Compiler Construction
PA03: A Lex Scanner for a Simple Calculator

Terence Henriod

September 15, 2015

Abstract

This assignment requires you to write a *flex*-style scanner for a simple four-function calculator.

1 The Assignment

The scanner should return separate tokens for each of the following:

- The '+' (addition) symbol [PLUS]
- The '-' (subtraction) symbol [MINUS]
- The '*' (multiplication) symbol [MULT]
- The '/' (division) symbol [DIV]
- The '(' and ')' (parentheses) [OPEN], [CLOSE]
- Positive integers (you need not worry about negative integers, nor about floating-point values) [INTEGER]
- The ';' (end-of-calculation) symbol [SEMI]

Whitespace such as spaces, tabs, and newlines should not be returned. All of the above tokens are predefined for you in a file called (calc.tab.h) which you should include, and you should use these symbolic values. For any other non-whitespace character, your scanner should return the symbolic value ERROR. Your scanner should terminate when end of file is reached, and should read all input from stdin.

Don't forget that you need to pass the parser the actual value of any integers you read in. yylval can be used for this purpose.

You will be given a parser and driver for this assignment in object form. Link this object (calc.tab.o) with the object created by compiling your flex output into a program called calc. The resulting program, when run, should function as a 4-function calculator. You are responsible for generating appropriate test cases for your scanner, and you may wish to create an alternate driver module to do so. (Makefile)

You may discuss this assignment with other students and work the problems together. However, your programs should be your own individual work. Remember that all assignments are to be turned in in class on the date due.

For this project I need you to link in these libraries -lfl, -lfl is for flex.

2 The Code

2.1 calc.lex

```
 /**
  * calc.lex
  *
  * This is a lex file for defining a tokenizer for a simple calculator.
  */

 /* DEFINITIONS */
%{
 #include <stdio.h>
 #include <errno.h>
 #include <limits.h>
 #include "calc.tab.h"

 #define STROL_ERROR -1
%}

ZERO_STRING   [0]
POSITIVE_INT  [1-9][0-9]*
WHITESPACE    [ \t\r\n]+
```

```
%%
 /* RULES */

{ZERO_STRING} {
  yylval = 0;
  return INTEGER;
}

{POSITIVE_INT} {
  int error = 0;
  yylval = extract_int(&error, yytext, yyleng);

  if (error != 0) {
    yyerror("The number (%s) is not properly formatted.", yytext);
    return ERROR;
  }

  return INTEGER;
}

[;] {return SEMI;}

[(] {return OPEN;}

[)] {return CLOSE;}

[+] {return PLUS;}

[-] {return MINUS;}

[*] {return MULT;}

[/] {return DIV;}

{WHITESPACE} {/* ignore */;}

. {/* unrecognized characters are errors */ return ERROR;}

%%
 /* USER CODE */

  int extract_int(int* error, char* yytext, const int yyleng) {
    char* end = yytext + yyleng;
    *error = 0;

    int number_value = strtol(yytext, &end, 10);

    if (errno == ERANGE) {
      // the documentation states that the returned value should
      // be LONG_[MIN|MAX] if there is a conversion failure, but I have found
      // that this is not the case. It seems that checking errno is the most
      // reliable method.
      *error = 1;
```

```
    }

    return number_value;
  }
```

2.2 test.sh

```
#! /bin/bash

#
# A script for testing the Flex tokenizer.
#

clear; clear; clear;

num_passed=0
num_failed=0

function unit_test {
  expression=$1
  expected=$2

  echo "~~~~~~~~~~~~~~~~~~~~~~"
  echo "CASE: "
  echo "$expression"
  echo ""

  result=$(echo $expression | bin/calc)

  if [[ $result = "" ]]; then
    result="FAIL"
  fi

  if [[ $result -eq $expected ]]; then
    message="passed"
    ((num_passed++))
  else
    message="FAILED"
    ((num_failed++))
  fi

  echo "expected: ""$expected"
  echo "got:      ""$result"
  echo ""
  echo "RESULT: "$message
  echo "~~~~~~~~~~~~~~~~~~~~~~"
}

echo "========================"
echo "| Building the project |"
echo "========================"
make

echo ""
```

```
echo "================="
echo "| Running Tests |"
echo "================="

unit_test '1;'                            1
unit_test '(2);'                          2
unit_test '1 + 2;'                        3
unit_test '7- 3;'                         4
unit_test '1*5;'                          5
unit_test '12 /2;'                        6
unit_test '1+2 *3;'                       7
unit_test $'4\n+\n4\n;'                   8
unit_test '((2 + 1)* 3);'                 9
unit_test '100 / (2*5);'                  10
unit_test '4@+4;'                         "FAIL"
unit_test '99 / 0;'                       "FAIL"
unit_test '00 + 1;'                       "FAIL"
unit_test '1 + 01;'                       "FAIL"
unit_test '99999999999999999999 + 1;'  "FAIL"

echo ""
echo "============================"
echo "PASSED: ""$num_passed"
echo "FAILED: ""$num_failed"
```

2.3 The Unit Test Output

```
========================
| Building the project |
========================
flex -o /home/t/Desktop/CS660/PA03/code/out/calc.lex.yy.c /home/t/Desktop/CS660/PA03/code/src/calc.lex
cc -o /home/t/Desktop/CS660/PA03/code/out/calc.lex.yy.o -c /home/t/Desktop/CS660/PA03/code/out/calc.lex
cc -o /home/t/Desktop/CS660/PA03/code/bin/calc /home/t/Desktop/CS660/PA03/code/out/calc.lex.yy.o /home/

=================
| Running Tests |
=================
~~~~~~~~~~~~~~~~~~~~~
CASE:
1;

expected: 1
got:      1

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~
CASE:
(2);

expected: 2
got:      2
```

```
RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
1 + 2;

expected: 3
got:      3

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
7- 3;

expected: 4
got:      4

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
1*5;

expected: 5
got:      5

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
12 /2;

expected: 6
got:      6

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
1+2 *3;

expected: 7
got:      7

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
4
+
4
;
```

```
expected: 8
got:      8

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
((2 + 1)* 3);

expected: 9
got:      9

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
100 / (2*5);

expected: 10
got:      10

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
4@+4;

expected: FAIL
got:      FAIL

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
99 / 0;

expected: FAIL
got:      FAIL

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
00 + 1;

expected: FAIL
got:      FAIL

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
1 + 01;
```

```
expected: FAIL
got:      FAIL

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~
CASE:
99999999999999999999 + 1;

expected: FAIL
got:      FAIL

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~

============================
PASSED: 15
FAILED: 0
```