Overview

All programs that take a set of programming specifications and translates them, ie. create a means to execute those specifications, are technically “compilers”, the term generally means a program that produces a separate executable from the compiler. A compiler that merely executes the original specifications is usually referred to as in “interpreter”[1]. Here we demonstrate the latter in the form of a calculator that performs simple arithmetic functions as well as handling division by zero.

[1] <https://en.wikipedia.org/wiki/Compiler>

Implementation

The method chosen here to demonstrate compiler technology is called “recursive descent parsing”. Recursive descent parsing is a kind of top-down parser built from a set of mutually exclusive procedures (or a non-recursive equivalent) where each such procedure usually implements one of the productions of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

This implementation also incorporates a “predictive” parsing technique rather than a “backtracking” technique. This simply means that the parser looks one character ahead instead rewinding the stream of read characters to where a different path may be taken. Predictive parsers is only possible for the class of grammars called LL(k), which are the “context-free grammars” for which there exists some positive integer k that allows a recursive descent parser to decide which production to use by examining only the next k tokens of input. [2]

[2] <https://en.wikipedia.org/wiki/Recursive_descent_parser>

Calculator Implementation and Division by Zero

Chosen for this project was an existing implementation[3] of a recursive descent parser with a language of that of a simple calculator. It handles arithmetic functions, but did not implement handling division by zero. The existing implementation was supplemented by handling division by zero where positive numerators divided by zero returned “Inf”; negative numerators divided by zero returned “-Inf” and numerators equal to zero divided by zero returned “Nan”[4].

[3] <http://www.dreamincode.net/forums/topic/234775-creating-a-recursive-descent-parser-oop-style/>

[4] <http://mathworld.wolfram.com/DivisionbyZero.html>

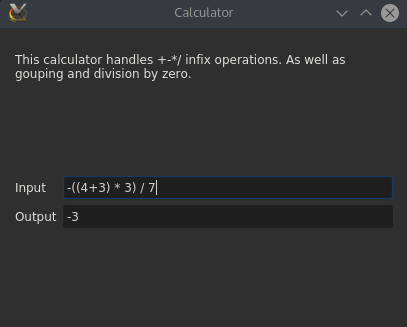
Calculator Grammar

A grammar is constructed by “productions”. Productions are the rules by which a parser generates the translation. Each value on the left hand side of the production (left of the '→' symbol) is defined by the terminals and non-terminals on the right-hand side. A terminal is a token that is atomic, where as a non-terminal is a composite of other non-terminals and terminals. Here, below is the grammar for our calculator example:

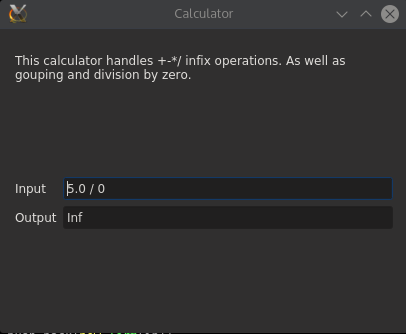
expr -> term ‘+’ term  
expr -> term ‘-‘ term  
exor -> term  
term -> unary ‘\*’ unary  
term -> unary ‘/’ unary  
term -> unary  
unary -> ‘-‘ factor  
unary -> ‘+’ factor  
unary -> factor  
factor -> '(' expr ')'  
factor -> number

Snapshots of the Calculator Compiler Application

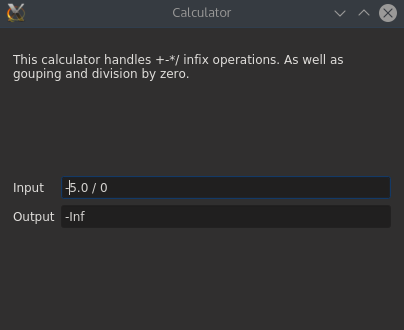
A normal arithmetic calculation:



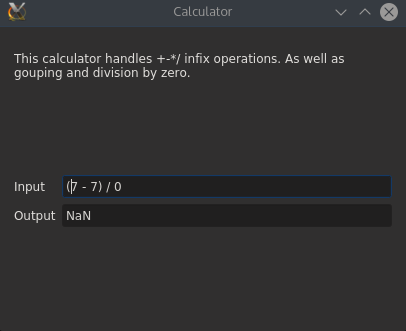
An example of handling division by zero where “Inf” is returned:



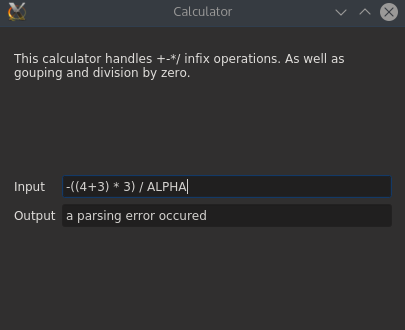
An example of handling division by zero where “-Inf” is returned:



An example of handling division by zero where “Nan” is returned:



An example of handling input that is outside of the defined language:



Source Code of the Calculator Compiler

parser.h

#ifndef PARSER\_H\_INCLUDED

#define PARSER\_H\_INCLUDED

#include <istream>

#include <deque>

#include <exception>

//pre-declare are classes so they can be used in a non-discriminate order

class Production;

class Expr;

class Term;

class Unary;

class Number;

class Factor;

class ParseError;

/\*\*

\* the base class for all of our productions

\*/

class Production {

public:

virtual ~*Production*(); //dose nothing, if we don't need to override it we don't want to

virtual double *getValue*()=0; //pure virtual, all children MUST implmnet this function

};

/\*\*

\* a simple class to handle parsing errors

\*/

class ParseError : public std::exception {

const char\* *what*() const throw();

