12

C++ Exercises
Set 7

Author(s): Channa Dias Perera, Alex Swaters, Ivan Yanakiev

19:57

March 28, 2023

49

In this exercise, we define a simple grammar that accepts a sequence of zero or more function calls of write, which takes it one or more arguments that are variables / numbers.

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 1: Scanner/lexer

```
ident                   [_[:alpha:]][_[:alnum:]]*
integral                [[:digit:]]+
real                    [[:digit:]]*\.{integral}
number                  {real}|{integral}
%%
[[:space:]]+            // ignore whitespace
write                   return Tokens::WRITE;
{ident}                 return Tokens::IDENT;
{number}                return Tokens::NUMBER;
.                       return matched()[0];
```

Listing 2: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 10:12:26 +0200

// $insert class_h
#include "tokens.h"

#include "Scanner.h"
```

Listing 3: Parser/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h
%token WRITE IDENT NUMBER
%%
                        // Zero or more calls to write
opt_write_calls:
  opt_write_calls write_call
|
  // empty
;
                        // A call to write
write_call:
  WRITE '(' args_list ')'
;

                        // One or more args in a list, with seperator
args_list:
  args_list ',' arg
```

```
|
  arg
;

arg:
  IDENT
|
  NUMBER
;
```

50

In this exercise, we define a grammar for a calculator that does: parentheses, operations, negation, multiplication, division, addition, subtraction, in that order of precedence.

The parser only defines tokens of things that can be returned by the parser (per the exercise definition).

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 4: Scanner/lexer

```
operation           "$"
integral            [[:digit:]]+
real                [[:digit:]]*\.{integral}
number              {real}|{integral}
%%
[[:space:]]+        // ignore whitespace
{number}            return Tokens::NUM;
{operation}         return Tokens::OP;
.                   return matched()[0];
```

Listing 5: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 10:39:23 +0200

// $insert class_h
#include "tokens.h"

#include "Scanner.h"
```

Listing 6: Parser/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token NUM

                        // supported operations
%left '-' '+'
%left '*' '/'

                        // NEG uses '-' but we already define '-' prec.
                        // Therefore we use NEG to define its precedence
                        // with %prec later.
%right NEG OP

%%
exp:
  NUM
|
  exp '+' exp
|
  exp '-' exp
|
  exp '*' exp
|
```

```
  exp '/' exp
|
  '-' exp %prec NEG
|
  OP exp
|
  '(' exp ')'
;
```

51

In this exercise, we discuss the grammar ambiguities introduced by the following grammar:

```
%token NUMBER
%%
expr:
  NUMBER
|
  number
|
  expr '+' expr
|
  expr '-' expr
;

number:
  NUMBER
;
```

We correct the grammar (grammar1) to remove the conflicts with all the priorities of the operators being the same.

We also correct the grammar (grammar2) file according to how bisonc++ would implicitly fix them (i.e: making it right associative), and removing a specific rule to resolve R/R conflicts.

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 7: Parser/grammar1

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token NUMBER
                        // Added left associativity to '+' and '-' to remove
                        // S/R conflict. On same line therefore same prio.
%left '+' '-'
%%
                        // Removed expr: NUMBER; rule, as this was R/R
                        // conflict
expr:
  number
|
  expr '+' expr
|
  expr '-' expr
;

number:
  NUMBER
;
```

Listing 8: Parser/grammar2

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h
```

```
                        // Added right associativity to '+' and '-' to remove
                        // S/R conflict as bisonc++ would resolve the conf.
                        // On same line therefore same prio.
%right '+' '-'
%token NUMBER
%%
                        // Removed number: NUMBER; rule to remove R/R conflict
                        // Also how bisonc++ would do it.

expr:
  NUMBER
|
  expr '+' expr
|
  expr '-' expr
;
```

Listing 9: Scanner/lexer

```
integral            [[:digit:]]+
real                [[:digit:]]*\.{integral}
number              {real}|{integral}
%%
[[:space:]]+        // ignore whitespace
{number}            return Tokens::NUMBER;
.                   return matched()[0];
```

