NAME

glex − lexical analyzer generator for GimML, derived from flex

SYNOPSIS

glex [−hisvwLTV? −ooutput −Pprefix] [−−help −−version] *[filename ...]*

OVERVIEW

This manual describes *glex*, a tool for generating programs that perform pattern-matching on text. This tool is very close to *flex*, from which it was derived. The manual includes both tutorial and reference sections:

Description
a brief overview of the tool

Some Simple Examples

Format Of The Input File

Patterns
the extended regular expressions used by glex

How The Input Is Matched
the rules for determining what has been matched

Actions
how to specify what to do when a pattern is matched

The Generated Scanner
details regarding the scanner that glex produces;
how to control the input source

Start Conditions
introducing context into your scanners, and
managing "mini-scanners"

Multiple Input Buffers
how to manipulate multiple input sources; how to
scan from strings instead of files

End-of-file Rules
special rules for matching the end of the input

Miscellaneous Functions
a summary of functions available to the actions

Interfacing With Gyacc
connecting glex scanners together with gyacc parsers

Options
glex command-line options, and the "%option"
directive

Performance Considerations
how to make your scanner go as fast as possible

Diagnostics
those error messages produced by glex (or scanners

it generates) whose meanings might not be apparent

Files
files used by glex

Deficiencies / Bugs
known problems with glex

See Also
other documentation, related tools

Author
includes contact information

DESCRIPTION

glex is a tool for generating *scanners:* programs which recognized lexical patterns in text. *glex* reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and GimML code, called *rules. glex* generates as output a GimML source file, **lexyy.ml,** which defines a routine **yylex.** When **yylex** is run on some machine state (of type **glex_data**), it analyzes its input for occurrences of the regular expressions. Whenever it finds one, it executes the corresponding GimML code.

SOME SIMPLE EXAMPLES

First some simple examples to get the flavor of how one uses *glex.* The following *glex* input specifies a scanner which whenever it encounters the string "username" will replace it with the user's login name:

```
%%
username    #put stdout (getlogin ())
```

where *getlogin* is defined as follows on Unix systems, for instance:

```
fun getlogin () =
  let val |[getline, kill, ...]| =
          inprocess "/usr/bin/whoami"
      val id = getline () before kill ()
  in
      substr (id, 0, size id-1)
  end;
```

By default, any text not matched by a *glex* scanner is copied to the output, so the net effect of this scanner is to copy its input file to its output with each occurrence of "username" expanded. (See file **example1.l** for the full definition.) In this input, there is just one rule. "username" is the *pattern* and the "#put stdout ..." is the *action.* The "%%" marks the beginning of the rules.

Here's another simple example:

```
%{
      val num_lines = ref 0;
      val num_chars = ref 0;
%}

%%
\n      inc num_lines; inc num_chars; continue
.       inc num_chars; continue

%%
        fun wc filename =
      let val f = infile filename
```

```
        val yyd = glex_data (f, fn _ => true)
    in
        num_lines := 0;
        num_chars := 0;
        yylex yyd;
        #close f ();
        #put stdout "# of lines = ";
        print stdout (pack (!num_lines));
        #put stdout ", # of chars = ";
        print stdout (pack (!num_chars));
        #put stdout "\n";
        #flush stdout ()
    end;
```

This scanner counts the number of characters and the number of lines in its input (it produces no output other than the final report on the counts). The first line declares two globals, "num_lines" and "num_chars", which are accessible both inside **yylex** and in the **wc** routine declared after the second "%%". There are two rules, one which matches a newline ("\n") and increments both the line count and the character count, and one which matches any character other than a newline (indicated by the "." regular expression). See file **example2.l** for the full example.

A somewhat more complicated example:

```
  (* scanner for a toy Pascal-like language *)

  DIGIT    [0-9]
  ID       [a-z][a-z0-9]*

%%

  {DIGIT}+    { #put stdout "An integer: ";
            #put stdout (glex_text yyd);
            #put stdout "\n";
            continue
              }

  {DIGIT}+"."{DIGIT}*        {
            #put stdout "A float: ";
            #put stdout (glex_text yyd);
            #put stdout "\n";
            continue
              }

  if|then|begin|end|procedure|function        {
            #put stdout "A keyword: ";
            #put stdout (glex_text yyd);
            #put stdout "\n";
            continue
              }

  {ID}          { #put stdout "An identifier: ";
              #put stdout (glex_text yyd);
              #put stdout "\n"; continue }

  "+"|"-"|"*"|"/"    { #put stdout "An operator: ";
                #put stdout (glex_text yyd);
```

```
                #put stdout "\n"; continue }

"{"[^}\n]*"}"     (* eat up one-line comments *) continue

[ \t\n]+        (* eat up whitespace *) continue

.           { #put stdout "Unrecognized character: ";
              #put stdout (glex_text yyd);
              #put stdout "\n"; continue }

%%

fun tokenize filename =
   let val f = infile filename
       val yyd = glex_data (f, fn _ => true)
   in
     yylex yyd;
     #flush stdout ()
   end;
```

This is the beginnings of a simple scanner for a language like Pascal. It identifies different types of *tokens* and reports on what it has seen.

The details of this example (available in file **example3.l**) will be explained in the following sections.

FORMAT OF THE INPUT FILE

The *glex* input file consists of three sections, separated by a line with just **%%** in it:

```
definitions
%%
rules
%%
user code
```

The *definitions* section contains declarations of simple *name* definitions to simplify the scanner specification, and declarations of *start conditions,* which are explained in a later section.

Name definitions have the form:

```
name definition
```

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-white-space character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

```
{DIGIT}+"."{DIGIT}*
```

is identical to

```
([0-9])+"."([0-9])*
```

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The *rules* section of the *glex* input contains a series of rules of the form:

```
pattern   action
```

where the pattern must be unindented and the action must begin on the same line.

See below for a further description of patterns and actions.

Finally, the user code section is simply copied to **lexyy.ml** verbatim. It is used for companion routines which call the scanner. The presence of this section is optional; if it is missing, the second **%%** in the input file may be skipped, too.

In the definitions and rules sections, any *indented* text or text enclosed in **%{** and **%}** is copied verbatim to the output (with the %{}'s removed). The %{}'s must appear unindented on lines by themselves.

In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors (this feature is present for *POSIX* compliance; see below for other such features).

In the definitions section (but not in the rules section), an unindented comment (i.e., a line beginning with "(*") is also copied verbatim to the output up to the next "*)".

