CSE 450: Translation of Programming Languages

Lecture 3: Using Lex

Before we get going…

- Remember: Project 1 is due Tuesday, midnight.
  - Ask questions if anything confuses you…
  - Stay after class if you still need group partners…
- The Textbooks…
  - I will associate each project with book sections.
  - Project 1:
    - ***Lex & Yacc***, **chapters 1** and ***2***
    - ***Compilers***, **chapter 1** (an overview of compilers), **2.6** (a hand-written lexer), **3.1** (details on the role of a lexer), and **3.5** (an introduction to 'lex')

Problems for you

Given the regular expressions:

r1 = 0 (10)* 1+
r2 = (01)+ 0?

1. Find a string corresponding to r1 but not r2.
2. Find a string corresponding to r2 but not r1.
3. Find a string corresponding to both r1 and r2.
4. Find a string corresponding to neither r1 nor r2.

Implementation

A lexical analyzer must be able to do three things:

1. Remove all whitespace and comments.

2. Identify tokens within a string.

3. Return the lexeme of a found token, as well as the line number it was found on.

How do we go about implementing this?

General Regular Expressions

Lets assume that we have *n* different tokens. We must also have *n* regular expressions. Call them R_1 … R_n.

Now define $R = R_1 \mid R_2 \mid \ldots \mid R_n$.

All lexemes will be found by R. Any lexeme found by R will will also be found by at least one of R_1 … R_n.

Technically, we need to test all possible endpoints for lexemes beginning from the start of the input stream, but in practice we can typically determine when no further results are possible.

Whitespace and Error Handling

How do we handle whitespace in the input stream? And what if ‘R’ does *not* match anything in the input string?

We can add false tokens. For whitespace, this is easy:

Whitespace = {‘ ’ | ‘\t’ | ‘\n’ }

Do not do anything with the token when its found.

For an error, add a new token Error. How can this catch everything, but is only used as a last resort?

Error = .

R = Number | Keyword | Type | Identifier | … | Error

Why Lex and Yacc?

In structured programming, there are two tasks that occur over and over:

Dividing input into meaningful units

and

Discovering the relationship among those units.

These two operations turn out to be easy to automate.

Lex and Yacc

Lex reads in a collection of regular expressions, and uses it to write a C or C++ program that will perform lexical analysis. This program is almost always *faster* than one you can write by hand.

Yacc reads in the output from lex and parses it using its own set of regular expression rules. This is almost always *slower* than a hand written parser, but *much faster* to implement. Yacc stands for "Yet Another Compiler Compiler".

Using Flex

The freeware versions of lex and yacc are called "flex" and "bison".

Lex and Flex are both already installed on the Sparcs.

On Windows, you must download Flex from the course web page. Once installed on Windows, you need to run Flex on the command line.

Running Flex

First, you must write your flex configuration file. Lets call it mylang.lex

To run flex under windows, type:

```
flex.exe -omylang.cpp mylang.lex
```

This will construct the C++ source code for this lexical analyzer. To compile it, load it into visual studio, or else under cygwin type:

```
gcc -omylang mylang.cpp -ll
```

Running Flex

Under UNIX, this is a similar process:

```
flex -omylang.cpp mylang.lex
```

and

```
gcc -o mylang mylang.cpp -ll
```

In either OS, you will now have the executable for your lexical analyzer.

The Simplest lex Program

In lex, you provide a set of regular expressions and the action that should be taken with each.

For example:

```
%%
.|\n        ECHO;
%%
main() { yylex(); }
```

Is the simplest lex program. What does it do?

A slightly more complex program

```
%{
/* This program recognizes days. */
using namespace std;
#include <iostream>
%}
%%
[\t ]+          /* Ignore Whitespace */;
Monday|Tuesday|Wednesday|Thursday|Friday|
Saturday|Sunday
  { cout << yytext << " is a day."; }
[a-zA-Z]+
  { cout << yytext << " is not a day."; }
%%
main() { yylex(); }
```

A tiny bit more…

```
%{
/* This program categorizes days. */
using namespace std;
#include <iostream>
%}
%%
[\t ]+              /* Ignore Whitespace */;
Monday|Tuesday|Wednesday|Thursday|Friday
  { cout << yytext << " is a week day."; }
Saturday|Sunday
  { cout << yytext << " is a weekend."; }
[a-zA-Z]+
  { cout << yytext << " is not a day."; }
%%
main() { yylex(); }
```

Structure of a lex program

Notice that all of the lex programs seem to have three sections, separated by a pair of percent signs "%%".

