SPLINT
reference

version 1.3.0

SPLINT

Alex Shibakov
January 30, 2024

0
Introduction

`SPLinT`[1]) (Simple Parsing and Lexing in TeX, or, following the great GNU tradition of creating recursive names, `SPLinT` Parses Languages in TeX) is a system (or rather a mélange of systems) designed to facilitate 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 `bison` and `flex` input file syntax have been developed, along with a macro collection that makes it possible to design and pretty print[2]) `bison` grammars and `flex` automata using `CWEB`. The `examples` directory contains a few other parsers designed to pretty print various languages (among them is `ld`, the language of the GNU linker).

2a **`CWEB` and literate programming**

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

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

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 `CTANGLE`. We refer an interested reader to [CWEB] for a detailed description of the syntax and use patterns of `CWEB`. The following is merely a brief overview of the approach.

Every `CWEB` document is split into *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

[1]) I was tempted to call the package `ParLALRgram` which stands for Parsing LALR Grammars or `PinT` for 'Parsing in TeX' but both sounded too generic. [2]) The term *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.

code or the *program* part). The C part of each [1]) 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 7b ⟩). 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 beween the fragments of the same section in the margins. This allows for quicker navigation between sections of the code and lets the reader to get a quick overview of what gets 'collected' in a given section.

The top level (`@**`) sections, introducing major portions of the code have also been given more prominent appearance. They display the chapter number using a large typeface and omit the marginal section reference. References to such sections are typeset as *cnnn* where *nnn* represents the chapter number. While such references obscure the page number, hopefully one keeps the number of chapters, as well as such references, small. This modification improves the appearance of the first chapter pages.

`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.

3a **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 visualizatin 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 [2]). 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

---

[1]) With the exception of the nameless `@c` (or `@p`) sections. [2]) The 'feature' so masterfully exploited by the International Obfuscated C Code Contest (IOCCC) participants.

different typographic devices (such as judicially chosen skips and indentation, or the choice of fonts) when (pretty) printing the code.

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 parcticular '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.

4a 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 specifiction.

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 'ridgidity' 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 grammar* [1] 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 requirement [2] 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.

pexp : (*pexp*) | *astring*

astring : ∘ | * *astring*

Informally, the language consists of 'strings of n *'s nested m parentheses deep'. After parsing such a string, one might be interested in the values of m and n.

---

[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 three states the parser may be in are 'start', 'parsing *pexp*' and 'parsing *astring*'. A quick glance at the grammar above shows that switching between the states is straightforward (we omit the discussion of the 'start' state for brevity): if the next symbol is `(`, parse the next *pexp*, otherwise, if the next symbol is `*`, parse *astring*. Finally, if the next symbol is `)` and we are parsing *pexp*, finish parsing it and look for the next input, otherwise, we are parsing *astring*, finish parsing it, make it a *pexp*, finish parsing a *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 *functions* resulting in what is known as a *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 parser specification 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:

pexp : `(` *pexp* `)` | *astring*

astring : ∘ | *astring* `*`

While the language is unchanged, so the algorithm above still works, the lookahead tokens are not *immediately* apparent upon looking at the productions. Some preprocessing must take place before one can decide on the choice of the parser states and the appropriate lookahead tokens. Such parser generating algorithms indeed exist and will produce 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 `bison` for the generation of the parsing algorithm, one must ensure that the grammar is LALR(1) [2]).

5a Using the `bison` parser

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

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

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

⟨ A silly example 5a ⟩ ≡ 6a

non_terminal :

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

See also sections 6a, 7a, and 7d.

This code is used in section 8a.

[1]) Both of these algorithms will use the parser stack more efficiently, effectively resolving the 'call stack depth' issue mentioned earlier. [2]) The newest versions of `bison` are capable of processing a *much* wider set of grammars, although `SPLinT` can only handle the `bison` output for LALR(1) parsers. [3]) 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 `bison` code so the snippet above can instead be presented without the distraction of `@=...@>`, looking more like the 'native' `bison` input

6a … if the syntax is correct. In case it is a bit off (note the missing colon after `whoops` below), 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. The layout of the original code is reproduced unchanged, including the braces and production separators (i.e. `|`) normally removed by the parser for presentation purposes.

⟨ A silly example 5a ⟩ + = 5a 7a

```
whoops
  term.1 term.2         { a ⇐ b; }
| term.3 other_term     { ϒ ⇐ ϒ1; }
| still more terms      { f(ϒ1); }
;
```

6b The TEX header that makes such output possible is quite plain. In the case of this document it begins as

```
\input limbo.sty
\input frontmatter.sty
\def\optimization{5}
\input yy.sty
```

[more code …]

The first two lines are presented here merely for completeness: there is no parsing-relevant code in them. The third line (`\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

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

[more code …]

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

```
\tokeneq {STRING}{{34}{115}{116}{114}{105}{110}{103}{34}}
\tokeneq {PERCENT_TOKEN}{{34}{37}{116}{111}{107}{101}{110}{34}}
```

[more code …]

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

7a To round off this short overview, I must mention a caveat associated with using the macros in this collection: while one of the greatest advantages of using CWEB is its ability to rearrange the code in a very flexible way, the parser will either give up or produce unintended output if this feature is abused while describing the grammar. For example, in the code below

⟨ A silly example 5a ⟩ += 6a 7d
  next_term :
    stuff ⟨ Rest of line 7c ⟩$a \Leftarrow f(x);$
  ⟨ A production 7b ⟩

7b the line titled ⟨ A production 7b ⟩ is intended to be a rule defined later. Notice that while it seems that the parser was able to recognize the first code fragment as a valid `bison` input, it misplaced the ⟨ Rest of line 7c ⟩, having erroneously assumed it to be a part of the action code for this grammar (later on we will go into the details of why it is necessary to collect all the non-verbatim output of CWEAVE, even that which contains no interesting C code; hint: it has something to do with money ($), also known as math and the way CWEAVE processes the 'gaps' between verbatim sections). The production line that follows did not fare as well: the parser gave up. There is simply no point in including such a small language fragment as a valid input for the grammar the parser uses to process the verbatim output.

⟨ A production 7b ⟩ = 7e
    `more stuff in this line {`$b \Leftarrow g(y);$`}`

See also section 7e.

This code is cited in sections 2a and 7b.

This code is used in sections 7a and 7d.

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

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

See also section 7f.

This code is cited in section 7b.

This code is used in sections 7a and 7d.

7d To correct this, one can provide a more complete grammar fragment to allow the parser to complete its task successfully. In some cases, this imposes too strict a constraint on the programmer. Instead, the parser that pretty prints `bison` grammars allows one to add *hidden context* to the code fragments above. The context is added inside `\vb` sections using CWEB's `@t...@>` facility. The CTANGLE output is not affected by this while the code above can now be typeset as:

⟨ A silly example 5a ⟩ += 7a
  next_term :
    stuff ⟨ *Rest of line* 7c ⟩ $a \Leftarrow f(x);$
  ⟨ A production 7b ⟩

7e ... even a single line can now be displayed properly.

⟨ A production 7b ⟩ += 7b
    more stuff in this line $b \Leftarrow g(y);$

7f With enough hidden context, even a small rule fragment can be typeset as intended. The 'action star' was inserted to reveal some of the context.

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

8a What makes all of this even more confusing is that `CTANGLE` will have no trouble outputting this as a(n almost, due to the intentionally bad `whoops` production above) valid `bison` file (as can be checked by looking into `sill.y`). The author happens to think that one should not fragment the software into pieces that are too small: `bison` is not C so it makes sense to write `bison` code differently. However, if the logic behind your code organization demands such fine fragmentation, hidden context provides you with a tool to show it off. A look inside the source of this document shows that adding hidden context can be a bit ugly so it is not recommended for routine use. The short example above is output in the file below.

⟨ `sill.y` 8a ⟩ =
⟨ A silly example 5a ⟩

8b **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 `\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, `\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 `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]) At the moment we are discussing debugging the output produced by `CWEAVE` when the included `bison` parser is used, *not* debugging parsers written with the help of this software: the latter topic is covered in more detail later on.

1
Terminology

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 `\yytable`) and adding the name of the parser (for example, `[main]`). To support this naming scheme, a number of macros work in unison to create and rename the 'data macros' accordingly [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 become a 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 refered 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 may 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.

2
Languages, scanners, parsers, and TEX

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

Computer generated (most likely)

In order to understand the parsing routines in this collection, it would help to gain some familiarity with the internals of the parsers produced by `bison` for its intended target: C. A person looking inside a parser delivered by `bison` would quickly discover that the parsing procedure itself (*yyparse*) occupies a rather small portion of the file. If (s)he were to further reduce the size of the file by removing all the preprocessor directives intended to anticipate every conceivable combination of the operating system, compiler, and C dialect, and various reporting and error logging functions it would become very clear that the most valuable product of `bison`'s labor is a collection of integer *tables* that control the actions of the parser routine. Moreover, the routine itself is an extremely concise and well-structured loop composed of **goto**'s and a number of numerical conditionals. If one 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 when compared to `bison`, was used to implement the lexer for this software suite. Lexing in TEX will be discussed in considerable detail later in this manual.

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

12a **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 *expansion* is likened to mastication, whereas TEX's main product, the typographic output is the result of its 'digestion' process. Not everything that goes through TEX's digestive tract ends up leaving a trace on the final page: a file full of `\relax`'s will produce no output, even though `\relax` is not a macro, and thus would have to be processed by TEX at the lowest level.

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

Is the `\count` array a good way to store tables in TEX? Probably not. The first factor is the *size* of this array: only 256 `\count` registers exist in a standard TEX (the actual number of such registers on a typical machine running TEX is significantly higher but this author is a great believer in standards, and to his knowledge, none of the standardization efforts in the TEX world has resulted in anything even close to the definitive masterpiece that is The TEXbook). The issue of size can be mitigated to some extent by using a number of other similar arrays used by TEX (`\catcode`, `\uccode`, `\dimen`, `\sfcode` and others can be used for this purpose as long as one takes care to restore the 'sane' values before 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 `\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 (`\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: `\token` registers, macros holding whole arrays, and associative arrays accessed through `\csname`...`\endcsname`. In the first two cases if care is taken to store such arrays in an appropriate form one can use TEX's `\ifcase` primitive to access individual elements. The trade-off is the speed of such access: it is *linear* in the size of the array for most operations, and worse than that for others, such as removing the last item of an array. Using clever ways of organizing such arrays, one can improve the linear access time to $O(\log n)$ by simply modifying the access macros but at the moment, a straightforward `\ifcase` is used after expanding a list macro or the contents of a `\token` n register in an *un*optimized parser. An *optimized* parser uses associative arrays.

The array discussion above is just as applicable to *stacks* (indeed, an array is the most common form of stack implementation). Since stacks pop up and disappear frequently (what else are stacks to do?), list macros are usually used to store them. The optimized parser uses a separate `\count` register to keep track of the top of the stack in the corresponding associative array [1]).

Let us now switch our attention to the code that implements the parser and scanner *functions*. If one has spent some time writing TEX macros of any sophistication (or any macros, for that matter) (s)he must be familiar with the general feeling of frustration and the desire to 'just call a function here and move on'. Macros [2]) produce *tokens*, however, and tokens must either expand to nothing or stay and be contributed to your input, or worse, be out of place and produce an error. One way to sustain a stream of execution with

---

1) Which means, unfortunately, that making such fully optimized parser *reentrant* would take an extraordinary amount of effort. Hence, if reentrancy is a requirement, stacks are better kept inside list macros. 2) Formally defined as '... special compile-time functions that consume and produce *syntax objects*' in [DHB].

macros is *tail recursion* (i.e. always expanding the *last token left standing*).

As we have already discussed, `bison`'s *yyparse*( ) is a well laid out loop organized as a sequence of **goto**'s (no reason to become religious about structured programming here). This fact, and the following well known trick, make C to TEX translation nearly straightforward. The macro TEXniques employed by the sample code below are further discussed elsewhere in this manual.

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

Given the code on the left (where **goto**'s are the only means of branching but can appear inside conditionals), one way to translate it into TEX is to define a set of macros (call them `\labelA`, `\labelAtail` and so forth for clarity) that end in `\next` (a common name for this purpose). Now, `\labelA` will implement the code that comes between `label A:` and `goto C;`, whereas `\labelAtail` is responsible for the code after `goto C;` and before `label B:` (provided no other **goto**'s intervene which can always be arranged). The conditional which precedes `goto C;` can now be written in TEX as presented on the right, where (condition) is an appropriate translation of the corresponding condition in the code being translated (usually, one of '=' or '≠'). Further details can be extracted from the TEX code that implements these functions where the corresponding C code is presented alongside the macros that mimic its functionality [1]). This concludes 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.

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

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

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

`SPLinT` comes with a customizeable input routine of moderate complexity (`\yyinput`) that classifies all TEX tokens into seven categories: 'normal' spaces (i.e. category 10 tokens, skipped by TEX's parameter scanning mechanism), 'explicit' spaces (includes the control sequences `\let` to ␣, as well as `\␣`), groups

[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`

(*avoid* using \bgroup and \egroup in your input but 'real', {...} groups are fine), active characters, normal characters (of all character categories that can appear in TEX input, including $, ^, #, a–Z, etc.), single letter control sequences, and multi-letter control sequences. Each of these categories can be processed separately to 'fine-tune' the input routine to the problem at hand. The input routine is not very fast, instead, flexibility was the main goal. Therefore, if speed is desirable, a customized input routine is a great place to start. As an example, a minimalistic \yyinputtrivial macro is included.

When \yyinput 'returns' by calling \yyreturn (which is a macro you design), your lexing routines have access to three registers: \yycp@, that holds the character value of the character just consumed by \yyinput, \yybyte, that most of the time holds the token just removed from the input, and \yybytepure, that (again, with very few exceptions) holds a 'normalized' version of the read character (i.e. a character of the same character code as \yycp@, and category 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 \vb{...}, the sequence \1 has nothing to do with the digit '1'). This mechanism makes it easy to examine the consumed token. It also forms the foundation of the 'hidden context' passing mechanism described later.

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

To examine every token in its path (including spaces that are easy to skip), the input routine uses one of the two well-known TEXnologies: \futurelet\next\examinenext or its equivalent \afterassignment\examinenext\let\next=␣. Recursively inserting one of these sequences, \yyinput can go through any list of tokens, as long as it knows where to stop (i.e. return an end of file character). The signal to stop is provided by the \yyeof 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 \yyeof can be eliminated if one is willing to invest the time in writing macros that juggle TEX's \token registers and only limit oneself to input from such registers (which is, aside from an obvious efficiency hit, a strain on TEX's memory, as you have to store multiple (3 in the general case) copies of your input to be able to back up when the lexer makes a wrong choice). 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 \vb{...} 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 {br,sp}ace. The \next sequence, seen by \examinenext contains a lot of information about the brace ahead: it knows its category code (left brace, so 1), its character code (in case there was, say a \catcode`\[=1␣ earlier) but not whether it is a 'real' brace (i.e. a character $\{_1$) or an implicit one (a \bgroup). There is no way to find that out until the control sequence 'launched' by \examinenext sees the token as a parameter.

If the next token is a 'real' brace, however, \examinenext's successor will never see the token itself: the braces are stripped by TEX's scanning mechanism. Even if it finds a \bgroup as the parameter, there is no guarantee that the actual input was not {\bgroup}. One way to handle this is by applying \string before consuming the next token. If prior to expanding \string care has been taken to set the \escapechar appropriately (remember, we know the character code of the next token in advance), as soon as one sees a character with \escapechar's character code, (s)he knows that an implicit brace has just been seen. One added complication to all this is that a very determined programmer can insert an *active* character (using, say, the \uccode mechanism) that has the *same* character code as the *brace* token that it has been \let to! Even setting this disturbing possibility aside, the \string mechanism (or, its cousin, \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 \string makes a sequence into characters, \csname ... \endcsname pair will make any sequence of characters into a control sequence so determining the end the character sequence produced by \string may prove impossible.

Huh ...

What we need is a backup mechanism: keeping a copy of the token sequence ahead, one can use \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 \iftrue{\else}\fi trick (and a generous sprinkling of \expandafters). Of course, some bookkeeping is needed to keep track of how deep inside the braced groups we are. For an implicit brace, more work is needed: read all the characters that \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 \outer control sequence. In other words, if the macro is used immediately after the assignment to the token register, it should always produce a correct count.

Needless to say, if such a general mechanism is desired, one has to look elsewhere. The added complications of treating spaces (TeX tends to ignore them most of the time) make this a torturous exercise in TeX's macro wizardry.

The included \yyinput has two ways of dealing with braces: strip them or view the whole group as a token. Pick one or write a different \yyinput. Spaces, implicit or explicit, are reported as a specially selected character code and consumed with a likeness of \afterassignment\moveon\let\next=␣. This behavior can be adjusted if needed.

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

15a 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* pass (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 be (in theory) replaced by an LALR parser, that parses *context-free* ones (or some subset thereof, which is still a super set of all regular languages). A common justification given for creating specialized lexers is efficiency and speed. The reality is somewhat more subtle. While we do care about the efficiency of parsing in TeX, having a specialized scanner is important for a number of different reasons.

The real advantage of having a dedicated scanner is the ease with which it can match incomplete inputs and back up. A parser can, of course, *recognize* any valid input that is also acceptable to a lexer, as well as *reject* any input that does not form a valid token. Between those two extremes, however, lies a whole realm of options that a traditional parser will have great difficulty exploring. Thus, to mention just one example, it is relatively easy to set up a DFA [1]) so that the *longest* matching input is accepted. The only straightforward way to do this with a traditional parser is to parse longer and longer inputs again and again. While this process can be optimized to a certain degree, the fact that a parser has a *stack* to maintain limits its ability to back up [2]).

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

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

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

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

---

[1]) Which stands for Deterministic Finite Automaton, a common (and mathematically unique) way of implementing a scanner for regular languages. Incidentally LALR mentioned above is short for Look Ahead Left to Right. [2]) It should be also mentioned that the fact that the lexing pass takes place prior to the parser consuming the resulting tokens allows one to process some grammars that are not context free. See, for example the *parser hack* used to process **typedef**s in C.

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

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

The 'hand-off' from the scanner to the parser is implemented through a pair of registers: `\yylval`, a token register containing the value of the returned token and `\yychar`, a `\count` register that contains the numerical value of the token to be returned.

Upon matching a token, the scanner passes one crucial piece of information to the 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 `\yybytepure` above). In cases when the tokens in `\yytext` are *not* $(c_{\rm ch}, c_{\rm cat})$ pairs, a few simple conventions are followed, some of which will be explained below. This sequence is provided merely for convenience and its typical use is to generate a key for an 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 CWEAVE 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 `CWEB`'s or any other literate programming tool's 'hypertext' structure) and a more diverse hidden context (or even arbitrary hidden text) mechanism, it is possible to use a strict parser to parse incomplete language fragments. For example, the productions that are needed to parse C's expressions form a complete subset of the grammar. If the grammar's 'start' symbol is changed to *expression* (instead of the *translation-unit* as it is in the full C grammar), a variety of incomplete C fragments can now be parsed and pretty-printed. Whenever such granularity is still too 'coarse', carefully supplied hidden context will give the pretty printer enough information to adequately process each fragment. A number of such *sub*-parsers can be tried on each fragment (this may sound computationally expensive, however, in practice, a carefully chosen hierarchy of parsers will finish the job rather quickly) until a correct parser produced the desired output (this approach is similar to, although not quite the same 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, however, 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

[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.

of tokens meant for the parser, it produces two [1]) hidden streams (called the `\yyformat` stream and the `\yystash` stream) and provides the parser with two strings (currently only strings of digits are used although arbitrary sequences of TEX tokens can be used as pointers) with the promise that *all the 'hints' between the beginning of the corresponding stream and the point labeled by the current stream pointer appeared among the characters up to and, possibly, including the ones matched as the current token.* The macros to extract the relevant parts of the streams (`\consumelistupto` and its cousins) are provided for the convenience of the parser designer.

What the parser does with these pointers and the information they provide, is up to the parser designer (the parsers for `flex` and `bison` syntax in this package use somewhat different strategies). The `\yystash` stream currently collects all the typesetting commands inserted by `CWEB` to be possibly used in displaying the action code in `bison` productions, for example. Because of this, it may appear in somewhat unexpected places, introducing spaces where the programmer did not neccessarily intend (such as at the beginning of the line, etc.). To mitigate this problem, the `\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 `symbols.sty`. The interested reader can consult the input routine macros in `yyinput.sty` for the details of the internal representation of the streams.

In the interest of full disclosure, it should be pointed 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 (in some cases multiple times), for *every* character in the input stream are rather complicated and therefore, slow. Whenever the use of such streams is not desired a simpler input routine can be written to speed up the process (see `\yyinputtrivial` for a working example of such macro).

Finally, while probably not directly related to the present discussion, this approach has one more interesting feature: after the parser is finished, the parser output and the streams exist 'statically', fully available for any last minute postprocessing or for debugging purposes, if necessary [2]). 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).

18a 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 `bison` get converted into TEX *macros* that drive the parser in *TEX*.

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

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

[1]) There would be no difficulty in splitting either of these streams into multiple 'substreams' by modifying the stream extraction macros accordingly. [2]) One may think of the parser output as an *executable abstract syntax tree* (AST or EAST if one likes cute abbreviations, or even eAST for an extra touch of modernity). This parser feature is used to implement the facility that allows easy referencing of productions in the section that implements the action code for one. See `yyunion.sty` and the source of this file (`splint.w`) for details. [3]) Incidentally, one of the names above is not used anymore. See the `cweb` directory after a successful build to find out which. Aren't footnotes great?!

the actions. The line

`\switchon\next\in\currentswitch`

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

You can give a case a readable label (say, `brackets`) and enclose this label in `\raw`...`\raw`. A word of caution: an '`a`' inside of `\raw`...`\raw` (which is most likely an $\mathtt{a}_{11}$ 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* if such characters can form tokens by themselves: 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`*anything*`\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 `\tracedfatrue`.

The functionality of the `\switchon` (as well as the `\switchonwithtype`, which is capable of some rudimentary type checking) macros has been implemented in a number of other macro packages (see [Fi] that discusses the well-known and widely used `\CASE` and `\FIND` macros). The macros in this collection have the additional property that the only assignments that persist after the `\switchon` completes are the ones performed by the user code inside the selected case.

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

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

This 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

[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. There is nothing special about non-letter characters, of course. To continue the `letter` example, placing a state named `let` *after* the `letter` one will make it essentially invisible to the switch macros for the reasons explained before this footnote.

as a symbolic name for a term in `bison`). Instead of going into the minutia of various flavors of `\yy`-macros, let me just mention that one can get by with only two 'idioms' and still be able to write parsers of arbitrary sophistication: `\yy(`n`)` can be treated as a token register containing the value of the n-th term of the rule's right hand side, $n > 0$. The left hand side of a production is accessed through `\yyval`. A convenient shortcut is `\yy0{`TEX material`}` which will expand (as in `\edef`) the 'TEX material' inside the braces and store the result in `\yyval` (note that this only works if a sequence of `0`s, possibly followed or preceeded by spaces are the only tokens between `\yy` and the opening braces; see the discussion of `\bb` macros below for a description of what happens in other cases). Thus, a simple way to concatenate the values of the first two production terms is `\yy0{\the\yy(1)\the\yy(2)}`. The included `bison` parser can also be used to provide support for 'symbolic names', analogous to `bison`'s `$[`name`]` but 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 in the part of the production preceeding the action). 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 makes it impossible to produce the *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 `\bb0`, i.e. the term implicitly inserted by `bison`. Be aware that there are no `\bb[·]` or `\bb(·)` versions of these macros, for obvious reasons [1]). A less obvious feature of this macro is its 'nonexpandable' nature. This means they cannot be used inside `\edef` (just like its `\yy`n counterpart, it makes several assignments which will not be executed by `\edef`). Thus, the most common use pattern is `\bb`n`{\toks`m`}` (where $n > 0$) with a subsequent expansion of `\toks`m [2]). Making these macros expandable is certainly possible but does not seem crucial for the intended limited use pattern.

Finally, the scripts included with `SPLinT` include a postprocessor (see the appropriate `Makefile` for further details) that allows the use of the 'native' `bison` term references (i.e. of the form `$`. . .) to access the value stack [3]). Utilizing the postprocessor allows any syntax for term references used by `bison` to be used inside `TeX`. . . C macros. In this case a typical idiom is `\the$[term_name]` to get the value of *term_name*. While storing a new value for the term (i.e. modifying the value stack) is also possible, it is not very straightforward and thus not recommended. This applies to all forms of term references discussed above.

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

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

---

[1]) The obvious reason is the mechanism used by `\yy[·]` and `\yy(·)` to make them expandable. These macros are basically references to the appropriate token registers. Since the `\bb` macros were designed for the environment where `\yylen` is unknown, establishing such references before the action is executed is not possible. A less obvious reason is the author's lazy approach.
[2]) Similar to how `\yy`n macros work, whenever $n > 0$, this pattern simply puts the contents of the braced group that follows in front of a (braced) single expansion of the appropriate value on the stack. If, as in the example above, the contents of the braced group are `\toks`m, this effectively stores the value of the braced group in the token register. In the case $n = 0$ the meaning is different: the stack value *corresponding to the implicit term* becomes the expanded (by `\edef`) contents of the braced group following `\bb`n. [3]) Incidentally, `bison` itself uses a preprocessor (`M4`) to turn its term references into valid C.

21a 'Optimization'

By default, the generated parser and scanner keep all of their tables in separate token registers. Each stack is kept in a single macro (this description is further complicated by the support for parser *namespaces* that exists even for unoptimized parsers but this subtlety will not be mentioned again—see the macros in the package for further details). Thus, every time a table is accessed, it has to be expanded making the table access latency linear in *the size of the table*. The same holds for stacks and the action 'switches', of course. While keeping the parser tables (which are immutable) in token registers does not have any better rationale than saving the control sequence memory (the most abundant memory in TEX), this way of storing *stacks* does have an advantage when multiple parsers get to play simultaneously. All one has to do to switch from one parser to another is to save the state by renaming the stack control sequences.

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.

21b *TEX* with a different *slant* or do you C an escape?

Some TEX productions below probably look like alien script. The authors of [Er] cite a number of reasons 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 `\def\texnspace{other}` at the beginning of this section can be uncommented and $^{\mathrm{nox}\bullet}(\,\Upsilon \leftarrow \langle\Upsilon_1\rangle\,)$ becomes `\noexpand\inmath{\yy0{\yy1{ }}}`. There is, however, more to this story. A look at the actual file will reveal that the line above was typed as

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

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

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

```
01          01
02 oexpand  02
```

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

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

Twice the escape!

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

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

3
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 detail further down.

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 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 directly 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 [1])) which can serve very diverse purposes: from collecting token declarations in the boostrapping pass to typesetting the grammar rules. This allows for a great deal of flexibility in where and when the parsing results are used. A clear divide between the parsing step and the typesetting step provides for better debugging facilities, as well as more reliable macro design.

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. The syntax tree generated by the parsers in the `bison` stack is not fully *faithful* in that it does not preserve every syntactic element from the original input. Thus, e.g. optional semicolons ($;_{\rm opt}$) never find their way into the tree and their original position is lost [2]).

[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 extravagnza is heading south so we shall stop here.
[2]) The opposite is true about the *whitespace* the parser sees (or *stash* as it is called in this document): all of it is carefully packaged into streams, as was described earlier.

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. A number of sections are empty, since the generated C is rather trivial.

⟨ bg.yy ch3 ⟩ =

....

⟨ Grammar parser C preamble 38i ⟩

....

⟨ Grammar parser bison options 26a ⟩

⟨**union**⟩ ⟨ *Union of grammar parser types* 39a ⟩

....

⟨ Grammar parser C postamble 38j ⟩

....

⟨ Tokens and types for the grammar parser 26b ⟩

⟨ Fake start symbol for rules only grammar 27d ⟩
⟨ Parser common productions 29c ⟩
⟨ Parser grammar productions 32a ⟩

24a Bootstrapping

Bootstrap parser is defined next. The purpose of the bootstrapping parser is to collect a minimal amount of information to 'spool up' the 'production' parsers. To understand its inner workings 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 precisely 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 `xxpression` example, specifically, the file `xxpression.w` in the `examples` directory for an example of using a different parser for this purpose). The reason it is necessary here is that a parser for an appropriate subset of the `bison` syntax is not yet available (indeed, *any* functional parser for a `bison` syntax subset would have to use the same scanner (unless you want to write a custom scanner for it), which would need to know how to output tokens, for which it would need a parser for a subset of `bison` syntax ... it is a 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 bootstrap parser is not strictly minimal in that it is also capable of parsing the ⟨`nterm`⟩ declarations. This ability is not utilized by the parsers in `SPLinT`, however (nor is the accompanying bootstrap lexer designed to output the ⟨`nterm`⟩ tokens), and was added for the scenarios other than bootstrapping.

