LECTURE 6

Scanning Part 2
In the previous lectures, we discussed how one might specify valid tokens in a language using regular expressions.

We then discussed how we can create a recognizer for the valid tokens specified by a regular expression using finite automata. In general,

• Create regular expressions to specify valid tokens.
• Convert regular expressions to NFA.
• Convert NFA to DFA.
• Minimize DFA.
Let’s say we completed the first four steps and now we have a minimized DFA which recognizes the regular set \{c, cac, cbc, cacac, cbcbc, cacbc, etc\} generated by \(c \ ( (a \ | \ b) \ c)^*\).

Note how this differs from last lecture’s example: we now require a starting \(c\).
How do we take our minimized DFA and practically implement a scanner? After all, finite automata are idealized machines. We didn’t actually build a physical recognizer yet! Well, we have two options.

- Represent the DFA using goto and case (switch) statements.
  - Handwritten scanners.

- Use a table to represent states and transitions. Driver program simply indexes table.
  - Auto-generated scanners.
  - The scanner generator Lex creates a table and driver in C.
  - Some other scanner generators create only the table for use by a handwritten driver.
Let’s make a sample handwritten scanner that realizes the functionality of our minimized DFA.
CASE STATEMENT STYLE

Our starting state is S1.

state = s1

Note: our pseudocode case statement does not feature fallthrough – rather, execution continues at the end of the case statement.
CASE STATEMENT STYLE

Create a loop that checks the current state.

```plaintext
state = s1
loop
    case state of
        s1:
        s2:
```
CASE STATEMENT STYLE

In each state, check the input character.

state = s1
loop
  case state of
    s1:
      case in_char of
        'c': state = s2
        else error
    s2:
      case in_char of
        'a': state = s1
        'b': state = s1
        else error
CASE STATEMENT STYLE

Append input char when valid and read new input char.

state = s1
token = ‘’
loop
  case state of
  s1:
    case in_char of
      ‘c’: state = s2
      else error
  s2:
    case in_char of
      ‘a’: state = s1
      ‘b’: state = s1
      ‘ ‘: state = s1
      ‘,’: state = s1
      return token
      else error
  token = token + in_char
read new in_char

If the new input char is a space, then return token
CASE STATEMENT STYLE

Append input char when valid and read new input char.

state = s1
token = ''
loop
  case state of
    s1:
      case in_char of
        'c': state = s2
        else error
    s2:
      case in_char of
        'a': state = s1
        'b': state = s1
        ' ': state = s1
        return token
        else error
  token = token + in_char
read new in_char

Outer cases represent our states.

Inner cases represent our transitions.
CASE STATEMENT STYLE

Append input char when valid and read new input char.

state = s1
token = ‘’
loop
  case state of
    s1:
      case in_char of
        ‘c’: state = s2
        else error
        return token
    s2:
      case in_char of
        ‘a’: state = s1
        ‘b’: state = s1
        ‘ ’: state = s1
        return token
        else error
  token = token + in_char
read new in_char

Invalid transitions are lexical errors.
There are some key differences between the formally-defined recognizer we created with the DFA and the practical code we implement.

• Peeking Ahead
  • Note that we peeked ahead in the example to see if a space was coming. How else would we know to accept our token yet?

• Identifying Keywords
  • Look like an identifier, but are actually reserved words.
  • You could build in states to recognize them but that would be messy.
  • Instead, treat it as an identifier and before you return the identifier, check if it’s a keyword.
So, why do we need to peek ahead? Why not just accept when we pick up ‘c’ or ‘cac’?

Scanners need to accept as many characters as they can to form a valid token.

For example, 3.14159 should be one literal token, not two (e.g. 3.14 and 159).

So when we pick up ‘4’, we peek ahead at ‘1’ to see if we can keep going or return the token as is. If we peeked ahead after ‘4’ and saw whitespace, we could return the token in its current form.

A single peek means we have a look-ahead of one character.
Some messy languages necessitate arbitrarily large look-aheads.

This can be accomplished by buffering all characters picked up after some possible point of acceptance. If we get an error, just place them back in the stream and return the token. Otherwise, add them to the token and keep buffering.

This can all be avoided by creating a language whose tokens aren’t commonly prefixes of one another.
An alternative approach to implementing our DFA is to create a transition table which is indexed by the state and input.

Essentially, the transition table is a two-dimensional array where

\[
\text{state} = \text{trans_table}[\text{state}][\text{input_char}]
\]
Let’s create a transition table for our example DFA.

<table>
<thead>
<tr>
<th>State</th>
<th>‘a’</th>
<th>‘b’</th>
<th>‘c’</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>S2</td>
<td>-</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S1</td>
<td>-</td>
<td>token</td>
</tr>
</tbody>
</table>

On an input of ‘c’ from S1 we can go to S2. No other input is valid from S1. On an input of ‘a’ or ‘b’ from S2 we can go to S1, but an input of ‘c’ is not valid. Alternatively, at state S2, we can return our token.
Let's create a transition table for our example DFA.

