ⟩
#undef *_register_table_d*
See also sections 219 and 226.
This code is used in section 207.

209 Same for assigning default values to each field.

⟨ Default outputs 209 ⟩ =
#define *_register_table_d*(*name*) *.output_##name* ⇐ 0, ▷ do not output any tables by default ◁
⟨ Table names 215 ⟩
#undef *_register_table_d*
See also sections 220 and 227.
This code is used in section 207.

210 Each descriptor is populated using the same approach.

⟨ Local variable and type declarations 207 ⟩ +=
#define *_register_table_d*(*name*) **struct** *table_d name##_desc* ⇐ {0};
⟨ Table names 215 ⟩
#undef *_register_table_d*

211 The reason to implement the table output routine as a macro is to avoid writing separate functions for tables of different types of data (stings as well as integers). The output is controlled by each table's *descriptor* defined below. A more sophisticated approach is possible but this code is merely a 'patch' so we are not after full generality [1]).

```
#define output_table(table_desc, table_name, stream)
        if (output_desc.output_##table_name) {
          int i, j ⇐ 0;
          fprintf(stream, table_desc.preamble, table_desc.name);
          for (i ⇐ 0; i < sizeof (table_name)/sizeof (table_name[0]) − 1; i++) {
            if (table_desc.formatter) {
              j +⇐ table_desc.formatter(stream, i);
```

[1]) A somewhat cleaner way to achieve the same effect is to use the `_Generic` facility of C11.

```
        }
        else {
          if (table_name[i]) {
            j +⇐ fprintf(stream, table_desc.separator, table_name[i]);
          }
          else {
            j +⇐ fprintf(stream, "%s", table_desc.null);
          }
        }
        if (j > MAX_PRETTY_LINE ∧ table_desc.prettify) {
          fprintf(stream, "\n");
          j ⇐ 0;
        }
      }
      if (table_desc.formatter) {
        table_desc.formatter(stream, −i);
      }
      else {
        if (table_name[i]) {
          fprintf(stream, table_desc.postamble, table_name[i]);
        }
        else {
          fprintf(stream, "%s", table_desc.null_postamble);
        }
      }
      if (table_desc.cleanup) {
        table_desc.cleanup(&table_desc);
      }
    }
```

⟨ Global variables and types 211 ⟩ =

```
struct table_d {
  ⟨ Generic table desciptor fields 212 ⟩
};
```

See also sections 216, 218, 224, and 233.

This code is used in section 201.

212 ⟨ Generic table desciptor fields 212 ⟩ =

```
char *name;
char *preamble;
char *separator;
char *postamble;
char *null_postamble;
char *null;
bool prettify;
int (*formatter)(FILE *, int);
void (*cleanup)(struct table_d *);
```

This code is used in section 211.

213 Tables are output first. The action output code must come last since it changes the values of the tables to achieve its goals. Again, a different approach is possible, that saves the data first but simplicity was deemed more important than total generality at this point.

⟨ Perform output 213 ⟩ =

```
⟨ Output all tables 214 ⟩
```

See also section 228.

This code is used in section 201.

214 One more application of 'gather the names first then process' technique.

⟨ Output all tables 214 ⟩ =

```
#define _register_table_d(name) output_table(name##_desc, name, tables_out);
  ⟨ Table names 215 ⟩
#undef _register_table_d
```

This code is used in section 213.

215 Tables will be output by each driver. Placeholder here, for `CWEAVE`'s piece of mind.

⟨ Table names 215 ⟩ =

This code is used in sections 208, 209, 210, 214, and 277.

216 Action output invokes a totally new level of dirty code. If tables, constants, and tokens are just data structures, actions are actually code. We can only hope to cycle through all the actions which is enough to use `bison` or `flex` successfully with TEX. The **switch** statement containing the actions is embedded in the parser code so to get access to each action we have to coerce *yyparse*( ) to jump to each case. This is where we need the table manipulation. This code is highly specific to the program used (since `bison` and `flex` code have to be 'reverse engineered' to make the parser and scanner functions do what we want), here we only declare the options controlling the level of detail and the type of actions output.

⟨ Global variables and types 211 ⟩ +=

```
static int bare_actions ⇐ 0;
  ▷ (static for local variables) and int to pacify the compiler (for a constant initializer and compatible type) ◁
static int optimize_actions ⇐ 0;
```

217 The first of the following options allows one to output an action switch without the actions themselves. It is useful when one needs to output a TEX parser for a grammar file that is written in C. In this case it will be impossible to cycle through actions (as no setup code has been executed), so the parser invocation is omitted.

The second option splits the action switch into several macros to speed up the processing of the action code.

The last argument of the 'flexible' macro below is supposed to be an extended description of each option which can be later utilized by a *usage*( ) function.

⟨ Raw option list 217 ⟩ =

```
_register_option("bare-actions", no_argument, &bare_actions, 1, "")
_register_option("optimize-actions", no_argument, &optimize_actions, 1, "")
```

This code is used in section 244.

218 The rest of the action output code mimics that for table output, starting with the descriptor. To make the output format more flexible, this descriptor should probably be turned into a specialized routine.

⟨ Global variables and types 211 ⟩ +=

```
struct action_d {
  char *preamble;
  char *act_setup;
  char *act_suffix;
  char *action1;
  char *actionn;
  char *postamble;
  void (*print_rule)(int);
  void (*cleanup)(struct action_d *);
};
```

219 ⟨ Output descriptor fields 208 ⟩ +=
bool *output_actions*:1;

220 Nothing is output by default, including actions.
⟨ Default outputs 209 ⟩ +=
.*output_actions* ⇐ 0,

221 ⟨ Local variable and type declarations 207 ⟩ +=
struct action_d *action_desc* ⇐ {0};

222 The function below outputs the TEX code of each action when the appropriate action is 'run' by the action output switch. The main concern in designing these functions is to make the code easier to look at. Further explanation is given in the grammar file. If the parser is doing its job, this is the only place where one would actually see these as functions (or, rather, macros).

In compliance with paragraph 6.10.8(2) [1]) of the ISO C11 standard the names of these macros do not start with an underscore, since the first letter of `TeX` is uppercase [2]).

#define `TeX_`(*string*) *fprintf*(*tables_out*, "␣␣␣␣␣␣␣␣␣␣%s%%\n", *string*)
⟨ C preamble 222 ⟩ =
#define `TeX__`(*string*, ...) *fprintf*(*tables_out*, "␣␣␣␣␣␣"*string*"%s\n", __VA_ARGS__, "%")

223 TEX tables

We begin with a few macros to facilitate the output of tables in the format that TEX can understand. There is really no good way to represent an array in TEX so a rather weak compromise was chosen. Further explanation of this choice is given in the TEX file that implements the TEX parser for the `bison` input grammar. Some tables require name adjustments due to TEX's reluctance to treat digits as part of a name.

#define *tex_table_generic*(*table_name*) *table_name*##*_desc.preamble* ⇐ "\\newtable{%s}{%%\n";
table_name##*_desc.separator* ⇐ "%d\\or␣";
table_name##*_desc.postamble* ⇐ "%d}%%\n";
table_name##*_desc.null_postamble* ⇐ "0}%%\n";
table_name##*_desc.null* ⇐ "0\\or␣";
table_name##*_desc.prettify* ⇐ *true*;
table_name##*_desc.formatter* ⇐ Λ;
table_name##*_desc.cleanup* ⇐ Λ;
*output_desc.output_*##*table_name* ⇐ 1;
#define *tex_table*(*table_name*) *tex_table_generic*(*table_name*);
table_name##*_desc.name* ⇐ #*table_name*;

224 Outputting constants. An approach paralleling the table output scheme is taken with constants. Since constants are C *macros* one has to be careful to avoid the temptation of using constant *names* directly as names for fields in structures. They will simply be replaced by the constants' values. When the names are concatenated with other tokens, however, the C preprocessor postpones the macro expansion until the concatenation is complete (see clauses 6.10.3.1, 6.10.3.2, and 6.10.3.3 of the ISO C Standard, [ISO/C11]). Unless the result of the concatenation is still expandable, the expansion will halt.

⟨ Global variables and types 211 ⟩ +=
struct const_d {
char ∗*format*;
char ∗*name*;
};

[1]) [...] *Any other predefined macro names shall begin with a leading underscore followed by an uppercase letter or a second underscore.* [2]) One might wonder why one of these functions is defined as a `CWEB` macro while the other is put into the preamble 'by hand'. It really makes no difference, however, the reason the second macro is defined explicitly is `CWEB`'s lack of awareness of 'variadic' macros which produces undesirable typesetting artefacts.

