Test 2

Programming Language Concepts

April 20, 2020

Name: Harry Park

1. (15 points) Write a program in C++ or C that includes two different enumeration types and has a significant number of operations using the enumeration types. Also write the same program using only integer variables.

   Compare the readability and predict the reliability differences between the two programs.

Github link: https://github.com/parkhwyy/gsu-plc/tree/master/Test2/test2_q1

Both programs give the same output. Enum is a value type data type which is used to declare a list of named integer constants and can be defined using the enum keyword inside a class or structure. The enum is used to give a name to each constant so that the constant integer can be called with its given name. Using names improves readability and reliability of code by making things more obvious.

2. (10 points) Perl allows both static and a kind of dynamic scoping. Write a Perl program that uses both and clearly shows the difference in effect of the two. Explain clearly the difference between the dynamic scoping described in this chapter and that implemented in Perl.

   Online Perl Compiler:

   `https://www.tutorialspoint.com/execute_perl_online.php`

Github link: https://github.com/parkhwyy/gsu-plc/tree/master/Test2/test2_q2.pl

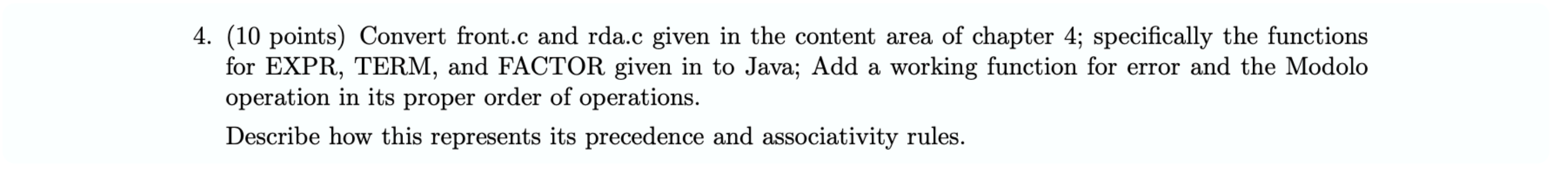
The first execution printing the value of x from the parent program using a subroutine demonstrates static scoping. The following execution demonstrates dynamic scoping, where the value of x is associated with the most recent environment. As described in the chapter, dynamic scoping is based on the calling sequence of subprograms, not on their spatial relationship to each other. Perl's dynamic scoping is unusual - Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines dynamically scoped local variable.

3. (10 points) Write three functions in C/C++: one that declares a large array statically, one that declares the same large array on the stack, and one that creates the same large array from the heap. Call each of the subprograms a large number of times (at least 100,000) and output the time required by each. Explain the results.

   Explain why or why not you cant do this in Java, what are the implication of this? If you can't specify which type of array can you not declare.

Github link: https://github.com/parkhwyy/gsu-plc/tree/master/Test2/test2_q3.c

It does not take long for static_fun to be called by the main function and executed a number of times. The array a was bound to memory cells before program execution began and remained bounded until the program terminated. There is not a significant difference between calling/executing stack_fun and static_fun. Compared to these, heap_fun took much longer time to be called and executed. The heap is a disorganized collection of storage cells, and a variable that is allocated fro the heap can only be referenced through a pointer. Managing the heap and obtaining references to variables from the heap is costly and complicated. This is why creating arrays of the same size from the heap required a significant amount of time in comparison to declaring a static array or an army on the stack. In Java, we cannot do this since all array memory allocations are from heap memory, and we cannot mention an array explicitly from stack..



4. (10 points) Convert front.c and rda.c given in the content area of chapter 4; specifically the functions for EXPR, TERM, and FACTOR given in to Java; Add a working function for error and the Modolo operation in its proper order of operations.

   Describe how this represents its precedence and associativity rules.

