Compiler Construction
PA04: A YACC Parser for a Simple Calculator

Terence Henriod

September 23, 2015

Abstract

This assignment requires you to write a bison-style parser for a simple four-function calculator. You will combine this parser with your scanner from the previous assignment to create a full four-function calculator.

1 The Assignment

I recommend you take the following steps to create your parser:

1. Decide which tokens you need (Hint: Consider the calc.tab.h file you were given for the previous assignment).
2. Consider the entire body of input to be a single nonterminal, which should be your start symbol.
3. This nonterminal should be produced from a list of expressions separated by semicolons.
4. Create a grammar for expressions and implement their rules. Your calculator should output the value of the expression once the semicolon is reached.
5. If a scanner error or parse error occurs, your calculator should indicate it and terminate.

Don't forget that you need to include a short main() function somewhere in your program, and that you must link your parser with your scanner object file and the libfl library.
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.

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"
#include "calc_utils.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("Bad number format.");
    return ERROR;
  }

  return INTEGER;
}

[;] {return SEMI;}

[(] {return OPEN;}

[)] {return CLOSE;}

[+] {return PLUS;}

[-] {return MINUS;}

[*] {return MULT;}

[/] {return DIV;}

{WHITESPACE} {/* ignore */;}

. {/* unrecognized characters are errors */
yyerror("Unrecognized token.");
    return ERROR;}

%%
 /* USER CODE */
```

2.2 calc.yacc

```
 /* DECLARATIONS */
%{
  #include <stdio.h>
  #include "calc_utils.h"
%}

%start line

%token INTEGER OPEN CLOSE PLUS MINUS MULT DIV SEMI ERROR

 // precedences
%left PLUS MINUS
%left MULT DIV

%%
 /* RULES */
line:
  /* empty */
  |
  line expression SEMI {
    printf("%d\n", $2);
  }
  ;

expression:
  expression PLUS term {
    int error = 0;
    int result = safe_add(&error, $1, $3);

    if (error != 0) {
      yyerror("Addition overflow.");
    } else {
      $$ = result;
    }
  }
  |
  expression MINUS term {
    $$ = $1 - $3;
  }
  |
  term {
    $$ = $1;
  }
  ;

term:
  term MULT factor {
    int error = 0;
    int result = safe_multiply(&error, $1, $3);

    if (error != 0) {
      yyerror("Multiplication overflow.");
    } else {
      $$ = result;
    }
  }
  |
  term DIV factor {
    if ($3 == 0) {
      yyerror("Division by zero is undefined.");
    } else {
      $$ = $1 / $3;
    }
  }
  |
  factor {
    $$ = $1;
  }
  ;

factor:
  OPEN expression CLOSE {
    $$ = $2;
  }
  |
  INTEGER {
    $$ = $1;
  }
  ;


%%
 /* USER CODE */
int main(int argc, char** argv) {
  return yyparse();
}

yyerror(char* s) {
  fprintf(stdout, "%s\n",s);
  exit(1);
}

yywrap() {
  // return 1 for funished with reading input
  return 1;
}
```

2.3 calc_utils.h

```
#ifndef CALC_UTILS_H
#define CALC_UTILS_H value

int
extract_int(int* error, char* yytext, const int yyleng);

int
safe_add(int* error, int x, int y);

int
safe_multiply(int* error, int x, int y);

#endif
```

2.4 calc_utils.c

```
#include "calc_utils.h"

