. nu face match la \n

Lab 3

Lexical Analysis II

Goals

In this lab you will:

1. Learn about left-context dependency
2. Get familiar with the issue of ambiguity
3. Use your skills to process a database configuration file

Resources

Table 3.1: Lab Resources

Resource	Link
Chomsky's hierarchy	`https://www.youtube.com/watch?v=224plb3bCog`
Noam Chomsky, MIT, Fundamental issues in linguistics	`https://www.youtube.com/watch?v=r514RhgISv0`

3.1 More on Lex Actions

In the previous laboratory, you have learned about the fundamentals of scanning, with respect to Lex. You practiced using regular expressions in lex, and compiled your first scanners.

Towards the end, you learned about the variables yytext and yyleng, which are useful in extracting the matched text, and it's character count respectively.

Concept 3.1.1: variables

The last character in a matched string can be accessed by `yytext[yyleng-1]`.

Occasionally, a Lex action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation.

- **yymore()**, which can be called to indicate that the next input expression recognized is to be tacked on to the end of this input. Normally, the next input string would overwrite the current entry in yytext.
- **yyless()**, which can be called to indicate that not all the characters matched by the currently successful expression are wanted right now. The argument n indicates the number of characters in yytext to be retained. Further characters previously matched are returned to the input.

This provides the same sort of lookahead offered by the `/` operator, but in a different form. For instance, if you consider a language which defines a string as a set of characters between quotation `"` marks, and provides that to include a `"` in a string it must be preceded by a scode}. The regular expression which matches that is somewhat confusing, so that it might be preferable to write:

Listing 3.1.1: sample.l C code

```
[^"]*   {
  if (yytext[yyleng-1] == '\\')
        yymore();
  else
        /* normal user processing */
  }
```

When the above would be faced with a string such as `"abc\"def"` first match the five characters `"abc\;` then the call to `yymore()` will cause the next part of the string, `"def`, to be tacked on the end. Note that the final quote terminating the string should be picked up in the commented section.

The function yyless() might be used to reprocess text in various circumstances. Consider the C problem of distinguishing the ambiguity of `=-a`. Suppose it is desired to treat this as `=- a` but print a message. A rule might be

Listing 3.1.2: sample.l C code

```
=-[a-zA-Z]   {
  printf("Op (=-) ambiguous\n");
  yyless(yyleng-1);
  /* action for =- */
  }
```

The above rule would print a message, return the letter after the operator to the input stream, and treat the operator as `=-`. Alternatively it might be desired to treat this as `= -a`. To do this, just return the minus sign as well as the letter to the input:

Listing 3.1.3: sample.l C code

```
=-[a-zA-Z]   {
  printf("Op (=-) ambiguous\n");
  yyless(yyleng-2);
  /* action for = */
  }
```

Note that the expressions for the two cases might more easily be written `=-/[A-Za-z]` in the first case and `=/-[A-Za-z]` in the second; no backup would be required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. The possibility of `=-3`, however, makes `=-/[^ \t\n]` a still better rule.

3.2 Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires sensitivity to prior context, and there are several ways of handling such problems.

The `^` operator, for example, is a prior context operator, recognizing immediately preceding left context just as $, that recognizes immediately-following right context. Adjacent left context could be extended, to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful, since often the relevant left context appeared some time earlier, such as at the beginning of a line.

Generally, there are three means of dealing with different environments:

- a simple use of flags, when only a few rules change from one environment to another
- the use of start conditions on rules
- the possibility of making multiple lexical analyzers all run together.

In each case, there are rules which recognize the need to change the environment in which the following input text is analyzed, and set some parameter to reflect the change. This may be a flag explicitly tested by the user's action code; such a flag is the simplest way of dealing with the problem, since Lex is not involved at all.

It may be more convenient, however, to have Lex remember the flags as initial conditions on the rules. Any rule may be associated with a start condition. It will only be recognized when Lex is in that start condition.