225 ⟨ Local variable and type declarations 207 ⟩ +=
```
#define _register_const_d(c_name) struct const_d c_name##_desc;
  ⟨ Constant names 230 ⟩
#undef _register_const_d
```

226 ⟨ Output descriptor fields 208 ⟩ +=
```
#define _register_const_d(c_name) bool output_##c_name:1;
  ⟨ Constant names 230 ⟩
#undef _register_const_d
```

227 ⟨ Default outputs 209 ⟩ +=
```
#define _register_const_d(c_name) .output_##c_name ⇐ 0,
  ⟨ Constant names 230 ⟩
#undef _register_const_d
```

228 ⟨ Perform output 213 ⟩ +=
```
  fprintf(tables_out, "%%\n%% constant definitions\n%%\n");
  ⟨ Output constants 229 ⟩
```

229 ⟨ Output constants 229 ⟩ =
```
  { int any_constants ⇐ 0;
#define _register_const_d(c_name)
  if (output_desc.output_##c_name) {
    const_out(tables_out, c_name##_desc, c_name)
    any_constants ⇐ 1;
  }
  ⟨ Constant names 230 ⟩
#undef _register_const_d
  if (any_constants) ;    ▷ this is merely a placeholder statement ◁
  }
```
This code is used in section 228.

230 Constants are very driver specific, so to make CWEAVE happy ...

⟨ Constant names 230 ⟩ =

This code is used in sections 225, 226, 227, and 229.

231 A macro to help with constant output.
```
#define const_out(stream, c_desc, c_name) fprintf(stream, c_desc.format, c_desc.name, c_name);
```

232 Action switch output routines modify the automata tables and therefore have to be output last. Since action output is highly automaton specific, we leave this section blank here, to pacify CWEAVE in case this file is typeset by itself.

⟨ Output action switch, if any 232 ⟩ =

This code is used in section 201.

233 Error codes

⟨ Global variables and types 211 ⟩ +=
```
  enum err_codes {
    ⟨ Error codes 234 ⟩ LAST_ERROR
  };
```

234 ⟨ Error codes 234 ⟩ =
```
  NO_MEMORY, BAD_STRING, BAD_MIX_FORMAT,
```
See also section 298.

This code is used in section 233.

235 A lot more care is necessary to output the token table. A number of precautions are taken to ensure that a maximum possible range of names can be passed safely to TEX. This involves some manipulation of `\catcode`'s and control characters. The complicated part is left to TEX so the output code can be kept simple. The helper function below is used to 'combine' two strings.

```
#define MAX_PRETTY_LINE 100
```

⟨ Auxiliary function declarations 235 ⟩ =

```
char *mix_string(char *format, ...);
```

This code is used in section 201.

236 ⟨ Auxiliary function definitions 236 ⟩ =

```
char *mix_string(char *format, ...)
{
  char *buffer;
  size_t size ⇐ 0;
  int length ⇐ 0;
  int written ⇐ 0;
  char *formatp ⇐ format;
  va_list ap, ap_save;
  va_start(ap, format);
  va_copy(ap_save, ap);
  size ⇐ strnlen(format, MAX_PRETTY_LINE * 5);
  if (size ⩾ MAX_PRETTY_LINE * 5) {
    fprintf(stderr, "%s: runaway string?\n", __func__);
    exit(BAD_STRING);
  }
  while ((formatp ⇐ strstr(formatp, "%"))) {
    switch (formatp[1]) {
    case 's':
      length ⇐ strnlen(va_arg(ap, char *), MAX_PRETTY_LINE * 5);
      if (length ⩾ MAX_PRETTY_LINE * 5) {
        fprintf(stderr, "%s: runaway string?\n", __func__);
        exit(BAD_STRING);
      }
      size +⇐ length;
      size −⇐ 2;
      formatp++;
      break;
    case '%':
      size−−;
      formatp +⇐ 2;
    default: printf("%s: cannot handle %%%c in mix string format\n", __func__, formatp[1]);
      exit(BAD_MIX_FORMAT);
    }
  }
  buffer ⇐ (char *) malloc(sizeof(char) * size + 1);
  if (buffer) {
    written ⇐ vsnprintf(buffer, size + 1, format, ap_save);
    if (written < 0 ∨ written > size) {
      fprintf(stderr, "%s: runaway string?\n", __func__);
      exit(BAD_STRING);
    }
  }
  else {
    fprintf(stderr, "%s: failed to allocate memory for the output string\n", __func__);
    exit(NO_MEMORY);
  }
```

```
    va_end(ap);
    va_end(ap_save);
    return buffer;
  }
```

This code is used in section 201.

237 Initial setup

Depending on the output mode (right now only TEX and 'tokens only' (in the `bison` 'driver') are supported) the format of each table, action field and token has to be set up.

```
⟨ Local variable and type declarations 207 ⟩ +=
  enum output_mode {
    ⟨ Output modes 238 ⟩ LAST_OUT
  };
```

238 And to calm down CWEAVE ...

```
⟨ Output modes 238 ⟩ =
```

This code is used in section 237.

239 TEX is the main output mode.

```
⟨ Establish defaults 239 ⟩ =
  enum output_mode mode ⇐ TEX_OUT;
```

This code is used in section 201.

240 Command line processing

This program uses a standard way of parsing the command line, based on *getopt_long*. At the heart of the setup are the array below with a couple of supporting variables.

```
⟨ Outer definitions 203 ⟩ +=
#include <unistd.h>
#include <getopt.h>
#include <string.h>
```

241

```
⟨ Local variable and type declarations 207 ⟩ +=
  const char *usage ⇐ "%s␣[options]␣output_file\n";
```

242

```
⟨ Command line processing variables 242 ⟩ =
  int c, option_index ⇐ 0;
  enum higher_options {
    NON_OPTION ⇐ FF16, ⟨ Higher index options 246 ⟩ LAST_HIGHER_OPTION
  };
  static struct option long_options[] ⇐ {
  ⟨ Long options array 244 ⟩
  {0, 0, 0, 0}};
```

This code is used in section 201.

243 The main loop of the command line option processing follows. This can be used as a template for setting up the option processing. The specific cases are added to in the course of adding new features.

```
⟨ Process command line options 243 ⟩ =
  opterr ⇐ 0;    ▷ we do our own error reporting ◁
  forever
  {
    c ⇐ getopt_long(argc, argv, ":"⟨ Short option list 245 ⟩, long_options, &option_index);
    if (c = −1) break;
    switch (c) {
```

```
    case 0:    ▷ it is a flag, the name is kept in long_options[option_index].name, and the value can be found in
          long_options[option_index].val ◁
      break;
    ⟨ Cases affecting the whole program 247 ⟩;
    ⟨ Cases involving specific modes 248 ⟩;
    case '?':
      fprintf(stderr, "Unknown option: ‘%s’, see ‘Usage’ below\n\n", argv[optind − 1]);
      fprintf(stderr, usage, argv[0]);
      exit(1);
      break;
    case ':':
      fprintf(stderr, "Missing argument for ‘%s’\n\n", argv[optind − 1]);
      fprintf(stderr, usage, argv[0]);
      exit(1);
      break;
    default:
      printf("warning: feature ‘%c’ is not yet implemented\n", c);
    }
  }
  if (optind ⩾ argc) {
    fprintf(stderr, "No output file specified!\n");
  }
  else {
    tables_out ⇐ fopen(argv[optind++], "w");
  }
  if (optind < argc) {
    printf("script files to be loaded: ");
    while (optind < argc) printf("%s ", argv[optind++]);
    putchar('\n');
  }
```

This code is used in section 201.

244 ⟨ Long options array 244 ⟩ =

```
#define _register_option(name, arg_flag, loc, val, exp) {name, arg_flag, loc, val},
  ⟨ Raw option list 217 ⟩
#undef _register_option
```

This code is used in section 242.

245 In addition to spelling out the full command line option name (such as `--help`) *getopt_long* gives the user a choice of using a shortcut (say, `-h`). As individual options are treated in drivers themselves, there are no shortcuts to supply at this point. We leave this section (and a number of others) empty to be filled in with the driver specific code to pacify `CWEAVE`.

⟨ Short option list 245 ⟩ =

This code is used in section 243.

246 Some options have one-letter ‘shortcuts’, whereas others only exist in ‘fully spelled-out’ form. To easily keep track of the latter, a special enumerated list is declared. To add to this list, simply add to the `CWEB` section below.

⟨ Higher index options 246 ⟩ =

This code is used in section 242.

247 ⟨ Cases affecting the whole program 247 ⟩ =

This code is used in section 243.

248 ⟨ Cases involving specific modes 248 ⟩ =

This code is used in section 243.

249 `bison` **specific routines**

The placeholder code left blank in the common routines is filed in with the code relevant to the output of parser tables in the following sections.

250 **Tables**

Here are all the parser table names. Some tables are not output but adding one to the list in the future will be easy: it does not even have to be done here.

⟨ Parser table names 250 ⟩ =

```
_register_table_d(yytranslate)
_register_table_d(yyr1)
_register_table_d(yyr2)
_register_table_d(yydefact)
_register_table_d(yydefgoto)
_register_table_d(yypact)
_register_table_d(yypgoto)
_register_table_d(yytable)
_register_table_d(yycheck)
_register_table_d(yyprhs)
_register_table_d(yyrhs)
_register_table_d(yytoknum)
_register_table_d(yystos)
_register_table_d(yytname)
```

See also section 255.

251 One special table requires a little bit more preparation. This is a table that lists the depth of the stack before an implicit terminal. It is not one of the tables that is used by `bison` itself but is needed if the symbolic name processing is to be implemented (`bison` has access to this information 'on the fly').

⟨ Variables and types local to the parser 251 ⟩ =

```
unsigned int yyrthree[YYNRULES + 1] ⇐ {0};
```

See also sections 258 and 291.

