Zlex

A Scanner-Generator

for Zlex Version 1.02. Februar

Zerksis D. Umrigar

1 Introduction

In order to introduce Zlex, the process of scanning is reviewed and some terms are introduced. Zlex is is compared with similar programs and the motivation for the development of yet another scanner generator is presented. An example is used to illustrate the operation of Zlex.

1.1 Scanner Generator Overview

Many programming applications require scanning a input stream and partitioning it into a token stream, where tokens are typically identified with non-overlapping subsequences of the input stream. For example, a natural-language interface will need to partition a stream of characters into a stream of words, which constitute the tokens. A compiler for a programming language will need to partition the stream of characters representing a program into a stream of literals, reserved words, operators and identifiers which constitute the tokens of the programming language.

A *scanner* is a program or portion of a program which performs the task of partitioning a input stream into a token stream. Writing a scanner is a very common programming task: all applications which analyze some form of text input will usually contain some kind of scanner (even though it may not be identified as such). Such scanners are usually written by hand, typically in a procedural programming language like C or Pascal, where knowledge about the syntax of tokens is intimately interwoven into programming constructs which specify how to recognize those tokens. Even though scanners are usually programs of relatively modest complexity, maintaining non-trivial hand-written scanners can still be a demanding task.

A *scanner-generator* is a program which is given a formal specification of the syntax of tokens and automatically generates a scanner from those specifications. Since the specifications are largely declarative in that they specify only *what* constitutes a token without specifying *how* to recognize one, they are much easier to maintain than hand-written scanners. A pattern language based on regular expressions is the formalization used to specify the syntax of tokens for most scanner generators. The efficiency of a automatically-generated scanner can be comparable to that of a typical hand-written scanner.

See Chapter 5 [Patterns], page 14. Zlex is such a scanner generator, automatically transforming a scanner specification into a scanner program. It accepts a scanner specification given in a file referred to as the *Zlex source file* and generates a C code file referred to as the *generated scanner file*. The Zlex source file contains (among other things) patterns specifying the syntax of tokens. For each pattern there is also a corresponding action, consisting of arbitrary C code, which is to be executed when the input matches the pattern. These actions are copied verbatim into the generated scanner. The generated scanner file needs to be compiled and linked with the rest of the program and with the Zlex library to produce an executable program.

The generated scanner provides a function which is the *main scanner function*. Whenever this scanner function is called, it scans its input stream looking for a match with any of the specified patterns. If it finds such a match, it executes the action corresponding to the pattern. If the action terminates in a return, it returns to its caller; otherwise it continues scanning the input looking for the next match. If no pattern matches the current input character, then the scanner executes a predefined action which defaults to merely echoing the unmatched character to the standard output.

Once the generated scanner has recognized a token, it is typically transformed into a small integer for processing by the rest of the program. A token typically has at least one attribute:

its *lexeme* which is the actual subsequence of the input-stream corresponding to that token. The generated scanner allows its actions to access the lexeme for the current token.

1.2 Background

Scanner-generators were popularized by the scanner-generator `lex` which is distributed with the popular Unix operating system. Unfortunately, `lex-generated` scanners had the reputation of being less efficient than hand-written scanners. An attempt to remedy this efficiency problem resulted in `flex` (see section "Flex" in *Flex - a scanner generator*) which was extremely successful in attaining this stated goal. Zlex is largely upward compatible with both `flex` and lex. Its raison d'etre is multi-faceted:

1. Even though `lex` and `flex` have a large number of features, they are not flexible enough to be used in certain situations.
2. Even though scanning is the problem in which `lex` and `flex` specialize, the lexical specifications for their own input languages are severely restrictive in not allowing the free-form input typical of modern programming languages.
3. Personal whimsical reasons: I wanted to play around with scanner-generation algorithms. I also suspect that I suffer from the NIH (not-invented-here) syndrome.

The performance of Zlex-generated scanners is comparable to those generated by `flex`, but its additional features enable tasks which would be very difficult if not impossible with `flex`.

1.3 An Example

The following Zlex program counts the number of lines, words and characters in its standard input, where a *word* is a maximal string of characters not containing a whitespace character (a whitespace character is defined to be either a space, tab or new-line).

```
/* Word-count program for stdin. */
%%
%{
  unsigned cc= 0;        /* # of chars seen so far. */
  unsigned wc= 0;        /* # of words seen so far. */
  unsigned lc= 0;        /* # of lines seen so far. */
%}
[^\t \n]+       wc++; cc+= yyleng;
[\t ]+          cc+= yyleng;
\n+             lc+= yyleng; cc+= yyleng;
<<EOF>>         printf("%d %d %d\n", lc, wc, cc);
```

The above program consists of two sections separated by a line containing only `%%`. The first section is the declarations section which is used to declare Zlex and C entities (in the above program it is empty). The second section contains the patterns along with the corresponding C actions.

Lines enclosed within decorated-braces `%{` and `%}` are copied directly into the generated C-file. In this example, the lines within decorated braces at the start of the second section are used to declare and initialize C variables local to the generated scanner function `yylex`. These variables are counters which keep track of the number of characters, words and newlines seen so far.

Line 10 in the second section consists of a pattern to match our specification of a *word*, followed by a C action. The '`[`' and '`]`' delimit a *character-class* which specifies a set of characters. A character-class is a regular expression which matches any character in that class. For example, `[\t \n]` matches any character which is a tab, blank or newline (Zlex allows C-style escape sequences starting with '`\`' within character-classes). The '`^`' at the beginning of a class denotes the negation of that character-class: hence `[^\t \n]` denotes any character except a tab, blank or newline, i.e. a non-whitespace character. The postfix operator '`+`' denotes one or more repetitions of the previous regular expression: hence `[^\t \n]+` denotes a sequence of one or more non-whitespace characters. Since Zlex always prefers the longest possible match, the specified regular expression will match "a maximal string of characters not containing a whitespace character" — namely a *word*.

The action for the first pattern simply increments the word count `wc` by 1 and increments the character count `cc` by the number of characters matched (the variable `yyleng` always contains the length of the current lexeme). Lines 11 and 12 handle blanks/tabs and newlines in a similar manner. Line 13 contains a special pattern which matches end-of-file and a action which prints out the values of the three counters.

Assuming that the above program is in the file '`wc.l`', it can be compiled and executed using a sequence of commands similar to the following:

```
$ zlex wc.l -o wc.c
$ cc wc.c -lzlex -o wc
$ wc
'Twas brillig, and the slithy toves
  Did gyre and gimble in the wabe:
All mimsy were the borogoves,
     And the mome raths outgrabe.
^D
⇒       4       23      135
$
```

The option '`-o`' for both Zlex and the C-compiler cc allows naming the output file. The first line transforms the Zlex file '`wc.l`' to a C file '`wc.c`'. The second line compiles the C-file into a executable, linking it with the Zlex library (which provides a default `main` program which merely calls the generated scanner function `yylex`). The third line runs the executable: the next six lines are input followed by an end-of-file (shown as a *^D*). This is followed by a line which is the executable's output containing the number of lines, number of words and number of characters.

1.4 Enhancements in Zlex

The enhancements provided by Zlex over `lex` and `flex` are the following:

16-bit character support

: Zlex supports the generation of scanners which process 16-bit character input. Unfortunately, due of the limitations of current editors, Zlex still requires its own source file to be specified using 8-bit characters. Hence 16-bit characters need to be specified using their character codes (possibly encapsulated within macros).

With this support for 16-bit characters, it should be possible to use Zlex to built Unicode scanners, even though Zlex does not know anything about Unicode per se.

Intra-token patterns
: These are patterns which can be recognized within other tokens. See Section 5.5 [Intra-Token Patterns], page 24. Intra-token patterns are useful for doing the pre-lexical processing required by some programming languages (for example, the deletion of a '\' followed by a newline character in C).

Column numbers
: Zlex supports obtaining the column number of the current token (in addition to the undocumented `yylineno` feature of `lex`. The method used does not require the generated scanner to test each incoming character to see if it is a newline.

Character count
: It is possible to access the count of the number of characters read from the current source file.

Sharing of code among multiple scanners
: Much of the code required for a Zlex scanner is linked in from the Zlex library. This library code can be shared among multiple scanners. The only code unique to each scanner will be a relatively small main scanner function and possibly several auxiliary functions (this will be in addition to several large data tables which will be unique to each scanner).

Ambiguous right-context patterns.
: Unlike other scanner generators, Zlex can handle ambiguous right context where the pattern to be matched overlaps with the trailing context. See Section 5.3.1 [Right Context], page 22. The worst case complexity of the method used to identify such ambiguous trailing context can be quadratic.

Interactive scanners
: As long as `<stdio>` input functions are not used ('`--stdio`' option see Section 16.3.6 [Options List], page 66), then all Zlex generated scanners can operate interactively without any performance degradation.

Code Scanners
: Zlex supports the generation of directly encoded scanners in addition to the more conventional scanners which interpret tables. At this point, this option does not appear particularly useful.

Whitespace within patterns
: There is an option '`--whitespace`' (see Section 16.3.6 [Options List], page 66) which makes Zlex more tolerant of spaces and comments in the Zlex file. This allows the Zlex programmer to format patterns so that they are more readable. See Section 5.6 [Whitespace Within Patterns], page 24.

2 The Scanning Process

When the main scanner function is entered, it initializes its data structures if it is the first time it has been called. It then enters a *select-act* loop, where it recognizes a pattern which matches a prefix of the current input, carries out the specified C action and then repeats the process on the unprocessed suffix of the input. If the action terminates in a return from the main scanner function, then when the scanner function is called again, it merely reenters the select-act loop.

The following possibilities arise as the scanner attempts to match its patterns with a prefix of its input:

1. The prefix of the input matches one or more patterns, but it is possible that a longer prefix could also match some patterns. In that case, unless one of the matched patterns is a intra-token pattern (see Section 5.5 [Intra-Token Patterns], page 24), the scanner looks for the longer match.
2. A single pattern matches a prefix of the input and there is no possibility that a longer prefix could match any pattern. In that case, the scanner executes the action associated with the pattern.
3. Multiple patterns match a prefix of the input and there is no possibility that a longer prefix could match any pattern. In that case, the scanner uses a disambiguating rule to choose a pattern and then executes the action associated with the chosen pattern.
4. No pattern matches a prefix of the input. In that situation, the scanner executes a predefined action.

2.1 Pattern Conflicts

When multiple patterns match a prefix of the input, the scanner needs to choose between these conflicting patterns. These choices are governed by the following rules:

1. An intra-token pattern (see Section 5.5 [Intra-Token Patterns], page 24) is preferred over all other patterns and is matched as soon as it is recognized, even though it may be a prefix of another pattern.
2. A pattern which matches a longer prefix of the input is preferred over one which matches a shorter prefix of the input.
3. If two patterns match the same prefix of the input, then the pattern which occurs earlier in the Zlex source file is preferred.

Hence given the scanner specification:

```
%%
while                        'Action for keyword while.'
[[:alpha:]_][[:alnum:]_]+    'Action for an identifier.'
```

The first pattern simply matches the keyword '`while`'. The second pattern matches an identifier which starts with an alphabetic character or '`_`' and is followed by one or more alphanumeric characters or '`_`'s. If the input is '`while`', the identifier pattern would match the prefixes '`w`', '`wh`', '`whi`', '`whil`' and '`while`'; the `while` keyword pattern would match the entire input. By rules (2) and (3), the '*Action for keyword* `while`.' will be that executed by the generated scanner.

2.2 Backtracking in Zlex Scanners

As a character is scanned by a Zlex scanner, it may tentatively be matched with a pattern, but subsequently it may be discovered that the tentative pattern match is incorrect and that the character needs to be rescanned for an alternate match. *Backtracking* refers to the rescanning of characters to identify alternate matches.

Most Zlex scanners will do some backtracking under normal operation. Backtracking can also be forced by the Zlex programmer by using the special `REJECT` action (see Section 8.6 [Rejecting to the Next Match], page 39).

As a Zlex scanner scans its input, it usually looks ahead by a single character to decide which pattern it is in, and whether it has reached the end of a pattern. That single character lookahead is not always sufficient: the scanner may have to scan several extra characters before it can be sure which action to take.

Consider the following scanner which ignores an alphabetic string if it is followed by a digit, but outputs an alphabetic string in blank-separated groups of upto 4 characters when it is not followed by a digit.

```
%%
[[:alpha:]]{4}              printf("%s ", yytext);
[[:alpha:]+/[[:digit:]]     /* No action. */
.|\n                        ECHO;
```

The first pattern matches a sequence of exactly four alphabetical characters (indicated by the '`{4}`'). The second pattern matches a sequence of one or more alphabetical characters only if it is followed by a digit (indicated by the trailing context '`/[[:digit:]]`'). The final pattern matches any single character.

Consider the input line

```
abcdefg
```

The scanner will scan all the characters in '`abcdefg`' before it realizes that the newline terminating this alphabetic string is not a digit and hence the second pattern cannot match. It will match the first pattern using '`abcd`' returning '`efg`' to the input stream. As part of the action of matching the first pattern it will output '`abcd `', and will then resume scanning. It will then look at the '`efg`' it pushed back, scanning past all three characters before realizing that the input does not match either of the first two patterns. The only alternative is the third pattern which it matches, `ECHO`ing '`e`', and pushing back '`fg`'. The same sequence of overscan and pushback repeats for '`fg`' with output '`f`' and pushback '`g`'. Finally the remaining '`g`' matches the last pattern. The output is:

```
abcd efg
```

Note that the '`e`' is scanned twice, the '`f`' thrice, and the '`g`' four times.

This sort of backtracking in Zlex is not inordinately expensive, but should be avoided if possible. As illustrated by the above example, the backtracking arises because of overlapping patterns: hence overlapping patterns should be avoided as far as possible.

2.3 Default Action

By default, characters which are not matched by any pattern are `ECHO`ed to `yyout` (see Chapter 9 [Scanner Output], page 42). This default action can be suppressed by specifying the `--suppress-default` option (see Section 16.3.6 [Options List], page 66).

A scanner which marks all lines containing character sequences which look like ANSI-C trigraphs (which start with a sequence of 2 '?'s) can be generated from the following:

```
%%
.*"??".+        printf("*** %s", yytext);
```

The '.' is a regular expression which matches any character except newline; the '*' is a postfix operator which specifies 0 or more repetitions of the preceeding regular expression; the '+' is a postfix operator which specifies 1 or more repetitions of the preceeding regular expression. Hence the pattern will match only lines containing a sequence of at least two '?' followed by at least one other (non-newline) character. The action specified for that pattern prints out the contents of the matching line (`yytext`) preceeded by a mark '`*** `'. Lines which do not match the specified pattern will be handled character by character by the default action and echoed to `yyout`.

For all applications except very simple filters, usually it is not a good idea to depend on this default behavior for the following reasons:

- Since it can be confusing to a user to have an application suddenly echo characters not recognized by its scanner, every possible character should be handled explicitly by scanner patterns. At minimum, the scanner should specify a final pattern like `.|\n` (with an error action) to ensure that every character will be matched. Such patterns will also need to be provided for every exclusive start state (see Section 5.3.2.2 [Start State Patterns], page 23).
- Since the default action processes only a single character at a time, it is somewhat inefficient (see Chapter 17 [Efficiency], page 70).

3 The Structure of the Zlex Input File

A Zlex file consists of upto three sections, with each section used for different purposes. The delimiter sequence `%%` on a line by itself is used to separate sections.

The text contained in these sections is of two types:

1. Text which is absorbed by Zlex.
2. Text which is treated as a block of code to be copied into the generated scanner. This text is delimited by being enclosed within braces, decorated braces ('`%{`' and '`%}`') or by being indented (see Section 3.2 [Code Blocks], page 8).

3.1 Comments

In a Zlex file, comments enclosed within '`/*`' and '`*/`' can span multiple lines. It is also possible to have comments within patterns if the '`--whitespace`' option is used (see Section 5.6 [Whitespace Within Patterns], page 24).

3.2 Code Blocks

Code blocks which are to be copied into the generated scanner can be delimited in any one of the following ways:

C-Code
: A C-code block starts with a left-brace '`{`'. Text starting with the '`{`', upto and including the *balancing* right-brace '`}`' is copied into the generated scanner. Note that braces enclosed within C-style comments, strings or character-constants are not counted when finding the balancing '`}`'. Escaped newlines ('`\`' followed by a newline character) are recognized within C-strings, but not within other constructs.

  The knowledge of C which is used to look for the balancing '`}`' is incomplete and relatively easy to fool. For example, Zlex would get hopelessly confused if a Zlex programmer were to `#define` the macro `END` to be '`}`', and then used `END` instead of '`}`': Zlex does not know anything about C-preprocessing.

Decorated-Brace Code
: This is signalled by a line which begins with a decorated left-brace `%{`. The *rest* of the line and all subsequent lines are copied to the generated scanner until a line starting with a decorated right-brace `%}` is encountered. The delimiting `%{` and `%}` are not copied. The contents of the copied text are not analyzed at all.

Indented Code
: The text is on a line which begins with a space or a tab and the first non-blank characters are not '`{`' or '`/*`'. In that case, the rest of the line is copied to the generated scanner without any further processing, as are all immediately following indented lines.

Pattern Code
: The text follows a pattern in section 2 of the Zlex file and does not start with a '`{`'. The entire line except for the pattern and immediately following whitespace are copied to the generated scanner without any further processing as though it was indented code. Note that pattern code blocks can only be used when the option '`--no-whitespace`' has been specified (the default), as with '`--whitespace`' the pattern code block will be regarded as part of the pattern.

Note that the text within decorated-brace, indented or pattern code blocks is not analyzed in any way. This has the advantage of language independence: if Zlex were to be retargeted to generate a scanner in a language other than C, there would be no change in the specifications for these code blocks. The disadvantage is that it is impossible for Zlex to recognize the terminating delimiter for the code block when it occurs within a target language construct like a comment or string.

