Programming Languages
CSCI-GA.2110.001 Spring 2015

Homework 1
ANSWERS

Note: There may be many correct answers to various of the questions. The answers below are just once instance of that.

1. Provide regular expressions for defining the syntax of the following.

   (a) Passwords consisting of letters and digits that contain exactly two upper case letters and exactly one digit, such that the digit is somewhere between the two upper case letters. They can be of any length (obviously at least three characters).
   [a-z]*[A-Z][a-z]*[0-9][a-z]*[A-Z][a-z]*

   (b) Floating point literals that specify an exponent, such as the following: 243.876E11 (representing 243.867×10^{11}). There must be at least one digit before the decimal point and one digit after the decimal point (before the "E").
   [0-9]*[0-9].[0-9][0-9]*E[0-9][0-9]*

   (c) Procedure names that: must start with a letter; may contain letters, digits, and _ (underscore); and must be no more than 15 characters.
   [a-zA-z]([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)([a-zA-Z0-9_] | ϵ)
   Note: ϵ denotes the empty string.

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 don't have to define the syntax for names and numbers).

```
x: int;

fun f(x: int, y: int)
  z:int;
{
    z = x+y-1;
    z = z + 1;
    return z;
}

proc g()
  a:int;
{
  a = 3;
  x = f(a, 2);
}
```

You only have to create grammar rules that are sufficient to parse the above program. Your starting non-terminal should be called PROG (for "program") and the above program should be able to be derived from PROG.

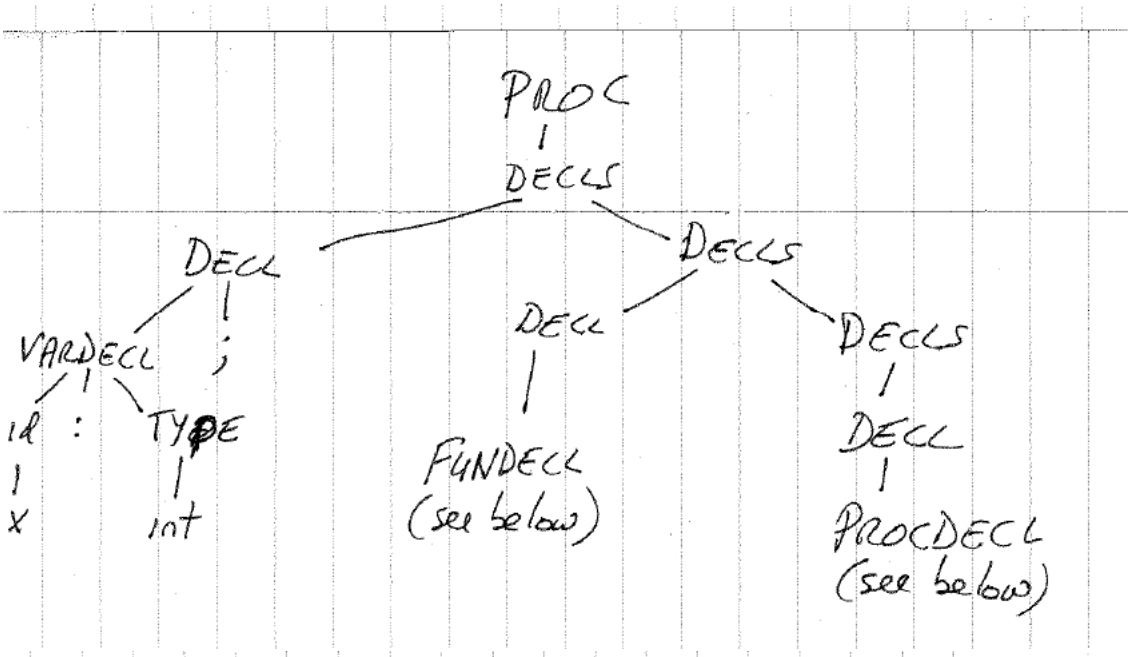
Here is one such grammar - though other, better, ones are certainly possible.