#include <errno.h>
#include <limits.h>

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 || number_value > INT_MAX) {
    // 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.
    //
    // also, since a long can differ in size from an int, we check for that too
    *error = 1;
  }

  return number_value;
}

int safe_add(int* error, int x, int y) {
  int sum = 0;
  int bits_in_int = (sizeof(int) * 8) - 1;
  *error = 0;

  if (number_of_bits(x) < bits_in_int &&
      number_of_bits(y) < bits_in_int) {
    sum = x + y;
  } else {
    *error = 1;
  }

  return sum;
}

int safe_multiply(int* error, int x, int y) {
  int product = 0;
  int bits_in_int = (sizeof(int) * 8) - 1;
  *error = 0;

  if (number_of_bits(x) + number_of_bits(y) <= bits_in_int) {
    product = x * y;
  } else {
    *error = 1;
  }

  return product;
}


int
number_of_bits(int n) {
  int n_bits = 0;

  if (n < 0) {
    n *= -1;
  }

  while (n > 0) {
    n_bits += 1;
    n >>= 1;
  }

  return n_bits;
}
```

2.5 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)

echo "$result"

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

integer_regex='^[0-9]+$'

if [[ $result = $expected ]] ; then
message="passed"
((num_passed++))
elif [[ $result =~ integer_regex && $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 '1 + 01;'                     $'syntax error'
unit_test '00 + 1;'                     $'syntax error'
unit_test '7 + + 2;'                    $'syntax error'
unit_test '4@+4;'                       $'Unrecognized token.'
unit_test '9999999999999999999999 + 1;'  $'Bad number format.'
unit_test '2100000000 + 10000;'         $'Addition overflow.'
unit_test '200000 * 9000;'              $'Multiplication overflow.'


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

3 Test Results

The results according to my own unit testing:

```
========================
| Building the project |
========================
bison -d  -o /home/t/Desktop/CS660/PA04/code/out/calc.tab.c
/home/t/Desktop/CS660/PA04/code/src/calc.yacc
cc -o /home/t/Desktop/CS660/PA04/code/out/calc_utils.o -c
/home/t/Desktop/CS660/PA04/code/src/calc_utils.c
cc -o /home/t/Desktop/CS660/PA04/code/out/calc.tab.o -c
/home/t/Desktop/CS660/PA04/code/out/calc.tab.c
-I/home/t/Desktop/CS660/PA04/code/out -L/usr/local/lib
-I/home/t/Desktop/CS660/PA04/code/out -I/home/t/Desktop/CS660/PA04/code/src -lfl
/home/t/Desktop/CS660/PA04/code/src/calc.yacc: In function ‘yyerror’:
/home/t/Desktop/CS660/PA04/code/src/calc.yacc:90:3: warning: incompatible
implicit declaration of built-in function ‘exit’ [enabled by default]
```

```
   exit(1);
   ^
flex -o /home/t/Desktop/CS660/PA04/code/out/calc.lex.yy.c
/home/t/Desktop/CS660/PA04/code/src/calc.lex
cc -o /home/t/Desktop/CS660/PA04/code/out/calc.lex.yy.o -c
/home/t/Desktop/CS660/PA04/code/out/calc.lex.yy.c
-I/home/t/Desktop/CS660/PA04/code/out -I/home/t/Desktop/CS660/PA04/code/src -lfl
cc -o /home/t/Desktop/CS660/PA04/code/bin/calc
/home/t/Desktop/CS660/PA04/code/out/calc.lex.yy.o
/home/t/Desktop/CS660/PA04/code/out/calc.tab.o
/home/t/Desktop/CS660/PA04/code/out/calc_utils.o -L/usr/local/lib
-I/home/t/Desktop/CS660/PA04/code/out -I/home/t/Desktop/CS660/PA04/code/src -lfl

=================
| 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:
1 + 01;

expected: syntax error
got:      syntax error

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

expected: syntax error
got:      syntax error

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
CASE:
7 + + 2;

expected: syntax error
got:      syntax error

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

expected: Unrecognized token.
got:      Unrecognized token.

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
CASE:
99999999999999999999999 + 1;

expected: Bad number format.
got:      Bad number format.

RESULT: passed
~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
CASE:
2100000000 + 10000;

expected: Addition overflow.
got:      Addition overflow.

RESULT: passed
```

```
~~~~~~~~~~~~~~~~~~~~~~~
~~~~~~~~~~~~~~~~~~~~~~~
CASE:
200000 * 9000;

expected: Multiplication overflow.
got:      Multiplication overflow.

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

============================
PASSED: 17
FAILED: 0
```