Compilers tutorial II: Advanced *lex* features

Start conditions are used to specify different *states* in which the lexical analyzer can be, based on specific rules and patterns. Each start condition includes a distinct set of regular expressions that are active when that condition is triggered. By using start conditions, *lex* allows for more flexible and modular specification of token recognition based on the lexical context.

To declare new start conditions, place their names in the declarations section of the *lex* source file:

```
%X STATE1 STATE2
```

To use a start condition in the rules section, enclose its name in `<>` at the beginning of a rule (the expression is only matched if the automaton is in that state):

```
<STATE1>expression    { printf("found expression in state 1"); }
```

To move *lex* to a specific start condition, execute `BEGIN(condition)` in the action of a rule. To return to the default initial state of *lex*, execute `BEGIN(INITIAL)` or simply `BEGIN(0)`. Notice that you may remove the parentheses and simply write `BEGIN condition` and `BEGIN 0`, if you prefer.

An example

To exemplify the usage of *lex* states, consider the following specification:

```
%X COMMENT
%%
.                       {;}
"/*"                    { BEGIN(COMMENT); }
<COMMENT>.              { ECHO; }
<COMMENT>\n             { printf(" "); }
<COMMENT>"*/"           { BEGIN(INITIAL); }
%%
extern int yylex();
int main() {
    yylex();
    return 0;
}
int yywrap() {
    return 1;
}
```

This *lex* specification ignores everything *except* comments. We could use this to check the spelling of text in code comments.

The first rule discards any character (except newline). The second rule matches `/*` and moves the lexical analyser to the `COMMENT` state. In that state, any

character (except newline) is printed while newline characters are replaced by spaces. Finally, when the closing `*/` is matched, the analyser moves back to the initial state.

Pre-declared functions and variables

For reference, the following table summarises the most relevant features of *lex*.

Name	Description
`int yylex(void)`	Call the lexical analyser
`char *yytext`	Pointer to the matched token
`yyleng`	Length of the matched token
`yylval`	Semantic value associated with a token
`YY_USER_ACTION`	Macro executed before every matched rule's action
`ECHO`	Print the matched string
`int yywrap(void)`	Called on end-of-file, return 1 to stop
`BEGIN condition`	Switch to a specific start condition
`INITIAL`	The default initial start condition (same as 0)
`%X condition(s)`	Declare the names of exclusive start conditions

All of these features should be familiar to *lex* users. An advanced feature that can simplify *lex* specifications is the `YY_USER_ACTION` macro: if we `#define` this macro, the corresponding code will be executed before every single action. Therefore, it is useful when the same code is repeated in all actions.

Exercises

The following exercises start with *your* solution to the previous exercises in file `lexer.l`. Alternatively, you could also use the original `lexer.l` file.

1. Many programming languages have block comments delimited by `/* ... */` that are allowed to span multiple lines. Modify the lexical analyser to support block comments. Specifically, it should discard all comments while maintaining the line and column numbers correctly updated. For simplicity, unterminated comments are allowed in our miniature programming language.

Test the lexical analyser on the following input:

```
factorial(integer n) =
    if n then n * factorial(n-1) else 1  /* recursive factorial
 */ #
```

The lexer should output the 19 tokens, followed by an error message on line 3, column 5, because `#` is an invalid character.

2. Modify the lexical analyser to recognize strings. For example, it should print `STRLIT("hello\n")` when given `"hello\n"` as input. Strings are sequences of characters (except "carriage return", "newline" and double quotation marks) and/or "escape sequences" delimited by double quotation marks. Escape sequences `\f`, `\n`, `\r`, `\t`, `\\` and `\"` are allowed, while any other escape sequences should show an error message like:

   ```
   Line x, column y: invalid escape sequence (\z)
   ```

Author

Raul Barbosa (University of Coimbra)

References

Levine, J. (2009). Flex & Bison: Text processing tools. O'Reilly Media.

Niemann, T. (2016) Lex & Yacc. https://epaperpress.com/lexandyacc

Barbosa, R. (2023). Petit programming language and compiler. https://github.com/rbbarbosa/Petit

Aho, A. V. (2006). Compilers: Principles, techniques and tools, 2nd edition. Pearson Education.