252 We populate this table below ...

⟨ Parser defaults 252 ⟩ =

```
assert(YYNRULES + 1 = sizeof (yyprhs)/sizeof (yyprhs[0]));
{
  int i, j;
  for (i ⇐ 1; i ⩽ YYNRULES; i++) {
    for (j ⇐ 0; yyrhs[yyprhs[i] + j] ≠ −1; j++) {
      assert(yyprhs[i] + j < sizeof (yyrhs));
      assert(j < yyr1[i]);
      if (⟨ This is an implicit term 253 ⟩) {
        ⟨ Find the rule that defines it and set yyrthree 254 ⟩
      }
    }
  }
}
```

253 ⟨ This is an implicit term 253 ⟩ =

```
(strlen(yytname[yyrhs[yyprhs[i] + j]]) > 1) ∧ (yytname[yyrhs[yyprhs[i] + j]][0] =
    'Υ') ∧ (yytname[yyrhs[yyprhs[i] + j]][1] = '©')
```

This code is used in section 252.

254 ⟨ Find the rule that defines it and set *yyrthree* 254 ⟩ =

```
int rule_number;
for (rule_number ⇐ 1; rule_number < YYNRULES; rule_number++) {
  if (yyr1[rule_number] = yyrhs[yyprhs[i] + j]) {
    yyrthree[rule_number] ⇐ j;
    break;
  }
}
assert(rule_number < YYNRULES);
```

This code is used in section 252.

255 . . . and add its name to the list.

⟨ Parser table names 250 ⟩ +=

```
_register_table_d(yyrthree)
```

256 Actions

There are several ways of making *yyparse*( ) execute all portions of the action code. The one chosen here makes sure that none of the tables gets written past its last element. To see how it works, it might be helpful to 'walk through' `bison`'s output to see how each change affects the generated parser.

⟨ Output parser semantic actions 256 ⟩ =

```
if (output_desc.output_actions) {
  int i, j;
  fprintf(tables_out, "%s", action_desc.preamble);
  if (not bare_actions) {
    yypact[0] ⇐ YYPACT_NINF;
    yypgoto[0] ⇐ −1;
    yydefgoto[0] ⇐ YYFINAL;
  }
  for (i ⇐ 1; i < sizeof(yyr1)/sizeof(yyr1[0]); i++) {
    fprintf(tables_out, action_desc.act_setup, i, yyr2[i] − 1);
    if (action_desc.print_rule) {
      action_desc.print_rule(i);
    }
    if (yyr2[i] > 0) {
      if (action_desc.action1) {
        fprintf(tables_out, "%s", action_desc.action1);
      }
    }
    for (j ⇐ 2; j ⩽ yyr2[i]; j++) {
      if (action_desc.actionn) {
        fprintf(tables_out, action_desc.actionn, j);
      }
    }
    if (not bare_actions) {
      yyr1[i] ⇐ YYNTOKENS;
      yydefact[0] ⇐ i;
      yyr2[i] ⇐ 0;
      yyparse(YYPARSE_PARAMETERS);
    }
    fprintf(tables_out, action_desc.act_suffix, i, yyr2[i] − 1);
  }
  fprintf(tables_out, "%s", action_desc.postamble);
  if (action_desc.cleanup) {
    action_desc.cleanup(&action_desc);
  }
}
```

257 Constants

⟨ Parser constants 257 ⟩ =
_register_const_d(YYEMPTY)
_register_const_d(YYPACT_NINF)
_register_const_d(YYEOF)
_register_const_d(YYLAST)
_register_const_d(YYNTOKENS)
_register_const_d(YYNRULES)
_register_const_d(YYNSTATES)
_register_const_d(YYFINAL)

This code is used in section 283.

258 Tokens

Similar techniques are employed in token output. Tokens are parser specific (the scanner only needs their numeric values) so we need *some* flexibility to output them in a desired format. For special purposes (say changing the way tokens are typeset) we can control the format tokens are output in.

⟨ Variables and types local to the parser 251 ⟩ +=
char ∗*token_format_char* ⇐ Λ;
char ∗*token_format_affix* ⇐ Λ;
char ∗*token_format_suffix* ⇐ Λ;
char ∗*bootstrap_token_format* ⇐ Λ;

259

⟨ Parser specific option list 259 ⟩ =
_register_option("token-format-char", *required_argument*, 0, TOKEN_FORMAT_CHAR, "")
_register_option("token-format-affix", *required_argument*, 0, TOKEN_FORMAT_AFFIX, "")
_register_option("token-format-suffix", *required_argument*, 0, TOKEN_FORMAT_SUFFIX, "")
_register_option("bootstrap-token-format", *required_argument*, 0, BOOTSTRAP_TOKEN_FORMAT, "")

See also sections 269, 286, and 289.

260

⟨ Higher index parser specific options 260 ⟩ =
TOKEN_FORMAT_CHAR, TOKEN_FORMAT_AFFIX, TOKEN_FORMAT_SUFFIX, BOOTSTRAP_TOKEN_FORMAT,

See also sections 270 and 285.

261

⟨ Handle parser output options 261 ⟩ =
case TOKEN_FORMAT_CHAR:
token_format_char ⇐ (**char** ∗) *malloc*((*strlen*(*optarg*) + 1) ∗ **sizeof**(**char**));
strcpy(*token_format_char*, *optarg*);
break;
case TOKEN_FORMAT_AFFIX:
token_format_affix ⇐ (**char** ∗) *malloc*((*strlen*(*optarg*) + 1) ∗ **sizeof**(**char**));
strcpy(*token_format_affix*, *optarg*);
break;
case TOKEN_FORMAT_SUFFIX:
token_format_suffix ⇐ (**char** ∗) *malloc*((*strlen*(*optarg*) + 1) ∗ **sizeof**(**char**));
strcpy(*token_format_suffix*, *optarg*);
break;
case BOOTSTRAP_TOKEN_FORMAT:
bootstrap_token_format ⇐ (**char** ∗) *malloc*((*strlen*(*optarg*) + 1) ∗ **sizeof**(**char**));
strcpy(*bootstrap_token_format*, *optarg*);
break;

See also sections 288 and 292.

262

⟨ Parser specific output descriptor fields 262 ⟩ =
bool *output_tokens*:1;

263 No tokens are output by default.

⟨ Parser specific default outputs 263 ⟩ =
```
.output_tokens ⇐ 0,
```

264 The only part of the code below that needs any explanation is the 'bootstrap' token output. In `bison` every token has three attributes: its 'macro name' (say, `STRING`) that is used by the parse code internally, its 'print name' (`"string"` to continue the example) that `bison` uses to print the token names in its diagnostic messages, and its numeric value (that can be assigned implicitly by `bison` itself or explicitly by the user). Only the 'print names' are kept in the *yytname* array so to reuse the scanner used by `bison` we either have to extract the token 'macro names' from the C code ourselves to pass them on to the lexer, or use a special 'stripped down' version of a `bison` grammar parser to extract the names from the parser's `bison` grammar. To do this, some token names would still need to be known to the scanner. These tokens are selected by hand to make the 'bootstrapping' parser operational. The token list for the `bison` grammar parser can be examined as part of the appropriate driver file.

⟨ Output parser tokens 264 ⟩ =
```
if (output_desc.output_tokens) {
  int i;
  int length;
  char token;
  char *token_name;
  bool too_creative ⇐ false;
  for (i ⇐ 258; i < sizeof (yytranslate)/sizeof (yytranslate[0]); i++) {
    token_name ⇐ yytname[yytranslate[i]];
    if (token_name) {
      fprintf(tables_out, token_format_affix, yytranslate[i], i);
      length ⇐ 0;
      while ((token ⇐ *token_name)) {
        if (token_format_char) {
          length +⇐ fprintf(tables_out, token_format_char, (unsigned int) token);
        }
        if (token < °40 ∨ token = °177) {
          too_creative ⇐ true;
        }
        token_name++;
      }
      fprintf(tables_out, token_format_suffix, too_creative ? ".unprintable." : yytname[yytranslate[i]]);
    }
  }
}
#ifdef BISON_BOOTSTRAP_MODE
  fprintf(tables_out, "\\bootstrapmodetrue\n");
  fprintf(tables_out, "%% token values needed to bootstrap the parser\n");
  bootstrap_tokens(bootstrap_token_format);
#endif
```

265 The size of the token name table is useful to determine, say, how many 'named' tokens the parser uses.

⟨ Output parser constants 265 ⟩ =
```
fprintf(tables_out, "\\constset{YYTRANSLATESIZE}{%d}%%\n", (int)(sizeof (yytranslate)/sizeof (yytranslate[0])));
```

266 Output modes

The code below can be easily extended and modified to output parser tables, actions, and constants in a language of one's choice. We are only interested in TEX, however, thus other modes are very rudimentary or non-existent at this point.

267 Token only mode

Token only output mode does exactly what is expected: outputs token names and values in the format of your choosing.

```
⟨ Parser specific output modes 267 ⟩ =
  TOKEN_ONLY_OUT,
```

See also sections 273 and 275.