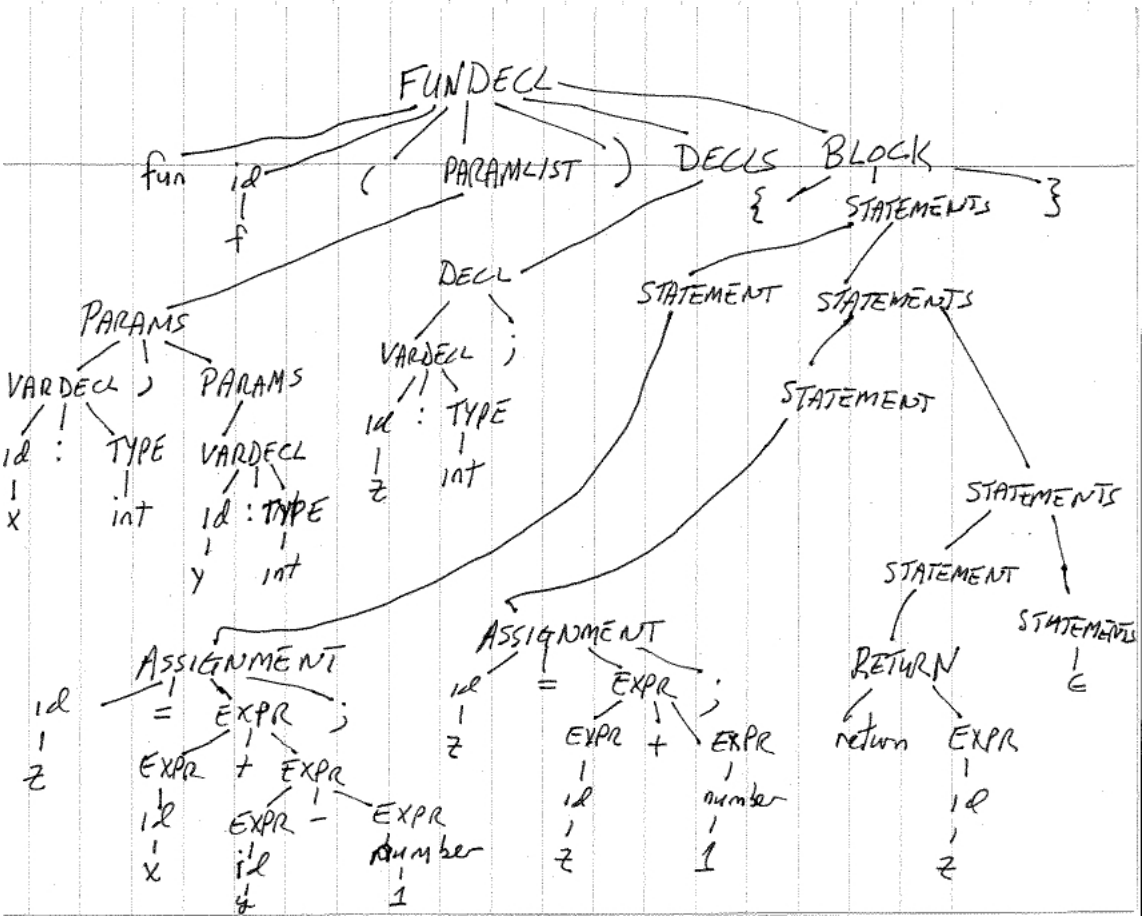
PROG → DECLS
DECLS → DECL DECLS | DECL
DECL → VARDECL ; | FUNDECL | PROCDECL
VARDECL → id : TYPE
TYPE → int | float
FUNDECL → fun id (PARAMLIST) DECLS BLOCK
PROCDECL → proc id (PARAMLIST) DECLS BLOCK
PARAMLIST → PARAMS | ϵ
PARAMS → VARDECL | VARDECL , PARAMS
BLOCK → { STATEMENTS }
STATEMENTS → STATEMENT STATEMENTS | ϵ
STATEMENT → ASSIGNMENT | RETURN
ASSIGNMENT → id = EXPR ;
RETURN → return EXPR ;
EXPR → id | number | EXPR + EXPR | EXPR * EXPR | (EXPR) | CALL
CALL → id (ARGLIST)
ARGLIST → ARGS | ϵ
ARGS → EXPR | EXPR , ARGS