The second, perhaps even more important function of the bootstrap process is to collect information about the scanner's states. The mechanism is slightly different from that for token definition gathering. While the token equivalences are collected purely in 'TEX mode', the bootstrap mode 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 (the *yytname* array) and are instead defined as ordinary (C) macros. The header file is the information the 'driver' file needs to output the state values for the use by the lexer.

Naturally, to accomplish their task, the lexer and the parser emplyed in state gathering need the state and token information, as well. Fortunately, the parser is a subset of the `flex` input parser that does not define any 'string' names for it tokens. Similarly, the lexer collects all the necessary tokens in the **INITIAL** state[1]).

To reiterate a point made in the middle of this section, the bootstrapping process described here is necessary to 'spool up' the `bison` and `flex` input parsers. A simpler procedure may be followed while designing other custom parsers where the programmer uses, say the full `bison` parser to collect information about the token equivalences (whether such information is needed to make the parser operational or just to facilitate the typesetting of the token names). By adding custom 'bootstrapping' macros to the ones defined in `yyunion.sty`, a number of different preprocessing tasks can be accomplished.

⟨ `bb.yy` 24a ⟩ =

....

⟨ Grammar parser C preamble 38i ⟩
 # define `BISON_BOOTSTRAP_MODE`

....

⟨ Grammar parser `bison` options 26a ⟩
⟨**union**⟩ ⟨ *Union of grammar parser types* 39a ⟩

....

⟨ Bootstrap parser C postamble 38k ⟩

....

⟨ Tokens and types for the grammar parser 26b ⟩

⟨ Fake start symbol for bootstrap grammar 27f ⟩
⟨ Parser bootstrap productions 30i ⟩

This code is cited in section 28b.

25a **Prologue and full parsers**

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

⟨ `bd.yy` 25a ⟩ =

....

⟨ Grammar parser C preamble 38i ⟩

....

⟨ Grammar parser `bison` options 26a ⟩
⟨**union**⟩ ⟨ *Union of grammar parser types* 39a ⟩

....

⟨ Grammar parser C postamble 38j ⟩

....

⟨ Tokens and types for the grammar parser 26b ⟩

⟨ Fake start symbol for prologue grammar 28b ⟩
⟨ Parser common productions 29c ⟩
⟨ Parser prologue productions 28d ⟩

25b 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. As a stopgap measure, the skeleton of a `flex` scanner is also parsed by this parser, as they have an almost identical structure. This is not a

---

1) An additional subtlety is the necessity to gracefully handle (and, in some cases, cause) the multiple possible *failures* for which the lexer redefines **enter** to fail immediately when attempting to switch states. Note that the bootstrap mode parser looks at sections other than those where the declarations reside and must fail quickly and quietly in such cases.

perfect arrangement, however, since it precludes one from putting the constructs that this parser does not recognize into the outline. To give an example, one cannot put `flex` specific options into such ‘skeleton’.

⟨ `bf.yy` 25b ⟩ =

....

⟨ Grammar parser C preamble 38i ⟩

....

⟨ Grammar parser `bison` options 26a ⟩

⟨union⟩ ⟨ *Union of grammar parser types* 39a ⟩

....

⟨ Grammar parser C postamble 38j ⟩

....

⟨ Tokens and types for the grammar parser 26b ⟩

⟨ Parser common productions 29c ⟩
⟨ Parser prologue productions 28d ⟩
⟨ Parser grammar productions 32a ⟩
⟨ Parser full productions 27b ⟩

26a 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

⟨define⟩ *lr.type* *canonical-lr*

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.

⟨ Grammar parser `bison` options 26a ⟩ =
⟨token table⟩ ⋆
⟨parse.trace⟩ ⋆ (set as **⟨debug⟩**)
⟨start⟩ *input*

This code is used in sections ch3, 24a, 25a, and 25b.

26b **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.

⟨ Tokens and types for the grammar parser 26b ⟩ = 30b

`"end of file"`$_m$	`«string»`	⟨`token`⟩	⟨`nterm`⟩
⟨`type`⟩	⟨`destructor`⟩	⟨`printer`⟩	⟨`left`⟩
⟨`right`⟩	⟨`nonassoc`⟩	⟨`precedence`⟩	⟨`prec`⟩
⟨`dprec`⟩	⟨`merge`⟩		

⟨ Global Declarations 27a ⟩

See also sections 30b and 35a.

This code is used in sections ch3, 24a, 25a, and 25b.

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

⟨ Global Declarations 27a ⟩ =

⟨code⟩	⟨default-prec⟩	⟨define⟩	⟨defines⟩
⟨error-verbose⟩	⟨expect⟩	⟨expect-rr⟩	⟨<flag>⟩
⟨file-prefix⟩	⟨glr-parser⟩	⟨initial-action⟩	⟨language⟩
⟨name-prefix⟩	⟨no-default-prec⟩	⟨no-lines⟩	⟨nondet. parser⟩
⟨output⟩	⟨require⟩	⟨skeleton⟩	⟨start⟩
⟨token-table⟩	⟨verbose⟩	⟨yacc⟩	$"\{...\}"_{\rm m}$
$"\%?\{...\}"_{\rm m}$	$"[identifier]"_{\rm m}$	**char**	epilogue
$"="_{\rm m}$	«identifier»	«identifier: »	⟨%⟩
$"\vert"_{\rm m}$	$"\%\{...\%\}"_{\rm m}$	$";"_{\rm m}$	<*tag*>
$"<*>"_{\rm m}$	$"<>"_{\rm m}$	**int**	⟨param⟩

This code is used in section 26b.

27b **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 `$[name]`. This parser is therefore used to generate the 'symbolic switch' to provide support for symbolic term names similar to the 'genuine' `bison`'s `$[...]` 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.

⟨ Parser full productions 27b ⟩ =

input : *prologue_declarations* ⟨%⟩ *grammar* $epilogue_{\rm opt}$ ⟨ Extract the grammar from a full file 27c ⟩

This code is used in section 25b.

27c ⟨ Extract the grammar from a full file 27c ⟩ =

```
\finishlist{\expandafter\yyfirstoftwo val Υ3 }    ▷ complete the list ◁
Ω\expandafter{\romannumeral 0
    \executelistat{\expandafter\yyfirstoftwo val Υ3 }{0}}
```

This code is used in section 27b.

27d Another subgrammar deals with the syntax of isolated `bison` rules. This is the most commonly used 'subparser' since a rules cluster is the most natural 'unit' to include in a `CWEB` file.

⟨ Fake start symbol for rules only grammar 27d ⟩ =

input : *grammar* $epilogue_{\rm opt}$ ⟨ Save the grammar 27e ⟩

This code is used in section ch3.

27e ⟨ Save the grammar 27e ⟩ =

```
\finishlist{\expandafter\yyfirstoftwo val Υ1 }    ▷ complete the list ◁
Ω\expandafter{\romannumeral 0
    \executelistat{\expandafter\yyfirstoftwo val Υ1 }{0}}
```

This code is used in section 27d.

27f 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.

⟨ Fake start symbol for bootstrap grammar 27f ⟩ =

input : *grammar_declarations* $\Omega = \Upsilon_1$

grammar_declarations :
  symbol_declaration $;_{\text{opt}}$ ⟨ Carry on 28a ⟩
  grammar_declarations symbol_declaration $;_{\text{opt}}$ $\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \text{val}\,\Upsilon_2\rangle$

$;_{\text{opt}}$: ◦ | ;

This code is used in section 24a.

28a The following is perhaps the most common action performed by the parser. It is done automatically by the parser code but this feature is undocumented so we supply an explicit action in each case.

⟨ Carry on 28a ⟩ =
  $\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1\rangle$

This code is used in sections 27f, 28g, 29c, 31a, 31b, 31d, 32b, 38c, and 38d.

28b 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 rules are not part of the official `bison` input grammar and are added to make the typesetting of 'file outlines' (e.g. ⟨ `bb.yy` 24a ⟩ above) possible.

⟨ Fake start symbol for prologue grammar 28b ⟩ =
  input : *prologue_declarations epilogue*$_{\text{opt}}$ ⟨ Save the declarations 28c ⟩
  prologue_declarations ⟨%⟩ ⟨%⟩ `epilogue` ⟨ Save the declarations 28c ⟩
  prologue_declarations ⟨%⟩ ⟨%⟩ ⟨ Save the declarations 28c ⟩

This code is used in section 25a.

28c ⟨ Save the declarations 28c ⟩ =
  `\finishlist{\expandafter\yyfirstoftwo` val Υ_1 `}` ▷ complete the list ◁
  Ω`\expandafter{\romannumeral0`
    `\executelistat{\expandafter\yyfirstoftwo` val Υ_1 `}{0}}`

This code is used in section 28b.

28d *Declarations: before the first* ⟨%⟩. We are now ready to deal with the specifics of the declarations themselves.

⟨ Parser prologue productions 28d ⟩ = 28g ▽
  prologue_declarations :
    ◦ ⟨ Start with an empty list of declarations 28e ⟩
    prologue_declarations prologue_declaration ⟨ Attach a prologue declaration 28f ⟩

See also sections 28g and 38g.

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

28e ⟨ Start with an empty list of declarations 28e ⟩ =
  `\initlist{\prologuedeclarationsprefix prologue_declarations}`
  $\Upsilon \leftarrow$ ⟨`{\prologuedeclarationsprefix prologue_declarations}{`$^{\text{nx}}\varnothing$`}`⟩
  def$_{\text{x}}$ `\prologuedeclarationsprefix{.\prologuedeclarationsprefix}`

This code is used in section 28d.

28f ⟨ Attach a prologue declaration 28f ⟩ =
  ⟨ Attach a productions cluster 32d ⟩

This code is used in section 28d.

28g Here is a list of most kinds of declarations that can appear in the prologue. The scanner returns the 'stream pointers' for all the keywords so the declaration 'structures' pass on those pointers to the grammar list. The original syntax has been left intact even though for the purposes of this parser some of the inline rules are unnecessary.

⟨ Parser prologue productions 28d ⟩ += △ 28d 38g ▽
  prologue_declaration :
    grammar_declaration ⟨ Carry on 28a ⟩

`%{...%}`	$\Upsilon \leftarrow \langle$ nx`\prologuecode` val $\Upsilon_1\rangle$
⟨⋆⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag` val $\Upsilon_1\rangle$
⟨`define`⟩ *variable value*	$\Upsilon \leftarrow \langle$ nx`\vardef{` val Υ_2 `}{` val Υ_3 `}`val $\Upsilon_1\rangle$
⟨`defines`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{defines}{ }`val $\Upsilon_1\rangle$
⟨`defines`⟩ «`string`»	$v_a \leftarrow$ ⟨ `defines` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`error-verbose`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{error verbose}{ }`val $\Upsilon_1\rangle$
⟨`expect`⟩ **int**	$v_a \leftarrow$ ⟨ `expect` ⟩⟨ Prepare a generic one parametric option 29b ⟩
⟨`expect-rr`⟩ **int**	$v_a \leftarrow$ ⟨ `expect-rr` ⟩⟨ Prepare a generic one parametric option 29b ⟩
⟨`file-prefix`⟩ «`string`»	$v_a \leftarrow$ ⟨ `file prefix` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`glr-parser`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{glr parser}{ }`val $\Upsilon_1\rangle$
⟨`initial-action`⟩ `{...}`	$\Upsilon \leftarrow \langle$ nx`\initaction` val $\Upsilon_2\rangle$
⟨`language`⟩ «`string`»	$v_a \leftarrow$ ⟨ `language` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`name-prefix`⟩ «`string`»	$v_a \leftarrow$ ⟨ `name prefix` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`no-lines`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{no lines}{ }`val $\Upsilon_1\rangle$
⟨`nondet. parser`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{nondet. parser}{ }`val $\Upsilon_1\rangle$
⟨`output`⟩ «`string`»	$v_a \leftarrow$ ⟨ `output` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`param`⟩ ⋄ *params*	... | $\Upsilon \leftarrow \langle$ nx`\paramdef{` val Υ_3 `}`val $\Upsilon_1\rangle$
⟨`require`⟩ «`string`»	$v_a \leftarrow$ ⟨ `require` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`skeleton`⟩ «`string`»	$v_a \leftarrow$ ⟨ `skeleton` ⟩⟨ Prepare one parametric option 29a ⟩
⟨`token-table`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{token table}{ }`val $\Upsilon_1\rangle$
⟨`verbose`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{verbose}{ }`val $\Upsilon_1\rangle$
⟨`yacc`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{yacc}{ }`val $\Upsilon_1\rangle$
`;`	$\Upsilon \leftarrow \langle$ nx$\varnothing\rangle$

params :

params `{...}`	$\Upsilon \leftarrow \langle$val Υ_1 nx`\braceit` val $\Upsilon_2\rangle$
`{...}`	$\Upsilon \leftarrow \langle$ nx`\braceit` val $\Upsilon_1\rangle$

29a 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.

prologue_declaration : ⟨`defines`⟩ «`string`» | ⟨`output`⟩ «`string`» | ⟨`require`⟩ «`string`»

The productions above are typical examples.

⟨ Prepare one parametric option 29a ⟩ =
$\Upsilon \leftarrow \langle$ nx`\oneparametricoption{` val v_a `}{` nx`\stringify` val Υ_2 `}`val $\Upsilon_1\rangle$

This code is used in section 28g.

29b A variation on the theme above where the parameter is not a «`string`».

prologue_declaration : ⟨`expect`⟩ **int** | ⟨`expect-rr`⟩ **int** | ⟨`start`⟩ *symbol*

A sample of the rules to which the code below applies are given above.

⟨ Prepare a generic one parametric option 29b ⟩ =
$\Upsilon \leftarrow \langle$ nx`\oneparametricoption{` val v_a `}{` val Υ_2 `}`val $\Upsilon_1\rangle$

This code is used in sections 28g and 29c.

29c *Grammar declarations.* These declarations can appear in both the prologue and the rules sections. Their treatment is very similar to the prologue-only options.

⟨ Parser common productions 29c ⟩ = 30c ▽

grammar_declaration :

precedence_declaration	⟨ Carry on 28a ⟩
symbol_declaration	⟨ Carry on 28a ⟩
⟨`start`⟩ *symbol*	$v_a \leftarrow$ ⟨ `start` ⟩⟨ Prepare a generic one parametric option 29b ⟩
code_props_type `{...}` *generic_symlist*	⟨ Assign a code fragment to symbols 30a ⟩
⟨`default-prec`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{default prec.}{ }`val $\Upsilon_1\rangle$
⟨`no-default-prec`⟩	$\Upsilon \leftarrow \langle$ nx`\optionflag{no default prec.}{ }`val $\Upsilon_1\rangle$
⟨`code`⟩ `{...}`	$\Upsilon \leftarrow \langle$ nx`\codeassoc{code}{ }`val $\Upsilon_2$val $\Upsilon_1\rangle$
⟨`code`⟩ «identifier» `{...}`	$\Upsilon \leftarrow \langle$ nx`\codeassoc{code}{` nx`\idit` val Υ_2 `}`val $\Upsilon_3$val $\Upsilon_1\rangle$

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

See also sections 30c, 30g, 31a, 31b, 37f, and 38h.
This code is used in sections ch3, 25a, and 25b.

30a ⟨ Assign a code fragment to symbols 30a ⟩ =
  $\pi_1(\Upsilon_1) \mapsto v_a$ ▷ name of the property ◁
  $\pi_1(\Upsilon_2) \mapsto v_b$ ▷ contents of the braced code ◁
  $\pi_2(\Upsilon_2) \mapsto v_c$ ▷ braced code format pointer ◁
  $\pi_3(\Upsilon_2) \mapsto v_d$ ▷ braced code stash pointer ◁
  $\pi_2(\Upsilon_1) \mapsto v_e$ ▷ code format pointer ◁
  $\pi_3(\Upsilon_1) \mapsto v_f$ ▷ code stash pointer ◁
  $\Upsilon \leftarrow \langle^{\rm nx}$`\codepropstype` { val v_a }{ val v_b }{ val Υ_3 }{ val v_c }{ val v_d }{ val v_e }{ val v_f }⟩

This code is used in section 29c.

30b ⟨ Tokens and types for the grammar parser 26b ⟩ + = 26b 35a
  ⟨`union`⟩

30c ⟨ Parser common productions 29c ⟩ + = 29c 30g
  union_name : ◦ | «identifier» . . . | ⟨ Turn an identifier into a term 38a ⟩
  grammar_declaration : ⟨`union`⟩ *union_name* `{...}` ⟨ Prepare union definition 30d ⟩
  symbol_declaration : ⟨`type`⟩ `<`*tag*`>` $symbols_1$ ⟨ Define symbol types 30e ⟩
  precedence_declaration :
    precedence_declarator $tag_{\rm opt}$ *symbols.prec* ⟨ Define symbol precedences 30f ⟩
  precedence_declarator :
    ⟨`left`⟩ | ⟨`right`⟩ | ⟨`nonassoc`⟩ | ⟨`precedence`⟩ . . . | $\Upsilon \leftarrow \langle^{\rm nx}$`\preckind` { `precedence` }val $\Upsilon_1\rangle$
  $\boldsymbol{tag}_{\rm opt}$: ◦ | `<`*tag*`>` . . . | ⟨ Prepare a `<`*tag*`>` 30h ⟩

30d ⟨ Prepare union definition 30d ⟩ =
  $\Upsilon \leftarrow \langle^{\rm nx}$`\codeassoc` { `union` }{ val Υ_2 }val $\Upsilon_3$val $\Upsilon_1\rangle$

This code is used in section 30c.

30e ⟨ Define symbol types 30e ⟩ =
  $\Upsilon \leftarrow \langle^{\rm nx}$`\typedecls` { $^{\rm nx}$`\tagit` val Υ_2 }{ val Υ_3 }val $\Upsilon_1\rangle$

This code is used in section 30c.

30f ⟨ Define symbol precedences 30f ⟩ =
  $\pi_3(\Upsilon_1) \mapsto v_a$ ▷ format pointer ◁
  $\pi_4(\Upsilon_1) \mapsto v_b$ ▷ stash pointer ◁
  $\pi_2(\Upsilon_1) \mapsto v_c$ ▷ kind of precedence ◁
  $\Upsilon \leftarrow \langle^{\rm nx}$`\precdecls` { val v_c }{ val Υ_2 }{ val Υ_3 }{ val v_a }{ val v_b }⟩

This code is used in section 30c.

30g The bootstrap grammar forms the smallest subset of the full grammar.
⟨ Parser common productions 29c ⟩ + = 30c 31a
  ⟨ Parser bootstrap productions 30i ⟩

30h ⟨ Prepare a `<`*tag*`>` 30h ⟩ =
  $\Upsilon \leftarrow \langle^{\rm nx}$`\tagit` val $\Upsilon_1\rangle$

This code is used in sections 30c, 31b, and 31c.

30i These are the two most important rules for the bootstrap parser. The reasons for the ⟨`token`⟩ declarations to be collected during the bootstrap pass are outlined in the section on bootstrapping. The ⟨`nterm`⟩ declarations are not strictly necessary for boostrapping the parsers included in `SPLinT` but they are added for the cases when the bootstrap mode is used for purposes other than bootstrapping `SPLinT`.
⟨ Parser bootstrap productions 30i ⟩ = 31c

symbol_declaration :
⟨**nterm**⟩ ⋄ *symbol_defs*$_1$ . . . | $\Upsilon \leftarrow \langle^{nx}$`\ntermdecls{`val Υ_3`}`val $\Upsilon_1\rangle$
⟨**token**⟩ ⋄ *symbol_defs*$_1$ . . . | $\Upsilon \leftarrow \langle^{nx}$`\tokendecls{`val Υ_3`}`val $\Upsilon_1\rangle$

See also sections 31c, 31d, 37e, and 37i.
This code is used in sections 24a and 30g.

31a *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 $\sqcup^{p_1}_{p_0}$). 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.

⟨ Parser common productions 29c ⟩ += 30g 31b
symbols.prec :
symbol.prec ⟨ Carry on 28a ⟩
symbols.prec symbol.prec $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}\sqcup^0_0$ val $\Upsilon_2\rangle$
symbol.prec :
symbol $\Upsilon \leftarrow \langle^{nx}$`\symbolprec{`val Υ_1`}{ }`$\rangle$
symbol **int** $\Upsilon \leftarrow \langle^{nx}$`\symbolprec{`val Υ_1`}{`val Υ_2`}`$\rangle$

31b *One or more symbols to be* ⟨**type**⟩*'d.*

⟨ Parser common productions 29c ⟩ += 31a 37f
⟨**union**⟩.*intval* :
symbols$_1$ *symbol*

symbols$_1$:
symbol ⟨ Carry on 28a ⟩
symbols$_1$ *symbol* $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}\sqcup^0_0$ val $\Upsilon_{symbol}\rangle$
generic_symlist :
generic_symlist_item ⟨ Carry on 28a ⟩
generic_symlist generic_symlist_item $\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}\sqcup^0_0$ val $\Upsilon_2\rangle$
generic_symlist_item : *symbol* | *tag* . . . | ⟨ Carry on 28a ⟩
tag : **<***tag***>** | **<*>** | **<>** . . . | ⟨ Carry on 28a ⟩

31c *One token definition.*

⟨ Parser bootstrap productions 30i ⟩ += 30i 31d
symbol_def :
<*tag***>** ⟨ Prepare a **<***tag***>** 30h ⟩
id | *id* **int** | *id string_as_id* | *id* **int** *string_as_id* . . . | $\Upsilon \leftarrow \langle^{nx}$`\onesymbol{`val Υ_1`}{`val Υ_2`}{`val Υ_3`}`$\rangle$

31d *One or more symbol definitions.*

⟨ Parser bootstrap productions 30i ⟩ += 31c 37e
symbol_defs$_1$:
symbol_def ⟨ Carry on 28a ⟩
symbol_defs$_1$ *symbol_def* ⟨ Add a symbol definition 31e ⟩

31e ⟨ Add a symbol definition 31e ⟩ =
$\pi_2(\Upsilon_2) \mapsto v_a$ ▷ the identifier ◁
$\pi_4(v_a) \mapsto v_b$ ▷ the format pointer ◁
$\pi_5(v_a) \mapsto v_c$ ▷ the stash pointer ◁
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}\sqcup^{\text{val } v_c}_{\text{val } v_b}$ val $\Upsilon_2\rangle$

This code is used in section 31d.

32a *The grammar section: between the two ⟨%⟩'s.* Finally, the following few short sections define the syntax of `bison`'s rules.

⟨ Parser grammar productions 32a ⟩ = 32b

grammar :
  rules_or_grammar_declaration ⟨ Start with a production cluster 32c ⟩
  grammar rules_or_grammar_declaration ⟨ Attach a productions cluster 32d ⟩

See also sections 32b, 35b, and 37h.
This code is used in sections ch3 and 25b.

32b **Rules syntax**

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. The type declarations for various non-terminals are used exclusively by the postprocessor whenever the 'native' `bison` term references are used (see elsewhere for details).

⟨ Parser grammar productions 32a ⟩ + = 32a 35b

⟨`union`⟩.*intval* :
    rhs id_colon named_ref $_{\text{opt}}$ *rhses*$_1$ |

rules_or_grammar_declaration :
  rules ⟨ Form a productions cluster 33a ⟩
  grammar_declaration ; ⟨ Carry on 28a ⟩
  error ; `\errmessage{parsing error!}`

rules : *id_colon named_ref* $_{\text{opt}}$ ◇ *rhses*$_1$ . . . | ⟨ Complete a production 33b ⟩

rhses$_1$:
  rhs ⟨ Start the right hand side 33c ⟩
  rhses$_1$ | ⟨ Insert local formatting 34b ⟩
    rhs ⟨ Add a right hand side to a production 34c ⟩
  rhses$_1$; ⟨ Carry on 28a ⟩

32c The next few actions describe what happens when a left hand side is attached to a rule.

⟨ Start with a production cluster 32c ⟩ =

`\initlist{\grammarprefix grammar}`
$\pi_1(\Upsilon_1) \mapsto v_a$ ▷ type of the last addition ◁
$\Upsilon \leftarrow \langle$`{\grammarprefix grammar}{`val v_a `}`$\rangle$
`\appendtolistx{\grammarprefix grammar}{`val Υ_1 `}`
def$_\text{x}$ `\grammarprefix{.\grammarprefix}`

This code is used in section 32a.

32d ⟨ Attach a productions cluster 32d ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a$ ▷ type of the last rule ◁
$\pi_1(\Upsilon_1) \mapsto v_c$ ▷ pointer to the accumulated rules ◁
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ type of the new rule ◁
let `default \positionswitchdefault`
switch (val v_b) ε `\positionswitch` ▷ determine the position of the first token in the group ◁
  ▷ determine the spacing between sections ◁
def$_\text{x}$ **next** { val v_a }
def$_\text{x}$ `default` { val v_b } ▷ reuse `\default` ◁
if$_\text{x}$ **next** `default`
    let `default \separatorswitchdefaulteq`
    switch (val v_a) ε `\separatorswitcheq`
else
    $v_a \leftarrow v_a +_\text{s} v_b$
    let `default \separatorswitchdefaultneq`
    switch (val v_a) ε `\separatorswitchneq`
fi

$\texttt{\appendtolistx}\{\,\text{val}\, v_c\,\}\{\,\text{val}\,\texttt{\postoks}\,\text{val}\, v_d \text{val}\, \Upsilon_2\,\}$
$\Upsilon \leftarrow \langle\{\,\text{val}\, v_c\,\}\{\,\text{val}\, v_b\,\}\rangle$

This code is used in sections 28f and 32a.

33a ⟨ Form a productions cluster 33a ⟩ =

$\pi_2(\Upsilon_1) \mapsto v_a$ ▷ `\prodheader` ◁
$\pi_2(v_a) \mapsto v_b$ ▷ `\idit` ◁
$\pi_4(v_b) \mapsto v_c$ ▷ format stream pointer ◁
$\pi_5(v_b) \mapsto v_d$ ▷ stash stream pointer ◁
$\pi_3(\Upsilon_1) \mapsto v_b$ ▷ `\rules` ◁
$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\oneproduction}\{\,\text{val}\, v_a \text{val}\, v_b\,\}\{\,\text{val}\, v_c\,\}\{\,\text{val}\, v_d\,\}\rangle$

This code is used in section 32b.

33b Several productions for a given nonterminal are collected in a ‘production cluster’:

rules :
id_colon named_ref$_{\text{opt}}$ (we simply return pointers below)
rhses$_1$ ⟨ Complete a production 33b ⟩

The inline action does nothing at the moment and is omitted in the main text.

⟨ Complete a production 33b ⟩ =

$\pi_4(\Upsilon_{id\_colon}) \mapsto v_a$ ▷ format stream pointer ◁
$\pi_5(\Upsilon_{id\_colon}) \mapsto v_b$ ▷ stash stream pointer ◁
$\texttt{\finishlist}\{\,\text{val}\,\Upsilon_{rhses_1}\,\}$ ▷ complete the list of rules ◁
$\Upsilon \leftarrow \langle^{\text{nx}}\texttt{\pcluster}\{^{\text{nx}}\texttt{\prodheader}\{\,\text{val}\,\Upsilon_{id\_colon}\,\}\{\,\text{val}\,\Upsilon_{named\_ref_{\text{opt}}}\,\}$
$\{\,\text{val}\, v_a\,\}\{\,\text{val}\, v_b\,\}\}\{^{\text{nx}}\texttt{\rules}\{^{\text{nx}}\texttt{\executelist}\{\,\text{val}\,\Upsilon_{rhses_1}\,\}\}\}\rangle$

This code is used in section 32b.

33c It is important to format the right hand side properly, since we would like to indicate that an action is inlined by an indentation.

***rhses*$_1$** : *rhs* ⟨ Start the right hand side 33c ⟩

The ‘layout’ of the `\rhs` ‘structure’ includes 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.

⟨ Start the right hand side 33c ⟩ =

`\initlist{\rhsesoneprefix rhses1}`
$\Upsilon \leftarrow \langle$`\rhsesoneprefix rhses1`$\rangle$
def$_{\text{x}}$ `\rhsesoneprefix{.\rhsesoneprefix }`
$\pi_{\vdash}(\Upsilon_{rhs}) \mapsto v_a$ val v_a
if (rhs = full)
  $\texttt{\appendtolistx}\{\,\text{val}\,\Upsilon\,\}\{\,\text{val}\,\Upsilon_{rhs}\,\}$
else ▷ right hand side does not end with an action, fake one ◁
  $\pi_{\{\}}(\Upsilon_{rhs}) \mapsto v_a$ ▷ rules ◁
  $\texttt{\yytoksempty}\, v_a \leftarrow \langle\, v_a \leftarrow \langle ⌜\ldots⌝ \rangle\,\rangle\{\,\}$
  $\texttt{\appendtolistx}\{\,\text{val}\,\Upsilon\,\}\{^{\text{nx}}\texttt{\rhs}\{\,\text{val}\, v_a{}^{\text{nx}}\texttt{\rarhssep}\{0\}\{0\}$
    $^{\text{nx}}\texttt{\actbraces}\{\,\}\{\,\}\{0\}\{0\}^{\text{nx}}\texttt{\bdend}\,\}\{\,\}\{^{\text{nx}}\text{rhs} = \text{full}\,\}\}$
fi

This code is used in section 32b.