3.3 Declarations Section

This section contains the declarations of Zlex and C entities. It is the first section in the Zlex source file and must be present (even though it may be empty). Hence the simplest possible Zlex source is

```
%%
```

which merely copies its input to its output using the default rule (see Section 2.3 [Default Action], page 6).

Besides comments, this declarations section can contain the following:

C Declarations
: These are copied directly into the generated scanner. They can be delimited in any of the ways outlined (see Section 3.2 [Code Blocks], page 8). Traditionally, decorated braces have been used to delimit these code blocks.

Macro Definitions
: The definition of a Zlex macro consists of a line starting with the name of the macro followed by whitespace, followed by a regular expression giving the definition of the macro (see Section 5.2.7 [Regular Expression Macros], page 17).

Directives
: This consists of a line starting with a `%` character followed by an alphabetic string specifying the directive. The directives currently accepted include:

  `%option`
  : This can be followed by options as they would be specified on the command-line (see Section 16.3.6 [Options List], page 66). The options actually specified on the command-line override the options specified in the source file using this directive. `%option` directives must precede all other directives.

  `%array`
  : Is equivalent to `%option --array`.

  `%pointer`
  : Is equivalent to `%option --pointer`.

  `%s` or `%S`
  : These directives are used to declare inclusive start states. The directive can be followed by one or more space-separated start state names (see Chapter 7 [Start States], page 30).

  `%x` or `%X`
  : These directives are used to declare exclusive start states. The directive can be followed by one or more space-separated start state names (see Chapter 7 [Start States], page 30).

  Obsolete lex Directives
  : These are allowed only for backward compatibility with lex. They have no effect on Zlex and are undocumented as far as Zlex is concerned.

3.4 Rules Section

This is the second section in the Zlex file. It consists of an optional initial C-code section (see Section 3.2 [Code Blocks], page 8) followed by pattern-action rules.

If the optional initial C-code section is present, it is copied into the beginning of the generated scanner function. It can be used to declare and initialize any local variables needed by the Zlex programmer.

A pattern-action rule consists of a pattern (see Chapter 5 [Patterns], page 14) followed by a action. The possibilities for an action are:

Empty Action
: No action is specified. This means that when a token corresponding to the pattern is recognized, it is simply discarded.

C Code Block
: The action consists of the concatenation of all the following code blocks which do not have any intervening pattern (see Section 3.2 [Code Blocks], page 8). Traditionally, the action is specified by a *single* brace-enclosed C code block which starts on the same line as the pattern. When a token specified by the pattern is recognized, the specified action is carried out.

Next Pattern Action '`|`'
: This special action specifies that the action for this pattern is the same as the action for the next pattern.

3.5 Code Section

The third section of the Zlex source file consists of C code typically containing the definitions of some of the C objects declared in the first section. It is simply copied unchanged into the generated scanner file.

3.6 Line Directives

At any point in a zlex source file, outside a code block or a comment a line which looks like

`%line` *nnn* *file-name*

will pretend that the following line is line number *nnn* from *file-name*. The *file-name* is any string not containing newlines enclosed within double quotes '`"`': it may contain ANSI-C escape sequences. Both the line number *nnn* and *file-name* are optional.

A `%line` directive, like all other directives, is only recognized when it occurs at the start of a line. It is useful to track the origin of source lines when a zlex file is generated automatically from another source file by a preprocessor. The `%line` directive is similar to the `#line` directive accepted by C-preprocessors.

If a `%line` directive occurs in section 2 of the zlex source file, then it may break old `lex` or `flex` programs which would regard the character sequence '`%line`' at the start of a line as a pattern. Hence the `%line` directive is not recognized in section 2 of the Zlex source file when the '`--lex-compat`' option is specified (see Section 16.3.6 [Options List], page 66).

4 C Interface Conventions

The Zlex programmer is provided with C objects for accessing information about and controlling the operation of the generated scanner. The C entities used for this interface are functions, variables and macros. For example, Zlex provides a main scanner function with default name `yylex`; the text of the last matched token can be accessed using variables with default names `yytext` and `yyleng`; Zlex provides the C macro `REJECT` to find an alternate way to tokenize the current input.

Certain conventions are used in naming these entities. Many of these names can be changed by the Zlex programmer. Many entities also have alternate names. When we refer to an entity in this manual we usually refer to it by its *common name*, which is the way it was referred to in historical implementations.

4.1 Naming Conventions

Certain conventions are used by Zlex in choosing the names for programmer-visible C entities.

4.1.1 Macros Names

There is a canonical form for all macro names defined by Zlex in the generated scanner. A *canonical macro name* starts with the prefix '`YY_`', has all letters capitalized, and words are separated by underscores. Examples of canonical macro names are `YY_REJECT` and `YY_NEW_FILE`.

In order to retain compatibility with `lex` and `flex`, alternate names are provided for some macros. These alternate names do not meet the above conventions for canonical names. For example, `REJECT` is an alternate name for `YY_REJECT`.

Unfortunately, there is no consistency whether the call of a macro M which does not require any arguments is written as M`()` or simply M. This inconsistency arises because of the need to maintain backward compatibility with `lex` and `flex`.

4.1.2 Variable Names

By default, the names of all variables begin with the prefix '`yy`', though this prefix can be changed by using the '`--prefix`' option (see Section 16.3.6 [Options List], page 66). The default variable names usually contain only lower-case letters, though in a few cases, they also contain underscores for backward compatibility. It is possible for the user to specify an arbitrary name for any variable by defining an appropriate C-macro.

For example, the default name of the variable which holds the length of the current lexeme is '`yyleng`' (see Section 6.2 [Lexeme Length], page 26). If at scanner generation time, the programmer specifies the option '`--prefix=lex_`', then the name of the variable will be '`lex_len`'. If the programmer `#define`s the macro `YY_LENG` to be `tokLength`, then the name will be '`tokLength`'.

4.1.3 Function Names

There is only one documented function in the generated scanner file: this is the main scanner function with default name `yylex`. Its name can be changed in a manner similar to variable names (see Section 4.1.2 [Variable Names], page 11), by either specifying the '`--prefix`' option during scanner generation or by defining the macro `YY_LEX`. For example, if during scanner generation the programmer specifies the option '`--prefix=scan`', then the name of the main scanning function will be '`scanlex`'. If the programmer `#define`s the macro `YY_LEX` to be `scan`, then the name of the function will be '`scan`'.

The other documented functions are those in the Zlex library. With one exception, the names of all these library functions start with the prefix '`yy`'. They do not contain any underscores, but the first letter of each word is capitalized. It is not possible to change these names as they are precompiled into the Zlex library when it is built during installation. Examples of these library function names are `yyCreateBuffer`, `yyTopState` and `yySwitchToBuffer`.

The one exception to the rule for library function names is the function `main` which is a default main program which simply invokes `yylex()`.

4.1.4 Private Names

Some names are used by Zlex for its own internal purposes. All these private names start with the prefix '`yy`' or '`YY`', independent of the '`--prefix`' option. The user is urged to avoid using names starting with these prefixes to prevent possible name clashes.

4.2 Scope of Names

The names provided by Zlex have three kinds of scope:

Scanner Function Scope
: If a name has scanner function scope, its use is meaningful only within the main scanner function. In practice, this means that the programmer can use these names only within the actions associated with patterns and not in any other functions. Examples of names with scanner function scope are `YY_REJECT` and `yy_act`.

File Scope
: Names with file scope are meaningful only within the generated scanner file. All macro names which do not have scanner function scope have file scope.

Program Scope
: Names with program scope can be used through-out the program. All public function names and variable names have program scope. No macro name has program scope.

The effect of using a name outside its intended scope is undefined. In practice, it will usually result in a compiler error when compiling the generated scanner.

4.3 Passing the Scanner State to Zlex Library Routines

Many of the Zlex library routines need to know the current state of private scanner variables to perform their tasks. This need is met by passing the state via an opaque pointer to `void`, `typedef`'d as a `YYDataHandle`. This pointer can be found in a variable with default name `yydataP` having `extern` linkage. Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_DATA_P` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

Since this variable has external linkage, it can be accessed from files other than the generated scanner file and passed as a handle to Zlex library routines.

5 Patterns

The pattern language used for expressing the syntax of tokens is essentially the language of regular expressions, extended with constructs which allow context-dependent matching.

Note that we distinguish between *patterns* and *regular expressions*. Patterns are regular expressions augmented with context-sensitive operators. All regular expressions are patterns but not all patterns are regular expressions.

5.1 Meta-Characters

In patterns, most characters usually stand for themselves: i.e. the occurrence of a particular character in a pattern specifies that that particular character should be matched. However some characters do not stand for themselves but are special meta-characters which tell Zlex how to combine patterns. The meta-characters used by Zlex are the following:

```
\ " ( ) { } < > . ? | + * / ^ $ , -
```

If a pattern is required to match any of the above characters, then the character can be quoted by preceeding it by a backslash ‘`\`’. If a pattern is required to match a backslash, then the backslash itself can be quoted by using `\\`. Any character other than a ‘`\`’ or ‘`"`’ can also be quoted by simply enclosing it within ‘`"`’ delimiters.

There are some contexts within which the above set of meta-characters is reduced. Since it can be difficult to remember exactly which characters are special within which contexts, it is advisable for the Zlex programmer to quote all non-alphanumeric characters which are to be matched literally.

5.2 Regular Expressions

Regular expressions are a concise notation for specifying the syntax of tokens. Regular expression syntax uses the following constructs and operators.

5.2.1 Single Characters

Any character which is not a meta-character is a regular expression which matches itself.

Examples

The regular expression `A` matches the character ‘`A`’.

The regular expression `#` matches the character ‘`#`’. However, since ‘`#`’ is a non-alphanumeric character it is advisable to quote it by escaping it using a ‘`\`’ as `\#` or by enclosing it within ‘`"`’ delimiters as `"#"`.

`+` is *not* a regular expression since ‘`+`’ is a meta-character.

5.2.2 Escape Sequences

The backslash character can be used to quote characters and specify character escapes in a manner similar to C. Specifically:

- `\a`, `\b`, `\f`, `\n`, `\r`, `\t`, `\v` are regular expressions which match the characters *BEL* (bell), *BS* (backspace), *FF* (form-feed), *NL* (newline), *CR* (carriage-return), *TAB* (tab) and *VT* (vertical-tab) respectively.
- '\' followed by upto 3 octal digits is a regular expression which matches the character with character code equal to the octal number.
- '`\x`' followed by a hexadecimal number is a regular expression which matches the character with character code equal to the hexadecimal number.
- '\' followed by any other character (except newline) is a regular expression which matches that character.

Examples

'`\x81`' matches the character whose decimal character code is 129.

'`\400`' should match the character whose decimal character code is 256. However, this is invalid since Zlex currently only supports character codes in the range 0–255.

5.2.3 Character Classes

A character class is a regular expression denoting a set of characters which will match any single character in the set. Character classes are specified by listing the members of the set enclosed within '`[`' and '`]`'. It is possible to denote ranges of characters in a character class by using *lo*-*hi*, where *lo* and *hi* are the first and last characters in the range. A negated character class is a character class whose first character is '`^`' and denotes the complement of the character class. Escape sequences (see Section 5.2.2 [Escape Sequences], page 15) are recognized within a character-class. The rules for recognizing special characters within character-classes are different from those for other patterns and are given below:

'\' This is the only character which is special through-out a character class. It is used to specify escape sequences. It can be included by quoting it with itself as '\\'.

'`^`' This is special only at the beginning of a character class (immediately after the '`[`' signalling the start of the character class). It signals the start of a negated character class.

'`-`' This is special and used to denote character ranges when used *within* a character class; it is not special and stands for itself when used at the beginning of the class (immediately after the '`[`' signalling the start of the character class or immediately after the negated-class operator '`^`') or at the end (just before the terminating '`]`').

'`]`' This is special when used after the beginning of the character class and terminates the character class; it is not special and stands for itself when used right at the beginning of the class (immediately after the '`[`' signalling the start of the character class or immediately after the negated-class operator '`^`').

Any whitespace character (except newline) is significant within a character class and specifies a character within the class. Comments are never recognized within a character class. Hence the character class `[/*a */ a]` would contain the four characters ' ', '/', '*' and 'a'. Newlines are not allowed directly within a character class and must be specified using the escape sequence '`\n`'.

Examples

`[lL]` matches any one of the characters 'l' or 'L'.

`[0-9a-fA-F]` matches any hexadecimal digit.

`[^0-9a-zA-Z]` matches any non-alphanumeric character.

`[-+]` or `[+-]` matches any one of the characters + or '-'.

`[\t \n]` matches a tab, space or newline character.

5.2.4 Named Character Classes

Specifying lowercase alphabetic characters using a pattern like `[a-z]` may not work with character sets other than ASCII as the character codes for lower-case letters may not always be contiguous in the underlying character set. To remedy this problem, POSIX introduced named character classes of the form `[:`*Name*`:]`. The class represented by `[:`*Name*`:]` is precisely the set of characters *c* for which the standard C-library function `is`*Name*`(`*c*`)` returns non-zero.

`[:alnum:]`
: An upper or lower-case alphabetic character or a decimal digit.

`[:alpha:]`
: An upper or lower-case alphabetic character.

`[:blank:]`
: A space ' ' or tab '`\t`' character.

`[:cntrl:]`
: A control character.

`[:digit:]`
: A digit.

`[:graph:]`
: Any printing character (not a control character) other than a space character.

`[:lower:]`
: A lower-case letter.

`[:print:]`
: Any printing character (not a control character), including a space character.

`[:punct:]`
: A punctuation character which is a printing character but not a space character or alphanumeric character.

`[:space:]`
: A space ' ', tab '`\t`', carriage return '`\r`', newline '`\n`', vertical tab '`\v`' or form-feed '`\f`' character.

`[:upper:]`
An upper-case letter.

`[:xdigit:]`
A hexadecimal digit.

These named classes cannot occur directly in a pattern but only as members of a character class. Hence `[:alpha:]` is not a valid pattern but `[[:alpha:]]` is.

5.2.5 Any Character Except Newline

'`.`' is a regular expression which matches any character except newline.

Example

`.|\n` is a pattern which matches any character ('`|`' is a regular expression operator specifying the union of two regular expressions).

5.2.6 String Quoting

A string of characters enclosed within double-quotes '`"`' is a regular expression which matches that string of characters. Escape sequences are recognized within the string. The characters which are special within the string are '`\`' (used for escaping characters) and '`"`' (used for terminating the string) and must be escaped with a preceeding '`\`' if they are to be included within the string. A newline cannot be contained directly within a string regular expression, but must be specified using a escape sequence as '`\n`'.

Examples

`"[]"` matches the string consisting of the two characters '`[]`'.

`"\x30\0\"\\\n"` matches the string containing five characters: the first character has the hexadecimal character code `30`, the second character has the character code 0, the third and fourth characters are '`"`' and '`\`' respectively, and the last character is the newline character.

5.2.7 Regular Expression Macros

A macro can be defined in section 1 of the Zlex file by specifying a macro name M followed by a regular expression R. Any use of `{`M`}` within a regular expression is expanded to `(`R`)`. Note that the definition is restricted to be a *regular expression*; it cannot be a *pattern* containing any context operators (see Chapter 5 [Patterns], page 14).

A macro name can contain alphanumeric characters or '`_`' or '`-`', but must start with a alphabetical character or '`_`'. When the macro is defined in section 1 of the Zlex file, the name must occur at the beginning of a line. This must be followed by whitespace followed by the macro definition on the same line. The regular expression comprising the macro definition can contain calls to other macros, including those which have not yet been defined. It is an error for a macro to contain a call to itself, either directly or indirectly via calls to other macros. A macro is not expanded until a call to the macro is encountered in section 2 of the Zlex file.

Whitespace is allowed within the defining regular expression (see Section 5.6 [Whitespace Within Patterns], page 24). When a macro name is used within braces, no whitespace or comments are allowed within the braces. This makes it easier for Zlex to disambiguate a macro use, from the start of a block of C-code.

Macros can be used to make patterns more readable if the Zlex programmer chooses suitable mnemonic macro names. They do not add anything to the expressive power of the pattern language since every use of a macro name is fully equivalent to its defining regular expression enclosed in parentheses.

Examples

In old `lex` and `flex` programs one often encounters macros like the following:

```
alpha           [a-zA-Z]
```

This is not portable across all character set and is no longer necessary since named character classes (see Section 5.2.4 [Named Character Classes], page 16) can be used instead.

5.2.8 Optional Regular Expression

If R is a regular expression which matches r', then R`?` is a regular expression which matches zero or one occurrences of r'.

Examples

`[-+]?` denotes an optional sign.

`\.?` can be used to denote an optional decimal point.

5.2.9 Zero or More Repetitions

If R is a regular expression which matches r', then R`*` is a regular expression which matches zero or more repetitions of r'. '`*`' is often referred to as the *Kleene-closure* or simply *closure* operator.

Examples

`[[:alnum:]]*` will match a sequence of zero or more alphanumeric characters.

`[*]*` will match a sequence of 0 or more '`*`'s.

5.2.10 One or More Repetitions

If R is a regular expression which matches r', then R`+` is a regular expression which matches one or more repetitions of r'.

Examples

`[[:xdigit:]]+` will match a sequence of one or more hexadecimal characters.

`.+` will match the rest of the current line provided it is nonempty (recall that '`.`' matches any character except a newline).

5.2.11 Counted Repetition

If R is a regular expression which matches *r'*, then R`{`*lo*`,`*hi*`}` where *lo* and *hi* are positive integers matches *lo* through *hi* occurrences of *r'*; R`{`*num*`}` where *num* is a positive integer matches exactly *num* occurrences of *r'*. R`{`*lo*`,}` matches at least *lo* occurrences of *r'*.