268
```
⟨ Handle parser related output modes 268 ⟩ =
case TOKEN_ONLY_OUT:
  ⟨ Prepare token only output environment 272 ⟩
  break;
```

See also sections 274 and 276.

269
```
⟨ Parser specific option list 259 ⟩ +=
  _register_option("token-only-mode", no_argument, 0, TOKEN_ONLY_MODE, "")
```

270
```
⟨ Higher index parser specific options 260 ⟩ +=
  TOKEN_ONLY_MODE,
```

271
```
⟨ Configure parser output modes 271 ⟩ =
case TOKEN_ONLY_MODE:
  mode ⇐ TOKEN_ONLY_OUT;
  break;
```

272
```
⟨ Prepare token only output environment 272 ⟩ =
  if (not token_format_char) {
    token_format_char ⇐ "{%u}";
  }
  if (not token_format_affix) {
    token_format_affix ⇐ "%% token: %d, token value: %d\n\\prettytoken@{";
  }
  if (not token_format_suffix) {
    token_format_suffix ⇐ "}%% %s\n";
  }
  output_desc.output_tokens ⇐ 1;
```

This code is used in section 268.

273 Generic output

Generic output is not programmed yet.

```
⟨ Parser specific output modes 267 ⟩ +=
  GENERIC_OUT,
```

274
```
⟨ Handle parser related output modes 268 ⟩ +=
case GENERIC_OUT:
  printf("This mode is not supported yet\n");
  exit(0);
  break;
```

275 TEX output

The TEX mode is the main reason for this software.

```
⟨ Parser specific output modes 267 ⟩ +=
  TEX_OUT,
```

276 ⟨ Handle parser related output modes 268 ⟩ +=
```
case TEX_OUT:
  ⟨ Set up TEX table output for parser tables 277 ⟩
  ⟨ Prepare TEX format for semantic action output 281 ⟩
  ⟨ Prepare TEX format for parser constants 283 ⟩
  ⟨ Prepare TEX format for parser tokens 284 ⟩
  break;
```

277 TEX tables. We begin with a few macros to facilitate the output of tables in the format that TEX can understand. There is really no good way to represent an array in TEX so a rather weak compromise was chosen. Further explanation of this choice is given in the TEX file that implements the TEX parser for the `bison` input grammar. Some tables require name adjustments due to TEX's reluctance to treat digits as part of a name.

⟨ Set up TEX table output for parser tables 277 ⟩ =
```
#define _register_table_d(name)tex_table(name);
  ⟨ Table names 215 ⟩
#undef _register_table_d
  yyr1_desc.name ⇐ "yyrone";
  yyr2_desc.name ⇐ "yyrtwo";
```
See also section 280.

This code is used in section 276.

278 The memory allocated for the *yytname* table is released at the end.

⟨ Helper functions declarations for for parser output 278 ⟩ =
```
void yytname_cleanup(struct table_d *table);
int yytname_formatter_tex(FILE *stream, int index);
int yytname_formatter(FILE *stream, int index);
```

279 There are a number of helper functions to output complicated names in TEX. The safest way seems to be to output a name as a sequence of its ASCII codes to accommodate names like `$end` safely. TEX's `^^...` convention is supported as well.

⟨ Helper functions for parser output 279 ⟩ =
```
void yytname_cleanup(struct table_d *table)
{
  free(table→separator);
  free(table→null);
}
int yytname_formatter_tex(FILE *stream, int index)
{
  char *token_name ⇐ yytname[index];
  unsigned char token;
  int length ⇐ 0;
  fprintf(stream, "\\addname␣");
  while ((token ⇐ *token_name)) {
    if (token < °40 ∨ token = °177) {     ▷ unprintable characters ◁
      fprintf(stream, "^^%c", token < °100 ? (unsigned char)(token + °100) : (unsigned char)(token − 100));
      length +⇐ 3;
    }
    else {
      fprintf(stream, "%c", token);
      length++;
    }
    token_name++;
  }
  fprintf(stream, "\n");
```

```
  return length;
}
int yytname_formatter(FILE *stream, int index)
{
  char *token_name;
  unsigned char token;
  int length ⇐ 0;
  bool too_creative ⇐ false;    ▷ to indicate if the name is too dangerous to print ◁
  fprintf(stream, "\\addname");
  if (index ⩾ 0) {    ▷ this is not the last name ◁
    token_name ⇐ yytname[index];
    if (token_name = Λ) {
      token_name ⇐ "$impossible";
    }
    while ((token ⇐ *token_name)) {
      length +⇐ fprintf(stream, "{%u}", (unsigned int) token);
      if (token < °40 ∨ token = °177) {
        too_creative ⇐ true;
      }
      token_name++;
    }
    fprintf(stream, "%% %s\n", too_creative ? ".unprintable." : yytname[index]);
  }
  else {    ▷ this is the last name ◁
    token_name ⇐ yytname[−index];
    if (token_name = Λ) {
      token_name ⇐ "$impossible";
    }
    while ((token ⇐ *token_name)) {
      length +⇐ fprintf(stream, "{%u}", (unsigned int) token);
      token_name++;
      if (token < °40 ∨ token = °177) {
        too_creative ⇐ true;
      }
    }
    fprintf(stream, "%% %s\n\\end\n%%\n",
        too_creative ? ".unprintable." : (yytname[−index] ? yytname[−index] : "end of array"));
  }
  return length;
}
```

See also section 282.

280 ⟨ Set up TEX table output for parser tables 277 ⟩ +=

```
yytname_desc.preamble ⇐ "%%\n\\newtable{yytname}{}\\tempca0\\relax%% a robust way to\
    \ add the yytname array\n";
yytname_desc.separator ⇐ Λ;
yytname_desc.postamble ⇐ Λ;
yytname_desc.null ⇐ Λ;
yytname_desc.null_postamble ⇐ Λ;
yytname_desc.prettify ⇐ false;
yytname_desc.formatter ⇐ yytname_formatter;
yytname_desc.cleanup ⇐ Λ;
output_desc.output_yytname ⇐ 1;
```

281 ⟨ Prepare TEX format for semantic action output 281 ⟩ =

```
if (optimize_actions) {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax␣%␣see␣the␣documentation␣for␣an␣explanation␣of␣this␣trick\n"
  "\\def\\yybigswitch#1{%%\n"
  "␣␣␣␣\\csname␣dobisonaction\\number␣#1\\parsernamespace\\endcsname\n"
  "}\\stashswitch{yybigswitch}%%\n";
  action_desc.act_setup ⇐ "\n\\expandafter\\def\\csname␣dobisonaction%d\\parsernamespa\
      ce\\endcsname{%%\n%%";
  action_desc.act_suffix ⇐ "}%%␣end␣of␣rule␣%d\n";
  action_desc.action1 ⇐ Λ;
  action_desc.actionn ⇐ Λ;
  action_desc.postamble ⇐ "\n\\catcode`\\/=12\\relax\n\n";
  action_desc.print_rule ⇐ print_rule;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
else {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax␣%␣see␣the␣documentation␣for␣an␣explanation␣of␣this␣trick\n"
  "\\def\\yybigswitch#1{%%\n"
  "␣␣\\ifcase#1\\relax\n";
  action_desc.act_setup ⇐ "␣␣␣␣␣␣␣\\or␣%%␣(rule␣%d)␣";
  action_desc.act_suffix ⇐ "";
  action_desc.action1 ⇐ Λ;
  action_desc.actionn ⇐ Λ;
  action_desc.postamble ⇐ "␣␣\\else\n␣␣\\fi\n}\\stashswitch{yybigswitch}%%\n\\catcode`\
      \\/=12\\relax\n\n";
  action_desc.print_rule ⇐ print_rule;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
```

This code is used in section 276.

282 Grammar rules are listed in a readable form alongside the action code to make it possible to quickly find an appropriate action.

⟨ Helper functions for parser output 279 ⟩ +=

```
void print_rule(int n)
{
  int i;
  fprintf(tables_out, "%s%s:␣␣", (n < 10 ∧ not optimize_actions ? "␣" : ""), yytname[yyr1[n]]);
  i ⇐ yyprhs[n];
  if (yyrhs[i] < 0) {
    fprintf(tables_out, "<empty>");
  }
  else {
    while (yyrhs[i] > 0) {
      fprintf(tables_out, "%s␣", yytname[yyrhs[i]]);
      i++;
    }
  }
  fprintf(tables_out, "\n");
}
```

283 TEX constant output is another place where the techniques described above are applied. As before, the macro handles the repetitive work of initialization, declaration, etc in each place where the corresponding constant is mentioned. The one exception is `YYPACT_NINF`, which has to be handled separately because the underscore in its name makes it difficult to use it as a command sequence name.

```
⟨ Prepare TEX format for parser constants 283 ⟩ =
#define _register_const_d(c_name) c_name##_desc.format ⇐ "\\constset{%s}{%d}%%\n";
  c_name##_desc.name ⇐ #c_name;
  output_desc.output_##c_name ⇐ 1;
  ⟨ Parser constants 257 ⟩
#undef _register_const_d
  YYPACT_NINF_desc.name ⇐ "YYPACTNINF";
```

This code is used in section 276.