The current start condition may be changed at any time. Finally, if the sets of rules for the different environments are very dissimilar, clarity may be best achieved by writing several distinct lexical analyzers, and switching from one to another as desired.

Consider the following problem: *copy the input to the output, changing the word magic to first on every line which began with the letter a, changing magic to second on every line which began with the letter b, and changing magic to third on every line which began with the letter c. All other words and all other lines are left unchanged.*

These requirements are so simple that the easiest way to do this job is with a flag:

Listing 3.2.1: sample.l — C code

```
int flag;
%%
^a      {flag = 'a'; ECHO;}
^b      {flag = 'b'; ECHO;}
^c      {flag = 'c'; ECHO;}
\n      {flag =  0 ; ECHO;}
magic   {
  switch (flag)
  {
    case 'a': printf("first"); break;
    case 'b': printf("second"); break;
    case 'c': printf("third"); break;
    default: ECHO; break;
  }
  }
```

To handle the same problem with start conditions, each start condition must be introduced to Lex in the definitions section with a line reading

`%Start name1 name2 ...`

where the conditions may be named in any order. The word Start may be abbreviated to s or S. The conditions may be referenced at the head of a rule with the `<>` brackets: `<name1>expression`

is a rule which is only recognized when Lex is in the start condition name1. To enter a start condition, execute the action statement `BEGIN name1;` which changes the start condition to name1. To resume the normal state, `BEGIN 0;` resets the initial condition of the Lex automaton interpreter. A rule may be active in several start conditions: `<name1,name2,name3>` is a legal prefix. Any rule not beginning with the `<>` prefix operator is always active.

The same example as before can be written:

Listing 3.2.2: sample.l — C code

```
^a                {ECHO; BEGIN AA;}
^b                {ECHO; BEGIN BB;}
^c                {ECHO; BEGIN CC;}
\n                {ECHO; BEGIN 0;}
<AA>magic         printf("first");
<BB>magic         printf("second");
<CC>magic         printf("third");
```

In the above, the logic is exactly the same as in the previous method of handling the problem, but Lex does the work rather than the user's code.

3.3 Recap from previous lab

3.3.1 Ambiguity

In the previous laboratory you learned about the way in which rules match characters from the input string, and saw that there are situations where it is possible that more than one regular expression matches the current input. Lex decides which rule by the following criteria:

1. Lex first chooses the longest match
2. Out of the rules that match and share the same character count, Lex chooses the rule that shows first in the rules table

To "overwrite" the above behavior, you should remember you can use `REJECT`.

Normally, Lex is partitioning the input string without looking for all possible matches of each expression. This means that each character is only counted once. If we take the example you have previously encountered, with *he/she* counting, you may remember that if we write the below rules for Lex,

Listing 3.3.1: sample.l — C code

```
she     s++;
he      h++;
\n      |
.       ;
```

this would not provide the expected behaviour if your desire was to also account for occurences of *he* inside the word she. This is because by default Lex matches the longest rule, in this case, the rule for *she*. The characters for *he* in this scenario are lost after *she* is processed.

Sometimes, you may want to overwrite this behavior, in which case the action `REJECT` comes in handy. `REJECT` basically means "go and execute the next possible alternative". The position of the input pointer when using this action is adjusted.

As such, when needing to count occurences of *he* even when they appear inside another word, *she*, one could use:

In this example, you may notice that *she* includes *he*, but not viceversa. However, there are situations where you may not know beforehand what characters should be part of each class, in which case you may want to use `REJECT` just in case.

Listing 3.3.2: sample.l — C code

```
he    {s++; REJECT;}
he    {h++; REJECT;}
\n    |
.     ;
```

3.3.2 More on Lex Source

By now you should know the basics about how a Lex source looks like and why. Besides defining rules and actions in Lex, you sometimes want to also define variables or functions in the Lex source. Let's see what happens to each code section you write in the Lex file when it gets transformed into the C scanner file.