No whitespace or comments are ever allowed between the starting '`{`' and the first digit of the repetition count. This restriction makes it easier for Zlex to disambiguate counted repetition from a C-code block.

Examples

`[[:alpha:]]{1,6}` matches any nonempty string of alphabetic characters upto 6 letters long.

`[[:digit:]]{3}` matches exactly three digits.

`[[:alnum:]]{5,}` matches a sequence of at least 5 alpha-numeric characters.

5.2.12 Concatenation

If R and S are regular expressions which match *r'* and *s'* respectively, then their juxtaposition RS is a regular expression which matches the concatenation of *r'* and *s'*.

Examples

The regular expression `[a-zA-Z_][-0-9a-zA-Z_]*` can be used to denote a Zlex macro name.

The regular expression `0[0-7]*` can be used to denote a octal number in ANSI-C.

5.2.13 Union

If R and S are regular expressions which match *r'* and *s'* respectively, then $R|S$ is a regular expression which matches either *r'* or *s'*.

Examples

The regular expression `[0-9]+[lL]?[uU]?|[0-9]+[uU]?[lL]?` denotes a ANSI-C integer which consists of a sequence of digits followed optionally by '`l`' or '`L`' (denoting *long*), or by '`u`' or '`U`' (denoting *unsigned*) in either order.

The above can be expressed slightly more succinctly by using parentheses to factor out the `[0-9]+`, as `[0-9]+([lL]?[uU]? | [uU]?[lL]?)`.

5.2.14 Operator Precedence

The precedence of the operators defined above is shown below. Operators which are grouped together have the same precedence. The groups are in order of decreasing precedence.

> Postfix operators: Closure operator '`*`', one or more operator '`+`', optional operator '`?`', counted repetition '`{`*lo*`,`*hi*`}`'.
>
> Concatenation by juxtaposition.
>
> Union operator '`|`'.

5.2.15 Grouping Regular Expression

If R is a regular expression, then `(`R`)` is also a regular expression equivalent to R. The parentheses are used for grouping regular expressions to override the default precedence of the regular expression operators.

Example

> `(a|b)c` matches either the string '`ab`' or '`ac`'. If the parentheses were omitted and the pattern was written as `a|bc`, then the pattern would match the string '`a`' or the string '`bc`'.

5.2.16 Operator Independence

Some of the operators are merely syntactic sugar and can be expressed in terms of the other operators. For example:

R`+` ⇒ *RR*`*`
R`{2,4}` ⇒ *RR*`|`*RRR*`|`*RRRR*
R`{2,}` ⇒ *RRR*`*`
`[/*-]` ⇒ `"/"|"*"|"-"`

5.2.17 Regular Expression Examples

The following examples use Zlex regular expressions to specify the syntax of comments in various programming languages.

5.2.17.1 Ada Comments

An Ada comment starts with the characters '`--`' and continues to end-of-line. A suitable pattern is

```
"--".*
```

5.2.17.2 Pascal Comments

Pascal has two commenting conventions. One of them is to enclose the body of a possibly multi-line comment within braces '{' and '}'. The body of the comment should not contain any '}' characters. An incorrect attempt to write a regular expression for such a comment is:

```
"{"(.|\n)*"}"            /* Wrong. */
```

The problem is that the regular expression does not enforce the restriction that the body of the comment should not contain any '}' characters. In fact, with Zlex's rule for preferring the longest match, the above regular expression will interpret all the text between the first '{' and the last '}' in a Pascal file as a comment!

A correct regular expression is:

```
"{"[^}]*"}"
```

Though this is correct, it has the disadvantage that it forces Zlex to save the text of a long comment. For more efficient ways of processing long comments, see Section 7.6 [Start States Example], page 32.

5.2.17.3 C Comments

A comment in C consists of any character sequence not containing the subsequence '*/' surrounded by '/*' and '*/'. The following is an incorrect attempt (which was published in a book) at writing a regular expression for a C comment (the spaces in the regular expression have been added for readability and can be ignored by Zlex):

```
("/*" "/"* ([^*/] | [^*]"/" | "*"[^/])* "*"* "*/") /* Wrong. */
```

Analyzing the above expression we realize that the expression within the inner parentheses corresponds to the body of the comment except for a possibly empty prefix containing only '/'s and a possibly empty suffix containing only '*'s. Analyzing the inner expression further, it specifies 0 or more repetitions of

- Any character except a '/' or '*'.
- Any character except a '*' followed by a '/'.
- A '*' followed by any character except a '/'.

Though the above seems correct, it is not. A counterexample is the valid comment '/**1/*/' which does not match the above regular expression. The problem is caused by ignoring the possible overlap between the subpatterns '[^*]"/"' and '"*"[^/]' where the negated character classes in both patterns may need to match the same character ('1' in the counterexample).

A solution which is claimed to be correct is the following:

```
("/*" [^*]* "*"+ ([^/*][^*]*"*"+)* "/")
```

The [^*]* deals with that prefix of the comment body which does not contain any '*'s. When a '*' occurs, we need to have a sequence of one or more of them ("*"+). The inner closure ([^/*][^*]*"*"+)* specifies that the sequence of '*'s be followed by 0 or more repetitions of text not starting with '/' or '*' and terminating in a sequence of '*'s. So irrespective of the number of iterations of the inner closure, the input character at the end of the closure must be a '*'. Hence a further '/' in the input terminates the comment.

Once again, this is not the recommended way to specify C comments in Zlex because of the possibly excessive growth of the text saved by Zlex. For the recommended method, see Section 7.6

[Start States Example], page 32. As the preceeding remark makes clear, these complicated regular expressions are mainly useful as exercises with which to plague students. What is more interesting is the non-eureka process by which these expressions may be constructed, but that is beyond the scope of this manual.

5.3 Context Operators

Sometimes it is necessary to match regular expressions only in certain contexts. This can be achieved by patterns which use additional syntax to specify the context in which a regular expression should be matched.

5.3.1 Right Context

Right context allows an input sequence to match a regular expression only if the input which immediately follows the matching sequence satisfies certain restrictions. Generalized right context allows the restrictions to be expressed via an arbitrary regular expression. End-of-line right context is a special case of generalized right context.

5.3.1.1 Generalized Right Context

One can define a real number in Modula-2 by means of the following macro definitions:

```
digit           [0-9]
sign            [-+]
exp             E{sign}?{digit}+
real            {digit}+\.{digit}*{exp}?
```

The above definition allows numbers like '`22.`' with an empty fraction and exponent. Unfortunately, constructs like '`1..10`' are commonly used in Modula-2 to indicate subranges, and should be scanned as three tokens '`1`', '`..`' and '`10`'. However since Zlex always prefers the longest match, the effect of the pattern `{real}` on the input '`1..10`' will be to scan the first token as `1.`, which is wrong for Modula-2. One solution is to scan a number as a real only if it is not followed by a '`.`' character. This can be achieved by suffixing the above pattern with a special right-context construct which imposes this restriction:

```
{real}/[^.]
```

'`/`' is the right-context operator. If R and C are arbitrary regular expressions, then R/C is a pattern which matches input R' iff R matches R' and the input after R' matches C. Note that the input which matched C is available to be rescanned.

Returning to the Modula-2 example, `{real}/[^.]` will not match the input `1..10`. Instead the `1` can be matched by a pattern for an integer, the `..` can be matched by an appropriate pattern, and the `10` can be matched by the pattern for an integer. On the other hand if the input is `1.+2`, then `{real}/[^.]` will match the '`1.`', since '`+`' matches `[^.]`. The '`+`' will then be rescanned and can be matched by a suitable pattern.

There are no restrictions on the regular expressions on either side of the '`/`'. Unfortunately, this freedom allows ambiguous patterns like `[a-zA-Z0-9]+/[0-9]+"#"`, for which there are multiple ways to match an input like '`aA12b123#`'. Specifically, the prefixes '`aA12b`', '`aA12b1`' and '`aA12b12`' all match the specified pattern. It is necessary for Zlex to use a disambiguating rule to resolve the ambiguity: it always matches the longest prefix. For the above example, Zlex would match '`aA12b12`'. Note that other scanner generators may get confused by similar patterns.

5.3.1.2 End of Line Right Context

It is sometimes necessary to match a regular expression *R* only at the end of a line. This can be achieved by using the pattern *R*`/\n`. The `$` end-of-line anchor is available to abbreviate this pattern to *R*`$`. The '`$`' character is special only at the end of a pattern.

5.3.1.3 Right Context Restrictions

A single pattern can contain only a single instance of a right-context operator. Hence a pattern like `[A-Za-z0-9]+/[\t ]+$` which attempts to recognize an alphanumeric word only when it occurs at the end of a line is illegal, since '`$`' provides an additional right-context operator. Instead, the pattern can be written as `[A-Za-z0-9]+/[\t ]+\n` which is legal.

5.3.2 Left Context

In a Zlex scanner, it is possible to use two methods for allowing left-context to influence a match. The first is useful when the interpretation of a token is affected by whether or not it is at the start of a line. The second is more general, and allows encapsulating the left-context into a state which selects a subset of the patterns which are allowed to match.

5.3.2.1 Start of Line Pattern

In a C preprocessor, '`#`' signals a preprocessor directive only if it occurs at the beginning of a line (preceeded optionally by whitespace). A pattern which recognizes a '`#`' only when it signals a preprocessor directive is the following:

```
^[\t \v\f]*\#
```

The '`^`' is the *start-of-line* anchor: the following pattern is matched only if the previous character was a newline character.

When a scanner uses one or more patterns containing the start-of-line anchor '`^`', it is possible to query and set the current start-of-line condition during scanning. See See Section 8.10 [Querying Beginning of Line], page 41 and See Section 8.11 [Setting Beginning of Line], page 41.

5.3.2.2 Start State Patterns

The generated scanner can be in one of several different states before it starts scanning the input for the next token: these states are known as *start states*. The Zlex programmer is required to name and declare all start states and can control the transitions between start states by using special actions. In a particular start state, only a subset of the patterns is used to recognize tokens; exactly which subset is to be selected is indicated by qualifying each pattern with the set of start states in which that pattern should be active. More information on start states can be found in Chapter 7 [Start States], page 30.

5.4 End of File Patterns

The special pattern `<<EOF>>` (which cannot contain any internal whitespace or comments) is used to match the end of the input file. It may be qualified with a set of start conditions using a syntax identical to that used for qualifying regular expressions. The end-of-file pattern is useful for doing special processing at end-of-file. The following example shows how it can be used to signal that a construct like a comment was not terminated before end-of-file was encountered:

```
<COMMENT><<EOF>>   fprintf(stderr, "EOF detected within comment.");
```

It is assumed that the scanner entered a `COMMENT` start state when a comment was encountered.

For special Zlex actions which can be used in `<<EOF>>` patterns, see Chapter 12 [End-of-File and Termination], page 51.

5.5 Intra-Token Patterns

Intra-token patterns are useful to do pre-lexical processing during the scanning process. More information on intra-token patterns can be found in Chapter 11 [Using Intra-Token Patterns], page 49.

5.6 Whitespace Within Patterns

Whitespace is allowed in Zlex patterns when the option '`--whitespace`' is specified (see Section 16.3.6 [Options List], page 66). Unfortunately, it is not possible to allow totally free-format input in order to retain as much backward-compatibility as possible with `flex` and `lex`. The rules for how whitespace within different constructs are as follows:

What is whitespace?
: Blanks, tabs and comments are regarded as equivalent whitespace. Newlines are treated somewhat differently.

Character classes and strings
: Whitespace (except newlines) is always significant and is included in the pattern. Newlines are not allowed directly in strings or character classes, but must be escaped. The effect of whitespace within these constructs is independent of the '`--whitespace`' option. This behavior is identical to the behavior of other common text processing utilities.

End-of-file Pattern
: Whitespace is never allowed in the end-of-file token `<<EOF>>` which is regarded as an indivisible token. This behavior is independent of the '`--whitespace`' option.

Macro Calls
: Whitespace is never allowed in macro calls of the form `{`*macro*`}` which are regarded as indivisible tokens. This behavior is independent of the '`--whitespace`' option.

Regular Expressions in Macro Definitions
: Spaces and tabs are always allowed and ignored within the regular expressions used for macro definitions in section 1 of the Zlex file. When the option '`--whitespace`' is not specified, the pattern is terminated by the first newline; when the option '`--whitespace`' is specified, newlines are allowed and ignored *provided they occur within parentheses*.

Patterns in Section 2
: When the '`--whitespace`' option is not specified, a pattern in section 2 is terminated by the first non-quoted whitespace which is not within a string or character class. When the '`--whitespace`' option is specified, whitespace which is not within a character class or string in a section 2 pattern is allowed and ignored; the pattern is terminated by the start of a brace-enclosed action or by the first newline which is not within parentheses.

One consequence of these rules is that when the '`--whitespace`' option is used, it is not possible to include a action for a pattern in section 2 of the Zlex file without enclosing the action within braces.

6 Accessing the Current Lexeme

Two variables with external linkage allow accessing the characters constituting the last matched token, as well as its length.

6.1 Current Lexeme Text: yytext

This is a variable which enables access to the sequence of characters which constitute the lexeme of the current token. This sequence of characters is always terminated by a *NUL* '\0' character. Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro YY_TEXT to the new name. Alternatively, the prefix used for the name can be changed by using the '--prefix' option (see Section 4.1.2 [Variable Names], page 11). Its default declaration depends on whether the option '--pointer' or '--array' is used (see Section 16.3.6 [Options List], page 66). When '--pointer' is used, its default declaration is char *yytext; when '--array' is used, its default definition is equivalent to

```
char *yytext[YYLMAX];
```

where YYLMAX is a macro which gives the size of the array. YYLMAX can be defined by the user in section 1 of the Zlex file if a value different from the default value (8192) is desired.

When yytext is declared to be an array and the length of a matched lexeme is greater than the value of YYLMAX, then the yytext array will silently overflow with unpredictable results. When yytext is declared to be a pointer, there is no possibility of overflow as the lexeme text is maintained within the scanner's buffer (which is grown dynamically as needed).

A scanner in which yytext is declared to be a pointer is usually faster than one in which it is declared to be an array. This fact, coupled with the overflow problem mentioned previously, make a %array declaration fairly useless except for backward compatibility with lex.

The Zlex programmer should always treat yytext as a read-only variable.

The following program fragment shows a pattern-action pair which matches the occurrence of an identifier at the beginning of a line and saves it in dynamic memory pointed to by the variable text.

```
%%

[[:alpha:]_][[:alnum:]_]*
  {  text= malloc(yyleng + 1); /* +1 for terminating NUL. */
     if (!text) { 'Call an error routine.' }
     strcpy(text, yytext);
  }
```

6.2 Current Lexeme Length: yyleng

This variable with declaration int yyleng holds the length of the current token. The *length of a token* is the number of characters in the lexeme of the token (not counting any terminating '\0'). Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro YY_LENG to the new name. Alternatively, the prefix used for the name can be changed by using the '--prefix' option (see Section 4.1.2 [Variable Names], page 11).

The Zlex programmer should always treat **yyleng** as a read-only variable.

The following program produces a histogram of word-lengths, where a word is defined to be a maximal sequence of characters not containing a space, tab or newline.

```
%{
enum { MAX_WORD_LEN= 10 };
static unsigned freq[MAX_WORD_LEN];
%}
%%
[^\t \n]+        { if (yyleng > MAX_WORD_LEN) {
                      'Signal error;'
                   }
                   else {
                     freq[yyleng]++;
                   }
                 }
[\t \n]+         /* No action. */
<<EOF>>          { unsigned i;
                   for (i= 0; i < MAX_WORD_LEN; i++) {
                     printf("%d: %d\n", i, freq[i]);
                   }
                 }
```

6.3 Concatenating Tokens: `yymore`

If an action contains a call to the `yymore()` macro, then the lexeme for that token is prefixed to the lexeme of the next token recognized. Effectively, this allows the programmer to recognize subtokens within a larger token. The canonical form `YY_MORE()` can also be used instead. The library function `yyMore(YYDataHandle)` can also be used from files other than the generated scanner file.

For example, let us suppose that an application requires printing out the input lines in reverse order, and printing the total number of words in the input. Whenever a token within a line is recognized the scanner executes a `yymore` action: hence when the '`\n`' terminating a line is finally matched, `yytext` contains the text for the entire line. This is saved in a stack of lines using a function `pushLine()` shown below. Finally at `<<EOF>>` this stack is traversed with the lines being printed in reverse order.

```
%{
#include <stdio.h>
#include <stddef.h>

typedef struct LineStruct {
  struct LineStruct *last;
  char *text;
} LineStruct;

static LineStruct *pushLine(LineStruct *lines,
                            const char *text, int textLen);

%}
%%
  /* Declare local variables. */
  int wc= 0;
  LineStruct *lines= NULL;
[\t ]+          yymore();
[^\t \n]+       wc++; yymore();
\n              lines= pushLine(lines, yytext, yyleng);
<<EOF>>         { LineStruct *p;
                  for (p= lines; p; p= p->last) fputs(p->text, stdout);
                  printf("# of words= %d\n", wc);
                }
%%
static LineStruct *
pushLine(LineStruct *lines, const char *text, int textLen)
{
  char *const savedText= malloc(textLen + 1);
  LineStruct *const lineP= malloc(sizeof(LineStruct));
  if (!savedText || !lineP) {
    fprintf(stderr, "Out of memory.\n"); exit(1);
  }
  strcpy(savedText, text);
  lineP->text= savedText; lineP->last= lines;
  return lineP;
}
```

A log of running the scanner generated from the above follows:

```
"Beware the Jabberwock, my son!
  The jaws that bite, the claws that catch!
 Beware the Jubjub bird, and shun
    The frumious Bandersnatch!"
^D
⇒
    The frumious Bandersnatch!"
 Beware the Jubjub bird, and shun
  The jaws that bite, the claws that catch!
"Beware the Jabberwock, my son!
# of words= 22
```

7 Start States

Start states allow the behavior of the scanner to depend on the left context within the input. Several actions allow the scanner to control or access its current start state.