284 Token definitions round off the TEX output mode.

```
⟨ Prepare TEX format for parser tokens 284 ⟩ =
  token_format_char ⇐ Λ;    ▷ do not output individual characters ◁
  if (not token_format_affix) {
    token_format_affix ⇐ "\\tokenset{%d}{%d}";
  }
  if (not token_format_suffix) {
    token_format_suffix ⇐ "%%␣%s\n";
  }
  if (not bootstrap_token_format) {
    bootstrap_token_format ⇐ "\\expandafter\\def\\csname␣token\\parsernamespace␣%s\\endcs\
        name{%d}%%␣%s\n";
  }    ▷ output_desc.output_tokens ⇐ 1; is no longer necessary as it is done entirely in TEX ◁
```

This code is used in section 276.

285 Command line options

We start with the most obvious option, the one begging for help.

```
⟨ Higher index parser specific options 260 ⟩ +=
  LONG_HELP,
```

286

```
⟨ Parser specific option list 259 ⟩ +=
  _register_option("help", no_argument, 0, LONG_HELP, "")
```

287

```
⟨ Shortcuts for command line options affecting parser output 287 ⟩ =
  "h"
```

See also section 290.

288

```
⟨ Handle parser output options 261 ⟩ +=
case 'h':    ▷ short help ◁
  fprintf(stderr, "Usage:␣%s␣[options]␣output_file\n", argv[0]);
  exit(0);
  break;    ▷ should not be needed ◁
case LONG_HELP:
  fprintf(stderr,
      "%s␣[--mode=TeX:options]␣output_file␣outputs␣tables\n""␣␣␣␣and␣constants␣for␣a␣TeX␣parser\n",
      argv[0]);
  exit(0);
  break;    ▷ should not be needed ◁
```

289 ⟨ Parser specific option list 259 ⟩ +=

```
_register_option("debug", optional_argument, 0, 'b', "")
_register_option("mode", required_argument, 0, 'm', "")
_register_option("table-separator", required_argument, 0, 'z', "")
_register_option("format", required_argument, 0, 'f', "")    ▷ name? ◁
_register_option("table", required_argument, 0, 't', "")    ▷ specific table ◁
_register_option("constant", required_argument, 0, 'c', "")    ▷ specific constant ◁
_register_option("name-length", required_argument, 0, 'l', "")    ▷ change MAX_NAME_LENGTH ◁
_register_option("token", required_argument, 0, 'n', "")    ▷ specific token ◁
_register_option("run-parse", required_argument, 0, 'p', "")    ▷ run the parser ◁
_register_option("parse-file", required_argument, 0, 'i', "")    ▷ input for the parser ◁
```

290 The string below is a list of short options.

⟨ Shortcuts for command line options affecting parser output 287 ⟩ +=

```
"z:m:f:t:"
```

291 A few options can be immediately discussed.

⟨ Variables and types local to the parser 251 ⟩ +=

```
char *table_separator ⇐ "%s␣";
```

292 ⟨ Handle parser output options 261 ⟩ +=

```
case 'm':    ▷ output mode ◁
  switch (optarg[0]) {
  case 'T': case 't':
    mode ⇐ TEX_OUT;
    break;
  case 'b': case 'B': case 'g': case 'G':
    mode ⇐ GENERIC_OUT;
    break;
  default:
    break;
  }
  break;
case 'z': table_separator ⇐ (char *) malloc((strlen(optarg) + 1) * sizeof(char));
  strcpy(table_separator, optarg);
  break;
```

293 `flex` **specific routines**

The output of the scanner automaton consists of similar steps to the parser output. The major difference is actions and constants.

294 **Tables**

As in the case of a parser we start with all the table names.

⟨ Scanner table names 294 ⟩ =

```
_register_table_d(yy_accept)
_register_table_d(yy_ec)
_register_table_d(yy_meta)
_register_table_d(yy_base)
_register_table_d(yy_def)
_register_table_d(yy_nxt)
_register_table_d(yy_chk)
```

295 Actions

The scanner function, *yylex*( ), has been reverse engineered to execute all portions of the action code. The method chosen here makes sure that none of the tables gets written past its last element.

⟨ Variables and types local to the scanner driver 295 ⟩ =

```
int max_yybase_entry ⇐ 0;
int max_yyaccept_entry ⇐ 0;
int max_yynxt_entry ⇐ 0;
int max_yy_ec_entry ⇐ 0;
```

See also sections 299 and 319.

296 The 'exotic' scanner constants treated below are the constants used to control the scanner code itself. Unfortunately they are not given any names which can be used by the 'driver' to output them in a simple way.

⟨ Compute exotic scanner constants 296 ⟩ =

```
{
  int i;
  for (i ⇐ 0; i < sizeof (yy_base)/sizeof (yy_base[0]); i++) {
    if (yy_base[i] > max_yybase_entry) {
      max_yybase_entry ⇐ yy_base[i];
    }
  }
  for (i ⇐ 0; i < sizeof (yy_nxt)/sizeof (yy_nxt[0]); i++) {
    if (yy_nxt[i] > max_yynxt_entry) {
      max_yynxt_entry ⇐ yy_nxt[i];
    }
  }
  for (i ⇐ 0; i < sizeof (yy_accept)/sizeof (yy_accept[0]); i++) {
    if (yy_accept[i] > max_yyaccept_entry) {
      max_yyaccept_entry ⇐ yy_accept[i];
    }
  }
  for (i ⇐ 0; i < sizeof (yy_ec)/sizeof (yy_ec[0]); i++) {
    if (yy_ec[i] > max_yy_ec_entry) {
      max_yy_ec_entry ⇐ yy_ec[i];
    }
  }
}
```

297 ⟨ Output scanner actions 297 ⟩ =

```
if (output_desc.output_actions) {
  int i, j;
  yyscan_t fake_scanner;
  fprintf (tables_out, "%s", action_desc.preamble);
  if (not bare_actions) {
    if (yylex_init (&fake_scanner)) {
      printf ("Cannot␣initialize␣the␣scanner\n");
    }
    yy_ec[0] ⇐ 0;
    yy_base[1] ⇐ max_yybase_entry;
    yy_chk[max_yybase_entry] ⇐ 1;
    yy_nxt[max_yybase_entry] ⇐ 1;
  }
  for (i ⇐ 1; i ⩽ max_yyaccept_entry; i++) {
    fprintf (tables_out, action_desc.act_setup, i);
    if (i = YY_END_OF_BUFFER) {
```

```
        fprintf(tables_out, "␣%%␣YY_END_OF_BUFFER\n%s\n", "␣␣␣␣␣␣␣␣␣␣\\yylexeofaction");
      }
      else {
        fprintf(tables_out, "\n");
        if (^not bare_actions) {
          ((struct yyguts_t *) fake_scanner)→yy_hold_char ⇐ 0;
          yy_accept[1] ⇐ i;
          yylex(Λ, fake_scanner);
        }
      }
      fprintf(tables_out, action_desc.act_suffix, i);
    }
    fprintf(tables_out, "␣␣␣␣␣␣%%␣end␣of␣file␣states:\n%s\n",
        "␣␣␣␣␣␣%#define␣YY_STATE_EOF(state)␣(YY_END_OF_BUFFER␣+␣state␣+␣1)");
    if (max_eof_state = 0) {    ▷ in case the user has not declared any states ◁
      max_eof_state ⇐ YY_STATE_EOF(INITIAL);
    }
    for ( ; i ⩽ max_eof_state; i++) {
      fprintf(tables_out, action_desc.act_setup, i);
      if (^not bare_actions) {
        fprintf(tables_out, "\n");
        ((struct yyguts_t *) fake_scanner)→yy_hold_char ⇐ 0;
        yy_accept[1] ⇐ i;
        yylex(Λ, fake_scanner);
      }
      fprintf(tables_out, action_desc.act_suffix, i);
    }
    fprintf(tables_out, "%s", action_desc.postamble);
    if (action_desc.cleanup) {
      action_desc.cleanup(&action_desc);
    }
  }
  ⟨ Compute magic constants 300 ⟩
  ⟨ Output states 302 ⟩;
  fprintf(tables_out, "\\constset{YYECMAGIC}{%d}%%\n", yy_ec_magic);
  fprintf(tables_out, "\\constset{YYMAXEOFSTATE}{%d}%%\n", max_eof_state);
```

298 ⟨ Error codes 234 ⟩ +=

```
BAD_SCANNER,
```

299 ⟨ Variables and types local to the scanner driver 295 ⟩ +=

```
int yy_ec_magic;
```

300 The 'magic' constants are similar to the 'exotic' ones mentioned above except the methods used to compute them rely on reverse engineering the scanner function. Since this changes the scanner tables it has to be done after the 'driver' has finished going through all the actions.

