Compilers tutorial I: Lexical analysis

Lex is a powerful tool commonly used in text processing and compiler construction. It is designed to generate lexical analysers, also known as lexers or tokenisers. By specifying patterns using regular expressions, *lex* allows us to instruct the analysers to recognise and process specific tokens in the input text.

A bit of theory: *Lex* takes the user-specified regular expressions as input and applies an algorithm, such as Thompson's construction algorithm, to convert them into equivalent NFAs (non-deterministic finite automata). These NFAs are then converted into DFAs (deterministic finite automata) using the subset construction algorithm. The DFA generated by *lex* is represented internally as a transition table, which captures the state transitions based on input characters, enabling efficient and deterministic token recognition during lexical analysis.

Using *lex*

To use *lex*, we specify patterns using regular expressions, along with corresponding actions. *Lex* then transforms these rules into a C program that functions as the lexical analyser. When the lexical analyser is executed, it scans the input text, recognises the patterns specified in the rules and triggers the corresponding actions which are user-written code fragments.

A *lex* source file has three sections separated with the `%%` delimiter:

```
...definitions...
%%
...rules...
%%
...subroutines...
```

The *rules* section contains our lexical specification: regular expressions matching the patterns we are interested in, paired with fragments of C code. A simple specification is:

```
%%
[0-9]+                      { printf("NATURAL\n"); }
```

This *lex* specification matches natural numbers, found in the input text, and the corresponding action is to print the word `NATURAL` each time. Any other unspecified patterns are directly copied to the output without modification.

We can add a rule for decimal numerals:

```
%%
[0-9]+                      { printf("NATURAL\n"); }
[0-9]*"."[0-9]+             { printf("DECIMAL\n"); }
```

If we ran this analyser, it would replace all naturals with the word `NATURAL` and all decimals with the word `DECIMAL`, leaving any other characters unchanged.

In the *definitions* section that comes before the rules, we can place abbreviations to avoid repetitions and make specifications easier to read. For example, we can use `{digit}` instead of `[0-9]` by placing `digit` in the definitions section. In the *subroutines* section that comes after the rules, we write any C functions we need, typically including functions `main()` and `yywrap()`. Therefore, our first complete example is as follows:

```
digit   [0-9]
%%
{digit}+                { printf("NATURAL\n"); }
{digit}*"."{digit}+     { printf("DECIMAL\n"); }
%%
extern int yylex();
int main() {
    yylex();    /* run the lexical analysis automaton */
    return 0;
}
int yywrap() {  /* called on EOF, return 1 to terminate */
    return 1;
}
```

Generating and running the lexical analyser

Having the above specification in a file named `lexer.l`, we obtain the C code for the lexical analyser by entering:

```
$ lex lexer.l
```

The generated source code is written to a file called `lex.yy.c` by default. We simply compile it using a C compiler:

```
$ cc lex.yy.c -o lexer
```

The resulting executable file `lexer` reads from `stdin` and writes to `stdout`. We can then run the analyser:

```
$ ./lexer
```

Try it with integers, decimals and other tokens.

A bit of theory: The transition table which represents the lexical analysis DFA is placed in the `lex.yy.c` file (around source line ~400). If it weren't for *lex* we would have to manually create those tables.

Regular expressions

When it comes to regular expressions, different tools may have slight variations in notation. For example, *lex* uses a notation where special characters are preceded by a backslash. The table below summarises the main notations used by *lex*.

Expression	Description	Examples
`x`	Character x	`a, 1`
`\x`	x, if x is a lex operator	`\", \\`
`"xy"`	xy, even if x or y are lex operators	`".", "*", "/*"`
`[x-z]`	Any character from x to z	`[0-9]`, `[a-z]`
`[xy]`	Either character x or y	`[abc], [-+], [eE]`
`xx|yy`	Either xx or yy	`cat|dog, if|else`
`[^x]`	Any character except x	`[^\n], [^a]`
`.`	Any character except newline	
`^x`	x, at the start of a line	
`x$`	x, at the end of a line	
`x*`	x, repeated 0 or more times	`[0-9]*, a*`
`x+`	x, repeated 1 or more times	`[a-z]+, [01]+`
`x?`	Optional x	`ab?c, [+-]?`
`(x)`	x, forcing association	`(aa|bb)+`
`x{n}`	n occurrences of x	`[0-9]{2}`
`x{n,m}`	n to m occurrences of x	`a{2,5}b*`
`{xx}`	Expand xx from the definitions	`{digit}`

Exercises

The following exercises start with the above code in file `lexer.l` and the final result is a lexical analyser for a miniature programming language.

1. Modify the above code to print the token *value* along with the token class. It should, for example, print `NATURAL(10)` when given `10` as input (and similarly for decimals). Notice that *lex* provides an external variable named `yytext` that points to the current string matched by the lexer.

2. In programming languages, the names of variables and functions are generically referred to as *identifiers*. Modify the code to print `IDENTIFIER(x)` whenever an identifier `'x'` is found. Identifiers consist of non-empty sequences of letters and digits, starting with a letter.

3. *Keywords* are reserved and cannot be used as identifiers. Modify the code to recognize the following tokens: `integer`, `double`, `if`, `then` and `else`. The key is to understand that *lex* always looks for the longest match and, in case there is a tie, it chooses the rule that comes first.

4. Whitespace is ignored in most languages (Python is a notable exception). Modify the code to ignore whitespace characters: spaces, tabs and newlines. This can be achieved by matching those characters to an empty action `{;}` that simply does nothing.

5. Programming languages use punctuation marks with specific meanings. Modify the code to recognize the following tokens: `( ) = , * / + -`

6. A final rule is included to match any other character that could not be recognized. This rule must necessarily be the last one. Modify the code to show an error message whenever an unrecognized character is found. The key is to add a rule for `. { printf("error..."); }` that will match *any single character that has not been matched by other rules*. The error message should show the line and column numbers. For this, you will need to add variables to the declarations section, which may include C code delimited by `%{ ... %}`, and update the column according to the external variable `yyleng` that stores the length of the token pointed to by `yytext`.

Finally, test the complete lexical analyser on the following input:

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

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

```
IDENTIFIER(factorial)
(
INTEGER
IDENTIFIER(n)
)
=
IF
IDENTIFIER(n)
THEN

...

Unrecognized character '#' (line 3, column 5)
```

Author

Raul Barbosa (University of Coimbra)

References

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

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

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

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