1

Introduction

3a SPLinT \(^1\) (Simple Parsing and Lexing in \TeX{}, or, following the great GNU tradition of creating recursive names, SPLinT Parses Languages in \TeX{} or, following the great GNU tradition of creating recursive names, \textsc{SPLinT}) is a system (or rather a mélange of systems) designed to facilitate the development of parsing macros in \TeX{} and (to a lesser degree) to assist one in documenting parsers written in other languages. As an application, parsers for \texttt{bison} and \texttt{flex} input file syntax have been developed, along with a macro collection that makes it possible to design and pretty print \(^2\) \texttt{bison} grammars and \texttt{flex} automata using \texttt{CWEB}. The \texttt{examples} directory contains a few other parsers designed to pretty print various languages (among them is \texttt{ld}, the language of the GNU linker).

3b \texttt{CWEB} and literate programming

Writing software in \texttt{CWEB} involves two programs. The first of these is \texttt{CTANGLE} that outputs the actual code, intended to be in C. In reality, \texttt{CTANGLE} cares very little about the language it produces. Among the exceptions are C comments and \#line directives that might confuse lesser software but \texttt{bison} is all too happy to swallow them (there are also some C specific constructs that \texttt{CTANGLE} tries to recognize). \texttt{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 \texttt{CTANGLE} to produce \texttt{bison} output is some very rudimentary post- and pre-processing.

Our main concern is thus \texttt{CWEAVE} that not only pretty prints the program but also creates an index, cross-references all the sections, etc. Getting \texttt{CWEAVE} to pretty print a language other than C requires some additional attention. A true digital warrior would probably try to decipher \texttt{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 \texttt{CWEB} command: the verbatim (@=...@>) output. The material to be output by this construct undergoes minimal processing and is put inside \verb!...! All that is needed now is a way to process this virtually straight text inside \TeX{}.

This manual, as well as nearly every other document that accompanies SPLinT is itself a source for a computer program (or, as is the case with this document, several programs) that is extracted using \texttt{CTANGLE}. We refer an interested reader to \cite{CWEB} for a detailed description of the syntax and use patterns of \texttt{CWEB}. The following is merely a brief overview of the approach.

Every \texttt{CWEB} document is split into \texttt{sections}, each divided into three parts (any one of which can be empty): the \TeX{} part, the middle part, and the C part (which should more appropriately be called the \texttt{code part}).

---

\(^1\) I was tempted to call the package \texttt{ParLALR}gram which stands for Parsing LALR Grammars or \texttt{PinT} for ‘Parsing in \TeX{}’ but both sounded too generic.  
\(^2\) The term \texttt{pretty printing} is used here in its technical sense as one might find that there is nothing pretty about the output of the parsing routines presented in this document.
The code part of each section carries a name for cross referencing purposes. The sections themselves are automatically numbered by `CWEAVE` and their code parts may be referenced from other sections, as well as included in other sections’ code parts using `CWEB`’s cross referencing syntax (such as `⟨ A production sa ⟩`). Using the same name for the C portion in several sections has the effect of merging the corresponding code fragments. When the section with such a name is used (included) later, all of the concatenated fragments are included as well, even the ones that appear after the point in the `CWEB` document where such inclusion takes place.

The original `CWEB` macros (from `cwebmac.tex`) used the numbers generated by `CWEAVE` to refer to specific sections. This was true for the table of contents, as well as the index entries. The macros used by SPLinT adopt a different convention, proposed by N. Ramsey for his literate programming software, `noweb`. In the new system (which will be referred to as the `noweb` style of cross referencing), each section is labelled by the page number where it starts and an alphabetic character that indicates the order of appearance of the section on the page. Also following `noweb`, the new macros display links between the fragments of the same section in the margins. This allows for quicker navigation between sections of the code and lets the reader get a quick overview of what gets ‘collected’ in a given section.

The top level (```@**```), introducing major portions of the code have also been given more prominent appearance and carry a chapter number in addition to the the `noweb` style section number (the latter is used for cross references, as the chapter number gives no indication as to where the said chapter is located).

`CWEB` also generates an index of all the identifiers (with some exceptions, such as single letter names) appearing in the C portion of each section, except those that appear inside the `verbatim` portions of the code (i.e. between ```@=` and `@>```). Since SPLinT uses the verbatim blocks extensively, additional indexing facilities have been implemented to provide indexing for the non-C languages handled by various SPLinT parsers.

## 4a Pretty (and not so pretty) printing

Pretty-printing can be narrowly defined as a way to organize the presentation of the program’s text. The range of visual devices used for this purpose is usually limited to indentation and discrete line skips, to mimic the capabilities of an old computer terminal. Some authors (see [ACM]) have replaced the term pretty printing with program visualization to refer to a much broader range of graphic tools for translating the code (and its meaning) into a richer medium. This manual uses the terms pretty printing and program visualization interchangeably.

Pretty printing in the broader sense above has been the subject of research for some time. The monograph [ACM] develops a methodical (if not formalized) approach to the design of visualization frameworks for programming languages (although the main focus is on procedural C-like languages).

A number of papers about pretty printing have appeared since, extending the research to new languages, and suggesting new visualizing rules. Unfortunately, most of this research is driven by rules of thumb and anecdotes (the approach fully embraced by this manual), although there have been a few rigorous studies investigating isolated visualization techniques (see, for example, the discussion of variable declaration placement in [Jo]).

Perhaps the only firm conclusion one can draw from this discussion is that writing the code and reading it are very different activities so facilitating the former may in turn make the latter more difficult and vice versa. Some well known languages try to arrive at a compromise where the syntax forces a certain style of presentation on the programmer. An example of a successful language in this group is Python with its meaningful white space. The author does not share the enthusiasm some programmers express for this approach.

On the other hand, a language like C does not enforce any presentation format. The authors of C even remarked that semicolons and braces were merely a nod to the compiler (or, one might add, static analysis software, see [KR]). It may thus seem reasonable that such redundant syntax elements may be replaced by different typographic devices (such as judicially chosen skips and indentation, or the choice of fonts) when (pretty) printing the code.

---

1) With the exception of the nameless ```@c``` sections.  
2) The ‘feature’ so masterfully exploited by the International Obfuscated C Code Contest (IOCCC) participants.
Even the critics of pretty printing usually concede that well indented code is easier to read. The practice of using different typefaces to distinguish between various syntactic elements (such as reserved words and general identifiers) is a subject of some controversy, although not as pronounced as some of the more drastic approaches (such as completely replacing the brace pairs with indentation as practiced by SPLinT for bison input or by the authors of ACM for the control statements in C).

The goal of SPLinT was not to force any particular ‘pretty printing philosophy’ on the programmer (although, if one uses the macros ‘as is’, some form of quiet approval is assumed ...) but rather to provide one with the tools necessary to implement one’s own vision of making the code readable.

One tacit assumption made by the author is that an integral part of any pretty printing strategy is extracting (some) meaning from the raw text. This is done by parsing the program, the subject we discuss next. It should be said that it is the parser design in TeX that SPLinT aims to facilitate, with pretty printing being merely an important application.

5a Parsing and parsers

At an abstract level, a parser is just a routine that transforms text. Naturally, not every possible transformation is beneficial, so, informally, the value of a parser lies in its ability to expose some meaning in the text. If valid texts are reduced to a small finite set (while each text can be arbitrarily long) one can concievably write a primitive string matching algorithm that recognizes whether any given input is an element of such set, and if it is, which one. Such ‘parsers’ would be rather limited and are only mentioned to illustrate the point that, in general, the texts being parsed are not required to follow any particular specification.

In practice, however, real world parsers rely on the presence of some structure in the input to do their work. The latter can be introduced by supplying a formal (computable) description of every valid input. The ‘rigidity’ of this specification directly affects the sophistication of the parsing algorithm required to process a valid input (or reject an invalid one).

Parsing algorithms normally follow a model where the text is processed a few symbols at a time and the information about the symbols already seen is carried in some easily accessible form. ‘A few symbols at a time’ often translates to ‘at most one symbol’, while ‘easily accessible’ reduces to using a stack-like data structure for bookkeeping.

A popular way of specifying structure is by using a formal grammar1) that essentially expresses how some (preferably meaningful) parts of the text relate to other parts. Keeping with the principle of making the information about the seen portions of the input easily accessible, practical grammars are normally required to express the meaning of a fragment in a manner that does not depend on the input that surrounds the fragment (i.e. to be context-free). Real-world languages rarely satisfy this requirement2) thus presenting a challenge to parser generating software that assumes the language is context-free.

Even the task of parsing all context-free languages is too ambitious in most practical scenarios, so further limitations on the grammar are normally imposed. One may require that the next action of the parsing algorithm must depend exclusively on the next symbol seen and one of the finitely many states the parser may be in. The action here simply refers to the choice of the next state, as well as the possible decision to consume more input or output a portion of the abstract syntax tree which is discussed below.

The same language may have more than one grammar and the choice of the latter normally has a profound effect on the selection of the parsing algorithm. Without getting too deep into the parsing theory, consider the following simple sketch.

\[
\begin{align*}
pexp & : \ ( pexp ) \ | \ astring \\
aststring & : \ \circ \ | \ \ast \ astring
\end{align*}
\]

Informally, the language consists of ‘strings of \(n\ \ast\)’s nested \(m\) parentheses deep’. After parsing such a string, one might be interested in the values of \(m\) and \(n\).

The three states the parser may be in are ‘start’, ‘parsing \(pexp\)’ and ‘parsing \(aststring\)’. A quick glance at the grammar above shows that switching between the states is straightforward (we omit the discussion of

---

1) While popular, formal grammars are not the only way of describing a language. For example, ‘powers of 2 presented in radix 3’ is a specification that cannot be defined by a context-free grammar, although it is possible to write a (very complex) grammar for it. 2) Processing typedef’s in C is a well known case of such a language defect.
the ‘start’ state for brevity): if the next symbol is (, parse the next \texttt{pexp}, otherwise, if the next symbol is *, parse \texttt{astring}. Finally, if the next symbol is ) and we are parsing \texttt{pexp}, finish parsing it and look for the next input, otherwise, we are parsing \texttt{astring}, finish parsing it, make it a \texttt{pexp}, finish parsing a \texttt{pexp} started by a parenthesis, and look for more input. This unnecessarily long (as well as incomplete and imprecise) description serves to present a simple fact that the parsing states are most naturally represented by individual \textit{functions} resulting in what is known as a \textit{recursive descent parser} in which the call stack is the ‘data structure’ responsible for keeping track of the parser’s state. One disadvantage of the algorithm above is that the maximal depth of the call stack reaches \( m + n \) which may present a problem for longer strings.

Computing \( m \) and \( n \) above now reduces to incrementing an appropriate variable upon exiting the corresponding function. More important, however, is the observation that this parsing algorithm can be extracted from the grammar in a very straightforward fashion. To better illustrate the rôle of the grammar in the choice of the parsing algorithm, consider the following syntax for the same language:

\[
\text{\texttt{pexp}} : \ ( \text{\texttt{pexp}} ) \mid \text{\texttt{astring}} \\
\text{\texttt{astring}} : \ \circ \mid \text{\texttt{astring}} \ast
\]

While the language is unchanged, so the algorithm above still works, the lookahead tokens are not \textit{immediately} apparent upon looking at the productions. Some preprocessing must take place before one can decide on the choice of parser states and the appropriate lookahead tokens. Such algorithms indeed exist and result in what is known as an LR parser for the fragment above (actually, a simpler LALR parser may be built for this grammar \(^1\)). One can see that some grammar types may make the selection of the parsing algorithm more involved. Since SPLinT relies on \texttt{bison} for the generation of the parsing algorithm, one must ensure that the grammar is LALR(1) \(^2\).

\section*{6a Using the \texttt{bison} parser}

The process of using SPLinT for writing parsing macros in \LaTeX{} is treated in considerable detail later in this document. A shorter (albeit somewhat outdated but still applicable) version of this process is outlined in \cite{Sh}, included as part of SPLinT’s documentation. We begin, instead, by explaining how one such parser can be used to pretty print a \texttt{bison} grammar. Following the convention mentioned above and putting all non-C code inside \texttt{CWEAVE}’s verbatim blocks, consider the following (meaningless) code fragment \(^3\). The fragment contains a mixture of C and \texttt{bison} code, the former appears outside of the verbatim blocks.

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

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

\begin{verbatim}
⟨ A silly example 6a ⟩ = 

\texttt{non_terminal:}

\texttt{term\_1 \ term\_2

\texttt{term\_3 \ other\_term

\texttt{still more terms

See also sections 7a, 7c, and 8c.

This code is used in section 8f.

\end{verbatim}

\footnote{Both of these algorithms will use the parser stack more efficiently, effectively resolving the ‘call stack depth’ issue mentioned earlier.}

\footnote{The newest versions of \texttt{bison} are capable of processing a \textit{much} wider set of grammars, although SPLinT can only handle the \texttt{bison} output for LALR(1) parsers.}

\footnote{The software included in the package contains a number of preprocessing scripts that reduce the necessity of using the verbatim blocks for every line of the \texttt{bison} code so the snippet above can instead be presented without the distraction of @=\ldots@>, looking more like the ‘native’ \texttt{bison} input}
... if the syntax is correct. In case it is a bit off (note the missing colon after whoops), 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 by CWEAVE. Only the verbatim portion is left unprocessed.

\begin{verbatim}
⟨ A silly example 6a ⟩ + =
whoops
    term.1 term.2 { a ⇐ b; }
   | term.3 other_term { T ⇐ \gamma_1; }
   | still more terms { f(\gamma_1); }
;
\end{verbatim}

The \TeX{} header that makes such output possible is quite plain. In the case of this document it begins as

\begin{verbatim}
\input limbo.sty
\input frontmatter.sty
\def\optimization{5}
\input yy.sty
[more code ...]
\end{verbatim}

The first two lines are presented here merely for completeness: there is no parsing-relevant code in them. The third line (\texttt{\def\optimization{5}}) may be ignored for now (we discuss some ways the parser code may be sped up later. 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

\begin{verbatim}
\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 ...]
\end{verbatim}

We will have a chance to discuss all the \texttt{\...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 \texttt{bo.tok} consists of a number of lines like the ones below:

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

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

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 \texttt{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

\begin{verbatim}
\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 ...]
\end{verbatim}
USING THE BISON PARSER

8a

the line titled \( \langle \text{A production } 8a \rangle \) 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 \texttt{bison} input, it misplaced the \( \langle \text{Rest of line } 8b \rangle \), 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 \texttt{CWEAVE}, even that which contains no interesting \texttt{C} code; hint: it has something to do with money ($), also known as math and the way \texttt{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.

\[
\langle \text{A production } 8a \rangle = \langle \text{Rest of line } 8b \rangle a \leftarrow f(x);
\]

See also section 8d.

This code is cited in sections 3b and 8a.

This code is used in sections 7c and 8c.

8b

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

\[
\langle \text{Rest of line } 8b \rangle = \langle \text{Rest of line } 8b \rangle \text{but not all of it}
\]

See also section 8e.

This code is cited in section 8a.

This code is used in sections 7c and 8c.

8c

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 \texttt{bison} grammars allows one to add hidden context to the code fragments above. The context is added inside \texttt{vrb} sections using \texttt{CWEB}'s \@t...@> facility. The \texttt{CTANGLE} output is not affected by this while the code above can now be typeset as:

\[
\langle \text{A silly example } 6a \rangle + = \langle \text{next_term} \rangle
\]

\[
\langle \text{A production } 8a \rangle
\]

\[
\langle \text{Rest of line } 8b \rangle a \leftarrow f(x);
\]

See also section 3b.

This code is cited in sections 7c and 8c.

8d

\[
\langle \text{A production } 8a \rangle + = \langle \text{next_term} \rangle
\]

\[
\langle \text{Rest of line } 8b \rangle a \leftarrow f(x);
\]

\[
\langle \text{A production } 8a \rangle
\]

\[
b \leftarrow g(y);
\]

8e

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.

\[
\langle \text{Rest of line } 8b \rangle + = \langle \text{Rest of line } 8b \rangle \text{but not all of it}
\]

8f

What makes all of this even more confusing is that \texttt{CTANGLE} will have no trouble outputting this as a(n almost, due to the intentionally bad \texttt{whoops} production above) valid \texttt{bison} file (as can be checked by looking into \texttt{sill.y}). The author happens to think that one should not fragment the software into pieces that are too small: \texttt{bison} is not \texttt{C} so it makes sense to write \texttt{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.
\[ \text{sill.y} \] =
\[ \text{A silly example} \]

9a On debugging

This concludes a short introduction to the bison grammar pretty printing using this macro collection. It would be incomplete, however, without a short reference to debugging \(^{1}\)). There is a fair amount of debugging 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 bison parser gives up too often or even crashes (the latter is most certainly a bug in the SPLiNT version of the bison 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 \texttt{\textbackslash trace...true} to activate the appropriate trace macros. This may produce a lot of output, even for small fragments, so turn it on for only 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, \texttt{\textbackslash 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. Most of these conditionals are documented in \texttt{yydebug.sty}, which provides a number of handy shortcuts for a few commonly encountered situations, as well.

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 all of the above fails, you are on your own: drop me a line if you figure out how to fix any bugs you find.

\(^{1}\) Here we are talking about debugging the output produced by \texttt{CWAVE} 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.
This short chapter is an informal listing of a few loose definitions of the concepts used repeatedly in this documentation. Most of this terminology is rather standard. Formal precision is not the goal here, instead, intuitive explanations are substituted whenever possible.

- **bison** (as well as **flex** parser(s)): 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 or flex 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 \texttt{\text{	extbackslash{}ytable}}) 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 \(^1\).

- **parser stack**: a collection of parsers, usually derived from a common set of productions, and sharing a common lexer. As the name suggests, the parsers in the collection are tried in order until the input is parsed successfully or every parser has been tried. This terminology may be the source of some confusion, since each parsing algorithm used by bison maintains several stacks. We will always refer to them by naming a specific task the stack is used for (such as the value stack or the state stack, etc.).

- **pretty printing** or **program visualization**: The terms above are used interchangeably in this manual to mean typesetting the program code in a way that emphasizes its meaning as seen by the author of the program \(^2\). It is usually assumed that such meaning is extracted by the software (a specially designed parser) and translated into a suitable visual representation.

1) To be precise, the **namespaces** in this manual, would more appropriately be referred to as **named scopes**. The **tag namespace** in C is an example of a (built-in) language namespace where the grammatical rôle of the identifier determines its association with the appropriate set.

2) Or the person typesetting the code.
**symbolic switch**: a macro (or an associative array of macros) that let the TeX parser generated by the package associate *symbolic term names* (called *named references* in the official *bison* documentation) 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**: (also referred to as a *named reference* in the *bison* manual): 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: $[name]$ (or even $name$ when the *name* has a tame enough syntax). 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, a *token* is 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.
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 could 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, deserves some attention, as well. There are a number of excellent programs 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 \textit{syntax-directed translation} (which requires an appropriate abstraction to represent bison’s actions inside \TeX{}). We shall begin by discussing the parsing process itself.
14a 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, the macro \texttt{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 \texttt{relax}’s will produce no output, even though \texttt{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 the memory. At the core of its generic programming environment is an array of \texttt{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 \texttt{count} array a good way to store tables in \TeX? Probably not. The first factor is the size of this array: only 256 \texttt{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 (\texttt{catcode, uccode, dimen, sfcode} and others can be used for this purpose as long as one takes care to restore the ‘sane’ values before the 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 utilization 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 \texttt{count} registers are used by the 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 (\texttt{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 save 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: \texttt{token} registers, macros holding whole arrays, and associative arrays accessed through \texttt{csname...\endsname}. In the first two cases if care is taken to store such arrays in an appropriate form one can use \TeX’s \texttt{ifcase} primitive to access individual elements. The trade-off is the speed of such access: it is \texttt{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 \texttt{ifcase} is used after expanding a list macro or the contents of a \texttt{token n} register in an unoptimized parser. An \textit{optimized} parser uses associative arrays.

The array discussion above is just as applicable to \textit{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 \texttt{count} register to keep track of the top of the stack in the corresponding associative array.

Let us now switch our attention to the code that implements the parser and scanner \textit{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 \footnote{Formally defined as ‘...special compile-time functions that consume and produce syntax objects’ in \cite{DHB}.} produce \textit{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 \textit{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 nearly straightforward. The macro \TeXniques employed by the sample code below are further discussed elsewhere in this manual.

\begin{verbatim}
label A: ...
  [more code ...]
  if(condition)
    goto C;
  [more code ...]
label B: ...
  [more code ...]
  goto A;
  [more code ...]
label C: ...
  [more code ...]
\end{verbatim}

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 the 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.

15a \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 performed 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 reviewing the concept of a token.

As commonly defined, a token is simply an element of a set (see the section on terminology earlier in this manual). 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 rôle of characters that combine to become the former. Since bison's internal representation for its tokens is non-negative integers, this is what the scanner must produce.

\TeX's tokens are a good deal more sophisticated: they can be either pairs \((c_{\text{ch}}, c_{\text{cat}})\), where \(c_{\text{ch}}\) is the character code and \(c_{\text{cat}}\) 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 \texttt{CWEAVE} is done with your input. This work becomes even more 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 \texttt{CWEAVE} into the page.

\splint comes with a customizable input routine of moderate complexity (\texttt{\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 \texttt{\_\_}, as well as \texttt{\_\_}), groups (avoid using \texttt{\_\_\_\group} and \texttt{\_\_\_\_\group} in your input but 'real', \{\ldots\} groups are fine), active characters, normal

\(^1\) 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.
characters (of all character categories that can appear in \TeX input, including $\$, $\$, $\$, $\$ 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 12 (to be even more precise (and to use nested parentheses), ‘normalized’ characters have the same category code as that of ‘, ’ at the point where \yyinput.sty is read)).

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 \verb\{...\}, the sequence \& 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 \TeXxologies: \futurelet
ext
examinenext or its equivalent \afterassignment
examinenext \let
ext=a. 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 \yysof sequence, 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 \yysof 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). Another approach to avoid the use of stop tokens is to store the whole input as a parameter for the appropriate macro. This scheme is remarkably powerful and can produce expandable versions of very complicated routines, although the amount of effort required to write such macros grows at a frightening rate. As the text inside \verb... is nearly always well structured, the care that \yyinput takes in processing such character lists is an overkill. In a more ‘hostile’ environment (such as the one encountered by the now obsolete \TeX macros), however, 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 \verb sequence. 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 \texttt{\atcode}{\string}{\char} earlier) but not whether it is a ‘real’ brace (i.e. a character \texttt{\char}) or an implicit one (a \texttt{\begin{group}}, 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 \begin{group} as the parameter, there is no guarantee that the actual input was not \begin{group}. One way to handle this is by applying \texttt{\string} before consuming the next token. If prior to expanding \texttt{\string} care has been taken to set the \texttt{\escapechar} appropriately (remember, we know the character code of the next token in advance), as soon as one sees a character with \texttt{\escapechar} 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 \texttt{\atcode} mechanism) that has the same character code as the brace token that it has been \texttt{\let} to! Even setting this disturbing possibility aside, the \texttt{\string} mechanism (or, its cousin, \texttt{\meaning}) is far from perfect: both produce a sequence of category 12 and 10 tokens that are mixed into the original input. 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 \texttt{\string} makes a sequence into characters, \texttt{\cname}...\\texttt{\endcname} pair will make any sequence of characters into a control sequence so determining the end the character sequence produced by \texttt{\string} may prove impossible. Huh...

What we need is a backup mechanism: keeping a copy of the token sequence ahead, one can use \texttt{\string} to see whether the next token 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 the brace has to be reinserted 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 \texttt{\iftrue\else\fi} trick (and a generous sprinkling of \texttt{\expandafter}). Of course, some bookkeeping is needed to keep track of how deep inside the braces groups we are. For an implicit brace, more work is needed: read all the characters that \texttt{\string} produced (and 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 \texttt{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 \texttt{yyinput} has two ways of dealing with braces: strip them or view the whole group as a token. Pick one or write a different \texttt{yyinput}. Spaces, implicit or explicit, are reported as a specially selected character code and consumed with a likeness of \texttt{afterassignment\moveon\let\next=␣}. This behavior can be adjusted if needed. Now that a steady stream of character codes is arriving at \yylex after \yyyreturn the job of converting it into numerical tokens is performed by the \texttt{scanner} (or \texttt{lexer}, or \texttt{tokenizer}, or even \texttt{tokener}), discussed in the next section.

17a 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 recalls 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 \footnote{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.} 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 \TeX techniques in implementing \texttt{yyparse} are fully applicable to \texttt{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 \texttt{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: `\yyval`, a token register containing the value of the returned token and `\yych`, 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 programmer: the character sequence representing the token just matched (`\yytext`). This is not the whole story though as 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 12 (see the discussion of `\yybytpure` above). In cases when the tokens in `\yytext` are not `(cchk, ccat)` 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 associative array.

The other two sequences are special ‘stream pointers’ that provide access to the extended scanner mechanism in order to implement the passing of the ‘formatting hints’ to the parser, as well as incorporate *CWEB* formatted code into the input, without introducing any changes to the original grammar. As the mechanism itself and the motivation behind it are somewhat subtle, let us 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 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 programmer providing enough context for the parser to do its job. For short examples like the one above, this may be 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, such context hints are the right way to handle exceptions, since using them forces the writer to emphasize the logical structure of the formal text.
If the pretty printing style is changed later on, 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 CWB’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 as the one employed by the General LR parsers).

This somewhat lengthy discussion brings us to the question directly related to the tools described in this manual: 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 the ‘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 altered.

A much more satisfying approach involves inserting the hints at the input stage and passing this information to the scanner and the parser as part of the token ‘values’. The hints themselves can masquerade as characters ignored by the scanner (white space 1), 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 now 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 2 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 3 hidden streams (called the \yystash stream and the \yyformat 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

---

1) Or even the ‘intercharacter space’, to make the hints truly invisible to the scanner. 2) Even if one were to somehow mitigate the effects of the lookahead in the parser, the scanner would still have to read the characters of the current token up to (and, in some cases, beyond) the (token’s) boundary which, in most cases, is the whitespace, possibly hiding the next hint. 3) There would be no difficulty in splitting either of these streams into multiple ‘substreams’ by modifying the stream extraction macros accordingly.
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 (\texttt{yyreadfifo} and its cousins) are provided for the convenience of the parser designer.

The \texttt{yystash} stream collects all the typesetting commands inserted by \texttt{CWEB} to be possibly used in displaying the action code in \texttt{bison} productions, for example. Because of this, it may appear in somewhat unexpected places, introducing spaces where the programmer did not necessarily intend (such as at the beginning of the line, etc.). To mitigate this problem, the \texttt{yystash} stream macros are implemented to be entirely invisible to the lexer. Making them produce spaces is also possible, and some examples are provided in \texttt{symbols.sty}. The interested reader can consult the input routine macros in \texttt{yyinput.sty} 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 \texttt{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 \footnote{One may think of the parser output as an executable abstract syntax tree (AST).}. 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).

\subsection*{20a Inside semantic actions: switch statements and 'functions' in \TeX}

So far we have looked at the lexer for your input, and a grammar ready to be put into action (we will talk about actions a few moments later). It is time to discuss how the tables produced by \texttt{bison} get converted into \TeX macros that drive the parser in \TeX.

The tables that drive the \texttt{bison} input parsers are collected in \texttt{(b,d,f,g,n)yytab.tex} and \texttt{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 \texttt{bison} (how this is done is detailed elsewhere, see \cite{Ah} for a good reference). It would suffice to mention here that there are three types of tables in this file: \footnote{One may think of the parser output as an executable abstract syntax tree (AST).} numerical tables such as \texttt{yytable} and \texttt{yycheck} (both are either \TeX's token registers in an unoptimized parser or associate arrays in an optimized version of such as discussed below), \footnote{One may think of the parser output as an executable abstract syntax tree (AST).} a string array \texttt{yytname}, and \footnote{One may think of the parser output as an executable abstract syntax tree (AST).} 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 \texttt{yymisc.sty}. The generated table files also contain constant and token declarations used by the parser.

The description of the process used to output \texttt{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

\begin{verbatim}
\switchon\next\in\currentswitch
\end{verbatim}

is responsible for calling an appropriate action in the current switch, as is easy to infer. A \texttt{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 \texttt{\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, \texttt{\setspecialcharsfrom\switchname} can be used to put the \TeX tokens in a form suitable for the consumption by \texttt{\switchon}'s. The most important step it performs is it \texttt{turns every token in the list into a character with the same character code and category 12}. Thus \texttt{X} becomes \texttt{\{12}. There are other ways of inserting tokens into a state: enclosing a token or a string of tokens in \texttt{\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 label 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 `a_1` unless you played with the 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\(^1\)). 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 a single ‘(` 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 differentiation between characters of different categories.

Finally, the construct `\statecomment` 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 `\tracedfattrue`.

The functionality of the `\switchon` (as well as the `\switchonwithtype`, which is capable of some rudimentary type checking) macros (for ‘historical’ reasons, one can also use `\action` as a synonym for the latter) has been implemented in a number of other macro packages (see \cite{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\{\ttoks0 \}{}` which will preserve all the changes to `\ttoks0` and have no other side effects (if, for example, in typical \TeX vernacular, `\textbf{next}` is used to implement tail recursion inside the group, after the `\tokreturn`, `\textbf{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 rôle 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`\{\text quant material\} which will expand (as in `\edef`) the ‘\TeX material’ inside the braces. Thus, a simple way to concatenate the values of the first two production terms is `\yy0`\{\text the\} `\yy(1)` `\text the\}` `\yy(2)`. The included `bison` parser can also be used to provide support for ‘symbolic names’, analogous to `bison`’s `\$[name]` but a bit more effort is required on the user’s part to initialize such support. Using symbolic names can make the parser more readable and maintainable, however.

There is also a `\bb n` macro, that has no analogue in the ‘real’ `bison` parsers, and provides access to the term values in the ‘natural order’ (e.g. `\bb1` is the last term read). Its intended use is with the ‘inline’ rules (see the main parser for such examples). As of version 3.0 `bison` no longer outputs `\yyrhs` and `\yyprhs`, which

\(^1\) One way to mitigate this is by putting such named states at the end of the switch, after the actions labelled by the standalone characters.
makes it impossible to produce the \textit{yyrthree} array necessary for processing such rules in the ‘left to right’ order. One might also note that the new notation is better suited for the inline rules since the value that is pushed on the stack is that of \texttt{bb0}, i.e. the term implicitly inserted by \texttt{bison}. Be aware that there are no \texttt{\[} or \texttt{\(} versions of these macros, for obvious reasons. A less obvious feature of this macro is its ‘nonexpandable’ nature. This means they cannot be used inside \texttt{\edef}. Thus, the most common use pattern is \texttt{\[\toks m\]} with a subsequent expansion of \texttt{\toks m}. Making these macros expandable is certainly possible but does not seem crucial for the intended limited use pattern.