PATTERNS

The patterns in the input are written using an extended set of regular expressions. These are:

```
x         match the character 'x'
.         any character (byte) except newline
[xyz]     a "character class"; in this case, the pattern
          matches either an 'x', a 'y', or a 'z'
[abj-oZ]  a "character class" with a range in it; matches
          an 'a', a 'b', any letter from 'j' through 'o',
          or a 'Z'
[^A-Z]    a "negated character class", i.e., any character
          but those in the class.  In this case, any
          character EXCEPT an uppercase letter.
[^A-Z\n]  any character EXCEPT an uppercase letter or
          a newline
r*        zero or more r's, where r is any regular expression
r+        one or more r's
r?        zero or one r's (that is, "an optional r")
r{2,5}    anywhere from two to five r's
r{2,}     two or more r's
r{4}      exactly 4 r's
{name}    the expansion of the "name" definition
          (see above)
"[xyz]\"foo"
          the literal string: [xyz]"foo
\X        if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v',
          then the ANSI-C interpretation of \x.
          Otherwise, a literal 'X' (used to escape
          operators such as '*')
\0        a NUL character (ASCII code 0)
\123      the character with octal value 123
\x2a      the character with hexadecimal value 2a
(r)       match an r; parentheses are used to override
          precedence (see below)
```

rs the regular expression r followed by the regular expression s; called "concatenation"

r|s either an r or an s

r/s an r but only if it is followed by an s. The text matched by s is included when determining whether this rule is the "longest match", but is then returned to the input before the action is executed. So the action only sees the text matched by r. This type of pattern is called trailing context". (There are some combinations of r/s that glex cannot match correctly; see notes in the Deficiencies / Bugs section below regarding "dangerous trailing context".)

^r an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).

r$ an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Note that glex's notion of "newline" is exactly whatever the C compiler used to compile glex interprets '\n' as; in particular, on some DOS systems you must either filter out \r's in the input yourself, or explicitly use r/\r\n for "r$".

<s>r an r, but only in start condition s (see below for discussion of start conditions)

<s1,s2,s3>r same, but in any of start conditions s1, s2, or s3

<*>r an r in any start condition, even an exclusive one.

<<EOF>> an end-of-file

<s1,s2><<EOF>> an end-of-file when in start condition s1 or s2

Note that inside of a character class, all regular expression operators lose their special meaning except escape ('\') and the character class operators, '-', ']', and, at the beginning of the class, '^'.

The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence. For example,

```
foo|bar*
```

is the same as

```
(foo)|(ba(r*))
```

since the '*' operator has higher precedence than concatenation, and concatenation higher than alternation ('|'). This pattern therefore matches *either* the string "foo" *or* the string "ba" followed by zero-or-more r's. To match "foo" or zero-or-more "bar"'s, use:

```
foo|(bar)*
```

and to match zero-or-more "foo"'s-or-"bar"'s:

```
(foo|bar)*
```

In addition to characters and ranges of characters, character classes can also contain character class *expressions.* These are expressions enclosed inside **[:** and **:]** delimiters (which themselves must appear between the '[' and ']' of the character class; other elements may occur inside the character class, too). The valid expressions are:

```
[:alnum:] [:alpha:] [:blank:]
[:cntrl:] [:digit:] [:graph:]
[:lower:] [:print:] [:punct:]
[:space:] [:upper:] [:xdigit:]
```

These expressions all designate a set of characters equivalent to the corresponding standard C **isXXX** function. For example, **[:alnum:]** designates those characters for which **isalnum()** returns true - i.e., any alphabetic or numeric. Some systems don't provide **isblank(),** so glex defines **[:blank:]** as a blank or a tab.

For example, the following character classes are all equivalent:

```
[[:alnum:]]
[[:alpha:][:digit:]
[[:alpha:]0-9]
[a-zA-Z0-9]
```

If you run scanner in case-insensitive mode, then **[:upper:]** and **[:lower:]** are equivalent to **[:alpha:].**

Some notes on patterns:

- A negated character class such as the example "[^A-Z]" above *will match a newline* unless "\n" (or an equivalent escape sequence) is one of the characters explicitly present in the negated character class (e.g., "[^A-Z\n]"). This is unlike how many other regular expression tools treat negated character classes, but unfortunately the inconsistency is historically entrenched. Matching newlines means that a pattern like [^"]* can match the entire input unless there's another quote in the input.
- A rule can have at most one instance of trailing context (the '/' operator or the '$' operator). The start condition, '^', and "<<EOF>>" patterns can only occur at the beginning of a pattern, and, as well as with '/' and '$', cannot be grouped inside parentheses. A '^' which does not occur at the beginning of a rule or a '$' which does not occur at the end of a rule loses its special properties and is treated as a normal character.

  The following are illegal:

  ```
  foo/bar$
  <sc1>foo<sc2>bar
  ```

  Note that the first of these, can be written "foo/bar\n".

  The following will result in '$' or '^' being treated as a normal character:

  ```
  foo|(bar$)
  foo|^bar
  ```

If what's wanted is a "foo" or a bar-at-the-beginning-of-a-line, the following could be used (the special '|' action is explained below):

```
foo     |
^bar    (* action goes here *)
```

However, to match a "foo" or a bar-followed-by-a-newline, the following cannot be used:

```
foo     |
bar$    (* action goes here *)
```

Although flex would accept it (making the scanner considerably slower), glex refuses it, and you would have to write something like:

```
foo     (* action goes here *)
bar$    (* same action repeated here *)
```

HOW THE INPUT IS MATCHED

When the generated scanner is run, it analyzes its input looking for strings which match any of its patterns. If it finds more than one match, it takes the one matching the most text (for trailing context rules, this includes the length of the trailing part, even though it will then be returned to the input). If it finds two or more matches of the same length, the rule listed first in the *glex* input file is chosen.

Once the match is determined, the whole scanner machine state is made available in a variable named **yyd.** The text corresponding to the match (called the *token)* can be obtained by calling **glex_text** on **yyd,** and its length can be computed by calling the **size** function on the latter text. If you just wish to get the length without the text, calling **glex_length yyd** in faster. The *action* corresponding to the matched pattern is then executed (a more detailed description of actions follows), and then the remaining input is scanned for another match.

If no match is found, then the *default rule* is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal *glex* input is:

```
%%
```

which generates a scanner that simply copies its input (one character at a time) to its output.

The buffer where the scanner puts the value of **glex_text yyd** grows dynamically to accommodate large tokens. While this means your scanner can accommodate very large tokens (such as matching entire blocks of comments), bear in mind that each time the scanner must resize its buffer it also must rescan the entire token from the beginning, so matching such tokens can prove slow. This buffer presently does *not* dynamically grow if a call to **glex_unput** results in too much text being pushed back; instead, an exception **Glex 3** is raised.