};

/\*\*

\* a class to parse a number from the input stream

\*/

class Number : public Production {

private:

double value;

public:

Number(std::istream& in);

///Override

double *getValue*();

};

/\*\*

\* a class to parse a factor from the input stream

\*/

class Factor : public Production {

private:

Production\* expr; //we need to use a Production pointer in order handle the dual behavoir of factors

public:

Factor(std::istream& in);

///Override

~*Factor*();

///Override

double *getValue*();

};

/\*\*

\* a class to parse unary operators and operands from an input stream

\*/

class Unary : public Production {

private:

int sign;

Factor\* value;

public:

Unary(std::istream& in);

///Override

~*Unary*();

///Override

double *getValue*();

};

/\*\*

\* a class to parse binary operators(\* and /) and operands from the input stream

\*/

class Term : public Production {

private:

std::deque<Unary\*> values;

std::deque<char> ops;

public:

Term(std::istream& in);

///Override

~*Term*();

///Override

double *getValue*();

};

/\*\*

\* a class to parse binary operators(+ and -) and operands from the input stream, acts as the full expreison parser becuase + and - have the lowest precdence.

\* this code works just like Term only it uses the + and - operators instead.

\*/

class Expr : public Production {

private:

std::deque<Term\*> values;

std::deque<char> ops;

public:

Expr(std::istream& in);

///Override

~*Expr*();

///Override

double *getValue*();

};

#endif // PARSER\_H\_INCLUDED

parser.cpp

#include <istream>

#include <deque>

#include <locale>

#include "parser.h"

/\*\*

\* static functions

\*/

//protptype the functions so compiler wont complain

static void ignoreSpace(std::istream& in);

static char getChar(std::istream& in);

/\*\*

\* $1 - an input stream

\* pre - none

\* post - all spaces up to the next non space char are removed

\*/

static void ignoreSpace(std::istream& in) {

while(isspace(in.peek())) {

in.get();

}

}

/\*\*

\* $1 - an input stream

\* pre - none

\* post - all preceding and following space is removed. 1 non-space char is removed

\*/

static char getChar(std::istream& in) {

ignoreSpace(in); //remove all preceding space

char ret = in.get();

ignoreSpace(in); //remove all following space

return ret;

}

/\*\*

\* Production

\*/

//dose nothing, if we don't need to override it we don't want to

Production::~*Production*() {}

///ParseError

const char\* ParseError::*what*() const throw() {

return "a parsing error occured";

}

/\*\*

\* Number

\*/

Number::Number(std::istream& in) {

ignoreSpace(in); //remove all preceding space

in>>value;

if(!in) {

throw ParseError();

return;

}

ignoreSpace(in); //remove all following space

}

///Override

double Number::*getValue*() {

return value;

}

/\*\*

\* Factor

\*/

Factor::Factor(std::istream& in) {

ignoreSpace(in); //remove all preceding space

if(in.peek() == '(') { //check to see if a paren was used

in.get();

expr = new Expr(in);

ignoreSpace(in); //remove all following space

if(in.peek() != ')') { //make sure the paren was matched

throw ParseError();

} else {

in.get();

}

} else { //if there is no paren then we just have a number

expr = new Number(in);

}

}

///Override

Factor::~*Factor*() {

delete expr;

}

///Override

double Factor::*getValue*() {

return expr->*getValue*();

}

/\*\*

\* Unary

\*/

Unary::Unary(std::istream& in) {

sign = 1; //initlize sign to 1

ignoreSpace(in); //get rid of any preceding space

while(in.peek() == '-' || in.peek() == '+') { //while we have an operator to parse

if(getChar(in) == '-') { //if the operator actully has an effect

sign = -sign;

}

}

value = new Factor(in); //parse the factor following the unary operators

}

///Override

Unary::~*Unary*() {

delete value;

}

///Override

double Unary::*getValue*() {

return sign \* value->*getValue*();

}

/\*\*

\* Term

\*/

Term::Term(std::istream& in) {

values.push\_back(new Unary(in)); //construct the first value

ignoreSpace(in); //ignore preceding space

while(in.peek() == '\*' || in.peek() == '/') {

ops.push\_back(getChar(in)); //push back the operator

values.push\_back(new Unary(in)); //push back the left operand

}

}

///Override

Term::~*Term*() {

for(unsigned int i=0;i<values.size();++i) {

delete values[i];

}

}

///Override

double Term::*getValue*() {

double ret = values[0]->getValue(); //get the first value

for(unsigned int i=1;i<values.size();++i) { //loop though the rest of the values

if(ops[i-1] == '\*') { //check to see which operator it is and preform the acoridng action

ret \*= values[i]->getValue();

} else {

if ((ret > 0) && (values[i]->getValue() == 0))

ret = 987654321.1;

else if ((ret < 0) && (values[i]->getValue() == 0))

ret = 987654322.1;

else if (( ret == 0) && (values[i]->getValue() == 0))

ret = 987654323.1;

else

ret /= values[i]->getValue();

}

}

return ret;

}

/\*\*

\* Expr

\*/

Expr::Expr(std::istream& in) {

ignoreSpace(in);

values.push\_back(new Term(in));

while(in.peek() == '+' || in.peek() == '-') {

ops.push\_back(getChar(in));

values.push\_back(new Term(in));

}

}

///Override

Expr::~*Expr*() {

for(unsigned int i=0;i<values.size();++i) {

delete values[i];

}

}

///Override

double Expr::*getValue*() {

double ret = values[0]->getValue();

for(unsigned int i=1;i<values.size();++i) {

if(ops[i-1] == '+') {

ret += values[i]->getValue();

} else {

ret -= values[i]->getValue();

}

}

return ret;

}