33d Using standard notation, here is what the middle action does. The part of the rule this action applies to is given below for reference. This action may have been omitted altogether but it serves as a good illustration of how ‘inline actions’ work.

***rhses*$_1$** : *rhses*$_1$ | ⟨ Insert local formatting 34b ⟩

The terms are counted from left (deeper in the value stack) to right (on top of the value stack) although Υ_0 (which is the same as Υ) is the *right*most term, i.e. the implicit action itself.

What the parser sees at this point are the first two terms on the stack (i.e. $rhses_1$ and `|`) and is ready to make a reduction which will push the value of the term corresponding to the inline action (i.e. ⟨ Insert local formatting 34b ⟩) on the stack.

The way `bison` does this is by introducing a new grammar term (named `$@`n for some integer n) for each inline action and adding a new rule that reduces an empty sequence of terms to `$@`n. The action for this rule is the inline action. In our case this would read as

`$@`n : ∘ ⟨ Insert local formatting 34b ⟩

...except the parser knows what the state of the stack is at this point and thus the code inside ⟨ Insert local formatting 34b ⟩ can now refer to the terms on the stack as described above.

⟨ Old 'Insert local formatting' 33d ⟩ =
`\appendtolistx{` val Υ_1 `}{` ^{nx}`\midf` val Υ_2 `}`

34a However, if the length of the rule preceding the inline action is not known to the parser in advance (as is the case for the parsers `SPLinT` generates using any version of `bison` that is ⩾ 3.0) a different way of accessing the stack is necessary. This notation is also more natural as it counts the terms from right to left, i.e. 'into the depths of the stack' (for example $_2\Upsilon$ is the register holding the value of $rhses_1$). It is worth noting that in this case Υ_0 and Υ are still the same register, the one that holds the value of the term corresponding to the inline action itself.

⟨ Newer 'Insert local formatting' 34a ⟩ =
$_2\Upsilon \to [v_a]$ $_1\Upsilon \to [v_b]$
`\appendtolistx{` val v_a `}{` ^{nx}`\midf` val v_b `}`

34b Finally, using the 'native' way of referring to term values results in the most natural code. In this case, one can mix numeric and symbolic references for both implicit and explicit rules.

⟨ Insert local formatting 34b ⟩ =
`\appendtolistx{` val Υ_{rhses} `}{` ^{nx}`\midf` val Υ_{mid} `}`

This code is cited in sections 33d and 34c.

This code is used in section 32b.

34c Productions are collected in a 'productions cluster' (not an official term) by the following action:

***rhses*₁** : $rhses_1$ `|` ⋄ rhs ... | ⟨ Add a right hand side to a production 34c ⟩

As can be seen in the code below, no pointers are provided for an *implicit* action (since there are no tokens associated with it).

Processing a set of rules involves a large number of reexpansions. This seems to be a good place to use a list to store the nodes (see `yycommon.sty` for details on list macros). 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 list 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.

The code below relies on the inline action ⟨ Insert local formatting 34b ⟩ above to store the relevant information from Υ_1 (corresponding to $rhses_1$) in Υ_3 (which is the inline action 'term' ⋄ in the production above).

⟨ Add a right hand side to a production 34c ⟩ =
$\pi_\vdash(\Upsilon_4) \mapsto v_a$ val v_a
if (rhs = full)
  `\appendtolistx{` val Υ_1 `}{` ^{nx}`\rrhssep` val Υ_2 val Υ_4 `}`
else
  $\pi_{\{\}}(\Upsilon_4) \mapsto v_a$
  `\yytoksempty` v_a ← ⟨ v_a ← ⟨⌜...⌝⟩ ⟩`{ }`
  `\appendtolistx{` val Υ_1 `}{` ^{nx}`\rrhssep` val Υ_2
    ^{nx}`\rhs{` val v_a ^{nx}`\rarhssep{0}{0}` ▷ streams have already been grabbed ◁
    ^{nx}`\actbraces{ }{ }{0}{0}` ^{nx}`\bdend }{ }{` ^{nx}rhs = full `}}`

fi
$\Upsilon \leftarrow \langle \text{val}\,\Upsilon_1 \rangle$

This code is used in section 32b.

35a ⟨ Tokens and types for the grammar parser 26b ⟩ += 30b
⟨empty⟩

35b The centerpiece of the grammar is the syntax of the right hand side of a production. Various 'precedence hints' must be attached to an appropriate portion of the rule, just before an action (which can be inline, implicit or both in this case).

⟨ Parser grammar productions 32a ⟩ += 32b 37h

rhs :

∘	⟨ Make an empty right hand side 35c ⟩
rhs symbol named_ref $_{\text{opt}}$	⟨ Add a term to the right hand side 35d ⟩
rhs `{...}` *named_ref* $_{\text{opt}}$	⟨ Add an action to the right hand side 35e ⟩
rhs `%?{...}`	⟨ Add a predicate to the right hand side 36a ⟩
rhs ⟨`empty`⟩	⟨ Add ⟨`empty`⟩ to the right hand side 36b ⟩
rhs ⟨`prec`⟩ *symbol*	⟨ Add a precedence directive to the right hand side 36c ⟩
rhs ⟨`dprec`⟩ **int**	⟨ Add a ⟨`dprec`⟩ directive to the right hand side 37a ⟩
rhs ⟨`merge`⟩ `<`*tag*`>`	⟨ Add a ⟨`merge`⟩ directive to the right hand side 37b ⟩

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

∘	⟨ Create an empty named reference 37c ⟩
`"[identifier]"` $_{\text{m}}$	⟨ Create a named reference 37d ⟩

35c The simplest form of the right hand side is an empty rule. In this case the parser must make a reduction based on the lookahead only (or the current state), i.e. no tokens are consumed from the input.

⟨ Make an empty right hand side 35c ⟩ =
$\Upsilon \leftarrow \langle{}^{\text{nx}}$`\rhs{ }{ }{`${}^{\text{nx}}\text{rhs} = \text{not full}$ `}`$\rangle$

This code is used in section 35b.

35d Adding a `bison` term to the right hand side involves collecting of several pieces of information. One of them is the (optional) symbolic named that can be used by the action code to refer to the place on the value stack that is allocated for this term.

rhs : *rhs symbol named_ref* $_{\text{opt}}$ ⟨ Add a term to the right hand side 35d ⟩

The space between the term and the preceeding part of the rule may depend on the type of rule element that appears at the end of the rule parsed so far.

⟨ Add a term to the right hand side 35d ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
`\yytoksempty` $v_b \leftarrow \langle\ \rangle$`{`
$\pi_4(\Upsilon_2) \mapsto v_c$
$\pi_5(\Upsilon_2) \mapsto v_d$
$v_b \leftarrow v_b +_{\text{sx}} [$ `{` $\text{val}\, v_c$ `}{` $\text{val}\, v_d$ `}` $]$
`}`
$\Upsilon \leftarrow \langle{}^{\text{nx}}$`\rhs{` $\text{val}\, v_a \text{val}\, v_b{}^{\text{nx}}$`\termname{` $\text{val}\,\Upsilon_2$ `}{` $\text{val}\,\Upsilon_3$ `}}{`${}^{\text{nx}}$␣ `}{`${}^{\text{nx}}\text{rhs} = \text{not full}$ `}`$\rangle$

This code is used in section 35b.

35e Action processing is somewhat complicated since the action can be either inline or terminal, affecting the typesetting.

rhs : *rhs* `{...}` *named_ref* $_{\text{opt}}$ ⟨ Add an action to the right hand side 35e ⟩

Additionally, an action may follow an empty rule in which case a special term must be added to aid the reader.

⟨ Add an action to the right hand side 35e ⟩ =

$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_b$ val v_b
if (rhs = full) ▷ the first half ends with an action ◁
  $v_a \leftarrow v_a +_{\mathrm{sx}}$ [$^{\mathrm{nx}}$`\arhssep{0}{0}`$^{\mathrm{nx}}$⌜...⌝] ▷ no pointers to streams ◁
fi
`\yytoksempty` v_a← ⟨ v_a← ⟨⌜...⌝⟩ ⟩{ }
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ the contents of the braced code ◁
$\pi_2(\Upsilon_2) \mapsto v_c$ ▷ the format stream pointer ◁
$\pi_3(\Upsilon_2) \mapsto v_d$ ▷ the stash stream pointer ◁
$\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\rhs`{ val v_a $^{\mathrm{nx}}$`\rarhssep`{ val v_c }{ val v_d }
  $^{\mathrm{nx}}$`\actbraces`{ val v_b }{ val Υ_3 }{ val v_c }{ val v_d }$^{\mathrm{nx}}$`\bdend` }
    { $^{\mathrm{nx}}$`\arhssep` }{ $^{\mathrm{nx}}$rhs = full }⟩

This code is used in section 35b.

36a ⟨ Add a predicate to the right hand side 36a ⟩ =

$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_b$ val v_b
if (rhs = full) ▷ the first half ends with an action ◁
  $v_a \leftarrow v_a +_{\mathrm{sx}}$ [$^{\mathrm{nx}}$`\arhssep{0}{0}`$^{\mathrm{nx}}$⌜...⌝] ▷ no pointers to streams ◁
fi
`\yytoksempty` v_a← ⟨ v_a← ⟨⌜...⌝⟩ ⟩{ }
$\pi_1(\Upsilon_2) \mapsto v_b$ ▷ the contents of the braced code ◁
$\pi_2(\Upsilon_2) \mapsto v_c$ ▷ the format stream pointer ◁
$\pi_3(\Upsilon_2) \mapsto v_d$ ▷ the stash stream pointer ◁
$\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\rhs`{ val v_a $^{\mathrm{nx}}$`\rarhssep`{ val v_c }{ val v_d }
  $^{\mathrm{nx}}$`\bpredicate`{ val v_b }{ }{ val v_c }{ val v_d }$^{\mathrm{nx}}$`\bdend` }{ $^{\mathrm{nx}}$`\arhssep` }{ $^{\mathrm{nx}}$rhs = full }⟩

This code is used in section 35b.

36b An empty right hand side may be specified explicitly by using ⟨`empty`⟩ as the sole token in the production. This will increase the readability of the grammar by making the programmer's intentions more transparent.

⟨ Add ⟨`empty`⟩ to the right hand side 36b ⟩ =

$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
`\yytoksempty` v_b← ⟨ ⟩{
  $\pi_4(\Upsilon_2) \mapsto v_c$
  $\pi_5(\Upsilon_2) \mapsto v_d$
  $v_b \leftarrow v_b +_{\mathrm{sx}}$ [{ val v_c }{ val v_d }]
}
$\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\rhs`{ val v_a val v_b $^{\mathrm{nx}}$⌜...⌝}{ $^{\mathrm{nx}}$␣ }{ $^{\mathrm{nx}}$rhs = not full }⟩

This code is used in section 35b.

36c ⟨ Add a precedence directive to the right hand side 36c ⟩ =

$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c$ val v_c
if (rhs = full)
  $\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\sprecop`{ val Υ_3 }val Υ_2⟩ ▷ reuse `\yyval` ◁
  `\supplybdirective` v_a Υ ▷ the directive is 'absorbed' by the action ◁
  $\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\rhs`{ val v_a }{ val v_b }{ $^{\mathrm{nx}}$rhs = full }⟩
else
  $\Upsilon \leftarrow$ ⟨$^{\mathrm{nx}}$`\rhs`{ val v_a $^{\mathrm{nx}}$`\sprecop`{ val Υ_3 }val Υ_2 }{ val v_b }{ $^{\mathrm{nx}}$rhs = not full }⟩
fi

This code is used in section 35b.

37a ⟨ Add a ⟨dprec⟩ directive to the right hand side 37a ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c$ val v_c
if (rhs = full)
  $\Upsilon \leftarrow$ ⟨ nx\dprecop { val Υ_3 }val Υ_2⟩ ▷ reuse \yyval ◁
  \supplybdirective $v_a \Upsilon$ ▷ the directive is 'absorbed' by the action ◁
  $\Upsilon \leftarrow$ ⟨ nx\rhs { val v_a }{ val v_b }{ nx rhs = full }⟩
else
  $\Upsilon \leftarrow$ ⟨ nx\rhs { val v_a nx\dprecop { val Υ_3 }val Υ_2 }{ val v_b }{ nx rhs = not full }⟩
fi

This code is used in section 35b.

37b ⟨ Add a ⟨merge⟩ directive to the right hand side 37b ⟩ =
$\pi_{\{\}}(\Upsilon_1) \mapsto v_a$
$\pi_{\leftrightarrow}(\Upsilon_1) \mapsto v_b$
$\pi_{\vdash}(\Upsilon_1) \mapsto v_c$ val v_c
if (rhs = full)
  $\Upsilon \leftarrow$ ⟨ nx\mergeop { nx\tagit val Υ_3 }val Υ_2⟩ ▷ reuse \yyval ◁
  \supplybdirective $v_a \Upsilon$ ▷ the directive is 'absorbed' by the action ◁
  $\Upsilon \leftarrow$ ⟨ nx\rhs { val v_a }{ val v_b }{ nx rhs = full }⟩
else
  $\Upsilon \leftarrow$ ⟨ nx\rhs { val v_a nx\mergeop { nx\tagit val Υ_3 }val Υ_2 }{ val v_b }{ nx rhs = not full }⟩
fi

This code is used in section 35b.

37c ⟨ Create an empty named reference 37c ⟩ =
$\Upsilon \leftarrow$ ⟨⟩

This code is used in section 35b.

37d ⟨ Create a named reference 37d ⟩ =
⟨ Turn an identifier into a term 38a ⟩

This code is used in section 35b.

37e **Identifiers and other symbols**

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

⟨ Parser bootstrap productions 30i ⟩ += 31d 37i
id :
  «identifier» ⟨ Turn an identifier into a term 38a ⟩
  char ⟨ Turn a character into a term 38b ⟩

37f ⟨ Parser common productions 29c ⟩ += 31b 38h
⟨ Definition of *symbol* 37g ⟩

37g ⟨ Definition of *symbol* 37g ⟩ =
symbol :
  id ⟨ Turn an identifier into a symbol 38c ⟩
  string_as_id ⟨ Turn a string into a symbol 38d ⟩

This code is used in section 37f.

37h ⟨ Parser grammar productions 32a ⟩ += 35b
id_colon : «identifier: » ⟨ Prepare the left hand side 38e ⟩

37i A string used as an «identifier».

⟨ Parser bootstrap productions 30i ⟩ += 37e
string_as_id : «string» ⟨ Prepare a string for use 38f ⟩

38a The remainder of the action code is trivial but we reserved the placeholders for the appropriate actions in case the parser gains some sophistication in processing low level types (or starts expecting different types from the scanner).

⟨ Turn an identifier into a term 38a ⟩ =
  $\Upsilon \leftarrow \langle^{nx}$`\idit` val $\Upsilon_1\rangle$

This code is used in sections 30c, 37d, 37e, 38e, and 38g.

38b ⟨ Turn a character into a term 38b ⟩ =
  $\Upsilon \leftarrow \langle^{nx}$`\charit` val $\Upsilon_1\rangle$

This code is used in section 37e.

38c ⟨ Turn an identifier into a symbol 38c ⟩ =
  ⟨ Carry on 28a ⟩

This code is used in section 37g.

38d ⟨ Turn a string into a symbol 38d ⟩ =
  ⟨ Carry on 28a ⟩

This code is used in section 37g.

38e ⟨ Prepare the left hand side 38e ⟩ =
  ⟨ Turn an identifier into a term 38a ⟩

This code is used in section 37h.

38f ⟨ Prepare a string for use 38f ⟩ =
  $\Upsilon \leftarrow \langle^{nx}$`\stringify` val $\Upsilon_1\rangle$

This code is used in sections 37i and 38g.

38g *Variable and value. The* «`string`» *form of variable is deprecated and is not* `M4`*-friendly. For example,* `M4` *fails for* `%define "[" "value"`.

⟨ Parser prologue productions 28d ⟩ + = 28g

variable : «identifier» | «`string`» . . . | ⟨ Prepare a string for use 38f ⟩

value : ○ | «identifier» | «`string`» | `{...}` . . . | $\Upsilon \leftarrow \langle^{nx}$`\bracedvalue` val $\Upsilon_1\rangle$

38h ⟨ Parser common productions 29c ⟩ + = 37f

epilogue$_{opt}$: ○ | ⟨%⟩ `epilogue`

38i 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 38i ⟩ =

This code is used in sections ch3, 24a, 25a, and 25b.

38j C postamble for the grammar parser. It is tricky to insert function definitions that use `bison`'s internal types, as they have to be inserted in a place that is aware of the internal definitions but before said definitions are used.

⟨ Grammar parser C postamble 38j ⟩ =

This code is used in sections ch3, 25a, 25b, and 38k.

38k ⟨ Bootstrap parser C postamble 38k ⟩ =
  ⟨ Grammar parser C postamble 38j ⟩
  ⟨ Bootstrap token output 38l ⟩

This code is used in section 24a.

38l ⟨ Bootstrap token output 38l ⟩ =

```
  void bootstrap_tokens(char *bootstrap_token_format){
#define _register_token_d(name) fprintf(tables_out, bootstrap_token_format, #name, name, #name);
      ⟨ Bootstrap token list 38m ⟩
#undef _register_token_d
      }
```

This code is used in section 38k.

38m 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 38m ⟩ =

```
  _register_token_d(ID)
```

_register_token_d(`PERCENT_TOKEN`)
_register_token_d(`STRING`)

This code is used in section 38l.

39a **Union of types**

This section of the `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 **union** into any data structure usable in TEX), this section is left (nearly) empty. The reason for the lonely type below is the postprocessor that facilitates the use of `bison` 'native' term references (see elsewhere). In order to translate such references into appropriate TEX code, the postprocessor must let `bison` calculate offsets into the value stack, which requires assigning types to various terminals and non-terminals. The specific type has no significance.

⟨ Union of grammar parser types 39a ⟩ =
int *intval*;

This code is used in sections ch3, 24a, 25a, and 25b.

4

The scanner for bison syntax

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.

```
⟨ lo.ll  ch4 ⟩ =
  ⟨ Grammar lexer definitions 41a ⟩
  ........................................
  ⟨ Grammar lexer C preamble 43c ⟩
  ........................................
  ⟨ Grammar lexer options 43d ⟩

  ⟨ Grammar token regular expressions 43e ⟩

  void define_all_states(void)
  {
          ⟨ Collect state definitions for the grammar lexer 42c ⟩
  }
```

41a **Definitions and state declarations**

It is convenient to abbreviate some commonly used subexpressions.

```
⟨ Grammar lexer definitions 41a ⟩ =                                                     42a ▽
  ⟨ Grammar lexer states 42d ⟩
  ⟨letter⟩        [.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_]
  ⟨notletter⟩     [.abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ_]^c \ [%{]
  ⟨id⟩            ⟨letter⟩ (⟨letter⟩ | [-0–9])*
  ⟨int⟩           [0–9]+
```

See also sections 42a and 42b.

This code is used in section ch4.

42a *Zero or more instances of backslash-newline. Following* `gcc`*, allow white space between the backslash and the newline.*

⟨ Grammar lexer definitions 41a ⟩ + = 41a 42b
⟨splice⟩ (\ [␣⟨f⟩⟨t⟩⟨v⟩]* ⟨n⟩)*

42b *An equal sign, with optional leading whitespaces. This is used in some deprecated constructs.*

⟨ Grammar lexer definitions 41a ⟩ + = 42a
⟨eqopt⟩ ([⟨␣⟩]* =)?

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 ch4.

42d *A* C*-like comment in directives/rules.*

⟨ Grammar lexer states 42d ⟩ = 42e
⟨**state-x**⟩f **SC_YACC_COMMENT**

See also sections 42e, 42f, 42g, 42h, 42i, 43a, and 43b.
This code is used in section 41a.

42e *Strings and characters in directives/rules.*

⟨ Grammar lexer states 42d ⟩ + = 42d 42f
⟨**state-x**⟩f **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 ⟩ + = 42e 42g
⟨**state-x**⟩f **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* Λ*, as this simplifies our implementation. We match angle brackets in nested pairs: several languages use them for generics/template types.*

⟨ Grammar lexer states 42d ⟩ + = 42f 42h
⟨**state-x**⟩f **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 ⟩ + = 42g 42i
⟨**state-x**⟩f **SC_PROLOGUE SC_BRACED_CODE SC_EPILOGUE SC_PREDICATE**

42i C *and* C++ *comments in code.*

⟨ Grammar lexer states 42d ⟩ + = 42h 43a
⟨**state-x**⟩f **SC_COMMENT SC_LINE_COMMENT**

43a *Strings and characters in code.*

⟨ Grammar lexer states 42d ⟩ + = △ 42i 43b ▽

⟨**state-x**⟩f **SC_STRING SC_CHARACTER**

43b Bracketed identifiers support.

⟨ Grammar lexer states 42d ⟩ + = △ 43a

⟨**state-x**⟩f **SC_BRACKETED_ID SC_RETURN_BRACKETED_ID**

43c ⟨ Grammar lexer C preamble 43c ⟩ =

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

This code is used in section ch4.

43d The code for the generated scanner is highly dependent on the options supplied. Most of the options below are essential for the scheme adopted in this package to work.

⟨ Grammar lexer options 43d ⟩ =

```
⟨option⟩f          bison-bridge
⟨option⟩f          noyywrap
⟨option⟩f          nounput
⟨option⟩f          noinput
⟨option⟩f          reentrant
⟨option⟩f          noyy_top_state
⟨option⟩f          debug
⟨option⟩f          stack
⟨output to⟩f       "lo.c"
```

This code is used in section ch4.

43e **Tokenizing with regular expressions**

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

⟨ Grammar token regular expressions 43e ⟩ =

⟨ Scan grammar white space 43f ⟩
⟨ Scan `bison` directives 44a ⟩
⟨ Do not support zero characters 47c ⟩
⟨ Scan after an identifier, check whether a colon is next 47d ⟩
⟨ Scan bracketed identifiers 48b ⟩
⟨ Scan a `yacc` comment 49c ⟩
⟨ Scan a C comment 49d ⟩
⟨ Scan a line comment 49e ⟩
⟨ Scan a `bison` string 49f ⟩
⟨ Scan a character literal 50a ⟩
⟨ Scan a tag 50c ⟩
⟨ Decode escaped characters 50f ⟩
⟨ Scan user-code characters and strings 51a ⟩
⟨ Strings, comments etc. found in user code 51b ⟩
⟨ Scan code in braces 51c ⟩
⟨ Scan prologue 52b ⟩
⟨ Scan the epilogue 52d ⟩
⟨ Add the scanned symbol to the current string 52f ⟩

This code is used in section ch4.

43f ⟨ Scan grammar white space 43f ⟩ =

INITIAL SC_AFTER_IDENTIFIER SC_BRACKETED_ID SC_RETURN_BRACKETED_ID++

▷ *comments and white space*

```
,                          warn⟨stray ‘,’ treated as white space⟩
```

```
[␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩]                    ↩
//.*                                continue
/*                                  \contextstate\YYSTART enter(SC_YACC_COMMENT) continue
▷ #line directives are not documented, and may be withdrawn or modified in future versions of bison
⊣ #line␣⟨int⟩(␣".*")?⟨n⟩           continue
```

This code is used in section 43e.

44a *For directives that are also command line options, the regex must be* "%..." *after* "[-_]"*'s are removed, and the directive must match the* --long *option name, with a single string argument. Otherwise, add exceptions to* ../build-aux/cross-options.pl. For most options the scanner returns a pair of pointers as the value.

⟨ Scan bison directives 44a ⟩ =

```
INITIAL++
  %binary                       return_p ⟨nonassoc⟩
  %code                         return_p ⟨code⟩
  %debug                        ⟨ Set ⟨debug⟩ flag 46a ⟩
  %default-prec                 return_p ⟨default-prec⟩
  %define                       return_p ⟨define⟩
  %defines                      return_p ⟨defines⟩
  %destructor                   return_p ⟨destructor⟩
  %dprec                        return_p ⟨dprec⟩
  %empty                        return_p ⟨empty⟩
  %error-verbose                return_p ⟨error-verbose⟩
  %expect                       return_p ⟨expect⟩
  %expect-rr                    return_p ⟨expect-rr⟩
  %file-prefix                  return_p ⟨file-prefix⟩
  %fixed-output-files           return_p ⟨yacc⟩
  %initial-action               return_p ⟨initial-action⟩
  %glr-parser                   return_p ⟨glr-parser⟩
  %language                     return_p ⟨language⟩
  %left                         return_p ⟨left⟩
  %lex-param                    ⟨ Return lexer parameters 46b ⟩
  %locations                    ⟨ Set ⟨locations⟩ flag 46c ⟩
  %merge                        return_p ⟨merge⟩
  %name-prefix                  return_p ⟨name-prefix⟩
  %no-default-prec              return_p ⟨no-default-prec⟩
  %no-lines                     return_p ⟨no-lines⟩
  %nonassoc                     return_p ⟨nonassoc⟩
  %nondeterministic-parser      return_p ⟨nondet. parser⟩
  %nterm                        return_p ⟨nterm⟩
  %output                       return_p ⟨output⟩
  %param                        ⟨ Return lexer and parser parameters 46d ⟩
  %parse-param                  ⟨ Return parser parameters 46e ⟩
  %prec                         return_p ⟨prec⟩
  %precedence                   return_p ⟨precedence⟩
  %printer                      return_p ⟨printer⟩
  %pure-parser                  ⟨ Set ⟨pure-parser⟩ flag 46f ⟩
  %require                      return_p ⟨require⟩
  %right                        return_p ⟨right⟩
  %skeleton                     return_p ⟨skeleton⟩
  %start                        return_p ⟨start⟩
  %term                         return_p ⟨token⟩
  %token                        return_p ⟨token⟩
  %token-table                  return_p ⟨token-table⟩
  %type                         return_p ⟨type⟩
  %union                        return_p ⟨union⟩
  %verbose                      return_p ⟨verbose⟩
```

```
%yacc                                        return_p ⟨yacc⟩
▷ deprecated
%default[-_]prec                             deprecated⟨ %default-prec ⟩
%error[-_]verbose                            deprecated⟨ %define parse.error verbose ⟩
%expect[-_]rr                                deprecated⟨ %expect-rr ⟩
%file-prefix⟨eqopt⟩                          deprecated⟨ %file-prefix ⟩
%fixed[-_]output[-_]files                    deprecated⟨ %fixed-output-files ⟩
%name[-_]prefix⟨eqopt⟩                       deprecated⟨ %name-prefix ⟩
%no[-_]default[-_]prec                       deprecated⟨ %no-default-prec ⟩
%no[-_]lines                                 deprecated⟨ %no-lines ⟩
%output⟨eqopt⟩                               deprecated⟨ %output ⟩
%pure[-_]parser                              deprecated⟨ %pure-parser ⟩
%token[-_]table                              deprecated⟨ %token-table ⟩
▷ semantic predicate
%? [␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩]*{                          enter(SC_PREDICATE) continue
%⟨id⟩ | %⟨notletter⟩ ([⟨❧⟩])+                 ⟨ Possibly complain about a bad directive 46g ⟩
=                                            return_p "="_m
|                                            return_p "|"_m
;                                            return_p ";"_m
⟨id⟩                                         ⟨ Prepare an identifier 46h ⟩
⟨int⟩                                        def_x next { \yylval { ^nx\anint { val \yytext }
                                                 { val \yyfmark }{ val \yysmark } } }next
                                             return_l int
0[xX] [0–9abcdefABCDEF]+                     def_x next { \yylval { ^nx\hexint { val \yytext }
                                                 { val \yyfmark }{ val \yysmark } } }next
                                             return_l int
▷ identifiers may not start with a digit; yet, don't silently accept 1foo as 1 foo
⟨int⟩⟨id⟩                                    fatal⟨ invalid identifier: val \yytext ⟩
▷ characters
'                                            enter(SC_ESCAPED_CHARACTER) continue
▷ strings
"                                            enter(SC_ESCAPED_STRING) continue
▷ prologue
%{                                           ⟨ Start assembling prologue code 47b ⟩
▷ code in between braces; originally preceded by \STRINGGROW but it is omitted here
{                                            \lonesting 0_R enter(SC_BRACED_CODE) continue
▷ a type
<*>                                          return_p "<*>"_m
<>                                           return_p "<>"_m
<                                            \lonesting = 0_R enter(SC_TAG) continue
%%                                           ⟨ Switch sections 47a ⟩
[                                            let \bracketedidstr = ∅
                                             \bracketedidcontextstate \YYSTART
                                             enter(SC_BRACKETED_ID) continue
⟨EOF⟩                                        \yyterminate   ▷ ⟨EOF⟩ in INITIAL ◁
[[%A–Za–z0–9_<>{}"'*;|=/,␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩]^c+ | .   ⟨ Process a bad character 45a ⟩
```

This code is used in section 43e.