7.1 Start State Types

Start states are of two types: *exclusive* and *inclusive*. When a *exclusive start state* is active, only those patterns whose qualifying start states include the name of that start state are selected. See Section 7.6 [Start States Example], page 32 for an example where start states are used to process C-style comments. When a *inclusive start state* is active, patterns which do not have any qualifying start states at all are also selected in addition to the patterns whose qualifying start states include the name of the active start state. This implies that a pattern with no qualifying start states is equivalent to the same pattern qualified by **all** the *inclusive* start states. Inclusive start states are useful to factor out the commonality within different start states (see Section 7.7 [Using Inclusive Start States], page 33).

Start state qualified patterns can occur only in section 2 of the Zlex file. The syntax for qualifying patterns is to prefix the pattern with the names of the start states separated by commas '`,`', and enclosed within angle brackets '`<`' and '`>`'. The following patterns are examples of start state qualified patterns:

```
<INITIAL>"/*"
<COMMENT>"*/"
<INITIAL,COMMENT>\n
```

where it is assumed that `INITIAL` and `COMMENT` are suitably declared start states.

7.2 Start State Declarations

Before a start state name can be used in section 2 of the Zlex file, it must be declared in section 1 of the Zlex file. An exclusive (inclusive) start state is declared in section 1 by a line starting with `%x` (`%s`) or `%X` (`%S`) followed by whitespace followed by the name of the start state on the same line. Multiple start states of the same type can be declared by including multiple names on the same line separated by space. The characters allowed within a start state name are identical to those allowed in a macro name: a sequence of alphanumeric or '`_`' or '`-`' characters starting with an alphabetic or '`_`' character.

The following are examples of start state declarations:

```
%x      COMMENT C_CODE          /* Exclusive start states. */
%s      RANGE SS_USE            /* Inclusive start states. */
```

In the generated scanner, the programmer declared start state names are `#define`d to be small integers. Hence the programmer should not use these names in any other context. All Zlex generated scanners predefine an inclusive start state called `INITIAL` which is the initial start state for the scanner when it is first called. `INITIAL` is `#define`d to be `0`; The user should not make any assumptions about the assignment of integers to other start states, and should always refer to them using their symbolic names.

7.3 Entering a Start State: BEGIN

The macro BEGIN is used to set the current start state. To set the current start state to one with name *ss*, BEGIN(*ss*) can be used. For backwards compatibility reasons, BEGIN *ss* without the parentheses can also be used.

The canonical name YY_BEGIN can be used instead; unlike BEGIN, the parentheses are always required. To begin start-state *ss* the form YY_BEGIN(*ss*) is used.

Since the INITIAL start state (see Section 7.2 [Start State Declarations], page 30) is #defined to be 0, BEGIN 0 is synonymous with BEGIN INITIAL.

The following example shows how inclusive start states can be used to recognize numbers in different bases depending on a specific directive. The base is set by a '%bin', '%oct' or '%hex' directive which must occur at the start of a line.

```
%s BIN OCT HEX
%%
^"%bin"                 BEGIN BIN;
^"%oct"                 BEGIN OCT;
^"%hex"                 BEGIN HEX;
<BIN>[01]+              'Action for a binary number.'
<OCT>[0-7]+             'Action for a octal number.'
<HEX>[a-fA-F0-9]+       'Action for a hexadecimal number.'
'Other non-qualified patterns.'
```

7.4 Accessing the Current Start State: YY_START

The macro YY_START returns the current start state (an unsigned integer). YYSTATE is synonymous with YY_START.

Accessing the current start state using YY_START allows the Zlex programmer to use start-state subroutines. For example, in the scanner for Zlex, C-style comments are allowed within several constructs. These comments are processed using an exclusive start state COMMENT (see Section 7.6 [Start States Example], page 32). When we are in a construct and see the start of a comment, we do a BEGIN COMMENT *after* saving the current start state in a global variable, say commentRet. Then when in the COMMENT state we see the end of the comment we do a BEGIN commentRet, which puts us back in the start state in which we originally saw the comment.

In the above situation, we could predict exactly how many start states we need to save at any time (exactly one). That may not be possible in general. Start state stacks may be used in such situations (see Section 7.5 [Start State Stacks], page 31).

7.5 Start State Stacks

In a Zlex scanner, start state stacks can be created and manipulated using three routines.

7.5.1 Pushing a Start State: `yy_push_state`

The macro `yy_push_state(`*ss*`)` pushes the current start-state on top of the start state stack and does a `BEGIN` *ss* action. The canonical name `YY_PUSH_STATE` may be used synonymously. From files other than the generated scanner, the programmer can call the Zlex library function `yyPushState` with prototype:

```
void
yyPushState(YYDataHandle d, YYState ss);
```

to push the current start state on the start state stack of the scanner specified by `d` and enter start state `ss`.

7.5.2 Popping the Start State Stack: `yy_pop_state`

The macro `yy_pop_state()` sets the current start state to the state on top of the start state stack and pops the start state stack. The canonical name `YY_POP_STATE` may be used synonymously. From files other than the generated scanner, the programmer can call the Zlex library function `yyPopState` with prototype:

```
void
yyPopState(YYDataHandle d);
```

to set the current start state to the state on top of the start state stack of the scanner specified by `d` and pop its start state stack.

7.5.3 The Top of the Start State Stack: `yy_top_state`

The macro `yy_top_state()` return the start state on top of the start state stack. The start state stack is not changed. The canonical name `YY_TOP_STATE` may be used synonymously. From files other than the generated scanner, the programmer can call the Zlex library function `yyTopState` with prototype:

```
YYState
yyTopState(YYDataHandle d);
```

to return the start state on top of the start state stack of the scanner specified by `d`.

7.6 Start States Example: C comments

The following example is the recommended way to process C-style comments using Zlex. It illustrates the use of exclusive start states to allow the scanner to process the comments in reasonable line-sized chunks.

When the generated scanner sees a '`/*`' it enters a exclusive start state named `COMMENT` where it is looking for the terminating '`*/`'. Because `COMMENT` is an exclusive start state, Zlex will ignore all patterns not qualified by `COMMENT` when in the `COMMENT` state.

```
%x                  COMMENT         /* Declare start-state. */
%%
"/*"                BEGIN COMMENT;
<COMMENT>"*/"       BEGIN INITIAL;
<COMMENT>[^*\n]+
<COMMENT>\n
<COMMENT>"*"+/[^/]
```

Line 1 declares the identifier `COMMENT` to be an exclusive start-state. Line 3 has a pattern for recognizing the '`/*`' which begins a comment. Since the pattern is not qualified by any start states, it will be active in all inclusive start states: namely `INITIAL`. Its action uses the special Zlex macro `BEGIN` (see Section 7.3 [Entering a Start State], page 31) to enter the special `COMMENT` state.

Line 4 recognizes the terminating '`*/`' only when the scanner is in the `COMMENT` state. Its action is to change the scanner state back to `INITIAL`. Once the scanner is back in the `INITIAL` state, the patterns prefixed by `COMMENT` are ignored, and other patterns (not shown) become active.

Line 5 recognizes any prefix of a comment line which does not contain '`*`'. Note the use of `\n` in the negated character class; if we had simply used the regular expression `[^*]+`, then it could conceivably match several lines of text — something which is undesirable as the `yytext` saved by the scanner may become excessively large.

Lines 6 and 7 recognize those portions of a comment not recognized by line 5. Line 6 recognizes a newline occurring within a comment. The given code does not have any action but if the scanner is keeping track of line numbers, an appropriate action would be to increment a line number counter. Line 7 recognizes '`*`'s occurring within a comment which are not followed by a '`/`'. We use `"*"+/[^/]` rather than simply `"*"/[^/]`, as it is always desirable to scan as large a token as possible to reduce scanner overhead.

7.7 Using Inclusive Start States

Inclusive start states are not strictly necessary, but are useful to capture the semantics of a state which is very similar to other inclusive states, in that all except a few tokens are processed identically. The implementation of Zlex itself provides a practical example of such uses of inclusive start states. In Zlex, the '`,`' character is usually an ordinary character: within a pattern it usually stands for itself specifying that a comma should be matched. The exceptions to this rule are:

- A comma is used to separate start state names within the qualifying start state list for a pattern,
- A comma is used within the counted repetition operator `{`*lo*`,`*hi*`}` to separate *lo* from *hi*.

The Zlex scanner defines two inclusive start states `RANGE` and `SS_USE` which return a comma as a special token. A highly simplified version of the code is shown below.

```
%s      RANGE           /* Start state for counted repetition. */
%s      SS_USE          /* Start state for start state list. */

%%

"{"                     BEGIN RANGE;

"<"                     BEGIN SS_USE;

<RANGE,SS_USE>","       return ',';

.                       return CHAR_TOK;
```

If a comma is encountered when the scanner is in either one of the states `RANGE` or `SS_USE` it is returned as the special token `','`. Otherwise it is simply returned as a '`CHAR_TOK`'. Note that any other characters will be matched using the patterns without any start-state qualifications: in the currently popular object-oriented parlance, `RANGE` and `SS_USE` *inherit* behavior from the patterns without start-state qualifications.

8 Input in a Zlex Scanner

A Zlex scanner reads its input from a `stdio FILE` pointer with default name `yyin`. For performance reasons, it buffers its input. Normally, it is the main scanner function which reads its input directly from the buffer, but it is also possible for the Zlex programmer to read directly from the buffer using the `input` macro. It is possible for the programmer to specify the method by which the scanner fills its buffer by defining the `YY_INPUT()` macro. The programmer is allowed to modify the characters in the scanner buffer and backtrack to alternate matches with the prefix of the input. It is also possible for the Zlex programmer to query the position in the current input stream, or the current line or column number. When patterns involving the start-of-line anchor '^' have been used, Zlex makes it possible to query and set the current start-of-line condition.

8.1 Input File Pointer: yyin

`yyin` is the default name of the variable with declaration `FILE *yyin` which Zlex uses to read its input. Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_IN` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

When the scanner function is first entered it initializes `yyin` to `stdin`, unless the user has already initialized it to a non-`NULL` `FILE` pointer. So if the generated scanner should read from a file other than the standard input, the programmer need only initialize `yyin` to a suitable `FILE` pointer. For example, the following main program illustrates how to setup the scanner to read from the file specified by the first command-line argument.

```
%{
#define YY_IN inFile    /* Use inFile instead of yyin. */
%}
%%
'Patterns go here.'
%%
int
main(int argc, const char *argv[])
{
  if (argc < 2) {
    'Usage error.'
  }
  if (!(inFile= fopen(argv[1], "r"))) {
    'File open error.'
  }
  return yylex();       /* Call generated scanner function. */
}
```

8.2 Direct Input: input

The `input()` macro returns the next character from the input buffer, returning -1 if end-of-file is encountered. If C++ is being used, then the alternate name `yy_input()` is used instead. The canonical name `YY_GET()` is also recognized. The Zlex library function `int yyGet(YYDataHandle)` can also be used to read the next character from the input. It returns -1 on EOF. Use `YY_GET()` to read the input when the call is within the scanner file. Outside the scanner file it is necessary to call `yyGet()` passing it the data handle of the relevant scanner.

The following excerpt illustrates a common use of `input()` to ignore C-style comments:

```
"/*"            { int ch0, ch1= ' ';
                  do {
                    ch0= ch1; ch1= input();
                  } while (ch1 != EOF && (ch0 != '*' || ch1 != '/'));
                  if (ch1 == EOF) error("EOF within comment.");
                }
```

Note that this is not the recommended way to process comments in Zlex. For the recommended method, see Section 7.6 [Start States Example], page 32.

8.3 Redefining the Input Macro `YY_INPUT`

`YY_INPUT(buf, result, maxSize)` provides input to Zlex buffers. It should fill the `char *buf` with upto `int maxSize` characters and return in `int result` either the number of characters read or `YY_EOF_IN` to indicate end-of-file. Its default definition uses the system `read` routine, but if the `--stdio` option is specified (see Section 16.3.6 [Options List], page 66), then its default definition uses the `fread` routine from the `stdio` library.

The definition of the macro `YY_EOF_IN` to be returned by `YY_INPUT` defaults to `YY_NULL` (see Section 8.4 [The Null Value], page 37), but it can be redefined by the programmer in section 1 of the Zlex source file to some other value.

This macro can be redefined in section 1 of the Zlex file to get input some other way. For example, Zlex currently supports processing of only 7-bit or 8-bit characters. However, it is possible to use Zlex to process words of size larger than that of a `char`, if those words can be mapped into characters without loss of information. This can be done as follows:

```
#define YY_INPUT(buf, result, n)   result= wordInput(buf, n)

int
wordInput(char *buf, unsigned n)
{
  Word *wordBuf= (Word *)malloc(n * sizeof(Word));
  unsigned nWords;
  int result;
  if (!wordBuf) { 'Signal memory allocation error.' }
  nWords= readWords(wordBuf, n); /* Read words from source. */
  if (nWords == 0) {
    result= YY_EOF_IN;
  }
  else {
    unsigned i;
    for (i= 0; i < nWords; i++) buf[i]= mapWordToChar(wordBuf[i]);
    result= nWords;
  }
  free(wordBuf);
  return result;
}
```

where `mapWordToChar()` maps a word into a character. Note that the scanner will maintain the current lexeme in `yytext` using characters; it will be the programmer's responsibility to map these characters back into `Words`.

8.4 The Null Value: `YY_NULL`

The macro `YY_NULL` is used for two purposes:

1. It is the value to be returned by `YY_INPUT` (see Section 8.3 [Redefining the Input Macro], page 36) on end-of-file.
2. It is the value returned by the scanner when end-of-file is encountered.

The default definition for `YY_NULL` is `0`, but the programmer can redefine this macro in a C-code section in section 1 of the Zlex file.

Zlex uses `YY_NULL` only for compatibility with undocumented behavior of **`flex`**. Its use is discouraged, as it is has two distinct purposes. Instead, the programmer should use `YY_EOF_IN` (see Section 8.3 [Redefining the Input Macro], page 36) or `YY_EOF_OUT` (see Section 13.6 [Termination Return Value], page 54) for each respective purpose.

8.5 Modifying Characters in the Input Stream

Two methods can be used by a Zlex programmer to force the generated scanner to insert characters into the input stream. The first of these is `YY_LESS` which returns characters from the current lexeme to the input stream; the other is `YY_UNPUT` which can be used to insert arbitrary characters (not necessarily from the current lexeme) into the input stream.

8.5.1 Rescanning Lexeme Text: `yyless`

`yyless(n)` returns all but the first n characters of the current lexeme back to the input stream. `yytext` and `yyleng` are suitably adjusted. The canonical form `YY_LESS(n)` can also be used. The library function `yyLess(YYDataHandle d, int n)` can also be used from files other than the generated scanner file.

Note that if it is necessary to look ahead in the input stream in order to recognize a token, it is preferable to use right context patterns (see Section 5.3.1 [Right Context], page 22). Note also that `yyless(0)` will cause the scanner to enter an infinite loop unless its state is changed in some way.

The following excerpt illustrates the use of `yyless` to generate multiple tokens from the same subsequence of the input stream. This may be useful in a situation where a single input subsequence signals both the end of a syntactic construct and the start of the next syntactic construct. If we assume that `xxx` is a Zlex macro defining the subsequence of interest then the following code should achieve our goal:

```
{xxx}         { if (flag == 0) {
                    flag= 1; yyless(0); return TOK0;
                  }
                  else {
                    flag= 0; return TOK1;
                  }
              }
```

We assume that `flag` is a suitably declared C variable, and `TOK0` and `TOK1` are the token values.

8.5.2 Unputting Characters: `unput`

`unput(c)` puts the character `c` onto the input stream to be the next character read. The Zlex programmer should ensure that 0 `<=` `c` `<` *character set size*; `unput` cannot be used to unput an `EOF` character. The contents of `yytext` are unaffected. Note that it is more efficient to use `yyless` if all that is desired is to unput a suffix of `yytext`.

The canonical form `YY_UNPUT(c)` may also be used. The Zlex library function `yyUnput(YYDataHandle d, int c)` may also be used from files other than the generated scanner file.

The following excerpt illustrates the use of `unput` to translate character sequences. If an application dictates that the input sequences '`%%(`' and '`%%)`' be translated to the sequences '`[`' and '`]`' respectively before any tokenizing occurs, and it is known that the sequences cannot occur within other tokens, then we can use the following pattern-action pairs:

```
"%%("   unput('[');
"%%)"   unput(']');
```

Note that this suffices only because it is specified that the sequences cannot occur within other tokens. If that is not the case, then the above code would not be correct and we would either need to redefine `YY_INPUT` (see Section 8.3 [Redefining the Input Macro], page 36) appropriately, or use intra-token patterns (see Chapter 11 [Using Intra-Token Patterns], page 49).

8.6 Forced Backtracking: REJECT

Backtracking can be forced to occur in a Zlex scanner to try alternate choices for a pattern by using the REJECT action. The initial choice of pattern is governed by the rules built into the generated scanner. When multiple patterns match the input to a Zlex generated scanner, the choice of pattern is governed by rules which first prefer the longest match and then the pattern which occurs earlier in the Zlex source file (see Section 2.1 [Pattern Conflicts], page 5).

REJECT transfers control to the action of the next pattern which matches the current lexeme or a prefix of the current lexeme. This action can also be referred to using the canonical name YY_REJECT.