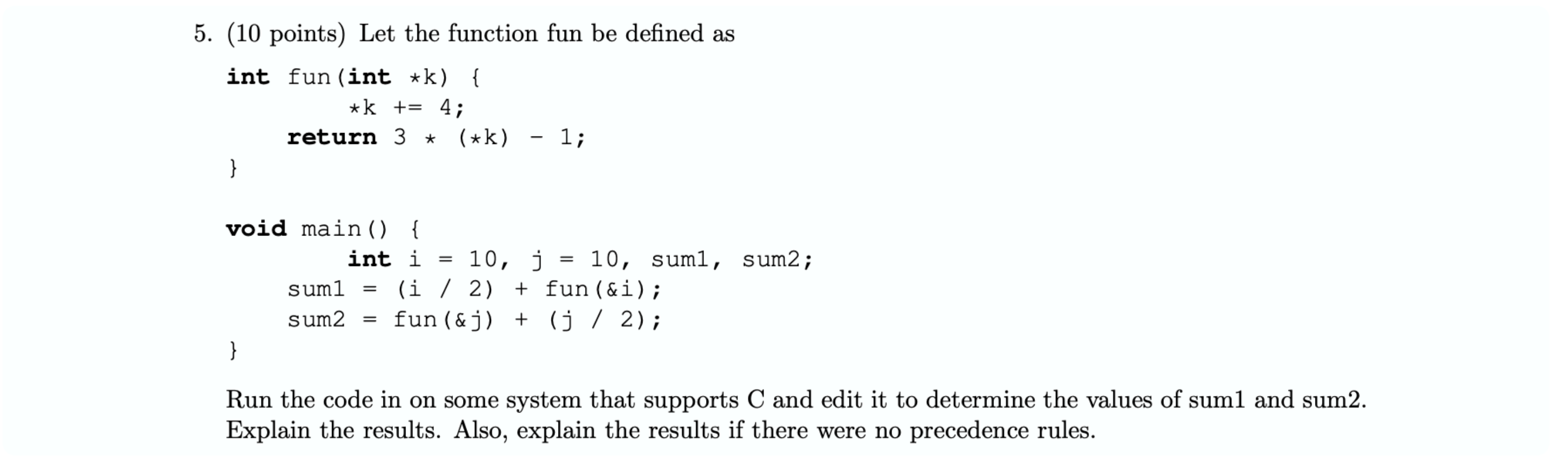
Github link: https://github.com/parkhwyy/gsu-plc/tree/master/Test2/test2_q4

<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | / | %) <factor>}
<factor> → id | int_constant | (<expr>)

In rda.java, the lexical analyzer gets the next lexeme and puts its token code in the global variable nextToken.
For each terminal symbol in the RHS, that terminal symbol is compared with nextToken. If they do not match, it is a syntax error.
If they match, the lexical analyzer is called to get the next input token. For each nonterminal, the parsing subprogram for that nonterminal is called.

Recursive-descent parsing subprograms are written with the convention that each one leaves the next token of input in nextToken.
So, whenever a parsing function begins, it assumes that nextToken has the code for the leftmost token of the input that has not yet been used in the parsing process (left-right associativity rules).

The part of the language that the expr() function parses consists of one or more terms, separated by either plus or minus operators. This is the language generated by the nonterminal <expr>. Therefore, first it calls the function that parses terms. Then it continues to call that function as long as it finds ADD_OP or SUB_OP tokens (which it passes over by calling lex) (precedence rules).


5. (10 points) Let the function fun be defined as

```
int fun(int *k) {
        *k += 4;
    return 3 * (*k) - 1;
}

void main() {
        int i = 10, j = 10, sum1, sum2;
    sum1 = (i / 2) + fun(&i);
    sum2 = fun(&j) + (j / 2);
}
```

Run the code in on some system that supports C and edit it to determine the values of sum1 and sum2. Explain the results. Also, explain the results if there were no precedence rules.

Github link: https://github.com/parkhwyy/gsu-plc/tree/master/Test2/test2_q5.c

Precedence:

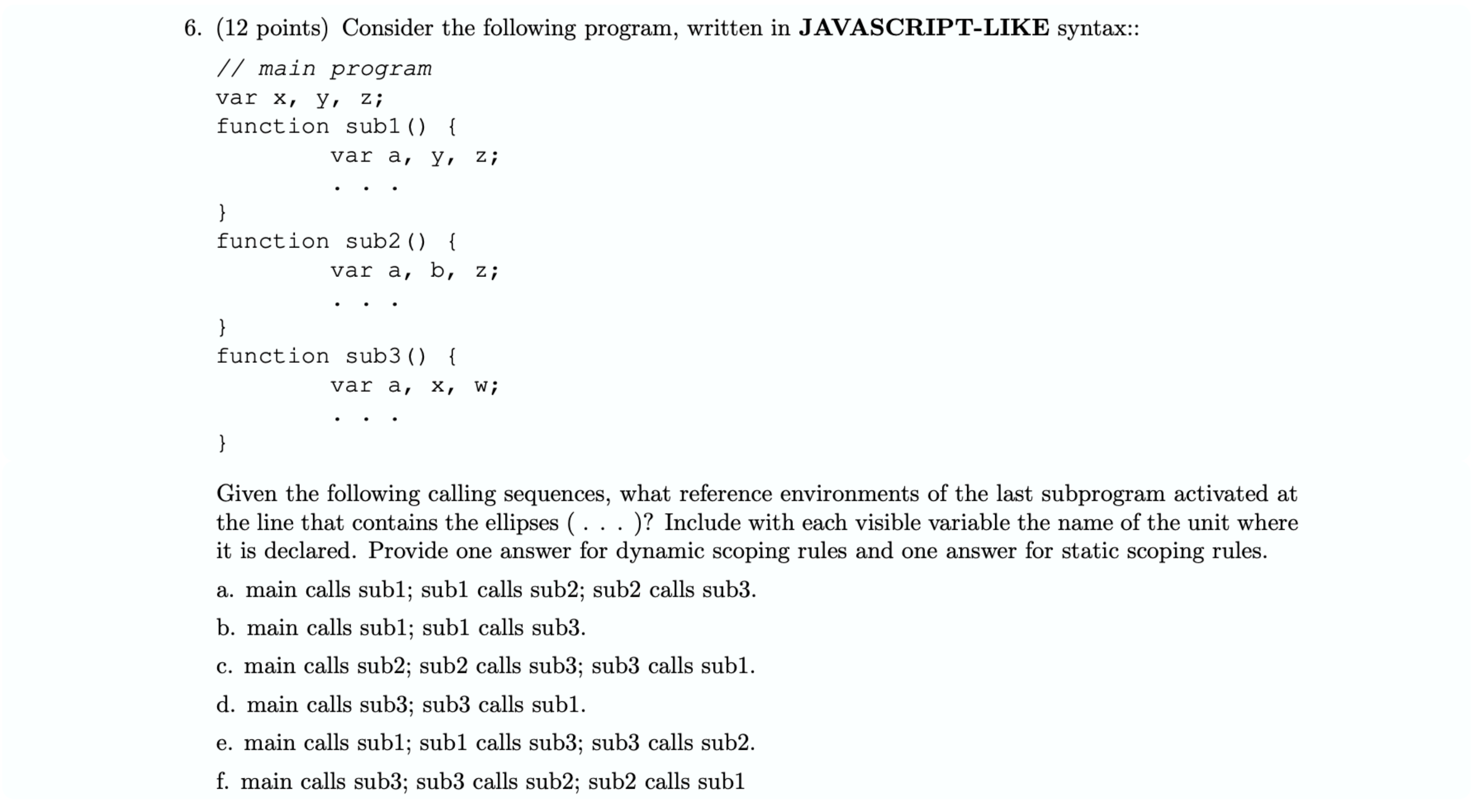
sum1 = (10 / 2) + (3 * 14 - 1) = 46

sum2 = (3 * 14 - 1) + (14 / 2) = 48

No precedence:

Sum1 = (10 / 2) + (3 * 14 - 1) = 46

Sum2 = ((3 * 14 - 1) + 14) / 2 = 27


6. (12 points) Consider the following program, written in **JAVASCRIPT-LIKE** syntax::

```
// main program
var x, y, z;
function sub1() {
        var a, y, z;
        . . .
}
function sub2() {
        var a, b, z;
        . . .
}
function sub3() {
        var a, x, w;
        . . .
}
```

Given the following calling sequences, what reference environments of the last subprogram activated at the line that contains the ellipses (. . .)? Include with each visible variable the name of the unit where it is declared. Provide one answer for dynamic scoping rules and one answer for static scoping rules.

a. main calls sub1; sub1 calls sub2; sub2 calls sub3.

b. main calls sub1; sub1 calls sub3.

c. main calls sub2; sub2 calls sub3; sub3 calls sub1.

d. main calls sub3; sub3 calls sub1.

e. main calls sub1; sub1 calls sub3; sub3 calls sub2.

f. main calls sub3; sub3 calls sub2; sub2 calls sub1

Dynamic scoping:

a. a, x, w (sub3) / b, z (sub2) / y (sub1)
b. a, x, w (sub3) / y, z (sub1)
c. a, y, z (sub1) / x, w (sub3) / b (sub2)
d. a, y, z (sub1) / x, w (sub3)
e. a, b, z (sub2) / x, w (sub3) / y (sub1)
f. a, y, z (sub1) / b (sub2) / x, w (sub3)