45a 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 45a ⟩ =

```
\expandafter let \expandafter next \csname lexspecial[val \yytextpure]\endcsname
```

```
if_x next ∘
    if_t [bad char]
        fatal⟨ invalid character(s): val\yytext ⟩
    fi
else
    \expandafter\lexspecialchar\expandafter{ next }{ val\yyfmark }{ val\yysmark }continue
fi
```

This code is used in section 44a.

46a ⟨ Set ⟨debug⟩ flag 46a ⟩ =

```
def_x next {\yylval{{parse.trace}{debug}{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨<flag>⟩
```

This code is used in section 44a.

46b ⟨ Return lexer parameters 46b ⟩ =

```
def_x next {\yylval{{lex-param}{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨param⟩
```

This code is used in section 44a.

46c ⟨ Set ⟨locations⟩ flag 46c ⟩ =

```
def_x next {\yylval{{locations}{ }{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨<flag>⟩
```

This code is used in section 44a.

46d ⟨ Return lexer and parser parameters 46d ⟩ =

```
def_x next {\yylval{{both-param}{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨param⟩
```

This code is used in section 44a.

46e ⟨ Return parser parameters 46e ⟩ =

```
def_x next {\yylval{{parse-param}{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨param⟩
```

This code is used in section 44a.

46f ⟨ Set ⟨pure-parser⟩ flag 46f ⟩ =

```
def_x next {\yylval{{api.pure}{pure-parser}{ val\yyfmark }{ val\yysmark }}}next
return_l ⟨<flag>⟩
```

This code is used in section 44a.

46g ⟨ Possibly complain about a bad directive 46g ⟩ =

```
if_t [bad char]
    warn⟨ invalid directive: val\yytext ⟩
fi
```

This code is used in section 44a.

46h At this point we save the spelling and the location of the identifier. The token is returned later, after the context is known.

⟨ Prepare an identifier 46h ⟩ =

```
def_x next {\yylval{{ val\yytextpure }{ val\yytext }
    { val\yyfmark }{ val\yysmark }}}next
let \bracketedidstr = ∅
enter(SC_AFTER_IDENTIFIER) continue
```

This code is used in section 44a.

47a ⟨ Switch sections 47a ⟩ =

```
add\percentpercentcount 1_R
if_ω \percentpercentcount = 2_R
    enter(SC_EPILOGUE)
fi
return_p ⟨%⟩
```

This code is used in section 44a.

47b ⟨ Start assembling prologue code 47b ⟩ =

```
def_x next {\postoks{{val\yyfmark}{val\yysmark}}}next
enter(SC_PROLOGUE) continue
```

This code is used in section 44a.

47c *Supporting* 0_8 *complexifies our implementation for no expected added value.*

⟨ Do not support zero characters 47c ⟩ =

```
SC_ESCAPED_CHARACTER SC_ESCAPED_STRING SC_TAG++
  0_8                                              warn⟨invalid null character⟩
```

This code is used in section 43e.

47d ⟨ Scan after an identifier, check whether a colon is next 47d ⟩ =

```
SC_AFTER_IDENTIFIER++
  [                    ⟨ Process the bracketed part of an identifier 47e ⟩
  :                    ⟨ Process a colon after an identifier 47f ⟩
  ⟨EOF⟩                ⟨ End the scan with an identifier 48a ⟩
  .                    ⟨ Process a character after an identifier 47g ⟩
```

This code is used in section 43e.

47e ⟨ Process the bracketed part of an identifier 47e ⟩ =

```
if_x \bracketedidstr ∅
    \bracketedidcontextstate \YYSTART  enter(SC_BRACKETED_ID)
    \yybreak continue
else
    \ROLLBACKCURRENTTOKEN
    enter(SC_RETURN_BRACKETED_ID)
    \yybreak{return_l «identifier»}
\yycontinue
```

This code is used in section 47d.

47f ⟨ Process a colon after an identifier 47f ⟩ =

```
if_x \bracketedidstr ∅
    enter(INITIAL)
else
    enter(SC_RETURN_BRACKETED_ID)
fi
return_l «identifier: »
```

This code is used in section 47d.

47g ⟨ Process a character after an identifier 47g ⟩ =

```
\ROLLBACKCURRENTTOKEN
if_x \bracketedidstr ∅
    enter(INITIAL)
else
    enter(SC_RETURN_BRACKETED_ID)
fi
return_l «identifier»
```

This code is used in section 47d.

48a ⟨End the scan with an identifier 48a⟩ =

```
if_x \bracketedidstr ∅
    enter(INITIAL)
else
    enter(SC_RETURN_BRACKETED_ID)
fi
\ROLLBACKCURRENTTOKEN
return_l «identifier»
```

This code is used in section 47d.

48b ⟨Scan bracketed identifiers 48b⟩ = 49a ▽

```
SC_BRACKETED_ID++
  ⟨EOF⟩                                   ⟨Complain about unexpected end of file inside brackets 48f⟩
  ⟨id⟩                                    ⟨Process bracketed identifier 48c⟩
  ]                                       ⟨Finish processing bracketed identifier 48d⟩
  [].A–Za–z0–9_/␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩]^c+ | .      ⟨Complain about improper identifier characters 48e⟩
```

See also section 49a.

This code is used in section 43e.

48c ⟨Process bracketed identifier 48c⟩ =

```
if_x \bracketedidstr ∅
    def_x \bracketedidstr { { val \yytextpure }{ val \yytext }
        { val \yyfmark }{ val \yysmark } }
    \yybreak continue
else
    \yybreak { warn⟨unexpected identifier
        in bracketed name: val \yytext } }
\yycontinue
```

This code is used in section 48b.

48d ⟨Finish processing bracketed identifier 48d⟩ =

```
enter_x \bracketedidcontextstate
if_x \bracketedidstr ∅
    \yybreak { warn⟨an identifier expected⟩ }
else
    if_ω \bracketedidcontextstate = state(INITIAL) ∘
        \expandafter \yylval \expandafter { \bracketedidstr }
        let \bracketedidstr = ∅
        \yybreak@ { return_l "[identifier]"_m }
    else
        \yybreak@ continue
    fi
\yycontinue
```

This code is used in section 48b.

48e ⟨Complain about improper identifier characters 48e⟩ =

```
fatal⟨invalid character(s) in bracketed name: val \yytext ⟩
```

This code is used in section 48b.

48f ⟨Complain about unexpected end of file inside brackets 48f⟩ =

```
enter_x \bracketedidcontextstate
fatal⟨unexpected end of file inside brackets⟩
```

This code is used in section 48b.

49a ⟨ Scan bracketed identifiers 48b ⟩ + = △ 48b

```
SC_RETURN_BRACKETED_ID++
    .                                   ⟨ Return a bracketed identifier 49b ⟩
```

49b ⟨ Return a bracketed identifier 49b ⟩ =

```
\ROLLBACKCURRENTTOKEN
\expandafter\yylval\expandafter{\bracketedidstr }
let \bracketedidstr = ∅
enter(INITIAL)
return_l "[identifier]"_m
```

This code is used in section 49a.

49c *Scanning a* `yacc` *comment. The initial* `/*` *is already eaten.*

⟨ Scan a `yacc` comment 49c ⟩ =

```
SC_YACC_COMMENT++
    ⟨EOF⟩                               fatal⟨ unexpected end of file in a comment ⟩
    */                                  enter_x \contextstate continue
    . | ⟨n⟩                             continue
```

This code is used in section 43e.

49d *Scanning a* C *comment. The initial* `/*` *is already eaten.*

⟨ Scan a C comment 49d ⟩ =

```
SC_COMMENT++
    ⟨EOF⟩                               fatal⟨ unexpected end of file in a comment ⟩
    *⟨splice⟩/                          \STRINGGROW enter_x \contextstate continue
```

This code is used in section 43e.

49e *Scanning a line comment. The initial* `//` *is already eaten.*

⟨ Scan a line comment 49e ⟩ =

```
SC_LINE_COMMENT++
    ⟨EOF⟩                               enter_x \contextstate \ROLLBACKCURRENTTOKEN
                                            continue
    ⟨n⟩                                 \STRINGGROW enter_x \contextstate continue
    ⟨splice⟩                            \STRINGGROW continue
```

This code is used in section 43e.

49f *Scanning a* `bison` *string, including its escapes. The initial quote is already eaten.*

⟨ Scan a `bison` string 49f ⟩ =

```
SC_ESCAPED_STRING++
    ⟨EOF⟩                               fatal⟨ unexpected end of file in a string ⟩
    "                                   ⟨ Finish a bison string 49g ⟩
    ⟨n⟩                                 fatal⟨ unexpected end of line in a string ⟩
```

This code is used in section 43e.

49g ⟨ Finish a `bison` string 49g ⟩ =

```
\STRINGFINISH
def_x next {\yylval{{val\laststring }{val\laststringraw }
    {val\yyfmark }{val\yysmark }}}next
enter(INITIAL)
return_l «string»
```

This code is used in section 49f.

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

⟨Scan a character literal 50a⟩ =

```
SC_ESCAPED_CHARACTER++
  ⟨EOF⟩        fatal⟨unexpected end of file in a literal⟩
  '            ⟨Return an escaped character 50b⟩
  ⟨n⟩          fatal⟨unexpected end of line in a literal⟩
```

This code is used in section 43e.

50b ⟨Return an escaped character 50b⟩ =

```
\STRINGFINISH
def_x next {\yylval{{val\laststring}{val\laststringraw}
    {val\yyfmark}{val\yysmark}}}next
\STRINGFREE
enter(INITIAL)
return_l char
```

This code is used in section 50a.

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

⟨Scan a tag 50c⟩ =

```
SC_TAG++
  >              ⟨Finish a tag 50d⟩
  ([<>]^c | ->)+ \STRINGGROW continue
  <              ⟨Raise nesting level 50e⟩
  ⟨EOF⟩          fatal⟨unexpected end of file in a literal⟩
```

This code is used in section 43e.

50d ⟨Finish a tag 50d⟩ =

```
add\lonesting −1_R
if_ω \lonesting < 0_R
    \STRINGFINISH
    def_x next {\yylval{{val\laststring}{val\laststringraw}
        {val\yyfmark}{val\yysmark}}}next
    \STRINGFREE
    enter(INITIAL)
    \yybreak{return_l <tag>}
else
    \STRINGGROW \yybreak continue
\yycontinue
```

This code is used in section 50c.

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.

⟨Raise nesting level 50e⟩ =

```
\STRINGGROW
add\lonesting 1_R
continue
```

This code is used in section 50c.

50f ⟨Decode escaped characters 50f⟩ =

```
SC_ESCAPED_STRING SC_ESCAPED_CHARACTER++
  \[0–7]{1,3}                \STRINGGROW continue
  \x[0–9abcdefABCDEF]+       \STRINGGROW continue
  \a                         \STRINGGROW continue
  \b                         \STRINGGROW continue
  \f                         \STRINGGROW continue
```

```
\n                                        \STRINGGROW continue
\r                                        \STRINGGROW continue
\t                                        \STRINGGROW continue
\v                                        \STRINGGROW continue
\(" | ' | ? | \)                            ▷ \["'?\] is shorter but confuses xgettext ◁
                                          \STRINGGROW continue
\(u | U [0–9abcdefABCDEF]{4}) [0–9abcdefABCDEF]{4}    \STRINGGROW continue
\(. | ⟨n⟩)                                fatal⟨ invalid character after \: val \yytext ⟩
```

This code is used in section 43e.

51a ⟨ Scan user-code characters and strings 51a ⟩ =

```
SC_CHARACTER SC_STRING++
  ⟨splice⟩ | \⟨splice⟩[⟨n⟩[]]^c          \STRINGGROW continue

SC_CHARACTER++
  '                                      \STRINGGROW enter_x \contextstate continue
  ⟨n⟩                                    fatal⟨ unexpected end of line instead of a character ⟩
  ⟨EOF⟩                                  fatal⟨ unexpected end of file instead of a character ⟩

SC_STRING++
  "                                      \STRINGGROW enter_x \contextstate continue
  ⟨n⟩                                    fatal⟨ unexpected end of line instead of a character ⟩
  ⟨EOF⟩                                  fatal⟨ unexpected end of file instead of a character ⟩
```

This code is used in section 43e.

51b ⟨ Strings, comments etc. found in user code 51b ⟩ =

```
SC_BRACED_CODE SC_PROLOGUE SC_EPILOGUE SC_PREDICATE++
  '            \STRINGGROW \contextstate \YYSTART enter(SC_CHARACTER) continue
  "            \STRINGGROW \contextstate \YYSTART enter(SC_STRING) continue
  /⟨splice⟩*   \STRINGGROW \contextstate \YYSTART enter(SC_COMMENT) continue
  /⟨splice⟩/   \STRINGGROW \contextstate \YYSTART enter(SC_LINE_COMMENT) continue
```

This code is used in section 43e.

51c *Scanning some code in braces (actions, predicates). The initial* `{` *is already eaten.*

⟨ Scan code in braces 51c ⟩ =

```
SC_BRACED_CODE SC_PREDICATE++
  { | <⟨splice⟩%      \STRINGGROW add \lonesting 1_R continue
  %⟨splice⟩>          \STRINGGROW add \lonesting −1_R continue
  <⟨splice⟩<            ▷ Tokenize <<% correctly (as << %) rather than incorrectly (as < <%). ◁
                      \STRINGGROW continue
  ⟨EOF⟩               fatal⟨ unexpected end of line inside braced code ⟩

SC_BRACED_CODE++
  }                   ⟨ Add closing brace to the braced code 51d ⟩

SC_PREDICATE++
  }                   ⟨ Add closing brace to a predicate 52a ⟩
```

This code is used in section 43e.

51d Unlike the original lexer, we do not return the closing brace as part of the braced code.

⟨ Add closing brace to the braced code 51d ⟩ =

```
add \lonesting −1_R
if_ω \lonesting < 0_R
    \STRINGFINISH
    def_x next { \yylval { { val \laststring }{ val \yyfmark }{ val \yysmark } } }next
    enter(INITIAL)
    \yybreak { return_l "{...}"_m }
```

```
else
    \STRINGGROW
    \yybreak continue
\yycontinue
```

This code is used in section 51c.

52a ⟨ Add closing brace to a predicate 52a ⟩ =

```
add\lonesting −1R
ifω \lonesting < 0R
    \STRINGFINISH
    defx next {\yylval{{val\laststring}{val\yyfmark}{val\yysmark}}}next
    enter(INITIAL)
    \yybreak{returnl "%?{...}"m}
else
    \STRINGGROW
    \yybreak continue
\yycontinue
```

This code is used in section 51c.

52b *Scanning some prologue: from* `%{` *(already scanned) to* `%}`.

⟨ Scan prologue 52b ⟩ =

```
SC_PROLOGUE++
  %}                ⟨ Finish braced code 52c ⟩
  ⟨EOF⟩             fatal⟨unexpected end of file inside prologue⟩
```

This code is used in section 43e.

52c ⟨ Finish braced code 52c ⟩ =

```
\STRINGFINISH
defx next {\yylval{{val\laststring}val\postoks{val\yyfmark}{val\yysmark}}}next
enter(INITIAL)
returnl "%{...%}"m
```

This code is used in section 52b.

52d *Scanning the epilogue (everything after the second* ⟨%⟩*, which has already been eaten).*

⟨ Scan the epilogue 52d ⟩ =

```
SC_EPILOGUE++
  ⟨EOF⟩             ⟨ Handle end of file in the epilogue 52e ⟩
```

This code is used in section 43e.

52e ⟨ Handle end of file in the epilogue 52e ⟩ =

```
\ROLLBACKCURRENTTOKEN
\STRINGFINISH
\yylval = \laststring
enter(INITIAL)
returnl epilogue
```

This code is used in section 52d.

52f *By default, grow the string obstack with the input.*

⟨ Add the scanned symbol to the current string 52f ⟩ =

```
SC_COMMENT SC_LINE_COMMENT SC_BRACED_CODE SC_PREDICATE SC_PROLOGUE SC_EPILOGUE SC_STRING SC_CHARACTER
SC_ESCAPED_STRING SC_ESCAPED_CHARACTER+
  .                 ↩

SC_COMMENT SC_LINE_COMMENT SC_BRACED_CODE SC_PREDICATE SC_PROLOGUE SC_EPILOGUE+
  ⟨n⟩               \STRINGGROW continue
```

This code is used in section 43e.

5

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.

⟨ **fip.yy** ch5 ⟩ =

....

⟨ Preamble for the **flex** parser 55c ⟩

....

⟨ Options for **flex** parser 53a ⟩

⟨**union**⟩

....

⟨ Postamble for **flex** parser 63e ⟩

....

⟨ Token definitions for **flex** input parser 54d ⟩

⟨ Productions for **flex** parser 55d ⟩

53a The selection of options for **bison** parsers suitable for **SPLinT** has been discussed earlier so we list them here without further comments.

⟨ Options for **flex** parser 53a ⟩ =

⟨**token table**⟩ ⋆

⟨**parse.trace**⟩ ⋆ (set as ⟨**debug**⟩)

⟨**start**⟩ *goal*

This code is used in sections ch5, 54a, 54b, and 54c.

[1]) For example, each regular expression definition in section 1 must start at the beginning of the line.

54a 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.

⟨`ddp.yy` 54a⟩ =

....

⟨Preamble for the **`flex`** parser 55c⟩

....

⟨Options for **`flex`** parser 53a⟩

⟨**union**⟩

....

⟨Postamble for **`flex`** parser 63e⟩

....

⟨Token definitions for **`flex`** input parser 54d⟩

⟨Exclusive productions for **`flex`** section 1 parser 56c⟩
⟨Productions for **`flex`** section 1 parser 56e⟩

54b A parser for section 2 (rules and actions). This subparser must also use a custom set up for its lexer as discussed above.

⟨`rap.yy` 54b⟩ =

....

⟨Preamble for the **`flex`** parser 55c⟩

....

⟨Options for **`flex`** parser 53a⟩

⟨**union**⟩

....

⟨Postamble for **`flex`** parser 63e⟩

....

⟨Token definitions for **`flex`** input parser 54d⟩

⟨Special **`flex`** section 2 parser productions 57p⟩
⟨Productions for **`flex`** section 2 parser 57r⟩

54c A parser for just the regular expression syntax. A custom lexer initialization must precede the use of this parser, as well.

⟨`rep.yy` 54c⟩ =

....

⟨Preamble for the **`flex`** parser 55c⟩

....

⟨Options for **`flex`** parser 53a⟩

⟨**union**⟩

....

⟨Postamble for **`flex`** parser 63e⟩

....

⟨Token definitions for **`flex`** input parser 54d⟩

⟨Special productions for regular expressions 59i⟩
⟨Rules for **`flex`** regular expressions 59k⟩

54d **Token and state declarations for the `flex` input scanner**

Needless to say, the original grammar used by **`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.

⟨ Token definitions for **flex** input parser 54d ⟩ = 55a

char	**num**	SECTEND	⟨**state**⟩
⟨**xtate**⟩	«name»	PREVCCL	⟨EOF⟩
⟨option⟩	⟨outfile⟩	⟨prefix⟩	⟨yyclass⟩
⟨header⟩	⟨extra type⟩	⟨tables⟩	⟨αn⟩
⟨$\alpha\beta$⟩	⟨ ⟩	⟨↦⟩	⟨0..9⟩
⟨❧⟩	⟨a..z⟩	⟨➾⟩	⟨.⟩
⟨␣⟩	⟨A..Z⟩	⟨0..Z⟩	⟨¬αn⟩
⟨¬$\alpha\beta$⟩	⟨¬ ⟩	⟨¬↦⟩	⟨¬0..9⟩
⟨¬❧⟩	⟨¬a..z⟩	⟨¬➾⟩	⟨¬.⟩
⟨¬␣⟩	⟨¬A..Z⟩	⟨¬0..Z⟩	

⟨**left**⟩ \ ∪

See also sections 55a and 55b.

This code is used in sections ch5, 54a, 54b, and 54c.

55a 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.

⟨ Token definitions for **flex** input parser 54d ⟩ + = 54d 55b

⟨top⟩	⟨pointer*⟩	⟨array⟩	⟨def⟩
⟨$\mathtt{def}_{\rm re}$⟩	⟨other⟩	⟨deprecated⟩	

55b *POSIX and **AT&T** **lex** place the precedence of the repeat operator, {}, below that of concatenation. Thus, **ab{3}** is **ababab**. Most other POSIX utilities use an* Extended Regular Expression (ERE) *precedence that has the repeat operator higher than concatenation. This causes **ab{3}** to yield **abbb**.*

*In order to support the POSIX and **AT&T** precedence and the **flex** precedence we define two token sets for the begin and end tokens of the repeat operator, $\{_{\rm p}$ and $\}_{\rm p}$. 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 **flex** to parse scanner files as per the **AT&T** and POSIX-mandated behavior.*

⟨ Token definitions for **flex** input parser 54d ⟩ + = 55a

$\{_{\rm p}$	$\}_{\rm p}$	$\{_{\rm f}$	$\}_{\rm f}$

55c The grammar for flex input

The original grammar has been carefully split into sections to facilitate the assembly of various subparsers in the **flex**'s stack. Neither the **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 C preamble.

⟨ Preamble for the **flex** parser 55c ⟩ =

This code is used in sections ch5, 54a, 54b, and 54c.

55d ⟨ Productions for **flex** parser 55d ⟩ = 56b

goal : *initlex* $sect_1$ *sect1end* $sect_2$ *initforrule* ⟨ Assemble a **flex** input file 56a ⟩

sect1end : SECTEND ⟨ Copy the value 63b ⟩

initlex : ∘

See also section 56b.

This code is used in section ch5.

56a ⟨ Assemble a `flex` input file 56a ⟩ =

`\finishlist{` val Υ_4 `}`

$\Upsilon \leftarrow$ ⟨val Υ_2 ^{nx}`\executelist{` val Υ_4 `}`⟩

This code is used in section 55d.

56b ⟨ Productions for `flex` parser 55d ⟩ + =

⟨ Productions for `flex` section 1 parser 56e ⟩

⟨ Productions for `flex` section 2 parser 57r ⟩

56c ⟨ Exclusive productions for `flex` section 1 parser 56c ⟩ =

goal : *sect*$_1$ ⟨ Assemble a `flex` section 1 file 56d ⟩

This code is used in section 54a.

56d ⟨ Assemble a `flex` section 1 file 56d ⟩ =

Ω`\expandafter{` val Υ_1 `}`

This code is used in section 56c.

56e ⟨ Productions for `flex` section 1 parser 56e ⟩ = 56o ▽

sect$_1$:

sect$_1$ *startconddecl namelist*$_1$ ⟨ Add start condition declarations 56f ⟩

sect$_1$ *options* ⟨ Add options to section 1 56g ⟩

∘ ⟨ Create an empty section 1 56h ⟩

error ⟨ Report an error in section 1 and quit 56i ⟩

startconddecl :

⟨**state**⟩ ⟨ Prepare a state declaration 56j ⟩

⟨**xtate**⟩ ⟨ Prepare an exclusive state declaration 56k ⟩

namelist$_1$:

namelist$_1$ «name» ⟨ Add a name to a list 56l ⟩

«name» ⟨ Start a *namelist*$_1$ with a name 56m ⟩

error ⟨ Report an error in *namelist*$_1$ and quit 56n ⟩

See also section 56o.

This code is used in sections 54a and 56b.

56f ⟨ Add start condition declarations 56f ⟩ =

$\Upsilon \leftarrow$ ⟨val Υ_1 ^{nx}`\flscondecl` val Υ_2`{` val Υ_3 `}`⟩

This code is used in section 56e.

56g ⟨ Add options to section 1 56g ⟩ =

$\Upsilon \leftarrow$ ⟨val Υ_1 val Υ_2⟩

This code is used in section 56e.

56h ⟨ Create an empty section 1 56h ⟩ =

$\Upsilon \leftarrow$ ⟨⟩

This code is used in section 56e.

56i ⟨ Report an error in section 1 and quit 56i ⟩ =

`\yyerror`

This code is used in section 56e.

56j ⟨ Prepare a state declaration 56j ⟩ =

$\Upsilon \leftarrow$ ⟨`{s}`val Υ_1⟩

This code is used in section 56e.

56k ⟨ Prepare an exclusive state declaration 56k ⟩ =

$\Upsilon \leftarrow$ ⟨`{x}`val Υ_1⟩

This code is used in section 56e.

56l ⟨ Add a name to a list 56l ⟩ =

$\Upsilon \leftarrow$ ⟨val Υ_1 ^{nx}`\flnamesep{ }{ }`^{nx}`\flname` val Υ_2⟩

This code is used in section 56e.

56m ⟨ Start a *namelist*$_1$ with a name 56m ⟩ =

$\Upsilon \leftarrow$ ⟨^{nx}`\flname` val Υ_1⟩

This code is used in section 56e.

56n ⟨ Report an error in *namelist*$_1$ and quit 56n ⟩ =

`\yyerror`

This code is used in section 56e.

56o ⟨ Productions for `flex` section 1 parser 56e ⟩ + = △ 56e

options :

⟨`option`⟩ *optionlist* ⟨ Start an options list 57a ⟩

⟨`pointer*`⟩ ⟨ Add a pointer option 57b ⟩

⟨`array`⟩ ⟨ Add an array option 57c ⟩

⟨`top`⟩ `\n` ⟨ Add a ⟨`top`⟩ directive 57d ⟩

⟨`def`⟩ ⟨`def`$_{re}$⟩ ⟨ Add a regular expression definition 57e ⟩

⟨`deprecated`⟩ ⟨ Output a deprecated option 57o ⟩

optionlist : *optionlist option* | ∘ . . . | ⟨ Make an empty option list 57g ⟩

option :

⟨outfile⟩ = «name» ⟨ Record the name of the output file 57h ⟩
⟨extra type⟩ = «name» ⟨ Declare an extra type 57i ⟩
⟨prefix⟩ = «name» ⟨ Declare a prefix 57j ⟩
⟨yyclass⟩ = «name» ⟨ Declare a class 57k ⟩
⟨header⟩ = «name» ⟨ Declare the name of a header 57l ⟩
⟨tables⟩ = «name» ⟨ Declare the name for the tables 57m ⟩
⟨other⟩ ⟨ Output a non-parametric option 57n ⟩

57a ⟨ Start an options list 57a ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\floptions{` val Υ_2 `}`⟩
This code is used in section 56o.

57b ⟨ Add a pointer option 57b ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flptropt` val Υ_1⟩
This code is used in section 56o.

57c ⟨ Add an array option 57c ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flarrayopt` val Υ_1⟩
This code is used in section 56o.

57d ⟨ Add a ⟨top⟩ directive 57d ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\fltopopt` val Υ_1 val Υ_2⟩
This code is used in section 56o.

57e ⟨ Add a regular expression definition 57e ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flredef` val Υ_1 val Υ_2⟩
This code is used in section 56o.

57f ⟨ Add an option to a list 57f ⟩ =
$\Upsilon \leftarrow$ ⟨val Υ_1 val Υ_2⟩
This code is used in section 56o.

57g ⟨ Make an empty option list 57g ⟩ =
$\Upsilon \leftarrow$ ⟨⟩
This code is used in section 56o.

57h ⟨ Record the name of the output file 57h ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{file}`val Υ_3⟩
This code is used in section 56o.

57i ⟨ Declare an extra type 57i ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{xtype}`val Υ_3⟩
This code is used in section 56o.

57j ⟨ Declare a prefix 57j ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{prefix}`val Υ_3⟩
This code is used in section 56o.

57k ⟨ Declare a class 57k ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{yyclass}`val Υ_3⟩
This code is used in section 56o.

57l ⟨ Declare the name of a header 57l ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{header}`val Υ_3⟩
This code is used in section 56o.

57m ⟨ Declare the name for the tables 57m ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{tables}`val Υ_3⟩
This code is used in section 56o.

57n ⟨ Output a non-parametric option 57n ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{other}`val Υ_1⟩
This code is used in section 56o.

57o ⟨ Output a deprecated option 57o ⟩ =
$\Upsilon \leftarrow$ ⟨^{nx}`\flopt{deprecated}`val Υ_1⟩
This code is used in section 56o.

57p ⟨ Special `flex` section 2 parser productions 57p ⟩ =
goal : *sect*$_2$ ⟨ Output section 2 57q ⟩
This code is used in section 54b.

57q ⟨ Output section 2 57q ⟩ =
`\finishlist{` val Υ_1 `}`
Ω`\expandafter{\expandafter\executelist\expandafter{` val Υ_1 `}}`
This code is used in section 57p.

57r This portion of the grammar was changed to make it possible to read the action code.
⟨ Productions for `flex` section 2 parser 57r ⟩ = 58e

***sect*$_2$** :

sect$_2$ *scon initforrule flexrule* `\n \n` ⟨ Add a rule to section 2 58a ⟩
sect$_2$ *scon* `{` *sect*$_2$ `}` ⟨ Add a group of rules to section 2 58b ⟩

○ ⟨ Start an empty section 2 58c ⟩
sect$_2$ \n ⟨ Add a bare action 58d ⟩
initforrule : ○ \flin@ruletrue **continue**

See also sections 58e and 59h.
This code is used in sections 54b and 56b.