REJECT performs a transfer of control — it is equivalent to an unconditional goto and the code immediately following the REJECT will never be executed. Also REJECT has function scope and hence it cannot be used outside the actions.

REJECT is useful when overlapping subsequences of the input are to be recognized as tokens. This is illustrated by the following scanner which outputs all the prefixes of the words in its input, where a word is a maximal sequence not containing tab, blank or newline.

```
%%
[^\t \n]+       printf("%s\n", yytext); REJECT;
.|\n
```

yytext is a NUL-terminated C-string giving the text of the current lexeme (see Section 6.1 [Lexeme Text], page 26). Given the word 'abc' the first pattern will match; its action will first output 'abc' on a separate line. When the REJECT action is executed, there is no other pattern to match 'abc'. Hence it will try to match a prefix of 'abc': 'ab' matches the first pattern. So it will again output 'ab' and execute a REJECT action. This REJECT results again in a match with the first pattern and an output of 'a'. The subsequent REJECT matches the second pattern with 'a' but no action is taken. Hence the output will be:

```
abc
ab
a
```

The REJECT action is not inordinately expensive.

8.7 Current Character Count: YY_CHAR_NUM

The macro YY_CHAR_NUM returns the number of characters read by the scanner from the current file or memory buffer upto the start of the current yytext. The position does not include any of the characters of yytext. The returned position is zero-origin: hence the character just after yytext will be at absolute position YY_CHAR_NUM + yyleng in the file or in-memory buffer.

The value returned by YY_CHAR_NUM will not be correct if the unput (see Section 8.5.2 [Unput], page 38) action is used.

Many scanning applications require tracking the current line and column number. If newlines can occur within other tokens, then the '--yylineno' option provides suitable facilities (see Section 8.8 [Current Line Number], page 40). If newlines cannot occur within other tokens, then the recommended method is illustrated by the following code fragment which shows how YY_CHAR_NUM can be used to compute the current column number within a line.

```
%{
  int lineStartPos= 0;  /* Starting YY_CHAR_NUM for a line. */
  int lineNum= 1;       /* 1 + # of '\n's seen so far. */
#define COL_NUM         (YY_CHAR_NUM - lineStartPos)
%}
%%
\n                 { lineStartPos= YY_CHAR_NUM + 1;  lineNum++;
                     'Other actions for a newline.'
                   }
```

The macro `COL_NUM` can now be used within other actions to access the column number.

8.8 Current Line Number: `yylineno`

If the '`--yylineno`' option is specified, when the scanner is generated, then the current line number (1-origin) is maintained in the variable whose default name is `yylineno` and declaration `int yylineno`. Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_LINENO` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

Unlike the implementation of `yylineno` by other scanner generators, a Zlex generated scanner does not test every character to see if it is a newline. It does these tests only when it is known that a lexeme contains or is followed by a newline character: this information is obtained using a hidden intra-token pattern (see Chapter 11 [Using Intra-Token Patterns], page 49). Hence scanning of lexemes which do not contain newlines is not slowed down except for a simple test of a flag which is performed on once per action rather than once per character.

Since a hidden intra-token pattern `+\n` (see Chapter 11 [Using Intra-Token Patterns], page 49 is used to implement the `yylineno` feature, this feature will not work if the user specifies a intra-token pattern which overlaps with the hidden pattern. It will also not work correctly if the programmer uses `unput` to put newline characters into the buffer.

This feature was added to Zlex for backward compatibility with an undocumented feature of `lex` (documented in `flex`). When newlines cannot occur within other tokens it is usually not necessary to use this feature as it is easy enough for the programmer to update a line number counter whenever a pattern containing a newline character is matched (see Section 8.7 [Character Count], page 39).

8.9 Current Column Number

If the '`--yylineno`' option is specified, then the macro `YY_COL_NUM` returns the 0-origin column number within the current line. If newlines cannot occur within other tokens, see the example in Section 8.7 [Character Count], page 39, for the recommended way to track this information.

The current column number is computed only when the Zlex programmer uses the `YY_COL_NUM` macro. The implementation uses a hidden intra-token pattern `+\n` (see Chapter 11 [Using Intra-Token Patterns], page 49 to implement the `YY_COL_NUM` macro. Hence this feature will not work if the user specifies a intra-token pattern which overlaps with the hidden pattern. It will also not work correctly if the programmer uses `unput` to put newline characters into the buffer.

8.10 Querying Beginning of Line: `YY_AT_BOL`

The macro call `YY_AT_BOL()` returns non-zero if the next token to be matched can match beginning-of-line patterns having a '^' anchor.

Note that this macro is provided only when there is at least one pattern which uses the beginning-of-line '^' anchor.

8.11 Setting Beginning of Line: `yy_set_bol`

The macro `yy_set_bol(`*v*`)` sets the beginning-of-line condition for the next pattern to true if *v* is non-zero; false if *v* is zero. When the beginning-of-line condition is set true, the next pattern can match beginning-of-line patterns having a '^' anchor; when it is set false, the next pattern cannot match beginning-of-line patterns having a '^' anchor.

The canonical macro name `YY_SET_BOL` can be used synonymously with `yy_set_bol`.

Note that these macros are provided only when there is at least one pattern which uses the beginning-of-line '^' anchor.

9 Output in a Zlex Scanner

Limited facilities are provided in a Zlex scanner for echoing the current lexeme to a `FILE` pointer with default name `yyout`

9.1 Output File Pointer: `yyout`

`yyout` is the default name of the variable with declaration `FILE *yyout` which Zlex uses to echo the current lexeme (see Section 9.2 [Echoing the Current Lexeme], page 42). Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_OUT` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

When the scanner function is first entered it initializes `yyout` to `stdout`, unless the user has already initialized it to a non-`NULL` `FILE` pointer. So if the generated scanner should echo to a file other than the standard output, the programmer need only initialize `yyout` to a suitable `FILE` pointer.

9.2 Echoing Lexeme Text: `ECHO`

The `ECHO` macro echoes the current lexeme to `yyout`. The canonical name `YY_ECHO` can also be used.

The following example removes all lines starting with `#`.

```
%%
^#.*\n          |
^#.*            /* No action: don't echo. */
.*\n            |
.*              ECHO;
```

The patterns `^#.*` and `.*` take care of processing the last line in the file when it does not end with a newline.

10 Buffer Management

For efficiency reasons, a Zlex scanner buffers its input. Hence if the Zlex programmer wishes to switch input to a new file, it is not sufficient to merely change `yyin` (see Section 8.1 [Input File Pointer], page 35), as the scanner will continue reading from its previously buffered input. It is necessary to switch to a buffer for the new file. Buffer management actions provide facilities for doing this.

Buffers need not necessarily be associated with files. It is possible to create buffers whose contents are taken from a string or some other in-memory structure. When the scanner reaches the end of an in-memory buffer, it does normal end-of-file processing.

Tokens are not allowed to span buffer boundaries.

10.1 The Buffer Type: `YY_BUFFER_STATE`

The handle used to refer to a buffer has the declaration:

```
typedef void *YYBufHandle;
```

For compatibility with `flex`, the programmer can also refer to this type using the macro `YY_BUFFER_STATE`. This opaque type can be passed to and returned from the buffer management actions.

10.2 The Current Buffer: `yy_current_buffer`

`yy_current_buffer` is the default name of a variable which contains a `YY_BUFFER_STATE` handle to the current buffer. Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_CURRENT_BUFFER` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

The user should never explicitly assign a value to this variable, but do so only implicitly by calling the appropriate buffer management routine (see Section 10.9 [Switching Buffers], page 47).

10.3 Creating a Buffer: `yy_create_buffer`

The macro call `yy_create_buffer(f, s)` creates a buffer for `FILE` pointer `f`, having space for at least `s` characters. (The macro `YY_BUF_SIZE` contains a recommended value for `s`.) The value returned is a `YY_BUFFER_STATE` (see Section 10.1 [Buffer Type], page 43).

The canonical name `YY_CREATE_BUFFER` can also be used for this macro. From files other than the Zlex source file, the library function with prototype

```
YY_BUFFER_STATE
yyCreateBuffer(YYDataHandle d, FILE *f, yy_size_t s);
```

can be used to create and initialize a buffer for the file with `FILE` pointer f, having space for at least `s` characters. It returns the handle of the newly created buffer, aborting execution on error.

10.4 Deleting a Buffer: yy_delete_buffer

The macro call `yy_delete_buffer(b)` deletes the buffer with `YY_BUFFER_STATE b`. `b` must have been previously returned by one of the buffer creation actions.

The canonical name `YY_DELETE_BUFFER` can also be used for this macro. From files other than the Zlex source file, the library function with prototype

```
void
yyDeleteBuffer(YYDataHandle d, YYBufHandle b);
```

can be used to delete buffer with handle `b` for the scanner with handle `d`.

10.5 Flushing a Buffer: yy_flush_buffer

The macro call `yy_flush_buffer(b)` flushes the buffer with `YY_BUFFER_STATE b`. When the scanner subsequently tries to read a character from the buffer, the buffer will be refreshed. There is no canonical name for the `yy_flush_buffer` macro as, for backwards compatibility with `flex`, the name `YY_FLUSH_BUFFER` does something somewhat different: specifically, it is used without any arguments to specify an action to flush the *current* buffer (equivalent to `yy_flush_buffer(YY_CURRENT_BUFFER)`).

From files other than the Zlex source file, the library function with prototype

```
void
yyFlushBuffer(YYDataHandle d, YY_BUFFER_STATE b);
```

can be used to flush buffer `b` for the scanner whose internal state is encapsulated in `d`.

10.6 Creating a In-Memory Buffer: yy_scan_buffer

The macro `yy_scan_buffer(memBuf, len)` creates and returns a `YY_BUFFER_STATE` which contains the contents of `char *memBuf` having a total of `yy_size_t len` bytes. `memBuf` is not copied: hence the programmer should ensure that `memBuf` is retained until the processing of the `YY_BUFFER_STATE` returned by `yy_scan_buffer` is completed. `memBuf` will be used when the newly created buffer is scanned: in fact, `memBuf` may even be temporarily modified during the course of scanning.

The last two bytes of `memBuf` must be *sentinel* characters (the sentinel character defaults to `'\0'` unless changed by the '`--sentinel`' option (see Section 16.3.6 [Options List], page 66)). If this is not true, then a `NULL YY_BUFFER_STATE` is returned. These two sentinel characters will not be scanned when the scanner switches to this buffer: hence the characters which will be scanned will be `memBuf[0]` ... `memBuf[len - 3]` inclusive.

The canonical name `YY_MEM_BUFFER` can also be used for this macro.

From files other than the Zlex source file, the library function with prototype

```
YY_BUFFER_STATE
yyMemBuffer(YYDataHandle d, char *memBuf, yy_size_t len);
```

can be used to create a memory buffer for the scanner whose state is encapsulated in `d`, with the other arguments being as defined for the macro. It returns the `YY_BUFFER_STATE` handle for the created buffer; `NULL` if `memBuf` does not have the two sentinel characters at its end; it aborts with an error message if it cannot create the buffer because it is out of memory.

It is important to realize that if the same memory area is used to create multiple Zlex buffers, then each Zlex buffer must be deleted before a new Zlex buffer is created from the same memory area.

The following function illustrates the use of in-memory buffers to paste tokens together as is required by the `##` operator in a C preprocessor. We assume that a `Token` is a `struct` with two fields: a small integer `tok` giving the token number, and another small integer `id` which gives the text associated with the token. We also assume the existence of the following routines:

`getIDString(id)`
: Returns the text associated with a `id`.

`getIDLen(id)`
: Returns the length of the text associated with a `id`.

`MALLOC()`
`FREE()`
: These are merely error-checking versions of the standard-library `malloc()` and `free()` respectively.

`error()`
: Prints out error messages.

`yylex()`
: The generated scanner function. We assume that it is setup to return a `Token` instead of simply a `int` (see Section 13.6 [Termination Return Value], page 54).

```
Token
tokenPaste(Token token1, Token token2)
/* Paste tokens token1 and token2 together, returning resulting token.
 * Signal an error if the pasted token is not proper.
 */
{
  const unsigned id1= token1.id;
  const unsigned id2= token2.id;
  const unsigned len1= getIDLen(id1);
  const unsigned len2= getIDLen(id2);
  const unsigned bufSize= len1 + len2 + 1 + 2;  /* 1 '\n' + 2 sentinel chars. */
  enum { AUTO_BUF_SIZE= 100 };
  char autoBuffer[AUTO_BUF_SIZE];
  char *const autoBuf= autoBuffer;
  char *const dynamicBuf= (bufSize <= AUTO_BUF_SIZE) ? NULL : MALLOC(bufSize);
  char *const *bufP= (dynamicBuf) ? &dynamicBuf : &autoBuf;
  Token tokenZ, eolToken;
  YY_BUFFER_STATE oldBuf= YY_CURRENT_BUFFER;
  YY_BUFFER_STATE pasteBuf;
  strncpy(*bufP, getIDString(id1), len1);
  strncpy(*bufP + len1, getIDString(id2), len2);
  *(*bufP + bufSize - 3)= '\n';
  *(*bufP + bufSize - 2)= *(*bufP + bufSize - 1)= '\0';
  pasteBuf= yy_scan_buffer(*bufP, bufSize);
  yy_switch_to_buffer(pasteBuf);
  tokenZ= yylex(); eolToken= yylex();
  if (eolToken.tok != '\n') {
    error("Invalid token produced by ## pasting of `%s' and `%s'.",
```

```
    getIDString(id1), getIDString(id2));
    }
    yy_delete_buffer(pasteBuf);
    yy_switch_to_buffer(oldBuf);
    if (dynamicBuf) FREE(dynamicBuf);
    return tokenZ;
  }
```

The function creates the in-memory buffer on the runtime stack if the required amount of memory is smaller than a predetermined amount; otherwise it creates the in-memory buffer on the heap. It uses `bufP` to point to the chosen buffer. It remembers the original Zlex buffer in the `YY_BUFFER_STATE` variable `oldBuf`. It then uses the standard library function `strncpy()` to copy the text of the tokens to be catenated into the chosen buffer. It terminates the copied text by a `'\n'` followed by the two required `'\0'` sentinel characters. It then creates a Zlex buffer using `yy_scan_buffer()`. It switches to the newly created buffer (see Section 10.9 [Switching Buffers], page 47) and then reads two tokens from it: it expects the first token to be the catenated token which is desired, and the second token to be a `'\n'`. It then deletes the created Zlex buffer and switches back to the original Zlex buffer `oldBuf`. Finally, if the in-memory buffer was allocated on the heap it frees it.

10.7 Creating a Buffer from In-Memory Bytes: `yy_scan_bytes`

The macro `yy_scan_bytes(bytes, len)` creates and returns a `YY_BUFFER_STATE` which contains the contents of `char *bytes` having a total of `yy_size_t len` bytes. The contents of `bytes` is copied into the newly created buffer. `bytes` itself will not be used at all when the newly created buffer is scanned and can be destroyed once the buffer has been created.

The canonical name `YY_BYTES_BUFFER` can also be used for this macro.

From files other than the Zlex source file, the library function with prototype

```
YY_BUFFER_STATE
yyBytesBuffer(YYDataHandle d, char *bytes, yy_size_t len);
```

can be used to create a memory buffer for the scanner whose state is encapsulated in `d`, with the other arguments being as defined for the macro. It returns the `YY_BUFFER_STATE` handle for the created buffer, aborting with an error message if it cannot create the buffer because it is out of memory.

10.8 Creating a Buffer from a In-Memory String: `yy_scan_string`

The macro `yy_scan_string(str)` creates and returns a `YY_BUFFER_STATE` which contains the contents of the *NUL*-terminated C-string `char *str` having a total of `yy_size_t len` bytes (not counting the terminating *NUL*). The contents of `str` is copied into the newly created buffer. `str` itself will not be used at all when the newly created buffer is scanned and can be destroyed once the buffer has been created.

The canonical name `YY_STRING_BUFFER` can also be used for this macro.

From files other than the Zlex source file, the library function with prototype

```
YY_BUFFER_STATE
yyStringBuffer(YYDataHandle d, char *str);
```

can be used to create a memory buffer for the scanner whose state is encapsulated in `d`, with the `str` argument as for the macro. It returns the `YY_BUFFER_STATE` handle for the created buffer, aborting with an error message if it cannot create the buffer because it is out of memory.

10.9 Switching Buffers: `yy_switch_to_buffer`

The macro `yy_switch_to_buffer(b)` sets up the scanner to scan from the previously created buffer identied by the `YY_BUFFER_STATE` `b`. The contents of either buffer are not affected.

The canonical name `YY_SWITCH_TO_BUFFER` can also be used for this macro.

From files other than the Zlex source file, the library function with prototype

```
void
yyStringBuffer(YYDataHandle d, YY_BUFFER_STATE b);
```

can be used to create a memory buffer for the scanner whose state is encapsulated in `d`, with the `b` argument as for the macro.

`yy_switch_to_buffer` should be the only way the Zlex programmer changes the current buffer.

