Marco Seman

CIS 400 Programming Languages

Dr. Jie Shen

October 9, 2018

Assignment #1

# Question 1

Compute the weakest precondition for each of the following sequences of assignment statements and their postconditions:

1. a := 2.0 \* b + 2;

b := a \* 2 + 2;

{b<0}

We need to substitute 0 for b and find the postcondition of the second function to find the weakest precondition of a. we then go to the first function and substitute -2 for a then divide to leave b by itself.

a\*2 < -2

{a < -1} // Weakest precondition

2\*b+2 < -1

2\*b < -3

{b > -3/2}

-3/2 < b < 0

1. a := 2 \* (2 \* b + 1);

b := 2 \* a + 1

{b > 5}

We need to substitute 5 for b and find the postcondition of the first, then substitute b for a.

2\*a+1 < 5

2\*a < 4

{a < 2 } weakest precondition

2\*(2\*b+1)< 2

4b+2< 2

4b < 0

b > 0

# Question 2

Given a grammar

<assign> 🡪 <id> = <expr>

<id> 🡪 A | B | C

<expr> 🡪 <id> + <expr>

| <id> \* <expr>

| ( <expr> )

| <id>

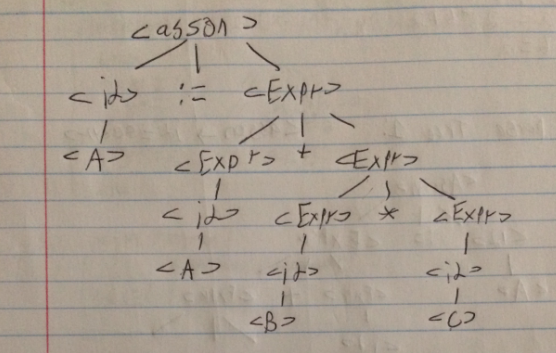
Show two different parse trees can be generated on the following statement:

A = A + B \* C

Is the grammar ambiguous ? If yes, how can you fix it ? Show the parse tree

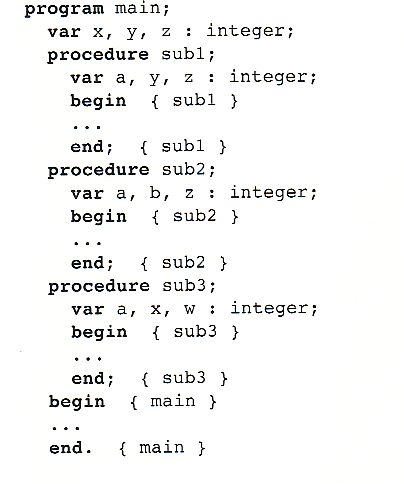
after the fix.

No the grammar is not ambiguous



# Question 3

Consider the following program:

Given the following calling sequences and assuming that dynamic scoping is used, what variables are visible during execution of the last subprogram activated? Include with each visible variable the name of the unit where it is declared.

1. **main** calls **sub1**; **sub1** calls **sub2**; **sub2** calls **sub3**.

Sub3: a, w, x

Sub2: b, z

Sub1: y

# Question 4

Do an Internet research and write a one-page summary about Flex and Bison, respectively.

Flex

Flex is a fast-lexical analyzer generator software tool where it is intended to quickly produce scanners as part of a compiler front end where it recognizes patterns in a set of texts. The program was originally written in C-programming Language by Vern Paxson, a Professor of Computer Science at the University of California, Berkley. The purpose for the use of Flex is for a program generator to produce a source code that recognizes regular expressions in an input to reveal the pattern specifications.

The major factor in Flex is its quick scanner capabilities where the scanner reads a sequence of characters or values to recognize textual units called “tokens”. Tokens are are typically keywords, identifiers, numbers, single characters, character strings, or EOLN and EOF(logical end-of-time and end-of-input markers). The first process of the scanner is to read a stream of character to generate such lexical tokens. The software also filters out what is called nontokens, which is anything that includes blank spaces (spaces, tabs, underlines) or comments. Also, Special characters such as “+” “- ““\*” will be translated by the software where it will return them as textual characters “PLUS” “MINUS” “TIMES”.

For example, if a value int A = B \* (C+D) + 5;, the analyzer will break it down as followed:

Type “int”

Variable “A”­­­

Equals

Variable “B”

Times

Left bracket

Variable “C”

Plus

Variable “D”

Right bracket

Plus

Literal “5”

The lexical analyzer takes the stream of inputted characters and reads them out from left to right to produce a set of tokens as an output.