58a The production below describes the most typical way a regular expression is assigned an action. The redundant term *initforrule* is a standard `bison` trick to make sure that the appropriate initializations happen at the right time.

sect₂ : *sect*$_2$ *scon initforrule flexrule* \n \n ⟨ Add a rule to section 2 58a ⟩

The original production has been modified so that the pretty printing parser has a chance to consume the action code. The second `\n` is output by the action processing code.

⟨ Add a rule to section 2 58a ⟩ =
\ifflcontinued@action
  v_b ← ⟨ \flactionc ⟩
else
  v_b ← ⟨ \flaction ⟩
fi
v_a\expandafter { \astformat@flaction }   ▷ capture the formatting action ◁
Υ ← ⟨val Υ_1⟩
\appendtolistx { val Υ_1 }{ val v_b ← ⟨ val Υ_2 ⟩{ val Υ_4 }val Υ_5 val Υ_6{ val v_a } }
let \astformat@flaction ∅   ▷ reset the format ◁

This code is used in section 57r.

58b For convenience, rules that are active in the same set of states may be grouped together. This pattern is the subject of the next production.

sect₂ : *sect*$_2$ *scon* { *sect*$_2$ } ⟨ Add a group of rules to section 2 58b ⟩

The original parser ignores the braces while the pretty printing parser uses the pointers associated with the braces to collect and process the accumulated stash. This is how comments and `CWEB` section references are typeset.

⟨ Add a group of rules to section 2 58b ⟩ =
Υ ← ⟨val Υ_1⟩
\finishlist { val Υ_4 }
\appendtolistx { val Υ_1 }{ ^{nx}\flactiongroup { val Υ_2 }val Υ_3{ ^{nx}\executelist { val Υ_4 } }val Υ_5 }

This code is used in section 57r.

58c Simple left recursive terms like *sect*$_2$ are very suitable for being implemented as a list (see the macros in `yycommon.sty` for the details on the list implementation). The 'type' of *sect*$_2$ is a (symbolic pointer to a) list of items built up from an empty initial list. This production initializes the list (with the name identical to the terminal on the left hand side of the production) and updates the list name (rather the name's prefix) for future invocations of this action.

⟨ Start an empty section 2 58c ⟩ =
\initlist { \secttwoprefix sect2 }
Υ ← ⟨\secttwoprefix sect2⟩
def$_x$ \secttwoprefix { \secttwoprefix . }

This code is used in section 57r.

58d ⟨ Add a bare action 58d ⟩ =
Υ ← ⟨val Υ_1⟩
\appendtolistx { val Υ_1 }{ ^{nx}\flbareaction val Υ_2 }

This code is used in section 57r.

58e ⟨ Productions for `flex` section 2 parser 57r ⟩ + = 57r 59h
scon_stk_ptr : ○

scon :
< *scon_stk_ptr* *namelist*$_2$ > ⟨ Create a list of start conditions 59a ⟩
< * > ⟨ Create a universal start condition 59b ⟩
∘ ⟨ Create an empty start condition 59c ⟩

namelist$_2$:
namelist$_2$, *sconname* ⟨ Add a start condition to a list 59d ⟩
sconname ⟨ Start a list with a start condition name 59e ⟩
error ⟨ Report an error in a start condition list 59f ⟩

sconname : «name» ⟨ Make a «name» into a start condition 59g ⟩

59a Start conditions are just names. The data structure that is output has location pointers for the streams to enable interaction with CWEB. These pointers are in turn the values of the angle bracket tokens that enclose the list of start conditions.

Start condition lists may be collected in their own sections, while the list itself may be followed by a comment. The pointers mentioned above are used to typeset the comments and section references.

⟨ Create a list of start conditions 59a ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flsconlist{` val Υ_1 `}{` val Υ_3 `}{` val Υ_4 `}`$\rangle$
This code is used in section 58e.

59b ⟨ Create a universal start condition 59b ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flsconuniv` val $\Upsilon_3\rangle$
This code is used in section 58e.

59c ⟨ Create an empty start condition 59c ⟩ =
$\Upsilon \leftarrow \langle\rangle$
This code is used in section 58e.

59d ⟨ Add a start condition to a list 59d ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\mathrm{nx}}$`\flnamesep` val $\Upsilon_2$val $\Upsilon_3\rangle$
This code is used in section 58e.

59e ⟨ Start a list with a start condition name 59e ⟩ =
⟨ Copy the value 63b ⟩
This code is used in section 58e.

59f ⟨ Report an error in a start condition list 59f ⟩ =
`\yyerror`
This code is used in section 58e.

59g ⟨ Make a «name» into a start condition 59g ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flname` val $\Upsilon_1\rangle$
This code is used in section 58e.

59h The syntax of regular expressions

The productions in this section define the syntax of `flex` regular expressions in detail. The same productions are used for parsing isolated regular expressions (e.g. to present example code). A few of these productions have been modified to suit the needs of the pretty printing parser.

⟨ Productions for `flex` section 2 parser 57r ⟩ += △ 58e
⟨ Rules for `flex` regular expressions 59k ⟩

59i ⟨ Special productions for regular expressions 59i ⟩ =
goal : *flexrule* ⟨ Output a regular expression 59j ⟩
This code is used in section 54c.

59j The parsed regular expression is output in the `\table` register. It is important to ensure that whenever this parser is used inside another parser that uses `\table` for output, the changes to this register stay local. The `\frexproc` macro in `yyunion.sty` ensures that all the changes are local to the parsing macro.

⟨ Output a regular expression 59j ⟩ =
$\Omega\Upsilon_1$
This code is used in section 59i.

59k Regular expressions are parsed using the following productions. There are two major cases: rules active only at the beginning of the line, and the rest. From the typesetting parser's point of view, there is not much difference between the two (certainly not enough to justify singling out the rules at the beginning of the line into their own production) but it was decided to keep the original grammar rules for consistency.

⟨ Rules for `flex` regular expressions 59k ⟩ = 60e ▽

flexrule :
- ^ *rule* ⟨Match a rule at the start of the line 60a⟩
- *rule* ⟨Match an ordinary rule 60c⟩
- ⟨`EOF`⟩ ⟨Match an end of file 60b⟩
- *error* ⟨Report an error and quit 60d⟩

See also sections 60e, 61a, 61g, 62h, and 63a.

This code is used in sections 54c and 59h.

60a ⟨Match a rule at the start of the line 60a⟩ =
v_a`\expandafter{\astformat@flrule}`
let `\astformat@flrule` ∅
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\flbolrule{` val Υ_2 `}{` val v_a `}`⟩

This code is used in section 59k.

60b ⟨Match an end of file 60b⟩ =
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\fleof` val Υ_1⟩

This code is used in section 59k.

60c ⟨Match an ordinary rule 60c⟩ =
v_a`\expandafter{\astformat@flrule}`
let `\astformat@flrule` ∅
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\flrule{` val Υ_1 `}{` val v_a `}`⟩

This code is used in section 59k.

60d ⟨Report an error and quit 60d⟩ =
`\yyerror`

This code is used in section 59k.

60e Another broad overview of regular expression types before diving into the details of various operations. Note that the only trailing context that `SPLinT` output lexer can process is the end of line (`$`) due to the way the scanner routine is written. It does not affect its ability to pretty print the appropriate rules (for a lexer that is produced by `flex` itself, for example).

⟨Rules for `flex` regular expressions 59k⟩ += 59k 61a

rule :
- re_2 *re* ⟨Match a regular expression with a trailing context 60f⟩
- re_2 *re* `$` ⟨Disallow a repeated trailing context 60g⟩
- *re* `$` ⟨Match a regular expression at the end of the line 60h⟩
- *re* ⟨Match an ordinary regular expression 60i⟩

re :
- *re* `|` *series* ⟨Match a sequence of alternatives 60j⟩
- *series* ⟨Match a sequence of singletons 60k⟩

$\boldsymbol{re_2}$: *re* `/` ⟨Prepare to match a trailing context 60l⟩

60f ⟨Match a regular expression with a trailing context 60f⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a \pi_3(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\flretrail{` val v_a `}{` val v_b `}{` val Υ_2 `}`⟩

This code is used in section 60e.

60g ⟨Disallow a repeated trailing context 60g⟩ =
`\yyerror`

This code is used in section 60e.

60h ⟨Match a regular expression at the end of the line 60h⟩ =
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\flreateol{` val Υ_1 `}`val Υ_2⟩

This code is used in section 60e.

60i ⟨Match an ordinary regular expression 60i⟩ =
⟨Copy the value 63b⟩

This code is used in section 60e.

60j ⟨Match a sequence of alternatives 60j⟩ =
$\Upsilon \leftarrow$ ⟨val $\Upsilon_1$$^{\rm nx}$`\flor` val $\Upsilon_2$val Υ_3⟩

This code is used in section 60e.

60k ⟨Match a sequence of singletons 60k⟩ =
⟨Copy the value 63b⟩

This code is used in section 60e.

60l ⟨Prepare to match a trailing context 60l⟩ =
$\Upsilon \leftarrow$ ⟨$^{\rm nx}$`\fltrail{` val Υ_1 `}{` val Υ_2 `}`⟩

This code is used in section 60e.

61a **Atoms**

Every regular expression is assembled of atomic subexpressions, each of which may be modified by an repetition operator that establishes how many times a given pattern can repeat to stay part of the original atom. New atomic expressions (or *singletons* as they are called below) can be formed the usual way, by enclosing a regular expression in parentheses.

As explained above, braced repetition operators may have different binding strengths, depending on the options supplied to `flex`. The pretty printing in both cases is identical as only the application scopes of the operator differ, and not its meaning.

⟨ Rules for `flex` regular expressions 59k ⟩ += 60e 61g

series :

series singleton	⟨ Extend a series by a singleton 61b ⟩
singleton	⟨ Match a singleton 61c ⟩
series $\{_\mathrm{p}$ **num** , **num** $\}_\mathrm{p}$	⟨ Match a series of specific length 61d ⟩
series $\{_\mathrm{p}$ **num** , $\}_\mathrm{p}$	⟨ Match a series of minimal length 61e ⟩
series $\{_\mathrm{p}$ **num** $\}_\mathrm{p}$	⟨ Match a series of exact length 61f ⟩

61b ⟨ Extend a series by a singleton 61b ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1 \mathrm{val}\, \Upsilon_2 \rangle$
This code is used in section 61a.

61c ⟨ Match a singleton 61c ⟩ =
⟨ Copy the value 63b ⟩
This code is used in section 61a.

61d ⟨ Match a series of specific length 61d ⟩ =
⟨ Create a series of specific length 61k ⟩
This code is used in section 61a.

61e ⟨ Match a series of minimal length 61e ⟩ =
⟨ Create a series of minimal length 61l ⟩
This code is used in section 61a.

61f ⟨ Match a series of exact length 61f ⟩ =
⟨ Create a series of exact length 62a ⟩
This code is used in section 61a.

61g ⟨ Rules for `flex` regular expressions 59k ⟩ += 61a 62h

singleton :

singleton *	⟨ Create a lazy series match 61h ⟩
singleton +	⟨ Create a nonempty series match 61i ⟩
singleton ?	⟨ Create a possible single match 61j ⟩
singleton $\{_\mathrm{f}$ **num** , **num** $\}_\mathrm{f}$	⟨ Create a series of specific length 61k ⟩
singleton $\{_\mathrm{f}$ **num** , $\}_\mathrm{f}$	⟨ Create a series of minimal length 61l ⟩
singleton $\{_\mathrm{f}$ **num** $\}_\mathrm{f}$	⟨ Create a series of exact length 62a ⟩
.	⟨ Match (almost) any character 62b ⟩
fullccl	⟨ Match a character class 62c ⟩
`PREVCCL`	⟨ Match a `PREVCCL` 62d ⟩
" *string* "	⟨ Match a string 62e ⟩
(*re*)	⟨ Match an atom 62f ⟩
char	⟨ Match a specific character 62g ⟩

61h ⟨ Create a lazy series match 61h ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flrepeat`{ val Υ_1 }⟩
This code is used in section 61g.

61i ⟨ Create a nonempty series match 61i ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flrepeatstrict`{ val Υ_1 }⟩
This code is used in section 61g.

61j ⟨ Create a possible single match 61j ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flrepeatonce`{ val Υ_1 }⟩
This code is used in section 61g.

61k ⟨ Create a series of specific length 61k ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flrepeatnm`{ val Υ_1 }{ val Υ_3 }{ val Υ_5 }⟩
This code is used in sections 61d and 61g.

61l ⟨ Create a series of minimal length 61l ⟩ =
$\Upsilon \leftarrow \langle^{\mathrm{nx}}$`\flrepeatgen`{ val Υ_1 }{ val Υ_3 }⟩
This code is used in sections 61e and 61g.

62a ⟨ Create a series of exact length 62a ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flrepeatn{` val Υ_1 `}{` val Υ_3 `}`$\rangle$
This code is used in sections 61f and 61g.

62b ⟨ Match (almost) any character 62b ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\fldot` val $\Upsilon_1\rangle$
This code is used in section 61g.

62c ⟨ Match a character class 62c ⟩ =
⟨ Copy the value 63b ⟩
This code is used in section 61g.

62d ⟨ Match a `PREVCCL` 62d ⟩ =
⟨ Copy the value 63b ⟩
This code is used in section 61g.

62e ⟨ Match a string 62e ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flstring{` val Υ_1 `}{` val Υ_2 `}{` val Υ_3 `}`$\rangle$
This code is used in section 61g.

62f ⟨ Match an atom 62f ⟩ =
v_a`\expandafter{\astformat@flparens}`
let `\astformat@flparens` ∅
$\Upsilon \leftarrow \langle^{nx}$`\flparens{` val Υ_1 `}{` val Υ_2 `}{` val Υ_3 `}{` val v_a `}`$\rangle$
This code is used in section 61g.

62g ⟨ Match a specific character 62g ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flchar` val $\Upsilon_1\rangle$
This code is used in section 61g.

62h **Characters**

Several facilities are available to specify sets of characters, including built-in characters classes such as *whitespace*, *printable characters*, *alphanumerics*, etc. Some simple boolean operaions are also supported to make specifying character classes more efficient.

⟨ Rules for `flex` regular expressions 59k ⟩ += 61g 63a

fullccl :
fullccl \ *braceccl* — ⟨ Subtract a character class 62i ⟩
fullccl ∪ *braceccl* — ⟨ Create a union of character classes 62j ⟩
braceccl — ⟨ Turn a basic character class into a character class 62k ⟩

braceccl :
`[` *ccl* `]` — ⟨ Create a character class 62l ⟩
`[ ^` *ccl* `]` — ⟨ Complement a character class 62m ⟩

ccl :
ccl **char** – **char** — ⟨ Add a range to a character class 62n ⟩
ccl **char** — ⟨ Add a character to a character class 62o ⟩
ccl *ccl_expr* — ⟨ Add an expression to a character class 62p ⟩
∘ — ⟨ Create an empty character class 62q ⟩

This code is used in section 62h.

62i ⟨ Subtract a character class 62i ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flccldiff{` val Υ_1 `}{` val Υ_3 `}`$\rangle$
This code is used in section 62h.

62j ⟨ Create a union of character classes 62j ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flcclunion{` val Υ_1 `}{` val Υ_3 `}`$\rangle$
This code is used in section 62h.

62k ⟨ Turn a basic character class into a character class 62k ⟩ =
⟨ Copy the value 63b ⟩
This code is used in section 62h.

62l ⟨ Create a character class 62l ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flbraceccl{` val Υ_1 `}{` val Υ_2 `}{` val Υ_3 `}`$\rangle$
This code is used in section 62h.

62m ⟨ Complement a character class 62m ⟩ =
$\Upsilon \leftarrow \langle^{nx}$`\flbracecclneg`
`{` val Υ_1 `}{` val Υ_3 `}{` val Υ_4 `}`$\rangle$

62n ⟨ Add a range to a character class 62n ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}$`\flcclrnge`
`{`^{nx}`\flchar` val Υ_2 `}{`^{nx}`\flchar` val Υ_4 `}`$\rangle$
This code is used in section 62h.

62o ⟨ Add a character to a character class 62o ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}$`\flchar` val $\Upsilon_2\rangle$
This code is used in section 62h.

62p ⟨ Add an expression to a character class 62p ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{nx}$`\flcclexpr` val $\Upsilon_2\rangle$
This code is used in section 62h.

62q ⟨ Create an empty character class 62q ⟩ =
$\Upsilon \leftarrow \langle\rangle$
This code is used in section 62h.

63a **Special character classes**

Various character classes are predefined in `flex`. These include alphabetic and alphanumeric characters, digits, blank characters, upper and lower case characters, etc.

⟨ Rules for `flex` regular expressions 59k ⟩ + = 62h

ccl_expr :

⟨αn⟩ | ⟨$\alpha\beta$⟩ | ⟨ ⟩ | ⟨↦⟩ | ⟨0..9⟩ | ⟨❧⟩ . . . | ⟨ Copy the value 63b ⟩
⟨a..z⟩ | ⟨➭⟩ | ⟨.⟩ | ⟨␣⟩ | ⟨0..Z⟩ | ⟨A..Z⟩ . . . | ⟨ Copy the value 63b ⟩
⟨¬αn⟩ | ⟨¬$\alpha\beta$⟩ | ⟨¬ ⟩ | ⟨¬↦⟩ | ⟨¬0..9⟩ | ⟨¬❧⟩ . . . | ⟨ Copy the value 63b ⟩
⟨¬➭⟩ | ⟨¬.⟩ | ⟨¬␣⟩ | ⟨¬0..Z⟩ | ⟨¬a..z⟩ | ⟨¬A..Z⟩ . . . | ⟨ Copy the value 63b ⟩

string : *string* **char** | ∘ . . . | ⟨ Make an empty regular expression string 63d ⟩

63b ⟨ Copy the value 63b ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1 \rangle$

This code is used in sections 55d, 59e, 60i, 60k, 61c, 62c, 62d, 62k, and 63a.

63c ⟨ Extend a `flex` string by a character 63c ⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\, \Upsilon_1{}^{\mathrm{nx}}$`\flchar` $\mathrm{val}\, \Upsilon_2 \rangle$

This code is used in section 63a.

63d ⟨ Make an empty regular expression string 63d ⟩ =
$\Upsilon \leftarrow \langle\rangle$

This code is used in section 63a.

63e The postamble is empty for now.

⟨ Postamble for `flex` parser 63e ⟩ =

This code is used in sections ch5, 54a, 54b, and 54c.

6
The lexer for `flex` syntax

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.

⟨ **`fil.ll`** ch6 ⟩ =

....

⟨ Preamble for **`flex`** lexer 65b ⟩

....

⟨ Options for **`flex`** input lexer 66a ⟩
⟨ Output file for **`flex`** input lexer 66b ⟩
⟨ State definitions for **`flex`** input lexer 66d ⟩
⟨ Definitions for **`flex`** input lexer 66e ⟩

⟨ Postamble for **`flex`** input lexer 67a ⟩
⟨ Common patterns for **`flex`** lexer 67b ⟩
⟨ Patterns for **`flex`** lexer 68c ⟩

⟨ Auxilary code for **`flex`** lexer 78d ⟩

65a Bootstrap lexer.

⟨ **`ssfs.ll`** 65a ⟩ =

....

⟨ Preamble for **`flex`** lexer 65b ⟩

....

⟨ Options for **`flex`** input lexer 66a ⟩
⟨ Output file for the bootstrap **`flex`** lexer 66c ⟩
⟨ Definitions for **`flex`** input lexer 66e ⟩

⟨ Common patterns for **`flex`** lexer 67b ⟩
⟨ Catchall rule for the bootstrap lexer 78f ⟩

⟨ Auxilary code for the bootstrap **`flex`** lexer 79a ⟩

65b ⟨ Preamble for **`flex`** lexer 65b ⟩ =

This code is used in sections ch6 and 65a.

66a 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.

⟨ Options for `flex` input lexer 66a ⟩ =

```
⟨option⟩f    caseless
⟨option⟩f    nodefault
⟨option⟩f    stack
⟨option⟩f    noyy_top_state
⟨option⟩f    nostdinit
⟨option⟩f    bison-bridge
⟨option⟩f    noyywrap
⟨option⟩f    nounput
⟨option⟩f    noinput
⟨option⟩f    reentrant
⟨option⟩f    debug
⟨option⟩f    stack
```

This code is used in sections ch6 and 65a.

66b ⟨ Output file for `flex` input lexer 66b ⟩ =

```
⟨output to⟩f    "fil.c"
```

This code is used in section ch6.

66c ⟨ Output file for the bootstrap `flex` lexer 66c ⟩ =

```
⟨output to⟩f    "ssfs.c"
```

This code is used in section 65a.

66d **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 66d ⟩ =

⟨**state-x**⟩f **SECT**$_2$ **SECT**$_2$ **PROLOG SECT**$_3$ **CODEBLOCK PICKUPDEF SC CARETISBOL NUM QUOTE**
⟨**state-x**⟩f **FIRSTCCL CCL ACTION RECOVER COMMENT ACTION_STRING PERCENT_BRACE_ACTION**
⟨**state-x**⟩f **OPTION LINEDIR CODEBLOCK_MATCH_BRACE**
⟨**state-x**⟩f **GROUP_WITH_PARAMS**
⟨**state-x**⟩f **GROUP_MINUS_PARAMS**
⟨**state-x**⟩f **EXTENDED_COMMENT**
⟨**state-x**⟩f **COMMENT_DISCARD**

This code is used in section ch6.

66e Somewhat counterintuitively, `flex` definitions do not *always* have to be fully formed regular expressions. For example, after

```
⟨BOGUS⟩    ^[a-
```

one can form the following action:

```
⟨BOGUS⟩t]    ;
```

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 66e ⟩ =

```
⟨␣+⟩    [⟨ ⟩]+
⟨␣*⟩    [⟨ ⟩]*
```

⟨NOT_WS⟩	[⟨ ⟩⟨r⟩⟨n⟩]^c
⟨⟵⟩	⟨r⟩?⟨n⟩
⟨NAME⟩	([⟨αβ⟩_] [⟨αn⟩_-]$_*$)
⟨NOT_NAME⟩	[⟨αβ⟩_*⟨n⟩]^c$_+$
⟨SCNAME⟩	⟨NAME⟩
⟨ESCSEQ⟩	(\([⟨n⟩]^c | [0–7]$_{\{1,3\}}$ | x [⟨0..Z⟩]$_{\{1,2\}}$))
⟨FIRST_CCL_CHAR⟩	([\⟨n⟩]^c | ⟨ESCSEQ⟩)
⟨CCL_CHAR⟩	([\⟨n⟩]]^c | ⟨ESCSEQ⟩)
⟨CCL_EXPR⟩	([: ^$_?$ [⟨αβ⟩]$_+$:])
⟨LEXOPT⟩	[porkacne]
⟨M4QSTART⟩	[[
⟨M4QEND⟩	]]

This code is used in sections ch6 and 65a.

67a ⟨ Postamble for `flex` input lexer 67a ⟩ =

This code is used in section ch6.

67b **Regular expressions for `flex` input scanner**

The code below treats ⟨`pointer`⟩ and ⟨`array`⟩ the same way it treats ⟨`option`⟩ while typesetting.

⟨ Common patterns for `flex` lexer 67b ⟩ =

INITIAL^{++}

⊣ ⟨␣+⟩	\flindented@codetrue **enter**(CODEBLOCK) **continue**
⊣ /*	**push state**(COMMENT) **continue**
⊣ #⟨␣*⟩line⟨␣+⟩	**push state**(LINEDIR) **continue**
⊣ %s ⟨NAME⟩?	**return**$_p$ ⟨**state**⟩
⊣ %x ⟨NAME⟩?	**return**$_p$ ⟨**xtate**⟩
⊣ %{ .* ⟨⟵⟩	⟨ Start a C code section 67c ⟩
⊣ %top [⟨ ⟩]* { [⟨ ⟩]* ⟨⟵⟩	⟨ Begin the ⟨top⟩ directive 67d ⟩
⊣ %top .*	**fatal**⟨ malformed '% top' directive ⟩
⟨␣+⟩	; ▷ discard ◁
⊣ %% .*	⟨ Start section 2 68a ⟩
⊣ %pointer .* ⟨⟵⟩	\flinc@linenum **return**$_l$ ⟨pointer*⟩
⊣ %array .* ⟨⟵⟩	\flinc@linenum **return**$_l$ ⟨array⟩
⊣ %option	**enter**(OPTION) **return**$_l$ ⟨option⟩
⊣ %⟨LEXOPT⟩⟨␣*⟩ [⟨0..9⟩]* ⟨␣*⟩⟨⟵⟩	\flinc@linenum **return**^{opt} ⟨deprecated⟩
⊣ %⟨LEXOPT⟩⟨␣+⟩ .* ⟨⟵⟩	\flinc@linenum **return**^{opt} ⟨deprecated⟩
⊣ %[porksexcan{}]^c .*	**fatal**⟨ unrecognized '%' directive: val \yytext ⟩
⊣ ⟨NAME⟩	⟨ Copy the name and start a definition 68b ⟩
⟨SCNAME⟩	\RETURNNAME
⊣ ⟨␣*⟩⟨⟵⟩	\flinc@linenum **continue** ▷ allows blank lines in section 1 ◁
⟨␣*⟩⟨⟵⟩	\flinc@linenum **continue** ▷ maybe end of comment line ◁

This code is used in sections ch6 and 65a.

67c ⟨ Start a C code section 67c ⟩ =

```
\flinc@linenum
\flindented@codefalse enter(CODEBLOCK)
continue
```

This code is used in section 67b.

67d Ignore setting *brace_start_line* as it is only used internally to report errors.

⟨ Begin the ⟨`top`⟩ directive 67d ⟩ =

```
\flinc@linenum
def \flbrace@depth { 1 }
push state(CODEBLOCK_MATCH_BRACE) continue
```

This code is used in section 67b.

68a ⟨ Start section 2 68a ⟩ =
def \flsectnum { 2 }**def** \flbracelevel { 0 }
enter(SECT$_2$ PROLOG) **return**$_p$ SECTEND
This code is used in section 67b.

68b ⟨ Copy the name and start a definition 68b ⟩ =
\fldidadeffalse **enter**(PICKUPDEF)
return$_{vp}$ ⟨def⟩
This code is used in section 67b.

68c ⟨ Patterns for flex lexer 68c ⟩ = 69a

COMMENT^{++}

*/	**continue**
*	**continue**
⟨M4QSTART⟩	**continue**
⟨M4QEND⟩	**continue**
[*⟨n⟩]^c	**continue**
⟨↩⟩	\flinc@linenum **continue**

COMMENT_DISCARD^{++} ▷ This is the same as COMMENT, but is discarded rather than output. ◁

*/	**continue**
*	**continue**
[*⟨n⟩]^c	**continue**
⟨↩⟩	\flinc@linenum **continue**

EXTENDED_COMMENT^{++}

)	**continue**
[⟨n⟩)]^c+	**continue**
⟨↩⟩	\flinc@linenum **continue**

LINEDIR^{++}

⟨n⟩	**continue**
[⟨0..9⟩]+	\fllinenum = \number val \yytext **continue**
" ["⟨n⟩]^c* "	**continue** ▷ ignore the file name in the line directives ◁
.	**continue** ▷ ignore spurious characters ◁

CODEBLOCK^{++}

⊣ %} .* ⟨↩⟩	\flinc@linenum **enter**(INITIAL) **continue**
⟨M4QSTART⟩	**continue**
⟨M4QEND⟩	**continue**
.	**continue**
⟨↩⟩	\flinc@linenum \ifflindented@code **enter**(INITIAL) **fi** **continue**

CODEBLOCK_MATCH_BRACE^{++}

}	⟨ Pop state if code braces match 68d ⟩
{	\flinc \flbrace@depth **continue**
⟨↩⟩	\flinc@linenum **continue**
⟨M4QSTART⟩	**continue**
⟨M4QEND⟩	**continue**
[{}⟨r⟩⟨n⟩]^c	**continue**
⟨EOF⟩	**fatal**⟨ Unmatched '{' ⟩

See also sections 69a, 69d, 71c, 71d, 71h, 75c, 76b, 77b, 78a, and 78c.
This code is used in section ch6.

68d ⟨ Pop state if code braces match 68d ⟩ =
\fldec \flbrace@depth
if$_\omega$ \flbrace@depth = 0_R ∘
  return$_x$\n
else

```
        continue
    fi
```

This code is used in section 68c.

69a ⟨Patterns for flex lexer 68c⟩ += 68c 69d

```
PICKUPDEF++
  ⟨␣+⟩                          continue
  ⟨NOT_WS⟩ [⟨r⟩⟨n⟩]^c*          ⟨Skip trailing whitespace, save the definition 69b⟩
  ⟨↩⟩                           ⟨Complain if not inside a definition, continue otherwise 69c⟩
```

69b ⟨Skip trailing whitespace, save the definition 69b⟩ =

```
def_x \flnmdef { { val \yytext }{ val \yytextpure }{ val \yyfmark }{ val \yysmark } }
\fldidadeftrue continue
```

This code is used in section 69a.

