Programming in C/C++
Exercises set six: parsers II

Christiaan Steenkist
Jaime Betancor Valado
Remco Bos

March 8, 2017

Exercise 37, substantial grammar extension

All these operators.

Code listings

Scanner for both 37 and 38

Listing 1: lexer

```
%filenames scanner
%interactive

digits      [0-9]+
optdigits   [0-9]*
exp         [eE][-+]?{digits}

%%

[ \t]+

{digits}                        |
{digits}"."{optdigits}{exp}?    |
{optdigits}"."{digits}{exp}?    return Parser::NUMBER;

[[:alpha:]_][[:alnum:]_]*       return Parser::IDENT;
```

```

"+="              return Parser::COMPADD;
"-="              return Parser::COMPSUB;
"*="              return Parser::COMPMUL;
"/="              return Parser::COMPDIV;
"<<="             return Parser::COMPLSH;
">>="             return Parser::COMPRSH;
"<<"                            return Parser::LSH;
">>"                            return Parser::RSH;

\n|.                            return matched()[0];
```

Parser

Listing 2: grammar

```
%class-name Parser

%filenames parser
%parsefun-source parse.cc

%baseclass-preinclude rulevalue.h
%stype RuleValue

%scanner ../scanner/scanner.h

%token  NUMBER
        IDENT

%right  '=' COMPADD COMPSUB COMPMUL
        COMPDIV COMPLSH COMPRSH
%left   '+' '-'
%left   '*' '/'
%right  '%' LSH RSH
%right  uMinus

// %debug

%%

lines:
    lines line
|
    line
;


line:
    expr '\n'
    {
        display($1);
    }
|
    error '\n'
    {
        prompt();
    }
|
    '\n'
    {
        prompt();
    }
;


expr:
    NUMBER
    {
        $$ = value();
    }
|
    IDENT
    {
        $$ = variable();
    }
|
    '-' expr            %prec uMinus
    {
        $$ = negate($2);
    }
|
  expr '*' expr
  {
    $$ = mul($1, $3);
  }
|
  expr '/' expr
  {
    $$ = div($1, $3);
  }
|
    expr '+' expr
    {
        $$ = add($1, $3);
    }
|
    expr '-' expr
    {
        $$ = sub($1, $3);

    }
|
    '(' expr ')'
    {
        $$ = $2;
    }
|
    expr '=' expr
    {
        $$ = assign($1, $3);
    }
|
  expr COMPADD expr
  {
    $$ = assign($1, add($1, $3));
  }
|
  expr COMPSUB expr
  {
    $$ = assign($1, sub($1, $3));
  }
```

```
|
  expr COMPMUL expr
  {
    $$ = assign($1, mul($1, $3));
  }
|
  expr COMPDIV expr
  {
    $$ = assign($1, div($1, $3));
  }
|
  expr COMPLSH expr
  {
    $$ = assign($1, lsh($1, toInt($3)));
  }
|
  expr COMPRSH expr
  {
    $$ = assign($1, rsh($1, toInt($3)));
  }
|
  expr '%' expr
  {
    $$ = mod($1, toInt($3));
  }
|
  expr LSH expr
  {
    $$ = lsh($1, toInt($3));
  }
|
  expr RSH expr
  {
    $$ = lsh($1, toInt($3));
  }
;
```

Listing 3: parser.h

```
...
    int toInt(RuleValue const &rv);
```

```
...
```

Listing 4: toint.cc

```
#include "parser.ih"
#include <cmath>

int Parser::toInt(RuleValue const &rv)
{
    return std::round(valueOf(rv));
}
```

Exercise 38, calculator

We were tasked with making a calculator that behaves differently than a user would expect.

Parser

Listing 5: grammar

```
%class-name Parser

%filenames parser
%parsefun-source parse.cc

%baseclass-preinclude rulevalue.h
%stype RuleValue

%scanner ../scanner/scanner.h

%token  NUMBER
        IDENT

%right  '=' COMPADD COMPSUB COMPDIV
        COMPMUL COMPLSH COMPRSH
%right  uMinus
%left   '+'
%right  '*'
%left   '/'
%right  '%' LSH RSH
```

```
%right  '-'

// %debug

%%

lines:
    lines line
|
    line
;


line:
    expr '\n'
    {
        display($1);
    }
|
    error '\n'
    {
        prompt();
    }
|
    '\n'
    {
        quit();
    }
;


expr:
    NUMBER
    {
        $$ = value();
    }
|
    IDENT
    {
        $$ = variable();
    }
|
    expr '-' expr
    {
$$ = add($1, $3);
    }
|
    expr '*' expr
    {
      $$ = div($1, $3);
    }
|
    expr '/' expr
    {
      $$ = sub($1, $3);
    }
|
    expr '+' expr
    {
    $$ = mul($1, $3);
    }
|
    '-' expr %prec uMinus
    {
        $$ = negate($2);
    }
|
    '(' expr ')'
    {
        //
    }
|
    expr '=' expr
    {
        $$ = assign($1, $3);
    }
|
    expr COMPADD expr
    {
      $$ = assign($1, add($1, $3));
    }
|
    expr COMPSUB expr
    {
      $$ = assign($1, sub($1, $3));
    }
|
    expr COMPMUL expr
    {
      $$ = assign($1, mul($1, $3));
    }
|
    expr COMPDIV expr
    {
      $$ = assign($1, div($1, $3));
    }
|
    expr COMPLSH expr
    {
      $$ = assign($1, lsh($1, toInt($3)));
    }
|
    expr COMPRSH expr
    {
      $$ = assign($1, rsh($1, toInt($3)));
    }
|
    expr '%' expr
    {
      $$ = mod($1, toInt($3));
    }
|
    expr LSH expr
    {
      $$ = lsh($1, toInt($3));
    }
|
    expr RSH expr
    {
      $$ = lsh($1, toInt($3));
    }
;
```

