Homework 1

Tushar Jain, N12753339
CSCI-GA.2110.001 Spring 2018 - Programming Languages

March 10, 2018

Note: All the supporting files can be found here: https://github.com/tshrjn/nyu-pl/hw1

1 Regular Expressions

Problem 1.a. Strings consisting of any number of lower case letters, upper case letters, and digits, such that the occurrences of upper case letters and digits are alternating. That is, there cannot be two upper case letters in the string without a digit somewhere in between them and there cannot be two digits in the string without an upper case letter somewhere between them. For example, aAbc3dEf4ghIjk5m would be an example of such a string. However, aAbc3dEfGhi4jk would not be a valid string, since there is no digit between E and G.

Solution.
$([a-z]*[A-Z][a-z]*[0-9])*(\epsilon|[a-b]*[A-Z][a-b]*)|([a-z]*[0-9][a-z]*[A-Z])*(\epsilon|[a-b]*[0-9][a-b]*)$
([A-Z][a-z]*[0-9][a-z]*[A-Z])*—([0-9][a-z]*[A-Z][a-z]*[0-9])*—[a-z]* □

Problem 1.b. Positive and negative floating point literals that specify a positive or negative exponent, such as the following: 243.876E11 (representing 243.867×10^{11}) and -135.24E-4 (representing 135.24×10^{4}). There must be at least one digit before the decimal point and one digit after the decimal point (before the E).

Solution.
$(-|\epsilon)[0-9][0-9]*.[0-9][0-9]*E(-|\epsilon)[0-9][0-9]*$

□

Problem 1.c. Variable names that (1) must start with a letter, (2) may contain any number of letters, digits, and (underscores), and (3) may not contain two consecutive underscores.

Solution.
$[a-zA-z]([0-9a-zA-Z]*\_[0-9a-zA-Z])*(\epsilon|_)$

□

Problem 2.a. Provide a simple context-free grammar for the language in which the following program is written. You can assume that the syntax of names and numbers are already defined using regular expressions (i.e. you dont have to define the syntax for names and numbers). In this language, a program consists of a sequence of function definitions.

Problem 2.b. Draw the parse tree for the above program, based on a derivation using your grammar.

Solution:
Please see figure attached in the zip for this question.

□

Problem 3.a. Define the terms static scoping and dynamic scoping.

Solution:
Static Scoping: In Static scoping, the structure of the program source code determines what variables you are referring to. Therefore the scope of the bindings can be determined at the compile time. That is , if a variable is not defined in the current scope, keep looking at the outer scopes till the time you don't get it.

Dynamic scoping: In Dynamic scoping, the runtime state of the program stack determines what variable you are referring to. Therefore the scope of the bindings can be determined at the run time. That is, if a function references a non-local name, the call chain is traversed backwards to find the first functions that define the names. □

Problem 3b. Give a simple example, in any language you like (actual or imaginary), that would illustrate the difference between static and dynamic scoping. That is, write a short piece of code whose result would be different depending on whether static or dynamic scoping was used.

Solution:

Python Code:

```
def main():
    x = 5
    def foo():
        print(x) # In static scoping prints 5, dynamic prints 7

    def bar():
        x = 7
        foo()

    foo()

if __name__ == '__main__':
    main()
```

□

Problem 3.c. Why do all modern programming languages use static scoping? That is, what is the advantage of static scoping over dynamic scoping?

Solution:
This is simply because in static scoping its easy to reason about and understand just by looking at code and thus, easier to debug. We can see what variables are in the scope just by looking at the text in the editor.

Also, it's hard to work with dynamic scoping. With lexical scope, a name always refers to its local lexical environment. With dynamic scope, each identifier has a global stack of bindings. Introducing a local variable with name x pushes a binding onto the global x stack (which may have been empty), which is popped off when the control flow leaves the scope. Evaluating x in any context always yields the top binding. □

Problem 4.a. Draw the call stack for the following program that would exist when the print statement is executed. Assume the language is statically scoped. Show at least the local variables and parameters (including any closure), the static and dynamic links, and identify which stack frame is for which procedure.

Solution: See attached fig. in file 4a.pdf. □

Problem 4.b. What would the program print?

Solution: The program would print $20, 17$.
z will be 20 and x will be 17 □

Problem 4.c. Suppose, instead, that the language was dynamically scoped. What would the program print?

Solution:
The program would print 2020.
z will be 20 and x will be 20 □

Problem 5.

Solution:

Pass by value: 1 2 3 4 5 6 7 8 9 10
Pass by reference: 1 45 3 4 5 6 7 8 9 10
Pass by value-result: 1 15 3 4 5 6 7 8 9 10
Pass by name: 1 2 3 4 5 6 7 8 45 10

□

Problem 5.

Solution:

Pass by value: 1 2 3 4 5 6 7 8 9 10
Pass by reference: 1 45 3 4 5 6 7 8 9 10
Pass by value-result: 1 15 3 4 5 6 7 8 9 10
Pass by name: 1 2 3 4 5 6 7 8 45 10

□

Problem 6. In Ada, define a procedure containing two tasks such that (1) the first task prints the odd numbers from 1 to 99 in order, (2) the second task prints the even numbers from 2 to 100 in order, and (3) for each odd number N, the value of N and N+1 are printed concurrently but there is no other concurrent printing of numbers. For example, 1 and 2 are printed concurrently, and 3 and 4 are printed concurrently, but 1 and 2 must be printed before 3 and 4.

Solution:

Code in MainProg.adb

```
with Text_Io, Ada.Integer_Text_Io;
use Text_Io, Ada.Integer_Text_Io;

procedure MainProg is
        package Int_Io is new Integer_Io(Integer);
    use Int_Io;

    task OddTask is
        entry okDone;
    end OddTask;

    task EvenTask is
        entry okDone;
    end EvenTask;

-- Odd Task

```

```
    task body OddTask is
    begin
            for k in 1 .. 99 loop
                if k rem 2 = 1 then
                        accept okDone;
                        Ada.Integer_Text_Io.Put(k);
                        EvenTask.okDone;
                end if;
            end loop;
    end OddTask;

-- Even Task

    task body EvenTask is
    begin
            for k in 2 .. 100 loop
                if k rem 2 = 0 then
                        accept okDone;
                        Ada.Integer_Text_Io.Put(k);
                        OddTask.okDone;
                end if;
            end loop;
    end EvenTask;


begin
        OddTask.okDone;
        -- Text_Io.put("Hello");
end MainProg;
```

□