69c ⟨Complain if not inside a definition, continue otherwise 69c⟩ =

```
\iffldidadef
    \yylval \expandafter { \flnmdef }
    \yybreak { \flinc@linenum enter(INITIAL) return_l ⟨def_re⟩ }
else
    \yybreak { fatal⟨incomplete name definition⟩ }
\yycontinue
```

This code is used in section 69a.

69d ⟨Patterns for flex lexer 68c⟩ += 69a 71c

```
OPTION++
  ⟨↩⟩                                   \flinc@linenum enter(INITIAL) continue
  ⟨␣+⟩                                  \floption@sensetrue continue
  =                                     return_c
  no                                    ⟨Toggle option_sense 70a⟩
  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 71a⟩
  posix-compat                          ⟨Set posix_compat 71b⟩
  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⟩
yywrap                                    return^opt ⟨other⟩
yy_push_state                             return^opt ⟨other⟩
yy_pop_state                              return^opt ⟨other⟩
yy_top_state                              return^opt ⟨other⟩
yy_scan_buffer                            return^opt ⟨other⟩
yy_scan_bytes                             return^opt ⟨other⟩
yy_scan_string                            return^opt ⟨other⟩
yyalloc                                   return^opt ⟨other⟩
yyrealloc                                 return^opt ⟨other⟩
yyfree                                    return^opt ⟨other⟩
yyget_debug                               return^opt ⟨other⟩
yyset_debug                               return^opt ⟨other⟩
yyget_extra                               return^opt ⟨other⟩
yyset_extra                               return^opt ⟨other⟩
yyget_leng                                return^opt ⟨other⟩
yyget_text                                return^opt ⟨other⟩
yyget_lineno                              return^opt ⟨other⟩
yyset_lineno                              return^opt ⟨other⟩
yyget_in                                  return^opt ⟨other⟩
yyset_in                                  return^opt ⟨other⟩
yyget_out                                 return^opt ⟨other⟩
yyset_out                                 return^opt ⟨other⟩
yyget_lval                                return^opt ⟨other⟩
yyset_lval                                return^opt ⟨other⟩
yyget_lloc                                return^opt ⟨other⟩
yyset_lloc                                return^opt ⟨other⟩
extra-type                                return_l ⟨extra type⟩
outfile                                   return_l ⟨outfile⟩
prefix                                    return_l ⟨prefix⟩
yyclass                                   return_l ⟨yyclass⟩
header (-file)_?                          return_l ⟨header⟩
tables-file                               return_l ⟨tables⟩
tables-verify                             return^opt ⟨other⟩
" ["⟨n⟩]^c_* "                            def_x \flnmstr { { val \yytext }{ val \yytextpure } }return_vp «name»
(([a–mo–z] | n[a–np–z]) [⟨αβ⟩-+]_*) | .   fatal⟨ unrecognized %option: val \yytext ⟩
```

70a ⟨ Toggle *option_sense* 70a ⟩ =

```
\iffloption@sense
    \floption@sensefalse
else
    \floption@sensetrue
fi continue
```

This code is used in section 69d.

71a ⟨ Set *lex_compat* 71a ⟩ =

```
\iffloption@sense
    \fllex@compattrue
else
    \fllex@compatfalse
fi return^opt ⟨other⟩
```

This code is used in section 69d.

71b ⟨ Set *posix_compat* 71b ⟩ =

```
\iffloption@sense
    \flposix@compattrue
else
    \flposix@compatfalse
fi return^opt ⟨other⟩
```

This code is used in section 69d.

71c The **RECOVER** state is never used for typesetting and is only added for completeness.

⟨ Patterns for `flex` lexer 68c ⟩ + = 69d 71d

```
RECOVER+
  .*⟨↩⟩                                  \flinc@linenum enter(INITIAL) continue
```

71d Like `bison`, `flex` allows insertion of C code in the middle of the input file.

⟨ Patterns for `flex` lexer 68c ⟩ + = 71c 71h

```
SECT2 PROLOG++
 ⊣ %{ .*           ⟨ Consume the brace and increment the brace level 71e ⟩
 ⊣ %} .*           ⟨ Consume the brace and decrement the brace level 71f ⟩
 ⊣ ⟨␣+⟩ .*         continue
 ⊣ ⟨NOT_WS⟩ .*     ⟨ Begin section 2, prepare to reread, or ignore braced code 71g ⟩
   .               continue
   ⟨↩⟩             \flinc@linenum continue
   ⟨EOF⟩           def \flsectnum { 0 }\yyterminate
```

71e All the code inside is ignored.

⟨ Consume the brace and increment the brace level 71e ⟩ =

```
\flinc \flbracelevel \yyless { 2 }continue
```

This code is used in section 71d.

71f ⟨ Consume the brace and decrement the brace level 71f ⟩ =

```
\fldec \flbracelevel \yyless { 2 }continue
```

This code is used in section 71d.

71g ⟨ Begin section 2, prepare to reread, or ignore braced code 71g ⟩ =

```
ifω \flbracelevel > 0R
    \yybreak continue
else
    \yybreak { \yysetbol { 1R }enter(SECT2) \yyless { 0 }continue }
\yycontinue
```

This code is used in section 71d.

71h A pattern below (for the character class processing) had to be broken into two lines. A special symbol (⊙) has been inserted to indicate that a break had occured.

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`.

⟨ Patterns for flex lexer 68c ⟩ += 71d 75c

```
SECT2++
 ⊣ ⟨␣*⟩⟨↩⟩                    \flinc@linenum continue  ▷ allow blank lines in section 2 ◁
 ⊣ ⟨␣*⟩%{                     ⟨ Start braced code in section 2 72a ⟩
 ⊣ ⟨␣*⟩<                      \ifflsf@skip@ws else enter(SC) fi \yylexreturnraw <
 ⊣ ⟨␣*⟩^                      \yylexreturnraw ^
   "                          enter(QUOTE) return_x \flquotechar
   {[⟨0..9⟩]                   ⟨ Process a repeat pattern 72b ⟩
   $([⟨ ⟩] | ⟨↩⟩)              \yyless{1}\yylexreturnraw\$
   ⟨␣+⟩%{                      ⟨ Process braced code in the middle of section 2 72c ⟩
   ⟨␣+⟩| .*⟨↩⟩                 ⟨ Process a deferred action 73a ⟩
 ⊣ ⟨␣+⟩/*                      ⟨ Process a comment inside a pattern 73b ⟩
 ⊣ ⟨␣+⟩                        ;  ▷ allow indented rules ◁
   ⟨␣+⟩                        ⟨ Decide whether to start an action or skip whitespace inside a rule 73c ⟩
   ⟨␣*⟩⟨↩⟩                     ⟨ Finish the line and/or action 73d ⟩
 ⊣ ⟨␣*⟩<<EOF>>                 ↩
   <<EOF>>                     return_p ⟨EOF⟩
 ⊣ %% .*                       ⟨ Start section 3 74a ⟩
   [(⟨FIRST_CCL_CHAR⟩ | ⟨CCL_EXPR⟩) ⊙
    (⟨CCL_CHAR⟩ | ⟨CCL_EXPR⟩)*   ⟨ Start processing a character class 74b ⟩
   {-}                         return_l \
   {+}                         return_l ∪
   {⟨NAME⟩} [⟨␣⟩]?             ⟨ Process a named expression after checking for whitespace at the end 74c ⟩
   /*                          ⟨ Decide if this is a comment 74d ⟩
   (?#                         ⟨ Determine if this is extended syntax or return a parenthesis 75a ⟩
   (?                          ⟨ Determine if this is a parametric group or return a parenthesis 75b ⟩
   (                           \flsf@push\yylexreturnraw\(
   )                           \flsf@pop\yylexreturnraw\)
   [/|*+?.(){}]                return_c
   .                           \RETURNCHAR
```

72a ⟨ Start braced code in section 2 72a ⟩ =

```
def\flbracelevel{1}
\indented@codefalse\doing@codeblocktrue
enter(PERCENT_BRACE_ACTION)
continue
```

This code is used in section 71h.

72b ⟨ Process a repeat pattern 72b ⟩ =

```
\yyless{1}enter(NUM)
\iffllex@compat
    \yybreak{return_l {p}
else
    \ifflposix@compat
        \yybreak@{return_l {p}
    else
        \yybreak@{return_l {f}
    fi
\yycontinue
```

This code is used in section 71h.

72c ⟨ Process braced code in the middle of section 2 72c ⟩ =

```
def\flbracelevel{1}
enter(PERCENT_BRACE_ACTION)
\ifflin@rule
    \fldoing@rule@actiontrue
    \flin@rulefalse
```

```
        \yybreak { return_x \n }
    else
        \yybreak continue
    \yycontinue
```

This code is used in section 71h.

73a This action has been changed to accomodate the new grammar. The separator (|) is treated as an ordinary (empty) action.

⟨ Process a deferred action 73a ⟩ =

```
    \ifflsf@skip@ws      ▷ whitespace ignored, still inside a pattern ◁
        \yylessafter { }
        \yybreak continue
    else
        \flinc@linenum
        \fldoing@rule@actiontrue
        \flin@rulefalse
        \flcontinued@actiontrue
        \unput { \n }
        enter(ACTION)
        \yybreak { return_x \n }
    \yycontinue
```

This code is used in section 71h.

73b ⟨ Process a comment inside a pattern 73b ⟩ =

```
    \ifflsf@skip@ws
        push state(COMMENT_DISCARD)
    else
        \unput { \/ * }
        def \flbracelevel { 0 }
        \flcontinued@actionfalse
        enter(ACTION)
    fi continue
```

This code is used in section 71h.

73c ⟨ Decide whether to start an action or skip whitespace inside a rule 73c ⟩ =

```
    \ifflsf@skip@ws
        \yybreak continue
    else
        def \flbracelevel { 0 }
        \flcontinued@actionfalse
        enter(ACTION)
        \ifflin@rule
            \fldoing@rule@actiontrue
            \flin@rulefalse
            \yybreak@ { return_x \n }
        else
            \yybreak@ continue
        fi
    \yycontinue
```

This code is used in section 71h.

73d ⟨ Finish the line and/or action 73d ⟩ =

```
    \ifflsf@skip@ws
        \flinc@linenum
        \yybreak continue
    else
```

```
    def \flbracelevel { 0 }
    \flcontinued@actionfalse
    enter(ACTION)
    \unput { \n }
    \ifflin@rule
        \fldoing@rule@actiontrue
        \flin@rulefalse
        \yybreak@ { return_x \n }
    else
        \yybreak@ continue
    fi
\yycontinue
```

This code is used in section 71h.

74a ⟨ Start section 3 74a ⟩ =

```
def \flsectnum { 3 }
enter(SECT3)
\yyterminate
```

This code is used in section 71h.

74b ⟨ Start processing a character class 74b ⟩ =

```
def_x \flnmstr { val \yytext }
\yyless { 1 }
enter(FIRSTCCL)
\yylexreturnraw [
```

This code is used in section 71h.

74c 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 74c ⟩ =

```
def_x \flend@ch { val \yytextlastchar }
if_ω \flend@ch = ‘\} ∘
    \flend@is@wsfalse
else
    \flend@is@wstrue
fi
v_a \expandafter { \astformat@flnametok }
let \astformat@flnametok ∅
def_x next { \yylval { { ^nx \flnametok { val \yytext }{ val v_a } }{ }{ val \yyfmark }{ val \yysmark } } }next
\iffflend@is@ws
    \unput { }
fi
return_l char
```

This code is used in section 71h.

74d ⟨ Decide if this is a comment 74d ⟩ =

```
\ifflsf@skip@ws
    push state(COMMENT_DISCARD)
    continue
else
    \yyless { 1 }
    \yylexreturnraw \/
fi
```

This code is used in section 71h.

75a ⟨ Determine if this is extended syntax or return a parenthesis 75a ⟩ =

```
\iffllex@compat
    \yybreak{\yyless{1}\flsf@push\yylexreturnraw(}
else
    \ifflposix@compat
        \yybreak@{\yyless{1}\flsf@push\yylexreturnraw(}
    else
        \yybreak@{push state(EXTENDED_COMMENT) }
    fi
\yycontinue
```

This code is used in section 71h.

75b ⟨ Determine if this is a parametric group or return a parenthesis 75b ⟩ =

```
\flsf@push
\iffllex@compat
    \yybreak{\yyless{1}}
else
    \ifflposix@compat
        \yybreak@{\yyless{1}}
    else
        \yybreak@{enter(GROUP_WITH_PARAMS)}
    fi
\yycontinue
\yylexreturnraw(
```

This code is used in section 71h.

75c ⟨ Patterns for flex lexer 68c ⟩ += 71h 76b

```
SC++
  ⟨␣*⟩⟨⟵⟩⟨␣*⟩          \flinc@linenum   ▷ allow blank lines and continuations ◁
  [,*]                   return_c
  >                      enter(SECT_2) return_c
  >^                     enter(CARETISBOL) \yyless{1}\yylexreturnraw>
  ⟨SCNAME⟩               \RETURNNAME
  .                      fatal⟨bad <start condition>: val\yytext⟩

CARETISBOL+
  ^                      enter(SECT_2) return_c

QUOTE++
  ["⟨n⟩]^c               \RETURNCHAR
  "                      enter(SECT_2) return_x\flquotechar
  ⟨⟵⟩                   fatal⟨missing quote⟩

GROUP_WITH_PARAMS++
  :                      enter(SECT_2) continue
  -                      enter(GROUP_MINUS_PARAMS) continue
  i                      \flsf@case@instrue continue
  s                      \flsf@dot@alltrue continue
  x                      \flsf@skip@wstrue continue

GROUP_MINUS_PARAMS++
  :                      enter(SECT_2) continue
  i                      \flsf@case@insfalse continue
  s                      \flsf@dot@allfalse continue
  x                      \flsf@skip@wsfalse continue

FIRSTCCL++
  ^[-]⟨n⟩]^c             enter(CCL) \yyless{1}\yylexreturnraw^
  ^(-|])                 \yyless{1}\yylexreturnraw^
```

. — **enter**(**CCL**) \RETURNCHAR

CCL^{++}
-[] ⟨n⟩]^c — \yyless{1}\yylexreturnraw-\RETURNCHAR
[] ⟨n⟩]^c — \RETURNCHAR
] — **enter**(**SECT**$_2$) **return**$_c$
. | ⟨↩⟩ — **fatal**⟨ bad character class ⟩

FIRSTCCL CCL^{++}
[:alnum:] — **set** ϒ and **return**^{ccl} ⟨αn⟩
[:alpha:] — **set** ϒ and **return**^{ccl} ⟨$\alpha\beta$⟩
[:blank:] — **set** ϒ and **return**^{ccl} ⟨ ⟩
[:cntrl:] — **set** ϒ and **return**^{ccl} ⟨↦⟩
[:digit:] — **set** ϒ and **return**^{ccl} ⟨0..9⟩
[:graph:] — **set** ϒ and **return**^{ccl} ⟨❧⟩
[:lower:] — **set** ϒ and **return**^{ccl} ⟨a..z⟩
[:print:] — **set** ϒ and **return**^{ccl} ⟨➾⟩
[:punct:] — **set** ϒ and **return**^{ccl} ⟨.⟩
[:space:] — **set** ϒ and **return**^{ccl} ⟨␣⟩
[:upper:] — **set** ϒ and **return**^{ccl} ⟨A..Z⟩
[:xdigit:] — **set** ϒ and **return**^{ccl} ⟨0..Z⟩
[:^alnum:] — **set** ϒ and **return**^{ccl} ⟨$\neg\alpha n$⟩
[:^alpha:] — **set** ϒ and **return**^{ccl} ⟨$\neg\alpha\beta$⟩
[:^blank:] — **set** ϒ and **return**^{ccl} ⟨¬ ⟩
[:^cntrl:] — **set** ϒ and **return**^{ccl} ⟨¬↦⟩
[:^digit:] — **set** ϒ and **return**^{ccl} ⟨¬0..9⟩
[:^graph:] — **set** ϒ and **return**^{ccl} ⟨¬❧⟩
[:^lower:] — **set** ϒ and **return**^{ccl} ⟨¬a..z⟩
[:^print:] — **set** ϒ and **return**^{ccl} ⟨¬➾⟩
[:^punct:] — **set** ϒ and **return**^{ccl} ⟨¬.⟩
[:^space:] — **set** ϒ and **return**^{ccl} ⟨¬␣⟩
[:^upper:] — **set** ϒ and **return**^{ccl} ⟨¬A..Z⟩
[:^xdigit:] — **set** ϒ and **return**^{ccl} ⟨¬0..Z⟩
⟨CCL_EXPR⟩ — **fatal**⟨ bad character class expression: val\yytext ⟩

NUM^{++}
[⟨0..9⟩]$_+$ — **return**$_v$ **num**
, — **return**$_c$
} — ⟨ Finish the repeat pattern 76a ⟩
. — **fatal**⟨ bad character inside {}'s ⟩
⟨↩⟩ — **fatal**⟨ missing ^{nx}\} ⟩

76a ⟨ Finish the repeat pattern 76a ⟩ =
enter(**SECT**$_2$)
\iffllex@compat
  \yybreak{**return**$_l$}$_p$}
else
  \ifflposix@compat
    \yybreak@{**return**$_l$}$_p$}
  else
    \yybreak@{**return**$_l$}$_f$}
  fi
\yycontinue

This code is used in section 75c.

76b ⟨ Patterns for flex lexer 68c ⟩ += 75c 77b
PERCENT_BRACE_ACTION^{++}
⟨␣∗⟩%} .$_*$ — **def** \flbracelevel{0}**continue**

```
ACTION+
  /*                                  push state(COMMENT)  continue

CODEBLOCK ACTION++
  reject                              continue
  yymore                              continue
⟨M4QSTART⟩                            continue
⟨M4QEND⟩                              continue
.                                     continue
⟨↩⟩                                   ⟨Process a newline inside a braced group 77a⟩
```

77a This actions has been modified to output \n.

⟨Process a newline inside a braced group 77a⟩ =

```
\flinc@linenum
if_ω \flbracelevel = 0_R
    \iffldoing@rule@action
        return_x \n
    else
        continue
    fi
    \fldoing@rule@actionfalse
    \fldoing@codeblockfalse
    enter(SECT_2)
else
    \iffldoing@codeblock
        \ifflindented@code
            \fldoing@rule@actionfalse
            \fldoing@codeblockfalse
            enter(SECT_2)
        fi
    fi
    continue
fi
```

This code is used in section 76b.

77b ⟨Patterns for flex lexer 68c⟩ += 76b 78a

```
ACTION++   ▷ reject and yymore( ) are checked for above, in PERCENT_BRACE_ACTION ◁
  {                                   \flinc \flbracelevel continue
  }                                   \fldec \flbracelevel continue
⟨M4QSTART⟩                            continue
⟨M4QEND⟩                              continue
[⟨αβ⟩_{}"'/⟨n⟩[]]^c+                  continue
[[]]                                  continue
⟨NAME⟩                                continue
'(['\⟨n⟩]^c | \.)*'                   continue
"                                     enter(ACTION_STRING) continue
⟨↩⟩                                   ⟨Process a newline inside an action 77c⟩
.                                     continue
```

77c This actions has been modified to output \n.

⟨Process a newline inside an action 77c⟩ =

```
\flinc@linenum
if_ω \flbracelevel = 0_R
    \iffldoing@rule@action
        return_x \n
    else
        continue
```

```
        fi
        \fldoing@rule@actionfalse
        enter(SECT2)
    fi
```

This code is used in section 77b.

78a ⟨ Patterns for flex lexer 68c ⟩ + = 77b 78c

```
  ACTION_STRING++
    ["\⟨n⟩]^c+                         continue
    \.                                 continue
    ⟨↩⟩                                \flinc@linenum enter(ACTION) continue
    "                                  enter(ACTION) continue
    .                                  continue

  COMMENT COMMENT_DISCARD ACTION ACTION_STRING+
    ⟨EOF⟩                              fatal⟨ EOF encountered inside an action ⟩

  EXTENDED_COMMENT GROUP_WITH_PARAMS GROUP_MINUS_PARAMS+
    ⟨EOF⟩                              fatal⟨ EOF encountered inside pattern ⟩

  SECT2 QUOTE FIRSTCCL CCL+
    ⟨ESCSEQ⟩                           ⟨ Process an escaped sequence 78b ⟩
```

78b ⟨ Process an escaped sequence 78b ⟩ =

```
    ifω \YYSTART = \number\csname flexstate\parsernamespace FIRSTCCL\endcsname ○
        enter(CCL)
    fi
    \RETURNCHAR
```

This code is used in section 78a.

78c ⟨ Patterns for flex lexer 68c ⟩ + = 78a

```
  SECT3++
    ⟨M4QSTART⟩                         continue
    ⟨M4QEND⟩                           continue
    [[]⟨n⟩]^c*(⟨n⟩?)                   continue
    (.|⟨n⟩)                            continue
    ⟨EOF⟩                              def \flsectnum{0}\yyterminate

  ⟨*⟩+
    .|⟨n⟩                              fatal⟨ bad character: val\yytext ⟩
```

78d ⟨ Auxilary code for flex lexer 78d ⟩ =

```
    void define_all_states(void)
    {
      ⟨ Collect state definitions for the flex lexer 78e ⟩
    }
```

This code is used in section ch6.

78e ⟨ Collect state definitions for the flex lexer 78e ⟩ =

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

This code is used in section 78d.

78f ⟨ Catchall rule for the bootstrap lexer 78f ⟩ =

```
  ⟨*⟩+
    .                                  \yyerrterminate
```

This code is used in section 65a.

79a The drive expects this function to be defined but the bootstrap lexer has no need for it. We leave it in to appease the compiler.

⟨ Auxilary code for the bootstrap `flex` lexer 79a ⟩ =
void *define_all_states*(**void**)
{
⟨ Collect state definitions for the bootstrap `flex` lexer 79b ⟩
}

This code is used in section 65a.

79b ⟨ Collect state definitions for the bootstrap `flex` lexer 79b ⟩ =
#define *_register_name*(*name*) *Define_State*(**#***name*, *name*) ▷ The **INITIAL** state is generated automatically ◁
#undef *_register_name*

This code is used in section 79a.

7

The name parser

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. The name parser is what allows one to automatically typeset term names such as `example1` and `%option_name` as *example*$_1$ and ⟨`option_name`⟩.

⟨ `small_parser.yy` ch7 ⟩ =

....

⟨ Name parser C preamble 85g ⟩

....

⟨ Bison options 81a ⟩

⟨**union**⟩ ⟨ *Union of parser types* 85i ⟩

....

⟨ Name parser C postamble 85h ⟩

....

⟨ Token and types declarations 81b ⟩

⟨ Parser productions 81c ⟩

81a ⟨ Bison options 81a ⟩ =

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

This code is used in section ch7.

81b ⟨ Token and types declarations 81b ⟩ =

`%`[`a`...`Z` `0`...`9`]∗	[`a`...`Z` `0`...`9`]∗	`opt`	`na`
`ext`	`l`	`r`	[`0`...`9`]∗
`*` or `?`	$\backslash c$	«meta identifier»	

This code is used in section ch7.

81c ⟨ Parser productions 81c ⟩ =

full_name :

identifier_string suffixes$_{\rm opt}$	⟨ Compose the full name 82a ⟩
«meta identifier»	⟨ Turn a «meta identifier» into a full name 82b ⟩
quoted_name suffixes$_{\rm opt}$	⟨ Compose the full name 82a ⟩

identifier_string :

%[a...Z0...9]*	⟨ Attach option name 82c ⟩
[a...Z0...9]*	⟨ Start with an identifier 83a ⟩
< [a...Z0...9]* >	⟨ Start with a tag 83b ⟩
' * or ? '	⟨ Start with a quoted string 83c ⟩
' \c '	⟨ Start with an escaped character 83d ⟩
' > '	⟨ Start with a > string 83f ⟩
' < '	⟨ Start with a < string 83e ⟩
' . '	⟨ Start with a . string 83j ⟩
' _ '	⟨ Start with an _ string 83g ⟩
' - '	⟨ Start with a - string 83h ⟩
' $ '	⟨ Start with a $ string 83i ⟩
$	⟨ Prepare a bison stack name 83k ⟩
qualifier	⟨ Turn a qualifier into an identifier 84a ⟩
identifier_string [a...Z0...9]*	⟨ Attach an identifier 84b ⟩
identifier_string qualifier	⟨ Attach qualifier to a name 84c ⟩
identifier_string [0...9]*	⟨ Attach an integer 84d ⟩

quoted_name :

" %[a...Z0...9]* "	⟨ Process quoted option 84f ⟩
" [a...Z0...9]* "	⟨ Process quoted name 84e ⟩

suffixes$_{\text{opt}}$:

∘	$\Upsilon \leftarrow \langle\rangle$
.	$\Upsilon \leftarrow \langle$ ${}^{\text{nx}}$\dotsp ${}^{\text{nx}}$\sfxnone $\rangle$
. *suffixes*	⟨ Attach suffixes 84g ⟩
. *qualified_suffixes*	⟨ Attach qualified suffixes 84h ⟩

suffixes :

[a...Z0...9]*	⟨ Start with a named suffix 84i ⟩
[0...9]*	⟨ Start with a numeric suffix 85a ⟩
suffixes .	⟨ Add a dot separator 85b ⟩
suffixes [a...Z0...9]*	⟨ Attach a named suffix 85d ⟩
suffixes [0...9]*	⟨ Attach integer suffix 85c ⟩
qualifier .	$\Upsilon \leftarrow \langle$ ${}^{\text{nx}}$\sfxn val Υ_1 ${}^{\text{nx}}$\dotsp $\rangle$
suffixes qualifier .	$\Upsilon \leftarrow \langle$val Υ_1 ${}^{\text{nx}}$\sfxn val Υ_2 ${}^{\text{nx}}$\dotsp $\rangle$

qualified_suffixes :

suffixes qualifier	⟨ Attach a qualifier 85e ⟩
qualifier	⟨ Start suffixes with a qualifier 85f ⟩

qualifier : opt | na | ext | l | r   . . . | $\Upsilon \leftarrow \langle$val $\Upsilon_1\rangle$

This code is used in section ch7.

82a ⟨ Compose the full name 82a ⟩ =

$\Upsilon \leftarrow \langle$val Υ_1 val $\Upsilon_2\rangle$ \namechars Υ

This code is used in section 81c.

82b ⟨ Turn a «meta identifier» into a full name 82b ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$ ${}^{\text{nx}}$\idstr { val v_a }{ val v_b }$\rangle$ \namechars Υ

This code is used in section 81c.

82c ⟨ Attach option name 82c ⟩ =

$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle$ ${}^{\text{nx}}$\optstr { val v_a }{ val v_b }$\rangle$

This code is used in section 81c.

83a ⟨ Start with an identifier 83a ⟩ =
$\pi_1(\Upsilon_1) \mapsto v_a$
$\pi_2(\Upsilon_1) \mapsto v_b$
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\idstr`{ val v_a }{ val v_b }$\rangle$
This code is used in sections 81c and 84a.

83b Tags are recognized as a separate syntax element although no special processing is performed by the name parser or the associated macros.

⟨ Start with a tag 83b ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\idstr`{ <val v_a> }{ <val v_b> }$\rangle$
This code is used in section 81c.

83c ⟨ Start with a quoted string 83c ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
`\sansfirst` v_b
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ val v_b }{ val v_b }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83d ⟨ Start with an escaped character 83d ⟩ =
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ val v_b }{ val v_b }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83e ⟨ Start with a `<` string 83e ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ < }{ < }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83f ⟨ Start with a `>` string 83f ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ `\greaterthan` }{ `\greaterthan` }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83g ⟨ Start with an `_` string 83g ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ `\uscoreletter` }{ `\uscoreletter` }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83h ⟨ Start with a `-` string 83h ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ - }{ - }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83i ⟨ Start with a `$` string 83i ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ `\safemath` }{ `\safemath` }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83j ⟨ Start with a `.` string 83j ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\chstr`{ . }{ . }${}^{\rm nx}$`\visflag`{ ${}^{\rm nx}$`\termvstring` }{ }$\rangle$
This code is used in section 81c.

83k ⟨ Prepare a `bison` stack name 83k ⟩ =
$\Upsilon \leftarrow \langle{}^{\rm nx}$`\bidstr`{ ${}^{\rm nx}$`\$` }{ `\safemath` }$\rangle$
This code is used in section 81c.

84a ⟨ Turn a qualifier into an identifier 84a ⟩ =
⟨ Start with an identifier 83a ⟩
This code is used in section 81c.