<table>
<thead>
<tr>
<th>State</th>
<th>'a'</th>
<th>'b'</th>
<th>'c'</th>
<th>Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>-</td>
<td>-</td>
<td>S2</td>
<td>-</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S1</td>
<td>-</td>
<td>token</td>
</tr>
</tbody>
</table>

Typically, the State and input columns form one table while the State and Return columns form another. They are usually separated to aid large look-ahead functionality. We can merge them here for convenience.
TABLE-DRIVEN SCANNING

How can we use the transition table to create a scanner?

A driver program uses the current state and input character to index into the table. We can either

• Move to a new state.
• Return a token.
• Raise an error.
HANDLING LEXICAL ERRORS

It is the responsibility of the driver program to handle lexical errors when the transition table reveals there is no valid action to take.

Typically, drivers will do the following:

• Throw away the current, invalid token.
• Skip forward until a character is found that can begin a new token.
• Restart the scanning algorithm.

This does not mean errors are being ignored. Rather, we’re usually trying to continue compilation to find more errors.
The driver must also save the images (spelling) of the tokens that are returned. This is useful for producing meaningful error messages as well as necessary for semantic analysis. For example, the valid tokens may be

```
id = id ;
```

This is not enough to verify that the code is semantically valid. But the images tell us

```
a = b ;
```

As long as a and b are compatible types, we’re good.
INTRO TO LEX

Lex is the single most popular scanner generator, due to its inclusion on most Unix systems.

Given tokens specified as regular expressions, Lex automatically generates a routine (yylex) that recognizes the tokens (and performs corresponding actions).

Typical usage of Lex involves creating a Lex file (.l) and passing it to the ‘lex’ program. The output is a C file which can be compiled with the Lex library (-ll). The executable is our scanner, into which we can pass some files to scan.
A Lex file takes the following form:

- **Definition section**: The definition section defines macros, imports, and code written in C.
- **Rules section**: The rules section associates regular expression patterns with C statements. When the lexer sees text in the input matching a given pattern, it will execute the associated C code.
- **User subroutines (C code)**: The C code section contains C statements and functions that are copied to the generated source file. Typically called by the rules in the rules section.
A Lex file takes the following form:

Specifically,

**Definition section**
```
%%
```

**Rules section**
```
%%
```

**User subroutines (C code)**

- **Rules:** `<regular expression>` ` <action>`
  - Each regular expression specifies a token.
  - Default action for anything that is not matched: copy to the output.

- **Action:** C/C++ code fragment specifying what to do when a token is recognized.
INTRO TO LEX

Lex regular expressions contains text characters and operators.

Letters of alphabet and digits are always text characters.
  • Regular expression `integer` matches the string “integer”.

Operators: “\[]^?-.*+ | ()$/\{\}%^<>”
  • When these characters happen in a regular expression, they have special meanings.
INTRO TO LEX

• ‘*’, ‘+’, ‘|’, ‘(‘, ‘(‘)’ -- used in regular expression.

• ‘ ‘ ‘ -- any character in between quote is a text character.

• ‘\’ -- escape character.

• ‘[‘ and ‘]’ -- used to specify a set of characters.
  • e.g: [a-z], [a-zA-Z]
  • Every character in it except ^, - and \ is a text character

• ‘^‘ -- not, used as the first character after the left bracket.
  • e.g: [^abc] – any character except ‘a’, ‘b’ or ‘c’.
INTRO TO LEX

- ‘.’ -- every character (except newline).
- ‘?’ – optional.
  - ab?c matches ‘ac’ or ‘abc’.
- ‘/’ -- used in character lookahead.
  - e.g. ab/cd -- matches ab only if it is followed by cd
- ‘{“}’ -- enclose a regular definition.
- ‘%’ -- has special meaning in lex.
- ‘$’ -- match the end of a line. ‘^’ -- match the beginning of a line.
  - ab$ == ab\n
example1.l  Counts the number of characters and lines in a file.

```c
int charcount=0, linecount=0;

charcount++;

\n{linecount++; charcount++;}

int main()
{
  yylex();
  printf("There were %d characters in %d lines\n", charcount, linecount);
  return 0;
}
```
AN EXAMPLE LEX PROGRAM

```c
#include <stdio.h>

int charcount = 0, linecount = 0;

int main()
{
    yylex();
    printf("There were %d characters in %d lines\n", charcount, linecount);
    return 0;
}
```

Definitions

Rules

C code

Remember to enclose C code within %{ and %} in the definitions.

Basic requirement – yylex() calls the generated scanner.
AN EXAMPLE LEX PROGRAM

Try the following:

$ lex example1.l
$ gcc lex.yy.c -l

The lex command always generates a C file called lex.yy.c. Compile it to try out your new scanner.

$ ./a.out < testfile.txt
AN EXAMPLE LEX PROGRAM

carnahan@diablo>ls
example1.l  testfile.txt
carnahan@diablo>lex example1.l
carnahan@diablo>ls
example1.l  lex.yy.c  testfile.txt
carnahan@diablo>gcc lex.yy.c –ll
carnahan@diablo>ls
a.out  example1.l  lex.yy.c  testfile.txt
carnahan@diablo>more testfile.txt
Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.

Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
carnahan@diablo>./a.out < testfile.txt
There were 232 characters in 2 lines
More on Lex.

Then we’ll move into Syntax Analysis.