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 \texttt{\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.

22a ‘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 \texttt{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 \texttt{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.

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.

22b \TeX{} with a different slant or do you C an escape?

Some \TeX{} productions below probably look like alien script. The authors of \cite{Ex} cite a number of reasons to view pretty printing of \TeX{} in general as 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 \texttt{\def\texttt{\texnspace}\{other\}} at the beginning of this section can be uncommented and \texttt{\(Y \leftarrow \{Y_1\}\)} becomes \texttt{\noexpand/inmath\{yy0\{yy 1\{\}\}\}}. There is, however, more to this story. A look at the actual file will reveal that the line above was typed as

\begin{verbatim}
\TeX\_("/noexpand/inmath/\{yy0/\{yy1\}\}\);\n\end{verbatim}

The ‘escape character’ is leaning the other way! The lore of \TeX{} is uncompromising: ‘\textbackslash{}’ 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 \texttt{\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 to look for an alternative is straightforward: ‘\textbackslash{}’ is a special character in C, as well (also an ‘escape’, in fact).
The line \texttt{TeX( "..." )}; is a \textit{macro-call} but ... in C. This function simply prints out (almost 'as-is') the line in parenthesis. An attempt at \texttt{TeX( "\noexpand" )}; would result in

\begin{verbatim}
01 oexpand 
02 oexpand 
\end{verbatim}

Other escape combinations \footnote{Here is a full list of \textit{defined} escaped characters in C: \texttt{\backslash a}, \texttt{\backslash b}, \texttt{\backslash f}, \texttt{\backslash n}, \texttt{\backslash r}, \texttt{\backslash t}, \texttt{\backslash v}, \texttt{\backslash\{octal digit\}}, \texttt{\backslash\{c\}}, \texttt{\backslash\{r\}}, \texttt{\backslash\{x\}}, \texttt{\backslash\{u\}}, \texttt{\backslash\{U\}}. Note that the last three combinations must be followed by a specific string of characters to appear in the input without generating errors.} are even worse: most are simply undefined. If anyone feels trapped without an escape, however, the same line can be typed as

\texttt{TeX( "\noexpand\inmath{\yy0{\yy1{}}}" );}

Twice the escape!

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

The input language for bison loosely follows the BNF notation, with a few enhancements, such as the syntax for actions, to implement the syntax-directed translation, as well as various declarations for tokens, nonterminals, etc.

On the one hand, the language is relatively easy to handle, is nearly whitespace agnostic, on the other, a primitive parser is required for some basic setup even at a very early stage, so the design must be carefully thought out. This bootstrapping step is discussed in more details later on.

The path chosen here is by no means optimal. What it lacks in efficiency, though, it may amply gain in practicality, as we reuse the original grammar used by bison to produce the parser(s) for both pretty printing and bootstrapping. Some minor subtleties arising from this approach are explained in later sections.

As was described in the discussion of parser stacks above, to pretty print a variety of grammar fragments, one may employ a parser stack derived from the original grammar. The most natural and common unit of a bison grammar is a set of productions. It is thus natural to begin our discussion of the parsers in the bison stack with the parser responsible for processing individual rules.

One should note that the productions below are not concerned with the typesetting of the grammar. Instead this task is delegated to the macros in yyunion.sty and its companions. The first pass of the parser merely constructs an ‘executable abstract syntax tree’ (or EAST) which can serve very diverse purposes: from collecting token declarations in the bootstrapping pass to typesetting the grammar rules.

It would be impossible to completely avoid the question of the visual presentation of the bison input, however. It has already been pointed out that the syntax adopted by bison is nearly insensitive to whitespace. This makes writing bison grammars easier. On the other hand, presenting a grammar is best done using a variety of typographic devices that take advantage of the meaningful positioning of text on the page: skips, indents, etc. Therefore, the macros for bison pretty printing trade a number of bison syntax elements (such as |, ;, action braces, etc.) for the careful placement of each fragment of the input on the page.

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. The bison sections (separated by %% (shown (pretty printed) as ⟨%⟩ below)) appear between the successive dotted lines.

⟨bg.yy 25a⟩ =

.................................................................
⟨Grammar parser C preamble 40b⟩
.................................................................

1) One may argue that EAST is still merely a syntactic construct requiring a proper macro framework for its execution and should be called a ‘weak executable syntax tree’ or WEST. This acronym extravaganza is heading south so we shall stop here.
26a 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 always necessary (see the xexpression example, specifically, the file xexpression.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 genuine ‘chicken and egg’ problem). Hence the need for ‘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 the 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’).

26a 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 always necessary (see the xexpression example, specifically, the file xexpression.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 genuine ‘chicken and egg’ problem). Hence the need for ‘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 the 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’).
27a The prologue parser is responsible for parsing various grammar declarations as well as parser options.

27b The 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.

27c The first two options below are essential for the parser operation as each of them makes bison produce additional tables (arrays) used in the operation (or bootstrapping) of bison parsers. The start symbol can be set implicitly by listing the appropriate production first. Modern bison also allows specifying the kind of parsing algorithm to be used (provided the supplied grammar is in the appropriate class): LALR(n), LR(n), GLR, etc. The default is to use the LALR(1) algorithm (with the corresponding assumption about the grammar) which can also be set explicitly by putting
in with the rest of the options. Using other types of grammars will wreak havoc on the parsing algorithm hardcoded into SPLinT (see yyparse.sty) as well as on the production of \stashed and \format streams.

This code is used in sections 25a, 26a, 27a, and 27b.

28a Token declarations

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 the 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.

This code is used in sections 25a, 26a, 27a, and 27b.

28b We continue with the list of tokens below, following the layout of the original parser.

This code is used in section 28a.

28c Extra tokens for typesetting flex state declarations and options are declared in addition to the ones that a standard bison parser recognizes. This extension of the original grammar has become unnecessary with the addition of the flex input parser(s) but is left as part of the extended grammar for convenience and 'historical' reasons.

This code is used in sections 28a, 32b, and 36c.

28d Grammar productions

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 the ‘genuine’ *bison*’s \[ \ldots \] syntax.

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.

\[ \langle \text{Parser full productions 28d} \rangle = \text{input} : \text{prologue declarations} \% \text{grammar epilogue opt} \quad \pi_2(\Upsilon_3) \hookrightarrow \Omega \]

This code is used in section 27b.

29a 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.

\[ \langle \text{Fake start symbol for rules only grammar 29a} \rangle = \text{input} : \text{grammar epilogue opt} \quad \pi_2(\Upsilon_1) \hookrightarrow \Omega \]

This code is used in section 25a.

29b The bootstrap parser has a very narrow set of goals: it is concerned with (token) 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). The parser can also parse (nterm) declarations but the bootstrap lexer ignores the (nterm) token, since the *bison* grammar does not use one. It also extends the syntax of a *grammar_declaration* by allowing a declaration with or without a 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 output by the bootstrap lexer is limited, hence most of the other rules are ignored.

\[ \langle \text{Fake start symbol for bootstrap grammar 29b} \rangle = \text{grammar declarations} \quad \Omega = \Upsilon_1 \]

This code is used in section 26a.

29c 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.

\[ \langle \text{Carry on 29c} \rangle = \Upsilon \langle \text{val } \Upsilon_1 \rangle \]

This code is used in sections 29b, 30b, 31a, 31b, 31f, 33b, 33d, 33e, 33g, 34c, 34b, 36b, 39h, and 39i.

29d Next comes 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. Note that these roles are not part of the official *bison* input grammar and are added to make the typesetting of ‘file outlines’ (e.g. (bb.yy 26a) above) possible.

\[ \langle \text{Fake start symbol for prologue grammar 29d} \rangle = \text{prologue declarations epilogue opt} \quad \pi_2(\Upsilon_1) \hookrightarrow \Omega \quad \pi_2(\Upsilon_1) \hookrightarrow \Omega \quad \pi_2(\Upsilon_1) \hookrightarrow \Omega \]

This code is used in section 27a.

29e *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.

\[ \langle \text{Parser prologue productions 29e} \rangle = \]
declarations

prologue_declarations :
  • prologue_declarations prologue_declaration

See also sections 30b, 31a, and 39.
This code is used in sections 27a and 27b.

30a  ⟨ Attach a prologue declaration 30a ⟩ =
     ⟨ Attach a productions cluster 34e ⟩
This code is used in section 29e.

30b  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.

⟨ Parse prologue productions 29e ⟩ + =

prologue_declaration :
  grammar_declaration
  ⟨ Carry on 29c ⟩
  ⟨*⟩
  ⟨(define) variable value⟩
  ⟨(defines) angle bracket⟩
  ⟨(defines) string⟩
  ⟨(error-verbose) angle bracket⟩
  ⟨(expect) int⟩
  ⟨(expect-rr) int⟩
  ⟨(file-prefix) string⟩
  ⟨(glr-parser) angle bracket⟩
  ⟨(initial-action) { ... }⟩
  ⟨(language) string⟩
  ⟨(name-prefix) string⟩
  ⟨(no-lines) angle bracket⟩
  ⟨(nonet. parser) angle bracket⟩
  ⟨(output) string⟩
  ⟨(param) angle bracket⟩
  ⟨(require) string⟩
  ⟨(skeleton) string⟩
  ⟨(token-table) angle bracket⟩
  ⟨(verbose) angle bracket⟩
  ⟨(yacc) angle bracket⟩
  ⟨token⟩

params :
  params { ... }
  { ... }

30c  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 30c ⟩ =
  ⟨ Carry on 31a ⟩

30d  ⟨ Prepare a generic one parametric option 30d ⟩ =
  ⟨ Carry on 31f ⟩
These rules handle 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. As was pointed out earlier the addition of the `flex` input parser to SPLinT made this extension of the original `bison` grammar obsolete but it was kept as part of the extended grammar for convenience and ‘historical’ reasons. The convenience results from simplifying the bootstrap procedure by using a single parser.

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

This code is used in sections 26a and 31a.

---

31c  Define `flex` option list

This code is used in section 31b.

31d  Define `flex` states

This code is used in section 31b.

31e  Add a `flex` option

This code is used in section 31b.

31f  Grammar declarations. These declarations can appear in both the prologue and the rules sections. Their treatment is very similar to the prologue-only options.

---

31a  These rules handle 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. As was pointed out earlier the addition of the `flex` input parser to SPLinT made this extension of the original `bison` grammar obsolete but it was kept as part of the extended grammar for convenience and ‘historical’ reasons. The convenience results from simplifying the bootstrap procedure by using a single parser.

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

---

31b  The syntax of `flex` options was extracted from `flex` documentation so it is not guaranteed to be correct.

This code is used in sections 26a and 31a.
code.props.type :
  (destructor) \( \tau \leftarrow \{ \text{destructor} \} \text{val} \tau \)
  (printer) \( \tau \leftarrow \{ \text{printer} \} \text{val} \tau \)

See also sections 32c, 32g, 33b, 33c, 33e, 39b, and 40a.
This code is used in sections 25a, 27a, and 27b.

32a \( \langle \text{Assign a code fragment to symbols} \rangle = \)
\[ \begin{align*}
\pi_1(\tau) & \rightarrow v_a & \triangleright \text{name of the property} < \\
\pi_2(\tau) & \rightarrow v_b & \triangleright \text{contents of the braced code} < \\
\pi_2(\tau) & \rightarrow v_c & \triangleright \text{braced code format pointer} < \\
\pi_3(\tau) & \rightarrow v_d & \triangleright \text{braced code stash pointer} < \\
\pi_3(\tau) & \rightarrow v_e & \triangleright \text{code format pointer} < \\
\pi_3(\tau) & \rightarrow v_f & \triangleright \text{code stash pointer} < \\
\tau & \leftarrow \langle \text{\codeprop} \{\text{val} v_a\}\{\text{val} v_b\}\{\text{val} \tau_3\}\{\text{val} v_c\}\{\text{val} v_e\}\{\text{val} v_f\}\rangle \\
\end{align*} \)

This code is used in section 31f.

32b \( \langle \text{Tokens and types for the grammar parser} \rangle + = \)
\( \langle \text{union} \rangle \)

32c \( \langle \text{Parser common productions} \rangle + = \)
\( \langle \text{union} \rangle \)

\textbf{grammar.declaration} : \( \langle \text{union} \rangle \) union.name \{ \ldots \}

\textbf{symbol.declaration} : \( \langle \text{type} \rangle < \text{tag} > \) symbols1

\textbf{precedence.declaration} :
\( \langle \text{precedence.declator} \rangle \text{tag_opt} \) symbols1
\( \text{prec_opt} \)

\textbf{precedence.declator} :
\( \langle \text{left} \rangle | \langle \text{right} \rangle | \langle \text{nonassoc} \rangle | \langle \text{precedence} \rangle \)
\text{tag_opt} : \( \circ | \langle \text{tag} \rangle \)

\( \tau \leftarrow \langle \text{\codeprop} \{\text{val} \tau_3\}\{\text{val} \tau_1\}\rangle \)

32d \( \langle \text{Prepare union definition} \rangle = \)
\( \tau \leftarrow \langle \text{\codeprop} \{\text{val} \tau_3\}\{\text{val} \tau_1\}\rangle \)

This code is used in section 32c.

32e \( \langle \text{Define symbol types} \rangle = \)
\( \tau \leftarrow \langle \text{\codeprop} \{\text{val} \tau_3\}\{\text{val} \tau_1\}\rangle \)

This code is used in section 32c.

32f \( \langle \text{Define symbol precedences} \rangle = \)
\( \pi_1(\tau) \rightarrow v_a & \triangleright \text{format pointer} < \\
\pi_2(\tau) \rightarrow v_b & \triangleright \text{stash pointer} < \\
\pi_2(\tau) \rightarrow v_c & \triangleright \text{kind of precedence} < \\
\tau & \leftarrow \langle \text{\codeprop} \{\text{val} v_a\}\{\text{val} \tau_2\}\{\text{val} \tau_3\}\{\text{val} v_e\}\{\text{val} v_b\}\rangle \)

This code is used in section 32c.

32g \( \langle \text{The bootstrap grammar forms the smallest subset of the full grammar.} \rangle = \)
\( \langle \text{Parser common productions} \rangle + = \)
\( \langle \text{Parser bootstrap productions} \rangle \)

32h \( \langle \text{Prepare a } \langle \text{tag} \rangle \rangle = \)
\( \tau \leftarrow \langle \text{\tag} \rangle \text{val} \tau_1 \)

This code is used in sections 32c, 33c, and 33f.
These are the two most important rules for the bootstrap parser. The reasons for the \langle(token)\rangle declarations to be collected during the bootstrap pass are outlined in the section on bootstrapping. The \langle(nterm)\rangle declarations are not strictly necessary for bootstrapping the parsers included in SPLinT but they are added for the cases when the bootstrap mode is used for purposes other than bootstrapping SPLinT.

\begin{align*}
\langle \text{Parser bootstrap productions} \rangle &= 33a \\
\text{symbol declaration :} &
\begin{align*}
\langle nterm \rangle \& \text{ symbol_defs1} \\
\langle token \rangle \& \text{ symbol_defs1}
\end{align*}
\end{align*}

See also sections 33f, 33g, 39a, and 39c.

This code is used in sections 26a and 32g.

\textit{Just like symbols_1 but accept int for the sake of POSIX. Perhaps the only point worth mentioning here is the inserted separator (\hspace{p_0}\{p_1\}, typeset as \␣p_1\). Like any other separator, it takes two parameters, the stream pointers \p_0 and \p_1. 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.}

\begin{align*}
\langle \text{Parser common productions} \rangle &= 31f \\
\text{symbols.prec :} &
\begin{align*}
symbol.prec & \\
\text{symbols.prec symbol.prec}
\end{align*}
\end{align*}

This code is cited in section 33c.

This code is used in sections 26a and 33c.

\text{One or more symbols to be (type)’d. The (List of symbols 33d) rules below are reused in the bootstrap parser and are put in a separate section for this reason.}

\begin{align*}
\langle \text{Parser common productions} \rangle &= 31f \\
\langle \text{List of symbols 33d} \rangle =
\end{align*}

\begin{align*}
symbol_1 : &
\begin{align*}
symbol & \\
symbol_1 symbol
\end{align*}
\end{align*}

This code is used in sections 26a and 33c.

\text{One token definition.}

\begin{align*}
\langle \text{Parser bootstrap productions} \rangle &= 33a \\
\text{symbol_def :} &
\begin{align*}
\langle \text{Token} \rangle & \\
id \mid id \text{ int} \mid id \text{ string_as_id} \mid id \text{ string_as_id}
\end{align*}
\end{align*}

\text{One or more symbol definitions.}

\begin{align*}
\langle \text{Parser bootstrap productions} \rangle &= 33a \\
\text{symbol_defs1 :} &
\begin{align*}
symbol_def & \\
symbol_defs1 symbol_def
\end{align*}
\end{align*}
34a ⟨Add a symbol definition 34a⟩ =
\[ \pi_2(T_2) \mapsto v_a \ \triangleright \ \text{the identifier} \]
\[ \pi_4(v_a) \mapsto v_b \ \triangleright \ \text{the format pointer} \]
\[ \pi_5(v_a) \mapsto v_c \ \triangleright \ \text{the stash pointer} \]
\[ \Upsilon \leftarrow (\text{val } T_1 \ {\text{val}} v_c \text{ val } T_2) \]
This code is used in section 33g.

34b The grammar section: between the two (\Upsilon)'s. Finally, the following few short sections define the syntax of bison’s rules.

⟨Parser grammar productions 34b⟩ =
\[ \text{grammar} \]
\[ \text{rules_or_grammar_declaration} \]
\[ \text{grammar_rules_or_grammar_declaration} \]
\[ \text{rules} \]
\[ \text{error} \]
\[ \text{rules} : \quad \text{id_colon named_ref_opt} \circ \text{rhses}_1 \]
\[ \text{rhses}_1 : \quad \text{rhs} \]
\[ \text{rhses}_1 \mid \text{rhs} \]
\[ \text{rhses}_1 ; \]

See also sections 34c, 36d, and 39d.

This code is used in sections 25a and 27b.

34c 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 34b⟩ + =
\[ \text{rules_or_grammar_declaration} : \quad \text{rules} \]
\[ \text{grammar_declaration ;} \]
\[ \text{error} ; \]
\[ \text{rules} : \quad \text{id_colon named_ref_opt} \circ \text{rhses}_1 \]
\[ \text{rhses}_1 : \quad \text{rhs} \]
\[ \text{rhses}_1 \mid \text{rhs} \]
\[ \text{rhses}_1 ; \]
\[ \text{Complete a production 35b} \]
\[ \text{Add a productions cluster 35a} \]
\[ \text{Carry on 29c} \]
\[ \text{error message} \{ \text{parsing error!} \} \]

34d The next few actions describe what happens when a left hand side is attached to a rule.

⟨Start with a production cluster 34d⟩ =
\[ \pi_1(T_1) \mapsto v_a \ \triangleright \ \text{the type of the last rule} \]
\[ \pi_2(T_1) \mapsto v_c \ \triangleright \ \text{accumulated rules} \]
\[ \pi_1(T_2) \mapsto v_b \ \triangleright \ \text{the type of the new rule} \]
\[ \Upsilon \leftarrow (\text{val } T_1 \ {\text{val}} v_c \text{ val } T_2) \]
This code is used in section 34b.

34e ⟨Attach a productions cluster 34e⟩ =
\[ \pi_3(T_1) \mapsto v_a \ \triangleright \ \text{type of the last rule} \]
\[ \pi_5(T_1) \mapsto v_c \ \triangleright \ \text{accumulated rules} \]
\[ \pi_1(T_2) \mapsto v_b \ \triangleright \ \text{type of the new rule} \]
\[ \text{let default \ positionswitchdefault} \]
\[ \text{switch (val } v_b \backslash \text{positionswitch} \ \triangleright \ \text{determine the position of the first token in the group} \]
\[ \text{def}_x \ \text{next \{ val } v_a \} \]
\[ \text{def}_x \ \text{default \{ val } v_b \} \ \triangleright \ \text{reuse } \text{default} \]
\[ \text{if}_x \ \text{next \default} \]
\[ \text{let default } \text{\ separatorswitchdefaulteq} \]
\[ \text{switch (val } v_a \backslash \text{separatorswitch ques} \]
\[ \text{else} \]
\[ v_a \leftarrow v_a +_s v_b \]
\[ \text{let default } \text{\ separatorswitchdefaultneq} \]
\[ \text{switch (val } v_a \backslash \text{separatorswitch neq} \]
\[ \text{fi} \]
\[ \Upsilon \leftarrow (\text{val } v_c \text{ val } \text{postoks } \text{val } v_d \text{ val } T_2 \{ \text{val } v_b \}) \]
This code is used in sections 30a and 34b.
35a \(\langle\text{Add a productions cluster}\rangle =\)
\[
\begin{align*}
\pi_2(Y_1) &\rightarrow v_a \triangleright \text{\textbackslash prodheader} \triangleright \\
\pi_3(v_a) &\rightarrow v_b \triangleright \text{\textbackslash idit} \triangleright \\
\pi_4(v_b) &\rightarrow v_c \triangleright \text{format stream pointer} \triangleright \\
\pi_5(v_b) &\rightarrow v_d \triangleright \text{stash stream pointer} \triangleright \\
\prod &\leftarrow (\text{\textbackslash oneproduction} \{\text{val } v_a \text{ val } v_b \}) \{\text{val } v_c \} \{\text{val } v_d \})
\end{align*}
\]

This code is used in section 34c.

35b \(\langle\text{Complete a production}\rangle =\)
\[
\begin{align*}
\pi_4(Y_1) &\rightarrow v_a \triangleright \text{format stream pointer} \triangleright \\
\pi_5(Y_1) &\rightarrow v_b \triangleright \text{stash stream pointer} \triangleright \\
\prod &\leftarrow (\text{\textbackslash pcluster} \{\text{\textbackslash prodheader} \{\text{val } Y_1 \} \{\text{val } Y_2 \} \{\text{val } v_a \} \{\text{val } v_b \}) \{\text{val } Y_4 \})
\end{align*}
\]

This code is used in section 34c.

35c 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 \texttt{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 implicit actions.
\[
\begin{align*}
\langle\text{Start the right hand side}\rangle =
\begin{align*}
\pi_i(Y_1) &\rightarrow v_a \triangleright \text{val } v_a \\
\pi_3(Y_1) &\rightarrow v_b \triangleright \text{the format pointer} \triangleright \\
\pi_4(Y_1) &\rightarrow v_c \triangleright \text{the stash pointer} \triangleright \\
\text{if} \ (\text{\texttt{rhs} = } \text{full})
\quad \Prod &\leftarrow (\text{\textbackslash rules} \{\text{val } Y_1 \} \{\text{val } v_b \} \{\text{val } v_c \})
\quad \text{else}
\quad \Prod &\leftarrow (\text{\textbackslash rules} \{\text{val } Y_1 \} \{\text{val } v_b \} \{\text{val } v_c \}) \{\text{val } Y_4 \}
\quad \text{def } \text{next} \{\text{val } v_a \}
\quad \text{if} \ \text{next} \varnothing
\quad \quad v_a &\leftarrow (\ldots )
\quad \fi
\quad \Prod &\leftarrow (\text{\textbackslash rules} \{\text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \} \{0\} \{0\} \{\text{\textbackslash actbraces } \} \{0\} \{0\} \{\text{\textbackslash bdend } \} \{\text{\textbackslash actbraces } \} \{0\} \{0\}) \{\text{\textbackslash actbraces } \} \{0\} \{0\} \{\text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \} \{\text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \text{\textbackslash rrhssep } \})
\quad \text{fi}
\end{align*}
\]

This code is used in section 34c.

35d Using standard notation, here is what the middle action does.
\[
\begin{align*}
\langle\text{Old ‘Insert local formatting’}\rangle =
\begin{align*}
\pi_i(Y_1) &\rightarrow \{Y_0\}
\Prod &\leftarrow (\text{\textbackslash midf } \text{val } Y_2)
\end{align*}
\end{align*}
\]

35e However, if the length of the rule preceding the inline action is not known a different way of accessing the stack is necessary.
\[
\begin{align*}
\langle\text{Insert local formatting}\rangle =
\begin{align*}
\pi_i(v_a) &\rightarrow [v_a]
\pi_i(Y_0) &\rightarrow \{Y_0\}
\Prod &\leftarrow (\text{\textbackslash midf } \text{val } v_b)
\end{align*}
\end{align*}
\]

This code is used in section 34c.

35f \(\langle\text{Old ‘Add a right hand side to a production}\rangle =\)
\[
\begin{align*}
\pi_n(Y_3) &\rightarrow v_a \triangleright \text{val } v_a
\text{if} \ (\text{\texttt{rhs} = } \text{full})
\quad \Prod &\leftarrow (\text{\textbackslash rules} \{\text{val } Y_3 \text{\textbackslash rrhssep } \text{val } Y_2 \text{val } Y_4 \}) \{\text{val } Y_2 \}
\quad \text{else}
\end{align*}
\]

This code is used in section 34c.
No pointers are provided for an *implicit* action. Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use an array to store AST nodes (*astarray*). While providing a noticeable speed up, this technique significantly complicates the debugging of the grammar. In particular, inspecting a parsed table supplies very little information if the AST nodes are not expanded. The macros in *yyunion.sty* provide a special debugging namespace where the expansion of the parser produced control sequences may be modified to safely expand the generated table.

36a No pointers are provided for an *implicit* action. Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use an array to store AST nodes (*astarray*). While providing a noticeable speed up, this technique significantly complicates the debugging of the grammar. In particular, inspecting a parsed table supplies very little information if the AST nodes are not expanded. The macros in *yyunion.sty* provide a special debugging namespace where the expansion of the parser produced control sequences may be modified to safely expand the generated table.

36b No pointers are provided for an *implicit* action. Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use an array to store AST nodes (*astarray*). While providing a noticeable speed up, this technique significantly complicates the debugging of the grammar. In particular, inspecting a parsed table supplies very little information if the AST nodes are not expanded. The macros in *yyunion.sty* provide a special debugging namespace where the expansion of the parser produced control sequences may be modified to safely expand the generated table.

36c No pointers are provided for an *implicit* action. Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use an array to store AST nodes (*astarray*). While providing a noticeable speed up, this technique significantly complicates the debugging of the grammar. In particular, inspecting a parsed table supplies very little information if the AST nodes are not expanded. The macros in *yyunion.sty* provide a special debugging namespace where the expansion of the parser produced control sequences may be modified to safely expand the generated table.

36d No pointers are provided for an *implicit* action. Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use an array to store AST nodes (*astarray*). While providing a noticeable speed up, this technique significantly complicates the debugging of the grammar. In particular, inspecting a parsed table supplies very little information if the AST nodes are not expanded. The macros in *yyunion.sty* provide a special debugging namespace where the expansion of the parser produced control sequences may be modified to safely expand the generated table.

}
SPLINT GRAMMAR PRODUCTIONS

37a

\[ \langle \text{Make an empty right hand side 37a} \rangle = \]
\[ \Upsilon \leftarrow (\text{\textasciitilde nx rhs } \{ \} \{ \text{\textasciitilde nx rhs = not full} \}) \]

This code is used in section 36d.

37b

\[ \langle \text{Add a term to the right hand side 37b} \rangle = \]
\[ \pi_1(\Upsilon_1) \mapsto v_a \]
\[ \pi_2(\Upsilon_2) \mapsto v_b \]
\[ \text{def}_3 \text{next } \{ \text{val } v_b \} \]
\[ \text{if}_b \text{next } \emptyset \]
\[ \begin{align*}
&\pi_4(\Upsilon_2) \mapsto v_c \\
&\pi_5(\Upsilon_2) \mapsto v_d \\
&v_b \leftarrow v_b + \text{sx } [\{ \text{val } v_c \} \{ \text{val } v_d \}] \\
&\text{fi}
\end{align*} \]
\[ \Upsilon \leftarrow (\text{\textasciitilde nx rhs } \{ \text{val } v_a \text{val } v_b \text{\textasciitilde termname } \{ \text{val } \Upsilon_2 \} \{ \text{val } \Upsilon_3 \} \{ \text{\textasciitilde nx rhs } = \text{not full} \}) \]

This code is used in section 36d.

37c

\[ \langle \text{Add an action to the right hand side 37c} \rangle = \]
\[ \pi_1(\Upsilon_1) \mapsto v_a \]
\[ \pi_2(\Upsilon_2) \mapsto v_b \]
\[ \text{if } ( \text{rhs = full} ) \quad \triangleright \text{ the first half ends with an action } \}
\[ \quad v_a \leftarrow v_a + \text{sx } [\text{\textasciitilde arhssep } (0) \{0\} \{\text{\textasciitilde nx arhssep}\} \{\text{\textasciitilde nx rhs } = \text{not full} \}] \quad \triangleright \text{no pointers to streams } \}
\[ \quad \text{fi}
\[ \text{def}_3 \text{next } \{ \text{val } v_a \} \]
\[ \text{if}_c \text{next } \emptyset \]
\[ v_a \leftarrow \{ \ldots \} \]
\[ \text{fi}
\[ \pi_1(\Upsilon_2) \mapsto v_b \quad \triangleright \text{the contents of the braced code } \}
\[ \pi_2(\Upsilon_2) \mapsto v_c \quad \triangleright \text{the format stream pointer } \]
\[ \pi_3(\Upsilon_2) \mapsto v_d \quad \triangleright \text{the stash stream pointer } \]
\[ \Upsilon \leftarrow (\text{\textasciitilde nx rhs } \{ \text{val } v_a \text{\textasciitilde arhssep } \{ \text{val } v_b \} \{ \text{val } v_c \} \{ \text{val } v_d \} \{ \text{\textasciitilde nx rhs } = \text{not full} \}) \}
\quad \text{\textasciitilde actbraces } \{ \text{val } v_b \} \{ \text{val } \Upsilon_2 \} \{ \text{val } \Upsilon_3 \} \{ \text{\textasciitilde nx arhssep } \{ \text{\textasciitilde nx rhs } = \text{full} \}) \]

This code is used in section 36d.