10.10 Buffer Management Example

The following example illustrates the use of the buffer management routines to implement a nested file inclusion facility similar to that of C. A line starting with the `#` character followed by the word '`include`' and a *file-name* is replaced by the contents of *file-name*.

```
%{
enum { MAX_INCL_DEPTH= 3 };

static YY_BUFFER_STATE inclStk[MAX_INCL_DEPTH];
static unsigned inclSP= 0;

static void includeFile(char *fName);

%}

fileName        [0-9a-zA-Z./]+

%x INCLUDE
%%
^[\t ]*#[\t ]*include   BEGIN INCLUDE;
<INCLUDE>{fileName}     includeFile(yytext); BEGIN INITIAL;
<INCLUDE>[\t ]+         /* No action. */
<INCLUDE>\n             BEGIN INITIAL;
<<EOF>>                 { if (inclSP == 0)
                              yyterminate();
                            else {
                              yy_switch_to_buffer(inclStk[--inclSP]);
                              BEGIN INCLUDE;
                            }
                        }
%%

static void includeFile(char *fName)
{
  if (inclSP == MAX_INCL_DEPTH) {
    fprintf(stderr, "Includes nested too deeply.\n");
    return;
  }
  inclStk[inclSP++]= YY_CURRENT_BUFFER;
  yyin= fopen(fName, "r");
  if (!yyin) {
    fprintf(stderr, "Could not open %s.\n", fName); exit(1);
  }
  yy_switch_to_buffer(yy_create_buffer(yyin, YY_BUF_SIZE));
}
```

11 Using Intra-Token Patterns

Sometimes it is necessary to do pre-lexical processing on the characters scanned by the generated scanner before they are tokenized by the scanner. An example of pre-lexical processing would be mapping certain sequences of characters into others. The easiest way to do so is to intercept the input to the scanner: Zlex provides a way for the Zlex programmer to do just that by redefining the `YY_INPUT()` macro. See Section 8.3 [Redefining the Input Macro], page 36. Though this is adequate for most situations, it is not appropriate for all situations since Zlex buffers its input. The following example illustrates the problem.

ANSI-C requires that all occurrences of escaped newlines (a '\' followed by a newline) in the input be deleted. It is required that this be done before tokens be recognized: for example a '/' followed by an escaped newline followed immediately by a '*' should be recognized as the start of a comment. This can be done relatively easily by defining `YY_INPUT()` to read the C source file into a buffer which is then processed to delete escaped newlines. Unfortunately this does not allow the scanner to keep track of the source line number of the current token for proper reporting of error messages. What is needed is to perform the pre-lexical processing incrementally as each token is scanned.

Intra-token patterns allow this type of incremental pre-lexical processing. When a *intra-token pattern* is recognized *within* a token, the scanning of the token is suspended and the action associated with the intra-token pattern is executed. Then the suspended scanning of the token is resumed. Thus, the only effects of recognizing the intra-token pattern are the possible side-effects (if any) of the intra-token action.

11.1 Intra-Token Pattern Syntax

Syntactically, an intra-token pattern consist of the '+' character followed by a regular expression subject to the following restrictions:

- The regular expression is restricted to match only fixed length regular expressions (thus precluding the use of the '?' (optional), '+' (one-or-more) or '*' (closure) operators among others).
- Qualifying an intra-token pattern by start states is not allowed.

For example, `+\\\n` is an intra-token pattern which matches an escaped newline.

11.2 Actions for Intra-Token Patterns

`yytext` and `yyleng` (see Chapter 6 [The Current Lexeme], page 26) are available as usual within intra-token pattern actions. The special `YY_BACKUP` action (see Section 11.3 [Backing Up], page 50) is also available within intra-token pattern actions. Since an intra-token pattern represents a interrupted scan of a token, the action for an intra-token pattern are subject to the following rather severe restrictions:

- The effect of any other of Zlex's action facilities except those explicitly mentioned above is undefined within an intra-token pattern action.
- An intra-token pattern is not allowed to share its action with that of any other pattern. Hence the action associated with an intra-token pattern or the pattern immediately before an intra-token pattern cannot be '|'.
- An intra-token pattern action should not terminate by performing a control transfer like a `return`.

11.3 Backing Up Within a Intra-Token Pattern: `YY_BACKUP`

Intra-token patterns are intended for doing pre-lexical processing which needs to be done incrementally during scanning. It is expected that this processing will usually involve translating the token matching the intra-token pattern to another token and then continue scanning. The `YY_BACKUP` action is tailored for such processing.

The macro `YY_BACKUP(`*len*`, `*string*`)` specifies an action to be used only within actions for intra-pattern tokens. The length *len* must be no greater than the length of the intra-token pattern and *string* should be a *NUL*-terminated C-string. The effect of `YY_BACKUP(`*len*`, `*string*`)` is to backup the scanner automaton over the last *len* characters of the intra-token pattern, replacing those *len* characters by *string*. `YY_BACKUP` may perform a control transfer: hence it should not be followed by any code to which control is expected to fall through after `YY_BACKUP`.

For example, to delete escaped newlines in a C-scanner, we could use the following pattern-action pair.

```
+\\\n           YY_BACKUP(2, "");
```

12 End-of-File and Termination

The generated scanner normally terminates when an EOF is received on the input stream. This default action can be changed in two ways:

1. An EOF on the input stream need not terminate the scanner provided the Zlex programmer sets up the input to come from another source.
2. A scanner can terminate before an EOF is received from the input stream.

12.1 Wrapping Up: yywrap

A function whose default prototype is int yywrap(void) is called by the scanner when it detects end-of-file on its current input stream yyin. If the function returns non-zero then the scanner proceeds to wrap-up its processing; it processes its <<EOF>> actions if any (see Section 5.4 [End of File Patterns], page 24) and if these actions do not change its flow of control, it returns YY_EOF_OUT (which defaults to 0) indicating an end-of-file token.

If the call to yywrap returns 0, then the scanner assumes that the function has set up yyin to continue scanning. It does not execute the actions associated with any <<EOF>> patterns but merely continues scanning.

Like Zlex variable names, the name of this function can be changed to an arbitrary name by defining the macro YY_WRAP to the new name. Alternatively, the prefix used for the name can be changed by using the '--prefix' option (see Section 4.1.3 [Function Names], page 12).

The Zlex library provides a yywrap() function which simply returns 1.

12.2 <<EOF>> Pattern Actions

If the yywrap function (see Section 12.1 [Wrapping Up], page 51) returns non-zero, then the scanner executes the actions associated with its <<EOF>> actions if any. These actions provide another opportunity for the programmer to reset the scanner so as to continue scanning. The following points need to be noted:

- yytext and yyleng are not defined for <<EOF>> patterns. Hence the actions for <<EOF>> patterns should not refer to these variables.
- If yyin is pointed to a new FILE pointer within an <<EOF>> action, then scanning will continue. For compatibility with old versions of flex, the YY_NEW_FILE and YY_RESTART actions (see Section 12.4 [Restarting a Scanner], page 52) may be used after resetting yyin but is not necessary.
- The scanner can switch to a new buffer by using a yy_switch_to_buffer action (see Section 10.9 [Switching Buffers], page 47).
- The scanner can execute a return statement.
- It can execute a yyterminate() action (see Section 12.3 [Terminating a Scanner], page 52).
- Control can simply fall off the end of the <<EOF>> action. In that case, the scanner will return to its caller with a YY_EOF_OUT (which defaults to YY_NULL).

12.3 Terminating a Scanner: `yyterminate`

`yyterminate()` terminates the scanner and returns a `YY_EOF_OUT` (which defaults to 0) to the caller. Subsequent calls to the scanner will continue to return with `YY_EOF_OUT`, until a `yyrestart`, `YY_NEW_FILE` or `yy_switch_to_buffer` action is executed.

The canonical form `YY_TERMINATE()` can also be used instead.

12.4 Restarting a Scanner: `yyrestart`

The action `yyrestart(fileP)` restarts scanning, taking input from the file with the `stdio` `FILE` pointer `fileP`. The current contents of the Zlex buffer are discarded. The canonical name `YY_RESTART` can also be used instead.

If `yyin` has been pointed to a new file, then the action `YY_NEW_FILE` (without any arguments) tells the scanner that a new file has been setup in `yyin`. `YY_NEW_FILE` is equivalent to `yyrestart(yyin)`.

13 The Main Scanner Function

The top-level scanner function in a Zlex scanner has default name `yylex`. The user can customize the name and declaration of the function, as well as define macros to cause certain actions to be taken.

13.1 The Main Scanner Function: `yylex`

The name of the main scanning function defaults to `yylex`, but like other names, the name of this function can be changed to an arbitrary name by defining the macro `YY_LEX` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.3 [Function Names], page 12). Hence to change the name of the main scanning function to `scan`, the programmer merely need have the line

```
#define YY_LEX scan
```

in the C-code section of section 1 of the Zlex file.

13.2 Declaring the Scanner Function: `YY_DECL`

The macro `YY_DECL` gives the default declaration of the scanner function. Its definition is equivalent to:

```
#ifndef YY_DECL
#define YY_DECL int YY_LEX(void)
#endif
```

The programmer can change this declaration by suitably '`#define`'ing `YY_DECL` in the C-code section of section 1 of the Zlex file. See Section 13.6 [Termination Return Value], page 54 for how to use `YY_DECL` to declare a scanner function which returns a `struct` rather than a `int`.

13.3 Initialization: `YY_USER_INIT`

The macro `YY_USER_INIT` can be defined by the programmer to be code which will be executed when the scanner first starts up, before the first scan. Its definition defaults to empty code. It is useful for initializing variables used by the programmer. Before the first call to the main scanner function, the only Zlex actions guaranteed to work are the buffer creation and switching routines (see Chapter 10 [Buffer Management], page 43).

The scanner's buffer is statically initialized to a special initialization state. If there is no buffer switching action before the first call to the scanner function, then the scanner buffer will still be in this special initialization state at the first call. The first call to the scanner checks whether the buffer is in this special initialization state: if it is, it creates a new buffer corresponding to the current `yyin`; if it not, then it assumes that the programmer has created and switched to a valid buffer and uses that buffer without modification. Note that the effect of using such a special initialization buffer in subsequent scanner calls is undefined.

13.4 Specifying a Pre-Action for every Pattern: `YY_USER_ACTION`

The macro `YY_USER_ACTION` can be defined by the programmer to be code which will always be executed before any matched rule action. Its definition defaults to empty code.

13.5 Specifying the Separator Between Actions: YY_BREAK

The user actions specified in the Zlex file are copied into the main scanner function as part of a switch statement. The macro `YY_BREAK` is used to separate the actions within the `switch` statement. Its definition defaults to `break`.

Redefining this macro appears to be of limited utility. This feature is included for compatibility with `flex`. The rationale for including this feature in `flex` was to prevent unreachable statement warnings when a user action naturally terminates with a control transfer like a `return`. With this feature, the user can define `YY_BREAK` to be empty while ensuring that every action terminates with a transfer of control (inserting explicit breaks, if necessary), thus avoiding the warnings.

13.6 Return Value on Termination: YY_EOF_OUT

The macro YY_EOF_OUT specifies the value to be returned by the scanner on end-of-file after `yywrap` returns 1 and the `<<EOF>>` actions (if any) do not reset the input `yyin`. Its definition defaults to `YY_NULL` (see Section 8.4 [The Null Value], page 37) but it can be redefined by the programmer in section 1 of the Zlex file.

For example, by using `YY_DECL` (see Section 13.2 [Declaring the Scanner Function], page 53) macro, it is possible for the Zlex programmer to make the scanner return a `struct` rather than a `int`. If this is done, then the value returned on end-of-file must also be a suitable `struct`: this can be achieved by defining `YY_EOF_OUT` to a call of a suitable function returning the suitable `struct`. Appropriate definitions and declarations are shown below:

```
%{

/* Define the type returned by the scanner. */
typedef struct {
  ...
} Token;

/* Declare a function returning a special EOF Token struct. */
Token eofToken(void);

/* Scanner declaration. */
#define YY_DECL Token YY_LEX(void)

/* EOF return value definition. */
#define YY_EOF_OUT eofToken()

%}
```

13.7 Pattern Numbers: yy_act and YY_NUM_RULES

Within the actions, the macro `yy_act` refers to the pattern number which is currently being matched where the patterns from the Zlex source file are numbered starting at 1. The macro `YY_NUM_RULES` refers to the total number of patterns for which actions exist in the generated scanner; this will usually be greater than the number of patterns explicitly specified by the programmer in the Zlex source file, since Zlex uses several pseudo-actions for its own purposes.

14 Debugging and Errors

Limited facilities are provided for debugging Zlex programs. A C macro and a variable control whether debugging messages are output as a pattern is recognized. In addition, it is also possible to obtain a more detailed trace detailing the action of the scanner as each individual character is scanned.

14.1 Debugging Control

Several equivalent macros and a single variable control whether debugging messages are output to `stderr` as patterns are matched.

If the '`--debug`' option is specified when the scanner file is generated (see Section 16.3.6 [Options List], page 66), or if the macro `YYDEBUG` is defined when the generated scanner file is compiled, and if at runtime, the variable with default name `yy_Zlex_debug` has a non-zero value, then messages are printed on `stderr` as patterns are matched. Where applicable, the printed messages include the source file name and line number of the matched pattern, as well as the contents of `yytext`. The format is similar to that of compiler error messages of popular compilers like `gcc`; this makes it possible to use tools like `emacs compile-mode` to point to the appropriate pattern in the source file. See section "Compiling within emacs" in *The GNU Emacs Manual*.

The macro `YY_ZL_DEBUG` is equivalent to `YYDEBUG`. This alternate name is useful when a project uses both Zlex as well as a parser generated by a member of the `yacc`-family of parser generators. The reason is that `YYDEBUG` is also used for similar purposes by such parser-generators; if the option `-DYYDEBUG` is passed as a C-compiler option to a '`Makefile`' for the project, both the generated Zlex scanner as well as the parser will run in debug-mode, resulting in rather confusing output.

The `extern` variable with default name `yy_zlex_debug` allows debugging messages to be turned on and off dynamically: messages are printed only when the variable has a non-zero value. When debugging is turned on as described above, the variable is declared in the generated scanner and initialized to `1`: hence message printing is initially enabled.

Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_ZLEX_DEBUG` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

14.2 A Debugging Example

Assume that the following scanner is defined in file '`debug.l`'.

```
/* Scanner which illustrates debugging messages. */

%option debug

%%

[[:digit:]]+                |
[[:alpha:]]+                REJECT;
.|\n
```

If the generated scanner is compiled and run on the input consisting of the single line

```
ab12
```

the following output is produced on `stderr`.

```
debug.l:8: yytext= 'ab'.
debug.l:8: yytext= 'a'.
debug.l:9: yytext= 'a'.
debug.l:8: yytext= 'b'.
debug.l:9: yytext= 'b'.
debug.l:7: yytext= '12'.
debug.l:7: yytext= '1'.
debug.l:9: yytext= '1'.
debug.l:7: yytext= '2'.
debug.l:9: yytext= '2'.
debug.l:9: yytext= '
'.
--EOF.
```

14.3 Scanner Tracing

If the scanner is compiled with the C macro `YYTRACE` `#define`d, then when the scanner is run it provides a detailed trace showing the action it takes at every character it scans. The trace shows the transitions of the underlying finite automaton. To decipher this trace, it is necessary to have compiled the scanner using the '`--trace`' option (see Section 16.3.6 [Options List], page 66). This is useful mainly for maintaining Zlex.

14.4 Runtime Errors: `yyerr`

It is possible for Zlex to encounter runtime errors under several conditions:

- Dynamic memory allocation fails.
- An error occurs while reading an input file.
- The programmer attempts to access an empty start state stack.
- The programmer attempts to switch to a `NULL` buffer.
- The scanner encounters a character which is not matched by any pattern and the `--suppress-default` option (see Section 16.3.6 [Options List], page 66) has been specified.

When a runtime error is encountered, the generated scanner writes a message on the `FILE` pointer `yyerr` and terminates execution of the program.

Like all other variable names, the name of this variable can be changed to an arbitrary name by defining the macro `YY_ERR` to the new name. Alternatively, the prefix used for the name can be changed by using the '`--prefix`' option (see Section 4.1.2 [Variable Names], page 11).

When the scanner function is first entered it initializes `yyerr` to `stderr`, unless the programmer has already initialized it to a non-`NULL` `FILE` pointer. So if the generated scanner should output error messages to a file other than the standard error, the programmer need only initialize `yyerr` to a suitable `FILE` pointer.

All error messages are preceeded by the string which is the value of the macro `YY_PROGRAM_NAME` which defaults to `"Zlex scanner"`. This macro can be redefined by the programmer in section 1 of the Zlex source file.

15 Multiple Scanners

Sometimes it is necessary to include multiple scanners in a program. For example, an application may need one scanner to scan a data file and another scanner to scan interactive user input.

Most of the global objects used by a generated scanner are declared `static`. Hence their names are local to the generated C file. Since different scanners are generated in different C files, the semantics of C preclude the possibility of a clash between the `static` names used in different scanners. However there will be a link-time clash between the `extern` names used for global objects declared in different scanners. To circumvent this problem, Zlex allows the programmer to choose the names for the `extern` objects using one of the following schemes:

1. Using the '`--prefix`' option (see Section 16.3.6 [Options List], page 66). This is the simpler alternative.
2. Defining macros giving the name of each `extern` object.

15.1 Prefix Option

By default the name of each `extern` scanner object starts with the prefix '`yy`'. This prefix can be changed by using the '`--prefix`' option (see Section 16.3.6 [Options List], page 66). The names which are affected are:

`yy_current_buffer`
: See Section 10.2 [Current Buffer], page 43.

`yy_Zlex_debug`
: See Section 14.1 [Debugging Control], page 55.

`yydataP`
: See Section 4.3 [Scanner State], page 13.

`yyerr`
: See Section 14.4 [Errors], page 56.

`yyin`
: See Section 8.1 [Input File Pointer], page 35.

`yyleng`
: See Section 6.2 [Lexeme Length], page 26.

`yylex`
: See Section 13.1 [Main Scanner Function], page 53.

`yylineno`
: See Section 8.8 [Current Line Number], page 40.

`yyout`
: See Section 9.1 [Output File Pointer], page 42.

`yytext`
: See Section 6.1 [Lexeme Text], page 26.

`yywrap`
: See Section 12.1 [Wrapping Up], page 51.