⟨ Compute magic constants 300 ⟩ =

```
{
  int i, j;
  char fake_yytext[YY_MORE_ADJ + 1];
  yyscan_t yyscanner;
  struct yyguts_t *yyg;
  if (yylex_init(&yyscanner)) {
    printf("Cannot␣initialize␣the␣scanner\n");
    exit(BAD_SCANNER);
  }
```

```
    yyg ⇐ (struct yyguts_t *) yyscanner;
    yyg→yy_start ⇐ 0;
    yy_set_bol(0);
    yyg→yytext_ptr ⇐ fake_yytext;
    yyg→yy_c_buf_p ⇐ yyg→yytext_ptr + 1 + YY_MORE_ADJ;
    fake_yytext[YY_MORE_ADJ] ⇐ 0;    ▷ *yy_cp ⇐ 0; ◁
    yy_accept[0] ⇐ 0;
    yy_base[0] ⇐ 0;
    for (i ⇐ 0; i < sizeof (yy_chk)/sizeof (yy_chk[0]); i++) {
      yy_chk[i] ⇐ 0;
    }
    for (i ⇐ 0; i < sizeof (yy_nxt)/sizeof (yy_nxt[0]); i++) {
      yy_nxt[i] ⇐ i;
    }
    yy_ec_magic ⇐ yy_get_previous_state(yyscanner);
  }
```

This code is used in section 297.

301 State names

There is no easy way to output the symbolic names for states, so this has to be done by hand while actions are output. The state names are accumulated in a list structure and are printed out after action output is complete.

Note that parsing the scanner file would not help (even though the extended lexer and scanner can recognize the `%x` option). All it can do is output the state *names* but not their numerical values, since the state names are macros and their values are only known to the `flex` generated scanner.

```
#define Define_State(st_name, st_num) do {
          struct lexer_state_d *this_state;
          this_state ⇐ malloc(sizeof(struct lexer_state_d));
          this_state→name ⇐ st_name;
          this_state→value ⇐ st_num;
          this_state→next ⇐ Λ;
          if (last_state) {
            last_state→next ⇐ this_state;
            last_state ⇐ this_state;
          }
          else {
            last_state ⇐ state_list ⇐ this_state;
          }
          if (YY_STATE_EOF(st_num) > max_eof_state) {
            max_eof_state ⇐ YY_STATE_EOF(st_num);
          }
        } while (0);
```

⟨ Scanner variables and types for C preamble 301 ⟩ =

```
int max_eof_state ⇐ 0;
struct lexer_state_d {
  char *name;
  int value;
  struct lexer_state_d *next;
};
struct lexer_state_d *state_list ⇐ Λ;
struct lexer_state_d *last_state ⇐ Λ;
```

302 ⟨ Output states 302 ⟩ =

```
{
  struct lexer_state_d *current_state;
  struct lexer_state_d *next_state;
  current_state ⇐ next_state ⇐ state_list;
  if (current_state) {
    fprintf(tables_out, "\\def\\setflexstates{%%\n""␣␣\\stateset{INITIAL}{%d}%%\n", INITIAL);
    while (current_state) {
      fprintf(tables_out, "␣␣\\stateset{%s}{%d}%%\n", current_state→name, current_state→value);
      current_state ⇐ current_state→next;
      free(next_state);
      next_state ⇐ current_state;    ▷ the name field is not deallocated because it is not allocated on the heap ◁
    }
    fprintf(tables_out, "}%%\n%%\n");
  }
}
```

This code is used in section 297.

303 Constants

⟨ Scanner constants 303 ⟩ =

```
_register_const_d(YY_END_OF_BUFFER_CHAR)
_register_const_d(YY_NUM_RULES)
_register_const_d(YY_END_OF_BUFFER)
```

This code is used in section 311.

304 Output modes

The output modes are the same as in the case of the parser with minor changes.

305 Generic output

Generic output is not programmed yet.

⟨ Scanner specific output modes 305 ⟩ =

```
GENERIC_OUT,
```

See also section 307.

306 ⟨ Handle scanner output modes 306 ⟩ =

```
case GENERIC_OUT:
  printf("This␣mode␣is␣not␣supported␣yet\n");
  exit(0);
  break;
```

See also section 308.

307 TEX mode

The TEX mode is the main focus of this software.

⟨ Scanner specific output modes 305 ⟩ +=

```
TEX_OUT,
```

308 ⟨ Handle scanner output modes 306 ⟩ +=

```
case TEX_OUT:
  ⟨ Set up TEX format for scanner tables 309 ⟩
  ⟨ Set up TEX format for scanner actions 310 ⟩
  ⟨ Prepare TEX format for scanner constants 311 ⟩
  break;
```

309 ⟨ Set up TEX format for scanner tables 309 ⟩ =

```
tex_table_generic(yy_accept);
yy_accept_desc.name ⇐ "yyaccept";
tex_table_generic(yy_ec);
yy_ec_desc.name ⇐ "yyec";
tex_table_generic(yy_meta);
yy_meta_desc.name ⇐ "yymeta";
tex_table_generic(yy_base);
yy_base_desc.name ⇐ "yybase";
tex_table_generic(yy_def);
yy_def_desc.name ⇐ "yydef";
tex_table_generic(yy_nxt);
yy_nxt_desc.name ⇐ "yynxt";
tex_table_generic(yy_chk);
yy_chk_desc.name ⇐ "yychk";
```

This code is used in section 308.

310 ⟨ Set up TEX format for scanner actions 310 ⟩ =

```
if (optimize_actions) {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax\n%\n"
  "\\def\\yydoactionswitch#1{%%\n"
  "␣␣␣␣\\let\\yylextail\\yylexcontinue\n"
  "␣␣␣␣\\csname␣doflexaction\\number␣#1\\parsernamespace\\endcsname\n"
  "␣␣␣␣\\yylextail\n"
  "}\\stashswitch{yydoactionswitch}%\n";
  action_desc.act_setup ⇐ "\n\\expandafter\\def\\csname␣doflexaction%d\\parsernamespac\
      e\\endcsname{%%\n""␣␣␣␣␣␣␣\\YYRULESETUP";
  action_desc.act_suffix ⇐ "}%%␣end␣of␣rule␣%d\n";
  action_desc.action1 ⇐ Λ;
  action_desc.actionn ⇐ Λ;
  action_desc.postamble ⇐ "\\catcode`\\/=12\\relax\n%\n";
  action_desc.print_rule ⇐ Λ;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
else {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax\n%\n"
  "\\def\\yydoactionswitch#1{%%\n␣␣\\let\\yylextail\\yylexcontinue\n"
  "␣␣\\ifcase#1\\relax\n";
  action_desc.act_setup ⇐ "␣␣␣␣␣␣␣\\or\n""␣␣␣␣␣␣␣\\YYRULESETUP␣%%␣(rule␣%d)␣";
  action_desc.act_suffix ⇐ "␣␣␣␣␣␣␣%%␣end␣of␣rule␣%d\n";
  action_desc.action1 ⇐ Λ;
  action_desc.actionn ⇐ Λ;
  action_desc.postamble ⇐ "␣␣\\else\n␣␣\\fi\n␣␣\\yylextail\n}\\stashswitch{yydoactions\
      witch}%\n\\catcode`\\/=12\\relax\n%\n";
  action_desc.print_rule ⇐ Λ;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
```

This code is used in section 308.

311 TEX constant output is another place where the techniques described above are applied. A few names have to be handled separately, because of the underscores in their names.

⟨ Prepare TEX format for scanner constants 311 ⟩ =

```
#define _register_const_d(c_name)  c_name##_desc.format ⇐ "\\constset{%s}{%d}%%\n";
```

```
  c_name##_desc.name ⇐ #c_name;
  output_desc.output_##c_name ⇐ 1;
  ⟨ Scanner constants 303 ⟩
#undef _register_const_d
  YY_END_OF_BUFFER_CHAR_desc.name ⇐ "YYENDOFBUFFERCHAR";
  YY_NUM_RULES_desc.name ⇐ "YYNUMRULES";
  YY_END_OF_BUFFER_desc.name ⇐ "YYENDOFBUFFER";
```

This code is used in section 308.

312 ⟨ Output exotic scanner constants 312 ⟩ =

```
fprintf(tables_out, "\\constset{YYMAXREALCHAR}{%ld}%%\n", sizeof (yy_accept)/(sizeof (yy_accept[0])) − 1);
fprintf(tables_out, "\\constset{YYBASEMAXENTRY}{%d}%%\n", max_yybase_entry);
fprintf(tables_out, "\\constset{YYNXTMAXENTRY}{%d}%%\n", max_yynxt_entry);
fprintf(tables_out, "\\constset{YYMAXRULENO}{%d}%%\n", max_yyaccept_entry);
fprintf(tables_out, "\\constset{YYECMAXENTRY}{%d}%%\n", max_yy_ec_entry);
```

313 Command line options

We start with the most obvious option, the one begging for help.

⟨ Higher index scanner specific options 313 ⟩ =

```
LONG_HELP,
```

314 ⟨ Scanner specific option list 314 ⟩ =

```
_register_option("help", no_argument, 0, LONG_HELP, "")
```

See also section 317.

315 ⟨ Shortcuts for command line options affecting scanner output 315 ⟩ =

```
"h"
```

See also section 318.

316 ⟨ Handle scanner output options 316 ⟩ =

```
case 'h':    ▷ short help ◁
  fprintf(stderr, "Usage: %s [options] output_file\n", argv[0]);
  exit(0);
  break;    ▷ should not be needed ◁
case LONG_HELP:
  fprintf(stderr,
      "%s [--mode=TeX:options] output_file outputs tables\n""    and constants for a TeX parser\n",
      argv[0]);
  exit(0);
  break;    ▷ should not be needed ◁
```

See also section 320.