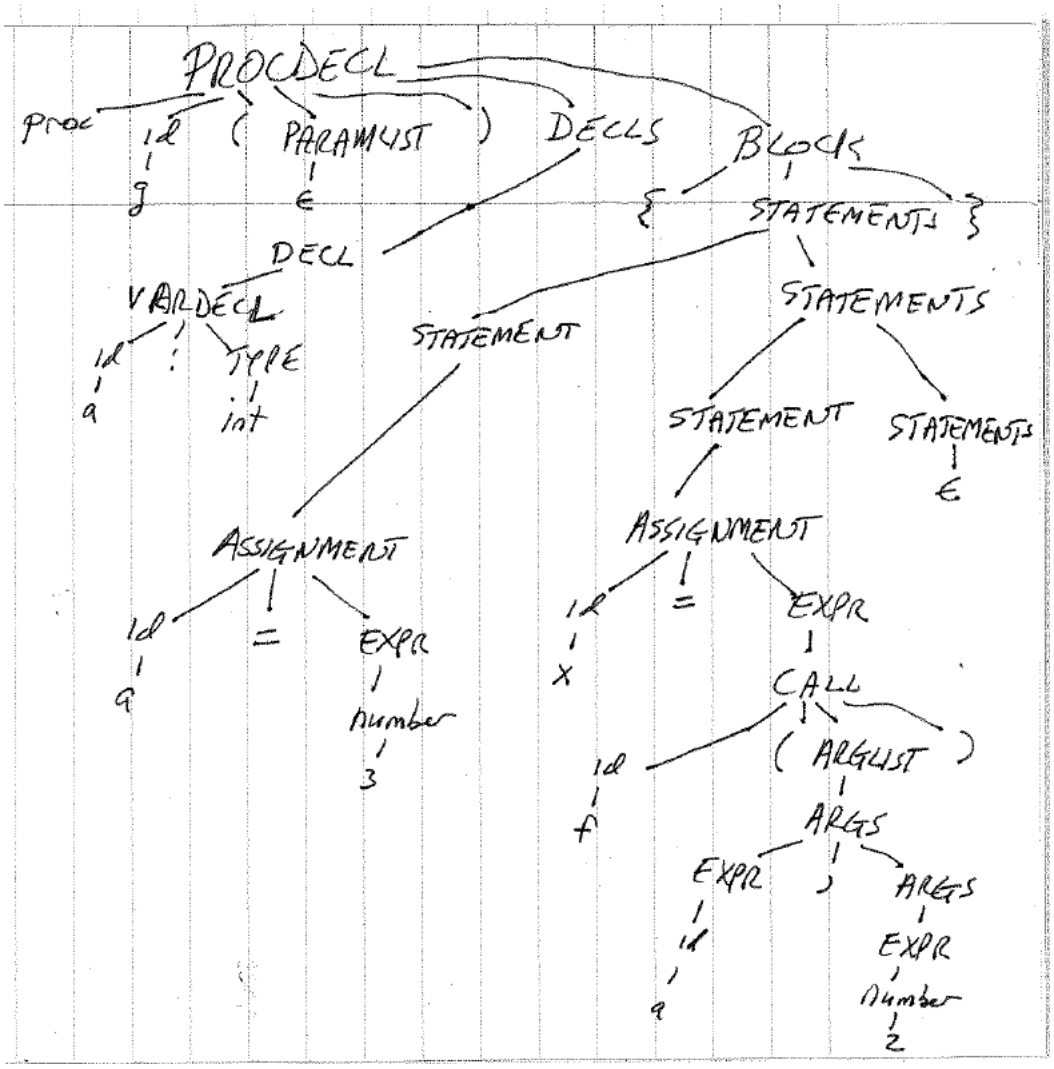
Note that the above grammar is ambiguous, which can be fixed by modifying the production for EXPR, as discussed in class (but not required for this assignment.

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

Below (in three parts) is the parse tree for the above program and grammar, which has an edge for every step in the derivation (every application of a production rule). It would also be acceptable to simplify the tree, removing some intermediate nodes and edges. Also, since `id` and `number` are terminals, their expansion to the actual identifiers (e.g. `x`, `f`, etc.) and numbers found in the program does not have to be shown.