The scanner feature in flex plays a big role in Flex due to it’s ability to recognize regular expressions when there is a pattern on an input. A regular expression is a set of strings that are used in a parser generator. The regular expressions include metacharacters which is used as an identifier for parts that match specific provided specifications in Flex. Metacharacters include alternating expressions such as a|b which denotes to {“a”, “b”} or it can repeat zeros many time with “\*” expression such as a|b\* denotes to {lamda, “a”, “b”, “bb”}.

Flex automatically generates a yylex() function where the parser is anticipated to obtain tokens from the token stream. The yylex() function also runs a rule extension for an .l file, which is the input file for Flex’s source file. The source file is first generated in lex language, where it is then taken to the lex compiler to be transformed from lex.i to C program which gets renamed to lex.yy.c. The lex.yy.c is then taken to the C compiler to compile a file that is executable called a.out. A stream of characters inputted are then taken from the output file (a.out) to generate a stream of token.

Bison

Bison is a parser generator with an input file with a .y extension that reads specifications of a context tree language. The parser is intended to warn the user about any ambiguous parsing that are in the input by reading sequences of tokens to determine whether they are syntactically consistent. The parser is included in action with each grammar rule where it takes the value of the previous action and returns new values. The Bison parser also uses a bottom-up parse tree approach that contains tokens and values that are spread from the leaves up to the root.

For context sensitive information, Bison uses semantic analysis to analyze the parse tree so the output of the semantic is annotated. The attributes in the grammar are used to describe the semantics of the program. The sematic phase is usually combined with the parser, and while the parser is running, information such as variables and other objects are being stored inside a symbol table. The information in the symbol table are used to perform context sensitive checking.

When transforming annotations in a parse tree to structure a tree, Bison uses an optimizer that applies semantics to facilitate the generation of the code more efficiently. Bison also uses a code generator to transform annotations from the parse tree into an object-oriented language that denotes semantics of the source code. There is also a peep-hole optimizer that inspects the program’s code and attempts to make machine dependent improvements to the code.

Bison also includes syntactical analysis where the lexical analyzer decides the rules of the grammar and determines whether the inputs are valid. If a local variable was undeclared in a program, the compiler will say that the value is an undeclared identifier. If a local variable was declared and not used, the compiler will say that it the variable has been declared but not initialized, which can potentially break your code if they were ignored. The importance of the syntactical analysis is to make sure that the structure of the input text is valid and consistent. The machine code is at its final stages after passing the stages of lexical and syntactical analysis, where the program can then be generated into machine code. The speed of which the program compiles is dependent on the quality.

When the parser is written, there is generally no output, however, the parse tree is implicitly constructed during the parse. As the parse is executed, it internally builds the structure of the program. The internal structure is based from the right-hand side of the production rule and when it is recognized, it is reduced to the left-hand side. The parsing isn’t complete until the whole program has been reduced from the start symbol of the grammar. A feature called y.tab.h is generated in Bison when the user specifies the -d option. The y.tab.h includes definitions of all the tokens that are in the Bison input, then the file is combined into the flex scanner. The y.tab.h defines the C functions that is in the yyparse() which is the parser.

A symbol table is a module that holes information that is required by the attributed grammar. The symbol table contains information about the attributes in the programming language and it scopes the information from each variable. In the bison file, the symbol table is developed as a module. When Bison reads tokens, it groups the tokens together using grammar rules. If the grammar entered is valid, then the result of the token sequence is reduced to a single grouping where the symbols are in the grammar’s start symbol.

<http://www.admb-project.org/tools/flex/compiler.pdf>

# Question 5

Consider the following EBNF grammar for a “Calculator Language”:

< calculation> 🡪 <expression> =

<expression> 🡪 <term> [ (- | /) <expression> ]

<term> 🡪 <value> [ (+ | \*) <term> ]

<value> 🡪 [ <sign> ] <unsigned> [ . <unsigned> ]

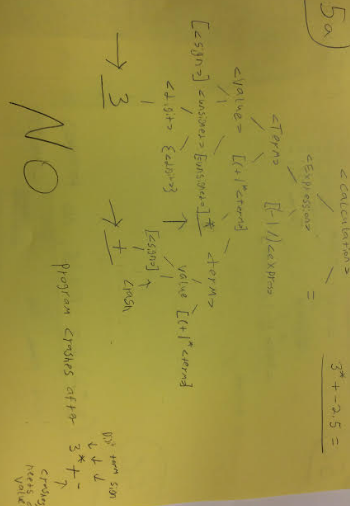
<unsigned> 🡪 <digit> { <digit> }

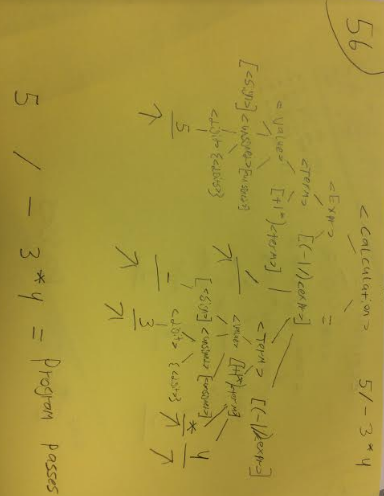
<digit> 🡪 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