317 ⟨ Scanner specific option list 314 ⟩ +=

```
_register_option("debug", optional_argument, 0, 'b', "")
_register_option("mode", required_argument, 0, 'm', "")
_register_option("table-separator", required_argument, 0, 'z', "")
_register_option("format", required_argument, 0, 'f', "")    ▷ name? ◁
_register_option("table", required_argument, 0, 't', "")    ▷ specific table ◁
_register_option("constant", required_argument, 0, 'c', "")    ▷ specific constant ◁
_register_option("name-length", required_argument, 0, 'l', "")    ▷ change MAX_NAME_LENGTH ◁
_register_option("token", required_argument, 0, 'n', "")    ▷ specific token ◁
_register_option("run-parse", required_argument, 0, 'p', "")    ▷ run the parser ◁
_register_option("parse-file", required_argument, 0, 'i', "")    ▷ input for the parser ◁
```

318 The string below is a list of short options.

⟨ Shortcuts for command line options affecting scanner output 315 ⟩ +=

```
"b::z:m:f:t:"
```

319 A few options can be immediately discussed.

⟨ Variables and types local to the scanner driver 295 ⟩ +=

```
int debug_level ⇐ 0;
char *table_separator ⇐ "%s␣";
```

320 ⟨ Handle scanner output options 316 ⟩ +=

```
case 'b': ▷ debug (level) ◁
  debug_level ⇐ optarg ? atoi(optarg) : 1;
  break;
case 'm': ▷ output mode ◁
  switch (optarg[0]) {
  case 'T': case 't':
    mode ⇐ TEX_OUT;
    break;
  case 'b': case 'B': case 'g': case 'G':
    mode ⇐ GENERIC_OUT;
    break;
  default:
    break;
  }
  break;
case 'z': table_separator ⇐ (char *) malloc((strlen(optarg) + 1) * sizeof(char));
  strcpy(table_separator, optarg);
  break;
```

321 Philosophy

This section should, perhaps, be more appropriately called *rant* but *philosophy* sounds more academic. The design of any software involves numerous choices, and `SPLinT` is no exception. Some of these choices are explained in the appropriate places in the package files. This section collects a few 'big picture' choices that did not fit elsewhere.

322 Why GPL

The choice of license for this project goes beyond merely showing the source. TEX, by its very nature is an open source language, so it is not a matter of hiding anything from the user or a potential developer. The C code is a different matter but the source is not that complicated. Reducing the licensing issue to the ability of someone else to see the source code is a great oversimplification. Without getting into too many details of so-called 'open source licenses' (other than GPL) and arguing with their advocates, let me simply express my lack of understanding at the arguments that purport that BSD-style licenses introduce more freedom by allowing a software vendor to incorporate the BSD-licensed software into their products. What benefit does one derive from such 'extension' of software freedom? Perhaps the hope that the 'open source' (for the lack of a better term) will prompt the vendor to follow the accepted free (or any other, for that matter!) software standards and make its software more interoperable with the free alternatives? A well-known software giant's *embrace, extend, extinguish* philosophy shows how naïve and misplaced such hopes are.

I am not going to argue for the benefits of free software at length, either (such benefits seem self-evident to me, although the readers should feel free to disagree). Let me just point out that software companies enjoy quite a few freedoms that we, as software consumers elect to afford them. Among such freedoms are the ability to renege on any promises made to potential users and withdraw any guarantees that such users might enjoy. Free software, of course, does not provide any guarantees, either but 'you get what you paid for'. As a result of such 'release of any responsibility', the claims of increased reliability or better support for the commercial software sound a bit hollow. Another well spread tactic is user brainwashing and changing the culture (usually for the worse) in order to promote new 'user-friendly' features of commercial software. Instead of taking advantage of computers as cognitive machines we have come to view them as advanced media players that we interact with through artificial, unnatural interfaces. Meaningless terminology ('UX' for 'user experience'? What in the world is 'user experience'?) proliferates, and programmers are happy to deceive themselves with their newly discovered business prowess.

One would hope that the somewhat higher standards of the 'real' manufacturers might percolate to the software world, however, the reality is very different. Not only has life-cycle 'engineering' got to the point where manufacturers can predict the life spans of their products precisely, embedded software in those products has become an enabling technology that makes this 'life design' much easier.

In effect, by embedding software in their products, hardware manufacturers not only piggy-back on software's perceived complexity, and argue that such complex systems cannot be made reliable, they have an added incentive to uphold this image. The software weighs nothing, memory is cheap, consumers are easy to deceive, thus 'software is expensive' and 'reliable software is prohibitively so'. Designing reliable software is quite possible, though, just look at programmable thermostats, simple cellphones and other 'invisible' gadgets we enjoy. The 'software ideology' with its 'IP' lingo is spreading like a virus even through the world of real things. We now expect products to break and are too quick to forgive sloppy engineering that goes into everyday things. We are also getting used to the idea that it is the manufacturers that get to dictate the terms of use for 'their' products and that we are merely borrowing 'their' stuff.

The GPL was conceived as an antidote to this scourge. This document is a remarkable piece of 'legal engineering': a self-propagating license with a clearly outlined set of goals. While by itself it does not guarantee reliability or quality, it does inhibit the spread of the 'IP' (which is sometimes sarcastically, though quite perceptively, 'deabbreviated' as *I*maginary *P*roperty) disease through software.

The industry has adapted, of course. So called (non GPL) 'open source licenses', that are supposed to be an improvement on GPL, are a sort of 'immune reaction' to the free software movement. Convince and confuse enough apathetic users and the protections granted by GPL are no longer visible.

323 Why not C++ or OOP in general

The choice of the language was mainly driven by æsthetic motives: C++ has a bloated and confusing standard, partially supported by various compilers. It seems that there is no agreement on what C++ really is or how to use some of its constructs. This is all in contrast to C with its well defined and concise body of specifications and rather well established stylistics. The existence of 'obfuscated C' is not good evidence of deficiency and C++ is definitely not immune to this malady.

Object oriented design has certainly taken on an aura of a religious dictate, universally adhered to and forcefully promoted by its followers. Unfortunately, the definition of what constitutes an 'object-oriented' approach is rather vague. A few abstract concepts are commonly tossed about to give the illusion of a well developed abstraction (such as 'polymorphism', 'encapsulation', and so on) but definitions vary in both length and contents, depending on the source.

On a syntactic level, some features of object-oriented languages are undoubtedly very practical (such as a **this** pointer in C++), however, many of those features can be effectively emulated with some clever uses of an appropriate preprocessor (there are a few exceptions, of course, **this** being one of them). The rest of the 'object-oriented philosophy' is just that: a design philosophy. Before that we had structured programming, now there are patterns, extreme, agile, reactive, etc. They might all find their uses, however, there are always numerous exceptions (sometimes even global variables and **goto**'s have their place, as well).

A pedantic reader might point out a few object-oriented features even in the TEX portion of the package and then accuse the author of being 'inconsistent'. I am always interested in possible improvements in style but I am unlikely to consider any changes based solely on the adherence to any particular design fad.

In short, OOP was not shunned simply because a 'non-OOP' language was chosen, instead, whatever approach or style was deemed most effective was used. The author's judgment was not always perfect, of course, and given a good reason, changes can be made, including the choice of the language. 'Make it object-oriented' is neither a good reason nor a clearly defined one, however.

324 Why not ∗TEX

Simple. I never use it and have no idea of how packages, classes, etc., are designed. I have heard it has impressive mechanisms for dealing with various problems commonly encountered in TEX. Sadly, my knowledge of ∗TEX machinery is almost nonexistent. This may change but right now I have tried to make the macros as generic as possible, hopefully making ∗TEX adaptation easy.

The following quote from [Ho] makes me feel particularly uneasy about the current state of development of various TEX variants: "*Finally, to many current programmers* `WEB` *source simply feels over-documented and even more important is that the general impression is that of a finished book: sometimes it seems like* `WEB` *actively discourages development. This is a subjective point, but nevertheless a quite important one.*"

Discouraging development seems like a good thing to me. Otherwise we are one step away from encouraging writing poor software with inadequate tools merely 'to encourage development'.

The feeling of a `WEB` source being *over-documented* is most certainly subjective, and, I am sure, not shared by all 'current programmers'. The advantage of using `WEB`-like tools, however, is that it gives the programmer the ability to place the vital information where it does not distract the reader ('developer', 'maintainer', call it whatever you like) from the logical flow of the code.

Some of the complaints in [Ho] are definitely justified, although it seems that a better approach would be to write an improved tool similar to `WEB`, rather than give up all the flexibility such a tool provides.

325 Why `CWEB`

`CWEB` is not as polished as TEX but it works and has a number of impressive features. It is, regrettably, a 'niche' tool and a few existing extensions of `CWEB` and software based on similar ideas do not enjoy the popularity they deserve. Literate philosophy has been largely neglected even though it seems to have a more logical foundation than OOP. Under these circumstances, `CWEB` seemed to be the best available option.

326 Why not GitHub, Bitbucket, etc

Git is an incredible tool and is used extensively in the development of `SPLinT`. The distribution archive is a Git repository. The use of centralized services such as GitHub, however, seems redundant. The standard cycle, 'clone-modify-create pull request' works the same even when 'clone' is replaced by 'download'. Thus, no functionality is lost. This might change if the popularity of the package unexpectedly increases.