ACTIONS

Each pattern in a rule has a corresponding action, which can be any arbitrary GimML expression returning a value of type **'token option,** where **'token** is the type of returned tokens. The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action. Empty actions are illegal, contrarily to **flex:** compiling **lexyy.ml** will produce a type error, since every action must return a token **SOME token** or inform **yylex** that it should proceed with the next token, by returning **NONE.** Abbreviations are provided to make this meaning more explicit, and **return token** has the same effect as **SOME token,** while **continue** has the same effect as **NONE.** For example, here is the specification for a program which deletes all occurrences of "zap me" from its input:

```
%%
"zap me"   continue
```

(It will copy all other characters in the input to the output since they will be matched by the default rule.)

Here is a program which compresses multiple blanks and tabs down to a single blank, and throws away whitespace found at the end of a line (see file **example5.l**):

```
%%
[ \t]+       #put stdout " "; continue
[ \t]+$      continue (* ignore this token *)
```

If the action contains a '{', then the action spans till the balancing '}' is found, and the action may cross multiple lines. Similarly if the action contains a '(', then the action spans till the balancing ')' is found. *glex* knows about GimML strings and comments and won't be fooled by braces found within them, but also allows actions to begin with **%{** and will consider the action to be all the text up to the next **%}** (regardless of ordinary braces inside the action).

An action consisting solely of a vertical bar ('|') means "same as the action for the next rule." See below for an illustration.

Actions can include arbitrary GimML code, and must compute a value of either the form **return** applied to some token that we wish to return to whatever routine called **yylex,** or of the form **continue** meaning that we wish to let **yylex** continue scanning. Each time **yylex()** is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

Contrarily to **flex,** actions are not free to modify the piece of text just recognized by **yylex,** and can only read it by calling **glex_text yyd** where **yyd** is a variable that will be defined in each action and will hold the current glex machine state.

There are a number of special directives which can be included within an action:

- **glex_begin (yyd,** <*start-condition-name*> **)** where <*start-condition-name*> is the name of a start condition places the scanner in the corresponding start condition (see below).
- **glex_less (yyd,** *n* **)** returns all but the first *n* characters of the current token back to the input stream, where they will be rescanned when the scanner looks for the next match. **glex_text yyd** is adjusted appropriately (e.g., **size (glex_text yyd)** or equivalently **glex_length yyd** will now be equal to *n*). For example, on the input "foobar" the following will write out "foobarbar":

  ```
  %%
  foobar    { #put stdout (glex_text yyd);
              glex_less(yyd, 3); continue }
  [a-z]+    #put stdout (glex_text yyd); continue
  ```

  An argument of 0 to **glex_less** will cause the entire current input string to be scanned again. Unless you've changed how the scanner will subsequently process its input (using **glex_begin,** for example), this will result in an endless loop. Calling **glex_less** on a negative number raises exception **Glex 6** ("less on negative argument"), while calling **glex_less** on an argument exceeding **glex_length yyd** raises exception **Glex 7** ("less on argument>glex_length").
- **glex_unput (yyd,** *c* **)** puts the character *c* back onto the input stream. It will be the next character scanned. The following action will take the current token and cause it to be rescanned enclosed in parentheses.

  ```
  {
  let val scanned = explode (glex_text yyd)
  (* copy text because glex_unput trashes glex_text. *)
  in
    glex_unput (yyd, ord ")");
    iterate
      glex_unput (yyd, ord c)
    | c in list rev scanned
  ```

```
  end;
  glex_unput (yyd, ord ")");
  continue
end
}
```

Note that since each **glex_unput** puts the given character back at the *beginning* of the input stream, pushing back strings must be done back-to-front.

An important potential problem when using **glex_unput** is that a call to **glex_unput** *destroys* the contents of *glex_text yyd*, starting with its rightmost character and devouring one character to the left with each call. If you need the value of yytext preserved after a call to **glex_unput** (as in the above example), you must first copy it elsewhere.

Finally, note that you cannot put back **EOF** to attempt to mark the input stream with an end-of-file.

- **glex_input yyd** reads the next character from the input stream. For example, the following is one way to eat up C comments:

```
%%
"/*" {
    let datatype state = SCANNING | FOUND_STAR
                       | END_OF_COMMENT
        val now = ref SCANNING
    in
      while !now<>END_OF_COMMENT do
       let val c = glex_input yyd
       in
         if c=ord "*"
           then now := FOUND_STAR
         else if c=ord "/"
           then (case !now of
                   FOUND_STAR =>
                   now := END_OF_COMMENT
                 | _ => now := SCANNING)
         else if c = ~1
           then error "EOF in comment"
           (* where error is user-defined *)
         else now := SCANNING
       end;
       continue
    end
    }
```

- **glex_current_buffer yyd** returns the scanner's current buffer. This is used mostly as argument to the next function.
- **glex_flush (yyd,** *buffer* **)** flushes the given buffer. The main use of **glex_flush** is to flush the scanner's internal buffer, by calling **glex_flush (yyd, glex_current_buffer yyd),** so that the next time the scanner attempts to match a token, it will first refill the buffer using **glex_input** (see The Generated Scanner, below). The general meaning of **glex_flush** is described below in the section Multiple Input Buffers.
- **yyterminate yyd** can be used to return the token 0. This means end of file to Gyacc generated parsers, and therefore indicates "all done" to the scanner's caller. **yyterminate** is also called when an end-of-file is encountered. **yyterminate** is not a GimML primitive, and is defined locally to the generated scanner by **glex.** Using the **-t** option tells **glex** not to generate any definition for

yyterminate, allowing you to define your own version of **yyterminate** instead. Note that, as soon as **glex** needs to define a default end-of-file action (e.g., you didn't specify an action for some **<<EOF>>** pattern), **glex** will define it as **yyterminate.** Since the latter returns 0, this forces the return type of **glex,** i.e., the type of returned tokens, to be **int.** Using the **-t** option therefore allows you to recover polymorphism in the generated scanner, or simply to have the generated scanner return tokens of some other type than **int.**

A number of options available in **flex** are **not** available in **glex,** namely the **REJECT** and **yymore()** options. They are hacks, were rarely used, proved to be hard to implement in **glex** and also had the effect of slowing down **flex** scanners considerably. For these reasons, it was decided to leave them out of **glex** scanners.

THE GENERATED SCANNER

The output of *glex* is the file **lexyy.ml,** which contains the scanning routine **yylex,** a number of tables used by it for matching tokens, and a number of auxiliary routines and macros, all of them being **local** to the definition of **yylex.** The type of **yylex** is **glex_data -> 'token** where **glex_data** is the type of internal scanner states, and **'token** is the type of returned tokens, usually **int.**

You can create an initial internal scanner state by using the function **glex_data : |[get : int -> string,getline : unit -> string,... : 'a]| * (unit -> bool) -> glex_data.** In other words, **glex_data** takes an input stream providing functions **get** to read a fixed number of characters, and **getline** to read a line, and also a function **yywrap : unit -> bool,** and returns the initial machine state. For example, you may define

```
val yydata = glex_data (stdin, fn _ => true);
```

to have the scanner read from stdin. The role of the **yywrap** function is discussed later; let's just say that having it return **true** always is the standard choice for scanners working only on one input source.