Listing 6: parser.h

```
// Generated by Bisonc++ V4.09.02 on Thu, 19 Mar 2015 19:21:46 +0100

#ifndef Parser_h_included
#define Parser_h_included

// $insert baseclass
#include "parserbase.h"
// $insert scanner.h
#include "../scanner/scanner.h"

#include <vector>
#include <unordered_map>
#include <string>

#undef Parser
class Parser: public ParserBase
{
    // $insert scannerobject
    Scanner d_scanner;

    std::vector<double> d_value;
    std::unordered_map<std::string, unsigned> d_symtab;

    bool d_display;

    public:
        Parser(bool run);
        int parse();

    private:
        void error(char const *msg);    // called on (syntax) errors
        int lex();                      // returns the next token from the
```

```
                                        // lexical scanner.
        void print();                   // use, e.g., d_token, d_loc

    // support functions for parse():
        void executeAction(int ruleNr);
        void errorRecovery();
        int lookup(bool recovery);
        void nextToken();
        void print__();
        void exceptionHandler__(std::exception const &exc);

    // added functions for the calculator:

        void display(RuleValue &e);
        void done();
        void prompt();

        RuleValue &add(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &assign(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &div(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &lsh(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &mul(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &negate(RuleValue &e);
        RuleValue &rsh(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue &sub(RuleValue &lvalue, RuleValue &rvalue);
        RuleValue value();
        RuleValue variable();

        double valueOf(RuleValue const &e);
        int toInt(RuleValue rv);
```

```

  void quit();
};


#endif
```

Exercise 39, functions

This was actually made before 36-38.

Code listings

Listing 7: grammar

```
%class-name Parser

%filenames parser
%parsefun-source parse.cc

%baseclass-preinclude rulevalue.h
%stype RuleValue

%scanner ../scanner/scanner.h

%token NUMBER IDENT

%right 'e' ln sin asin sqrt deg grad rad
%left '^'

// %debug

%%

lines:
    lines line
|
    line
;


```

```
line:
    expr '\n'
    {
        display($1);
    }
|
    error '\n'
    {
        prompt();
    }
|
    '\n'
    {
        prompt();
    }
;

expr:
    NUMBER
    {
        $$ = value();
    }
|
    IDENT
    {
        $$ = variable();
    }
|
  'e' '^' expr
  {
    $$ = exp($3);
  }
|
  ln expr
  {
    $$ = ln($2);
  }
|
  sin expr
  {
    $$ = sin($2);
  }
|
  asin expr
  {
    $$ = asin($2);
  }
|
  sqrt expr
  {
    $$ = sqrt($2);
  }
|
  '|' expr '|'
  {
    $$ = abs($2);
  }
|
  deg expr
  {
    $$ = deg($2);
  }
|
  rad expr
  {
    $$ = rad($2);
  }
|
  grad expr
  {
    $$ = grad($2);
  }
;
```

parser.h snippet

```
  // arithmetic functions:

    void display(double &value);
    void done();
    void prompt();
    RuleValue &exp(RuleValue &value);
    RuleValue &ln(RuleValue &value);
    RuleValue &sin(RuleValue &value);
    RuleValue &asin(RuleValue &value);
    RuleValue &sqrt(RuleValue &value);
    RuleValue &abs(RuleValue &value);

    RuleValue &deg(RuleValue &value);
    RuleValue &grad(RuleValue &value);
    RuleValue &rad(RuleValue &deg);
    RuleValue &rad(RuleValue &grad);

    double const pi = 3.14159;
    double const e = 2.71828;
```

Implementations

Listing 8: abs.cc

```
#include "parser.ih"

RuleValue &Parser::abs(RuleValue &value)
{
  return RuleValue(abs(valueOf(value)));
}
```