84b ⟨ Attach an identifier 84b ⟩ =
$\pi_2(\Upsilon_1) \mapsto v_a$
$v_a \leftarrow v_a +_{\rm sx} [\,\sqcup\,]$
$\pi_1(\Upsilon_2) \mapsto v_b$
$v_a \leftarrow v_a +_{\rm s} v_b$
$\pi_3(\Upsilon_1) \mapsto v_b$
$v_b \leftarrow v_b +_{\rm sx} [\,\sqcup\,]$
$\pi_2(\Upsilon_2) \mapsto v_c$
$v_b \leftarrow v_b +_{\rm s} v_c$
$\Upsilon \leftarrow \langle {}^{\rm nx}$`\idstr{` val v_a `}{` val v_b `}`$\rangle$
This code is used in sections 81c and 84c.

84c ⟨ Attach qualifier to a name 84c ⟩ =
⟨ Attach an identifier 84b ⟩
This code is used in section 81c.

84d 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,[1]) as id_1) 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 84d ⟩ =
$\Upsilon \leftarrow \langle$val $\Upsilon_1{}^{\rm nx}$`\dotsp`${}^{\rm nx}$`\sfxi` val $\Upsilon_2\rangle$
This code is used in section 81c.

84e ⟨ Process quoted name 84e ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\rm nx}$`\idstr{` val v_a `}{` val v_b `}`${}^{\rm nx}$`\visflag{`${}^{\rm nx}$`\termvstring }{ }`$\rangle$
This code is used in section 81c.

84f ⟨ Process quoted option 84f ⟩ =
$\pi_1(\Upsilon_2) \mapsto v_a$
$\pi_2(\Upsilon_2) \mapsto v_b$
$\Upsilon \leftarrow \langle {}^{\rm nx}$`\optstr{` val v_a `}{` val v_b `}`${}^{\rm nx}$`\visflag{`${}^{\rm nx}$`\termvstring }{ }`$\rangle$
This code is used in section 81c.

84g ⟨ Attach suffixes 84g ⟩ =
$\Upsilon \leftarrow \langle {}^{\rm nx}$`\dotsp` val $\Upsilon_2\rangle$
This code is used in sections 81c and 84h.

84h ⟨ Attach qualified suffixes 84h ⟩ =
⟨ Attach suffixes 84g ⟩
This code is used in section 81c.

84i ⟨ Start with a named suffix 84i ⟩ =
$\Upsilon \leftarrow \langle {}^{\rm nx}$`\sfxn` val $\Upsilon_1\rangle$
This code is used in section 81c.

---

[1]) This allows, for example, names like ⌜$term_0$⌝ while leaving ⌜*char2int*⌝ in its 'natural' form.

85a ⟨Start with a numeric suffix 85a⟩ =
$\Upsilon \leftarrow \langle{}^{\mathrm{nx}}$`\sfxi` $\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 81c.

85b ⟨Add a dot separator 85b⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\dotsp`$\rangle$
This code is used in section 81c.

85c ⟨Attach integer suffix 85c⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\sfxi` $\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 81c.

85d ⟨Attach a named suffix 85d⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\sfxn` $\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 81c.

85e ⟨Attach a qualifier 85e⟩ =
$\Upsilon \leftarrow \langle \mathrm{val}\,\Upsilon_1{}^{\mathrm{nx}}$`\qual` $\mathrm{val}\,\Upsilon_2\rangle$
This code is used in section 81c.

85f ⟨Start suffixes with a qualifier 85f⟩ =
$\Upsilon \leftarrow \langle{}^{\mathrm{nx}}$`\qual` $\mathrm{val}\,\Upsilon_1\rangle$
This code is used in section 81c.

85g C preamble. In this case, there are no 'real' actions that our grammar performs, only TEX output, so this section is empty.
⟨Name parser C preamble 85g⟩ =
This code is used in section ch7.

85h C postamble. It is tricky to insert function definitions that use `bison`'s internal types, as they have to be inserted in a place that is aware of the internal definitions but before said definitions are used.
⟨Name parser C postamble 85h⟩ =
This code is used in section ch7.

85i Union of types.
⟨Union of parser types 85i⟩ =
This code is used in section ch7.

8
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.

```
⟨small_lexer.ll ch8⟩ =
  ⟨Lexer definitions 87a⟩
  ..............................
  ⟨Lexer C preamble 88b⟩
  ..............................
  ⟨Lexer options 88c⟩

  ⟨Regular expressions 88d⟩

  void define_all_states(void)
  {
          ⟨Collect all state definitions 87b⟩
  }
```

87a The tokens consumed by the name parser must represent a relatively fine classification of various identifier substrings to be able to detect various suffixes.

```
⟨Lexer definitions 87a⟩ =
  ⟨Lexer states 88a⟩
  ⟨letter⟩          [_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ]
  ⟨c-escchar⟩       \[fnrtv]
  ⟨wc⟩              ([\'"$.<>]^c \ [_abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0–9] | \.)
  ⟨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 ch8.

87b ⟨Collect all state definitions 87b⟩ =

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

This code is used in section ch8.

88a Strings and characters in directives/rules.

⟨ Lexer states 88a ⟩ =

⟨**state-x**⟩$_f$ **SC_ESCAPED_STRING SC_ESCAPED_CHARACTER**

This code is used in section 87a.

88b ⟨ Lexer C preamble 88b ⟩ =

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

This code is used in section ch8.

88c ⟨ Lexer options 88c ⟩ =

```
⟨option⟩f        bison-bridge
⟨option⟩f        noyywrap
⟨option⟩f        nounput
⟨option⟩f        noinput
⟨option⟩f        reentrant
⟨option⟩f        noyy_top_state
⟨option⟩f        debug
⟨option⟩f        stack
⟨output to⟩f     "small_lexer.c"
```

This code is used in section ch8.

88d ⟨ Regular expressions 88d ⟩ =

⟨ Scan white space 88e ⟩
⟨ Scan identifiers 88f ⟩

This code is used in section ch8.

88e White space skipping.

⟨ Scan white space 88e ⟩ =

[␣⟨f⟩⟨n⟩⟨t⟩⟨v⟩] **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.

⟨ Scan identifiers 88f ⟩ =

```
%binary               return_v %[a...Z0...9]*
%code                 return_v %[a...Z0...9]*
%debug                return_v %[a...Z0...9]*
%default-prec         return_v %[a...Z0...9]*
%define               return_v %[a...Z0...9]*
%defines              return_v %[a...Z0...9]*
%destructor           return_v %[a...Z0...9]*
%dprec                return_v %[a...Z0...9]*
%empty                return_v %[a...Z0...9]*
%error-verbose        return_v %[a...Z0...9]*
%expect               return_v %[a...Z0...9]*
%expect-rr            return_v %[a...Z0...9]*
%file-prefix          return_v %[a...Z0...9]*
%fixed-output-files   return_v %[a...Z0...9]*
%initial-action       return_v %[a...Z0...9]*
%glr-parser           return_v %[a...Z0...9]*
%language             return_v %[a...Z0...9]*
%left                 return_v %[a...Z0...9]*
%lex-param            return_v %[a...Z0...9]*
```

```
%locations                              return_v %[a...Z0...9]*
%merge                                  return_v %[a...Z0...9]*
%name-prefix                            return_v %[a...Z0...9]*
%no-default-prec                        return_v %[a...Z0...9]*
%no-lines                               return_v %[a...Z0...9]*
%nonassoc                               return_v %[a...Z0...9]*
%nondeterministic-parser                return_v %[a...Z0...9]*
%nterm                                  return_v %[a...Z0...9]*
%output                                 return_v %[a...Z0...9]*
%param                                  return_v %[a...Z0...9]*
%parse-param                            return_v %[a...Z0...9]*
%prec                                   return_v %[a...Z0...9]*
%precedence                             return_v %[a...Z0...9]*
%printer                                return_v %[a...Z0...9]*
%pure-parser                            return_v %[a...Z0...9]*
%require                                return_v %[a...Z0...9]*
%right                                  return_v %[a...Z0...9]*
%skeleton                               return_v %[a...Z0...9]*
%start                                  return_v %[a...Z0...9]*
%term                                   return_v %[a...Z0...9]*
%token                                  return_v %[a...Z0...9]*
%token-table                            return_v %[a...Z0...9]*
%type                                   return_v %[a...Z0...9]*
%union                                  return_v %[a...Z0...9]*
%verbose                                return_v %[a...Z0...9]*
%yacc                                   return_v %[a...Z0...9]*
%default[-_]prec                        return_v %[a...Z0...9]*
%error[-_]verbose                       return_v %[a...Z0...9]*
%expect[-_]rr                           return_v %[a...Z0...9]*
%fixed[-_]output[-_]files               return_v %[a...Z0...9]*
%name[-_]prefix                         return_v %[a...Z0...9]*
%no[-_]default[-_]prec                  return_v %[a...Z0...9]*
%no[-_]lines                            return_v %[a...Z0...9]*
%pure[-_]parser                         return_v %[a...Z0...9]*
%token[-_]table                         return_v %[a...Z0...9]*
%(⟨letter⟩ | [0–9] | [-_] | % | [<>])+  return_v %[a...Z0...9]*

▷ suffixes

opt                                     return_v opt
na                                      return_v na
ext                                     return_v ext
l                                       return_v l
r                                       return_v r

▷ delimeters

[<>$._’"]                               return_c
⟨c-escchar⟩                             return_v \c
⟨wc⟩                                    return_v * or ?

▷ identifiers and other names

⟨id_strict⟩                             ⟨ Prepare to process an identifier 90a ⟩
⟨meta_id⟩                               ⟨ Prepare to process a meta-identifier 90b ⟩
⟨int⟩                                   return_v [0...9]*

▷ everything else

.                                       ⟨ React to a bad character 90c ⟩
```

This code is used in section 88d.

90a ⟨ Prepare to process an identifier 90a ⟩ =
return$_v$ [`a...Z0...9`]∗

This code is used in section 88f.

90b ⟨ Prepare to process a meta-identifier 90b ⟩ =
return$_v$ «meta identifier»

This code is used in section 88f.

90c ⟨ React to a bad character 90c ⟩ =
if$_t$ [`bad char`]
  fatal⟨ `invalid character(s):` val `\yytext` ⟩
fi

This code is used in section 88f.

9

Forcing bison and flex to output TEX

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

91a **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 91a ⟩ 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 91a ⟩ =
  ⟨ Auxiliary function definitions 100a ⟩
  int main(int argc, char **argv)
  {
    ⟨ Local variable and type declarations 93c ⟩
    ⟨ Establish defaults 101b ⟩
    ⟨ Command line processing variables 101e ⟩
    ⟨ Process command line options 101f ⟩
    switch (mode) {
      ⟨ Various output modes 92a ⟩
    default: break;
    }
    fprintf(stderr, "Outputting␣tables␣and␣actions\n");
    if (tables_out) {
      fprintf(stderr, "␣␣␣tables␣...␣");
      ⟨ Perform output 96a ⟩
      fprintf(stderr, "actions␣...␣");
      ⟨ Output action switch, if any 99f ⟩
```

```
    }
    else {
      fprintf(stderr, "No output, exiting\n");
      exit(0);
    }
    fprintf(stderr, "done, cleaning up\n");
    ⟨Clean up 93b⟩
    return 0;
  }
```

This code is cited in section 91a.

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 91a.

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⟩ = 101c

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

See also section 101c.

This code is used in section 98a.

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, **const** data should be constant. Even if one succeeds in getting access, there is no reason to believe that the data is not allocated in a write-only region of the memory. The cleanest way to get write access then is to eliminate **const** altogether. The code should have the same semantics after that, and the trick is only marginally bad.

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

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

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.

```
#define forever for ( ; ; )
```

⟨ Common code for C preamble 93a ⟩ =

```
#include <stdio.h>
  FILE *tables_out;
```

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' [1]).

⟨ Clean up 93b ⟩ =

```
  fclose(tables_out);
```

This code is used in section 91a.

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 `CWEB` approach.

⟨ Local variable and type declarations 93c ⟩ = 94b ▽

```
  struct output_d {
    ⟨ Output descriptor fields 93d ⟩
  };
  struct output_d output_desc ⇐ {⟨ Default outputs 94a ⟩};
```

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

This code is used in section 91a.

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

⟨ Output descriptor fields 93d ⟩ = 97b ▽

```
#define _register_table_d(name) bool output_##name:1;
  ⟨ Table names 96c ⟩
#undef _register_table_d
```

See also sections 97b and 98e.

This code is used in section 93c.

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

94a Same for assigning default values to each field.

⟨ Default outputs 94a ⟩ = 97c
```
#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 99a.

This code is used in section 93c.

94b Each descriptor is populated using the same approach.

⟨ Local variable and type declarations 93c ⟩ += 93c 97d
```
#define _register_table_d(name) struct table_d name##_desc ⇐ {0};
  ⟨ Table names 96c ⟩
#undef _register_table_d
```

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

⟨ Global variables and types 94c ⟩ = 94e
```
  static int optimize_tables ⇐ 0;
```
See also sections 94e, 96d, 97a, 98c, and 99g.

This code is used in section 98a.

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

⟨ Options without arguments 94d ⟩ = 96e
```
  register_option_("optimize-tables", no_argument, &optimize_tables, 1, "")
```
See also section 96e.

This code is used in section 102a.

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 (stings as well as integers). The output is controlled by each table's *descriptor* defined below. A more sophisticated approach is possible but this code is merely a 'patch' so we are not after full generality [1]).

```
#define output_table(table_desc, table_name, stream)
        if (output_desc.output_##table_name) {
          int i, j ⇐ 0;
          if (optimize_tables) {
            fprintf(stream, "\\setoptopt{%s}%%\n", table_desc.name);
            if (not table_desc.optimized_numeric) {
              fprintf(stream, "\\beginoptimizednonnumeric{%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, "%s", table_desc.null);
            }
          }
          if (j > MAX_PRETTY_LINE ∧ table_desc.prettify) {
            fprintf (stream, "\n");
            j ⇐ 0;
          }
        }
        if (table_desc.formatter) {
          table_desc.formatter(stream, −i);
        }
        else {
          if (table_name[i]) {
            fprintf (stream, table_desc.postamble, table_name[i]);
          }
          else {
            fprintf (stream, "%s", table_desc.null_postamble);
          }
        }
        if (table_desc.cleanup) {
          table_desc.cleanup(&table_desc);
        }
      }
    }
```

⟨ Global variables and types 94c ⟩ + ≡ 94c 96d

```
  struct table_d {
    ⟨ Generic table desciptor fields 95a ⟩
  };
```

95a ⟨ Generic table desciptor 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.

⟨ Perform output 96a ⟩ = 99b

⟨ Output all tables 96b ⟩

See also section 99b.

This code is used in section 91a.

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

⟨ Output all tables 96b ⟩ =

```
#define _register_table_d(name) output_table(name##_desc, name, tables_out);
  ⟨ Table names 96c ⟩
#undef _register_table_d
```

This code is used in section 96a.

96c Tables will be output by each driver. Placeholder here, for CWEAVE's piece of mind.

⟨ Table names 96c ⟩ =

This code is used in sections 93d, 94a, 94b, 96b, and 109a.

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 bison and flex to generate TEX. The **switch** statement containing the actions is embedded in the parser function so to get access to each action one has to coerce *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 bison and 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.

⟨ Global variables and types 94c ⟩ += 94e 97a

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

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 C. In this case it will be impossible to cycle through actions (as no setup code has been executed), so the parser invocation is omitted.

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

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

⟨ Options without arguments 94d ⟩ += 94d

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

97a The rest of the action output code mimics that for table output, starting with the descriptor. To make the output format more flexible, this descriptor should probably be turned into a specialized routine.

⟨ Global variables and types 94c ⟩ += 96d 98c

```
struct action_d {
   char *preamble;
   char *act_setup;
   char *act_suffix;
   char *action_1;
   char *action_n;
   char *postamble;
   void(*print_rule)(int);
   void(*cleanup)(struct action_d *);
};
```

97b ⟨ Output descriptor fields 93d ⟩ += 93d 98e

```
bool output_actions:1;
```

97c Nothing is output by default, including actions.

⟨ Default outputs 94a ⟩ += 94a 99a

```
.output_actions ⇐ 0,
```

97d ⟨ Local variable and type declarations 93c ⟩ += 94b 98d

```
struct action_d action_desc ⇐ {0};
```

97e 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 case one wishes to use the 'native' `bison` way of referencing terms (i.e. something along the lines of `\the$[term]`) these macros should be used with a trailing underscore (say, `TeXa_`) to let the postprocessor know that the code inside should be postprocessed. The postprocessor will then create three files: one, destined for `CWEAVE`, will use the same macro withough the underscore (i.e. `TeXa` to continue our example, and have the native `bison` terms replaced wih the appropriate TEX commands. Another file is intended for `CTANGLE`, where the whole macro will be replaced first with a special identifier, which in turn, after `CTANGLE` finishes, will be replaced by an appropriately constructed call to `TeX__`. The third file will contain the substitutions.

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

```
#define TeX_(string) fprintf(tables_out, "␣␣␣␣␣␣␣␣␣␣%s%%\n", string)
#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, "␣␣␣␣␣␣␣␣␣␣/yylexcomplain{%s}/yylexerrterminate%%\n", message)
```

⟨ C preamble 97e ⟩ = 98a

```
#define TeX__(string, ...) fprintf(tables_out, "␣␣␣␣␣␣␣␣␣␣"string"%%\n", __VA_ARGS__)
```

See also section 98a.

---

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.

98a 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 preamble 97e ⟩ += 97e
  ⟨ Outer definitions 92b ⟩;
  ⟨ Global variables and types 94c ⟩
  ⟨ Auxiliary function declarations 99i ⟩

98b 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.

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

98c An approach paralleling the table output scheme is taken with constants. Since constants are C *macros* one has to be careful to avoid the temptation of using constant *names* directly as names for fields in structures. They will simply be replaced by the constants' values. When the names are concatenated with other tokens, however, the C preprocessor postpones the macro expansion until the concatenation is complete (see clauses 6.10.3.1, 6.10.3.2, and 6.10.3.3 of the ISO C Standard, [ISO/C11]). Unless the result of the concatenation is still expandable, the expansion will halt [1]).

⟨ Global variables and types 94c ⟩ += 97a 99g

```
struct const_d {
  char *format;
  char *name;
  int value;
};
```

98d ⟨ Local variable and type declarations 93c ⟩ += 97d 100b

```
#define _register_const_d(c_name, ...) struct const_d c_name##_desc;
  ⟨ Constant names 99d ⟩
#undef _register_const_d
```

98e ⟨ Output descriptor fields 93d ⟩ += 97b

```
#define _register_const_d(c_name, ...) bool output_##c_name:1;
  ⟨ Constant names 99d ⟩
#undef _register_const_d
```

1) Another trick to halt expansion is to use *function macros* which will expand only when they are followed by parentheses. Since we do not have control over how constants are defined by `bison`, we cannot take advantage of this feature of the C preprocessor.

99a ⟨ Default outputs 94a ⟩ + = 97c

```
#define _register_const_d(c_name, ...) .output_##c_name ⇐ 0,
  ⟨ Constant names 99d ⟩
#undef _register_const_d
```

99b ⟨ Perform output 96a ⟩ + = 96a

```
  fprintf(tables_out, "%%\n%% constant definitions\n%%\n");
  ⟨ Output constants 99c ⟩
```

99c ⟨ Output constants 99c ⟩ =

```
  { int any_constants ⇐ 0;
#define _register_const_d(c_name, ...)
  if (output_desc.output_##c_name) {
    const_out(tables_out, c_name##_desc)
    any_constants ⇐ 1;
  }
  ⟨ Constant names 99d ⟩
#undef _register_const_d
  if (any_constants) ;    ▷ this is merely a placeholder statement ◁
  }
```

This code is used in section 99b.

99d Constants are very driver specific, so to make CWEAVE happy ...

⟨ Constant names 99d ⟩ =

This code is used in sections 98d, 98e, 99a, and 99c.

99e A macro to help with constant output.

```
#define const_out(stream, c_desc) fprintf(stream, c_desc.format, c_desc.name, c_desc.value);
```

99f 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 99f ⟩ =

This code is used in section 91a.

99g **Error codes**

⟨ Global variables and types 94c ⟩ + = 98c

```
  enum err_codes {
    ⟨ Error codes 99h ⟩ LAST_ERROR
  };
```

99h ⟨ Error codes 99h ⟩ = 115a

```
  NO_MEMORY, BAD_STRING, BAD_MIX_FORMAT,
```

See also section 115a.

This code is used in section 99g.

99i A lot more care is necessary to output the token table. A number of precautions are taken to ensure that a maximum possible range of names can be passed safely to TeX. This involves some manipulation of \catcode's and control characters. The complicated part is left to TeX so the output code can be kept simple. The helper function below is used to 'combine' two strings.

```
#define MAX_PRETTY_LINE 100
```

⟨ Auxiliary function declarations 99i ⟩ =

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

This code is used in section 98a.

100a ⟨ Auxiliary function definitions 100a ⟩ =

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

This code is used in section 91a.

100b **Initial setup**

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

⟨ Local variable and type declarations 93c ⟩ + = 98d 101d

```
enum output_mode {
  ⟨ Output modes 101a ⟩ LAST_OUT
};
```

101a And to calm down CWEAVE . . .

⟨ Output modes 101a ⟩ =

This code is used in section 100b.

101b TEX is the main output mode.

⟨ Establish defaults 101b ⟩ =

```
enum output_mode mode ⇐ TEX_OUT;
```

This code is used in section 91a.

101c **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 92b ⟩ + = 92b

```
#include <unistd.h>
#include <getopt.h>
#include <string.h>
```

101d ⟨ Local variable and type declarations 93c ⟩ + = 100b

```
const char *usage ⇐ "%s␣[options]␣output_file\n";
```

101e ⟨ Command line processing variables 101e ⟩ =

```
int c, option_index ⇐ 0;
enum higher_options {
  NON_OPTION ⇐ FF16, ⟨ Higher index options 102c ⟩ LAST_HIGHER_OPTION
};
static struct option long_options[ ] ⇐ {
⟨ Long options array 102a ⟩
{0, 0, 0, 0}};
```

This code is used in section 91a.

101f 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 101f ⟩ =

```
opterr ⇐ 0;    ▷ we do our own error reporting ◁
forever
{
  c ⇐ getopt_long(argc, argv, (char[ ]){':', ⟨ Short option list 102b ⟩}, long_options, &option_index);
  if (c = −1) break;
  switch (c) {
  case 0:    ▷ it is a flag, the name is kept in long_options[option_index].name, and the value can be found in
        long_options[option_index].val ◁
    break;
  ⟨ Cases affecting the whole program 103c ⟩;
  ⟨ Cases involving specific modes 103d ⟩;
  case '?':
    fprintf(stderr, "Unknown␣option:␣'%s',␣see␣'Usage'␣below\n\n", argv[optind − 1]);
    fprintf(stderr, usage, argv[0]);
    exit(1);
    break;
```

```
    case ':':
      fprintf(stderr, "Missing␣argument␣for␣‘%s’\n\n", argv[optind − 1]);
      fprintf(stderr, usage, argv[0]);
      exit(1);
      break;
    default:
      printf("warning:␣feature␣‘%c’␣is␣not␣yet␣implemented\n", c);
    }
  }
  if (optind ⩾ argc) {
    fprintf(stderr, "No␣output␣file␣specified!\n");
  }
  else {
    tables_out ⇐ fopen(argv[optind++], "w");
  }
  if (optind < argc) {
    printf("script␣files␣to␣be␣loaded:␣");
    while (optind < argc) printf("%s␣", argv[optind++]);
    putchar('\n');
  }
```

This code is used in section 91a.

102a ⟨ Long options array 102a ⟩ =

```
#define register_option_(name, arg_flag, loc, val, exp) {name, arg_flag, loc, val},
  ⟨ Options without shortcuts 103b ⟩
  ⟨ Options with shortcuts 103a ⟩
  ⟨ Options without arguments 94d ⟩
#undef register_option_
```

This code is used in section 101e.

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, ...) , val dd_##arg_flag
  ⟨ Options with shortcuts 103a ⟩
#undef register_option_
#undef dd_optional_argument
#undef dd_required_argument
#undef dd_no_argument
```

This code is used in section 101f.

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 `CWEB` section below.

⟨ Higher index options 102c ⟩ =

```
#define register_option_(name, arg_flag, loc, val, ...) val,
  ⟨ Options without shortcuts 103b ⟩
#undef register_option_
```

This code is used in section 101e.

103a ⟨ Options with shortcuts 103a ⟩ =
This code is used in sections 102a and 102b.

103b ⟨ Options without shortcuts 103b ⟩ =
This code is used in sections 102a and 102c.

103c ⟨ Cases affecting the whole program 103c ⟩ =
This code is used in section 101f.

103d ⟨ Cases involving specific modes 103d ⟩ =
This code is used in section 101f.

103e **`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.

103f **Tables**

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

⟨ Parser table names 103f ⟩ = 104d

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

See also section 104d.

103g 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.

The *yyrimplicit* array will be used by the table output code, together with the postprocessor to output right hand side lengths for the term references that require them in the case when the 'native' `bison` references are used.

⟨ Variables and types local to the parser 103g ⟩ = 106b

```
  unsigned int yyrthree[YYNRULES + 1] ⇐ {0};
  int yyrimplicit[YYNRULES + 1] ⇐ {0};
#ifdef BISON_IS_CRIPPLED
  unsigned int yyrhs[YYNRULES + 1] ⇐ {−1};
  unsigned int yyprhs[YYNRULES + 1] ⇐ {0};
#endif
```

See also sections 106b and 113b.

104a We populate this table below . . .

⟨ Parser defaults 104a ⟩ =
```
#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] ≠ −1; j++) {
        assert(yyprhs[i] + j < sizeof (yyrhs));
        assert(j < yyr1[i]);
        if (⟨ This is an implicit term 104b ⟩) {
          ⟨ Find the rule that defines it and set yyrthree 104c ⟩
        }
      }
    }
  }
#endif
```

104b ⟨ This is an implicit term 104b ⟩ =
```
  (strlen(yytname[yyrhs[yyprhs[i] + j]]) > 1) ∧ (yytname[yyrhs[yyprhs[i] + j]][0] =
      '$') ∧ (yytname[yyrhs[yyprhs[i] + j]][1] = '@')
```
This code is used in section 104a.

104c ⟨ Find the rule that defines it and set *yyrthree* 104c ⟩ =
```
  int rule_number;
  for (rule_number ⇐ 1; rule_number < YYNRULES; rule_number++) {
    if (yyr1[rule_number] = yyrhs[yyprhs[i] + j]) {
      yyrthree[rule_number] ⇐ j;
      break;
    }
  }
  assert(rule_number < YYNRULES);
```
This code is used in section 104a.

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

⟨ Parser table names 103f ⟩ += 103f
```
  _register_table_d(yyrthree)
```

104e We list some macros that are used to assist the post processor and take advantage of the *yyrimlicit* array. As at this time the size of the array is unknown (the preamble is included before the parser file by `mkeparser.w` so the number of rules is unknown at this point), we declare the array as a pointer.

```
#define BZ(term, anchor) ( ( ( YYSTYPE ∗ ) &(term) ) − ( ( YYSTYPE ∗ ) &(anchor) ) +1 )
#define BZZ(term, anchor) ( ( yyrimplicit_p[yyn] ⇐ ( (yyrimplicit_p[yyn] < 0) ? yyrimplicit_p[yyn] : ( (
        YYSTYPE ∗ ) &(term) ) − ( ( YYSTYPE ∗ ) &(anchor) ) +1 ) ) , ( ( YYSTYPE ∗ ) &(term) ) − (
        ( YYSTYPE ∗ ) &(anchor) ) +1 )
```

⟨ C setup code specific to `bison` 104e ⟩ =
```
  int ∗yyrimplicit_p;
```

104f ⟨ Output parser semantic actions 104f ⟩ = 105a
```
  yyrimplicit_p ⇐ yyrimplicit;
```
See also section 105a.

105a **Actions**

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

⟨ Output parser semantic actions 104f ⟩ += 104f

```
if (output_desc.output_actions) {
  int i, j;
  fprintf(tables_out, "%s", action_desc.preamble);
  if (not bare_actions) {
    yypact[0] ⇐ YYPACT_NINF;
    yypgoto[0] ⇐ −1;
    yydefgoto[0] ⇐ YYFINAL;
  }
  for (i ⇐ 1; i < sizeof (yyr1)/sizeof (yyr1[0]); i++) {
    fprintf(tables_out, action_desc.act_setup, i, yyr2[i] − 1);
    if (action_desc.print_rule) {
      action_desc.print_rule(i);
    }
    if (yyr2[i] > 0) {
      if (action_desc.action_1) {
        fprintf(tables_out, "%s", action_desc.action_1);
      }
    }
    for (j ⇐ 2; j ⩽ yyr2[i]; j++) {
      if (action_desc.action_n) {
        fprintf(tables_out, action_desc.action_n, j);
      }
    }
    if (not bare_actions) {
      yyr1[i] ⇐ YYNTOKENS;
      yydefact[0] ⇐ i;
      yyrimplicit[i] ⇐ −yyr2[i];
      yyr2[i] ⇐ 0;
      yyparse(YYPARSE_PARAMETERS);
    }
    fprintf(tables_out, action_desc.act_suffix, i, yyr2[i] − 1);
  }
  fprintf(tables_out, "%s", action_desc.postamble);
  if (action_desc.cleanup) {
    action_desc.cleanup(&action_desc);
  }
}
for (int i ⇐ 1; i < YYNRULES + 1; i++) {
  if (yyrimplicit[i] > 0) {
    fprintf(tables_out, "\\yyimplicitlengthset{%d}{%d}%%\n", i, yyrimplicit[i]);
  }
}
```

105b **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 105b ⟩ =

```
_register_const_d(YYEMPTY)
_register_const_d(YYPACT_NINF)
```

```
  _register_const_d(YYLAST)
  _register_const_d(YYNTOKENS)
  _register_const_d(YYNRULES)
  _register_const_d(YYNSTATES)
  _register_const_d(YYFINAL)
#ifndef YYEOF
  _register_const_d(YYSYMBOL_YYEOF)
#endif
```

This code is used in section 111b.