Whenever **yylex yydata** is called, it scans tokens from the input file *stdin,* since this is what was specified as input source in the definition of **yydata** above. It continues until it either reaches an end-of-file (at which point it returns the value 0) or one of its actions returns a token by using a *return* command.

If the scanner reaches an end-of-file, subsequent calls are undefined unless the input source is changed through the use of **glex_switch** (see Multiple Input Buffers below).

If **yylex** stops scanning due to having executed some action that returns a token through the **return** function. The scanner may then be called again and it will resume scanning where it left off.

By default the scanner reads its input one line at a time, by using the **getline** field of the input stream given to **glex_data.** This allows scanning from stdin, say, by interleaving line entries by the user and scanning activities. Such a scanner is called *interactive* for this reason. Calling **glex_set_interactive (yyd, false)** where **yyd** is the internal scanner state changes this to a non-interactive scanner, which will instead read whole blocks of the input stream at once, disregarding line breaks, using the **get** field of the input stream given to **glex_data.** Non-interactive scanners are meant to be faster than interactive scanners, but may appear to behave strangely on stdin. Putting a scanner back to interactive mode is done by calling **glex_set_interactive (yyd, true),** and you can discover whether a scanner is interactive or not by calling **glex_interactive yyd.**

When the scanner receives an end-of-file indication from its input source, it then checks the **yywrap** function that was provided to **glex_data.** If **yywrap()** returns false, then it is assumed that the function has gone ahead and changed the input source to some other by using **glex_switch,** and scanning continues. If it returns true, then the scanner calls the corresponding end-of-file action. Note that in either case, the start condition remains unchanged; it does *not* revert to **INITIAL.**

START CONDITIONS

glex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with "<sc>" will only be active when the scanner is in the start condition named "sc". For example,

```
<STRING>[^"]*        { (* eat up the string body ... *)
        ...
        }
```

will be active only when the scanner is in the "STRING" start condition, and

```
<INITIAL,STRING,QUOTE>\.        { (* handle an escape ... *)
        ...
        }
```

will be active only when the current start condition is either "INITIAL", "STRING", or "QUOTE".

Start conditions are declared in the definitions (first) section of the input using unindented lines beginning with either **%s** or **%x** followed by a list of names. The former declares *inclusive* start conditions, the latter *exclusive* start conditions. A start condition is activated using the **glex_begin** function, e.g. **glex_begin (yyd,** <sc> **).** Until the next **glex_begin** action is executed, rules with the given start condition will be active and rules with other start conditions will be inactive. If the start condition is *inclusive*, then rules with no start conditions at all will also be active. If it is *exclusive*, then *only* rules qualified with the start condition will be active. A set of rules contingent on the same exclusive start condition describe a scanner which is independent of any of the other rules in the *glex* input. Because of this, exclusive start conditions make it easy to specify "mini-scanners" which scan portions of the input that are syntactically different from the rest (e.g., comments).

If the distinction between inclusive and exclusive start conditions is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%

<example>foo   do_something (); continue

bar            something_else (); continue
```

is equivalent to

```
%x example
%%

<example>foo   do_something (); continue

<INITIAL,example>bar    something_else (); continue
```

Without the **<INITIAL,example>** qualifier, the *bar* pattern in the second example wouldn't be active (i.e., couldn't match) when in start condition **example.** If we just used **<example>** to qualify *bar*, though, then it would only be active in **example** and not in **INITIAL,** while in the first example it's active in both, because in the first example the **example** startion condition is an *inclusive* **(%s)** start condition.

Also note that the special start-condition specifier **<*>** matches every start condition. Thus, the above example could also have been written;

```
%x example
%%

<example>foo   do_something (); continue

<*>bar    something_else (); continue
```

The default rule (to print any unmatched character to stdout) remains active in start conditions. It is equivalent to:

```
<*>.|\n     #put stdout (glex_text yyd); continue
```

glex_begin (yyd, 0) returns to the original state where only the rules with no start conditions are active. This state can also be referred to as the start-condition "INITIAL", so **glex_begin (yyd, INITIAL)** is equivalent to **glex_begin (yyd, 0)**

To illustrate the uses of start conditions, here is a scanner which provides two different interpretations of a string like "123.456". By default it will treat it as three tokens, the integer "123", a dot (’.’), and the integer "456". But if the string is preceded earlier in the line by the string "expect-floats" it will treat it as a single token, the floating-point number 123.456 (see file **example6.l**):

```
%s expect

%%
expect-floats       glex_begin (yyd, expect); continue

<expect>[0-9]+"."[0-9]+     {
        #put stdout "Found a float, = ";
        #put stdout (glex_text yyd);
        #put stdout "\n";
        #flush stdout ();
        continue
        }
<expect>\n          {
        (* that’s the end of the line, so
         * we need another "expect-number"
         * before we’ll recognize any more
         * numbers
         *)
        glex_begin (yyd, INITIAL);
        continue
        }

[0-9]+     {
        #put stdout "Found an integer, = ";
        #put stdout (glex_text yyd);
        #put stdout "\n";
        #flush stdout ();
        continue
        }

"."         { #put stdout "Found a dot\n";
          #flush stdout (); continue }
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line. (See file **example7.l.**) Note that the declaration of **line_num** is at the beginning of what is normally the rules section. However, this declaration is indented, and this tells **glex** to ouput it verbatim into the **lexyy.ml** file. The effect is that this declares a variable **line_num,** initialized to **ref 1,** that will be seen by all actions, but which will remain local to the generated scanner: it will be invisible from the outside.

```
%x comment
%%
   val line_num = ref 1

"/*"        glex_begin (yyd, comment); continue

<comment>[^*\n]*        { (* eat anything that’s not a ’*’ *)
```

```
                    continue }
<comment>"*"+[^*/\n]*   { (* eat up '*'s not followed by '/'s *)
                    continue }
<comment>\n             inc line_num; continue
<comment>"*"+"/"        glex_begin (yyd, INITIAL); continue
```

This scanner goes to a bit of trouble to match as much text as possible with each rule. In general, when attempting to write a high-speed scanner try to match as much possible in each rule, as it's a big win.