15.2 Redefining Name Macros

Another method to ensure that the names of `extern` objects do not clash between multiple scanners is to redefine the macros which specify the names of these objects. This should be done in a C-code section in section 1 of the Zlex file. The macros and the objects they name are:

`YY_CURRENT_BUFFER`
: Names `yy_current_buffer` (see Section 10.2 [Current Buffer], page 43).

`YY_DATA_P`
: Names `yydataP` (see Section 4.3 [Scanner State], page 13).

`YY_ERR`
: Names `yyerr` (see Section 14.4 [Errors], page 56).

`YY_IN`
: Names `yyin` (see Section 8.1 [Input File Pointer], page 35).

`YY_LENG`
: Names `yyleng` (see Section 6.2 [Lexeme Length], page 26).

`YY_LEX`
: Names `yylex` (see Section 13.1 [Main Scanner Function], page 53).

`YY_LINENO`
: Names `yylineno` (see Section 8.8 [Current Line Number], page 40).

`YY_OUT`
: Names `yyout` (see Section 9.1 [Output File Pointer], page 42).

`YY_TEXT`
: Names `yytext` (see Section 6.1 [Lexeme Text], page 26).

`YY_WRAP`
: Names `yywrap` (see Section 12.1 [Wrapping Up], page 51).

`YY_ZLEX_DEBUG`
: Names `yy_Zlex_debug` (see Section 14.1 [Debugging Control], page 55).

The generated scanner sets these macros to their default values (the default values factor in the '`--prefix`' option if it has been specified) only if they have not already been defined in section 1 of the Zlex file. Hence the Zlex programmer can easily make the scanner use different external names, by simply defining these macros to suitable names in section 1 of the Zlex file.

15.3 Renaming Example

For example, to build a scanner which uses the names `yylex1`, `yytext1`, `yyleng1`, `yyin1`, `yyout1` for some of the `extern` objects, section 1 of the Zlex file would contain the following `#define`'s:

```
%{

#define YY_LEX  yylex1
#define YY_TEXT yytext1
#define YY_LENG yyleng1
#define YY_IN   yyin1
#define YY_OUT  yyout1

%}
```

The rest of the program would access this generated scanner as follows: The main scanning function would be called as `yylex1()`. The lexeme text and length of the current token would be found in `yytext1` and `yyleng1`. The `FILE` pointers `yyin1` and `yyout1` would be used for the input and output files of the generated scanner. The function `yywrap()` will be called on end-of-file; this will be the `yywrap()` provided by the Zlex library, unless the Zlex programmer defines one elsewhere in the program.

16 Invoking Zlex

The command line needed to invoke Zlex has the format:

`zlex [`*Options List*`]` *lex-file* `[`*lex-file*`...]`

- If no *lex-file*s are specified on the command line and the '`--lex-compat`' option is not specified, then a help message is printed on the standard output.
- If no *lex-file*s are specified on the command line and the '`--lex-compat`' option is specified, then Zlex reads the source file from its standard input.
- If multiple *lex-files*s are specified, then their concatenation is treated as a single logical file.
- A *lex-file* specified by the single character '`-`' stands for the standard input.

16.1 Option Conventions

A word which constitutes a command-line argument has two possible types: it is a *option word* if it begin with a '`-`' or '`--`' (with certain exceptions noted below), or if it follows an option word which requires an argument. Otherwise it is a *non-option word*. An option word specifies the value of a Zlex option; a non-option word specifies a file name.

- Options with short single character names must begin with a single '`-`'.
- Options with long multiple-character names must begin with '`--`'. The name can consists of any alphanumeric characters along with '`-`' and '`_`' characters.
- When a option is specified using a long name, it is sufficient to specify an unambiguous prefix of its name.
- If an option has a value which must be one of several prespecified values, then it is sufficient to specify an unambiguous prefix of the value, in a manner similar to long option names.
- It is possible to specify an option value in the same word as the option name. For short option names, there should not be any intervening characters between the short name and the value. For long option names, the long name should be separated from the value using a single '`=`' character.
- If an option has an *optional* value, then the value must be provided in the same word as the option name, as outlined above.
- If an option has a *required* value, then the value may be provided in the same word as the option name as outlined above, or it may be provided in the next word. In the latter case, the entire next word is taken to be the value (even if it looks like an option starting with '`-`' or '`--`').
- Short names for multiple options which are not allowed to have any values may be combined into a single word. For example, if the options '`-l`' and '`-7`' are not allowed to have any values, then instead of specifying them using two words as '`-l -7`', they can be specified using a single word '`-l7`'.
- If an option is given two incompatible values, then the option which is specified later dominates.
- If the option consisting simply of the two characters '`--`' is specified, then all the remaining words on the command line will not be treated as options irrespective of whether they start with '`-`' or '`--`'. This makes it possible to specify file names starting with a '`-`'.
- Option words and non-option words may be arbitrarily interspersed.

- Command-line options always override the options specified elsewhere (see Section 16.2 [Option Sources], page 60).

16.2 Option Sources

Besides the command-line, Zlex can read its options from several different sources. In order of increasing priority these sources are the following:

- The file '`zlex.opt`'. The file should contain only option names and values separated by whitespace (newline counts as whitespace). In addition it may contain comments enclosed within '`/*`' and '`*/`'. The file is searched for using Zlex's search list (see Section 16.4 [Data Search List], page 68).

  The use of this file for setting defaults, makes it possible for a site to setup default options for Zlex different from its builtin defaults. It also makes it easy to drop Zlex into a GUI toolset where options are set using a graphical user interface.
- The environment variable `ZLEX_OPTIONS`. If this variable is set, then its value should contain only options and option values separated by whitespace as on the command-line. The procedure for setting environment variables depends on the system you are using: under the UNIX shell `csh` the `setenv` command can be used, under the MS-DOS command-interpreter the `set` command can be used; under the UNIX shell `sh` or `ksh` the `export` command can be used.
- It is also possible to specify options directly within the Zlex source file using the `%option`, `%array` or `%pointer` directives (see Section 3.3 [Declarations Section], page 9).

Options specified by the environment variable `ZLEX_OPTIONS` overrides the options specified in the '`zlex.opt`' file. Options specified in the Zlex source file override options specified in the '`zlex.opt`' file or `ZLEX_OPTIONS` environment variable. Finally, command-line options always override options specified by all other sources.

16.3 Options

Zlex provides a largely orthogonal set of options. We can roughly classify the options according to which aspect of Zlex's functionality they affect.

16.3.1 Runtime Input Options

These options control how the generated Zlex scanner treats its input.

`--16-bit`
: Generate a 16-bit scanner which supports upto 65536 characters. It is the programmers responsibility to ensure that all input characters have a character code between 0 and 65535 inclusive. This option should be used if the generated scanner is to support Unicode characters.

`--7-bit`
`-7`
: Generate a 7-bit scanner which supports upto 128 characters. It is the programmers responsibility to ensure that all input characters have a character code between 0 and 127 inclusive.

`--8-bit`
`-8`
: Generate a 8-bit scanner which supports upto 256 characters. It is the programmers responsibility to ensure that all input characters have a character code between 0 and 255 inclusive.

`--ignore-case[=1|0]`
`-i[1|0]`
`--caseless[=1|0]`
`--case-insensitive[=1|0]`
: Controls the case sensitivity of the generated scanner. If the value of the option is specified as 1, then the generated scanner will not distinguish between upper-case and lower-case characters. Note however, that the generated scanner will retain the case of the characters constituting the current lexeme in `yytext`. If this option is not specified, then the generated scanner will be case-sensitive.

`--sentinel=CHAR-CODE`
`-S CHAR-CODE`
: Use the character with decimal code `CHAR-CODE` as the sentinel character. Scanning the sentinel character is likely to be slower than scanning non-sentinel characters; this option allows the programmer to change the sentinel character to a character which may not occur frequently in the scanner input. If this option is not specified, then the sentinel character defaults to the character whose code is 0.

`--stdio[=1|0]`
: By default, the generated scanner uses the underlying systems `read()` function to read its input. If this option is specified, it uses a the `stdio` `fread()` function instead. It may be necessary to specify this option if your system does not take kindly to mixing `stdio` `FILE` descriptors with `read()`. The generated scanner may be somewhat slower, and its interactive operation may suffer, depending on the implementation of the `fread()` function provided by the `stdio` library. This option will not have any effect if the programmer has redefined the `YY_INPUT` macro (see Section 8.3 [Redefining the Input Macro], page 36).

16.3.2 Runtime Algorithm Options

The options described in this section affect aspects of the algorithm used by the generated scanner. Options which affect scanner tables are described in Section 16.3.4 [Table Scanner Options], page 64. Options which affect generated scanners which minimize their use of tables are described in Section 16.3.3 [Code Scanner Options], page 62.

`--array[=1|0]`
: Implement `yytext` as an array instead of the default pointer. This will usually lead to a slower scanner.

`--backup-optimize[=1|0]`
: If value of this option is specified as 1, then the generated scanner optimized for backing up. This may slow down the scanning of normal patterns somewhat, but is likely to speedup the scanning of patterns which require the scanner to backup. This option should be used only if the scanner's patterns require a lot of backing up, or if the scanner

makes heavy use of `REJECT`. By default, backup-optimization is off. See Chapter 17 [Efficiency], page 70.

`--default-action echo|error|fatal|ignore`
: This option controls the action of the generated scanner if it encounters a input character which does not match any action. The effects of the different option values is described below:

  `echo` The unmatched character is echoed to `yyout`. This is the default.

  `error` An error message is output to `yyerr` and scanning continues.

  `fatal` An error message is output to `yyerr` and the program is terminated.

  `ignore` The unmatched character is simply ignored and scanning continues.

`--equiv-classes[=1|0]`
`--ecs[=1|0]`
`-E[1|0]`
: It is often the case that the behavior of the generated scanner is identical for certain classes of input characters. When this option is specified, Zlex partitions the input characters into equivalence classes such that the behavior of the generated scanner is guaranteed to be identical for all the characters in an equivalence class. This option usually leads to large space savings while paying only a small time cost and is on by default.

`--prefix PREFIX`
`-P PREFIX`
: Use prefix as the prefix of certain global names in the generated scanner (the default prefix is 'yy'). See Section 15.1 [Prefix Option], page 57.

`--reject[=1|0]`
: Support REJECT actions in the generated scanner. If `--reject=0` is specified, then any `REJECT` action in the source file will result in a compile-time errors when the generated scanner is compiled. The default is to support `REJECT` actions.

`--yylineno[=1|0]`
: Maintain the current line number within the input in the global variable `yylineno` (see Section 8.8 [Current Line Number], page 40). The default action is to not support `yylineno`. Using `yylineno` may lead to a somewhat larger scanner but will not slow down the matching of patterns which do not contain newlines. It is the programmer's responsibility to suitably update `yylineno` after scanner actions like `yyless` or `unput` (see Section 8.5 [Modifying Input], page 37). yylineno is maintained on a per buffer basis and is automatically saved and restored on a buffer switch.

16.3.3 Code Scanner Options

Zlex supports the generation of code-scanners which do not use an explicit scanner state or scan tables. Instead these scanners use the program counter to implicitly maintain the scanner state. The current implementation is disappointing: the generated scanners are fairly large but are not appreciably faster.

Some limitatitions are imposed by the current implementation on code scanners:

- Code scanners cannot be built for any source file which contains a right context pattern (see Section 5.3.1 [Right Context], page 22) in which *both* the regular expression and the context match strings of indeterminate length.
- Code scanners cannot be built with backing up optimized using `--backup-optimize=1`.

When Zlex builds a code scanner it analyzes each state before deciding what kind of code to build for a state. The kinds of code built for a state are:

Linear Code
: The code for the state is organized as a linear sequence of tests on the current input character to find the next state.

Binary Code
: The code for the state is organized as a binary search on the current input character to find the next state.

Switch Code
: The code for the state is organized as a switch on the current input character to find the next state.

The options for generating code scanners allow the programmer to control the parameters of the algorithm Zlex uses for choosing between the above code alternatives:

`--code-scan[=1|0]`
: This options generates a scanner with the DFA encoded in code instead of the default encoding within scanner tables.

`--bin-code-param N`
: A binary search is used for a state only when the number of input ranges (a *range* is a sequence of consecutive input characters which have the same successor state) is less than or equal to `N`. Otherwise switch code is used. The default value of `N` is 16.

`--lin-code-param N`
: A linear search is used for a state only when the number of successor states is less than or equal to `N`. Otherwise binary search or switch code is used. The default value of `N` is 4.

`--transition-compress[=1|0]`
: This option compresses transitions from non-switch states in the scanner. If the code for a successor state of a non-switch state has not already been output, then the code for the successor is directly inserted within the code for the predecessor state, rather than a transition to separate code for the successor.

If both `--bin-code-param` and `--lin-code-param` are specified as 0, then only switch code is produced for all state transitions.

In addition, if the compiler supports labels as first class objects and provided a method to access the addresses of code labels, switches are directly coded as a branch through a jump table. For example, `gcc` allows taking the address of a *label* using `&&`*label*. See section "Labels as Values" in *Using and Porting GNU cc*. Specifically, the following macros can be used to support this:

`YY_LABEL_VARS`
: If this macro is defined, then the compiler allows access to the address of a label.

`YY_LABEL_TYPEDEF(`*type*`)`

This macro should expand to a `typedef` used to define the *type* used to represent the address of a code label. For `gcc`, the default definition is:

```
#ifndef YY_LABEL_TYPEDEF /* How compiler declares vars containing labels. */
#define YY_LABEL_TYPEDEF(type) typedef void *type
#endif
```

`YY_LABEL_ADDR(`*label*`)`

This macro should expand to the address of *label*. For `gcc`, the default definition is:

```
#ifndef YY_LABEL_ADDR /* How compiler takes the address of a label. */
#define YY_LABEL_ADDR(label) &&label
#endif
```

16.3.4 Table Scanner Options

Zlex supports 3 options which control the details of the table compression algorithm used and 3 options which control the type of table entries. This leads to a total of 9 different kinds of table accesses.

The options are the following:

`--compress` *compression˙algorithm*

This option controls the table compression algorithm used. The permissible values for *compression_algorithm* are shown below ordered from the least compressed and fastest, to the most compressed and slowest:

`none` No table compression is used.

`comb` A comb-compression algorithm is used to compress the tables.

`iterative`
A iterative comb-compression algorithm is used to compress the tables.

`--table` *entry˙type*

This option controls the type of table entries. The permissible values for *entry_type* are shown below ordered from the most memory intensive but fastest, to the least memory intensive and slowest:

`address` Each Table entry contain pre-computed addresses. This avoids computing the addresses at runtime.

`difference`
Each table entry contain the difference between the first entry for the current state in the table and the first entry of the successor state in the table.

`state` Each table entry directly contains the number of the successor state.

`--col-waste-percent` *percent*

When `--compress=no` and `--table=state`, use a 2-dimensional table with the # of columns a power of 2 if the percentage of wasted columns is <= *percent*. The default value is 50%.

`--align[=1|0]`
`-a[1|0]` By default, Zlex generated scanners do not explicitly specify the types for various integral quantities. Instead, Zlex sets up the generated scanner so that when it is compiled, the compiler automatically chooses the smallest integral type which can accomodate all possible values in the range of the quantity. This approach has the advantage that generated scanners are conservative in their use of memory, as well as allowing a Zlex scanner to be generated on one machine, while being compiled and executed on another machine. Unfortunately, such an approach can cause performance problems on some architectures which may perform suboptimally on smaller integral types. This option allows the programmer to force the types for all integral quantities to be `int`.

16.3.5 Miscellaneous Options

This section describes miscellaneous Zlex options including options to print out generated scanner statistics and options to turn on runtime tracing in the generated scanner.

`--debug[=1|0]`
`-d[1|0]` Turn on debugging in the generated scanner. A similar effect can be obtained by defining `YYDEBUG` or `YY_ZL_DEBUG` when compiling the scanner (see Chapter 14 [Debugging], page 55).

`--help`
`-h` Print out a help message giving a brief description of all the options, as well as a description of Zlex's current environment.

`--lex-compat`
`-l` Maximize lex compatibility (equivalent to `--array --yylineno --reject`). Also if no *lex-files* are specified on the command-line, then read the zlex source file from the standard input.

`--line-dir`
Output `#line` directives to the generated scanner (default).

`--output` *filename*
`-o` *filename*
The generated scaner is written to *filename*. By default, the name of the generated scanner is '`lex.yy.c`'.

`--to-stdout[=1|0]`
`-t[1|0]` Write the generated scanner to stdout, as opposed to the default action of writing it to a file '`lex.yy.c`'.

`--trace[=`*trace_file*`]`
Run Zlex in trace mode, generating extensive output. This option is used mainly for maintaining Zlex. To use this option it is necessary that the C preprocessor symbol `DO_TRACE` is defined when Zlex is built. If *trace_file* is omitted, then the trace is produced in a file whose name is the basename of the first source file with its extension '`.l`' (if any) removed and extension '`.trc`' added.

`--verbose[=1|0]`
`-v[1|0]` Generate somewhat verbose statistics describing DFA on stdout. No statistics are generated by default.

`--version`
`-V` Print the currrent version number of Zlex and exit.

`--whitespace`
`-w` Allow whitespace within lex patterns (see Section 5.6 [Whitespace Within Patterns], page 24).

16.3.6 Alphabetical Listing of all Options

This section contains a short description of all options, sorted by long option name. Each option contains a reference to the section where it is discussed in more detail.

`--16-bit` Generate a unicode-ready scanner which handles 16-bit characters. See Section 16.3.1 [Runtime Input Options], page 60.

`--7-bit`
`-7` Generate a scanner which handles only 7-bit characters. See Section 16.3.1 [Runtime Input Options], page 60.