37d

\[ \langle \text{Add a predicate to the right hand side 37d} \rangle = \]
\[ \pi_1(\Upsilon_1) \mapsto v_a \]
\[ \pi_2(\Upsilon_2) \mapsto v_b \]
\[ \text{if } ( \text{rhs = full} ) \quad \triangleright \text{the first half ends with an action } \}
\[ \quad v_a \leftarrow v_a + \text{sx } [\text{\textasciitilde arhssep } (0) \{0\} \{\text{\textasciitilde nx arhssep}\} \{\text{\textasciitilde nx rhs } = \text{not full} \}] \quad \triangleright \text{no pointers to streams } \}
\[ \quad \text{fi}
\[ \text{def}_3 \text{next } \{ \text{val } v_a \} \]
\[ \text{if}_d \text{next } \emptyset \]
\[ v_a \leftarrow \{ \ldots \} \]
\[ \text{fi}
\[ \pi_1(\Upsilon_2) \mapsto v_b \quad \triangleright \text{the contents of the braced code } \}
\[ \pi_2(\Upsilon_2) \mapsto v_c \quad \triangleright \text{the format stream pointer } \]
\[ \pi_3(\Upsilon_2) \mapsto v_d \quad \triangleright \text{the stash stream pointer } \]
\[ \Upsilon \leftarrow (\text{\textasciitilde nx rhs } \{ \text{val } v_a \text{\textasciitilde arhssep } \{ \text{val } v_b \} \{ \text{val } v_c \} \{ \text{val } v_d \} \{ \text{\textasciitilde nx rhs } = \text{full} \}) \}
\quad \text{\textasciitilde bpredicate } \{ \text{val } v_b \} \{ \text{\textasciitilde nx arhssep } \{ \text{\textasciitilde nx rhs } = \text{full} \}) \]

This code is used in section 36d.

37e

\[ \langle \text{Add } \emptyset \text{ to the right hand side 37e} \rangle = \]
\[ \pi_1(\Upsilon_1) \mapsto v_a \]
\[ \pi_2(\Upsilon_2) \mapsto v_b \]
def x next { val v_b }
def x next
else
    π₄(T₂) ↦ v_c
    π₅(T₂) ↦ v_d
    v_b ← v_b + nx { { val v_c } { val v_d } }
fi
Υ ← "nx rhs { val v_a val v_b nx... } { nx rhs = not full }")

This code is used in section 36d.

38a ⟨ Add a precedence directive to the right hand side 38a ⟩ =
π₁(T₁) ↦ v_a
π₂(T₁) ↦ v_b
π₃(T₁) ↦ v_c val v_c
if ( rhs = full )
    Y ← "nx sprecop { val T₃ } val T₂ } ▷ reuse \yyval <
    \supplydirective v_a Y ▷ the directive is ‘absorbed’ by the action ◄
    Y ← "nx rhs { val v_a T_val v_b } { nx rhs = not full }")
else
    Y ← "nx rhs { val v_a nx sprecop { val T₃ } val T₂ } { val v_b } { nx rhs = not full }")
fi

This code is used in section 36d.

38b ⟨ Add a ⟨ dprec ⟩ directive to the right hand side 38b ⟩ =
π₁(T₁) ↦ v_a
π₂(T₁) ↦ v_b
π₃(T₁) ↦ v_c val v_c
if ( rhs = full )
    Y ← "nx dprecop { val T₃ } val T₂ } ▷ reuse \yyval <
    \supplydirective v_a Y ▷ the directive is ‘absorbed’ by the action ◄
    Y ← "nx rhs { val v_a } { val v_b } { nx rhs = not full }")
else
    Y ← "nx rhs { val v_a nx dprecop { val T₃ } val T₂ } { val v_b } { nx rhs = not full }")
fi

This code is used in section 36d.

38c ⟨ Add a ⟨ merge ⟩ directive to the right hand side 38c ⟩ =
π₁(T₁) ↦ v_a
π₂(T₁) ↦ v_b
π₃(T₁) ↦ v_c val v_c
if ( rhs = full )
    Y ← "nx mergeop { nx \tagit val T₃ } val T₂ } ▷ reuse \yyval <
    \supplydirective v_a Y ▷ the directive is ‘absorbed’ by the action ◄
    Y ← "nx rhs { val v_a } { val v_b } { nx rhs = not full }")
else
    Y ← "nx rhs { val v_a nx mergeop { nx \tagit val T₃ } val T₂ } { val v_b } { nx rhs = not full }")
fi

This code is used in section 36d.

38d ⟨ Create an empty named reference 38d ⟩ =
Y ← "

This code is used in section 36d.

38e ⟨ Create a named reference 38e ⟩ =
⟨ Turn an identifier into a term 39f ⟩

This code is used in section 36d.
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.

\[ \langle \text{Parser bootstrap productions} \rangle + = \]
\[ id : \langle \text{identifier} \rangle \]
\[ \text{char} \]

\[ \langle \text{Parser common productions} \rangle + = \]
\[ \text{id : } \langle \text{identifier} \rangle \]
\[ \text{string as id} \]

This code is used in sections 26a and 39b.

\[ \langle \text{Definition of symbol} \rangle = \]
\[ symbol : \]
\[ id \]
\[ string as id \]

This code is used in sections 26a and 39b.

\[ \langle \text{Turn an identifier into a term} \rangle \]
\[ \langle \text{Turn a character into a term} \rangle \]

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).

\[ \langle \text{Turn an identifier into a symbol} \rangle = \]
\[ \langle \text{Prepare the left hand side} \rangle \]

Variable and value. The \text{string} form of variable is deprecated and is not M4-friendly. For example, M4 fails for \text{\%define } "[" "value"].

\[ \langle \text{Prepare a string for use} \rangle = \]
\[ \langle \text{Prepare a string for use} \rangle \]
\[ \langle \text{Prepare a string for use} \rangle \]

This code is used in sections 39e and 39l.
value: o | <identifier> | <string> | {...}

\( Y \leftarrow \{^{\text{\textbackslash braced value}} \text{val \textbackslash T_1}\} \)

40a \{Parser common productions \}\(+ =
epilogue_{\text{opt}}: o \mid \%\) epilogue

40b 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 \}\(40b\) =
This code is used in sections 25a, 26a, 27a, and 27b.

40c C postamble for the grammar parser. It is tricky to insert function definitions that use \texttt{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 \}\(40c\) =
\#define YYPRINT(file, type, value) yyprint(file, type, value)
static void yyprint(FILE *file, int type, YYSTYPE value)
{}
This code is used in sections 25a, 26b, 27b, and 40d.

40d \{Bootstrap parser C postamble \}\(40d\) =
\{Grammar parser C postamble \}\(40c\)
\{Bootstrap token output \}\(40e\)
This code is used in section 26a.

40e \{Bootstrap token output \}\(40e\) =
void bootstrap_tokens(char *bootstrap_token_format){
\#define _register_token_d(name) fprintf(tables_out, bootstrap_token_format, #name, name, #name);
\{Bootstrap token list \}\(40f\)\}
\#undef _register_token_d
}
This code is used in section 40d.

40f 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 \}\(40f\) =
_register_token_d(ID)
_register_token_d(PERCENT_TOKEN)
_register_token_d(STRING)
This code is used in section 40e.

40g Union of types. This section of the \texttt{bison} input lists the types that may appear on the value stack. Since \TeX does not provide any mechanism for type checking (nor is it clear how to translate a C \texttt{union} into any data structure usable in \TeX), this section is left empty.
\{Union of grammar parser types \}\(40g\) =
This code is used in sections 25a, 26a, 27a, and 27b.
The scanner for bison syntax

41a The fact that bison has a relatively straightforward grammar is partly 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.

\begin{verbatim}
\texttt{define all states (void)}
\end{verbatim}

41b Definitions and state declarations

It is convenient to abbreviate some commonly used subexpressions.

\begin{verbatim}
\texttt{Grammar lexer definitions} =
\texttt{Grammar lexer states} =
\texttt{letter} = [.abcdefgijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_]
\texttt{notletter} = [.abcdefgijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_] \setminus \%\{\}
\texttt{id} = \texttt{letter}\texttt{((letter) | [-0-9]).}
\texttt{int} = \texttt{[0-9]+}
\end{verbatim}

See also sections 42a and 42b.

This code is used in section 41a.
42a  Zero or more instances of backslash-newline. Following gcc, allow white space between the backslash and the newline.
\{Grammar lexer definitions 41b\} += \langle$splice$\rangle \big(\backslash[\![\backslash]()]\backslash\backslash\big)\star (\backslash\backslash)\star

42b  An equal sign, with optional leading whitespaces. This is used in some deprecated constructs.
\{Grammar lexer definitions 41b\} += \langle$eqopt$\rangle (\langle[\!\!\!\!\!\!]\rangle\star=)\? (\langle[\!\!\!\!\!\!]\rangle\star=)\?

42c  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 with using no 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 42c\} =
\#define register name (name) Define State(#name, name)
\#include "lo_states.h"
\#undef register name
This code is used in section 41a.

42d  A C-like comment in directives/rules.
\{Grammar lexer states 42d\} =
\{states-x\}_1: SC_YACC_COMMENT
See also sections 42e, 42f, 42g, 42h, 42i, 43a, and 43b.
This code is used in section 41b.

42e  Strings and characters in directives/rules.
\{Grammar lexer states 42d\} +=
\{states-x\}_1: SC_ESCAPED_STRING SC_ESCAPED_CHARACTER

42f  A identifier was just read in directives/rules. Special state to capture the sequence ‘identifier:’.
\{Grammar lexer states 42d\} +=
\{states-x\}_1: SC_AFTER_IDENTIFIER

42g  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 \lambda, as this simplifies our implementation. We match angle brackets in nested pairs: several languages use them for generics/template types.
\{Grammar lexer states 42d\} +=
\{states-x\}_1: SC_TAG

42h  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 42d\} +=
\{states-x\}_1: SC_PROLOGUE SC_BRACED_CODE SC_EPILOGUE SC_PREDICATE

42i  C and C++ comments in code.
\{Grammar lexer states 42d\} +=
\{states-x\}_1: SC_COMMENT SC_LINE_COMMENT
Strings and characters in code.

\[ \langle \text{Grammar lexer states} \rangle + = \]
\[ \{ \text{states-}\times \}: \ SC\text{\_STRING} SC\text{\_CHARACTER} \]

Bracketed identifiers support.

\[ \langle \text{Grammar lexer states} \rangle + = \]
\[ \{ \text{states-}\times \}: \ SC\text{\_BRACKETED\_ID} SC\text{\_RETURN\_BRACKETED\_ID} \]

\[ \langle \text{Grammar lexer \text{C} preamble} \rangle = \]
\[ \#\text{include} <\text{stdint.h}> \]
\[ \#\text{include} <\text{stdbool.h}> \]

This code is used in section 41a.

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.

\[ \langle \text{Grammar lexer options} \rangle = \]
\[ \{ \text{bison-bridge}\} \ast \]
\[ \{ \text{noyywrap}\} \ast \]
\[ \{ \text{nouput}\} \ast \]
\[ \{ \text{noinput}\} \ast \]
\[ \{ \text{reentrant}\} \ast \]
\[ \{ \text{noyy\_top\_state}\} \ast \]
\[ \{ \text{debug}\} \ast \]
\[ \{ \text{stack}\} \ast \]
\[ \{ \text{outfile}\} \]
\[ \text{lo.c} \]

This code is used in section 41a.

Tokenizing with regular expressions

Here is a full list of regular expressions recognized by the bison scanner.

\[ \langle \text{Grammar token regular expressions} \rangle = \]
\[ \langle \text{Scan grammar white space} \rangle \]
\[ \langle \text{Scan flex directives and options} \rangle \]
\[ \langle \text{Scan bison directives} \rangle \]
\[ \langle \text{Do not support zero characters} \rangle \]
\[ \langle \text{Scan after an identifier, check whether a colon is next} \rangle \]
\[ \langle \text{Scan bracketed identifiers} \rangle \]
\[ \langle \text{Scan a yacc comment} \rangle \]
\[ \langle \text{Scan a \text{C} comment} \rangle \]
\[ \langle \text{Scan a line comment} \rangle \]
\[ \langle \text{Scan a bison string} \rangle \]
\[ \langle \text{Scan a character literal} \rangle \]
\[ \langle \text{Scan a tag} \rangle \]
\[ \langle \text{Decode escaped characters} \rangle \]
\[ \langle \text{Scan user-code characters and strings} \rangle \]
\[ \langle \text{Strings, comments etc. found in user code} \rangle \]
\[ \langle \text{Scan code in braces} \rangle \]
\[ \langle \text{Scan prologue} \rangle \]
\[ \langle \text{Scan the epilogue} \rangle \]
\[ \langle \text{Add the scanned symbol to the current string} \rangle \]

This code is used in section 41a.
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.

This code is used in section 43e.
This code is used in section 45e.

%verbos
%yacc

▷ deprecated
%default[-]prec
%error[-]verbose
%expect[-]rr
%file-prefix(eqopt)
%fixed[-]output[-]files
%name[-]prefix(eqopt)
%no[-]default[-]prec
%no[-]lines
%output(eqopt)
%pure[-]parser
%token[-]table

▷ semantic predicate

%? ([t(r)(a)(n)\{]|\{[a*])\}
= |
;
(id)
(int)

0[xX] 0-9abcdefABCDEF+

▷ identifiers may not start with a digit; yet, don’t silently accept ifoo as 1 foo
(int)(id)
▷ characters
’,
▷ strings
”
▷ prologue
%
▷ code in between braces; originally preceded by \STRINGGROW
{
▷ a type
<<>
<>
<
%[

EOF
][X-Za-z0-9_<-]{" '*;|=/,|(|](a)(n)\{)}\]+ | .

return_p (verbose)
return_p (yacc)

deprecated (\%default-prec)
deprecated (\%define parse.error verbose)
deprecated (\%expect-rr)
deprecated (\%file-prefix)
deprecated (\%fixed-output-files)
deprecated (\%name-prefix)
deprecated (\%no-default-prec)
deprecated (\%no-lines)
deprecated (\%output)
deprecated (\%pure-parser)
deprecated (\%token-table)

enter(SC_PREDICATE) continue
(Possibly complain about a bad directive 47a)
return_p "=" "m
return_p "!" "m
return_p ";" "m
(Prepare an identifier 47b)
def_eq next {yyval {"\%anint {val\yytext } {val\yytoken mark } {val\yytoken mark } } } ) next
return; int
def_eq next {yyval {"\%hexint {val\yytext } {val\yytoken mark } {val\yytoken mark } } } ) next
return; int

fatal (invalid identifier: val\yytext )

enter(SC_ESCAPED_CHARACTER) continue

enter(SC_ESCAPED_STRING) continue

(Start assembling prologue code 47d)

\lonesting 0\l enter(SC_BRACED_CODE) continue

return_p "<<>" "m
return_p "<>" "m
\lonesting = 0\l enter(SC_TAG) continue
(Switch sections 47c)
let \bracketedidstr = \O
\bracketedidcontextstate \ystart
enter(SC_BRACKETED_ID) continue

\yterminate > (EOF) in INITIAL <
(Process a bad character 46b)
Some additional constructs needed to typeset simple flex declarations. This is not part of the original bison scanner.

\{ Scan flex directives and options 46a \}=

\begin{verbatim}
INITIAL:
  %option
  %x
  %s
\end{verbatim}

This code is used in section 43e.

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 46b \}=

\begin{verbatim}
def x next {\require{csname flex}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Set \langle debug \rangle flag 46c \}=

\begin{verbatim}
def x next {\require{debug}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Return lexer parameters 46d \}=

\begin{verbatim}
def x next {\require{lex-param}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Set \langle locations \rangle flag 46e \}=

\begin{verbatim}
def x next {\require{locations}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Return lexer and parser parameters 46f \}=

\begin{verbatim}
def x next {\require{both-param}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Return parser parameters 46g \}=

\begin{verbatim}
def x next {\require{parse-param}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.

\{ Set \langle pure-parser \rangle flag 46h \}=

\begin{verbatim}
def x next {\require{api.pure}
  \expandafter\expandafter\expandafter{\require{state-x}}
  \expandafter{\require{state-s}}
\end{verbatim}

This code is used in section 44a.
Possibly complain about a bad directive:

```plaintext
if | bad char |
    warn("invalid directive: val\yytext")
fi
```

This code is used in section 44a.

At this point we save the spelling and the location of the identifier. The token is returned later, after the context is known.

```plaintext
\langle Prepare an identifier \rangle =
\begin{verbatim}
def next { \yyval { \{ val\yytextpure \} { val\yytext } \} { val\yyfmark \} { val\yysmark } } next 
    let \bracketedidstr = \emptyset 
    enter( SC_AFTER_IDENTIFIER ) continue 
\end{verbatim}
```

This code is used in section 44a.

Switch sections:

```plaintext
\langle Switch sections \rangle =
\begin{verbatim}
add\percentpercentcount \R_if \percentpercentcount = 2 \R enter( SC_EPILOGUE )
fi
\end{verbatim}
```

This code is used in section 44a.

Start assembling prologue code:

```plaintext
\langle Start assembling prologue code \rangle =
\begin{verbatim}
def next { \postoks { \{ val\yyfmark \} { val\yysmark } } next 
    enter( SC_PROLOGUE ) continue 
\end{verbatim}
```

This code is used in section 44a.

Supporting 0ₜ complexifies our implementation for no expected added value.

```plaintext
\langle Do not support zero characters \rangle =
\begin{verbatim}
SC_ESCAPED_CHARACTER SC_ESCAPED_STRING SC_TAG:
0ₜ warn("invalid null character")
\end{verbatim}
```

This code is used in section 43e.

Scan after an identifier, check whether a colon is next:

```plaintext
\langle Scan after an identifier, check whether a colon is next \rangle =
\begin{verbatim}
SC_AFTER_IDENTIFIER:
[ : \langle EOF \rangle .
\end{verbatim}
```

This code is used in section 43e.

Process the bracketed part of an identifier:

```plaintext
\langle Process the bracketed part of an identifier \rangle =
\begin{verbatim}
if \bracketedidstr \emptyset 
    \bracketedidcontextstate \YSTART enter( SC_BRACKETED_ID )
    let next = continue 
else 
    \ROLLBACKCURRENTTOKEN 
    enter( SC_RETURN_BRACKETED_ID )
    def next { return "identifier" }
fi
next
\end{verbatim}
```

This code is used in section 47f.
Tokenizing with Regular Expressions

48a ⟨Process a colon after an identifier⟩ =
  if \bracketedidstr ≠ ∅
    enter(INITIAL)
  else
    enter(SC_RETURN_BRACKETED_ID)
  fi
  return «identifier: »
This code is used in section 47f.

48b ⟨Process a character after an identifier⟩ =
  \ROLLBACKCURRENTTOKEN
  if \bracketedidstr ≠ ∅
    enter(INITIAL)
  else
    enter(SC_RETURN_BRACKETED_ID)
  fi
  return «identifier»
This code is used in section 47f.

48c ⟨End the scan with an identifier⟩ =
  if \bracketedidstr ≠ ∅
    def next = continue
  else
    def next = warn «unexpected identifier in bracketed name: \yytext »
  fi
This code is used in section 48d.

48d ⟨Scan bracketed identifiers⟩ =
  SC_BRACKETED_ID:
    ⟨EOF⟩
    ⟨id⟩
    ⟨[ . A-Za-z0-9_/\(\)/\{\}\}\]\⟩
  ⟨Complain about unexpected end of file inside brackets⟩
  ⟨Process bracketed identifier⟩
  ⟨Finish processing bracketed identifier⟩
  ⟨Complain about improper identifier characters⟩
See also section 49c.
This code is used in section 43e.

48e ⟨Process bracketed identifier⟩ =
  if \bracketedidstr ≠ ∅
    def \bracketedidstr {{\val\yytextpure} {\val\yytext}}
      {\val\yyfmark} {\val\yysmark}
    let next = continue
  else
    def next = warn «an identifier expected»
  fi
  next
This code is used in section 48d.

48f ⟨Finish processing bracketed identifier⟩ =
  enter \bracketedidcontextstate
  if \bracketedidstr ≠ ∅
    def next = warn «an identifier expected»
  else
    if \bracketedidcontextstate = state(INITIAL) ∘
This code is used in section 48d.

49a ⟨Complain about improper identifier characters 49a⟩ =
    \texttt{fatal}(invalid\ character(s)\ in\ bracketed\ name: \texttt{val}\ yytext )
This code is used in section 48d.

49b ⟨Complain about unexpected end of file inside brackets 49b⟩ =
    \texttt{enter},\texttt{bracketedidcontextstate}
    \texttt{fatal}(unexpected\ end\ of\ file\ inside\ brackets)
This code is used in section 48d.

49c ⟨Scan bracketed identifiers 48d⟩ + =
    \texttt{SC\_RETURN\_BRACKETED\_ID}:
    .
    \texttt{(Return\ a\ bracketed\ identifier\ 49d)}

49d ⟨Return a bracketed identifier 49d⟩ =
    \texttt{\ROLLBACKCURRENTTOKEN}
    \texttt{\expandafter}{\yyval}\expandafter{\bracketedidstr }
    \texttt{let}\ \texttt{bracketedidstr} = ∅
    \texttt{enter}(INITIAL)
    \texttt{return} \texttt{"[identifier]"}_m
This code is used in section 49c.

49e \textit{Scanning a yacc comment. The initial /* is already eaten.}
    ⟨Scan a yacc comment 49e⟩ =
    \texttt{SC\_YACC\_COMMENT}:
    \texttt{(EOF)}
    \texttt{/*}
    \texttt{.}|(\texttt{*})
This code is used in section 43e.

49f \textit{Scanning a C comment. The initial /* is already eaten.}
    ⟨Scan a C comment 49f⟩ =
    \texttt{SC\_COMMENT}:
    \texttt{(EOF)}
    \texttt{*(splice)/}
This code is used in section 43e.

49g \textit{Scanning a line comment. The initial // is already eaten.}
    ⟨Scan a line comment 49g⟩ =
    \texttt{SC\_LINE\_COMMENT}:
    \texttt{(EOF)}
    \texttt{\texttt{\texttt{(splice)}}}
This code is used in section 43e.
50a  **Scanning a bison string, including its escapes. The initial quote is already eaten.**

\[
\langle \text{Scan a bison string} \rangle =
\]

\[
\text{SC_ESCAPED_STRING:
\langle EOF \rangle
\text{"}
\text{(n)}
\]

This code is used in section 43e.

50b  \[
\langle \text{Finish a bison string} \rangle =
\]

\[
\text{\$STRINGFINISH}
\text{def next (yylval{\{val\laststring\}{val\laststringraw}}
\{val\yysmark\}{val\yysmark\}})next
\text{enter(INITIAL)}
\text{return \langle string\rangle}
\]

This code is used in section 50a.

50c  **Scanning a bison character literal, decoding its escapes. The initial quote is already eaten.**

\[
\langle \text{Scan a character literal} \rangle =
\]

\[
\text{SC_ESCAPED_CHARACTER:
\langle EOF \rangle
\text{'}
\text{(n)}
\]

This code is used in section 43e.

50d  \[
\langle \text{Return an escaped character} \rangle =
\]

\[
\text{\$STRINGFINISH}
\text{def next (yylval{\{val\laststring\}{val\laststringraw}}
\{val\yysmark\}{val\yysmark\}})next
\text{\$STRINGFREE}
\text{enter(INITIAL)}
\text{return char}
\]

This code is used in section 50c.

50e  **Scanning a tag. The initial angle bracket is already eaten.**

\[
\langle \text{Scan a tag} \rangle =
\]

\[
\text{SC_TAG:
> ([<]c | ->)+
< \langle EOF \rangle
\]

This code is used in section 43e.

50f  \[
\langle \text{Finish a tag} \rangle =
\]

\[
\text{\$STRINGFINISH}
\text{addlonesting -1R}
\text{if_ \{lonesting <0R
\text{\$STRINGFINISH}
\text{def next (yylval{\{val\laststring\}{val\laststringraw}}
\{val\yysmark\}{val\yysmark\}})next
\text{\$STRINGFREE}
\text{enter(INITIAL)}
\text{def next (\langle return \langle tag\rangle \rangle}
\text{else
\text{\$STRINGGROW let next = continue
\fi
next
\]

This code is used in section 50e.
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.

\[
\text{\{Raise nesting level 51a\} =}\\
\text{\{STRINGGROW add\text{lonesting} 1}R}\\
\text{\{continue\}}
\]

This code is used in section 50e.

\[
\text{\{Decode escaped characters 51b\} =}\\
\text{\text{SC}\_ESCAPED\_STRING SC}\_ESCAPED\_CHARACTER:\\\n\text{\{\[0-7\][1,3]\}}\\
\text{\{\[x [0-9abcdefABCDEF]+\}}\\
\text{\{\a}\\
\text{\{\f}\\
\text{\{\n}\\
\text{\{\r}\\
\text{\{\t}\\
\text{\{\v}\\
\text{\{\" | ? | \} is shorter but confuses xgettext \&}}\\
\text{\{\[u | U[0-9abcdefABCDEF]_{(4)}\]}}\\
\text{\{\[. | \{\a\}]}\\
\]
\]
\]
\]
\]
\]

This code is used in section 43e.

\[
\text{\{Scan user-code characters and strings 51c\} =}\\
\text{\text{SC}\_CHARACTER SC}\_STRING:\\\n\text{\{\{splice\}\{\{splice\}\{\a\}\}\}}\\
\text{\{\{STRINGGROW continue\}}
\]

\[
\text{\text{SC}\_CHARACTER:}\\
\text{\{\}\\
\text{\{\n\}\\
\text{\{\EOF\}}
\]
\]

This code is used in section 43e.

\[
\text{\{Strings, comments etc. found in user code 51d\} =}\\
\text{\text{SC}\_BRACED\_CODE SC}\_PROLOGUE SC}\_EPILOGUE SC}\_PREDICATE:\\\n\text{\{\}\\
\text{\{\{splice\}\}}\\
\text{\{\{STRINGGROW contextstate YYSTART enter(SC\_CHARACTER)continue\}}
\]
\]
\]
\]

This code is used in section 43e.

\[
\text{\{Scanning some code in braces (actions, predicates). The initial \{ is already eaten. 51e\} =}\\
\text{\text{SC}\_BRACED\_CODE SC}\_PREDICATE:\\\n\text{\{\}}\\
\text{\{\%\{splice\}\}}\\
\text{\{\(<\{splice\}\}<\}}\\
\text{\{\EOF\}}
\]

\text{\{fatal\{unexpected end of line inside braced code\}}
\]

This code is used in section 43e.
SC_BRACED_CODE:
}  ⟨Add closing brace to the braced code 52a⟩

SC_PREDICATE:
}  ⟨Add closing brace to a predicate 52b⟩

This code is used in section 43e.

52a Unlike the original lexer, we do not return the closing brace as part of the braced code.
⟨Add closing brace to the braced code 52a⟩ =
addlonesting −1R
if lonesting < 0R
\STRINGFINISH
def next {yyval{{val\laststring}{val\yyfmark}{val\yysmark}}}{next
\
def next {return "$\{\ldots\}$"m}enter(INITIAL)
else
\STRINGGROW
let next = continue
fi
next

This code is used in section 51e.

52b ⟨Add closing brace to a predicate 52b⟩ =
addlonesting −1R
if lonesting < 0R
\STRINGFINISH
def next {yyval{{val\laststring}{val\yyfmark}{val\yysmark}}}{next
\
def next {return "$\%\{\ldots\}\$"m}enter(INITIAL)
else
\STRINGGROW
let next = continue
fi
next

This code is used in section 51e.

52c Scanning some prologue: from $\%\{$ (already scanned) to $\%\}$.
⟨Scan prologue 52c⟩ =

\SC_PROLOGUE:
$\%$
⟨EOF⟩  ⟨Finish braced code 52d⟩
\fatal(unexpected end of file inside prologue)⟩

This code is used in section 43e.

52d ⟨Finish braced code 52d⟩ =
\STRINGFINISH
def next {yyval{{val\laststring}{val\yyfmark}{val\yysmark}}}{next
\\enter(INITIAL)
return "$\%\{\ldots\}\$"m

This code is used in section 52c.

52e Scanning the epilogue (everything after the second $\%\}$, which has already been eaten).
⟨Scan the epilogue 52e⟩ =
\SC_EPILOGUE:
⟨EOF⟩  ⟨Handle end of file in the epilogue 53a⟩

This code is used in section 43e.
53a  ⦁ Handle end of file in the epilogue ⦁ =
\ROLLBACKCURRENTTOKEN
\STRINGfinish
\yyval = \laststring
enter(INITAL)
return; epilogue

This code is used in section 52e.

53b  ⦁ By default, grow the string obstack with the input. ⦁ =
\STRINGGROW

This code is used in section 43e.
The flex parser stack

The scanner generator, flex, uses bison to produce a parser for its input language. Its lexer is output by flex itself so both are reused to generate the parser and the scanner for pretty printing flex input.

This task is made somewhat complicated by the dependence of the flex input scanner on the correctly placed whitespace 1), as well as the reliance of the said scanner on rather involved state switching. Therefore, making subparsers for different fragments of flex input involves not only choosing an appropriate subset of grammar rules to correctly process the grammatic constructs but also setting up the correct lexer states.

The first subparser is designed to process a complete flex file. This parser is not currently part of any parser stack and is only used for testing. This is the only parser that does not rely on any custom adjustments to the lexer state to operate correctly.

\[
\text{\langle fip.yy 55a\rangle} = \\
\text{\langle Preamble for the flex parser 57c\rangle} \\
\text{\langle Options for flex parser 55b\rangle} \\
\text{\langle token table\rangle} \\
\text{\langle parse.trace\rangle} \ast \quad \text{(set as \langle debug\rangle)} \\
\text{\langle start\rangle} \quad \text{goal} \\
\text{\langle Postamble for flex parser 66i\rangle} \\
\text{\langle Token definitions for flex input parser 56d\rangle} \\
\text{\langle Productions for flex parser 57d\rangle} \\
\]

The selection of options for bison parsers suitable for SPLinT have been discussed earlier so we list them here without further comments.

\[
\text{\langle Options for flex parser 55b\rangle} = \\
\text{\langle token table\rangle} \ast \\
\text{\langle parse.trace\rangle} \ast \quad \text{(set as \langle debug\rangle)} \\
\text{\langle start\rangle} \quad \text{goal} \\
\]

This code is used in sections 55a, 56a, 56b, and 56c.