Note that start-conditions names are really integer values and can be stored as such. Thus, the above could be extended in the following fashion:

```
%x comment foo
%%
   val line_num = ref 1
   val comment_caller = ref INITIAL

"/*"        { comment_caller := INITIAL;
              glex_begin (yyd, comment); continue }

<foo>"/*"   { comment_caller := foo;
              glex_begin (yyd, comment); continue }

<comment>[^*\n]*        { (* eat anything that's not a '*' *)
                    continue }
<comment>"*"+[^*/\n]*   { (* eat up '*'s not followed by '/'s *)
                    continue }
<comment>\n             inc line_num; continue
<comment>"*"+"/"        { glex_begin (yyd, !comment_caller);
                    continue }
```

Furthermore, you can access the current start condition using the integer-valued **glex_start** function. For example, the above assignments to *comment_caller* could instead be written

```
comment_caller := glex_start yyd
```

Note that start conditions do not have their own name-space; %s's and %x's declare names in the same fashion as **val** declarations.

Finally, here's an example of how to match C-style quoted strings using exclusive start conditions, including expanded escape sequences (see file **example9.l**):

```
%{

exception Yyerr of string * string;
fun yyerr (yyd, msg, s) = (glex_begin (yyd, 0);
        glex_flush (yyd, glex_current_buffer yyd);
        raise Yyerr (msg, s));

fun read_octal (s, i, n) = ...

val yyvalue = ref "";
%}

%x str
```

```
%%

   val string_buf = ref (outstring "");

\"      { #seek (!string_buf) 0; #truncate (!string_buf) ();
         glex_begin (yyd, str); continue }

<str>\"        { (* saw closing quote - all done *)
       glex_begin (yyd, INITIAL);
       (* return string constant token type and
        * value to parser
        *)
       yyvalue := #convert (!string_buf) ();
       return 1
       }

<str>\n        {
       (* error - unterminated string constant *)
       (* generate error message *)
       yyerr (yyd, "unterminated string constant", "")
       }

<str>\\[0-7]{1,3} { (* octal escape sequence *)
            let val yytext = glex_text yyd
               val result = read_octal (yytext, 1,
                                        size yytext)
                     in
                        if result > 0xff
                           then yyerr(yyd, "constant out of bounds: ",
                            yytext)
                        else #put (!string_buf) (chr result)
                     end; continue
       }

<str>\\[0-9]+ {
       (* generate error - bad escape sequence; something
        * like '\48' or '\0777777'
        *)
       yyerr (yyd, "bad escape sequence", glex_text yyd)
       }

<str>\\n  #put (!string_buf) "\n"; continue
<str>\\t  #put (!string_buf) "\t"; continue
<str>\\r  #put (!string_buf) "\^M"; continue
<str>\\b  #put (!string_buf) "\^H"; continue
<str>\\f  #put (!string_buf) "\f"; continue

<str>\\(.|\n)  { #put (!string_buf)
                 (substr (glex_text yyd, 1, glex_length yyd));
                 continue }

<str>[^\\\n\"]+  #put (!string_buf) (glex_text yyd); continue
```

Note in passing that the **yyerr** function, which is meant to raise an exception when some syntactically

incorrect text is encountered, calls **glex_begin** to revert the scanner to the **INITIAL** start condition. Indeed, if an error is encountered while parsing a string, i.e. when the start condition is **str,** this allows the next call to **yylex** to continue scanning as though we had reverted to normal scanning. Otherwise the scanner will still expect to read in the rest of a string. Note that **yyerr** also calls **glex_flush** to discard any unmatched input. Assuming the scanner is interactive (which it is by default), the next call will refill the buffer by reading the next line.

Often, such as in some of the examples above, you wind up writing a whole bunch of rules all preceded by the same start condition(s). Glex makes this a little easier and cleaner by introducing a notion of start condition *scope.* A start condition scope is begun with:

```
<SCs>{
```

where *SCs* is a list of one or more start conditions. Inside the start condition scope, every rule automatically has the prefix *<SCs>* applied to it, until a *'}'* which matches the initial *'{'.* So, for example,

```
<ESC>{
   "\\n"   return "\n"
   "\\r"   return "\^M"
   "\\f"   return "\f"
   "\\0"   return "\^@"
}
```

is equivalent to:

```
<ESC>"\\n"  return "\n"
<ESC>"\\r"  return "\^M"
<ESC>"\\f"  return "\f"
<ESC>"\\0"  return "\^@"
```

Start condition scopes may be nested.

Three routines are available for manipulating stacks of start conditions:

glex_push_state : glex_data * int -> unit;

Calling **glex_push_state (yyd,** *new_state* **)** pushes the current start condition onto the top of the start condition stack and switches to *new_state* as though you had used **glex_begin (yyd,** *new_state* **)** (recall that start condition names are also integers).

glex_pop_state : glex_data -> unit;

Calling **glex_pop_state yyd** pops the top of the stack and switches to it via **glex_begin.** If the stack was empty, it raises exception **Glex 5** instead.

glex_top_state : glex_data -> int;

Calling **glex_top_state yyd** returns the top of the stack without altering the stack's contents. This raises **Glex 5** if the stack is empty.

The start condition stack grows dynamically and so has no built-in size limitation.

MULTIPLE INPUT BUFFERS

Some scanners (such as those which support "include" or "use" or "open" directives to include files) require reading from several input streams. To this end, *glex* provides a mechanism for creating and switching between multiple input buffers. An input buffer is created by using:

```
glex_buffer : glex_data
  * |[get : int -> string,getline : unit -> string,... : 'a]|
  -> glex_buffer
```

glex_buffer (yyd, stream) takes an internal scanner machine state **yyd** and an input stream **stream** and

creates a buffer associated with the given stream. It returns an object of type **glex_buffer,** which may then be passed to other routines (see below). You select a particular buffer to scan from using:

```
glex_switch : glex_data * glex_buffer -> unit
```

Calling **glex_switch (yyd,** *new_buffer* **)** switches the scanner's input buffer so subsequent tokens will come from *new_buffer.* It is recommended to use **glex_switch** inside **yywrap** to set things up for continued scanning, when closing an included file. Note also that switching input sources via either **glex_switch** or **yywrap** does *not* change the start condition. You can also clear the current contents of a buffer using:

```
glex_flush : glex_data * glex_buffer -> unit
```

This function discards the buffer's contents, so the next time the scanner attempts to match a token from the buffer, it will first fill the buffer anew using **glex_input** Note that **glex_flush** takes two arguments. The main argument is the second one, the buffer whose contents we wish to discard. So one may wonder why a first argument of type **glex_data** is needed. The reason is that, after discarding the contents of *buffer,* **glex_flush (yyd,** *buffer* **)** examines whether *buffer* is **yyd's** current buffer. If so, it tells **glex** to refill the buffer using **glex_input** next time it is called.