On the other hand, GitHub and its cousins are commercial entities, whose availability in the future is not guaranteed (nothing is certain, of course, no matter what distribution method is chosen). Keeping `SPLinT` as an archive of a Git repository seems like an efficient way of being ready for an unexpected change.

327 Bibliography

This list of references is not meant to be exhaustive or complete. These are merely the papers and the books mentioned in the body of the program above. Naturally, this project has been influenced by many outside ideas but it would be impossible to list them all due to time and (human) memory limitations.

* * *

[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.

[Do] Jean-luc Doumont, *Pascal pretty-printing: an example of "preprocessing with TEX"*, TUGboat **15** (3), 1994—Proceedings of the 1994 TUG Annual Meeting

[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.

[Go] Pedro Palao Gostanza, *Fast scanners and self-parsing in TEX*, TUGboat **21** (3), 2000—Proceedings of the 2000 Annual Meeting.

[Gr] Andrew Marc Greene, *BASIX—an interpreter written in TEX*, TUGboat **11** (3), 1990—Proceedings of the 1990 TUG Annual Meeting.

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

[Ho] Taco Hoekwater, *LuaTEX says goodbye to Pascal*, TUGboat **30** (3), pp. 136–140, 2009—EuroTEX 2009 Proceedings.

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

[ISO/C11] *ISO/IEC 9899—Programming languages—C (C11)*, December 2011, draft available at http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

[Jo] Derek M. Jones, *The New C Standard: An Economic and Cultural Commentary*, available at http://www.knosof.co.uk/cbook/cbook.html.

[La] *The* `l3regex` *package: regular expressions in TEX*, The LATEX3 Project.

[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.

328 Index. This section is, perhaps, the most valuable product of CWEB's labors. It lists references to definitions (set in *italic*) as well as uses for each C identifier used in the source. Special facilities have been added to extend indexing to `bison` grammar terms and TEX control sequences encountered in `bison` actions. Definitions of tokens (via ⟨`token`⟩, ⟨`nterm`⟩ and ⟨`type`⟩ directives) are underlined. The `bison` and TEX entries are put in distinct sections of the index in order to keep the separation between the C entries and the rest. It may be worth noting that the *definition* of the symbol is listed under both its 'macro name' (such as `CHAR`, typeset as **char** in the case of the grammar below), as well as its 'string' name if present (to continue the previous example, `"char"` is synonymous with **char** after a declaration such as '⟨`token`⟩ **char** `"char"`'), while the *use* of the term lists whichever token form was referenced at the point of use (both forms are accessible when the entry is typeset for the index and a macro can be written to mention the other form as well). The quotes indicate that the 'string' form of the token's name was used. A section set in *italic* references the point where the corresponding term appeared on the left hand side of a production.

A production:

left_hand_side :
term$_1$ *term*$_2$ *term*$_3$ `\do \something` Υ_1

inside the TEX part of a `CWEB` section will generate several index entries, as well, including the entries for any material inside the action, mimicking `CWEB`'s behavior for the *inline* C (`|...|`). Such entries (except for the references to C code inside actions) are labeled with °, to provide a reminder of their origin.

This parser collection, as well as the indexing facilities therein have been designed to showcase the broadest range of options available to the user and thus it does not always exhibit the most sane choices one could make (for example, using a full blown parser for term *names* is poor design but it was picked to demonstrate multiple parsers in one program). The same applies to the way the index is constructed (it would be easy to agree to only use the 'string' name of the token if it is available, thus avoiding referencing the same token in two different parts of the index).

TEX control sequences are listed following the index of all `bison` entries. The two indices are separated by a *dinkus* (∗ ∗ ∗). Since it is nearly impossible to determine at what point a TEX macro is defined (and most of them are defined outside of the `CWEB` sources), only their uses are listed (to be more precise, *every* appearance of a macro is assumed to be its use). In a few cases, a 'graphic' representation for a control sequence is also listed (for example, π_1 represents `\getfirst`). The index entries are ordered alphabetically using control sequence names.

Bison and TEX index

A LIST OF ALL SECTIONS

⟨ Local variable and type declarations 207, 210, 221, 225, 237, 241 ⟩ Used in section 201.
⟨ Long options array 244 ⟩ Used in section 242.
⟨ Make an empty right hand side 72 ⟩ Used in section 71.
⟨ Name parser C postamble 187 ⟩ Used in section 165.
⟨ Name parser C preamble 186 ⟩ Used in section 165.
⟨ Outer definitions 203, 240 ⟩ Used in section 201.
⟨ Output action switch, if any 232 ⟩ Used in section 201.
⟨ Output all tables 214 ⟩ Used in section 213.
⟨ Output constants 229 ⟩ Used in section 228.
⟨ Output descriptor fields 208, 219, 226 ⟩ Used in section 207.
⟨ Output exotic scanner constants 312 ⟩
⟨ Output modes 238 ⟩ Used in section 237.
⟨ Output parser constants 265 ⟩
⟨ Output parser semantic actions 256 ⟩
⟨ Output parser tokens 264 ⟩
⟨ Output scanner actions 297 ⟩
⟨ Output states 302 ⟩ Used in section 297.
⟨ Parser bootstrap productions 52, 57, 58, 82, 86 ⟩ Used in sections 22 and 51.
⟨ Parser common productions 44, 47, 51, 53, 54, 56, 83, 94 ⟩ Used in sections 21, 23, and 24.
⟨ Parser constants 257 ⟩ Used in section 283.
⟨ Parser defaults 252 ⟩
⟨ Parser full productions 29 ⟩ Used in section 24.
⟨ Parser grammar productions 60, 61, 71, 85 ⟩ Used in sections 21 and 24.
⟨ Parser productions 168 ⟩ Used in section 165.
⟨ Parser prologue productions 35, 37, 39, 93 ⟩ Used in sections 23 and 24.
⟨ Parser specific default outputs 263 ⟩
⟨ Parser specific option list 259, 269, 286, 289 ⟩
⟨ Parser specific output descriptor fields 262 ⟩
⟨ Parser specific output modes 267, 273, 275 ⟩
⟨ Parser table names 250, 255 ⟩
⟨ Perform output 213, 228 ⟩ Used in section 201.
⟨ Possbly complain about a bad directive 127 ⟩ Used in section 118.
⟨ Prepare TEX format for parser constants 283 ⟩ Used in section 276.
⟨ Prepare TEX format for parser tokens 284 ⟩ Used in section 276.
⟨ Prepare TEX format for scanner constants 311 ⟩ Used in section 308.
⟨ Prepare TEX format for semantic action output 281 ⟩ Used in section 276.
⟨ Prepare a string for use 92 ⟩ Used in section 86.
⟨ Prepare an identifier 128 ⟩ Used in section 118.
⟨ Prepare one parametric option 38 ⟩ Used in sections 37 and 44.
⟨ Prepare the left hand side 91 ⟩ Used in section 85.
⟨ Prepare to process an identifier 198 ⟩ Used in section 197.
⟨ Prepare token only output environment 272 ⟩ Used in section 268.
⟨ Prepare union definition 48 ⟩ Used in section 47.
⟨ Process a bad character 120 ⟩ Used in section 118.
⟨ Process a character after an identifier 135 ⟩ Used in section 132.
⟨ Process a colon after an identifier 134 ⟩ Used in section 132.
⟨ Process bracketed identifier 138 ⟩ Used in section 137.
⟨ Process command line options 243 ⟩ Used in section 201.
⟨ Process the bracketed part of an identifier 133 ⟩ Used in section 132.
⟨ Raise nesting level 153 ⟩ Used in section 151.
⟨ Raw option list 217 ⟩ Used in section 244.
⟨ React to a bad character 199 ⟩ Used in section 197.

⟨ Turn a character into a term 88 ⟩ Used in section 82.
⟨ Turn a qualifier into an identifier 173 ⟩ Used in section 168.
⟨ Turn a string into a symbol 90 ⟩ Used in section 84.
⟨ Turn an identifier into a symbol 89 ⟩ Used in section 84.
⟨ Turn an identifier into a term 87 ⟩ Used in section 82.
⟨ Union of grammar parser types 100 ⟩ Used in sections 21, 22, 23, and 24.
⟨ Union of parser types 188 ⟩ Used in section 165.
⟨ Variables and types local to the parser 251, 258, 291 ⟩
⟨ Variables and types local to the scanner driver 295, 299, 319 ⟩
⟨ Various output modes 202 ⟩ Used in section 201.
⟨ C postamble 201 ⟩ Cited in section 201.
⟨ C preamble 222 ⟩
⟨ `flex` options parser productions 40 ⟩ Used in sections 22 and 39.
⟨ `bb.yy` 22 ⟩
⟨ `bd.yy` 23 ⟩
⟨ `bf.yy` 24 ⟩
⟨ `bg.yy` 21 ⟩
⟨ `lo.ll` 101 ⟩
⟨ `sill.y` 11 ⟩
⟨ `small_lexer.ll` 189 ⟩
⟨ `small_parser.yy` 165 ⟩

Contents (splint)