---

1) For example, each regular expression definition in section 1 must start at the beginning of the line.
56a A parser for section 1 (definitions and declarations). This parser requires a custom lexer, as discussed above, to properly set up the state. Short of this, the lexer may produce the wrong kind of tokens or even generate an error.

\[
\text{\langle ddp.yy 56a \rangle} =
\]
\[
\text{\langle Preamble for the \texttt{flex} parser \ 57c \rangle}
\]
\[
\text{\langle Options for \texttt{flex} parser \ 55b \rangle}
\]
\[
\text{\langle union \rangle}
\]
\[
\text{\langle Postamble for \texttt{flex} parser \ 66i \rangle}
\]
\[
\text{\langle Token definitions for \texttt{flex} input parser \ 56d \rangle}
\]
\[
\text{\langle Exclusive productions for \texttt{flex} section 1 parser \ 58c \rangle}
\]
\[
\text{\langle Productions for \texttt{flex} section 1 parser \ 58c \rangle}
\]

56b A parser for section 2 (rules and actions). This subparser must also use a custom set up for its lexer as discussed above.

\[
\text{\langle rap.yy 56b \rangle} =
\]
\[
\text{\langle Preamble for the \texttt{flex} parser \ 57c \rangle}
\]
\[
\text{\langle Options for \texttt{flex} parser \ 55b \rangle}
\]
\[
\text{\langle union \rangle}
\]
\[
\text{\langle Postamble for \texttt{flex} parser \ 66i \rangle}
\]
\[
\text{\langle Token definitions for \texttt{flex} input parser \ 56d \rangle}
\]
\[
\text{\langle Special \texttt{flex} section 2 parser productions \ 60j \rangle}
\]
\[
\text{\langle Productions for \texttt{flex} section 2 parser \ 60l \rangle}
\]

56c A parser for just the regular expression syntax. A custom lexer initialization must precede the use of this parser, as well.

\[
\text{\langle rep.yy 56c \rangle} =
\]
\[
\text{\langle Preamble for the \texttt{flex} parser \ 57c \rangle}
\]
\[
\text{\langle Options for \texttt{flex} parser \ 55b \rangle}
\]
\[
\text{\langle union \rangle}
\]
\[
\text{\langle Postamble for \texttt{flex} parser \ 66i \rangle}
\]
\[
\text{\langle Token definitions for \texttt{flex} input parser \ 56d \rangle}
\]
\[
\text{\langle Special productions for regular expressions \ 62d \rangle}
\]
\[
\text{\langle Rules for \texttt{flex} regular expressions \ 62f \rangle}
\]

56d Token and state declarations for the \texttt{flex} input scanner

Needless to say, the original grammar used by \texttt{flex} was not designed with pretty printing in mind (and why would it be?). Instead, efficiency was the goal which resulted in a number of lexical constructs being processed ‘on the fly’, as the lexer encounters them. Such syntax fragments never reach the parser, and
would not have a chance to be displayed by our routines, unless some grammar extensions and alterations were introduced.

To make the pretty printing possible, a number of new tokens have been introduced below that are later used in a few altered or entirely new grammar productions.

\[
\langle \text{Token definitions for flex input parser 56d} \rangle =
\]

\[
\begin{align*}
\text{char} & \quad \text{num} \\
\langle \text{xtate} \rangle & \quad \langle \text{name} \rangle \\
\langle \text{option} \rangle & \quad \langle \text{outfile} \rangle \\
\langle \text{header} \rangle & \quad \langle \text{extra type} \rangle \\
\langle \alpha \beta \rangle & \quad \langle 0..9 \rangle \\
\langle \star \rangle & \quad \langle . \rangle \\
\langle \wedge \rangle & \quad \langle A..Z \rangle \\
\langle \Leftrightarrow \rangle & \quad \langle \top \rangle \\
\langle \left \rangle & \quad \langle \cup \rangle \\
\langle \right \rangle & \quad \langle \cup \rangle \\
\end{align*}
\]

See also sections 57a and 57b.

This code is used in sections 55a, 56a, 56b, and 56c.

57a We introduce an additional option type to capture all the non-parametric options used by the flex lexer. The original lexer processes these options at the point of recognition, while the typesetting parser needs to be aware of them.

\[
\langle \text{Token definitions for flex input parser 56d} \rangle +=
\]

\[
\begin{align*}
\langle \text{top} \rangle & \quad \langle \text{array} \rangle \\
\langle \text{def} \rangle & \quad \langle \text{other} \rangle \\
\langle \text{deprecated} \rangle & \\
\end{align*}
\]

57b \textit{POSIX} and \textit{AT&T lex} place the precedence of the repeat operator, \{\}, below that of concatenation. Thus, \texttt{ab\{3\}} is \texttt{ababab}. Most other \textit{POSIX} utilities use an Extended Regular Expression (ERE) precedence that has the repeat operator higher than concatenation. This causes \texttt{ab\{3\}} to yield \texttt{abbb}.

In order to support the \textit{POSIX} and \textit{AT&T} precedence and the \textit{flex} precedence we define two token sets for the begin and end tokens of the repeat operator, \{p and \}p. The lexical scanner chooses which tokens to return based on whether posix\_compat or lex\_compat are specified. Specifying either posix\_compat or lex\_compat will cause \textit{flex} to parse scanner files as per the \textit{AT&T} and POSIX-mandated behavior.

\[
\langle \text{Token definitions for flex input parser 56d} \rangle +=
\]

\[
\begin{align*}
\langle \text{f} \rangle & \quad \langle \text{f} \rangle \\
\end{align*}
\]

57c \textbf{The grammar for flex input}

The original grammar has been carefully split into sections to facilitate the assembly of various subparsers in the \textit{flex}’s stack. Neither the \textit{flex} parser nor its scanner are part of the bootstrap procedure which simplifies both the input file organization, as well as the macro design. Some amount of preprocessing is still necessary, however, to extract the state names from the lexer file (see above for the explanation). We can nevertheless get away with an empty \texttt{C} preamble.

\[
\langle \text{Preamble for the flex parser 57c} \rangle =
\]

This code is used in sections 55a, 56a, 56b, and 56c.

57d \textbf{Productions for flex parser 57d} =

\[
\text{goal} : \quad \text{initlex} \text{ sect1 \ sect1end sect2 initforrule}
\]

\[
\text{sect1end} : \quad \text{SECTEND}
\]

\[
\text{initlex} : \quad \circ
\]

See also section 58b.

This code is used in section 55a.
58a  \{ Assemble a flex input file 58a \} =  
\begin{align*}
\Upsilon & \leftarrow (\text{val } T_2 \text{ val } T_4) \\
\end{align*}

This code is used in section 57d.

58b  \{ Productions for flex parser 57d \} + =  
\begin{align*}
\langle \text{Productions for flex section 1 parser 58c} \rangle \\
\langle \text{Productions for flex section 2 parser 60l} \rangle \\
\end{align*}

58c  \{ Exclusive productions for flex section 1 parser 58c \} =
\begin{align*}
g\text{oal} & : \ sect_1 \\
\end{align*}

This code is used in section 56a.

58d  \{ Assemble a flex section 1 file 58d \} =  
\begin{align*}
f\text{lex} & \expandafter{\text{val } T_1} \\
\end{align*}

This code is used in section 58c.

58e  \{ Productions for flex section 1 parser 58e \} =
\begin{align*}
\text{sect}_1 : \\
\text{sect}_1 \text{ startconddecl namelist}_1 \\
\text{sect}_1 \text{ options} \\
\text{error} \\
\text{startconddecl} : \\
\text{(state)} \\
\text{(xtate)} \\
\text{namelist}_1 : \\
\text{name} \\
\text{name} \\
\text{error} \\
\end{align*}

See also section 59d.

This code is used in sections 56a and 58b.

58f  \{ Add start condition declarations 58f \} =  
\begin{align*}
\Upsilon & \leftarrow (\text{val } T_1 \text{ lexcondcl } \text{val } T_2 \{ \text{val } T_4 \}) \\
\end{align*}

This code is used in section 58e.

58g  \{ Add options to section 1 58g \} =  
\begin{align*}
\Upsilon & \leftarrow (\text{val } T_1 \text{ val } T_2) \\
\end{align*}

This code is used in section 58e.

58h  \{ Create an empty section 1 58h \} =  
\begin{align*}
\Upsilon & \leftarrow () \\
\end{align*}

This code is used in section 58e.

58i  \{ Report an error in section 1 and quit 58i \} =  
\begin{align*}
\text{yyerror} \\
\end{align*}

This code is used in section 58e.

58j  \{ Prepare a state declaration 58j \} =  
\begin{align*}
\Upsilon & \leftarrow (\{ s \} \text{val } T_1) \\
\end{align*}

This code is used in section 58e.

58k  \{ Prepare an exclusive state declaration 58k \} =  
\begin{align*}
\Upsilon & \leftarrow (\{ x \} \text{val } T_1) \\
\end{align*}

This code is used in section 58e.
59a \{ Add a name to a list 59a \} = 
\[ \Upsilon \leftarrow (\text{val } T_1 \text{\textbackslash{}flnamesep } \{ \} \text{\textbackslash{}fname} \text{val } T_2) \]
This code is used in section 58e.

59b \{ Start a \textit{namelist} with a name 59b \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}fname } \text{val } T_1) \]
This code is used in section 58e.

59c \{ Report an error in \textit{namelist} and quit 59c \} = 
\texttt{\textbackslash{}yyerror}
This code is used in section 58e.

59d \{ Productions for \texttt{flex} section 1 parser 58e \} + = 
\begin{align*}
\textit{options} : & \quad \text{(Start an options list 59e)} \\
\textit{option} : & \quad \text{(Add a pointer option 59f)} \\
\textit{array} : & \quad \text{(Add an array option 59g)} \\
\textit{top} : & \quad \text{(Add a \texttt{\langle top\rangle} directive 59h)} \\
\textit{def} : & \quad \text{(Add a regular expression definition 59i)} \\
\textit{deprecated} : & \quad \text{(Output a deprecated option 60i)} \\
\textit{optionlist} : & \quad \text{(Make an empty option list 60a)} \\
\textit{outfile} : & \quad \text{(Record the name of the output file 60b)} \\
\textit{extra type} : & \quad \text{(Declare an extra type 60c)} \\
\textit{prefix} : & \quad \text{(Declare a prefix 60d)} \\
\textit{yyclass} : & \quad \text{( Declare a class 60e)} \\
\textit{header} : & \quad \text{( Declare the name of a header 60f)} \\
\textit{tables} : & \quad \text{( Declare the name for the tables 60g)} \\
\textit{other} : & \quad \text{(Output a non-parametric option 60h)}
\end{align*}

59e \{ Start an options list 59e \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}floptions } \{ \text{val } T_2) \]
This code is used in section 59d.

59f \{ Add a pointer option 59f \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}flptropt } \text{val } T_1) \]
This code is used in section 59d.

59g \{ Add an array option 59g \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}flarrayopt } \text{val } T_1) \]
This code is used in section 59d.

59h \{ Add a \texttt{\langle top\rangle} directive 59h \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}fstopopt } \text{val } T_1 \text{val } T_2) \]
This code is used in section 59d.

59i \{ Add a regular expression definition 59i \} = 
\[ \Upsilon \leftarrow (\text{\textbackslash{}flredef } \text{val } T_1 \text{val } T_2) \]
This code is used in section 59d.

59j \{ Add an option to a list 59j \} = 
\[ \Upsilon \leftarrow (\text{val } T_1 \text{val } T_2) \]
This code is used in section 59d.
This code is used in section 59d.

60b \langle Record the name of the output file \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{file\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60c \langle Declare an extra type \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{xtype\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60d \langle Declare a prefix \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{prefix\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60e \langle Declare a class \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{yyclass\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60f \langle Declare the name of a header \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{header\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60g \langle Declare the name for the tables \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{tables\}\val\Upsilon_3 \rangle \right. \]
This code is used in section 59d.

60h \langle Output a non-parametric option \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{other\}\val\Upsilon_1 \rangle \right. \]
This code is used in section 59d.

60i \langle Output a deprecated option \rangle =
\[ \Upsilon \leftarrow \langle \text{\textbackslash nx\ flopt\{deprecated\}\val\Upsilon_1 \rangle \right. \]
This code is used in section 59d.

60j \langle Special \texttt{flex} section 2 parser productions \rangle =
\[
\textbf{goal}:
\quad \text{sect}_2
\]
\[ (\text{Output section 2 60k}) \]
This code is used in section 56b.

60k \langle Output section 2 \rangle =
\[ \Omega\Upsilon_1 \]
This code is used in section 60j.

60l This portion of the grammar was changed to make it possible to read the action code.
\[
\text{Productions for \texttt{flex} section 2 parser 60l} =
\quad \text{sect}_2:
\quad \quad \text{sect}_2 \text{ scon initforrule flexrule} \texttt{n \n \n}
\quad \quad \text{sect}_2 \text{ scon \{ sect}_2 \}
\quad \quad \quad \circ
\quad \quad \text{sect}_2 \texttt{n}
\quad \textbf{initforrule} : \quad \circ
\]
\[
\text{Add a rule to section 2 61a}
\quad \text{Add a group of rules to section 2 61b}
\quad \text{Start an empty section 2 61c}
\quad \text{Add a bare action 61d}
\]
\[ \textbackslash flin@rule\texttt{true continue} \]
See also sections 61e and 62c.
This code is used in sections 56b and 58b.
61a \{Add a rule to section 2\ 61a\} = \iff\text{continued@action} \nu_b \leftarrow \langle \text{flactionc} \rangle \else \nu_b \leftarrow \langle \text{flaction} \rangle \fi \nu_a \expandafter{\text{\astformat@flaction}} \triangleleft \text{capture the formatting action} \langle \\rangle = \text{\textbackslash yypushx} \{ \text{val } \nu_\text{a} \} \text{\textbackslash on\astarray} \Upsilon \leftarrow \langle \text{\astarray\textbackslash lastcs} \rangle \text{on} \text{\astformat@flaction} \triangleleft \text{reset the format} \langle \\rangle \This code is used in section 60l.

61b \{Add a group of rules to section 2\ 61b\} = \Upsilon \leftarrow \langle \text{val } \Upsilon_\text{1}\text{\textbackslash flactiongroup}\text{val } \Upsilon_\text{2}\text{val } \Upsilon_\text{3}\text{val } \Upsilon_\text{4}\rangle \This code is used in section 60l.

61c \{Start an empty section 2\ 61c\} = \Upsilon \leftarrow () \This code is used in section 60l.

61d \{Add a bare action\ 61d\} = \Upsilon \leftarrow \langle \text{val } \Upsilon_\text{1}\text{\textbackslash flbareaction}\text{val } \Upsilon_\text{2} \rangle \This code is used in section 60l.

61e \{Productions for \texttt{flex} section 2 parser 60l\} + = \begin{align*} \text{\textbackslash scon\_stk\_ptr} : & \circ \\ \text{scon} : & \langle \ < \text{scon\_stk\_ptr} \text{namelist}_2 \rangle \\ & \langle \ast \rangle \\ & \circ \\ \text{namelist}_2 : & \langle \text{namelist}_2, \text{sconname} \rangle \\ \text{sconname} & \langle \text{error} \rangle \\ \text{sconname} : & \langle \text{\texttt{name}} \rangle \\ \text{\textbackslash scon\_list} : & \langle \text{val } \Upsilon_\text{1}\{ \text{val } \Upsilon_\text{2}\} \{ \text{val } \Upsilon_\text{3}\} \{ \text{val } \Upsilon_\text{4}\} \rangle \text{\textbackslash on\astarray \text{\textbackslash lastcs} \rangle \text{on} \text{sconname} \triangleleft \text{\Make a \texttt{name} into a start condition} \langle \\rangle \This code is used in section 60l.

61f \{Create a list of start conditions 61f\} = \Upsilon \leftarrow \langle \text{\textbackslash flsconlist}\{ \text{val } \Upsilon_\text{1}\} \{ \text{val } \Upsilon_\text{3}\} \{ \text{val } \Upsilon_\text{4}\} \rangle \This code is used in section 60l.

61g \{Create a universal start condition 61g\} = \Upsilon \leftarrow \langle \text{\textbackslash flsconuniv}\text{val } \Upsilon_\text{3} \rangle \This code is used in section 60l.

61h \{Create an empty start condition 61h\} = \Upsilon \leftarrow () \This code is used in section 60l.

61i \{Add a start condition to a list 61i\} = \Upsilon \leftarrow \langle \text{val } \Upsilon_\text{1}\text{\textbackslash flnamesep}\text{val } \Upsilon_\text{2}\text{val } \Upsilon_\text{3} \rangle \This code is used in section 60l.

61j \{Start a list with a start condition name 61j\} = \langle \text{\texttt{Copy the value}} \rangle \This code is used in section 60l.
62a ⟨Report an error compiling a start condition list 62a⟩ = yyerror
This code is used in section 61e.

62b ⟨Make a «name» into a start condition 62b⟩ =
\( \Upsilon \leftarrow ("\text{name} \text{val} \Upsilon_1) \)
This code is used in section 61e.

62c ⟨Productions for flex section 2 parser 60l⟩ + = 61e
 ⟨Rules for flex regular expressions 62f⟩

62d ⟨Special productions for regular expressions 62d⟩ =
goal:
flexrule

62e The parsed regular expression is output in the \texttt{table} register. It is important to ensure that whenever this parser is used inside another parser that uses \texttt{table} for output, the changes to this register stay local. The \texttt{\textbackslash frexproc} macro in yyunion.sty ensures that all the changes are local to the parsing macro.

62f ⟨Rules for flex regular expressions 62f⟩ =

flexrule:
\( ^\text{\textbackslash rule} \)
\( \Upsilon \leftarrow ("\text{\textbackslash flbolrule} \text{val} \Upsilon_2)\{\text{val} \var v_a \} \)
\( \Upsilon \leftarrow ("\text{\textbackslash fleof} \text{val} \Upsilon_1)\{\text{val} \var v_a \} \)
See also sections 63a, 63i, 64d, 65f, and 66e.

62g ⟨Match a rule at the beginning of the line 62g⟩ =
\( \var v_a\text{\textbackslash expandafter}\{\text{\textbackslash astformat@flrule} \}
\text{\textbackslash astformat@flrule} \varnothing
\Upsilon \leftarrow ("\text{\textbackslash flbolrule} \text{val} \Upsilon_2)\{\text{val} \var v_a \} \)
This code is used in section 62f.

62h ⟨Match an ordinary rule 62h⟩ =
\( \var v_a\text{\textbackslash expandafter}\{\text{\textbackslash astformat@flrule} \}
\text{\textbackslash astformat@flrule} \varnothing
\Upsilon \leftarrow ("\text{\textbackslash flrule} \text{val} \Upsilon_1)\{\text{val} \var v_a \} \)
This code is used in section 62f.

62i ⟨Match an end of file 62i⟩ =
\( \Upsilon \leftarrow ("\text{\textbackslash fleof} \text{val} \Upsilon_1) \)
This code is used in section 62f.

62j ⟨Report an error and quit 62j⟩ = yyerror
This code is used in section 62f.
63a  \langle \text{Rules for \textit{flex} regular expressions } 62f \rangle + =

\begin{align*}
\text{rule} : \\
\text{re}_2 \text{ re} \\
\text{re}_2 \text{ re} \$, \\
\text{re} \$, \\
\text{re} \\
\text{re} : \\
\text{re} | \text{series} \\
\text{series} \\
\text{re}_2 : \text{ re} / \\
\end{align*}

\begin{align*}
\langle \text{Match a regular expression with a trailing context } 63b \rangle \\
\langle \text{Disallow a repeated trailing context } 63c \rangle \\
\langle \text{Match a regular expression at the end of the line } 63d \rangle \\
\langle \text{Match an ordinary regular expression } 63e \rangle \\
\langle \text{Match a sequence of alternatives } 63f \rangle \\
\langle \text{Match a sequence of singletons } 63g \rangle \\
\langle \text{Prepare to match a trailing context } 63h \rangle \\
\end{align*}

\begin{align*}
63b  \langle \text{Match a regular expression with a trailing context } 63b \rangle = \\
\pi_2(\Upsilon_1) \Rightarrow v_a \pi_3(\Upsilon_1) \Rightarrow v_b \\
\Upsilon \leftarrow \langle \text{\textit{fl}re\textit{trail}} \{ \text{val } v_a \} \{ \text{val } v_b \} \{ \text{val } \Upsilon_2 \} \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63c  \langle \text{Disallow a repeated trailing context } 63c \rangle = \\
\text{\textit{yyerror}} \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63d  \langle \text{Match a regular expression at the end of the line } 63d \rangle = \\
\Upsilon \leftarrow \langle \text{\textit{fl}reateol} \{ \text{val } \Upsilon_1 \} \{ \text{val } \Upsilon_2 \} \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63e  \langle \text{Match an ordinary regular expression } 63e \rangle = \\
\langle \text{Copy the value } 66f \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63f  \langle \text{Match a sequence of alternatives } 63f \rangle = \\
\Upsilon \leftarrow \langle \text{\textit{fl}or} \{ \text{val } \Upsilon_1 \} \{ \text{val } \Upsilon_2 \} \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63g  \langle \text{Match a sequence of singletons } 63g \rangle = \\
\langle \text{Copy the value } 66f \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63h  \langle \text{Prepare to match a trailing context } 63h \rangle = \\
\Upsilon \leftarrow \langle \text{\textit{fl}trail} \{ \text{val } \Upsilon_1 \} \{ \text{val } \Upsilon_2 \} \rangle \\
\text{This code is used in section } 63a.
\end{align*}

\begin{align*}
63i  \langle \text{Rules for \textit{flex} regular expressions } 62f \rangle + = \\
\text{series} : \\
\text{series \textit{singleton}} \\
\text{\textit{singleton}} \\
\text{series} \{ p \ \text{num} , \ \text{num} \}_p \\
\text{series} \{ p \ \text{num} , \}_p \\
\text{series} \{ p \ \text{num} \}_p \\
\end{align*}

\begin{align*}
\langle \text{Extend a series by a singleton } 63j \rangle \\
\langle \text{Match a singleton } 63k \rangle \\
\langle \text{Match a series of specific length } 64a \rangle \\
\langle \text{Match a series of minimal length } 64b \rangle \\
\langle \text{Match a series of exact length } 64c \rangle \\
\end{align*}

\begin{align*}
63j  \langle \text{Extend a series by a singleton } 63j \rangle = \\
\Upsilon \leftarrow \langle \text{val } \Upsilon_1 \text{val } \Upsilon_2 \rangle \\
\text{This code is used in section } 63i.
\end{align*}

\begin{align*}
63k  \langle \text{Match a singleton } 63k \rangle = \\
\langle \text{Copy the value } 66f \rangle \\
\text{This code is used in section } 63i.
\end{align*}
64a ⟨Match a series of specific length 64a⟩ =  
    ⟨Create a series of specific length 64h⟩  
This code is used in section 63i.

64b ⟨Match a series of minimal length 64b⟩ =  
    ⟨Create a series of minimal length 64i⟩  
This code is used in section 63i.

64c ⟨Match a series of exact length 64c⟩ =  
    ⟨Create a series of exact length 64j⟩  
This code is used in section 63i.

64d ⟨Rules for flex regular expressions 62f⟩ + =  
    singleton:  
        singleton *  
        singleton +  
        singleton ?  
        singleton {f num, num}l  
        singleton {f num}l  
        .  
    fullcl  
    PREVCCL  
    " string "  
    ( re )  
    char  

64e ⟨Create a lazy series match 64e⟩ =  
    Υ ← (\n<flrepeat> {val Υ₁})  
This code is used in section 64d.

64f ⟨Create a nonempty series match 64f⟩ =  
    Υ ← (\n<flrepeatstrict> {val Υ₁})  
This code is used in section 64d.

64g ⟨Create a possible single match 64g⟩ =  
    Υ ← (\n<flrepeatonce> {val Υ₁})  
This code is used in section 64d.

64h ⟨Create a series of specific length 64h⟩ =  
    Υ ← (\n<flrepeattm> {val Υ₁} {val Υ₃} {val Υ₅})  
This code is used in sections 64a and 64d.

64i ⟨Create a series of minimal length 64i⟩ =  
    Υ ← (\n<flrepeatsen> {val Υ₁} {val Υ₃})  
This code is used in sections 64b and 64d.

64j ⟨Create a series of exact length 64j⟩ =  
    Υ ← (\n<flrepeaten> {val Υ₁} {val Υ₃})  
This code is used in sections 64c and 64d.

64k ⟨Match (almost) any character 64k⟩ =  
    Υ ← (\n<fldot> val Υ₁)  
This code is used in section 64d.
65a \{ Match a character class 65a \} =
  \{ Copy the value 66f \}
This code is used in section 64d.

65b \{ Match a PREVCCL 65b \} =
  \{ Copy the value 66f \}
This code is used in section 64d.

65c \{ Match a string 65c \} =
  \( \gamma \leftarrow ("\langle \text{flstring} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_2 \} \{ \text{val} \, \gamma_3 \} \rangle \) \)
This code is used in section 64d.

65d \{ Match an atom 65d \} =
  \( v_a \\expandafter{\\astformat@flparens} \)
\( \text{let} \ \astformat@flparens \emptyset \)
  \( \gamma \leftarrow ("\langle \text{flparens} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_2 \} \{ \text{val} \, \gamma_3 \} \{ \text{val} \, v_a \} \rangle \) \)
This code is used in section 64d.

65e \{ Match a specific character 65e \} =
  \( \gamma \leftarrow ("\langle \text{flchar} \text{ val} \, \gamma_1 \rangle \) \)
This code is used in section 64d.

65f \{ Rules for flex regular expressions 62f \} + =
  \( \textit{fulcl} : \)
  \( \textit{fulcl} \setminus \textit{braceccl} \)
  \( \textit{fulcl} \cup \textit{braceccl} \)
  \( \textit{braceccl} : \)
  \( [ \ \textit{ccl} ] \)
  \( [ ^ \ \textit{ccl} ] \)
  \( \textit{ccl} : \)
  \( \textit{ccl} \text{ char} \setminus \text{ char} \)
  \( \textit{ccl} \text{ char} \)
  \( \textit{ccl} \text{ expr} \)
  \( \circ \)
  \( \text{\textit{fulcl} \setminus \textit{braceccl}} \)
  \( \text{\textit{fulcl} \cup \textit{braceccl}} \)
  \( \text{\textit{braceccl}} : \)
  \( [ \ \textit{ccl} ] \)
  \( [ ^ \ \textit{ccl} ] \)

65g \{ Subtract a character class 65g \} =
  \( \gamma \leftarrow ("\langle \text{flccldiff} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_3 \} \rangle \) \)
This code is used in section 65f.

65h \{ Create a union of character classes 65h \} =
  \( \gamma \leftarrow ("\langle \text{flcclunion} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_3 \} \rangle \) \)
This code is used in section 65f.

65i \{ Turn a basic character class into a character class 65i \} =
  \( \{ \text{Copy the value 66f} \} \)
This code is used in section 65f.

65j \{ Create a character class 65j \} =
  \( \gamma \leftarrow ("\langle \text{flbraceccl} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_2 \} \{ \text{val} \, \gamma_3 \} \rangle \) \)
This code is used in section 65f.

65k \{ Create a complementary character class 65k \} =
  \( \gamma \leftarrow ("\langle \text{flbracecclneg} \{ \text{val} \, \gamma_1 \} \{ \text{val} \, \gamma_3 \} \{ \text{val} \, \gamma_4 \} \rangle \) \)
This code is used in section 65f.
66a  ⟨ Add a range to a character class 66a ⟩ =
    \( \Upsilon \leftarrow (\text{val } T_1 \triangleleft f\text{lclr}n\text{ge} \{^* f\text{lchar} \text{ val } T_2 \} \{^* f\text{lchar} \text{ val } T_4 \}) \)

This code is used in section 65f.

66b  ⟨ Add a character to a character class 66b ⟩ =
    \( \Upsilon \leftarrow (\text{val } T_1 \triangleleft f\text{lchar} \text{ val } T_2) \)

This code is used in section 65f.

66c  ⟨ Add an expression to a character class 66c ⟩ =
    \( \Upsilon \leftarrow (\text{val } T_1 \triangleleft f\text{lcclexpr} \text{ val } T_2) \)

This code is used in section 65f.

66d  ⟨ Create an empty character class 66d ⟩ =
    \( \Upsilon \leftarrow \{ \} \)

This code is used in section 65f.

66e  ⟨ Rules for flex regular expressions 62f ⟩ + =

\[ \begin{align*}
\text{ccl_expr} & : \\
\alpha & | \beta | ( ) | (\rightarrow) | \{^* . 9\} | \{^* \} \\
\alpha . Z & | (\rightarrow) | . | 0 . Z | \{^* A . Z\} \\
^* \alpha & | ^* \beta | (\rightarrow) | \{^* 0 . 9\} | \{^* \} \\
\rightarrow & | . | \{^* .\} | \{^* 0 . Z\} | \{^* a . Z\} | \{^* A . Z\}
\end{align*} \]

\( \text{string} : \text{ string char | o} \)  (Make an empty regular expression string 66h)

66f  ⟨ Copy the value 66f ⟩ =
    \( \Upsilon \leftarrow (\text{val } T_1) \)

This code is used in sections 57d, 61j, 63c, 63g, 63k, 65a, 65b, 65i, and 66e.

66g  ⟨ Extend a flex string by a character 66g ⟩ =
    \( \Upsilon \leftarrow (\text{val } T_1 \triangleleft f\text{lchar} \text{ val } T_2) \)

This code is used in section 66e.

66h  ⟨ Make an empty regular expression string 66h ⟩ =
    \( \Upsilon \leftarrow \{ \} \)

This code is used in section 66e.

66i  This is needed to get the yytoknum array. A trivial declaration suffices.

\( \langle \text{Postamble for flex parser 66i} \rangle = \)

\#define YYPRINT(file, type, value) yyprintf(file, type, value)

static void yyprintf(FILE *file, int type, YYSTYPE value)
{
    \}

This code is used in sections 55a, 56a, 56b, and 56c.
The original lexer for flex grammar relies on a few rules that use ‘trailing context’. The lexing mechanism implemented by SPLinT cannot process such rules properly in general. The rules used by flex match fixed-length trailing context only, which makes it possible to replace them with ordinary patterns and use `yyless()` in the actions.

\[
\langle \text{Preamble for flex lexer} \rangle = \\
\langle \text{Options for flex input lexer} \rangle \\
\langle \text{State definitions for flex input lexer} \rangle \\
\langle \text{Definitions for flex input lexer} \rangle \\
\langle \text{Postamble for flex input lexer} \rangle \\
\langle \text{Patterns for flex lexer} \rangle \\
\langle \text{Auxiliary code for flex lexer} \rangle
\]

67b This code is used in section 67a.