1. Any line which is not part of a Lex rule or action which begins with a blank or tab is copied into the Lex generated program. Such source input prior to the first `%%` delimiter will be external to any function in the code; if it appears immediately after the first `%%`, it appears in an appropriate place for declarations in the function written by Lex which contains the actions. This material must look like program fragments, and should precede the first Lex rule. As a side effect of the above, lines which begin with a blank or tab, and which contain a comment, are passed through to the generated program. This can be

used to include comments in either the Lex source or the generated code. The comments should follow the host language convention.

2. Anything included between lines containing only `%{ and %}` is copied out as above. The delimiters are discarded. This format permits entering text like preprocessor statements that must begin in column 1, or copying lines that do not look like programs.

3. Anything after the third `%%` delimiter, regardless of formats, etc., is copied out after the Lex output.

Definitions intended for Lex are given before the first `%%` delimiter. Any line in this section not contained between `%{ and %}`, and begining in column 1, is assumed to define Lex substitution strings (translations, such as the digits translation D example from the previous laboratory).

3.4 Practice problems

To review what you have learned so far, try solving some exercises.

Exercise 3.4.1 *

Implement a scanner that copies the input to the output adding 7 to numbers between [7,12], and multiplying negative numbers [,-100], by 10. lex file

Exercise 3.4.2 *

Implement a scanner that identifies strings marked with `"`. A group of characters in the input that is considered a string may also contain quotes inside as long as they are escaped with `\`.
Example:
Input: `aaa "string\" with\" escaped\" \"string"`
Output: `"string\" with\" escaped\" \"string"`

Exercise 3.4.3 **

Write a Lex specification which allows for the following behavior.
The input is C code that includes:
1. a for statement
2. printf instructions
3. statements for variables with identifiers that consist of lowercase/uppercase letters, _ and -, but which start with a letter
4. integers

The output is a sequence of tokens of the following type:
1. NUM - for numbers
2. VAR - for variables
3. KEY - for keyword
4. operators for =, +, separators, {,} etc.

Example:
Input:

Listing 3.4.1: input.c — C code

```c
for (i = 1; i < 11; i++)
  {
    printf("%d ", i);
  }
  return 0;
```

Output:

Terminal

```
KEY ( var = NUM; var < NUM; VAR + + ) { KEY ( STRING  , VAR ) } KEY NUM ;
```

Exercise 3.4.4 **

In the zip file associated with this laboratory you should have a db_.config folder. Open it and follow the steps below.

1. Run the Hello World example. Use the default main by adding -ll to the gcc instruction.

   Terminal

   ```
   >  lex hello.l
   >  gcc -o hello lex.yy.c -ll
   >  echo "Hello from the other side"
   >  ./hello
   >
   ```

2. process a database configuration file that has the structure presented in the *test.in* file
   - Step 1: create a header file with all the symbols that the tokenizer should recognize, called myscanner.h
   - Step 2: create the input file for Lex
     (a) include myscanner.h in the first part of the file with `%{ %}`
     (b) write down the actions and rules
     (c) implement `yywrap()` to include the in your C code.
   - Step 3:

     Terminal

     ```
     >  lex myscanner.l
     >  gcc myscanner.c  lex.yy.c  -ll -o myscanner
     >  echo "db_name:" | ./myscanner
     >  echo "db nam_e:" | ./myscanner
     >  echo "db_name : nosql" | ./myscanner
     > ./myscanner < test.in
     ```

   - Step 4: Check the correct order of tokens in *step.in* - see *myscanner2.c*

- Step 5: Input - Output:

```
Terminal

> gcc myscanner2.c  lex.yy.c  -ll -o myscanner
> ./myscanner < test.in
db_type is set to mysql
db_name is set to testdata
db_table_prefix is set to test_
db_port is set to 1092
```

```
Terminal

>  ./myscanner < test_unexpected.in
db_type is set to mysql
unexpected character
db_name is set to testdata2
db_table_prefix is set to test_
db_port is set to 1092
```

```
Terminal

>  ./myscanner < test_error.in
db_type is set to mysql
Syntax error in line 1, expected an IDENTIFIER, but found 23
```