Go to the first, previous, next, last section, table of contents.

Start conditions

`flex` 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 `BEGIN` action. Until the next `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 `flex` 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();

bar            something_else();
```

is equivalent to

```
%x example
%%

<example>foo   do_something();

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

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' starting 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();
```

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

The default rule (to `ECHO' any unmatched character) remains active in start conditions. It is equivalent to:

```
<*>.|\\n     ECHO;
```

`BEGIN(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 `BEGIN(INITIAL)' is equivalent to `BEGIN(0)'. (The parentheses around the start condition name are not required but are considered good style.)

BEGIN actions can also be given as indented code at the beginning of the rules section. For example, the following will cause the scanner to enter the "SPECIAL" start condition whenever `yylex()' is called and the global variable enter_special is true:

```
        int enter_special;

%x SPECIAL
%%
        if ( enter_special )
            BEGIN(SPECIAL);

<SPECIAL>blahblahblah
...more rules follow...
```

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 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:

```
%{
#include <math.h>
%}
%s expect

%%
expect-floats        BEGIN(expect);

<expect>[0-9]+"."[0-9]+      {
            printf( "found a float, = %f\n",
                    atof( yytext ) );
            }
<expect>\n           {
            /* that's the end of the line, so
             * we need another "expect-number"
             * before we'll recognize any more
             * numbers
             */
            BEGIN(INITIAL);
            }

[0-9]+      {

Version 2.5               December 1994                        18

            printf( "found an integer, = %d\n",
                    atoi( yytext ) );
            }

"."         printf( "found a dot\n" );
```

Here is a scanner which recognizes (and discards) C comments while maintaining a count of the current input line.

```
%x comment
%%
```

```
        int line_num = 1;

"/*"         BEGIN(comment);

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(INITIAL);
```

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
%%
        int line_num = 1;
        int comment_caller;

"/*"         {
             comment_caller = INITIAL;
             BEGIN(comment);
             }

...

<foo>"/*"    {
             comment_caller = foo;
             BEGIN(comment);
             }

<comment>[^*\n]*        /* eat anything that's not a '*' */
<comment>"*"+[^*/\n]*   /* eat up '*'s not followed by '/'s */
<comment>\n             ++line_num;
<comment>"*"+"/"        BEGIN(comment_caller);
```

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

```
comment_caller = YY_START;
```

Flex provides `YYSTATE` as an alias for `YY_START` (since that is what's used by AT&T `lex`).

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

Finally, here's an example of how to match C-style quoted strings using exclusive start conditions, including expanded escape sequences (but not including checking for a string that's too long):

```
%x str

%%
        char string_buf[MAX_STR_CONST];
        char *string_buf_ptr;

\"      string_buf_ptr = string_buf; BEGIN(str);

<str>\"        { /* saw closing quote - all done */
        BEGIN(INITIAL);
        *string_buf_ptr = '\0';
        /* return string constant token type and
         * value to parser
         */
        }
```

```
<str>\n        {
        /* error - unterminated string constant */
        /* generate error message */
        }

<str>\\[0-7]{1,3} {
        /* octal escape sequence */
        int result;

        (void) sscanf( yytext + 1, "%o", &result );

        if ( result > 0xff )
                /* error, constant is out-of-bounds */

        *string_buf_ptr++ = result;
        }

<str>\\[0-9]+ {
        /* generate error - bad escape sequence; something
         * like '\48' or '\0777777'
         */
        }

<str>\\n  *string_buf_ptr++ = '\n';
<str>\\t  *string_buf_ptr++ = '\t';
<str>\\r  *string_buf_ptr++ = '\r';
<str>\\b  *string_buf_ptr++ = '\b';
<str>\\f  *string_buf_ptr++ = '\f';

<str>\\(.|\n)  *string_buf_ptr++ = yytext[1];

<str>[^\\\n\"]+        {
        char *yptr = yytext;

        while ( *yptr )
                *string_buf_ptr++ = *yptr++;
        }
```

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). Flex 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 '\r';
    "\\f"   return '\f';
    "\\0"   return '\0';
}
```

is equivalent to:

```
<ESC>"\\n"  return '\n';
<ESC>"\\r"  return '\r';
<ESC>"\\f"  return '\f';
<ESC>"\\0"  return '\0';
```

Start condition scopes may be nested.

Three routines are available for manipulating stacks of start conditions:

```
`void yy_push_state(int 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 `BEGIN new_state' (recall that start condition names are also integers).

`void yy_pop_state()'
pops the top of the stack and switches to it via BEGIN.

`int yy_top_state()'
returns the top of the stack without altering the stack's contents.

The start condition stack grows dynamically and so has no built-in size limitation. If memory is exhausted, program execution aborts.

To use start condition stacks, your scanner must include a `%option stack' directive (see Options below).

Go to the first, previous, next, last section, table of contents.