Static scoping:

a. a, x, w (sub3) / y, z (main) / b (sub2)
b. a, x, w (sub3) / y, z (main)
c. a, y, z (sub1) / x (main) / b (sub2) / w (sub3)
d. a, y, z (sub1) / x (main) / w (sub3)
e. a, b, z (sub2) / x, y (main) / w (sub3)
f. a, y, z (sub1) / x (main) / b (sub2) / w (sub3)

7. (8 points) Evaluate $a > b > c$ in terms of mathematics (logical inequalities. Evaluate the same expression in terms of a C based language. Do the two mean the same thing? What does each expression say? If the two are the same show how they are the same or different.

Mathematics: a is greater than b AND b is greater than c.

C based language:
C does not have a boolean type and uses int instead. When the program first evaluates "a > b," it returns 0 (false) or 1 (true) as an integer. Then, it compares the returned value (0 or 1) with c. It does not process the same as the statement in terms of mathematics.

8. (15 points) Assume the following rules of associativity and precedence for expressions:

```
Precedence
Highest
    - (unary), (prefix) ++, (prefix) --
    (postfix) ++, (postfix) --
    +, *, &
    -, /, %, not
    < , <=, >=, >, !=
    =, +=, *=, /=
    and
    or, xor
Lowest
    Associativity
    Left to right
```

Show the order of evaluation of the following expressions by parenthesizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression a * b + c / d the order of evaluation would be represented as

```
((a * b)^1 + c)2   / d)3
```

Also rewrite the expression where there are no precedence rules and the statement is given right to left associativity, For example for the same problem above

```
d / c + b * a
```

If you feel the statement can't be rewritten without parenthesis explain why?

```
a. a * b - 1 + c
b. ++a * (b - 1) / c % d
c. (a - b) / c & (d * e / a - 3)
d. -a or c = d and e
e. a > b xor c or d <= 17
```

a. ((a * b)^1 - (1 + c)^2)^3

b. (((++a)^1 * (b - 1)^2)^3 / c)^4 % d)^5

c. ((a - b)^1 / (c & ((((d * e)^2 / a)^3 - 3)^4))^5)^6

d. ((-a)^1 or ((c = d)^2 and e)^3)^4

e. (((a > b)^1 xor c)^3 or (d <= 17)^2)^4

Assuming we have to get the same result as when we follow the precedence rules above, regardless of the order of operations on the left:

a. (c + 1) - b * a
Parentheses are needed to calculate and keep the value of (c + 1) first and do the - operation afterwards.

b. d % c / ++a * 1 - b

c. (b - a) / c & 3 - a / e * d
Parentheses are needed to calculate and keep the valued of (b - a) first and do the / operation afterwards.

d. -a or e and d = c

e. (17 <= d) or c xor b > a
Parentheses are needed to calculate and keep the value of (17 <= d) first and do the or operation afterwards.

9. (10 points) Write a BNF description of the precedence and associativity rules defined for the expressions in problem above.

```
<exp> -> <exp> or <or_exp>
        | <exp> xor <or_exp>
        | <or_exp>

<or_exp> -> <or_exp> and <and_exp>
           | <and_exp>

<and_exp> -> <and_exp> = <eq_exp>
            | <and_exp> += <eq_exp>
            | <and_exp> *= <eq_exp>
            | <and_exp> /= <eq_exp>
            | <eq_exp>

<eq_exp> -> <eq_exp> < <ineq_exp>
           | <eq_exp> <= <ineq_exp>
           | <eq_exp> >= <ineq_exp>
           | <eq_exp> > <ineq_exp>
           | <eq_exp> != <ineq_exp>
           | <ineq_exp>

<ineq_exp> -> <ineq_exp> - <term>
             | <ineq_exp> / <term>
             | <ineq_exp> % <term>
             | not <term>
             | <term>

<term> -> <term> + <factor>
         | <term> * <factor>
         | <term> & <factor>
         | <factor>

<factor> -> <postfix> ++
          | <postfix> --
          | <postfix>

<postfix> -> - <prefix>
           | ++ <prefix>
           | -- <prefix>
           | <prefix>

<prefix> -> ( <exp> ) | <operand>
<operand> -> a | b | c | d | e
```