67c There are a few options that are necessary to ensure that the lexer functions properly. Some of them (like `caseless`) directly affect the behavior of the scanner, others (e.g. `noyy_top_state`) prevent generation of unnecessary code.

\[
\langle \text{Options for flex input lexer} \rangle = \\
\langle \text{caseless} \rangle_1 * \\
\langle \text{nodefault} \rangle_1 * \\
\langle \text{stack} \rangle_1 * \\
\langle \text{noyy_top_state} \rangle_1 * \\
\langle \text{nostdinit} \rangle_1 *
\]

This code is used in section 67a.
Additional options for flex input lexer 68a =
(bison-bridge)*
(noyywrap)*
(nounput)*
(noinput)*
(reentrant)*
(debug)*
(stack)*
(outfile) = "fil.c"

This code is used in section 67a.

Regular expression and state definitions

The lexer uses a large number of states to control its operation. Both section 1 and section 2 rules rely on the scanner being in the appropriate state. Otherwise (see symbols.sty example) the lexer may parse the same fragment in a wrong context.

State definitions for flex input lexer 68b =

State definitions for flex input lexer 68c =

Somewhat counterintuitively, flex definitions do not always have to be fully formed regular expressions. For example, after

\begin{verbatim}
\text{(BOGUS)}
^ \[a-
\end{verbatim}

one can form the following action:

\begin{verbatim}
\text{(BOGUS)t}
\end{verbatim}

although without the ‘‘‘-‘ in the definition of ‘(BOGUS)’ flex would have put a ‘’’ inside the character class. We will assume such (rather counterproductive) tricks are not used. If the definition is not a well-formed regular expression the pretty printing will be suspended.

Definitions for flex input lexer 68c =

Postamble for flex input lexer 68d =

This code is used in section 67a.
69a Regular expressions for flex input scanner

The code below treats (pointer) and (array) the same way it treats (option) while typesetting.

\[\text{Patterns for flex lexer 69a} =\]

\[
\begin{align*}
\text{INITIAL:} & \quad \backslash flindented\text{coderestrue enter(CODEBLOCK)continue} \\
\text{− (\_\_)} & \quad \backslash flinc@linenum \text{return (pointer)} \\
\text{− /} & \quad \text{push state(COMMENT) continue} \\
\text{− \#(\_\_\_)} & \quad \text{push state(LINEDIAR) continue} \\
\text{− %a (NAME)} & \quad \text{returnp (state)} \\
\text{− %x (NAME)} & \quad \text{returnp (xtate)} \\
\text{− \% (.→)} & \quad \text{(Start a C code section 69b)} \\
\text{− %top (\{ \},( \}),(→)} & \quad \text{(Begin the top directive 69c)} \\
\text{− %top .} & \quad \text{fatal(malformed ‘% top’ directive)} \\
\text{− (\_\_)} & \quad \text{; ▶ discard ▼} \\
\text{− % pointer .(→)} & \quad \text{(Start section 2 69d)} \\
\text{− %array .(→)} & \quad \text{\textbackslash flinc@linenum return (pointer)} \\
\text{− %option} & \quad \text{\textbackslash flinc@linenum return (array)} \\
\text{− % (LEXOPT)(\_\_)\{0..\}} & \quad \text{enter(OPTION) return (option)} \\
\text{− % (LEXOPT)(\_\_) .(→)} & \quad \text{fatal(unrecognized ‘%’ directive: vall\textbackslash \text{yytext} \)}} \\
\text{− % \(\text{porksexcan}()\)} & \quad \text{\textbackslash flinc@linenum continue ▶ allows blank lines in section 1 ▼} \\
\text{− (NAME)} & \quad \text{\textbackslash flinc@linenum continue ▶ maybe end of comment line ▼} \\
\text{(SCNAME)} & \quad \text{\textbackslash flinc@linenum continue ▶ \textbackslash flinc@linenum \text{continue} ▼} \\
\text{− (\_\_)} & \quad \text{\textbackslash flinc@linenum continue ▶ ▼} \\
\text{− (\_\_)} & \quad \text{\textbackslash flinc@linenum continue ▶ ▼} \\
\text{\textbackslash flindented\text{codestrue enter(CODEBLOCK)continue}} & \quad \text{\textbackslash flindented\text{codestrue enter(CODEBLOCK)continue}} \\
\end{align*}\]

See also sections 69f, 70b, 71b, 73a, 73b, 73f, 77b, 78b, 79b, 79c, 79d, and 80b.

This code is used in section 67a.

69b \(\text{Start a C code section 69b} =\)

\[
\backslash flinc@linenum \\
\backslash flindented\text{codestrue enter(CODEBLOCK)continue}
\]

This code is used in section 69a.

69c Ignore setting brace_start_line as it is only used internally to report errors.

\(\text{Begin the (top) directive 69c} =\)

\[
\backslash flinc@linenum \text{\text{def} \text{\textbackslash flbrace@depth} \{1\} \text{push state(CODEBLOCK MATCH BRACE continue}}
\]

This code is used in section 69a.

69d \(\text{Start section 2 69d} =\)

\[
\text{def\text\text{\textbackslash flsectnum} \{2\} \text{def} \text{\textbackslash flbracelevel} \{0\} \text{enter(SECT2PROLOG) returnp SECTEND}}
\]

This code is used in section 69a.

69e \(\text{Copy the name and start a definition 69e} =\)

\[
\text{\textbackslash fliddadeffalse enter(PICKUPDEF) returnp (def)}
\]

This code is used in section 69a.

69f \(\text{Patterns for flex lexer 69a} + =\)

\[
\text{COMMENT:} \quad */ \quad \text{continue} \\
\text{⋆} \quad \text{continue} \\
\text{(M4QSTART)} \quad \text{continue}
\]

This code is used in section 69a.

69a 69f 69b 70b 71b 73a 73b 73f 77b 78b 79b 79c 79d 80b
70a 〈Pop state if code braces match 70a〉 =
\fdec\nibrace@depth
def \nibrace@depth = 0R o
  return \nelse
  continue
fi
This code is used in section 69f.

70b 〈Patterns for flex lexer 69a〉 + =
\fpick<\ndef:

\(\langle + \rangle\)
\(\langle NOT_WS \rangle \langle [r] (n) \rangle \langle \rangle \)
\(\langle \langle \rangle \langle \rangle \)

〈Skip trailing whitespace, save the definition 70c〉 =
def \nifindef \{ \n\val\yytext \}\n\val\yytextpure \}\n\val\yymark \}\n\val\yysmark \}\n\fidda\n\ntrue continue

This code is used in section 70b.
292 SPLINT
293
71a ⟨Complain if not inside a definition, continue otherwise 71a⟩ =
    \iffldidadef
    \yyval\expandafter{\flnmdef }
    def next {\flinclinenum enter(INITIAL) return ⟨def ew⟩ }
    else
    def next {fatal ⟨incomplete name definition⟩ }
  fi next
This code is used in section 70b.

71b ⟨Patterns for flex lexer 69a⟩ + =

  \flinclinenum enter(INITIAL) continue
  \floption@sense\true continue
  return.
  (Toggle option sense 72a)

  7bit    return@opt (other)
  8bit    return@opt (other)
  align   return@opt (other)
  always-interactive return@opt (other)
  array   return@opt (other)
  ansi-definitions return@opt (other)
  ansi-prototypes return@opt (other)
  backup  return@opt (other)
  batch   return@opt (other)
  bison-bridge return@opt (other)
  bison-locations return@opt (other)
  c++ return@opt (other)
  caseful | case-sensitive return@opt (other)
  caseless | case-insensitive return@opt (other)
  debug return@opt (other)
  default return@opt (other)
  ecs return@opt (other)
  fast return@opt (other)
  full return@opt (other)
  input return@opt (other)
  interactive return@opt (other)
  lex-compat (Set lex_compat 72b)
  posix-compat (Set posix_compat 72c)
  main return@opt (other)
  meta-ecs return@opt (other)
  never-interactive return@opt (other)
  perf-report return@opt (other)
  pointer return@opt (other)
  read return@opt (other)
  reentrant return@opt (other)
  reject return@opt (other)
  stack return@opt (other)
  stdinit return@opt (other)
  stdout return@opt (other)
  unistd return@opt (other)
  unput return@opt (other)
  verbose return@opt (other)
  warn return@opt (other)
  yylineno return@opt (other)
  yymore return@opt (other)
  yyurap return@opt (other)
  yy_push_state return@opt (other)
yy_pop_state
yy_top_state
yy_scan_buffer
yy_scan_bytes
yy_scan_string
yalloc
yyrealloc
yyfree
yyget_debug
yyset_debug
yyget_extra
yyset_extra
yyget_leng
yyget_text
yyget_lineno
yyset_lineno
yyget_in
yyset_in
yyget_out
yyset_out
yyget_lval
yyset_lval
yyget_lloc
yyset_lloc
extra-type
outfile
prefix
yyclass
header(-file)?
tables-file
tables-verify
"[^\n]+" .
((\[a–m[z|n[a–np[z]](\alphaβ)+]+)
|.
fatal\{unrecognized %option: val\yytext\}

72a \{Toggle option\_sense 72a\} =
    \iffloption@sense
    \floption@sensefalse
    else
    \floption@sensetrue
    fi continue
This code is used in section 71b.

72b \{Set lex\_compat 72b\} =
    \iffloption@sense
    \fllex@compattrue
    else
    \fllex@compatfalse
    fi return^{opt}(\other)
This code is used in section 71b.

72c \{Set posix\_compat 72c\} =
    \iffloption@sense
    \fposix@compattrue
    else
    \fposix@compatfalse
    fi return^{opt}(\other)
This code is used in section 71b.
The RECOVER state is never used for typesetting and is only added for completeness.

Like bison, flex allows insertion of C code in the middle of the input file.

All the code inside is ignored.

A pattern below (for the character class processing) had to be broken into two lines. A symbol (⊙) was inserted to indicate that a break had occurred. The macros for flex typesetting use a different mechanism from that of bison macros and allow typographic corrections to be applied to sections of the flex code represented by various nonterminals. These corrections can also be delayed. For the details, an interested reader may consult yyunion.sty.
\%(\FIRST\_CCL\_CHAR | (CCL\_EXPR))\%∗
⟨CCL\_CHAR | (CCL\_EXPR)⟩.
{−} \{+} return \$
⟩
⟨}
\{\NAME\} \{\;\}
/*
(?# ⟨Determine if this is a comment 76d⟩
( ⟨Determine if this is a parametric group or return a parenthesis 77a⟩
)/∗?⟨⟨⟩}⟩
\∖\RETURNCHAR

74a \langle Start braced code in section 2 \rangle =
def \flbrace\{1\}
\indent@codefalse \doing@codeblocktrue
enter(\PERCENT\_BRACE\_ACTION)
continue
This code is used in section 73f.

74b \langle Process a repeat pattern \rangle =
\yyless\{1\}enter(\NUM)
\iffflex@compat
  def next \{return \{p\}\}
else
  \iffposix@compat
    def next \{return \{p\}\}
  else
    def next \{return \{f\}\}
  fi
fi
next
This code is used in section 73f.

74c \langle Process braced code in the middle of section 2 \rangle =
def \fbracelevel\{1\}
enter(\PERCENT\_BRACE\_ACTION)
\iflin@rule
  \fdoing@rule@actiontrue
  \flin@rulefalse
  def next \{return \\n\}
else
  let next continue
fi
next
This code is used in section 73f.

74d This action has been changed to accomodate the new grammar. The separator (|) is treated as an ordinary (empty) action.
\langle Process a deferred action \rangle =
\iffison@skip@ws \whitespace ignored, still inside a pattern \rangle
\yylessafter\{}
let next continue
else
  \flinc@linenum
  \fdoing@rule@actiontrue
  \flin@rulefalse

\flcontinued@actiontrue
\unput{\n } enter(ACTION)
def x next {"return:\n } fi next
This code is used in section 73f.

75a ⟨Process a comment inside a pattern 75a⟩ =
\ifflsf@skip@ws
push state(COMMENT_DISCARD)
elset\unput{\/*}
def flbracelevel{0}
\flcontinued@actionfalse enter(ACTION)
fi continue
This code is used in section 73f.

75b ⟨Decide whether to start an action or skip whitespace inside a rule 75b⟩ =
\ifflsf@skip@ws
let next continue
elset def flbracelevel{0}
\flcontinued@actionfalse enter(ACTION)
\ifflin@rule
\fldoing@rule@actiontrue
\flin@rulefalse
def next {"return:\n }
elset else
let next continue
fi
fi next
This code is used in section 73f.

75c ⟨Finish the line and/or action 75c⟩ =
\ifflsf@skip@ws
\flinc@linenum
let next continue
elset def flbracelevel{0}
\flcontinued@actionfalse enter(ACTION)
\unput{\n } enter(ACTION)
\ifflin@rule
\fldoing@rule@actiontrue
\flin@rulefalse
def next {"return:\n }
elset else
let next continue
fi
fi next
This code is used in section 73f.
76a  (Start section 3 76a) =
   def flsectnum {3}
      enter(SECT3)
      yyterminate
   This code is used in section 73f.

76b  (Start processing a character class 76b) =
   def flnmstr {val yytext}
      yyless {1}
      enter(FIRSTCCL)
      yylexreturnraw[
   This code is used in section 73f.

76c  Return a special char and return the whitespace back into the input. The braces and the possible trailing whitespace will be dealt with by the typesetting code.
   (Process a named expression after checking for whitespace at the end 76c) =
   def flend@ch {val yytextlastchar}
      if\flend@ch = {'
         \flend@is@wsfalse
      else
         \flend@is@wstrue
      fi\expandafter{\astformat@flnametok}
      let \astformat@flnametok Ø
      def next {yyval{{\flnametok{val yytext}{val v_a}}{ }{val yyfmark}{val yysmark} }}next
         \iffiflend@is@w
            \unput { }
         fi
      \return{char}
   This code is used in section 73f.

76d  (Decide if this is a comment 76d) =
   \iffiflsf@skip@ws
      push state(COMMENT_DISCARD)
      continue
   else
      yyless {1}
      yylexreturnraw/
   fi
   This code is used in section 73f.

76e  (Determine if this is extended syntax or return a parenthesis 76e) =
   \iffiflex@compat
      def next {yyless {1}\flsf@push\yylexreturnraw ()
   else
      \iffilposix@compat
         def next {yyless {1}\flsf@push\yylexreturnraw ()
      else
         def next { push state(EXTENDED_COMMENT) }\fi
      fi next
   This code is used in section 73f.
(Determine if this is a parametric group or return a parenthesis 77a) =
\lfslf@push
\iffilex@compat
def next {\yyless{1}}
else
  \iffposix@compat
def next {\yyless{1}}
else
def next {enter(GROUP_WITH_PARAMS)}
fi
fi next
\yylexreturnraw{
This code is used in section 73f.

(Patterns for flex lexer 69a) + =

SC:
⟨\(\mu\rangle\langle\leftrightarrow\rangle\langle\mu\rangle\)
[\[\star\]]  
>  
\(\langle\text{SCNAME}\rangle\)
.

CARET ISHEK
~

QUOTE:
\[^{\langle\(n\)\rangle}\]
"  
\(\langle\leftarrow\rangle\)

GROUP_WITH_PARAMS:
:\  
i  
s  
x

GROUP_MINUS_PARAMS:
:\  
i  
s  
x

FIRSTCCL:
\[^{-}\langle\(n\)\rangle\]
\(^{-}\langle\[\langle\rangle\]\)
.

CCL:
\[^{-}\langle\[\langle\rangle\]\]
\[^{-}\langle\langle\rangle\] \]
\[^{-}\langle\langle\rangle\] \]
.

FIRSTCCL CCL:
[:alnum:]  
[:alpha:]  
[:blank:]  
[:cntrl:]  
[:digit:]  
set \(\gamma\) and return\(^{ccl}\) \((\alpha\beta)\)
set \(\gamma\) and return\(^{ccl}\) \((\alpha\beta)\)
set \(\gamma\) and return\(^{ccl}\) \((\alpha\beta)\)
set \(\gamma\) and return\(^{ccl}\) \((\alpha\beta)\)
set \(\gamma\) and return\(^{ccl}\) \((\alpha\beta)\)
[:graph:] set \( \Upsilon \) and return\(^{ccl}\) (\( \ast \))
[:lower:] set \( \Upsilon \) and return\(^{ccl}\) (a..z)
[:print:] set \( \Upsilon \) and return\(^{ccl}\) (\( \cdot \))
[:punct:] set \( \Upsilon \) and return\(^{ccl}\) (\( \cdot \))
[:space:] set \( \Upsilon \) and return\(^{ccl}\) (\( \cdot \))
[:upper:] set \( \Upsilon \) and return\(^{ccl}\) (A..Z)
[:xdigit:] set \( \Upsilon \) and return\(^{ccl}\) (0..Z)
[:alnum:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \alpha \beta \))
[:alpha:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \alpha \beta \))
[:blank:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \))
[:cntrl:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \mapsto \mapsto \))
[:digit:] set \( \Upsilon \) and return\(^{ccl}\) (~0..9)
[:graph:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \))
[:lower:] set \( \Upsilon \) and return\(^{ccl}\) (~a..z)
[:print:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \))
[:punct:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \))
[:space:] set \( \Upsilon \) and return\(^{ccl}\) (~\( \))
[:upper:] set \( \Upsilon \) and return\(^{ccl}\) (~A..Z)
[:xdigit:] set \( \Upsilon \) and return\(^{ccl}\) (~0..Z)

\( \langle \text{CCL_EXPR} \rangle \)

NUM:
\[ [(0..9)]\]

\( \langle \text{---} \rangle \)

78a \( \langle \text{Finish the repeat pattern 78a} \rangle = \)

\( \text{enter(SECT2)} \)
\( \text{\iffllex@compat} \)
\( \text{def next \{ return \} } \)\( p \)\}
\else
\( \text{\ifflposix@compat} \)
\( \text{def next \{ return \} } \)\( p \)\}
\else
\( \text{def next \{ return \} } \)\( f \)\}
\fi
\fi next

This code is used in section 77b.

78b \( \langle \text{Patterns for flex lexer 69a} \rangle + = \)

\( \text{PERCENT_BRACE_ACTION:} \)
\( \langle \text{---} \rangle \)
\( \text{ACTION} \)
\( / * \)
\( \text{CODEBLOCK ACTION:} \)
\( \text{reject} \)
\( \text{yymore} \)
\( \text{\( (M4QSTART) \)} \)
\( \text{\( (M4QEND) \)} \)
\( \langle \text{---} \rangle \)

\( \text{def\{f1bracelevel\} 0} \text{ continue} \)

\( \text{push state(COMMENT) continue} \)

\( \text{continue} \)
\( \text{continue} \)
\( \text{continue} \)
\( \text{continue} \)
\( \langle \text{Process a newline inside a braced group 79a} \rangle \)
This actions has been modified to output \n.

\{Process a newline inside a braced group \} = \flinc@linenum
if \flbracelevel = 0
  \ifdef@rule@action
    return \n
  else
    continue
  fi
  \ifdef@codeblockfalse
  enter(SECTION)
else
  \ifdef@codeblockfalse
  \ifdef@indentfalse
  enter(SECTION)
  \ifdef@rule@actionfalse
  return x
  \ifdef@rule@actionfalse
  \ifdef@codeblockfalse
  enter(SECTION)
fi
fi
continue
fi
This code is used in section 78b.

\{Patterns for flex lexer \} = + = \△
ACTION: \( reject \) and \( gymore() \) are checked for above, in \PERCENT\_BRACE\_ACTION \( \langle \)
\{ \}
\( M4Q\_START \)
\( M4Q\_END \)
\[ \langle \alpha\beta \rangle \{^+ / \nu \}[1]^{c+} \]
\[ [1] \]
\( \langle NAME \rangle \)
\' \( [^\nu]^{c} | \nu \).\'
\'
\( \langle \to \rangle \)
\'.

This actions has been modified to output \n.

\{Process a newline inside an action \} = \flinc@linenum
if \flbracelevel = 0
  \ifdef@rule@action
    return \n
  else
    continue
  fi
  \ifdef@codeblockfalse
  enter(SECTION)
fi
This code is used in section 78b.

\{Patterns for flex lexer \} = + = \△
ACTION\_STRING: \( ^{c+} \)
\( . \)
\{\texttt{\textasciitilde}\}

\texttt{\textasciitilde}

\begin{verbatim}
\texttt{COMMENT \texttt{COMMENT\_DISCARD} \texttt{ACTION} \texttt{ACTION\_STRING}}
\texttt{(EOF)}
\texttt{EXTENDED\_COMMENT \texttt{GROUP\_WITH\_PARAMS} \texttt{GROUP\_MIN\_MAX\_PARAMS}}
\texttt{(EOF)}
\texttt{SECT\_QUOTE \texttt{FIRST\_CCL\_CCL}}
\texttt{(ESCSEQ)}
\end{verbatim}

\texttt{80a} \{Process an escaped sequence 80a\} =
\begin{verbatim}
if_\texttt{\textbackslash YSTART = \textbackslash number\textbackslash csname flexstate\textbackslash parsernamespace FIRST\textbackslash CCL\textbackslash endcsname} 
  \texttt{enter(CCL)}
fi
\texttt{\textbackslash RETURN\textbackslash CHAR}
\end{verbatim}

This code is used in section 79d.

\texttt{80b} \{Patterns for \texttt{flex} lexer 69a\} +=
\begin{verbatim}
\texttt{SECT\_3:}
  \texttt{\{M4QSTART\}}
  \texttt{\{M4QEND\}}
  [\texttt{\{n\}\texttt{*}}(\texttt{\{n\}\texttt{?}})
  \texttt{. | (\{n\}))
  \texttt{(EOF)}
  \texttt{\{\texttt{*}\}}
  \texttt{. | (\{n\})}
\end{verbatim}

\texttt{80c} \{Auxiliary code for \texttt{flex} lexer 80c\} =
\begin{verbatim}
\texttt{void \texttt{define\_all\_states}(void)}
\{
  \texttt{(Collect state definitions for the \texttt{flex} lexer 80d)}
\}
\end{verbatim}

This code is used in section 67a.

\texttt{80d} \{Collect state definitions for the \texttt{flex} lexer 80d\} =
\begin{verbatim}
\texttt{\#define _register\_name(name) Define\_State(#name, name)}
\texttt{\#include \texttt{\"fil\_states\_h\"}}
\texttt{\#undef _register\_name}
\end{verbatim}

This code is used in section 80c.
The name parser

81a What follows is an example parser for the term 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⟩ =

..............................
⟨Name parser C preamble⟩
..............................
⟨Bison options⟩

⟨union⟩  (Union of parser types)

..............................
⟨Name parser C postamble⟩
..............................
⟨Token and types declarations⟩
⟨Parser productions⟩

This code is used in section 81a.

81b ⟨Bison options⟩ =
⟨token table⟩*
⟨parse.trace⟩*  (set as ⟨debug⟩)
⟨start⟩  full_name

This code is used in section 81a.

81c ⟨Token and types declarations⟩ =
%[a...Z0...9]*     [a...Z0...9]*
ext 1
* or ? \c

opt na
r [0...9]*

⟨meta identifier⟩

This code is used in section 81a.

81d ⟨Parser productions⟩ =

full_name:
identifier_string suffixes_{opt}
⟨meta identifier⟩
quoted_name suffixes_{opt}

⟨Compose the full name⟩
⟨Turn a «meta identifier» into a full name⟩
⟨Compose the full name⟩
This code is used in section 81a.

82a  ⟨Compose the full name 82a⟩ =
     $\tau \leftarrow \langle \text{val } \tau_1 \text{val } \tau_2 \rangle \backslash \text{namechars } \tau$
This code is used in section 81d.

82b  ⟨Turn a «meta identifier» into a full name 82b⟩ =
     $\pi_1(\tau_1) \mapsto v_a$
     $\pi_2(\tau_1) \mapsto v_b$
     $\tau \leftarrow \langle ^{\text{nx}\backslash \text{dotsp}^{\text{nx}}\backslash \text{sfxnone} } \rangle$
This code is used in section 81d.

82c  ⟨Attach option name 82c⟩ =
     $\pi_1(\tau_1) \mapsto v_a$
     $\pi_2(\tau_1) \mapsto v_b$
     $\tau \leftarrow \langle ^{\text{nx}\backslash \text{optstr} } \rangle$
This code is used in section 81d.
83a  ⟨Start with an identifier 83a⟩ =
\[\pi_1(\bar{Y}_1) \mapsto v_a\]
\[\pi_2(\bar{Y}_1) \mapsto v_b\]
\[\bar{Y} \leftarrow (\mathrm{idstr} \{val v_a\} \{val v_b\})\]
This code is used in sections 81d and 83l.

83b  ⟨Start with a tag 83b⟩ =
\[\pi_1(\bar{Y}_2) \mapsto v_a\]
\[\pi_2(\bar{Y}_2) \mapsto v_b\]
\[\bar{Y} \leftarrow (\mathrm{idstr} \{<val v_a>\} \{<val v_b>\})\]
This code is used in section 81d.

83c  ⟨Start with a quoted string 83c⟩ =
\[\pi_1(\bar{Y}_2) \mapsto v_a\]
\[\pi_2(\bar{Y}_2) \mapsto v_b\]
\[\text{\sansfirst v}_b\]
\[\bar{Y} \leftarrow (\mathrm{chstr} \{val v_a\} \{val v_b\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83d  ⟨Start with an escaped character 83d⟩ =
\[\pi_1(\bar{Y}_2) \mapsto v_a\]
\[\pi_2(\bar{Y}_2) \mapsto v_b\]
\[\bar{Y} \leftarrow (\mathrm{chstr} \{val v_a\} \{val v_b\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83e  ⟨Start with a < string 83e⟩ =
\[\bar{Y} \leftarrow (\mathrm{chstr} \{<\} \{<\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83f  ⟨Start with a > string 83f⟩ =
\[\bar{Y} \leftarrow (\mathrm{chstr} \{\greaterthan\} \{\greaterthan\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83g  ⟨Start with an _ string 83g⟩ =
\[\bar{Y} \leftarrow (\mathrm{chstr} \{\uscoreletter\} \{\uscoreletter\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83h  ⟨Start with a - string 83h⟩ =
\[\bar{Y} \leftarrow (\mathrm{chstr} \{-\} \{-\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83i  ⟨Start with a $ string 83i⟩ =
\[\bar{Y} \leftarrow (\text{\safemath} \{\safemath\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83j  ⟨Start with a . string 83j⟩ =
\[\bar{Y} \leftarrow (\text{\safemath} \{.\} \{.\} \text{\visflag \termvstring} \{\} \{\})\]
This code is used in section 81d.

83k  ⟨Prepare a bison stack name 83k⟩ =
\[\bar{Y} \leftarrow (\mathrm{bidstr} \{\safemath\} \{\safemath\})\]
This code is used in section 81d.

83l  ⟨Turn a qualifier into an identifier 83l⟩ =
⟨Start with an identifier 83a⟩
This code is used in section 81d.
84a ⟨Attach an identifier 84a⟩ =
π₂(Y₁) ↦ va
v_a ← v_a +sx [ v ]
π₁(Y₂) ↦ vb
v_b ← v_b +sv_b
π₃(Y₁) ↦ vb
v_b ← v_b +sx [ v ]
π₂(Y₂) ↦ vc
v_b ← v_b +sv_c
Υ ← ⟨\text{idstr} \{ val v_a \} \{ val v_b \}⟩

This code is used in sections 81d and 84b.

84b ⟨Attach qualifier to a name 84b⟩ =
⟨Attach an identifier 84a⟩

This code is used in section 81d.

84c An integer at the end of an identifier (such as id1) is interpreted as a suffix (similar to the way METAFONT treats identifiers, and mft typesets them) to mitigate a well-intentioned but surprisingly inconvenient feature of CTANGLE, namely outputting something like id.1 as id\_.1 in an attempt to make sure that integers do not interfere with structure dereferences. For this to produce meaningful results, a stricter interpretation of \[a...z0...9]* syntax is required, represented by the id\_strict syntax below.

⟨Attach an integer 84c⟩ =
Υ ← ⟨\text{id\_strict} \{ val Y_a \} \{ val Y_b \}⟩

This code is used in section 81d.

84d ⟨Process quoted name 84d⟩ =
π₁(Y₂) ↦ va
π₂(Y₂) ↦ vb
Υ ← ⟨\text{idstr} \{ val v_a \} \{ val v_b \} \text{visflag} \{ \text{termvstring} \} \{ \}⟩

This code is used in section 81d.

84e ⟨Process quoted option 84e⟩ =
π₁(Y₂) ↦ va
π₂(Y₂) ↦ vb
Υ ← ⟨\text{optstr} \{ val v_a \} \{ val v_b \} \text{visflag} \{ \text{termvstring} \} \{ \}⟩

This code is used in section 81d.

84f ⟨Attach suffixes 84f⟩ =
Υ ← ⟨\text{dotsp} \text{val Y}_2⟩

This code is used in sections 81d and 84g.

84g ⟨Attach qualified suffixes 84g⟩ =
⟨Attach suffixes 84f⟩

This code is used in section 81d.

84h ⟨Start with a named suffix 84h⟩ =
Υ ← ⟨\text{sfxn} \text{val Y}_1⟩

This code is used in section 81d.

84i ⟨Start with a numeric suffix 84i⟩ =
Υ ← ⟨\text{sfxi} \text{val Y}_1⟩

This code is used in section 81d.

---

1) This allows, for example, names like \(\text{term}_0\) while leaving \(\text{char2int}\) in its ‘natural’ form.
SPLINT

85a \{ Add a dot separator 85a \} = \\[ \text{\textbackslash Y} \leftarrow \langle \text{\textbackslash T}_1 \text{\textbackslash dotsp} \rangle \] \\
This code is used in section 81d.

85b \{ Attach integer suffix 85b \} = \\[ \text{\textbackslash Y} \leftarrow \langle \text{\textbackslash T}_1 \text{\textbackslash sfxi} \text{\textbackslash val \textbackslash T}_2 \rangle \] \\
This code is used in section 81d.

85c \{ Attach a named suffix 85c \} = \\[ \text{\textbackslash Y} \leftarrow \langle \text{\textbackslash T}_1 \text{\textbackslash sfxn} \text{\textbackslash val \textbackslash T}_2 \rangle \] \\
This code is used in section 81d.

85d \{ Attach a qualifier 85d \} = \\[ \text{\textbackslash Y} \leftarrow \langle \text{\textbackslash T}_1 \text{\textbackslash qual} \text{\textbackslash val \textbackslash T}_2 \rangle \] \\
This code is used in section 81d.