106a Constants defined to maintain compatibility with the older versions of `bison`.

⟨ Parser virtual constants 106a ⟩ =

```
  _register_const_d(YYSYMBOL_YYEOF, YYEOF)
```

This code is used in section 111b.

106b **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 103g ⟩ += 103g 113b

```
  char *token_format_char ⇐ Λ;
  char *token_format_affix ⇐ Λ;
  char *token_format_suffix ⇐ Λ;
  char *bootstrap_token_format ⇐ Λ;
```

106c ⟨ Parser specific options without shortcuts 106c ⟩ = 108c

```
  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 108c and 112c.

106d ⟨ Handle parser output options 106d ⟩ = 112e

```
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 112e and 113c.

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

```
  bool output_tokens:1;
```

107a No tokens are output by default.

⟨ Parser specific default outputs 107a ⟩ =

```
.output_tokens ⇐ 0,
```

107b 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 107b ⟩ =

```
if (output_desc.output_tokens) {
  int i;
  int length;
  char token;
  char *token_name;
  bool too_creative ⇐ false;

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

107c The size of the token name table is useful to determine, say, how many 'named' tokens the parser uses.

⟨ Output parser constants 107c ⟩ =

```
fprintf(tables_out, "\\constset{YYTRANSLATESIZE}{%d}%%\n", (int)(sizeof (yytranslate)/sizeof (yytranslate[0])));
```

107d 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.

108a **Token only mode**

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

```
⟨Parser specific output modes 108a⟩ =
  TOKEN_ONLY_OUT,
```

108f ▽

See also sections 108f and 108h.

108b
```
⟨Handle parser related output modes 108b⟩ =
case TOKEN_ONLY_OUT:
  ⟨Prepare token only output environment 108e⟩
  break;
```

108g ▽

See also sections 108g and 108i.

108c
```
⟨Parser specific options without shortcuts 106c⟩ +=
  register_option_("token-only-mode", no_argument, 0, TOKEN_ONLY_MODE, "")
```

△ 106c 112c ▽

108d
```
⟨Configure parser output modes 108d⟩ =
case TOKEN_ONLY_MODE:
  mode ⇐ TOKEN_ONLY_OUT;
  break;
```

108e
```
⟨Prepare token only output environment 108e⟩ =
  if (not token_format_char) {
    token_format_char ⇐ "{%u}";
  }
  if (not token_format_affix) {
    token_format_affix ⇐ "%%␣token:␣%d,␣token␣value:␣%d\n\\prettytoken@{";
  }
  if (not token_format_suffix) {
    token_format_suffix ⇐ "}%%␣%s\n";
  }
  output_desc.output_tokens ⇐ 1;
```

This code is used in section 108b.

108f **Generic output**

Generic output is not programmed yet.

```
⟨Parser specific output modes 108a⟩ +=
  GENERIC_OUT,
```

△ 108a 108h ▽

108g
```
⟨Handle parser related output modes 108b⟩ +=
case GENERIC_OUT:
  printf("This␣mode␣is␣not␣supported␣yet\n");
  exit(0);
  break;
```

△ 108b 108i ▽

108h **TEX output**

The TEX mode is the main reason for this software.

```
⟨Parser specific output modes 108a⟩ +=
  TEX_OUT,
```

△ 108f

108i
```
⟨Handle parser related output modes 108b⟩ +=
case TEX_OUT:
  ⟨Set up TEX table output for parser tables 109a⟩
  ⟨Prepare TEX format for semantic action output 110b⟩
```

△ 108g

```
⟨ Prepare TEX format for parser constants 111b ⟩
⟨ Prepare TEX format for parser tokens 112a ⟩
break;
```

109a 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 109a ⟩ = 110a ▽

```
#define _register_table_d(name)tex_table(name);
  ⟨ Table names 96c ⟩
#undef _register_table_d
  yyr1_desc.name ⇐ "yyrone";
  yyr2_desc.name ⇐ "yyrtwo";
```

See also section 110a.

This code is used in section 108i.

109b The memory allocated for the *yytname* table is released at the end.

⟨ Helper functions declarations for for parser output 109b ⟩ =

```
void yytname_cleanup(struct table_d *table);
int yytname_formatter_tex(FILE *stream, int index);
int yytname_formatter(FILE *stream, int index);
```

109c 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 109c ⟩ = 111a ▽

```
void yytname_cleanup(struct table_d *table)
{
  free(table→separator);
  free(table→null);
}
int yytname_formatter_tex(FILE *stream, int index)
{
  char *token_name ⇐ yytname[index];
  unsigned char token;
  int length ⇐ 0;

  fprintf(stream, "\\addname␣");
  while ((token ⇐ *token_name)) {
    if (token < °40 ∨ token = °177) {   ▷ unprintable characters ◁
      fprintf(stream, "^^%c", token < °100 ? (unsigned char)(token + °100) : (unsigned char)(token − 100));
      length ⇐+ 3;
    }
    else {
      fprintf(stream, "%c", token);
      length++;
    }
    token_name++;
  }
  fprintf(stream, "\n");
  return length;
}
int yytname_formatter(FILE *stream, int index)
{
  char *token_name;
  unsigned char token;
  int length ⇐ 0;
  bool too_creative ⇐ false;   ▷ to indicate if the name is too dangerous to print ◁
```

```
    fprintf (stream, "\\addname");
    if (index ⩾ 0) {      ▷ this is not the last name ◁
      token_name ⇐ yytname[index];
      if (token_name = Λ) {
        token_name ⇐ "Υimpossible";
      }
      while ((token ⇐ *token_name)) {
        length ⇐± fprintf (stream, "{%u}", (unsigned int) token);
        if (token < °40 ∨ token = °177) {
          too_creative ⇐ true;
        }
        token_name++;
      }
      fprintf (stream, "%%␣%s\n", too_creative ? ".unprintable." : yytname[index]);
    }
    else {      ▷ this is the last name ◁
      token_name ⇐ yytname[−index];
      if (token_name = Λ) {
        token_name ⇐ "Υimpossible";
      }
      while ((token ⇐ *token_name)) {
        length ⇐± fprintf (stream, "{%u}", (unsigned int) token);
        token_name++;
        if (token < °40 ∨ token = °177) {
          too_creative ⇐ true;
        }
      }
      fprintf (stream, "%%␣%s\n\\end\n%%\n",
            too_creative ? ".unprintable." : (yytname[−index] ? yytname[−index] : "end␣of␣array"));
    }
    return length;
  }
```

See also section 111a.

110a ⟨Set up TEX table output for parser tables 109a⟩ += 109a

```
  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;
```

110b ⟨Prepare TEX format for semantic action output 110b⟩ =

```
  if (optimize_actions) {
    action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
    "\\catcode`\\/=0\\relax␣%␣see␣the␣documentation␣for␣an␣explanation␣of␣this␣trick\n"
    "\\def\\yybigswitch#1{%%\n"
    "␣␣␣␣\\csname␣dobisonaction\\number␣#1\\parsernamespace\\endcsname\n"
    "}\\stashswitch{yybigswitch}%%\n";
    action_desc.act_setup ⇐ "\n\\expandafter\\def\\csname␣dobisonaction%d\\parsernamespa\
        ce\\endcsname{%%\n%%";
    action_desc.act_suffix ⇐ "}%%␣end␣of␣rule␣%d\n";
```

```
    action_desc.action_1 ⇐ Λ;
    action_desc.action_n ⇐ Λ;
    action_desc.postamble ⇐ "\n\\catcode‘\\/=12\\relax\n\n";
    action_desc.print_rule ⇐ print_rule;
    action_desc.cleanup ⇐ Λ;
    output_desc.output_actions ⇐ 1;
  }
  else {
    action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
    "\\catcode‘\\/=0\\relax␣%␣see␣the␣documentation␣for␣an␣explanation␣of␣this␣trick\n"
    "\\def\\yybigswitch#1{%%\n"
    "␣␣\\ifcase#1\\relax\n";
    action_desc.act_setup ⇐ "␣␣␣␣␣␣\\or␣%%␣(rule␣%d)␣";
    action_desc.act_suffix ⇐ "";
    action_desc.action_1 ⇐ Λ;
    action_desc.action_n ⇐ Λ;
    action_desc.postamble ⇐ "␣␣\\else\n␣␣\\fi\n}\\stashswitch{yybigswitch}%%\n\\catcode‘\
        \\/=12\\relax\n\n";
    action_desc.print_rule ⇐ print_rule;
    action_desc.cleanup ⇐ Λ;
    output_desc.output_actions ⇐ 1;
  }
```

This code is used in section 108i.

111a 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.

⟨ Helper functions for parser output 109c ⟩ + ≡ 109c

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

111b 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 exceptions are `YYPACT_NINF` and `YYSYMBOL_YYEOF` that have to be handled separately because the underscore in its name makes it difficult to use it as a command sequence name.

⟨ Prepare TEX format for parser constants 111b ⟩ ≡

```
#define _register_const_d(c_name) c_name##_desc.format ⇐ "\\constset{%s}{%d}%%\n";
  c_name##_desc.name ⇐ #c_name;
  c_name##_desc.value ⇐ c_name;
  output_desc.output_##c_name ⇐ 1;
  ⟨ Parser constants 105b ⟩
```

```
#undef _register_const_d
#ifdef YYEOF    ▷ other values have already been set correctly ◁
#define _register_const_d(c_name, vvalue) c_name##_desc.format ⇐ "\\constset{%s}{%d}%%\n";
  c_name##_desc.name ⇐ #c_name;
  c_name##_desc.value ⇐ vvalue;
  output_desc.output_##c_name ⇐ 1;
  ⟨Parser virtual constants 106a⟩
#undef _register_const_d
#endif
  YYPACT_NINF_desc.name ⇐ "YYPACTNINF";
  YYSYMBOL_YYEOF_desc.name ⇐ "YYSYMBOLxYYEOF";
```

This code is used in section 108i.

112a Token definitions round off the TEX output mode.

⟨Prepare TEX format for parser tokens 112a⟩ =

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

This code is used in section 108i.

112b **Command line options**

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

112c ⟨Parser specific options without shortcuts 106c⟩ + = 108c

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

112d ⟨Shortcuts for command line options affecting parser output 112d⟩ =

```
  , 'h'
```

112e ⟨Handle parser output options 106d⟩ + = 106d 113c

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

112f ⟨Parser specific options with shortcuts 112f⟩ =

```
  register_option_("debug", optional_argument, 0, 'b', "")
  register_option_("mode", required_argument, 0, 'm', "")
  register_option_("table-separator", required_argument, 0, 'z', "")
  register_option_("format", required_argument, 0, 'f', "")    ▷ name? ◁
  register_option_("table", required_argument, 0, 't', "")    ▷ specific table ◁
  register_option_("constant", required_argument, 0, 'c', "")    ▷ specific constant ◁
```

```
register_option_("name-length", required_argument, 0, 'l', "")    ▷ change MAX_NAME_LENGTH ◁
register_option_("token", required_argument, 0, 'n', "")    ▷ specific token ◁
register_option_("run-parse", required_argument, 0, 'p', "")    ▷ run the parser ◁
register_option_("parse-file", required_argument, 0, 'i', "")    ▷ input for the parser ◁
```

113a The string below is a list of short options.

113b A few options can be discussed immediately.

⟨ Variables and types local to the parser 103g ⟩ + ≡ 106b

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

113c ⟨ Handle parser output options 106d ⟩ + ≡ 112e

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

113d 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.

113e Tables

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

⟨ Scanner table names 113e ⟩ ≡

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

113f Actions

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

⟨ Variables and types local to the scanner driver 113f ⟩ ≡ 115b

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

See also sections 115b and 119g.

114a 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.

⟨ Compute exotic scanner constants 114a ⟩ =

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

114b ⟨ Output scanner actions 114b ⟩ =

```
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, "␣%%␣YY_END_OF_BUFFER\n%s\n", "␣␣␣␣␣␣␣␣␣␣\\yylexeofaction");
    }
    else {
      fprintf(tables_out, "\n");
      if (not bare_actions) {
        ((struct yyguts_t *) fake_scanner)→yy_hold_char ⇐ 0;
        yy_accept[1] ⇐ i;
```

```
            if (i % 10 = 0) {
              fprintf(stderr, ".");
            }
            yylex(Λ, fake_scanner);
          }
        }
        fprintf(tables_out, action_desc.act_suffix, i);
      }
      fprintf(tables_out, "␣␣␣␣␣␣%%␣end␣of␣file␣states:\n%s\n",
          "␣␣␣␣␣␣%#define␣YY_STATE_EOF(state)␣(YY_END_OF_BUFFER␣+␣state␣+␣1)");
      if (max_eof_state = 0) {    ▷ in case the user has not declared any states ◁
        max_eof_state ⇐ YY_STATE_EOF(INITIAL);
      }
      for ( ; i ⩽ max_eof_state; i++) {
        fprintf(tables_out, action_desc.act_setup, i);
        if (not bare_actions) {
          fprintf(tables_out, "\n");
          ((struct yyguts_t *) fake_scanner)→yy_hold_char ⇐ 0;
          yy_accept[1] ⇐ i;
          yylex(Λ, fake_scanner);
        }
        fprintf(tables_out, action_desc.act_suffix, i);
      }
      fprintf(tables_out, "%s", action_desc.postamble);
      if (action_desc.cleanup) {
        action_desc.cleanup(&action_desc);
      }
    }
    ⟨ Compute magic constants 115c ⟩
    ⟨ Output states 116b ⟩;
    fprintf(tables_out, "\\constset{YYECMAGIC}{%d}%%\n", yy_ec_magic);
    fprintf(tables_out, "\\constset{YYMAXEOFSTATE}{%d}%%\n", max_eof_state);
```

115a ⟨ Error codes 99h ⟩ += 99h

```
    BAD_SCANNER,
```

115b ⟨ Variables and types local to the scanner driver 113f ⟩ += 113f 119g

```
    int yy_ec_magic;
```

115c The 'magic' constants are similar to the 'exotic' ones mentioned above except the methods used to compute them rely on reverse engineering the scanner function. Since this changes the scanner tables it has to be done after the 'driver' has finished going through all the actions.

⟨ Compute magic constants 115c ⟩ =

```
    {
      int i, j;
      char fake_yytext[YY_MORE_ADJ + 1];
      yyscan_t yyscanner;
      struct yyguts_t *yyg;
      if (yylex_init(&yyscanner)) {
        printf("Cannot␣initialize␣the␣scanner\n");
        exit(BAD_SCANNER);
      }
      yyg ⇐ (struct yyguts_t *) yyscanner;
      yyg→yy_start ⇐ 0;
      yy_set_bol(0);
      yyg→yytext_ptr ⇐ fake_yytext;
```

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

This code is used in section 114b.

116a **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.

```
#define Define_State(st_name, st_num) do {
          struct lexer_state_d *this_state;
          this_state ⇐ malloc(sizeof(struct lexer_state_d));
          this_state→name ⇐ st_name;
          this_state→value ⇐ st_num;
          this_state→next ⇐ Λ;
          if (last_state) {
            last_state→next ⇐ this_state;
            last_state ⇐ this_state;
          }
          else {
            last_state ⇐ state_list ⇐ this_state;
          }
          if (YY_STATE_EOF(st_num) > max_eof_state) {
            max_eof_state ⇐ YY_STATE_EOF(st_num);
          }
        } while (0);
⟨ Scanner variables and types for C preamble 116a ⟩ =
  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;
```

116b
```
⟨ Output states 116b ⟩ =
  {
    struct lexer_state_d *current_state;
    struct lexer_state_d *next_state;
    current_state ⇐ next_state ⇐ state_list;
    if (current_state) {
```

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

This code is used in section 114b.

117a **Constants**

The few hard coded constants needed for the lexer to work are listed here.

⟨ Scanner constants 117a ⟩ =

```
  _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 118b.

117b **Output modes**

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

117c **Generic output**

Generic output is not programmed yet.

⟨ Scanner specific output modes 117c ⟩ = 117e

```
  GENERIC_OUT,
```

See also section 117e.

117d ⟨ Handle scanner output modes 117d ⟩ = 117f

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

See also section 117f.

117e **TEX mode**

The TEX mode is the main focus of this software.

⟨ Scanner specific output modes 117c ⟩ + = 117c

```
  TEX_OUT,
```

117f ⟨ Handle scanner output modes 117d ⟩ + = 117d

```
case TEX_OUT:
  ⟨ Set up TEX format for scanner tables 117g ⟩
  ⟨ Set up TEX format for scanner actions 118a ⟩
  ⟨ Prepare TEX format for scanner constants 118b ⟩
  break;
```

117g ⟨ Set up TEX format for scanner tables 117g ⟩ =

```
  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 117f.

118a ⟨ Set up TEX format for scanner actions 118a ⟩ =

```
if (optimize_actions) {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax\n%\n"
  "\\def\\yydoactionswitch#1{%%\n"
  "␣␣␣␣\\let\\yylextail\\yylexcontinue\n"
  "␣␣␣␣\\csname␣doflexaction\\number␣#1\\parsernamespace\\endcsname\n"
  "␣␣␣␣\\yylextail\n"
  "}\\stashswitch{yydoactionswitch}%\n";
  action_desc.act_setup ⇐ "\n\\expandafter\\def\\csname␣doflexaction%d\\parsernamespac\
      e\\endcsname{%%";
  action_desc.act_suffix ⇐ "}%%␣end␣of␣rule␣%d\n";
  action_desc.action_1 ⇐ Λ;
  action_desc.action_n ⇐ Λ;
  action_desc.postamble ⇐ "\\catcode`\\/=12\\relax\n%\n";
  action_desc.print_rule ⇐ Λ;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
else {
  action_desc.preamble ⇐ "%\n%␣the␣big␣switch\n%\n"
  "\\catcode`\\/=0\\relax\n%\n"
  "\\def\\yydoactionswitch#1{%%\n␣␣\\let\\yylextail\\yylexcontinue\n"
  "␣␣\\ifcase#1\\relax\n";
  action_desc.act_setup ⇐ "␣␣␣␣␣␣\\or\n""␣␣␣␣␣␣\\YYRULESETUP␣%%␣(rule␣%d)␣";
  action_desc.act_suffix ⇐ "␣␣␣␣␣␣%%␣end␣of␣rule␣%d\n";
  action_desc.action_1 ⇐ Λ;
  action_desc.action_n ⇐ Λ;
  action_desc.postamble ⇐ "␣␣\\else\n␣␣\\fi\n␣␣\\yylextail\n}\\stashswitch{yydoactions\
      witch}%\n\\catcode`\\/=12\\relax\n%\n";
  action_desc.print_rule ⇐ Λ;
  action_desc.cleanup ⇐ Λ;
  output_desc.output_actions ⇐ 1;
}
```

This code is used in section 117f.

118b TEX constant output is another place where the techniques described above are applied. A few names are handled separately, because they contain underscores.

⟨ Prepare TEX format for scanner constants 118b ⟩ =

```
#define _register_const_d(c_name) c_name##_desc.format ⇐ "\\constset{%s}{%d}%%\n";
  c_name##_desc.name ⇐ #c_name;
  c_name##_desc.value ⇐ c_name;
  output_desc.output_##c_name ⇐ 1;
  ⟨ Scanner constants 117a ⟩
#undef _register_const_d
  YY_END_OF_BUFFER_CHAR_desc.name ⇐ "YYENDOFBUFFERCHAR";
```

```
YY_NUM_RULES_ desc.name ⇐ "YYNUMRULES";
YY_END_OF_BUFFER_ desc.name ⇐ "YYENDOFBUFFER";
```

This code is used in section 117f.

119a ⟨ Output exotic scanner constants 119a ⟩ =

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

119b **Command line options**

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

119c ⟨ Scanner specific options without shortcuts 119c ⟩ =

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

119d ⟨ Shortcuts for command line options affecting scanner output 119d ⟩ =

```
, 'h'
```

119e ⟨ Handle scanner output options 119e ⟩ = 119h

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

See also section 119h.

119f ⟨ Scanner specific options with shortcuts 119f ⟩ =

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

119g A few options can be immediately discussed.

⟨ Variables and types local to the scanner driver 113f ⟩ += 115b

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

119h ⟨ Handle scanner output options 119e ⟩ += 119e

```
case 'b':    ▷ debug (level) ◁
  debug_level ⇐ optarg ? atoi(optarg) : 1;
  break;
case 'm':    ▷ output mode ◁
  switch (optarg[0]) {
```

```
    case 'T': case 't':
      mode ⇐ TEX_OUT;
      break;
    case 'b': case 'B': case 'g': case 'G':
      mode ⇐ GENERIC_OUT;
      break;
    default:
      break;
    }
    break;
  case 'z': table_separator ⇐ (char *) malloc((strlen(optarg) + 1) * sizeof(char));
    strcpy(table_separator, optarg);
    break;
```

10
Philosophy

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

121a **On typographic convention**

It must seem quite perplexing to some readers that a manual focussing 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, e m p h a s i s) 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 presense openly violates the 'universally gray pages are a must' dogma.

One final remark concerns the abundance of footnotes in this manual. I confess, there is almost no reason for it ... except *I like footnotes*! They help introduce the air of mystery and adventure to an otherwise boring text. They are akin to the author wispering a secret in the reader's ear [1]).

122a **Why GPL**

Selecting the license for this project involves more than the availability of the source code. TEX, by its very nature is an interpreted [2]) 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 renege 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 *I*maginary *P*roperty) disease through software.

The industry has adapted, of course. So called (non GPL) 'open source licenses', that are supposed to be an improvement on GPL, are a sort of 'immune reaction' to the free software movement. Describing GPL as 'viral', creating dismissive acronims 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.

---

[1]) Breaking convention by making the pages even less uniform is an added bonus. [2]) There are some exceptions to this, in the form of preloaded *formats*.

123a Why not C++ or OOP in general

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

Object oriented design has certainly taken on an aura of a religious dictate, universally adhered to and forcefully promoted by its followers. Unfortunately, the definition of what constitutes an 'object-oriented' approach is rather vague. A few 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 rarely, if ever [1]), 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 [2]). This may change but right now I have tried to make the macros as generic as possible, hopefully making ∗TEX adaptation easy [3]).

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

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

The feeling of a `WEB` source being *over-documented* is most certainly subjective, and, I am sure, not shared by all 'current programmers'. The advantage of using `WEB`-like tools, however, is that it gives the programmer the ability to place 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.

---

[1]) In some cases, a publisher would only accept a LATEX document, sadly. Better than most alternatives though. [2]) I am well familiar with the programming that went into LATEX, which is of highest quality. I do not share the design philosophy though and try to use only the most standard features [3]) Unfortunately some redesign would be certainly necessary. Thus, `SPLinT` relies on the way plain TEX allocates token registers so if the corresponding scheme in LATEX is drastically different, this portion of the macros would have to be rewritten.

124a
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.

124b
Some CWEB idiosynchrasies

CWEB was among the first tools for literate programming intended for public use [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 [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 ot 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 *exp* 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 \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 type @t} TeX stuff {@> which inserts \hbox{} TeX stuff {} 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 \hbox{} left behind *before* the 'TeX stuff'. The programmer 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 \lastbox approach unless the \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

---

1) The original WEB was designed to support DEK's TeX and METAFONT projects and was inteded for PASCAL family languages.
2) Quirks would be a better term.

were possible [1]).

Continuing with the theme of inserting TEX material into CWEAVE output, another one of CWEB's inflexibilities is the lack of means of inserting TEX *between* sections. While inserting pure text may be arranged by putting a codeless section after the one with the code (appropriate macros can be written to suppress the generation of a reference to such a section), inserting command sequences that affect, say, the typesetting style of the consequent sections is not so easy. The trick with a 'fake' section below will be quite visible in the final output which is almost always undesirable. Using the @t mechanism is also far from ideal.

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 \B macro results in rather wasteful multiple reading of the tokens in the C portion, not to mention the absense of any guarantee that \B can find the end of its argument (the macros used by SPLinT look for the \par inserted by CWEAVE whenever \B is output but an unsuspecting programmer may disrupt this mechanism by inserting h{is, her} own \par using the @t facility with the aim to put a picture in the middle of the code, for example.

The authors of 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 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 CWEB requires some macro acrobatics, to say the least.

Also absent is a facility to explicitly inhibit the indexing of a specific word (in CWEAVE's own source, the references for *pp* fill up several lines in the index) or limit it to definitions only (as 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 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 SPLinT that deal with bison and 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 SPLinT approximately follows that of noweb: the section reference consists of two parts, where the first is the page number the section starts on, and the the second is the index of the section within the page. Within the page, sections are indexed by (sequences of) letters of the aphabet (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 noweb, since 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 noweb, are the references in the margins that allow the reader to jump from one chink 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 CWEAVE itself. In the case of 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 [2]) style airing of grievances, let me state once again that CWEB is a remarkable

---

[1]) 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. [2]) Yes, I am old enough to know what this means.

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 CWEB itself or even write one's own literate programming tool.

126a Why not GitHub©, Bitbucket©, etc

Git is fantastic software that is used extensively in the development of SPLinT. The distribution archive is a Git repository. The use of centralized services such as GitHub© [1]), however, seems redundant. The standard cycle, 'clone-modify-create pull request' works the same even when 'clone' is replaced by 'download'. Thus, no functionality is lost. This might change if the popularity of the package unexpectedly increases.

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

[1]) A recent aquisition of GitHub© 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.

11
Checklists

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`
  - ▫ `xxpression` (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 `xxpression` examples are (nearly) identical?

509

12
Bibliography

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

* * *

[ACM] Ronald M. Baecker, Aaron Marcus, *Human Factors and Typography for More Readable Programs*, Reading, Massachusetts: Addison-Wesley, 1990, xx+344 pp.

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

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

[CWEB] Donald E. Knuth and Silvio Levy *The* `CWEB` *System of Structured Documentation*, Reading, Massachusetts: Addison-Wesley, 1993, iv+227 pp.

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

[DEK2] Donald E. Knuth *The future of TEX and* `METAFONT`, TUGboat **11** (4), p. 489, 1990.

[DHB] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman, *Syntactic Abstraction in Scheme*, Lisp Symb. Comput. 5, **4** (Dec. 1992), pp. 295–326.

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

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

[Fi] Jonathan Fine, *The* `\CASE` *and* `\FIND` *macros*, TUGboat **14** (1), pp. 35–39, 1993.

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

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

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

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

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

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

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

[KR] B. Kernighan, D. Ritchie, *The* C *programming language*, Englewood Cliffs, NJ: Prentice Hall, 1978.

[La] *The* `l3regex` *package: regular expressions in TEX*, The LATEX3 Project.
[Pa] Vern Paxson et al., *Lexical Analysis With Flex, for Flex 2.5.37*, July 2012. http://flex.sourceforge.net/manual/.
[Ra] Norman Ramsey, *Literate programming simplified*, IEEE Software, **11** (5), pp. 97–105, 1994.
[Sh] Alexander Shibakov, *Parsers in TEX and using* `CWEB` *for general pretty-printing*, TUGboat **35** (1), 2014, available as part of the documentation supplied with `SPLinT`.
[Wo] Marcin Woliński, `Pretprin`—*a LATEX2ε package for pretty-printing texts in formal languages*, TUGboat **19** (3), 1998—Proceedings of the 1998 TUG Annual Meeting.

13
Index

This section is, perhaps, the most valuable product of CWEB's labors. It lists references to definitions (set in *italic*) as well as uses for each C identifier used in the source. Special facilities have been added to extend 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:

left_hand_side :
    $term_1$ $term_2$ $term_3$ \do \something \with Υ_1

inside the TEX part of a CWEB section will generate several index entries, as well, including the entries for any material inside the action, mimicking CWEB's behavior for the *inline* C (|...|). Such entries (except for the references to 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 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, π_1 represents \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 Υ_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, π_1 appears under 'p'). One should also be aware that the indexing of some terms has been suppressed, since they appear too often.

Bison, Flex, and TeX indices

A LIST OF ALL SECTIONS

CONTENTS (SPLINT)