Listing 10: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 11:13:17 +0200

// $insert class_h
#include "tokens.h"
#include "Scanner.h"
```

There are two issues:

1) There is a reduce/reduce conflict, i.e: two or more rules can apply to the same sentence. In this case, to match a NUMBER token, either the number rule or the expr rule can be used (the latter through a reduction to NUMBER).

   In grammar1, we fix this by removing the unnecessary expr -> NUMBER; rule.

   bisonc++ resolves the conflict by selecting the first rule that applies through the grammar (which is the expr rule for NUMBER in this case), so for grammar2, we remove the number -> NUMBER rule.

2) The other error is the shift/reduce conflict, due to the rules involving '+' / '-'. This means that if the parser reads expr op expr op, where op is either '+' or '-', and with the second op just being read, "expr op expr" is in the exact and can be _reduced_. Alternatively, the latter op can be _shifted_ into the stack as well. Hence, the conflict.

To fix this, we can define associativity rules. We have fixed grammar1 for standard mathematical reading (i.e: left associativity).

However, bisonc++ prefers to shift, meaning that the operators are right associative, (as the stack gets reduced top down, using the rightmost operators first). Thus, we specify right associativity in grammar2.

52

In this exercise, we extend the following grammar:

```
%token NR
%left '+'
%left '*'
```

```
%right '-'

%%

expr:
  NR
|
  '-' expr
|
  expr '+' expr
|
  expr '*' expr
;
```

to also support inequality checks (!=) and exponentiation (^). The former has less priority than addition (+) and the latter has higher priority than multiplication (*) but less than unary negation (-).

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 11: Scanner/lexer

```
integral            [[:digit:]]+
real                [[:digit:]]*\.{integral}
number              {real}|{integral}
%%
[[:space:]]+        // ignore whitespace
{number}            return Tokens::NR;
"!="                return Tokens::INEQ;
.                   return matched()[0];
```

Listing 12: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 11:39:24 +0200

// $insert class_h
#include "tokens.h"
#include "Scanner.h"
```

Listing 13: Parser/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token NR
                                // Supported operations, INEQ tokenizes the !=
                                // string
%right INEQ
%left '+'
%left '*'
%right '^'
                                // No need for NEG / UMINUS etc., since there is
                                // no subtraction. Can define precedence on '-'
                                // immediately.
%right '-'

%%

expr:
  NR
|
  expr INEQ expr
|
  '-' expr
|
  expr '+' expr
```

```
|
  expr '*' expr
|
  expr '^' expr
;
```

53

We extend the following grammar:

```
%token VAR
%token NR
%left '+'
%left '*'
%right '-'

%%

expr:
  NR
|
  VAR
|
  '-' expr
|
  expr '+' expr
|
  expr '*' expr
;
```

according to the grammar: it is.
When in doubt: ask on the mailing list

to support indexing operations. It is not yet know whether constructions like 3[4] can be considered a valid indexable expression, thus we design our grammar to take a choice to disallow this into account (i.e: make it easy to modify for this choice).

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 14: Scanner/lexer

```
ident               [_[:alpha:]][_[:alnum:]]*
integral            [[:digit:]]+
real                [[:digit:]]*\.{integral}
number              {real}|{integral}
%%
[[:space:]]+        // ignore whitespace
{ident}             return Tokens::VAR;
{number}            return Tokens::NR;
.                   return matched()[0];
```

Listing 15: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 11:54:47 +0200

// $insert class_h
#include "tokens.h"
#include "Scanner.h"
```

Listing 16: Parser/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token VAR
%token NR
```

```
%left '+'
%left '*'
%right '-'