85e \{ Start suffixes with a qualifier 85e \} = \\[ \text{\textbackslash Y} \leftarrow \langle \text{"\textbackslash qual} \text{\textbackslash val \textbackslash T}_1 \rangle \] \\
This code is used in section 81d.

85f C preamble. In this case, there are no ‘real’ actions that our grammar performs, only \LaTeX\ output, so this section is empty. \\
\{ Name parser C preamble 85f \} = \\
This code is used in section 81a.

85g C postamble. It is tricky to insert function definitions that use \texttt{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 85g \} = \\
\#define YYPRINT(file, type, value) yyprint(file, type, value) \\
static void yyprint(FILE *file, int type, YYSTYPE value) \\
{} \\
This code is used in section 81a.

85h Union of types. \\
\{ Union of parser types 85h \} = \\
This code is used in section 81a.
The name scanner

The scanner for lexing term names is admittedly *ad hoc* and rather redundant. A minor reason for this is to provide some flexibility for name typesetting. Another reason is to let the existing code serve as a template for similar procedures in other projects. At the same time, it must be pointed out that this scanner is executed multiple times for every *bison* section, so its efficiency directly affects the speed at which the parser operates.

```plaintext
⟨small_lexer.ll⟩ =
⟨Lexer definitions⟩

...............
⟨Lexer C preamble⟩
...............
⟨Lexer options⟩

⟨Regular expressions⟩

void define_all_states(void)
{
   (Collect all state definitions)
}

⟨Lexer definitions⟩ =
⟨Lexer states⟩

   ⟨letter⟩
   ⟨c-escchar⟩ [\fnrtv]
   ⟨wc⟩ [\"$.<>\] [\abcdefghijklmnopqrstuvwxyzABCDEFJKLMNOPQRSTUVWXYZ]\ \.)
   ⟨id⟩ (letter)((letter) [\-0-9]).
   ⟨id_strict⟩ (letter)((letter) [\-0-9]).(letter))?
   ⟨meta_id⟩ *(id_strict) *?
   ⟨int⟩ [0-9]+.

This code is used in section 87a.

87c ⟨Collect all state definitions⟩ =

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

This code is used in section 87a.
88a Strings and characters in directives/rules.
\[\langle \text{Lexer states } 88a \rangle = \langle \text{states}-x \rangle : \text{SC_ESCAPED_STRING SC_ESCAPED_CHARACTER}\]
This code is used in section 87b.

88b \[\langle \text{Lexer C preamble } 88b \rangle = \#\text{include } <\text{stdint.h}>\]
\[\#\text{include } <\text{stdbool.h}>\]
This code is used in section 87a.

88c \[\langle \text{Lexer options } 88c \rangle = \langle \text{bison-bridge} \rangle f *\]
\[\langle \text{noyywrap} \rangle f *\]
\[\langle \text{noinput} \rangle f *\]
\[\langle \text{reentrant} \rangle f *\]
\[\langle \text{noyy_top_state} \rangle f *\]
\[\langle \text{debug} \rangle f *\]
\[\langle \text{stack} \rangle f *\]
\[\langle \text{outfile} \rangle f "\text{smalllexer.c}"\]
This code is used in section 87a.

88d \[\langle \text{Regular expressions } 88d \rangle = \langle \text{Scan white space } 88e \rangle\]
\[\langle \text{Scan identifiers } 88f \rangle\]
This code is used in section 87a.

88e White space skipping.
\[\langle \text{Scan white space } 88e \rangle = \[\text{\(f\langle n\text{t}\langle v\rangle\}\]}\]
continue
This code is used in section 88d.

88f 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.
\[\langle \text{Scan identifiers } 88f \rangle = \]
\[
\%\text{binary} \quad \text{return, } \%[a...Z0...9]*
\%\text{code} \quad \text{return, } \%[a...Z0...9]*
\%\text{debug} \quad \text{return, } \%[a...Z0...9]*
\%\text{default-prec} \quad \text{return, } \%[a...Z0...9]*
\%\text{define} \quad \text{return, } \%[a...Z0...9]*
\%\text{defines} \quad \text{return, } \%[a...Z0...9]*
\%\text{destructor} \quad \text{return, } \%[a...Z0...9]*
\%\text{dprec} \quad \text{return, } \%[a...Z0...9]*
\%\text{empty} \quad \text{return, } \%[a...Z0...9]*
\%\text{error-verbose} \quad \text{return, } \%[a...Z0...9]*
\%\text{expect} \quad \text{return, } \%[a...Z0...9]*
\%\text{expect-rr} \quad \text{return, } \%[a...Z0...9]*
\%\text{file-prefix} \quad \text{return, } \%[a...Z0...9]*
\%\text{fixed-output-files} \quad \text{return, } \%[a...Z0...9]*
\%\text{initial-action} \quad \text{return, } \%[a...Z0...9]*
\%\text{glr-parser} \quad \text{return, } \%[a...Z0...9]*
\%\text{language} \quad \text{return, } \%[a...Z0...9]*
\%\text{left} \quad \text{return, } \%[a...Z0...9]*
\%\text{lex-param} \quad \text{return, } \%[a...Z0...9]*
\]
This code is used in section 88d.
\begin{verbatim}
90a  \langle Prepare to process an identifier 90a \rangle =
    \texttt{return } v [a...Z0...9]*
This code is used in section 88f.

90b  \langle Prepare to process a meta-identifier 90b \rangle =
    \texttt{return } \texttt{\textbackslash meta identifier}
This code is used in section 88f.

90c  \langle React to a bad character 90c \rangle =
    \texttt{if } t[bad char]
        \texttt{fatal(\texttt{invalid character(s): }val\texttt{\textbackslash ytext })}
    \texttt{fi}
This code is used in section 88f.
\end{verbatim}
10

Forcing bison and flex to output \TeX

91a 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.

91b 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 91b} 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 91b} =
(Auxiliary function definitions 99g}

int main(int argc, char **argv)
{
    (Local variable and type declarations 93c}
    (Establish defaults 101a}
    (Command line processing variables 101d}
    (Process command line options 101e}
    switch (mode) {
        (Various output modes 92a}
    default: break;
}
fprintf(stderr,"Outputting tables and actions\n");
if (tables_out) {
    fprintf(stderr,"\tablestables...\tablestables...");
    (Perform output 96a}
    fprintf(stderr,"\actions\actions...");
    (Output action switch, if any 99c}

91
}  
else {  
    fprintf(stderr,"No output, exiting\n");  
    exit(0);  
}  
fprintf(stderr,"done, cleaning up\n");  

return 0;  
}  

This code is cited in section 91b.

92a 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 92a⟩ =  

This code is used in section 91b.

92b Standard library declarations for memory management routines, some syntactic sugar, command line processing, and variadic functions are all that is needed.  

⟨Outer definitions 92b⟩ =  

\#include <stdlib.h>  
\#include <stdbool.h>  
\#include <stdarg.h>  
\#include <assert.h>  
\#include <string.h>  

See also section 101b.  

This code is used in section 97f.

92c 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.

As of version 3.0 of bison some critical arrays, namely, yyprhs and yyrhs are no longer generated (even internally) which significantly reduces bison’s useability as a parser generator. As an example, the yyrthree array, which is necessary for processing ‘inline’ actions is computed in bs.w using the two arrays mentioned in the previous sentence. There does not seem to be any other way to access this information. A number of tools (GNU and otherwise) have taken the path of narrowing the field of application to a few use cases envisioned by the maintainers. This includes compilers, as well.

There is a 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 (well, there is logic to it, however choosing poor design in order to punish a few bad players seems like a rather inferior option).

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: the loopholes that make them
possible have been steadily getting on the chopping block of the C standards committee. Indeed, \texttt{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 \texttt{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 \texttt{gcc} does not seem to mind, and it gets rid of warnings about dropping a \texttt{const} qualifier whenever an \texttt{assert} is encountered. Since the macro is not recursively expanded, this will only work if \ldots \texttt{FUNCTION} is treated as a pseudo-variable, as it is in \texttt{gcc}, not a macro.

\begin{verbatim}
#define const
#define \_PRETTY\_FUNCTION\_ (char *) \_PRETTY\_FUNCTION\_
#define \_FUNCTION\_ (char *) \_FUNCTION\
\end{verbatim}

\texttt{93a} 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.

\begin{verbatim}
#define forever for ( ; ; )
\end{verbatim}

\texttt{93b} 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’\footnote{1).}

\begin{verbatim}
#include <stdio.h>
FILE *tables_out;
\end{verbatim}

\texttt{93c} 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 \texttt{CWEB} approach.

\begin{verbatim}
\{ Local variable and type declarations 93c \} =
\end{verbatim}

\begin{verbatim}
\texttt{94b} struct output_d {
( Output descriptor fields 93d )
};
struct output_d output_desc \{\{ Default outputs 94a \}\};
\end{verbatim}

See also sections 94b, 97b, 98c, 100a, and 101c.

This code is used in section 91b.

\texttt{93d} To declare each table field in the global output descriptor, all one has to do is to provide a general pattern.

\begin{verbatim}
\{ Output descriptor fields 93d \} =
#define _register_table_d(name) bool output_##name:1;
\end{verbatim}

\begin{verbatim}
\texttt{97b} \langle Table names 96c \rangle
#undef _register_table_d
\end{verbatim}

See also sections 97b and 98d.

This code is used in section 93c.

\footnote{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 \texttt{CTANGLE’d} code}
94a  Same for assigning default values to each field.

\begin{verbatim}
Default outputs 94a =
#define _register_table_d(name) .output_##name <= 0,  // do not output any tables by default
(Table names 96c)
#undef _register_table_d
See also sections 97c and 98e.
This code is used in section 93c.
\end{verbatim}

94b  Each descriptor is populated using the same approach.

\begin{verbatim}
(Table names 96c)
#define _register_table_d(name) struct table_d name##_desc = {0};
Table names 96c)
#undef _register_table_d
\end{verbatim}

94c  The flag --optimize-tables affects the way tables are output.

\begin{verbatim}
(global variables and types 94c)
static int optimize_tables <= 0;
See also sections 94e, 96d, 97a, 98b, and 99d.
This code is used in section 97f.
\end{verbatim}

94d  It is set using the command line option below.

\begin{verbatim}
(options without arguments 94d)
register_option("optimize-tables", no_argument, &optimize_tables, 1,"")
See also section 96e.
This code is used in section 102a.
\end{verbatim}

94e  The reason to implement the table output routine as a macro is to avoid writing separate functions for tables of different types of data (strings 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).

\begin{verbatim}
#define output_table (table_desc, table_name, stream)
if (output_desc.output_##table_name) {
  int i, j <= 0;
  if (optimize_tables) {
    fprintf(stream, "\setoptopt{%s}%%\n", table_desc.name);
    if (table_desc.optimized_numeric) {
      fprintf(stream, "\beginoptoptimizednonnumeric{%s}%%\n", table_desc.name);
    }
    for (i = 0; i < sizeof (table_name)/sizeof (table_name[0]) - 1; i++) {
      if (table_desc.formatter) {
        table_desc.formatter (stream, i);
      }
      else {
        fprintf (stream, table_desc.optimized_numeric, table_desc.name, i, table_name[i]);
      }
    }
    if (table_desc.formatter) {
      table_desc.formatter (stream, -i);
    }
    else {

1) A somewhat cleaner way to achieve the same effect is to use the _Generic facility of C11.
fprintf(stream, table_desc.optimized_numeric, table_desc.name, i, table_name[i]);
}
if (table_desc.cleanup) {
    table_desc.cleanup(&table_desc);
}
else {
    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);
        }
        else {
            if (table_name[i]) {
                j = fprintf(stream, table_desc.separator, table_name[i]);
            }
            else {
                j = fprintf(stream, "\", table_desc.null);
            }
        }
        if (j > MAXPRETTY_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, "\", table_desc.null_postamble);
        }
    }
    if (table_desc.cleanup) {
        table_desc.cleanup(&table_desc);
    }
}

(Global variables and types 94c) + =

struct table_d {
    (Generic table descriptor fields 95a)
};

95a (Generic table descriptor fields 95a) =
    char *name;
    char *preamble;
    char *separator;
    char *postamble;
    char *null_postamble;
    char *null;
char *optimized_numeric;
bool prettify;
int(*formatter)(FILE *int);
void(*cleanup)(struct table_d *);

This code is used in section 94e.

96a 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.

\(\langle\text{Perform output 96a}\rangle = \langle\text{Output all tables 96b}\rangle\)

See also section 98f.

This code is used in section 91b.

96b One more application of ‘gather the names first then process’ technique.

\(\langle\text{Output all tables 96b}\rangle = \#\text{define register_table_d(name) output_table(name##.desc, name, tables_out)}\); 

\(\langle\text{Table names 96c}\rangle = \#\text{undef register_table_d}\)

This code is used in section 96a.

96c Tables will be output by each driver. Placeholder here, for \texttt{CWEAVE’s} piece of mind.

\(\langle\text{Table names 96c}\rangle = \)

This code is used in sections 93d, 94a, 94b, 96b, and 108b.

96d Action output invokes a totally new level of dirty code. If tables, constants, and tokens are just data structures, actions are executable commands. We can only hope to cycle through all the actions, which is enough to successfully use \texttt{bison} and \texttt{flex} to generate \TeX. The \texttt{switch} statement containing the actions is embedded in the parser function so to get access to each action one has to coerce \texttt{yyparse()} to jump to each case. Here is where we need the table manipulation. The appropriate code is highly specific to the program used (since \texttt{bison} and \texttt{flex} parsing and scanning functions had to be ‘reverse engineered’ to make them do what we want), so at this point we simply declare the options controlling the level of detail and the type of actions output.

\(\langle\text{Global variables and types 94c}\rangle + = \)

\(\text{static int bare_actions} = 0;\)

\(\text{static int optimize_actions} = 0;\)

96e 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 \texttt{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 \texttt{usage()} function.

\(\langle\text{Options without arguments 94d}\rangle + = \)

\(\text{register_option(“bare-actions”, no_argument, &bare_actions, 1,”“)}\)

\(\text{register_option(“optimize-actions”, no_argument, &optimize_actions, 1,”“)}\)
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.

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

Nothing is output by default, including actions.

```c
bool output_actions;1;
```

Each function below outputs the TeX code of the appropriate action when the 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.\(^2\)

```c
#define TeX__(string, ...) fprintf(tables_out,"\%s\n", string)
#define TeX(string) TeX__(string)
#define TeXb(string) TeX__(string)
#define TeXa(string) TeX__(string)
#define TeXf(string) TeX__(string)
#define TeXfo(string) TeX__(string)
#define TeXao(string) TeX__(string)
#define YY_FATAL_ERROR(message) 
    fprintf(tables_out,"\%s\n", message)
    #define TeX__(string, ...) fprintf(tables_out,"\%s\n", string"
```

See also section 97f.

If a full parser is not needed, the lexing mechanism is not required. To satisfy the compiler and the linker, the lexer and other functions still have to be declared and defined, since these functions are referred to in the body of the parser. The details of these declarations can be found in the driver code.

```c
#define TeX__(string, ...) fprintf(tables_out,"\%s\n", string"
```

\(^1\)[..] 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.
We begin with a few macros to facilitate the output of tables in the format that \TeX\ can understand. As there is no perfect way to represent an array in \TeX\ a rather weak compromise was settled upon. Further explanation of this choice is given in the \TeX\ file that implements the \TeX\ parser for the bison input grammar.

\begin{verbatim}
#define tex_table_generic(table_name) table_name##.desc.preamble \"%newtable%s\%\n\";
table_name##.desc.separator \"%d\or\n\";
table_name##.desc.postamble \"%d\%\n\";
table_name##.desc.null_postamble \"%d\%\n\";
table_name##.desc.optimized_numeric \"\expandafter\def\csname␣%s\parsername\mespace␣%d\endcsname{%d}\%\n\";
table_name##.desc.prettify \true;
table_name##.desc.formatter \Lambda;
table_name##.desc.cleanup \Lambda;
#define tex_table(table_name) tex_table_generic(table_name);
table_name##.desc.name \#table_name;
\end{verbatim}

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 \textit{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.

\begin{verbatim}
struct const_d { char *format; char *name; 
};
\end{verbatim}

\begin{verbatim}
#define register_const_d(c_name) struct const_d c_name##.desc;
#define register const_d
\end{verbatim}

\begin{verbatim}
#define register_const_d(c_name) bool output##c_name:1;
#define register const_d
\end{verbatim}

\begin{verbatim}
#define register_const_d(c_name) .output##c_name \0;
#define register const_d
\end{verbatim}

\begin{verbatim}
#define register const_d(c_name)
if (output_desc.output##c_name) {
    const_out(tables_out, c_name##.desc, c_name)
    any_constants \0;
}
#define register const_d
\end{verbatim}
PLINT COMMON ROUTINES

497 414

{Constant names 99a}
#define _register_const_d
    if (any_constants);  ▶ this is merely a placeholder statement ◀
}
This code is used in section 98f.

99a Constants are very driver specific, so to make CWEAVE happy ...
{Constant names 99a}
This code is used in sections 98c, 98d, 98e, and 98g.

99b 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);

99c 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 99c}
This code is used in section 91b.

99d Error codes

{Global variables and types 94c} + =
    enum err_codes {
        {Error codes 99e} LAST_ERROR
    };

99e {Error codes 99e} =
    NO_MEMORY, BAD_STRING, BAD_MIX_FORMAT,
See also section 114a.
This code is used in section 99d.

99f 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 MAXPRETTY_LINE 100
{Auxiliary function declarations 99f} =
    char *mix_string(char *format, ...);
This code is used in section 97f.

99g {Auxiliary function definitions 99g} =
    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 = strlen(format, MAXPRETTY_LINE * 5);
if (size \geq MAX_PRETTY_LINE * 5) {
    fprintf(stderr, "\%s:\runaway\_string\n", __func__);
    exit(BAD\_STRING);
}
while ((formatp = strstr(formatp, "\%"))) {
    switch (formatp[1]) {
    case 's':
        length = strlen(va_arg(ap, char), MAX_PRETTY_LINE * 5);
        if (length \geq 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 \lor 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 91b.

100a 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.

\begin{verbatim}
enum output_mode {
    \langle Output modes 100b \rangle\_LAST\_OUT
};
\end{verbatim}

100b And to calm down CWAVE ...
TEX is the main output mode.

Establish defaults

\[
\text{enum output_mode mode} \leftarrow \text{TEX_OUT};
\]

This code is used in section 91b.

### 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

\[
\text{#include <unistd.h>}
\]

\[
\text{#include <getopt.h>}
\]

\[
\text{#include <string.h>}
\]

Local variable and type declarations

\[
\text{const char *usage} \leftarrow \text{"%s \[options\] output_file\n"};
\]

Command line processing variables

\[
\text{int c, option_index} \leftarrow 0;
\]

\[
\text{enum higher_options} \{
   \text{NON_OPTION} \leftarrow \text{FF}, \text{ LAST_HIGHER_OPTION}
\}
\]

\[
\text{static struct option long_options[]} \leftarrow \{
   \text{\{0, 0, 0, 0\}}
\}
\]

This code is used in section 91b.

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

\[
\text{opterr} \leftarrow 0; \quad \triangleright \text{we do our own error reporting} \leftarrow
\text{\textbf{forever} \{ c \leftarrow \text{getopt_long (argc, argv, (char [])\{'\', \{Short option list 102b\}}}, long_options, \&option_index \};
\]

\[
\text{if } (c = -1) \text{ break;}
\]

\[
\text{switch (c) } \{
   \text{case 0: } \triangleright \text{it is a flag, the name is kept in long_options[option_index].name, and the value can be found in long_options[option_index].val \leftarrow}
\]

\[
\text{break;}
\]

\[
\text{\{Cases affecting the whole program 102f\}};
\]

\[
\text{\{Cases involving specific modes 102g\}};
\]

\[
\text{case ' '?:}
\]

\[
\text{fprintf (stderr, "Unknown_option: \%s\}, see \textit{Usage} below\n\n", argv[optind - 1]);}
\]

\[
\text{fprintf (stderr, usage, argv[0]);}
\]

\[
\text{exit(1);}
\]

\[
\text{break;}
\]

\[
\text{case ' ':}
\]

\[
\text{fprintf (stderr, "Missing_argument_for \%s\}, argv[optind - 1]);}
\]

\[
\text{fprintf (stderr, usage, argv[0]);}
\]

\[
\text{exit(1);}
\]

\[
\text{break;}
\]

\[
\text{default:}
\]

\[
\text{printf (\textit{\"warning: \%s\}} is not yet implemented\n\n", c);}\n\]

\[
\text{}}\n\]

\[
\text{if (optind} \ni \text{argc} \{}
\]

\[
\text{fprintf (stderr, \textit{\"No output file specified\} \n\n});}
\]
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 91b.

102a ⟨Long options array 102a⟩ =
#define register_option(name, arg_flag, loc, val, exp) {name, arg_flag, loc, val},
    ⟨Options without shortcuts 102c⟩
    ⟨Options with shortcuts 102d⟩
    ⟨Options without arguments 94d⟩
#undef register_option
This code is used in section 101d.

102b 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 102b⟩ =
#define dd_optional_argument '','','
#define dd_required_argument '',''
#define dd_no_argument
#define register_option(name, arg_flag, loc, val, exp), val dd
    ⟨Options with shortcuts 102d⟩
#undef register_option
#undef dd_optional_argument
#undef dd_required_argument
#undef dd_no_argument
This code is used in section 101e.

102c 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 CWEAVE section below.

⟨Higher index options 102c⟩ =
#define register_option(name, arg_flag, loc, val, exp) val,
    ⟨Options without shortcuts 102e⟩
#undef register_option
This code is used in section 101d.

102d ⟨Options with shortcuts 102d⟩ =
This code is used in sections 102a and 102b.

102e ⟨Options without shortcuts 102e⟩ =
This code is used in sections 102a and 102c.

102f ⟨Cases affecting the whole program 102f⟩ =
This code is used in section 101e.

102g ⟨Cases involving specific modes 102g⟩ =
This code is used in section 101e.
103a 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.

103b 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.

\[
\text{\small{Parser table names 103b}} =
\begin{align*}
\text{\texttt{register\_table\_d(yytranslate)}} \\
\text{\texttt{register\_table\_d(yyr1)}} \\
\text{\texttt{register\_table\_d(yyr2)}} \\
\text{\texttt{register\_table\_d(yydefact)}} \\
\text{\texttt{register\_table\_d(yydefgoto)}} \\
\text{\texttt{register\_table\_d(yypact)}} \\
\text{\texttt{register\_table\_d(yypgoto)}} \\
\text{\texttt{register\_table\_d(yytable)}} \\
\text{\texttt{register\_table\_d(yycheck)}} \\
\text{\texttt{register\_table\_d(yytoknum)}} \\
\text{\texttt{register\_table\_d(yystos)}} \\
\text{\texttt{register\_table\_d(yytname)}} \\
\text{\texttt{register\_table\_d(yyprhs)}} \\
\text{\texttt{register\_table\_d(yyrhs)}}
\end{align*}
\]

See also section 104c.

103c 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’). The ‘new’ bison (starting with version 3.0) does not generate yyprhs and yyrhs or any other arrays that contain similar information, so we fake them here if such a crippled version of bison is used.

\[
\text{\small{Variables and types local to the parser 103c}} =
\begin{align*}
\text{\texttt{unsigned int yyrthree[YYNRULES + 1] \leftarrow \{0\};}} \\
\text{\texttt{unsigned int yyrhs[YYNRULES + 1] \leftarrow \{-1\};}} \\
\text{\texttt{unsigned int yyprhs[YYNRULES + 1] \leftarrow \{0\};}}
\end{align*}
\]

#ifndef BISON_IS_CRIPPLED
assert(YYNRULES + 1 = sizeof (yyprhs)/sizeof (yyprhs[0]));
{ int i, j;
  for (i = 1; i <= YYNRULES; i++) {
    for (j = 0; yyrhs[yyprhs[i] + j] \neq -1; j++) {
      assert(yyrhs[i] + j < sizeof (yyprhs));
      assert(j < yyr1[i]);
      if (\{This is an implicit term 104a\}) {
        \{Find the rule that defines it and set yyrthree 104b\}
      }
    }
  }
}
#endif

103d We populate this table below…

\[
\text{\small{Parser defaults 103d}} =
\begin{align*}
\text{\texttt{assert(YYNRULES + 1 = sizeof (yyprhs)/sizeof (yyprhs[0]));}} \\
\{ int i, j;
  for (i = 1; i <= YYNRULES; i++) {
    for (j = 0; yyrhs[yyprhs[i] + j] \neq -1; j++) {
      assert(yyrhs[i] + j < sizeof (yyprhs));
      assert(j < yyr1[i]);
      if (\{This is an implicit term 104a\}) {
        \{Find the rule that defines it and set yyrthree 104b\}
      }
    }
  }
} \\
\text{\texttt{unsigned int yyrthree[YYNRULES + 1] \leftarrow \{0\};}} \\
\text{\texttt{unsigned int yyrhs[YYNRULES + 1] \leftarrow \{-1\};}} \\
\text{\texttt{unsigned int yyprhs[YYNRULES + 1] \leftarrow \{0\};}}
\end{align*}
\]

See also sections 105b and 112a.
104a (This is an implicit term 104a) =

\( (\text{strlen}(\text{yytname}[\text{yyrhs}[i] + j])) > 1 \) \& \( (\text{yytname}[\text{yyrhs}[i] + j][0] = \text{'}Y\text{'}) \) \& \( (\text{yytname}[\text{yyrhs}[i] + j][1] = \text{'}@\text{')}) \\

This code is used in section 103d.

104b (Find the rule that defines it and set yythree 104b) =

```c
int rule_number;
for (rule_number = 1; rule_number < YYRULES; rule_number++)
    if (yy1[rule_number] = yyrhs[yyprhs[i] + j])
        yythree[rule_number] = j;
break;
assert(rule_number < YYRULES);
```

This code is used in section 103d.

104c ... and add its name to the list.

```c
register_table_d(yythree)
```

104d 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.

```c
if (output_desc.output_actions) {
    int i, j;
    fprintf(tables_out, "\%s", action_desc.preamble);
    if (!bare_actions) {
        yypact[0] = YYPACT_NINF;
        yypgoto[0] = -1;
        yydefgoto[0] = YYFINAL;
    }
    for (i = 1; i < sizeof(yy1)/sizeof(yy1[0]); i++)
        fprintf(tables_out, action_desc.act_setup, i, yy2[i] - 1);
    if (action_desc.print_rule)
        action_desc.print_rule(i);
    if (yy2[i] > 0)
        if (action_desc.action1)
            fprintf(tables_out, "\%s", action_desc.action1);
    }
    for (j = 2; j <= yy2[i]; j++)
        if (action_desc.actionn)
            fprintf(tables_out, action_desc.action_n, j);
    }
    if (!bare_actions) {
        yy1[i] = YYNTOKENS;
        yydefact[0] = i;
        yy2[i] = 0;
        yyparse(YYPARSE_PARAMETERS);
    }
    fprintf(tables_out, action_desc.act_suffix, i, yy2[i] - 1);
```
SPLINT ACTIONS

105a Constants
A generic list of constants to be used later in different contexts is defined below. As before, the appropriate macro will be defined generically to do what is required with these names (for example, we can turn each name into a string for reporting purposes).

(Parser constants 105a) =
  .register const d (YYEMPTY)
  .register const d (YPACT_NINF)
  .register const d (YYEOF)
  .register const d (YYLAST)
  .register const d (YYTOKENTS)
  .register const d (YYRULES)
  .register const d (YYSTATES)
  .register const d (YYFINAL)
This code is used in section 110b.

105b 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 103c) + =
  char *token_format_char = Λ;
  char *token_format_affix = Λ;
  char *token_format_suffix = Λ;
  char *bootstrap_token_format = Λ;

(Parser specific options without shortcuts 105c) =
  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 107d and 111c.

105d (Handle parser output options 105d) =
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 111e and 112b.

106a ⟨Parser specific output descriptor fields 106a⟩ =

    bool output_tokens:1;

106b No tokens are output by default.

    ⟨Parser specific default outputs 106b⟩ =
    output_tokens ⇐ 0.

106c 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 106c⟩ =
    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, "\\"\bootstrapsmodetrue\n");
    fprintf(tables_out, "\%token_values_needed_to_bootstrap_the_parser\n");
    bootstrap_tokens(bootstrap_token_format);
    #endif

106d The size of the token name table is useful to determine, say, how many 'named' tokens the parser uses.

    ⟨Output parser constants 106d⟩ =
    fprintf(tables_out, "\\constset{YYTRANSLATESIZE}{%d}\%\n", (int)(sizeof(yytranslate)/sizeof(yytranslate[0])));
107a 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.

107b Token only mode

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

\begin{verbatim}
\langle Parser specific output modes 107b \rangle =
\token_only_out,
\end{verbatim}

See also sections 107g and 107i.

107c \langle Handle parser related output modes 107c \rangle =
\begin{verbatim}
case \text{TOKEN ONLY OUT}:
\langle Prepare token only output environment 107f \rangle
break;
\end{verbatim}

See also sections 107h and 108a.

107d \langle Parser specific options without shortcuts 105c \rangle + =
\begin{verbatim}
register_option("token-only-mode", no_argument, 0, \text{TOKEN ONLY MODE}, ")
\end{verbatim}

107e \langle Configure parser output modes 107e \rangle =
\begin{verbatim}
case \text{TOKEN ONLY MODE}:
mode \assign \text{TOKENONLY OUT};
break;
\end{verbatim}

107f \langle Prepare token only output environment 107f \rangle =
\begin{verbatim}
if (\text{!}}token_format_char) {
  token_format_char \assign \"{\%u}\";
}
if (\text{!}}token_format_affix) {
  token_format_affix \assign \"\%d, token_value: \%d\n\prettytoken\";
}
if (\text{!}}token_format_suffix) {
  token_format_suffix \assign \"\%s\"
}
output_desc.output_tokens \assign 1;
\end{verbatim}

This code is used in section 107c.

107g Generic output

Generic output is not programmed yet.

\begin{verbatim}
\langle Parser specific output modes 107b \rangle + = GENERIC_OUT,
\end{verbatim}

107h \langle Handle parser related output modes 107c \rangle + =
\begin{verbatim}
case GENERIC_OUT:
  printf("This mode is not supported yet\n");
  exit(0);
  break;
\end{verbatim}

107i \TeX output

The \TeX mode is the main reason for this software.

\begin{verbatim}
\langle Parser specific output modes 107b \rangle + = \text{TEX OUT},
\end{verbatim}
$\TeX$ OUTPUT

108a ⟨Handle parser related output modes 107c ⟩ + =

\hspace{1em} case $\TeX$\_OUT:
\hspace{3em} ⟨Set up $\TeX$ table output for parser tables 108b ⟩
\hspace{3em} ⟨Prepare $\TeX$ format for semantic action output 109b ⟩
\hspace{3em} ⟨Prepare $\TeX$ format for parser constants 110b ⟩
\hspace{3em} ⟨Prepare $\TeX$ format for parser tokens 111a ⟩
\hspace{1em} break;

108b 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 108b ⟩ =

\hspace{1em} #define _register_table_d(name) tex_table(name);
\hspace{1em} ⟨Table names 96c ⟩
\hspace{3em} #undef _register_table_d
\hspace{3em} yyr1_desc.name ⇐ "yyrone";
\hspace{3em} yyr2_desc.name ⇐ "yyrtwo";
\hspace{3em} See also section 109a.
\hspace{3em} This code is used in section 108a.

108c The memory allocated for the $yytname$ table is released at the end.

⟨ Helper functions declarations for for parser output 108c ⟩ =

\hspace{3em} void $yytname$\_cleanup(struct table_d ∗table);
\hspace{3em} int $yytname$\_formatter\_tex(FILE ∗stream, int index);

108d There are a number of helper functions to output complicated names in $\TeX$. The safest way seems to be to output those as sequences of ASCII codes to accommodate names like $\$end safely. $\TeX$'s ^^... convention is supported as well.

⟨ Helper functions for parser output 108d ⟩ =

\hspace{3em} void $yytname$\_cleanup(struct table_d ∗table)
\hspace{6em} { 
\hspace{9em} free(table->separator);
\hspace{9em} free(table->null);
\hspace{6em} }
\hspace{3em} int $yytname$\_formatter\_tex(FILE ∗stream, int index)
\hspace{6em} { 
\hspace{9em} char ∗token\_name ⇐ $yytname$[index];
\hspace{9em} unsigned char token;
\hspace{9em} int length ⇐ 0;
\hspace{9em} fprintf(stream, "\addname\n");
\hspace{9em} while ((token ⇐ ∗token\_name)) {
\hspace{12em} if (token < 0.40 ∨ token = 0.177) { unprintable characters ⬤
\hspace{15em} fprintf(stream, "^^%c", token < 0.100 ? (unsigned char)(token + 0.100) : (unsigned char)(token - 100));
\hspace{12em} length ⋅ 3;
\hspace{12em} } else { 
\hspace{15em} fprintf(stream, "\%c", token);
\hspace{15em} length + +;
\hspace{12em} }
\hspace{9em} token\_name + +;
\hspace{9em} }
\hspace{9em} fprintf(stream, "\n");
\hspace{9em} return length;
\hspace{6em} }
\hspace{3em} int $yytname$\_formatter(FILE ∗stream, int index)
\hspace{6em} { 
\hspace{9em} }
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 = "$impossible";
  }
  while ((token <= *token_name)) {
    length += fprintf(stream, "{%u}", (unsigned int) token);
    if (token < \s 40 \not token = \s 177) {
      too_creative = true;
    }
    token_name++;
  }
  fprintf(stream, "\%\%\#a\n\n", too_creative ? ".unprintable." : yytname[index]);
}
else {  // this is the last name
  token_name = yytname[−index];
  if (token_name == Λ) {
    token_name = "$impossible";
  }
  while ((token <= *token_name)) {
    length += fprintf(stream, "{%a}", (unsigned int) token);
    token_name++;
    if (token < \s 40 \not token = \s 177) {
      too_creative = true;
    }
  }
  fprintf(stream, "\%\%\%a\n\n\n", too_creative ? ".unprintable." : (yytname[−index] ? yytname[−index] : "end␣of␣array"));
}
return length;

See also section 110a.

109a  ⟨Set up \TeX\ table output for parser tables 108b⟩ + =

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.optimized_numeric = Λ;

yytname_desc.prettify = false;

yytname_desc.formatter = yytname_formatter;

yytname_desc.cleanup = Λ;

output_desc.output yytname = 1;

109b  ⟨Prepare \TeX\ format for semantic action output 109b⟩ =

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 doiisonaction\number_%\par\ns\name\endcsname\n"
This code is used in section 108a.

110a Grammar rules are listed in a readable form alongside the action code to make it possible to quickly find an appropriate action. The rules are not output if a crippled bison is used.