Finally, the **glex_current_buffer** function returns an object of type **glex_buffer,** the current buffer of the argument internal scanner machine state given as argument.

Here is an example of using these features for writing a scanner which expands include files (see file **example10.l;** the **<<EOF>>** feature is discussed below):

```
%{
exception CannotOpen of string;
%}

(* the "incl" state is used for picking up the name
 * of an include file
 *)
%x incl

%{
val include_stack = ref (nil : glex_buffer list);
fun last (a::nil) = a
  | last (_ :: rest) = last rest;
%}

%%
include           glex_begin (yyd, incl); continue

[a-z]+            #put stdout (glex_text yyd); continue
[^a-z\\n]*\n?     #put stdout (glex_text yyd); continue

<incl>[ \t]*      (* eat the whitespace *) continue
<incl>[^ \t\n]+   { (* got the include file name *)
      let val yytext = glex_text yyd
            (* save yytext *)
      in
      include_stack := glex_current_buffer yyd
                  :: !include_stack;
      let val yyin = infile yytext
      in
        glex_switch (yyd, glex_buffer (yyd, yyin));
```

```
      glex_begin (yyd, INITIAL);
      continue
    end handle IO _ => (glex_begin (yyd, INITIAL);
                        glex_flush (yyd,
                          glex_current_buffer yyd);
                        (case !include_stack of
                             nil => ()
                           | _ =>
                             glex_switch (yyd,
                                last (!include_stack)));
                        include_stack := nil;
                        raise CannotOpen yytext)
    end
    }

  <<EOF>> {
    case !include_stack of
        nil => yyterminate ()
      | buf::rest =>
       (glex_switch (yyd, buf);
        include_stack := rest;
        continue)
    }
```

END-OF-FILE RULES

The special rule "<<EOF>>" indicates actions which are to be taken when an end-of-file is encountered and yywrap() returns **true** (i.e., indicates no further files to process). The action must finish by doing one of three things:

- executing a *return* statement;
- executing the special **yyterminate** action; this actually does *return 0* by default;
- or, switching to a new buffer using **glex_switch** as shown in the example above.

<<EOF>> rules may not be used with other patterns; they may only be qualified with a list of start conditions. If an unqualified <<EOF>> rule is given, it applies to *all* start conditions which do not already have <<EOF>> actions. To specify an <<EOF>> rule for only the initial start condition, use

```
<INITIAL><<EOF>>
```

These rules are useful for catching things like unclosed comments. An example:

```
%x quote
%%

...other rules for dealing with quotes...

<quote><<EOF>>   {
       error( "unterminated quote" );
       yyterminate()
       }
<<EOF>>  {
       case !include_stack of
           nil => yyterminate ()
         | buf::rest =>
```

```
    (glex_switch (yyd, buf);
     include_stack := rest;
     continue)
}
```

MISCELLANEOUS FUNCTIONS

- **glex_set_interactive : glex_data * bool -> unit** can be used to control whether the current buffer is considered *interactive.* An interactive buffer is processed slightly more slowly than a non-interactive one, but must be used when the scanner's input source is indeed interactive to avoid problems due to waiting to fill buffers (see the discussion of the **−I** flag below). A **true** value in the macro invocation marks the buffer as interactive, a zero value as non-interactive. **glex_set_interactive** must be invoked prior to beginning to scan the buffer that is (or is not) to be considered interactive.
- **glex_interactive : glex_data -> bool** returns whether the given machine state is interactive or not.
- **glex_set_bol : glex_data * bool -> unit** can be used to control whether the current buffer's scanning context for the next token match is done as though at the beginning of a line. A **true** macro argument makes rules anchored with
- **glex_at_bol : glex_data -> bool** returns **true** if the next token scanned from the current buffer will have '^' rules active, false otherwise.
- **glex_text : glex_data -> string** returns the text of the current token. The value returned by **glex_text** may change after calls to other scanner functions.
- **glex_length : glex_data -> int** returns the length of the current token. **glex_length yyd** is the same as calling **size (glex_text yyd),** except it is faster.
- **glex_current_buffer : glex_data -> glex_buffer** returns the current buffer associated with the given scanner machine state.
- **glex_loc : glex_data -> intarray** returns a mutable array of 4 integers (start line, start position, end line, end position) giving the location of the current scanned token in the current buffer associated with the given scanner machine state.
- **glex_start : glex_data -> init** returns an integer value corresponding to the current start condition. You can subsequently use this value with **glex_begin : glex_data * int -> unit** to return to that start condition.

INTERFACING WITH GYACC

One of the main uses of *glex* is as a companion to the *gyacc* parser-generator. *gyacc* parsers expect to call a scanning routine to find the next input token. This routine is given as second argument to the **gyacc_data** function, which builds an initial parser machine state. This scanning routine may be named as you wish; **glex** will provide one called **yylex** The **yylex** routine is supposed to return the type of the next token as well as putting any associated value in some reference **yylval** that is given as fourth argument to **gyacc_data.** To use *glex* with *gyacc*, you have to be aware that *gyacc* generates a file **parse_tab_h.ml** from the grammar file **parse.y.** The **parse_tab_h.ml** contains definitions of all the **%tokens** appearing in the *gyacc* input. You should then write **open parse_tab_h** in the declarations section of the *glex* scanner to make it aware of all the **%tokens** that should be returned to the *gyacc* generated parser. For example, if one of the tokens is "TOK_NUMBER", part of the scanner might look like:

```
%{
open "parse_tab_h";
%}

%%

[0-9]+      yyvalue := glex_text yyd; return TOK_NUMBER
```

- **Glex : int -> exn** is the constructor for all exceptions returned by **glex** generated scanners. A text explanation of an exception **Glex** *n* can be obtained by calling **glexmsg** *n*. For example, **glexmsg 2** returns **scanner input buffer overflow .**

OPTIONS

glex has the following options:

−h generates a "help" summary of *glex's* options to *stdout* and then exits. **−?** and **−−help** are synonyms for **−h.**

−i instructs *glex* to generate a *case-insensitive* scanner. The case of letters given in the *glex* input patterns will be ignored, and tokens in the input will be matched regardless of case. The matched text given in *glex_text yyd* will have the preserved case (i.e., it will not be folded).

−s causes the *default rule* (that unmatched scanner input is echoed to *stdout)* to be suppressed. If the scanner encounters input that does not match any of its rules, it raises the exception **Glex 8** (scanner jammed). This option is useful for finding holes in a scanner's rule set.