%%
expr:
  NR
|
  VAR
|
  '-' expr
|
  expr '+' expr
|
  expr '*' expr
|


                                    // An expression can now be something
                                    // that is indexed
  indexedExpr
;

indexedExpr:
  indexable indexingList
;
                          // We can modify to remove indexable: NR; in the
                          // future if we wish to disallow things like 3[4].
indexable:
  VAR
|
  NR
;
                          // One or more indexings, to be applied to an
                          // indexable.
indexingList:
  oneIndex
|
  indexingList oneIndex
;
                          // A single indexing
oneIndex:
  '[' expr ']'
;
```

this is getting TC: as a hint: you don't need the indexedExpr rule

7

54

In this exercise, we construct a grammar to accept either:
- A list of WORD tokens that are separated by spaces
- A list of WORD tokens that are separated by commas

However, we do not accept lists that separate by both spaces and commas.

grammaropt contains a grammar for lists that can have 0 or more words
grammarreq contains a grammar for lists that can have 1 or more words.

Note: We only hand in Scanner.ih from the generated files from flexc++, as that is the only file we modify from what it generates. We do not submit any generated files from bisonc++ since we do not modify any of files it generates.

Listing 17: Scanner/lexer

```
%%
[[:alpha:]]+          return Tokens::WORD;
(.|\n)                // Ignore rest
```

forgot to return ;) ?

Listing 18: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Mon, 27 Mar 2023 12:02:23 +0200

// $insert class_h
```

```
#include "tokens.h"
#include "Scanner.h"
```

Listing 19: Parser/grammaropt

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token WORD
%%
list:
                                // A list of no words
|
                                // A list of 1 word
  WORD
|
                                // 2 WORDs that are separated by a space
                                // Which means our subsequent list of 0 or more
                                // WORDs has to be space separated
  WORD WORD plain
|
                                // 2 WORDs that are separated by a comma
                                // Which means our subsequent list of 0 or more
                                // WORDs has to comma separated
  WORD ',' WORD separated
;

                                // A space separated list of 0 or more WORDs
plain:
  plain
  WORD
|
  // empty
;

                                // A comma separated list of 0 or more WORDs
separated:
  separated
  ','
  WORD
|
  // empty
;
```

avoid such complex openings. assuming the context of the exercise use the optional building blocks for input and for a single word input. Then use rules to handle either blank or comma delimited lists.

Listing 20: Parser/grammarreq

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token WORD
%%
list:
                                // A list of 1 word

  WORD
|
                                // 2 WORDs that are separated by a space
                                // Which means our subsequent list of 0 or more
                                // WORDs has to be space separated
  WORD WORD plain
|
                                // 2 WORDs that are separated by a comma
                                // Which means our subsequent list of 0 or more
                                // WORDs has to comma separated
  WORD ',' WORD separated
;
                                // A space separated list of 0 or more WORDs
plain:
  plain
  WORD
|
  // empty
```

```
;

                                    // A comma separated list of 0 or more WORDs
separated:
  separated
  ','
  WORD
|
  // empty
;
```

55

In this exercise, we design an interactive parser for a program that calculates z-scores.

Parser in Parser1/Parser.h provides a prompt via the rule prompt.
Parser in Parser2/Parser.h provides a prompt via a slight modification to Parser::lex (in Parser2/Parser.ih).

Note: For Parser1, we do not submit the Parser.h or Parser.ih files since we do not make any modifications to it after bisonc++ generates them. For both parsers, we do not include the Scanner/tokens.h file it generates it, for the same reason. For Scanner, we only submit Scanner.ih since that is the only file we modify from those that flexc++ generates.

Listing 21: Scanner/lexer

```
%interactive
integral            [[:digit:]]+
double1             -?[[:digit:]]+\.?[[:digit:]]*
double2             -?\.[[:digit:]]+
double              {double1}|{double2}

%%
{integral}                          return Tokens::INTEGRAL;
{double}                            return Tokens::DOUBLE;
([[:space:]]{-}[\n])+               // Ignore irrelevant whitespace
(.|\n)                              return matched()[0];
```

Listing 22: Scanner/Scanner.ih

```
// Generated by Flexc++ V2.11.00 on Tue, 28 Mar 2023 17:45:13 +0200

// $insert class_h
#include "tokens.h"
#include "Scanner.h"
```