```c
#include <stdio.h>

#define YYSTYPE YYSTYPE

void print_rule(int n)
{
    fprintf(tables_out, "%s\n", yytoken[yylval]);
    if (yyval) {
        fprintf(tables_out, "%s", yytoken[yylval]);
        i++;
    }
}
```

110b \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.

```c
#include <stdio.h>

#define YYSTYPE YYSTYPE

void print_rule(int n)
{
    fprintf(tables_out, "%s\n", yytoken[yylval]);
    if (yyval) {
        fprintf(tables_out, "%s", yytoken[yylval]);
        i++;
    }
}
```
111a  Token definitions round off the \TeX output mode.
\begin{sloppypar}
\langle Prepare \TeX format for parser tokens \rangle =
\begin{verbatim}
c_name##_desc.name \Leftrightarrow \#c_name;
output_desc.output##c_name \Leftrightarrow 1;
\end{verbatim}
\end{sloppypar}
\begin{sloppypar}
\langle Parser constants \rangle
\begin{verbatim}
\textbf{#define} \_register_const_d
\textbf{YYPACT_NINF\_desc.name} \Leftrightarrow "YYPACTNINF";
\end{verbatim}
\end{sloppypar}

This code is used in section 108a.

111b  Command line options

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

111c  \langle Parser specific options without shortcuts \rangle + =
\begin{verbatim}
\textbf{register\_option}("help", \textbf{no\_argument}, 0, \textbf{LONG\_HELP}, ":")
\end{verbatim}

111d  \langle Shortcuts for command line options affecting parser output \rangle =
\begin{verbatim}
', 'h'
\end{verbatim}

111e  \langle Handle parser output options \rangle + =
\begin{verbatim}
case 'h': // short help
fprintf(stderr, "Usage:\%s\[options\] \output_file\n", argv[0]);
exit(0);
break; // should not be needed
\end{verbatim}
\begin{sloppypar}
\langle LONG\_HELP \rangle
\begin{verbatim}
fprintf(stderr,
"%s\[--mode=\TeX:options]\output_file\outputs,\tables\""\texttt{...}and\_constants,for\_a,\TeX,\parser\n", argv[0]);
exit(0);
break; // should not be needed
\end{verbatim}
\end{sloppypar}

111f  \langle Parser specific options with shortcuts \rangle =
\begin{verbatim}
\textbf{register\_option}("debug", \textbf{optional\_argument}, 0, 'b', ":")
\textbf{register\_option}("mode", \textbf{required\_argument}, 0, 'm', ":")
\textbf{register\_option}("table\_separator", \textbf{required\_argument}, 0, 'z', ":")
\textbf{register\_option}("format", \textbf{required\_argument}, 0, 'f', ":") // name?
\textbf{register\_option}("table", \textbf{required\_argument}, 0, 't', ":") // specific table
\textbf{register\_option}("constant", \textbf{required\_argument}, 0, 'c', ":") // specific constant
\textbf{register\_option}("name\_length", \textbf{required\_argument}, 0, 'l', ":") // change \texttt{MAX\_NAME\_LENGTH}
\textbf{register\_option}("token", \textbf{required\_argument}, 0, 'n', ":") // specific token
\textbf{register\_option}("run\_parse", \textbf{required\_argument}, 0, 'p', ":") // run the parser
\textbf{register\_option}("parse\_file", \textbf{required\_argument}, 0, 'i', ":") // input for the parser
\end{verbatim}

111g  The string below is a list of short options.
A few options can be discussed immediately.

\[\text{Variables and types local to the parser 103c} = \]
\[\text{char *table_separator} \leftarrow "\%s\";\]

\[\text{Handle parser output options 105d} = \]
\[\text{case }'\pi': \triangleright \text{output mode} \leftarrow \]
\[\text{switch }\text{(optarg[0])} \{\]
\[\text{case }'T': \text{case }'t':\]
\[\text{mode} \leftarrow \text{TEX_OUT};\]
\[\text{break};\]
\[\text{case }'b': \text{case }'B': \text{case }'g': \text{case }'G':\]
\[\text{mode} \leftarrow \text{GENERIC_OUT};\]
\[\text{break};\]
\[\text{default}:\]
\[\text{break};\]
\[\}\]
\[\text{case }'z': \text{table_separator} \leftarrow (\text{char *}) \text{malloc(}}(\text{strlen(\text{optarg}) + 1)}\times \text{sizeof(char)}\text{);}\]
\[\text{strcpy(\text{table_separator}, \text{optarg});}\]
\[\text{break};\]

\[\text{flex specific routines}\]

The output of the scanner automaton follows the steps similar to the ones taken during the parser output. The major difference is in the output of actions and constants.

\[\text{Tables}\]

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

\[\text{Scanner table names 112d} = \]
\[\text{.register_table_d(\text{yy.accept})}\]
\[\text{.register_table_d(\text{yy.ec})}\]
\[\text{.register_table_d(\text{yy.meta})}\]
\[\text{.register_table_d(\text{yy.base})}\]
\[\text{.register_table_d(\text{yy.def})}\]
\[\text{.register_table_d(\text{yy.next})}\]
\[\text{.register_table_d(\text{yy.chk})}\]

\[\text{Actions}\]

The scanner function, \text{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.

\[\text{Variables and types local to the scanner driver 112e} = \]
\[\text{int \text{max.yybase_entry} \leftarrow 0};\]
\[\text{int \text{max.yyaccept_entry} \leftarrow 0};\]
\[\text{int \text{max.yynext_entry} \leftarrow 0};\]
\[\text{int \text{max.yy.ec_entry} \leftarrow 0};\]

See also sections 114b and 118f.

\[\text{The ‘exotic’ scanner constants treated below are the constants used to control the scanner code itself. Unfortunately they are not given any names that can be used by the ‘driver’ to output them in a simple way.}\]

\[\text{Compute exotic scanner constants 112f} = \]
\[\text{for } (i \leftarrow 0; i < \text{sizeof(\text{yy_base})/sizeof(\text{yy_base[0]})}; i++) \{\]
\[\text{if } (\text{yy_base[i]} > \text{max.yybase_entry}) \{\]
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];
    }
}

113a (Output scanner actions 113a) =
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_base[2] = 0;
        yy_chk[0] = 2;
        yy_chk[max yybase_entry] = 1;
        yy_nxt[max yybase_entry] = 1;
        yy_nxt[0] = 1;
        fprintf (stderr, "max entry: %d\n", max yybase_entry);
    }
    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, "\n", yy_end_of_buffer, "\n\n", ".yylexeofaction");
        } else {
            fprintf (tables_out, "\n");
            if (not bare_actions) {
                ((struct yyguts_t *) fake_scanner)->yy_hold_char = 0;
                yy_accept[1] = i;
                if (i % 10 = 0) {
                    fprintf (stderr, ".\n");
                }
                yylex (fake_scanner);
            }
        }
    }
    fprintf (tables_out, action_desc.act_suffix, i);
}
fprintf (tables_out, "\n\n", yy_end_of_file_states, "\n");
"\#define YY_STATE_EOF(state) \( \text{(YY_END_OF_BUFFER} + \text{state} + 1) \)\);

if (max_eof_state = 0) {  \( \triangleright \) in case the user has not declared any states <
    max_eof_state \( \leftarrow \) YY_STATE_EOF(INITIAL);
}

for (; i \( \leq \) max_eof_state; i++) {
    fprintf(tables_out, action_desc.act_setup, i);
    if (!bare_actions) {
        fprintf(tables_out, "\n");
        (struct yyguts_t *) fake_scanner) \( \rightarrow \) yy_hold_char \( \leftarrow 0 \);
        yy_accept[1] \( \leftarrow i \);
        yylex(A, fake_scanner);
    }
    fprintf(tables_out, action_desc.act_suffix, i);
}

fprintf(tables_out, action_desc.postamble);
if (action_desc.cleanup)
    action_desc.cleanup(&action_desc);
}

\text{⟨Compute magic constants 114c⟩}
\text{⟨Output states 115b⟩};
fprintf(tables_out, "\constset{YYECMAGIC}{%d}\n", yy_ec_magic);
fprintf(tables_out, "\constset{YYMAXEOFSTATE}{%d}\n", max_eof_state);

114a \text{⟨Error codes 99e⟩} + = \text{BAD_SCANNER},

114b \text{⟨Variables and types local to the scanner driver 112e⟩} + = \text{int yy_ec_magic};

114c 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.

\text{⟨Compute magic constants 114c⟩} =
{
    int i, j;
    char fake_yytext[YY_MORE_ADJ + 1];
    yyscanner_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 \( \leftarrow 0 \);
    yy_set_bol(0);
    yyg->yytext_ptr \( \leftarrow \) fake_yytext;
    yyg->yy_c_buf_p \( \leftarrow \) yyg->yytext_ptr + 1 + YY_MORE_ADJ;
    fake_yytext[YY_MORE_ADJ] \( \leftarrow 0 \);  \( \triangleright \text{ yy_cp } \leftarrow 0; <
    yy_accept[0] \( \leftarrow 0 \);
    yy_base[0] \( \leftarrow 0 '\);
    for (i \( \leftarrow 0 \); i < sizeof (yy_chk)/sizeof (yy_chk[0]); i++) {
        yy_chk[i] \( \leftarrow 0 \);
    }
    for (i \( \leftarrow 0 \); i < sizeof (yy_nzt)/sizeof (yy_nzt[0]); i++) {
        yy_nzt[i] \( \leftarrow i \);
This code is used in section 113a.

### 115a State names

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

Note that parsing the scanner file is only partially helpful (even though the extended parser and scanner can recognize the `%x` option). All that can be done is output the state names but not their numerical values, since all such names are macros whose values are only known to the `flex` generated scanner.

```c
#define DefineState(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 115a) =

```c
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;
```

### 115b Output states 115b)

```c
{ 
    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" "\L\stateset{\%d}{\%d\%\n", INITIAL);
        while (current_state) {
            fprintf(tables_out, "\L\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 113a.

### Constants

```
{ Scanner constants 116a } =
  .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 117b.

### Output modes

The output modes are the same as those in the parser driver with some minor changes.

### Generic output

Generic output is not programmed yet.

```
{ Scanner specific output modes 116c } =
  GENERIC_OUT,
```

See also section 116c.

### Handle scanner output modes 116d

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

See also section 116d.