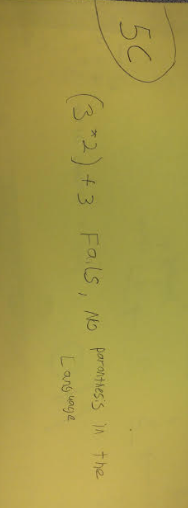
<sign> 🡪 + | -

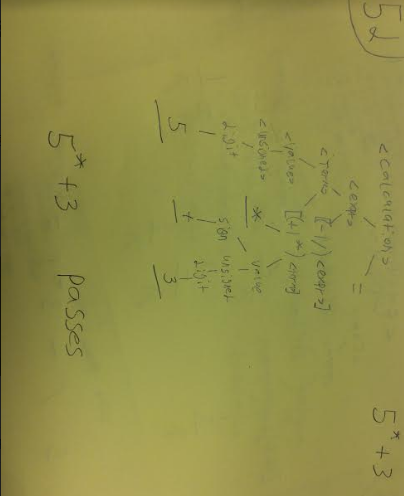
which of the following sentences are in the language generated by this grammar ?

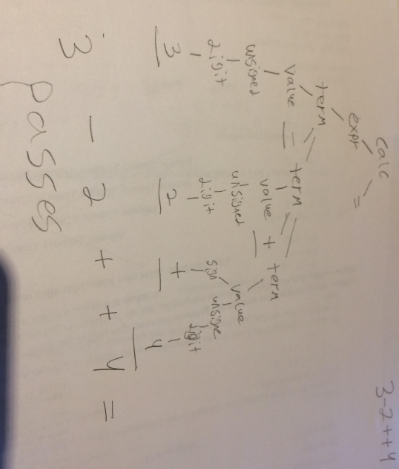
1. 3 \* + - 2.5 = No
2. 5 / - 3 \* 4 = Yes
3. (3 \* 2 ) + 3 = No
4. 5 \*+3 = Yes
5. 3-2++4= Yes











# Question 6

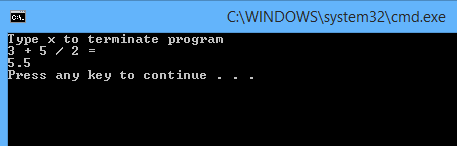
Write a pseudo-interpreter of the “Calculator Language” defined in Question 5 in C or C++ language. The evaluation of the user-input expression begins once a “=” character is read. Until then, the characters are buffered in a user\_input string variable. The only characters that are accepted are the terminals of the calculator language: (0 1 2 3 4 5 6 7 8 9 + - \* / . =).

The interpreter can be implemented by a while loop and terminates when a character ‘x’ is read. Your interpreter is required not only to identify if an input string belongs to the language defined by the grammar in Q5, but also to evaluate the input string and print out the evaluation result on computer screen.

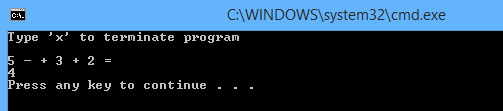
The interpreter works by using the EBNF grammar recursively to resolve the input string into its component parts. There should be a single procedure for each non-terminal in the grammar. In the case of a successful parsing, your code should print out the evaluation result of an input expression. In the event of a failed parsing, an error message is output to the screen.

You should try the following test cases and report the result of each case:

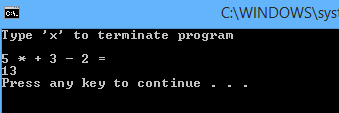
1. 3 + 5 / 2 =



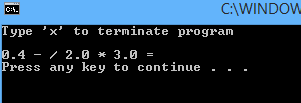
1. 5 – +3 + 2 =



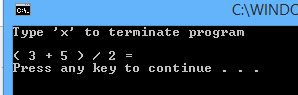
1. 5 \*+3 – 2 =



1. 0.4 - / 2.0 \* 3.0 =



1. (3 + 5) / 2 =



Source Code:

// CalculatorLanguage.cpp : Defines the entry point for the console application.