Section one is the **definition** section. Here we introduce any code that we want at the top of the C program. All C code should be inside "%{" and "%}".

Section two is the **rules** section. Here we link *patterns* with the *action* that they should trigger.

Section three is the **user sub-routines** section. Lex will copy these sub-routines after the code it generates.

Definition Section

The first section of a lex program is copied at the start of generated code. In our example this consisted of:

```
%{
/* This program categorizes days. */
using namespace std;
#include <iostream>
%}
```

This is for comments, include statements, C-function pre-declarations, and can be used to setup some aspects of lex for future sections.

Rules Section

The rules section links patterns to actions.

```
[\t ]+              /* Ignore Whitespace */;
Monday|Tuesday|Wednesday|Thursday|Friday
  { cout << yytext << " is a week day."; }
Saturday|Sunday
  { cout << yytext << " is a weekend."; }
[a-zA-Z]+
  { cout << yytext << " is not a day."; }
```

Lex has a full regular expression implementation to use in your patterns. Actions can be anything from printing something out to calling a C function of yours.

User Sub-routines Section

The user sub-routines section is for any additional C or C++ code that you want to include. The only required line is:

```
main() { yylex(); }
```

This is the main function for the resulting program. Lex builds the yylex() function that is called, and will do all of the work for you.

Other functions here can be called from the rules section when certain keywords are found.

Designing Patterns

Designing the proper patterns in lex can be very tricky, but you are provided with a broad range of options for your regular expressions.

.	A dot will match any single character *except* a newline.
*,+	Star and plus used to match zero/one or more of the preceding expressions.
?	Matches zero or one copy of the preceding expression.

Designing Patterns (2)

	A logical ‘or’ statement - matches either the pattern before it, or the pattern after.
^	Matches the very beginning of a line.
$	Matches the end of a line.
/	Matches the preceding regular expression, but only if followed by the subsequent expression.

Designing Patterns (3)

[ ] Brackets are used to denote a character class, which matches any single character within the brackets. If the *first* character is a '^', this negates the brackets causing them to match any character except those listed. The '-' can be used in a set of brackets to denote a range. C escape sequences must use a '\'.

" " Match everything within the quotes literally - don't use any special meanings for characters.

( ) Group everything in the parentheses as a single unit for the rest of the expression.

Example Patterns

[0-9]
A single digit.

[0-9]+
An integer.

[0-9]+(\.[0-9]+)?
An integer or floating point number.

[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
Integer, floating point, or scientific notation.

Example Number Identification

```
%{
  using namespace std;
  #include <iostream>
%}
%%
[\t ]+  /* Ignore Whitespace */;

[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
{ cout << yytext << " :number" << endl; }

[a-zA-Z]+
{ cout << yytext << " :NOT number" << endl; }
%%
main() { yylex(); }
```

Tracking line numbers

```
%{
  using namespace std;
  #include <iostream>
  int line_num = 1;
%}
%%
[\t ]+     /* Ignore Whitespace */;

\n         { line_num++; }

[+-]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
{ cout <<line_num<<" : " << yytext << endl; }
%%
main() { yylex(); }
```

More patterns…

What regular expression can we use to detect comments?

```
#.*
```

What about literal strings?

Does this work? `\".*\"`

What about: `\"[^"]*\"`

We need to use: `\"[^"\n]*\"`

Counting Words

```
%{
#include <iostream>
using namespace std;
int char_count = 0, word_count = 0, line_count = 0;
}%
word  [^ \t\n]+
eol   \n
%%
{word}      { word_count++ ; char_count += yyleng; }
{eol}       { char_count++; line_count++; }
.           char_count++;
%%
main() {
      yylex();
      cout << line_count << " " << word_count << " "
           << char_count << endl;
}
```

Counting Words in a file…

What if we want to be able to count words in a file?

```
main(int argc, char * argv[])
{
   if (argc > 1) {
      FILE *file = fopen(argv[1], "r");
      if (!file) {
        cerr << "Error opening " << argv[1] << endl;
        exit(1);
      }
      yyin = file;
   }
   yylex();
   cout << line_count << " " << word_count << " "
        << char_count << endl;
   return 0;
}
```

Problems for you

Design a regular expression that will identify all IP addresses, such as 10.0.0.1 or 72.14.207.99 .

Design a regular expression that will find all lines with a capital ‘A’, not in the first or last position.

Write a regular expression for people names that is triggered by honorifics (Mr, Ms, Senator, etc).