−t instructs *glex* not to provide a default definition for the **yyterminate** function. The default definition is to return **SOME 0.** The token 0 is interpreted by **gyacc** generated parsers as meaning end-of-file. The **−t** option is useful to detect unmatched end-of-file conditions, or to produce a scanner returning tokens of type other than **int.**

−v specifies that *glex* should write to *stderr* a summary of statistics regarding the scanner it generates. Most of the statistics are meaningless to the casual *glex* user, but the first line identifies the version of *glex* (same as reported by **−V),** and the next line the flags used when generating the scanner, including those that are on by default.

−w suppresses warning messages.

−L instructs *glex* to generate **#line** directives. (This is the opposite of **flex's** behavior.) With this option, *glex* peppers the generated scanner with #line directives so error messages in the actions will be correctly located with respect to either the original *glex* input file (if the errors are due to code in the input file), or **lexyy.ml** (if the errors are *glex's* fault -- you should report these sorts of errors to the email address given below). However, since *GimML* does not about #line directives, this option is off by default.

−T makes *glex* run in *trace* mode. It will generate a lot of messages to *stderr* concerning the form of the input and the resultant non-deterministic and deterministic finite automata. This option is mostly for use in maintaining *glex.*

−V prints the version number to *stdout* and exits. **−−version** is a synonym for **−V.**

−ooutput
directs glex to write the scanner to the file **output** instead of **lexyy.ml.**

−Pprefix
changes the default *yy* prefix used by *glex* for all globally-visible variable and function names to instead be *prefix.* This actually only applies to **yylex.** For example, **−Pfoo** changes the name of **yylex** to **foolex.** It does not change the name of the default output file: you have to use the **-o** option to do this.

This option lets you easily link together multiple *glex* programs into the same executable.

glex also provides a mechanism for controlling options within the scanner specification itself, rather than from the glex command-line. This is done by including **%option** directives in the first section of the scanner specification. You can specify multiple options with a single **%option** directive, and multiple directives in the first section of your glex input file.

Most options are given simply as names, optionally preceded by the word "no" (with no intervening whitespace) to negate their meaning. A number are equivalent to flex flags or their negation:

caseful or

```
case-sensitive  opposite of -i (default)

case-insensitive or
caseless        -i option

default         opposite of -s option
verbose         -v option
warn            opposite of -w option
                (use "%option nowarn" for -w)
```

Two options take string-delimited values, offset with '=':

```
%option outfile="ABC"
```

is equivalent to **-oABC,** and

```
%option prefix="XYZ"
```

is equivalent to **-PXYZ.**

PERFORMANCE CONSIDERATIONS

One area where the user can increase a scanner's performance arises from the fact that the longer the tokens matched, the faster the scanner will run. This is because with long tokens the processing of most input characters takes place in the (short) inner scanning loop, and does not often have to go through the additional work of setting up the scanning environment (e.g., **glex_text yyd)** for the action. Recall the scanner for C comments:

```
%x comment
%%
   val line_num = ref 1

"/*"        glex_begin (yyd, comment); continue

<comment>[^*\n]*        { (* eat anything that's not a '*' *)
                        continue }
<comment>"*"+[^*/\n]*   { (* eat up '*'s not followed by '/'s *)
                        continue }
<comment>\n             inc line_num; continue
<comment>"*"+"/"        glex_begin (yyd, INITIAL); continue
```

This could be sped up by writing it as:

```
%x comment
%%
   val line_num = ref 1

"/*"        glex_begin (yyd, comment); continue

<comment>[^*\n]*        { (* eat anything that's not a '*' *)
                        continue }
<comment>[^*\n]*\n      inc line_num; continue
<comment>"*"+[^*/\n]*   continue
<comment>"*"+[^*/\n]*\n inc line_num; continue
<comment>"*"+"/"        glex_begin (yyd, INITIAL); continue
```

Now instead of each newline requiring the processing of another action, recognizing the newlines is "distributed" over the other rules to keep the matched text as long as possible. Note that *adding* rules does *not* slow down the scanner! The speed of the scanner is independent of the number of rules or (modulo the considerations given at the beginning of this section) how complicated the rules are with regard to operators such as '*' and '|'.

A final note: *glex* is slow when matching NUL's (the character \^@, of code 0), particularly when a token contains multiple NUL's. It's best to write rules which match *short* amounts of text if it's anticipated that the text will often include NUL's.

Another final note regarding performance: as mentioned above in the section How the Input is Matched, dynamically resizing the input buffer to accommodate huge tokens is a slow process because it presently requires that the (huge) token be rescanned from the beginning. Thus if performance is vital, you should attempt to match "large" quantities of text but not "huge" quantities, where the cutoff between the two is at about 8K characters/token.

DIAGNOSTICS

warning, rule cannot be matched indicates that the given rule cannot be matched because it follows other rules that will always match the same text as it. For example, in the following "foo" cannot be matched because it comes after an identifier "catch-all" rule:

```
[a-z]+    got_identifier()
foo       got_foo()
```

warning, **−s** *option given but default rule can be matched* means that it is possible (perhaps only in a particular start condition) that the default rule (match any single character) is the only one that will match a particular input. Since **−s** was given, presumably this is not intended.

warning, all start conditions already have <<EOF>> rules means that you have tried to specify an <<EOF>> rule for all start conditions which didn't have one yet. But you have in fact specified <<EOF>> rules for all start conditions.

warning, <start-condition> specified twice: self-explanatory.

warning, trailing context made variable due to preceding '|' action: **glex** cannot cope with variable trailing context rules. The latter are rules of the form *r/s* where *r* and *s* match text of non-fixed length. The use of a '|' action together with trailing context rules makes **glex** think that they may be variable trailing context rules. See the Deficiencies / Bugs section below.

no action found - (exception **Glex 0)** An internal error in the generated scanner occurred: it reached a state for which no action exists. This means there is a bug in the way **glex** generates its tables, typically; this is not meant to happen.

end of buffer missed - (exception **Glex 1)** In old version of **glex,** this might have occurred in a scanner which is reentered after an exception has been raised from a scanner's action but not caught. This should not happen any longer.

scanner input buffer overflow - (exception **Glex 2)** This should not happen, and means that the scanner did not manage to allocate its input buffer. This should not happen, since memory allocation always succeeds in GimML.

(exception **Glex 3)** You used **glex_unput** to push back so much text that the scanner's buffer could not hold both the pushed-back text and the current token in the input buffer. Ideally the scanner should dynamically resize the buffer in this case, but at present it does not.

(exception **Glex 4)** This should not happen, and means that the scanner did not manage to reallocate its start-condition stack. This should not happen, since memory allocation always succeeds in GimML.

start-condition stack underflow - (exception **Glex 5) glex_pop_state** or **glex_top_state** was called while the start-condition stack was empty.

less on negative argument - (exception **Glex 6) glex_less** was called on a strictly negative argument.