`--8-bit`
`-8` Generate a scanner which handles 8-bit characters (default). See Section 16.3.1 [Runtime Input Options], page 60.

`--align[=1|0]`
`-a[1|0]` Generate a scanner which does not use short integer tables (default: `0`). See Section 16.3.4 [Table Scanner Options], page 64.

`--array[=1|0]`
Implement yytext as an array (default: `0`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`--backup-optimize[=1|0]`
Optimize the scanner for patterns which require it to backup (default: `0`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`-c` Deprecated option included for POSIX compliance.

`--bin-code-param` *n*
Binary search in code-scanner if # input ranges `<=` *n* (default: `16`). See Section 16.3.3 [Code Scanner Options], page 62.

`--code-scan[=1|0]`
Generated scanner encodes DFA in code instead of tables (default: `0`). See Section 16.3.3 [Code Scanner Options], page 62.

`--compress` *type*
`-C` *type* Use table compression of *type* `no`, `comb` or `iterative` (default: `comb`). See Section 16.3.4 [Table Scanner Options], page 64.

`--col-waste-percent` *percent*
: When –compress=none & –table=state, use a 2-dimensional table with the # of columns a power of 2 if the percentage of wasted columns is `<=` *percent* (`0 <=` *percent* and *percent* `<=` 100) (default: `50`). See Section 16.3.4 [Table Scanner Options], page 64.

`--debug[=1|0]`
`-d[1|0]` Turn on debugging in generated scanner (default: `0`). See Section 16.3.5 [Miscellaneous Options], page 65.

`--default-action` *act*
`-s` *act* Set default action for unmatched character to *act* `echo`, `error`, `fatal` or `ignore` (default: `echo`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`--ecs[=1|0]`
`-E[1|0]`
`--equiv-classes[=1|0]`
: Partition input characters into equivalence classes (default: `1`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`--help`
`-h` Print summary of options and exit. See Section 16.3.5 [Miscellaneous Options], page 65.

`--ignore-case[=1|0]`
`-i[1|0]`
`--caseless[=1|0]`
`--case-insensitive[=1|0]`
: Generate a case insensitive scanner (default: `0`). See Section 16.3.1 [Runtime Input Options], page 60.

`--lex-compat`
`-l` Lex compatibility (equivalent to –array –yylineno –reject). See Section 16.3.5 [Miscellaneous Options], page 65.

`--lin-code-param` *n*
: Linear search in code-scanner if # successors `<=` *n* (default: `4`). See Section 16.3.3 [Code Scanner Options], page 62.

`--line-dir[=1|0]`
: Output #line directives to generated scanner (default: `1`). See Section 16.3.5 [Miscellaneous Options], page 65.

`-n` Deprecated option included for POSIX compliance.

`--output` *filename*
`-o` *filename*
: Generate scanner in file *filename* (default: `lex.yy.c`). See Section 16.3.5 [Miscellaneous Options], page 65.

`--prefix` *prefix*
`-P` *prefix* Use *prefix* as prefix of certain global names in generated scanner (default: `yy`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`--sentinel` *char-code*
`-S` *char-code*
: Use character with decimal code *char-code* as sentinel (default: `0`). See Section 16.3.1 [Runtime Input Options], page 60.

`--reject[=1|0]`
: Allow REJECT actions in generated scanner (default: `1`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

`--stdio[=1|0]`
: Use standard I/O for all scanner input (default: `0`). See Section 16.3.1 [Runtime Input Options], page 60.

`--table` *type*
`-T` *type*
: Use table entries of *type* `address`, `difference` or `state` (default: `state`). See Section 16.3.4 [Table Scanner Options], page 64.

`--to-stdout[=1|0]`
`-t[1|0]`
: Write generated scanner to stdout (default: `0`). See Section 16.3.5 [Miscellaneous Options], page 65.

`--trace[=`*trace˙file*`]`
`-T[`*trace˙file*`]`
: Run zlex in trace mode, generating extensive output in file *trace_file* which defaults to stdout. See Section 16.3.5 [Miscellaneous Options], page 65.

`--transition-compress[=1|0]`
: Compress transitions for non-switch states in code scanner (default: `0`). See Section 16.3.3 [Code Scanner Options], page 62.

`--verbose[=1|0]`
`-v[1|0]`
: Generate somewhat verbose statistics describing DFA (default: `0`). See Section 16.3.5 [Miscellaneous Options], page 65.

`--version`
`-V`
: Print version number and exit. See Section 16.3.5 [Miscellaneous Options], page 65.

`--whitespace[=1|0]`
`-w[1|0]`
: Allow whitespace within lex patterns (default: `0`). See Section 16.3.5 [Miscellaneous Options], page 65.

`--yylineno[=1|0]`
: Maintain count of line number in global variable yylineno (default: `0`). See Section 16.3.2 [Runtime Algorithm Options], page 61.

16.4 Data Search List

When Zlex is run, it looks for certain data files (a skeleton file '`zlexskl.c`' and an options file '`zlex.opt`' (see Section 16.2 [Option Sources], page 60)) in certain standard directories (the skeleton file *must* exist, but the option file need not exist). The search list specifying these standard directories is fixed when Zlex is installed; it can be printed out using Zlex's the '`--help`' option (see Section 16.3.6 [Options List], page 66).

The search list consists of a list of colon-separated directory names (the directory names may or may not have terminating slashes) or environment variables (starting with a '`$`'). If a directory name starts with a '`$`', then the first (only the first) '`$`' must be repeated. An empty component in the search list specifies the current directory. Typically the search list contains the current directory. Also typically, the environment variable `ZLEX_SEARCH_PATH` is present in the search list — this causes Zlex to check if the variable is set in the environment. If it is, then Zlex expects it to specify a search list which it recursively searches.

Typically, the search list compiled into Zlex looks something like the following:

`$ZLEX_SEARCH_PATH:.:$HOME:/usr/local/share/zlex-`*Version*

Since the search list will typically contain an environment variable like `ZLEX_SEARCH_PATH` it is possible to change the set of standard directories searched by Zlex even after installation by specifying a value for the variable. For example, if with the above search list, `ZLEX_SEARCH_PATH` is set to `/usr/lib:/usr/opt/lib`, then the effective search list becomes:

`/usr/lib:/usr/opt/lib:.:$HOME:/usr/local/share/zlex-`*Version*

17 Efficiency

To produce a high performance scanner, the Zlex programmer needs to understand the performance tradeoffs between different Zlex features.

17.1 Patterns versus Actions

The primary consideration used when designing Zlex was to maximize the performance in the basic task of a scanner: recognizing tokens. The performance of special actions was a secondary consideration, except that the presence of such actions was not allowed to impact the performance of those parts of the scanner which did not depend on them.

These design decisions make it desirable for the Zlex programmer to use patterns rather than actions whenever possible. Many of the actions involve a function call with its consequent overhead. For example, it is preferable to process comments using start states (see Section 7.6 [Start States Example], page 32), rather than processing them using `input` (see Section 8.2 [Direct Input], page 35).

17.2 Maximizing Token Length

There is some overhead involved in setting up for scanning a token and completing a token. To minimize this overhead, it is preferable to maximize the token length. For example, when scanning through a comment (see Section 7.6 [Start States Example], page 32), it is preferable to process the comment a line at a time, rather than a single character at a time.

17.3 Backtracking

Backtracking (both that caused during scanning due to overlapping patterns, and that forced by explicit `REJECT`s) will naturally lead to somewhat lower performance because characters will be scanned multiple times. The backtracking performance of Zlex is reasonably good and backtracking can be used moderately within a scanner without impacting the overall performance terribly.

17.4 Generalized Right Context Efficiency

In a generalized right context pattern of the form *RE*`/`*context* the efficiency of the pattern matching depends on the form of *RE* and *context*. If the length of the string which matches *RE* is m, and the length of the string matching *context* is n then:

- If either *RE* or *context* match only tokens of a fixed length, then the total number of characters scanned will be $m + n$.
- It neither *RE* or *context* match only tokens of a fixed length, and there is no overlap between the end of tokens which match *RE* and the beginning of tokens which match *context*, then the total number of characters scanned will be $2 * (m + n)$.
- It neither *RE* or *context* match only tokens of a fixed length, and there is overlap between the end of tokens which match *RE* and the beginning of tokens which match *context*, then in the worst-case, the total number of characters scanned can be quadratic in the length of the strings.

Hence generalized right context with the context overlapping the regular expression should be avoided if possible.

18 Distributing Zlex Scanners

When a program is completed and is ready for distribution, there are two common distribution models used:

Binary Distribution
: This is the model usually used for commercial software. In that case, the compiled Zlex generated scanner along with some of the compiled code from the Zlex library will form part of the distributed executable.

Source Distribution
: This is the usual model used for non-commercial software and is the preferred model. In that case, the distribution will need to include the generated scanner file as well as some of the sources from the Zlex library.

18.1 Distributing Zlex Library Sources

There are two possibilities for distributing the Zlex library sources:

1. Distribute the complete Zlex library. In that case, the main '`Makefile`' will need to be setup to compile the entire Zlex library and link it into the final executable.
2. Distribute only that part of the library which is really required by the generated scanner.

(1) is conceptually straightforward. (2) is also straightforward, except for figuring out which parts of the Zlex library are required by the generated scanner. Fortunately, the Zlex distribution comes with a shell script which automates that task.

The script understands the interdependencies of the modules which constitute the Zlex library. When it is run, it analyzes the object file for the generated scanner and produces a C file which contains all the source code which will be required by the generated scanner. The distribution should include this C file as well as the file '`libZlexp.h`' which will be found in the Zlex library source directory.

The script is called `mklibsrc`; it usually resides in the Zlex library source directory. It can be invoked as:

SCRIPT_PATH`/mklibsrc` *OFILE* `[`*LIB_SRC_DIR*`]` `[`*DEST_FILE*`]`

where the parameters are defined as follows:

SCRIPT_PATH
: The path to the `mklibsrc` script.

OFILE
: The name of the object file for the generated scanner.

LIB_SRC_DIR
: The directory where the Zlex library sources reside — it defaults to *SCRIPT_PATH*.

DEST_FILE
: The name of the C file which should contain the generated sources — it defaults to '`libsrc.c`'.

For example, if we assume that the environmental variable `ZLEX_LIB_SRC` contains the path to the Zlex library source directory, then the invocation:

```
$ZLEX_LIB_SRC/mklibsrc scan.o
```

will produce a file 'libsrc.c' which contains all the source code required from the Zlex library for the scanner object file 'scan.o'.

It is unlikely but possible that the generated scanner can be compiled with different options which affect which routines will be required from the Zlex library. In that case, it is necessary to repeat the above procedure for each scanner object file produced using the different options. The script will accumulate the code in the specified C file.

The mklibsrc script is automatically generated by m4 using a skeletal script. The interdependencies among the library modules are automatically extracted using a Perl script. The main portability problem within the mklibsrc script is likely to be the command 'nm -u' which is used to analyze the scanner object file.

19 Bugs and Deficiencies

Internal versions of the Zlex scanner generator have been used by me since late 1993. In 1996, it was used by about 20 students in a compiler course: they uncoverd 2 bugs. There has been a major rewrite since then.

19.1 Suspicions

Normally, when a program has been completed, one has a reasonable idea where bugs might still lurk. On that basis, for what they are worth, I present my current suspicions:

The Scanner Generator
: I feel fairly good about the code which actually generates scanners. Under normal error-free operation, the weakest area may be the code which compresses the scanner tables. However, the bugs most likely to manifest themselves will probably be `assert`ion failures caused by an erroneous Zlex source file. In that case, the Zlex programmer can simply correct the error in the Zlex source file and continue on with reasonable confidence (after submitting a bug report of course).

The Runtime System
: Unfortunately the runtime system (the generated scanner along with the Zlex library) is rather complex. Because of the need for backward compatibility, it has many poorly integrated features. Bugs probably lurk within the complexity and in the interactions among the features.

19.2 Bug Reports

First you will need to be sure that you have found a Zlex bug:

- If when running the Zlex scanner generator, it bombs with a assertion failure or a core dump, you can be sure that you have uncovered a bug.
- If when running a Zlex generated scanner, it bombs or does not do what you intended it to do, it is much more difficult to be sure whether the problem is within the Zlex runtime system or in your code. If it bombs within a Zlex library routine, make sure that you called it with the proper parameters. If it bombs within the main scanner function, make sure that it is not within your actions.

If you are sure that you have uncovered a bug, try to distil it down to a test program which is as short as possible while still exhibiting the bug. Record a log which exhibits the bug. Make sure that you mention the version of Zlex you are using in your bug report.

Bug reports can be mailed to:

```
zdu@acm.org
```

Appendix A Syntax of Zlex Programs

An informal description of the lexical and grammatical syntax of Zlex programs follows:

A.1 Tokens

This is an informal description of the lexical syntax of non-trivial Zlex tokens.

`ACT_TOK` A sequence of actions including C-brace actions, decorated brace actions, indented actions, or a newline after a section 2 pattern.

`CHAR_TOK` A character which is not a lex meta-character within its current context.

`COLON_BEGIN_TOK`
: `"[:"`.

`COLON_END_TOK`
: `":]"`.

`EOF_PAT_TOK`
: `"<<EOF>>"`.

`ID_TOK` An identifier.

`LEX_DIR_TOK`
: A lex directive at the start of a line in section 1.

`MACRO_TOK`
: A macro call within braces.

`NEXT_ACT_TOK`
: '`|`' action (which is not part of a pattern).

`NL_TOK` A newline which terminates an option line.

`NUM_TOK` A number which occurs within braces as a repetition count.

`OPTION_LINE_TOK`
: The rest of the line after a `%option`.

`OPTION_TOK`
: `^"%option"`.

`SEC_TOK` `^"%%"`.

`SS_ID_TOK`
: An identifier which is used as a start-state name.

`STARTX_TOK`
: `^"%"[xX]` signalling the start of an exclusive start state declaration.

`START_TOK`
: `^"%"[sS]` signalling the start of an inclusive start state declaration.

`X_OPTION_TOK`
: `^("%array" | "%pointer")`.

A.2 Grammar

This is a grammar for Zlex programs using a yacc-like notation. This grammar is a slightly massaged version of one extracted automatically from the current implementation of Zlex.

```
lexProgram
  : section1 SEC_TOK section2
  ;
section1
  : options restSection1
  | options
  ;
options
  : nonEmptyOptions
  | /* EMPTY */
  ;
nonEmptyOptions
  : optionLine
  | nonEmptyOptions optionLine
  ;
optionLine
  : OPTION_TOK OPTION_LINE_TOK NL_TOK
  | X_OPTION_TOK NL_TOK
  ;
restSection1
  : section1Line
  | restSection1 section1Line
  ;
section1Line
  : startDec
  | def
  | LEX_DIR_TOK OPTION_LINE_TOK NL_TOK
  ;
startDec
  : START_TOK ssDefList
  | STARTX_TOK ssDefList
  ;
ssDefList
  : ssDefList SS_ID_TOK
  | /* EMPTY */
  ;
def
  : ID_TOK regExp
  | ID_TOK
  ;
section2
  : ACT_TOK sec2Patterns
  ;
sec2Patterns
  : sec2Patterns actPatterns
```

```
  | /* EMPTY */
  ;
actPatterns
  : patternActions
  | '+' regExp ACT_TOK
  ;
patternActions
  : pattern ACT_TOK
  | pattern NEXT_ACT_TOK patternActions
  ;
pattern
  : optSSList regExp optRightContext
  | optSSList '^' regExp optRightContext
  | optSSList rightContext
  | optSSList '^' rightContext
  | optSSList EOF_PAT_TOK
  ;
optRightContext
  : rightContext
  | /* EMPTY */
  ;
rightContext
  : '$'
  | '/' regExp
  ;
optSSList
  : /* EMPTY */
  | '<' ssUseList '>'
  ;
ssUseList
  : ssUseList ',' SS_ID_TOK
  | SS_ID_TOK
  ;
regExp
  : regExp '|' catRegExp
  | catRegExp
  ;
catRegExp
  : catRegExp postRegExp
  | postRegExp
  ;
postRegExp
  : postRegExp '*'
  | postRegExp '?'
  | postRegExp '+'
  | postRegExp numRange
  | baseRegExp
  ;
```

```
baseRegExp
  : '(' regExp ')'
  | '.'
  | CHAR_TOK
  | MACRO_TOK
  | '[' classElements ']'
  | '[' '^' classElements ']'
  ;
classElements
  : classElement
  | classElements classElement
  ;
classElement
  : CHAR_TOK
  | CHAR_TOK '-' CHAR_TOK
  | COLON_BEGIN_TOK ID_TOK COLON_END_TOK
  ;
numRange
  : '{' NUM_TOK '}'
  | '{' NUM_TOK ',' '}'
  | '{' NUM_TOK ',' NUM_TOK '}'
  ;
```

Appendix B Copying Conditions

The reason for the addendum is that the output of Zlex — the generated scanner file — contains code chunks which are verbatim copies of sizable sections of Zlex sources. These chunks include the code for parts of the 'yylex' function, as well as code for Zlex library functions. If only the terms of the GPL were to be applied to the code within the generated scanner file, the effect would be to restrict the use of Zlex output to free software. Hence this document amends the terms of the GNU General Public License to explicitly allow the use of the output of Zlex in non-free programs.

The addendum has not been added because of sympathy for people who want to make software proprietary. *Software should be free.* Unfortunately, it appears that limiting Zlex's use to free software does little to encourage people to make other software free. So the addendum makes the practical conditions for using Zlex match the practical conditions for using other free tools.

Questions and comments regarding Zlex can be directed to me at zdu@acm.org

The above conditions were derived from the copying conditions published for `bison` by the Free Software Foundation, Inc.

Concept Index

V

W

Y

Z

Index of Names Used in the Generated Scanner

Short Contents

Table of Contents