// Marco Seman

// CIS 400: Programming Language

// Professor Jie Shen

// October 9, 2018

#include "stdafx.h"

#include <iostream>

#include <string>

#include <math.h>

#include <algorithm>

#include <array>

using namespace std;

// mathmatical values for the instances

float Result(float Val1, char op1, float Val2, char op2, float Val3)

{

if (op1 == '+' && op2 == '+')

return Val1 + Val2 + Val3;

if (op1 == '+' && op2 == '-')

return Val1 + Val2 - Val3;

if (op1 == '+' && op2 == '\*')

return Val1 + Val2 \* Val3;

if (op1 == '+' && op2 == '/')

return Val1 + Val2 / Val3;

if (op1 == '-' && op2 == '+')

return Val1 - Val2 + Val3;

if (op1 == '-' && op2 == '-')

return Val1 - Val2 - Val3;

if (op1 == '-' && op2 == '\*')

return Val1 - Val2 \* Val3;

if (op1 == '-' && op2 == '/')

return Val1 - Val2 / Val3;

if (op1 == '\*' && op2 == '+')

return Val1 \* Val2 + Val3;

if (op1 == '\*' && op2 == '-')

return Val1 \* Val2 - Val3;

if (op1 == '\*' && op2 == '\*')

return Val1 \* Val2 \* Val3;

if (op1 == '\*' && op2 == '/')

return Val1 \* Val2 / Val3;

if (op1 == '/' && op2 == '+')

return Val1 / Val2 + Val3;

if (op1 == '/' && op2 == '-')

return Val1 / Val2 - Val3;

if (op1 == '/' && op2 == '\*')

return Val1 / Val2 \* Val3;

if (op1 == '/' && op2 == '/')

return Val1 / Val2 / Val3;

return 0;

}

int main()

{

float value1, value2, value3;

cout << "Type 'x' to terminate program" << endl << endl;

bool quit = false;

// terminates until user enters x

while (quit == false)

{

string expr1;

cin >> expr1;

char op1;

// first value entered by user

if (expr1 == "x")

{

quit = true;

break;

}

else

{

// convert string to float

value1 = ::atof(expr1.c\_str());

cin >> op1;

}

string expr2;

cin >> expr2;

if (expr2 == "+")

{

cin >> value2;

}

else if (expr2 == "-")

{

cin >> value2;

value2 = value2\*(-1);

}

else

{

value2 = ::atof(expr2.c\_str());

}

string expr3;

char op2;

cin >> op2;

cin >> expr3;

if (expr3 == "+")

{

cin >> value3;

}

else if (expr3 == "-")

{

cin >> value3;

value3 = value3\*(-1);

}

else

{

value3 = ::atof(expr3.c\_str());

}

char equal;

cin >> equal;

if (equal == '=')

{

float result = Result(value1, op1, value2, op2, value3);

cout << result << endl;;

}

}

}

/\*

float sign(char function[])

{

if (function[0] == '+')

{

// function.erase(function.begin());

}

else if (function[0] == '-')

{

// function.erase(function.begin());

}

}

float digit(char function[])

{

float value = function[0];

return value;

}

float unsign(char function[])

{

float value1 = 0;

{

int level = 0; // every iteration of level represents the next decimal place

while (isdigit(function[0]))

return value1;

}

float value(char function[])

{

float value1;

float value2 = 1;

if (!isdigit(function[0]))

{

value2 = sign(function);

}

value1 = unsign(function) \* value2;

return value1;

}

float term(char function[])

{

float value1 = value(function);

switch (function[0])

{

case '+':

{

value1 = value1 + term(function);

break;

}

case '\*':

{

value1 = value1 \* term(function);

break;

}

}

return value1;

}

float expression(char function[])

{

float value1 = value(function);

switch (function[0])

{

case '-':

{

value1 = value1 - term(function);

break;

}

case '/':

{

value1 = value1 / term(function);

break;

}

}

return value1;

}

float calculation(char function[])

{

float value1 = expression(function);

return value1;

}

\*/

Further Explanation

1. I/O

You should provide Screen output

2. Test Cases

Illegal expression

Integer expression w/o unary (+/-)

Float type expression w/o unary (+/-)

3. No operator precedence. The evaluation result may contradict with mathematical

calculation.

4. While-loop evaluation with ‘x’ to terminate

5. Source code and a simple documentation.

6. Scan the input string from left to right and use leftmost derivation

7. Each recursion function will return the result of a part of expression. At the top level,

you need to print out the result of the input expression after being evaluated.

8. You may use the following routines to convert a string to an integer or a float type

variable:

#include <math.h>

#include <stdlib.h>

int i = atoi(“-1”); float r = atof(“-1.3”)