Although **glex** itself may generate calls to **glex_less** in the case of trailing context rules, this can only be raised by calls to **glex_less** you did yourself, unless **glex** is buggy.

less on argument>glex_length (exception **Glex 7) glex_less** was called on an argument that exceeds the length of the matched input. The same comment as for the previous exception applies.

scanner jammed - (exception **Glex 8)** a scanner compiled with **−s** has encountered an input string which wasn't matched by any of its rules. This error can also occur due to internal problems.

FILES

lexyy.ml

generated scanner.

DEFICIENCIES / BUGS

Some trailing context patterns cannot be properly matched and generate warning messages ("Variable trailing context rule at line <n>: please make the head contant-length or remove the trailing context (after rule matches the beginning of the second part, such as "zx*/xy*", where the 'x*' matches the 'x' at the beginning of the trailing context. (Note that the POSIX draft states that the text matched by such patterns is undefined.)

glex does not know how to match any trailing context *r/s* where either *r* and *s* may both match text of non-fixed lengths. **flex** would be able to match some of these so-called *variable trailing context* rules, at the price of incurring a great slow-down in the generated scanner.

Furthermore, for some trailing context rules, parts which are actually fixed-length are not recognized as such, leading to the abovementioned error message. In particular, parts using '|' or {n} (such as "foo{3}") are always considered variable-length.

Combining trailing context with the special '|' action can result in *fixed* trailing context being turned into the erroneous *variable* trailing context. For example, in the following:

```
%%
abc     |
xyz/def
```

Use of **glex_unput** invalidates **glex_text** and **glex_length.**

Pattern-matching of NUL's is substantially slower than matching other characters.

Dynamic resizing of the input buffer is slow, as it entails rescanning all the text matched so far by the current (generally huge) token.

Due to both buffering of input and read-ahead, you cannot intermix calls to GimML input routines like **get** or **getline** on the same input file as the one that the generated scanner is reading, and expect it to work. Call **glex_input** instead.

The total table entries listed by the **−v** flag excludes the number of table entries needed to determine what rule has been matched. The number of entries is equal to the number of DFA states.

The *glex* internal algorithms, which are also those of *flex*, need documentation.

SEE ALSO

flex(1), lex(1), yacc(1), sed(1), awk(1).

John Levine, Tony Mason, and Doug Brown, *Lex & Yacc*, O'Reilly and Associates. Be sure to get the 2nd edition.

M. E. Lesk and E. Schmidt, *LEX − Lexical Analyzer Generator*

Alfred Aho, Ravi Sethi and Jeffrey Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley (1986). Describes the pattern-matching techniques used by *flex* (deterministic finite automata).

AUTHOR

Jean Goubault-Larrecq, by modifying the source of **flex.**

flex was created by Vern Paxson, with the help of many ideas and much inspiration from Van Jacobson. Original version by Jef Poskanzer. The fast table representation (not used in **glex)** is a partial implementation of a design done by Van Jacobson. The implementation was done by Kevin Gong and Vern Paxson.

Thanks to the many *flex* beta-testers, feedbackers, and contributors, especially Francois Pinard, Casey Leedom, Robert Abramovitz, Stan Adermann, Terry Allen, David Barker-Plummer, John Basrai, Neal Becker, Nelson H.F. Beebe, benson@odi.com, Karl Berry, Peter A. Bigot, Simon Blanchard, Keith Bostic, Frederic Brehm, Ian Brockbank, Kin Cho, Nick Christopher, Brian Clapper, J.T. Conklin, Jason Coughlin, Bill Cox, Nick Cropper, Dave Curtis, Scott David Daniels, Chris G. Demetriou, Theo Deraadt, Mike Donahue, Chuck Doucette, Tom Epperly, Leo Eskin, Chris Faylor, Chris Flatters, Jon Forrest, Jeffrey Friedl, Joe Gayda, Kaveh R. Ghazi, Wolfgang Glunz, Eric Goldman, Christopher M. Gould, Ulrich Grepel, Peer Griebel, Jan Hajic, Charles Hemphill, NORO Hideo, Jarkko Hietaniemi, Scott Hofmann, Jeff Honig, Dana Hudes, Eric Hughes, John Interrante, Ceriel Jacobs, Michal Jaegermann, Sakari Jalovaara, Jeffrey R. Jones, Henry Juengst, Klaus Kaempf, Jonathan I. Kamens, Terrence O Kane, Amir Katz, ken@ken.hilco.com, Kevin B. Kenny, Steve Kirsch, Winfried Koenig, Marq Kole, Ronald Lamprecht, Greg Lee, Rohan Lenard, Craig Leres, John Levine, Steve Liddle, David Loffredo, Mike Long, Mohamed el Lozy, Brian Madsen, Malte, Joe Marshall, Bengt Martensson, Chris Metcalf, Luke Mewburn, Jim Meyering, R. Alexander Milowski, Erik Naggum, G.T. Nicol, Landon Noll, James Nordby, Marc Nozell, Richard Ohnemus, Karsten Pahnke, Sven Panne, Roland Pesch, Walter Pelissero, Gaumond Pierre, Esmond Pitt, Jef Poskanzer, Joe Rahmeh, Jarmo Raiha, Frederic Raimbault, Pat Rankin, Rick Richardson, Kevin Rodgers, Kai Uwe Rommel, Jim Roskind, Alberto Santini, Andreas Scherer, Darrell Schiebel, Raf Schietekat, Doug Schmidt, Philippe Schnoebelen, Andreas Schwab, Larry Schwimmer, Alex Siegel, Eckehard Stolz, Jan-Erik Strvmquist, Mike Stump, Paul Stuart, Dave Tallman, Ian Lance Taylor, Chris Thewalt, Richard M. Timoney, Jodi Tsai, Paul Tuinenga, Gary Weik, Frank Whaley, Gerhard Wilhelms, Kent Williams, Ken Yap, Ron Zellar, Nathan Zelle, David Zuhn, and those whose names have slipped my marginal mail-archiving skills but whose contributions are appreciated all the same.

Thanks to Keith Bostic, Jon Forrest, Noah Friedman, John Gilmore, Craig Leres, John Levine, Bob Mulcahy, G.T. Nicol, Francois Pinard, Rich Salz, and Richard Stallman for help with various distribution headaches.

Thanks to Esmond Pitt and Earle Horton for 8-bit character support; to Benson Margulies and Fred Burke for C++ support; to Kent Williams and Tom Epperly for C++ class support; to Ove Ewerlid for support of NUL's; and to Eric Hughes for support of multiple buffers.

This work was primarily done when I was with the Real Time Systems Group at the Lawrence Berkeley Laboratory in Berkeley, CA. Many thanks to all there for the support I received.

Send comments to Jean.Goubault@dyade.fr (or to vern@ee.lbl.gov for comments about flex).