Listing 9: asin.cc

```
#include "parser.ih"

RuleValue &Parser::asin(RuleValue &value)
{
  if (valueOf(value) <= 1 || valueOf(value) >= -1)
    return RuleValue(asin(valueOf(value)));
  else
    error("Value (radians) out of interval -1 < value
   < 1");
}
```

Listing 10: deg.cc

```
#include "parser.ih"
```

```

RuleValue &Parser::deg(RuleValue &value)
{
  return RuleValue(2 * Parser::pi * valueOf(value) /
   360);
}
```

Listing 11: done.cc

```
#include "parser.ih"

void Parser::done()
{
    cout << "Bye\n";
    ACCEPT();
}
```

Listing 12: exp.cc

```
#include "parser.ih"

RuleValue &Parser::exp(RuleValue &value)
{
  return RuleValue(Parser::e ^ valueOf(value));
}
```

Listing 13: grad.cc

```
#include "parser.ih"

RuleValue &Parser::grad(RuleValue &value)
{
  return RuleValue(2 * Parser::pi * valueOf(value) /
   400);
}
```

Listing 14: ln.cc

```
#include "parser.ih"

RuleValue &Parser::log(RuleValue &value)
{
  if (valueOf(value) >= 0)
```

```
    return RuleValue(log(valueOf(value)));
  else
    error("Value may not be negative");
}
```

Listing 15: raddeg.cc

```
#include "parser.ih"

RuleValue &Parser::rad(RuleValue &deg)
{
  return RuleValue((360 * deg) / (2 * Parser::pi));
}
```

Listing 16: radgrad.cc

```
#include "parser.ih"

RuleValue &Parser::rad(RuleValue &grad)
{
  return RuleValue((400 * grad) / (2 * pi));
}
```

Listing 17: sin.cc

```
#include "parser.ih"

RuleValue &Parser::sin(RuleValue &value)
{
  return RuleValue(sin(valueOf(value)));
}
```

Listing 18: sqrt.cc

```
#include "parser.ih"

RuleValue &Parser::sqrt(RuleValue &value)
{
  if (valueOf(value) >= 0)
    return RuleValue(sqrt(valueOf(value)));
  else
    error("Value may not be negative");
}
```

Exercise 40, polymorphic value type class

We attempted to make a polymorphic value type class.

Code listings

Scanner

Listing 19: lexer.ll

```
NUM   [0-9]

%%

{NUM}+           return ParserBase::Tokens__::INT;
{NUM}*"."{NUM}+  return ParserBase::Tokens__::DOUBLE;
QUIT             return ParserBase::Tokens__::QUIT;
[\t ]            // ignore
\n               return matched()[0];
.+               return ParserBase::Tokens__::STRING;
```

Parser

Listing 20: grammar.gr

```
%token INT STRING DOUBLE QUIT

%polymorphic INT: int; STRING: std::string; DOUBLE:
    double;

%scanner Scanner.h

%type <INT> intline
%type <STRING> stringline
%type <DOUBLE> doubleline

%%

lines:
  lines '\n' line
|
  line
```

```
;

line:
  intline
  {
    $$ = $1;
  }
|
  stringline
  {
    $$ = $1;
  }
|
  doubleline
  {
    $$ = $1;
  }
|
  QUIT
  {
    quit();
  }
;

intline:
  INT
  {
    $$ = getInt();
    showInt($$);
  }
;

stringline:
  STRING
  {
    $$ = getString();
    showString($$);
  }
;

```

```
doubleline:
  DOUBLE
  {
    $$ = getDouble();
    showDouble($$);
  }
;
```

Listing 21: Parser.ih

```
...
#include <cstdlib>
...
```

Listing 22: Parser.h

```
...
  // my own functions:
    int getInt();
    std::string getString();
    double getDouble();

    void showInt(int &someInt);
    void showString(std::string &someString);
    void showDouble(double &someDouble);
    void quit();
...
```

Listing 23: getdouble.cc

```
#include "Parser.ih"

double Parser::getDouble()
{
  return atof(d_scanner.matched().c_str());
}
```

Listing 24: getint.cc

```
#include "Parser.ih"

int Parser::getInt()
{
  return atol(d_scanner.matched().c_str());
}
```

Listing 25: getstring.cc

```
#include "Parser.ih"

string Parser::getString()
{
  return d_scanner.matched();
}
```

Listing 26: quit.cc

```
#include "Parser.ih"

void Parser::quit()
{
  ACCEPT();
}
```

Listing 27: showdouble.cc

```
#include "Parser.ih"

void Parser::showDouble(double &someDouble)
{
  cout << someDouble << '\n';
}
```

Listing 28: showint.cc

```
#include "Parser.ih"

void Parser::showInt(int &someInt)
{
  cout << someInt << '\n';
}
```

Listing 29: showstring.cc

```
#include "Parser.ih"

```

```cpp
void Parser::showString(string &someString)
{
  cout << someString << '\n';
}
```