Listing 23: Parser1/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token DOUBLE INTEGRAL

%%
                                    // Display the prompt, and then take in commands
start: prompt commands;

                                    // A one or more list of commands
commands:
  commands command
|
  command
;

command:
                                    // No more commands
```

```
  '\n'
  {
    ACCEPT();
  }
|
  'm' number nextPrompt
|
  's' number nextPrompt
|
  score nextPrompt
;

score:
  number
|
  number INTEGRAL
;

number:
  DOUBLE
|
  INTEGRAL
;

prompt:
  {
    std::cout << "? ";
  }
;

nextPrompt:
  '\n' prompt
;
```

these repetitions should be avoided. define a rule handling the commands ending in a \n and then call the prompt.

Listing 24: Parser2/grammar

```
%scanner ../Scanner/Scanner.h
%token-path ../Scanner/tokens.h

%token DOUBLE INTEGRAL EMPTY

%%

commands:
  commands command
|
  command
;

command:
  EMPTY
  {
    ACCEPT();
  }
|
  'm' number '\n'
|
  's' number '\n'
|
  score '\n'
;

score:
  DOUBLE
|
  DOUBLE INTEGRAL
;

number:
  DOUBLE
|
  INTEGRAL
```

Accords to the exercise: only one cout << ? should be used

```
;
```

Listing 25: Parser2/Parser.h

```
// Generated by Bisonc++ V6.04.03 on Tue, 28 Mar 2023 18:20:53 +0200

#ifndef Parser_h_included
#define Parser_h_included

// $insert baseclass
#include "Parserbase.h"
// $insert scanner.h
#include "../Scanner/Scanner.h"


class Parser: public ParserBase
{
    // $insert scannerobject
    Scanner d_scanner;
    bool d_prompt = true;
    size_t d_lastToken = '\n';

    public:
        Parser() = default;
        int parse();

    private:
        void error();                       // called on (syntax) errors
        int lex();                          // returns the next token from the
                                            // lexical scanner.
        void print();                       // use, e.g., d_token, d_loc
        void exceptionHandler(std::exception const &exc);

    // support functions for parse():
        void executeAction_(int ruleNr);
        void errorRecovery_();
        void nextCycle_();
        void nextToken_();
        void print_();
};


#endif
```

Listing 26: Parser2/Parser.ih

```
// Generated by Bisonc++ V6.04.03 on Tue, 28 Mar 2023 17:49:18 +0200

    // Include this file in the sources of the class Parser.

// $insert class.h
#include "Parser.h"


inline void Parser::error()
{
    std::cerr << "Syntax error\n";
}

// $insert lex
inline int Parser::lex()
{
                        // If we are ready to display a prompt, display it
                        // before we get the command
    if (d_prompt)
    {
        std::cout << "? ";
        d_prompt = false;
    }
```

(see the previous page)

```
    size_t token;
    token = d_scanner.lex();
    if (token == '\n')
    {
                                // Two newlines in a row -> should exit
        if (d_lastToken == '\n')
            return Tokens_::EMPTY;
        else
                                // Otherwise, note that we should disp. prompt
                                // next
          d_prompt = true;
    }

    d_lastToken = token;

    return token;
}

inline void Parser::print()
{
}

inline void Parser::exceptionHandler(std::exception const &exc)
{
    throw;                  // re-implement to handle exceptions thrown by actions
}


    // Add here includes that are only required for the compilation
    // of Parser's sources.



    // UN-comment the next using-declaration if you want to use
    // int Parser's sources symbols from the namespace std without
    // specifying std::

//using namespace std;
```

Listing 27: main.ih

```
                    // Only have one of the two lines uncommented, to choose
                    // either parser.
//#include "Parser1/Parser.h"
#include "Parser2/Parser.h"
#include <iostream>

using namespace std;
```

Listing 28: main.cc

```
#include "main.ih"

int main()
{
  Parser parser;
  parser.parse();
}
```