### T\TeX{} mode

The \TeX{} mode is the main focus of this software.

```
{ Scanner specific output modes 116c } +=
  TEX_OUT,
```

```
{ Handle scanner output modes 116d } +=
case TEX_OUT:
  (Set up \TeX{} format for scanner tables 116g)
  (Set up \TeX{} format for scanner actions 117a)
  (Prepare \TeX{} format for scanner constants 117b)
  break;
```

```
{ Set up \TeX{} format for scanner tables 116g } =
  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 116f.
\[\text{117a} \quad \text{Set up \TeX format for scanner actions 117a} = \]
\[
\begin{align*}
\text{if} \ (\text{optimize-actions}) \ {\{}
\quad \text{action-desc.preamble} & \leftarrow "\%n\%_\text{the_big_switch}\n\%
\quad "\text{\catcode'}\!\!=0\relax
\quad "\text{\def\yydoactionswitch#1{%%}
\quad "\text{\let\yydoactionswitch\yycontinue}
\quad "\text{\protect\ifcase#1\relax}
\quad "\text{\expandafter\protect\def\csname doflexaction%d\parsername\endcsname{%%}
\quad "\text{\expandafter\protect\let\csname doflexaction%d\ parsername\endcsname\protect\ifnum#1=1\relax}
\quad "\text{\protect\else}
\quad "\text{\expandafter\protect\fi}
\quad "\text{\expandafter\protect\yylextail}
\quad "\text{\protect\stashswitch{yydoactionswitch}%%}
\quad "\text{\expandafter\protect\catcode'}\!\!=12\relax
\quad "\text{\expandafter\protect\constset{%s}{%d}%%}
\quad "\text{\constset{YYMAXREALCHAR}{%ld}%%}
\quad "\text{\constset{YYBASEMAXENTRY}{%d}%%}
\quad "\text{\constset{YYNXTMAXENTRY}{%d}%%}
\quad "\text{\constset{YYMAXRULENO}{%d}%%}
\quad "\text{\constset{YYECMAXENTRY}{%d}%%}
\quad \}
\end{align*}
\]

This code is used in section 116f.

\[\text{117b} \quad \text{\TeX constant output is another place where the techniques described above are applied. A few names are handled separately, because they contain underscores.} \]

\[
\begin{align*}
\text{\#define\ register\ const\ d\ (c\ name) \ c\ name##\ desc\ format} & \leftarrow "/\text{constset}{%s}{%d}%%\n"
\quad "\text{c\ name##\ desc\ name} \leftarrow c\ name;
\quad \text{output-desc.output##c\ name} \leftarrow 1;
\end{align*}
\]

\[
\begin{align*}
\text{\#undef\ register\ const\ d}
\quad \text{YY_END_OF_BUFFER\_ desc\ name} & \leftarrow "\text{YYENDOFBUFFERCHAR}"
\quad \text{YY_NUM_RULES\_ desc\ name} \leftarrow "\text{YYNUMRULES}"
\quad \text{YY_END_OF_BUFFER\_ desc\ name} \leftarrow "\text{YYENDOFBUFFER}"
\end{align*}
\]

This code is used in section 116f.

\[\text{117c} \quad \text{Output exotic scanner constants 117c} = \]

\[
\begin{align*}
\text{\fprintf{\ tables\ out}{%ld}{%d}%%}
\quad \text{\fprintf{\ tables\ out}{%d}%%}
\quad \text{\fprintf{\ tables\ out}{%d}%%}
\quad \text{\fprintf{\ tables\ out}{%d}%%}
\quad \text{\fprintf{\ tables\ out}{%d}%%}
\quad \text{\fprintf{\ tables\ out}{%d}%%}
\end{align*}
\]
Command line options

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

Scanner specific options without shortcuts

register_option_("help", no_argument, 0, LONG_HELP, ")

Scanner specific options with shortcuts

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', "")
register_option_("constant", required_argument, 0, 'c', "")
register_option_("name-length", required_argument, 0, 'l', "")
register_option_("token", required_argument, 0, 't', "")
register_option_("run-scan", required_argument, 0, 'p', "")
register_option_("scan-file", required_argument, 0, 'i', ")

A few options can be immediately discussed.

Variables and types local to the scanner driver

int debug_level <= 0;
char *table_separator <= "%s";

Handle scanner output options

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;

break;

case 'z': table_separator <= (char *) malloc(strlen(optarg) + 1) * sizeof(char));
strcpy(table_separator, optarg);
break;
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’ viewpoints that did not fit elsewhere.

**121b On typographic convention**

It must seem quite perplexing to some readers that a manual focusing on *pretty-printing* shows such a wanton disregard for good typographic style. Haphazard choice of layouts to present programming constructs, random overabundance of fonts on almost every page are just a few of the many typographic sins and design guffaws so amply manifested in this opus. The author must take full responsibility for the lack of taste in this document and has only one argument in his defense: this is not merely a book for a good night read but a piece of technical documentation.

In many ways, the goal of this document is somewhat different from that of a well-written manual: to display the main features prominently and in logical order. After all, this is a package that is intended to help *write* such manuals so it must inevitably present some use cases that exhibit a variety of typographic styles achievable with SPLinT. Needless to say, *variety* and *consistency* seldom go hand in hand and it is the consistency that makes for a pretty page. One of the objectives has been to reveal a number of quite technical programming constructs so one should keep in mind that it is assumed that the reader will want to look up the input files to see how some (however ugly and esoteric) typographic effects have been achieved.

On the other hand, to quote a cliché, beauty is in the eyes of the beholder so what makes a book readable (or even beautiful) may well depend on the reader’s background. As an example, letterspacing as a typographic device is almost universally reviled in Western typography (aside from a few niche uses such as setting titles). In Russian, however (at least until recently), letterspacing has been routinely used for emphasis (or, as a Russian would put it, *emphasis*) in lieu of, say, *italics*. Before I hear any objections from typography purists, let me just say that this technique fits in perfectly with the way emphasis works in the Russian speech: the speaker slowly enunciates the sounds of each word (incidentally, emphasizing *emphasis* this way is a perfect example of the inevitable failure of any attempted letterspaced highlighting in most English texts). Letterspaced sentences are easy to find on a page, and they set a special reading rhythm, which is an added bonus in many cases, although their presence openly violates the ‘universally gray pages are a must’ dogma.
Selecting the license for this project involves more than the availability of the source code. \TeX, by its very nature is an interpreted\footnote{There are some exceptions to this, in the form of preloaded formats.} language, so it is not a matter of hiding anything from the reader or a potential programmer. 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. Short of getting into too many details of the so-called ‘open source licenses’ (other than GPL) and arguing with their advocates, let me simply express my lack of understanding of the arguments purporting 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 the software companies enjoy quite a few freedoms that we, as software consumers elect to afford them. Among such freedoms are the ability to renounce on any promises made to us and withdraw any guarantees that we might enjoy. 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. Free software, of course, does not provide any guarantees, either but ‘you get what you paid for’.

Another well spread industry 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 all too 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 (or worse, malicious) 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 license is a remarkable piece of ‘legal engineering’: a self-propagating contract 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 Imaginary Property) 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. Describing GPL as ‘viral’, creating dismissive acronyms such as FLOSS to refer to the free software, and spreading outright misinformation about GPL are just a few of the tactics employed by the software companies. Convince and confuse enough apathetic users and the protections granted by GPL are no longer visible.
123a Why not C++ or OOP in general

The choice of the language was mainly driven by aesthetic 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 informal 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 content, depending on the source.

On the 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 has not always been 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.

123b 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-documentated 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-documentated 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 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 (see below for a few similar criticisms of CWEB), 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.

123c 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.
Some CWEB idiosyncrasies

CWEB was among the first tools for literate programming intended for public use\textsuperscript{1}. By almost every measure it is a very successful design: the program mostly does what is intended, was used in a number of projects, and made a significant contribution to the practice of literate programming. It also gave rise to a multitude of similar software packages (see, for example, noweb by N. Ramsey, [Ra]), which proves the vitality of the approach taken by the authors of CWEB.

While the value of CWEB is not in dispute, it would be healthy to outline a few deficiencies\textsuperscript{2}) that became apparent after intensive (ab)use of this software. Before we proceed to list our criticisms, however, the author must make a disclaimer that not only most of the complaints below stem from trying to use CWEB outside of its intended field of application but such use has also been hampered by the author’s likely lack of familiarity with some of CWEB’s features.

The first (non)complaint that must be mentioned here is CWEB’s narrow focus on C-styled languages. The ‘grammar’ used to process the input is hard coded in CWEAVE, so any changes to it inevitably involve rewriting portions of the code and rebuilding CWEAVE. As C11 came to prominence, a few of its constructs have been left behind by CWEAVE. Among the most obvious of these are variadic macros and compound literals. The former is only a problem in CWEB’s @d style definitions (which are of questionable utility to begin with) while the lack of support for the latter may be somewhat amended by the use of @[...@] and @; constructs to manipulate CWEAVE’s perception of a given chunk as either an expr or a stmt. This last mechanism of syntactic markup is spartan but remarkably effective, although the code thus annotated tends to be hard to read in the editor (while resulting in just as beautifully typeset pages, nonetheless).

Granted, CWEB’s stated goal was to bring the technique of literate programming to C, C++, and related languages so the criticism above must be viewed in this context. Since CWEAVE outputs \TeX, one avenue for customizing its use to one’s needs is modifying the macros in cwebmac.tex. SPLinT took this route by rewriting a number of macros, ranging from simple operator displays (replacing, say, ‘=' with ‘<=’) to extensively customizing the indexing mechanism.

Unfortunately, this strategy could only take one thus far. The \TeX output produced by CWEAVE does not always avail itself to this approach readily. To begin with, while combining its ‘chunks’ into larger ones, CWEAVE dives in and out of the math mode unpredictably, so any macros trying to read their ‘environment’ must be ready to operate both inside and outside of the math mode and leave the proper mode behind when they are done. The situation is not helped by the fact that both the beginning and the end of the math mode in \TeX are marked by the same character ($\$, and it costs you, indeed) so ‘expandable’ macros are difficult to design.

Adding to these difficulties is CWEAVE’s facility to insert raw \TeX material in the middle of its input (the @t{}@ construct). While rather flexible, by default it puts all such user supplied \TeX fragments inside an \texttt{\Large\textbackslash hbox} which brings with it all the advantages, and, unfortunately, disadvantages of grouping, inability to introduce line breaks within the fragment, etc. There is, of course, an easy fix to most of these woes, outlined in CWEB’s manual: one can simply add \texttt{\Large\textbackslash hbox\texttt{@t}}\TeX stuff \texttt{@> which inserts \texttt{\Large\textbackslash hbox}\TeX stuff \texttt{@} into CWEAVE’s output. The cost of this hack (aside from looking and feeling rather ugly on the editor screen, not to mention disrupting the editor’s brace accounting) is a superfluous \texttt{\Large\textbackslash hbox} left behind before the \TeX stuff\textsuperscript{3}). The programmer’s provided \TeX code is unable to remove this box (at the macro level, i.e. in \TeX’s ‘mouth’ using D. Knuth’s terminology, one may still succeed with the \texttt{\Large\textbackslash lastbox} approach unless the \texttt{\Large\textbackslash hbox} was inserted in the main vertical mode) and it may result in an unwanted blank line, slow down the typesetting, etc. Most of these side-effects are easily treatable but it would still be nice if a true ‘asm style’ insertion of raw \TeX were possible\textsuperscript{3}.

In general, the lack of structure in CWEAVE’s generated \TeX seems to hinder even seemingly legitimate uses of cwebmac.tex macros. Even such a natural desire as to use a different type size for the C portions of the CWEB input is unexpectedly tricky to implement. Modifying the \texttt{\Large\textbackslash B} macro results in rather wasteful multiple reading of the tokens in the C portion, not to mention the absence of any guarantee that \texttt{\Large\textbackslash B} can find the end of its argument (the macros used by SPLinT look for the \texttt{\Large\textbackslash par} inserted by CWEAVE whenever \texttt{\Large\textbackslash B} is output

\textsuperscript{1)} The original WEB was designed to support DEK’s \TeX and METAFONT projects and was intended for PASCAL family languages.\textsuperscript{2)} Quirks would be a better term.\textsuperscript{3)} It must be said that in the majority of cases such side-effects are indeed desirable, and save the programmer some typing but it seems that the @t facility was not well thought out in its entirety.
but an unsuspecting programmer may disrupt this mechanism by inserting \texttt{\textbackslash h\{is, her\} own \textbackslash par} using the \texttt{\textbackslash @t} facility with the aim to put a picture in the middle of the code, for example.

The authors of \texttt{CWEB} understood the importance of the cross-referencing facilities provided by their program. There are several control sequences dedicated to indexing alone (which itself has been the subject of criticism aimed at \texttt{CWEB}). The indexing mechanism addresses a number of important needs, although it does not seem to be as flexible as required in some instances. For example, most book indices are split into sections according to the first letter of the indexed word to make it easier to find the desired term in the index (or to establish that it is not indexed). Doing so in \texttt{CWEB} requires some macro acrobatics, to say the least.

Also absent is a facility to explicitly inhibit the indexing of a specific word (in \texttt{CWEAVE}'s own source, the references for \texttt{pp} fill up several lines in the index) or limit it to definitions only (as \texttt{CWEAVE} automatically does for single letter identifiers). This too, can be fixed by writing new indexing macros.

Finally, the index is created at the point of \texttt{CWEAVE} invocation, before any pagination information becomes available. It is therefore difficult to implement any page oriented referencing scheme. Instead, the index and all the other cross referencing facilities are tied to section numbers. In the vast majority of cases, this is a superior scheme: sections tend to be short and the index creation is fast. Sometimes, however, it is useful to provide the page information to the index macros. Unfortunately, after the index creation is completed, any connection between the words in the original document and those in the index is lost.

The indexing macros in \texttt{SPLINT} that deal with \texttt{bison} and \texttt{flex} code have the advantage of being able to use the page numbers so a better indexing scheme is possible. The section numbering approach taken by \texttt{SPLINT} approximately follows that of \texttt{noweb}: the section reference consists of two parts, where the first is the page number the section starts on, and the second is the index of the section within the page. Within the page, sections are indexed by (sequences of) letters of the alphabet (a...z and, in the rarest of cases, aa...zz and so on). Numbering the sections themselves is not terribly complicated. Where it gets interesting, is during the production of the index entries based on this system. When the sections are short, just referencing the section where the term appears works well. Sometimes, however, a section is split between two or more pages, in which case the indexing macros provide a compromise: whenever the term appears on a page different from the one on which the corresponding section starts, the index entry for that term uses the page number instead of the section reference. The difference between the two is easy to see, since the page number does not have any alphabetic characters in it.

This is not exactly how the references work in \texttt{noweb}, since \texttt{noweb} ignores the \TeX{} portion of the section and only references the code chunks but it is similar in spirit. Other conveniences, also borrowed from \texttt{noweb}, are the references in the margins that allow the reader to jump from one chunk to the next whenever the code chunk is composed of several sections. All of these changes are implemented with macros only, so, for example, the finer section number/page number scheme is not available for the index entries produced by \texttt{CWEAVE} itself. In the case of \texttt{CWEB} generated entries only the section numbers are used (which in most cases do provide the correct page number as part of the reference, however).

To conclude this Festivus\footnote{Yes, I am old enough to know what this means.} style airing of grievances, let me state once again that \texttt{CWEB} is a remarkable tool, and incredibly useful as it is, although it does test one’s ability to write sophisticated \TeX{} if subtle effects are desired. Finally, when all else fails, one is free to modify \texttt{CWEB} itself or even write one’s own literate programming tool.

**125a Why not GitHub\textsuperscript{©}, Bitbucket\textsuperscript{©}, etc**

Git is fantastic software that is used extensively in the development of \texttt{SPLINT}. The distribution archive is a Git repository. The use of centralized services such as GitHub\textsuperscript{©} \footnote{A recent acquisition of GitHub\textsuperscript{©} by a company that not so long ago used expletives to refer to the free software movement only strengthens my suspicions, although everyone is welcome to draw their own conclusions.}, 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\textsuperscript{©} 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 \texttt{SPLINT} as an archive of a Git repository seems like an efficient way of being ready for an unexpected change.
127a This (experimental) section serves to aid in the testing and extension of SPLinT by formalizing a number of procedures in the form of a checklist. After having witnessed first hand the effectiveness of checklists in aviation, the author feels that a similar approach will be beneficial in programming, as well. Most of these tests can and should be automated but the applicable situations are rather rare so the automation has not been implemented yet.

**General checklist.**
- Have the checklists in this section been followed?
- Have *all* the examples been built and tested?
  - `make`: this would build the `ld` parser, as well as other parts, like `ssfo.pdf`, etc.
  - `symbols`
  - `xexpression` (both `make` and `make test`)
  - `expression` (both `make` and `make test`)
  - once in a while it is useful to run a tool like `diffpdf` to check that the generated output does not change unexpectedly
  - `parsec` (not part of SPLinT)
- Have the changes been documented?
  - If any limitations have been removed, has this been reflected in the documentation, examples, such as `symbols.sty`?
  - If any new conditionals have been added, does `yydebug.sty` provide a way to check their status, if appropriate?
  - If any new script option has been added, has the script documentation been updated?
  - If a new process has been introduced, has it been reflected in any of the checklists in this section?

**Rewriting checklist.**
- Is the output of the new system identical?
  - once in a while it is useful to run a tool like `diffpdf` to check that the generated output does not change unexpectedly
  - has `diff` been used to check that `.gdx` and `.gdy` files produced are (nearly) identical?
  - has `diff` been used to check that `.sns` files produced by `symbols` and `xexpression` examples are (nearly) identical?
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.

∗∗∗


[Ho] Taco Hoekwater, \texttt{Lua\TeX} says goodbye to Pascal, TUGboat 30 (3), pp. 136–140, 2009—Euro\TeX 2009 Proceedings.


[La] The \texttt{13regex} package: regular expressions in \TeX, The \LaTeX3 Project.

[Pa] Vern Paxson et al., \textit{Lexical Analysis With Flex, for Flex 2.5.37}, July 2012.
\url{http://flex.sourceforge.net/manual/}.


[Sh] Alexander Shibakov, \textit{Parsers in \TeX and using \texttt{CWEB} for general pretty-printing}, TUGboat \textbf{35} (1), 2014, available as part of the documentation supplied with SPLINT.

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 the indexing to bison grammar terms, flex regular expression names and state names, as well as flex options, and \TeX{} control sequences encountered in bison actions. Definitions of tokens (via ⟨token⟩, ⟨nterm⟩ and ⟨type⟩ directives) are typeset in bold. 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 original syntax of bison allows the programmer to declare tokens such as { and } and the indexing macros honor this convention even though in a typeless environment such as the one the present typesetting parser is operating in such declarations are redundant. The indexing macros also record the use of such character tokens. 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:

\[
\text{left\_hand\_side} : \quad \text{term}_1 \text{ term}_2 \text{ term}_3 \quad \text{\textbackslash do\textbackslash something \Ye_1}
\]

inside the \TeX{} part of a \texttt{CWEB} section will generate several index entries, as well, including the entries for any material inside the action, mimicking \texttt{CWEB}’s behavior for the \texttt{inline} C (\ldots). Such entries (except for the references to the 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 only use the ‘string’ name of the token if available, thus avoiding referencing the same token twice).

\TeX{} control sequences are listed following the index of all bison and flex entries. The different sections of the index 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 \texttt{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 appears in the index (for example, \Ye_1 represents \textbackslash getfirst). The index entries are ordered alphabetically. The latter may not be entirely obvious in the cases when the ‘graphical
representation’ of the corresponding token manifests a significant departure from its string version (such as $Y_1$ instead of yy (1)). Incidentally, for the examples on this page (as well an example in the section about \TeX~ pretty-printing) both the ‘graphic’ as well as ‘text’ versions of the control sequence are indexed. It is instructive to verify that their location in the index corresponds to the ‘spelling’ of their visual representation (thus, $\pi_1$ appears under ‘p’). One should also be aware that the indexing of some terms has been suppressed, since they appear too often.
A LIST OF ALL SECTIONS

\{ A production 8a, 8d \} Cited in sections 3b and 8a. Used in sections 7c and 8c.
\{ A silly example 6a, 7a, 7c, 8c \} Used in section 8f.
\{ Add (empty) to the right hand side 37e \} Used in section 36d.
\{ Add a flex option 31e \} Used in section 31b.
\{ Add a \langle top \rangle directive 59h \} Used in section 59d.
\{ Add a \langle dprec \rangle directive to the right hand side 38b \} Used in section 36d.
\{ Add a \langle merge \rangle directive to the right hand side 38c \} Used in section 36d.
\{ Add a bare action 61d \} Used in section 60l.
\{ Add a character to a character class 66b \} Used in section 65f.
\{ Add a dot separator 85a \} Used in section 81d.
\{ Add a group of rules to section 2 61b \} Used in section 60l.
\{ Add a name to a list 59a \} Used in section 58e.
\{ Add a pointer option 59f \} Used in section 59d.
\{ Add a precedence directive to the right hand side 38a \} Used in section 36d.
\{ Add a predicate to the right hand side 37d \} Used in section 36d.
\{ Add a productions cluster 35a \} Used in section 34c.
\{ Add a range to a character class 66a \} Used in section 65f.
\{ Add a regular expression definition 59i \} Used in section 59d.
\{ Add a right hand side to a production 36a \} Used in section 34c.
\{ Add a rule to section 2 61a \} Used in section 60l.
\{ Add a start condition to a list 61i \} Used in section 61e.
\{ Add a symbol definition 34a \} Used in section 33g.
\{ Add a term to the right hand side 37b \} Used in section 36d.
\{ Add an action to the right hand side 37c \} Used in section 36d.
\{ Add an array option 59g \} Used in section 59d.
\{ Add an expression to a character class 66c \} Used in section 65f.
\{ Add an option to a list 59j \} Used in section 59d.
\{ Add an optional semicolon 36b \} Used in section 34c.
\{ Add closing brace to a predicate 52b \} Used in section 51e.
\{ Add closing brace to the braced code 52a \} Used in section 51e.
\{ Add options to section 1 58g \} Used in section 58c.
\{ Add start condition declarations 58f \} Used in section 58c.
\{ Add the scanned symbol to the current string 53b \} Used in section 43c.
\{ Additional options for flex input lexer 68a \} Used in section 67a.
\{ Assemble a flex input file 58a \} Used in section 57d.
\{ Assemble a flex section 1 file 58d \} Used in section 58c.
\{ Assign a code fragment to symbols 32a \} Used in section 31f.
\{ Attach a named suffix 85c \} Used in section 81d.
\{ Attach a productions cluster 34e \} Used in sections 30a and 34b.
\{ Attach a prologue declaration 30a \} Used in section 29e.
\{ Attach a qualifier 85d \} Used in section 81d.
\{ Attach an identifier 84a \} Used in sections 81d and 84b.
\{ Attach an integer 84c \} Used in section 81d.
\{ Attach integer suffix 85b \} Used in section 81d.
\{ Attach option name 82c \} Used in section 81d.
\{ Attach qualified suffixes 84g \} Used in section 81d.
\{ Attach qualifier to a name 84b \} Used in section 81d.
(Attach suffixes 84f) Used in sections 81d and 84g.
(Auxiliary code for flex lexer 80c) Used in section 67a.
(Auxiliary function declarations 99f) Used in section 97f.
(Auxiliary function definitions 99g) Used in section 91b.
Begin section 2, prepare to reread, or ignore braced code 73e Used in section 73b.
Begin the (top) directive 69c Used in section 69a.
Bison options 81b Used in section 81a.
Bootstrap parser C postamble 40d Used in section 26a.
Bootstrap token list 40f Used in section 40e.
Bootstrap token output 40e Used in section 40d.
Carry on 29c Used in sections 29b, 30b, 31a, 31b, 31f, 33b, 33d, 33d, 33e, 33g, 34c, 36b, 39h, and 39i.
Cases affecting the whole program 102f Used in section 101c.
Cases involving specific modes 102g Used in section 101e.
Clean up 93b Used in section 91b.
Collect all state definitions 87c Used in section 87a.
Collect state definitions for the flex lexer 80d Used in section 80c.
Collect state definitions for the grammar lexer 42c Used in section 41a.
Command line processing variables 101d Used in section 91b.
Common code for C preamble 93a
Complain about improper identifier characters 49a Used in section 48d.
Complain about unexpected end of file inside brackets 49b Used in section 48d.
Complain if not inside a definition, continue otherwise 71a Used in section 70b.
Complete a production 35b Used in section 34c.
Compose the full name 82a Used in section 81d.
Compute exotic scanner constants 112f
Compute magic constants 114c Used in section 113a.
Configure parser output modes 107e
Constant names 99a Used in sections 98c, 98d, 98e, and 98g.
Consume the brace and decrement the brace level 73d Used in section 73b.
Consume the brace and increment the brace level 73c Used in section 73b.
Copy the name and start a definition 69e Used in section 69a.
Copy the value 66f Used in sections 57d, 61j, 63e, 63g, 63k, 65a, 65b, 65i, and 66e.
Create a character class 65j Used in section 65f.
Create a complementary character class 65k Used in section 65f.
Create a lazy series match 64e Used in section 64d.
Create a list of start conditions 61f Used in section 61e.
Create a named reference 38e Used in section 36d.
Create a nonempty series match 64f Used in section 64d.
Create a possible single match 64g Used in section 64d.
Create a series of exact length 64i Used in sections 64c and 64d.
Create a series of minimal length 64i Used in sections 64b and 64d.
Create a series of specific length 64h Used in sections 64a and 64d.
Create a union of character classes 65h Used in section 65f.
Create a universal start condition 61g Used in section 61e.
Create an empty character class 66d Used in section 65f.
Create an empty named reference 38d Used in section 36d.
Create an empty section 1 58h Used in section 58e.
Create an empty start condition 61h Used in section 61e.
Decide if this is a comment 76d Used in section 73f.
Decide whether to start an action or skip whitespace inside a rule 75b Used in section 73f.
Declare a class 60e Used in section 59d.
Declare a prefix 60d Used in section 59d.
(Declare an extra type 60c) Used in section 59d.
(Declare the name for the tables 60g) Used in section 59d.
(Declare the name of a header 60f) Used in section 59d.
(Decode escaped characters 51b) Used in section 43e.
(Default outputs 94a, 97c, 98e) Used in section 93c.
(Define flex option list 31c) Used in section 31b.
(Define flex states 31d) Used in section 31b.
(Define symbol precedences 32f) Used in section 32c.
(Define symbol types 32e) Used in section 32c.
(Definition of symbol 39c) Used in sections 26a and 39b.
(Definitions for flex input lexer 68c) Used in section 67a.
(Determine if this is a parametric group or return a parenthesis 77a) Used in section 73f.
(Determine if this is extended syntax or return a parenthesis 76e) Used in section 73f.
(Disallow a repeated trailing context 63c) Used in section 63a.
(Do not support zero characters 47e) Used in section 43e.
(End the scan with an identifier 48c) Used in section 47f.
(Error codes 99c, 114a) Used in section 99d.
(Establish defaults 101a) Used in section 91b.
(Exclusive productions for flex section 1 parser 58c) Used in section 56a.
(Extend a flex string by a character 66g) Used in section 66c.
(Extend a series by a singleton 63) Used in section 63i.
(Fake start symbol for bootstrap grammar 29b) Used in section 26a.
(Fake start symbol for prologue grammar 29d) Used in section 27a.
(Fake start symbol for rules only grammar 29a) Used in section 25a.
(Find the rule that defines it and set yythird 104b) Used in section 103d.
(Finish a bison string 50b) Used in section 50a.
(Finish a tag 50f) Used in section 50c.
(Finish braced code 52d) Used in section 52c.
(Finish processing bracketed identifier 48f) Used in section 48d.
(Finish the line and/or action 75c) Used in section 73f.
(Finish the repeat pattern 78a) Used in section 77b.
(Generic table descriptor fields 95a) Used in section 94e.
(Global Declarations 28b) Used in section 28a.
(Global variables and types 94c, 94e, 96d, 97a, 98b, 99d) Used in section 97f.
(Grammar lexer C preamble 43c) Used in section 41a.
(Grammar lexer definitions 41b, 42a, 42b) Used in section 41a.
(Grammar lexer options 43d) Used in section 41a.
(Grammar lexer states 42d, 42e, 42f, 42g, 42h, 42i, 42a, 43a, 43b) Used in section 41b.
(Grammar parser C postamble 40c) Used in sections 25a, 27a, 27b, and 40d.
(Grammar parser C preamble 40b) Used in sections 25a, 26a, 27a, and 27b.
(Grammar parser bison options 27c) Used in sections 25a, 26a, 27a, and 27b.
(Grammar token regular expressions 43e) Used in section 41a.
(Handle end of file in the epilogue 53a) Used in section 52e.
(Handle parser output options 105d, 111e, 112b).
(Handle parser related output modes 107c, 107h, 108a).
(Handle scanner output modes 116d, 116f).
(Handle scanner output options 118d, 118g).
Helper functions declarations for for parser output 108c).
Helper functions for parser output 108d, 110a).
(Higher index options 102c) Used in section 101d.
(Insert local formatting 35e) Used in section 34c.
(Lexer C preamble 88b) Used in section 87a.
Lexer definitions 87b Used in section 87a.
Lexer options 88c Used in section 87a.
Lexer states 88a Used in section 87b.
List of symbols 33d Cited in section 33c. Used in sections 26a and 33c.
Local variable and type declarations 93c, 94b, 97d, 98c, 100a, 101c Used in section 91b.
Long options array 102a Used in section 101d.
Make a «name» into a start condition 62b Used in section 61e.
Make an empty option list 60a Used in section 59d.
Make an empty regular expression string 66h Used in section 66e.
Make an empty right hand side 37a Used in section 36d.
Match (almost) any character 64k Used in section 64d.
Match a PREVCL 65b Used in section 64d.
Match a character class 65a Used in section 64d.
Match a regular expression at the end of the line 63d Used in section 63a.
Match a regular expression with a trailing context 63b Used in section 63a.
Match a rule at the beginning of the line 62g Used in section 62f.
Match a sequence of alternatives 63f Used in section 63a.
Match a sequence of singletons 63g Used in section 63a.
Match a series of exact length 64c Used in section 63i.
Match a series of minimal length 64b Used in section 63i.
Match a series of specific length 64a Used in section 63i.
Match a singleton 63k Used in section 63i.
Match a specific character 65e Used in section 64d.
Match a string 65c Used in section 64d.
Match an atom 65d Used in section 64d.
Match an end of file 62i Used in section 62f.
Match an ordinary regular expression 63e Used in section 63a.
Match an ordinary rule 62h Used in section 62f.
Name parser C postamble 85g Used in section 81a.
Name parser C preamble 85f Used in section 81a.
Old ‘Add a right hand side to a production’ 35f
Old ‘Insert local formatting’ 35d
Options for flex input lexer 67c Used in section 67a.
Options for flex parser 55b Used in sections 55a, 56a, 56b, and 56c.
Options with shortcuts 102d Used in sections 102a and 102b.
Options without arguments 94d, 96e Used in section 102a.
Options without shortcuts 102e Used in sections 102a and 102c.
Outer definitions 92b, 101b Used in section 97f.
Output a deprecated option 60i Used in section 59d.
Output a non-parametric option 60i Used in section 59d.
Output a regular expression 62e Used in section 62d.
Output action switch, if any 99c Used in section 91b.
Output all tables 96b Used in section 96a.
Output constants 98g Used in section 98f.
Output descriptor fields 93d, 97b, 98d Used in section 93c.
Output exotic scanner constants 117c
Output modes 100b Used in section 100a.
Output parser constants 106d
Output parser semantic actions 104d
Output parser tokens 106c
Output scanner actions 113a
Output section 2 60k Used in section 60j.
\{ Output states 115b \} Used in section 113a.
\{ Parser bootstrap productions 33a, 33f, 33g, 39a, 39e \} Used in sections 26a and 32g.
\{ Parser common productions 31f, 32c, 32g, 33b, 33c, 33e, 39b, 40a \} Used in sections 25a, 27a, and 27b.
\{ Parser constants 105a \} Used in section 110b.
\{ Parser defaults 103d \}
\{ Parser full productions 28d \} Used in section 27b.
\{ Parser grammar productions 34b, 34c, 36d, 39d \} Used in sections 25a and 27b.
\{ Parser productions 81d \} Used in section 81a.
\{ Parser prologue productions 29e, 30b, 31a, 39f \} Used in sections 27a and 27b.
\{ Parser specific default outputs 106b \}
\{ Parser specific options with shortcuts 111f \}
\{ Parser specific options without shortcuts 105c, 107d, 111c \}
\{ Parser specific output descriptor fields 106a \}
\{ Parser specific output modes 107b, 107g, 107i \}
\{ Parser table names 103b, 104c \}
\{ Patterns for \texttt{flex} lexer 69a, 69f, 70b, 71b, 73a, 73b, 73f, 77b, 78b, 79b, 79d, 80b \} Used in section 67a.
\{ Perform output 96a, 98f \} Used in section 91b.
\{ Pop state if code braces match 70a \} Used in section 69f.
\{ Possibly complain about a bad directive 47a \} Used in section 44a.
\{ Postamble for \texttt{flex} input lexer 68d \} Used in section 67a.
\{ Postamble for \texttt{flex} parser 66i \} Used in sections 55a, 56a, 56b, and 56c.
\{ Preamble for \texttt{flex} lexer 67b \} Used in section 67a.
\{ Preamble for the \texttt{flex} parser 57c \} Used in sections 55a, 56a, 56b, and 56c.
\{ Prepare \TeX{} format for parser constants 110b \} Used in section 108a.
\{ Prepare \TeX{} format for parser tokens 111a \} Used in section 108a.
\{ Prepare \TeX{} format for scanner constants 117b \} Used in section 116f.
\{ Prepare \TeX{} format for semantic action output 109b \} Used in section 108a.
\{ Prepare a \texttt{bison} stack name 83k \} Used in section 81d.
\{ Prepare a \texttt{tag} 32h \} Used in sections 32c, 33e, and 33f.
\{ Prepare a generic one parametric option 30d \} Used in sections 30b and 31f.
\{ Prepare a state declaration 58j \} Used in section 58e.
\{ Prepare a string for use 39k \} Used in sections 39e and 39f.
\{ Prepare an exclusive state declaration 58k \} Used in section 58e.
\{ Prepare an identifier 47b \} Used in section 44a.
\{ Prepare one parametric option 30c \} Used in section 30b.
\{ Prepare the left hand side 39j \} Used in section 39d.
\{ Prepare to match a trailing context 63h \} Used in section 63a.
\{ Prepare to process a meta-identifier 90b \} Used in section 88f.
\{ Prepare to process an identifier 90a \} Used in section 88f.
\{ Prepare token only output environment 107f \} Used in section 107c.
\{ Prepare union definition 32d \} Used in section 32c.
\{ Process a bad character 46b \} Used in section 44a.
\{ Process a character after an identifier 48b \} Used in section 47f.
\{ Process a colon after an identifier 48a \} Used in section 47f.
\{ Process a comment inside a pattern 75a \} Used in section 73f.
\{ Process a deferred action 74d \} Used in section 73f.
\{ Process a named expression after checking for whitespace at the end 76c \} Used in section 73f.
\{ Process a newline inside a braced group 79a \} Used in section 78b.
\{ Process a newline inside an action 79c \} Used in section 79b.
\{ Process a repeat pattern 74b \} Used in section 73f.
\{ Process an escaped sequence 80a \} Used in section 79d.
\{ Process braced code in the middle of section 2 74c \} Used in section 73f.
\{ Process bracketed identifier 48e \} Used in section 48d.
\{ Process command line options 101e \} Used in section 91b.
\{ Process quoted name 84d \} Used in section 81d.
\{ Process quoted option 84e \} Used in section 81d.
\{ Process the bracketed part of an identifier 47g \} Used in section 47f.
\{ Productions for flex parser 57d, 58b \} Used in section 55a.
\{ Productions for flex section 1 parser 58e, 59d \} Used in sections 56a and 58b.
\{ Productions for flex section 2 parser 60f, 61e, 62c \} Used in sections 56b and 58b.
\{ Raise nesting level 51a \} Used in section 50e.
\{ React to a bad character 90c \} Used in section 88f.
\{ Record the name of the output file 60b \} Used in section 59d.
\{ Regular expressions 88d \} Used in section 87a.
\{ Report an error and quit 62j \} Used in section 62f.
\{ Report an error compiling a start condition list 62a \} Used in section 61e.
\{ Report an error in namelist and quit 59c \} Used in section 58e.
\{ Report an error in section 1 and quit 58i \} Used in section 58e.
\{ Rest of line 86, 8e \} Cited in section 8a. Used in sections 7c and 8c.
\{ Return a bracketed identifier 49d \} Used in section 49c.
\{ Return an escaped character 50d \} Used in section 50c.
\{ Return lexer and parser parameters 46f \} Used in section 44a.
\{ Return lexer parameters 46d \} Used in section 44a.
\{ Return parser parameters 46g \} Used in section 44a.
\{ Rules for flex regular expressions 62f, 63a, 63i, 64d, 65f, 66e \} Used in sections 56c and 62c.
\{ Scan bison directives 44a \} Used in section 43e.
\{ Scan flex directives and options 46a \} Used in section 43e.
\{ Scan a C comment 49f \} Used in section 43e.
\{ Scan a bison string 50a \} Used in section 43e.
\{ Scan a yacc comment 49c \} Used in section 43e.
\{ Scan a character literal 50c \} Used in section 43e.
\{ Scan a line comment 49g \} Used in section 43e.
\{ Scan a tag 50e \} Used in section 43e.
\{ Scan after an identifier, check whether a colon is next 47f \} Used in section 43e.
\{ Scan bracketed identifiers 48d, 49c \} Used in section 43e.
\{ Scan code in braces 51e \} Used in section 43e.
\{ Scan grammar white space 43f \} Used in section 43e.
\{ Scan identifiers 88f \} Used in section 88d.
\{ Scan prologue 52c \} Used in section 43e.
\{ Scan the epilogue 52e \} Used in section 43e.
\{ Scan user-code characters and strings 51c \} Used in section 43e.
\{ Scan white space 88e \} Used in section 88d.
\{ Scanner constants 116a \} Used in section 117b.
\{ Scanner specific options with shortcuts 118c \} Used in section 118b.
\{ Scanner specific output modes 116c, 116e \} Used in section 116c.
\{ Scanner table names 112d \} Used in section 112d.
\{ Scanner variables and types for C preamble 115a \} Used in section 115a.
\{ Set \{ debug \} flag 46c \} Used in section 44a.
\{ Set \{ locations \} flag 46e \} Used in section 44a.
\{ Set \{ pure-parser \} flag 46h \} Used in section 44a.
\{ Set up \TeX\ format for scanner actions 117a \} Used in section 116f.
\{ Set up \TeX\ format for scanner tables 116g \} Used in section 116f.
\{ Set up \TeX\ table output for parser tables 108b, 109a \} Used in section 108a.
Set lex_compat 72b  
Set postx_compat 72c  
Short option list 102b  
Shortcuts for command line options affecting parser output 111d  
Shortcuts for command line options affecting scanner output 118c  
Skip trailing whitespace, save the definition 70c  
Special flex section 2 parser productions 60j  
Special productions for regular expressions 62d  
Start a C code section 69b  
Start a namelist with a name 59b  
Start a list with a start condition name 61j  
Start an empty section 2 61c  
Start an options list 59c  
Start assembling prologue code 47d  
Start braced code in section 2 74a  
Start processing a character class 76b  
Start section 2 69d  
Start section 3 76a  
Start suffixes with a qualifier 85e  
Start the right hand side 35c  
Start with a ~ string 83h  
Start with a . string 83j  
Start with a < string 83e  
Start with a > string 83f  
Start with a $ string 83i  
Start with a named suffix 84h  
Start with a numeric suffix 84i  
Start with a production cluster 34d  
Start with a quoted string 83c  
Start with a tag 83b  
Start with an _ string 83g  
Start with an escaped character 83d  
Start with an identifier 83a  
State definitions for flex input lexer 68b  
Strings, comments etc. found in user code 51d  
Subtract a character class 35c  
Switch sections 47c  
Table names 96c  
This is an implicit term 104a  
Toggle option sense 72a  
Token and types declarations 81c  
Token definitions for flex input parser 56d, 57a, 57b  
Tokens and types for the grammar parser 28a, 28c, 32b, 36c  
Turn a «meta identifier» into a full name 82b  
Turn a basic character class into a character class 65i  
Turn a character into a term 39g  
Turn a qualifier into an identifier 83j  
Turn a string into a symbol 39l  
Turn an identifier into a symbol 39h  
Union of grammar parser types 40g  
Union of parser types 85h
 Variables and types local to the parser \(103c, 105b, 112a\)

 Variables and types local to the scanner driver \(112e, 114b, 118f\)

 Various output modes \(92a\)  Used in section 91b.

 C postamble 91b  Cited in section 91b.

 C preamble 97e, 97f

 flex options parser productions 31b  Used in sections 26a and 31a.

 \(bb.yy\) 26a  Cited in section 29d.

 \(bd.yy\) 27a

 \(bf.yy\) 27b

 \(bg.yy\) 25a

 \(ddp.yy\) 56a

 \(fil.ll\) 67a

 \(fip.yy\) 55a

 \(lo.ll\) 41a

 \(rap.yy\) 56b

 \(rep.yy\) 56c

 \(sill.y\) 8f

 \(small lexer.ll\) 87a

 \(small parser.yy\) 81a
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>3</td>
</tr>
<tr>
<td>CWEB and literate programming</td>
<td>3</td>
</tr>
<tr>
<td>Pretty (and not so pretty) printing</td>
<td>4</td>
</tr>
<tr>
<td>Parsing and parsers</td>
<td>5</td>
</tr>
<tr>
<td>Using the <code>bison</code> parser</td>
<td>6</td>
</tr>
<tr>
<td>On debugging</td>
<td>9</td>
</tr>
<tr>
<td>Terminology</td>
<td>11</td>
</tr>
<tr>
<td>Languages, scanners, parsers, and TeX</td>
<td>13</td>
</tr>
<tr>
<td>Arrays, stacks and the parser</td>
<td>14</td>
</tr>
<tr>
<td>TeX into tokens</td>
<td>15</td>
</tr>
<tr>
<td>Lexing in TeX</td>
<td>17</td>
</tr>
<tr>
<td>Inside semantic actions: switch statements and ‘functions’ in TeX</td>
<td>20</td>
</tr>
<tr>
<td>‘Optimization’</td>
<td>22</td>
</tr>
<tr>
<td>TeX with a different slant or do you C an escape?</td>
<td>22</td>
</tr>
<tr>
<td>The <code>bison</code> parser stack</td>
<td>25</td>
</tr>
<tr>
<td>Token declarations</td>
<td>28</td>
</tr>
<tr>
<td>Grammar productions</td>
<td>28</td>
</tr>
<tr>
<td>The scanner for <code>bison</code> syntax</td>
<td>41</td>
</tr>
<tr>
<td>Definitions and state declarations</td>
<td>41</td>
</tr>
<tr>
<td>Tokenizing with regular expressions</td>
<td>43</td>
</tr>
<tr>
<td>The <code>flex</code> parser stack</td>
<td>57</td>
</tr>
<tr>
<td>Token and state declarations for the <code>flex</code> input scanner</td>
<td>57</td>
</tr>
<tr>
<td>The grammar for <code>flex</code> input</td>
<td>57</td>
</tr>
<tr>
<td>The lexer for <code>flex</code> syntax</td>
<td>67</td>
</tr>
<tr>
<td>Regular expression and state definitions</td>
<td>68</td>
</tr>
<tr>
<td>Regular expressions for <code>flex</code> input scanner</td>
<td>69</td>
</tr>
<tr>
<td>The name parser</td>
<td>81</td>
</tr>
<tr>
<td>The name scanner</td>
<td>87</td>
</tr>
<tr>
<td>Forcing <code>bison</code> and <code>flex</code> to output TeX</td>
<td>91</td>
</tr>
<tr>
<td>Common routines</td>
<td>91</td>
</tr>
<tr>
<td>Error codes</td>
<td>99</td>
</tr>
<tr>
<td>Initial setup</td>
<td>100</td>
</tr>
<tr>
<td>Command line processing</td>
<td>101</td>
</tr>
<tr>
<td><code>bison</code> specific routines</td>
<td>103</td>
</tr>
<tr>
<td>Tables</td>
<td>103</td>
</tr>
<tr>
<td>Actions</td>
<td>104</td>
</tr>
<tr>
<td>Constants</td>
<td>105</td>
</tr>
<tr>
<td>Tokens</td>
<td>105</td>
</tr>
<tr>
<td>Output modes</td>
<td>107</td>
</tr>
<tr>
<td>Token only mode</td>
<td>107</td>
</tr>
<tr>
<td>Generic output</td>
<td>107</td>
</tr>
<tr>
<td>TeX output</td>
<td>107</td>
</tr>
<tr>
<td>Command line options</td>
<td>111</td>
</tr>
<tr>
<td><code>flex</code> specific routines</td>
<td>112</td>
</tr>
<tr>
<td>Tables</td>
<td>112</td>
</tr>
<tr>
<td>Actions</td>
<td>112</td>
</tr>
<tr>
<td>State names</td>
<td>115</td>
</tr>
<tr>
<td>Constants</td>
<td>116</td>
</tr>
<tr>
<td>Output modes</td>
<td>116</td>
</tr>
<tr>
<td>Generic output</td>
<td>116</td>
</tr>
<tr>
<td>Chapter</td>
<td>Page</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-------</td>
</tr>
<tr>
<td><strong>TeX mode</strong></td>
<td>485</td>
</tr>
<tr>
<td>Command line options</td>
<td>491</td>
</tr>
<tr>
<td><strong>Philosophy</strong></td>
<td>498</td>
</tr>
<tr>
<td>On typographic convention</td>
<td>499</td>
</tr>
<tr>
<td>Why GPL</td>
<td>500</td>
</tr>
<tr>
<td>Why not C++ or OOP in general</td>
<td>501</td>
</tr>
<tr>
<td>Why not *TEX</td>
<td>502</td>
</tr>
<tr>
<td>Why CWEB</td>
<td>503</td>
</tr>
<tr>
<td>Some CWEB idiosyncrasies</td>
<td>504</td>
</tr>
<tr>
<td>Why not GitHub©, Bitbucket©, etc</td>
<td>505</td>
</tr>
<tr>
<td><strong>Checklists</strong></td>
<td>506</td>
</tr>
<tr>
<td><strong>Bibliography</strong></td>
<td>507</td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td>508</td>
</tr>
</tbody>
</table>