PROC
DECLS
DECL
DECLS
VARDECL
;
DECL
DECLS
id
:
TYPE
FUNDECL
(see below)
DECL
PROCDECL
(see below)
x
int


FUNDECL
fun
id
(
PARAMLIST
)
DECLS
BLOCK
f
{
STATEMENTS
}
PARAMS
DECL
STATEMENT
STATEMENTS
VARDECL
PARAMS
VARDECL
;
STATEMENT
id
:
TYPE
id
:
TYPE
id
:
TYPE
z
int
x
int
y
int
STATEMENTS
ASSIGNMENT
ASSIGNMENT
STATEMENT
RETURN
STATEMENTS
id
=
EXPR
;
id
=
EXPR
;
return
EXPR
∈
z
EXPR
+
EXPR
EXPR
+
EXPR
id
id
number
id
z
z
1
z
EXPR
-
EXPR
id
number
x
1
4
1



3. (a) Define the terms *static scoping* and *dynamic scoping*.

In static scoping, the body of a function is evaluated in the environment of the function definition. In dynamic scoping, the body of a function is evaluated in the environment of the function call.

(b) 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.

```
procedure A()
  x: integer = 5;

 procedure B()
 begin
  print(x); (* static scoping prints 5, dynamic prints 10 *)
 end;
```

```
 procedure C()
   x: integer = 10;
 begin
   B();
 end;

begin (*A*)
  C();
end;
```

(c) In a block structured, statically scoped language, what is the rule for resolving variable references (i.e. given the use of a variable, how does one find the declaration of that variable)?

Given the use of (reference to) a non-local variable in a function, the corresponding variable declaration is found (at compile time) by looking at the scope surrounding the definition of the function, then in the next outer scope, and so on. At run time, the variable is found by traversing the static chain for the predetermined number of hops, corresponding to the difference between the nesting levels of the variable's use and definition.

(d) In a block structured but dynamically scoped language, what would the rule for resolving variable references be?

Given the use of a non-local variable in a function, the corresponding variable declaration is found by looking at calling function to see if the variable is defined there. If not, the calling function of the calling function is examined to determine if the variable is defined there, and so on. That is, the the dynamic chain is traversed, and each encountered stack frame is checked, until the variable is found.

4. (a) Draw the state of the stack, including all relevant values (e.g. variables, return addresses, dynamic links, static links, closures), at the time that the `writeln(y)` is executed.

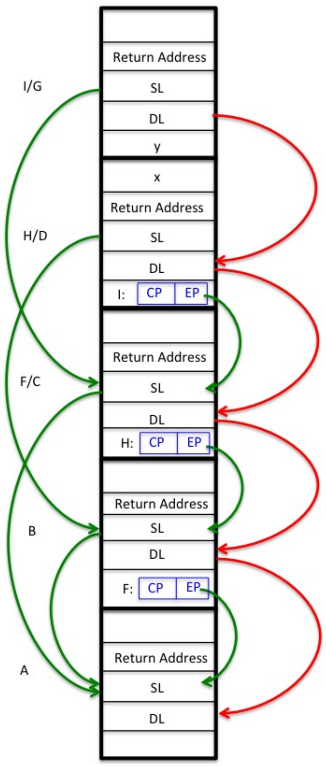
```
procedure A;

    procedure B(procedure C)
       procedure  D(procedure I);
         x: integer := 6;
       begin (* D *)
         I(x);
       end;
    begin (* B *)
      C(D);
    end;

    procedure F(procedure H)
      procedure G(y: integer)
      begin (* G *)
         writeln(y); (* draw state of stack when this is executed *))
      end;
    begin (* F *)
```

```
        H(G);
    end;

begin (* A *)
  B(F);
end;
```



(b) Explain why closures on the heap are needed in some languages, and give an example of a program (in any syntax you like) in which a closure would need to be allocated on the heap.

An object needs to be heap allocated if the object outlives the function that created it. In this case, the object is a closure representing a function, thus the closure would have to be allocated if the function represented by the closure outlives the defining function. For example,

```
function A(x: integer)

  procedure B()
  begin
    print(x+1);
  end

  procedure C()
```

```
    z: integer = 4;
  begin
    print(x+2);
  end

begin (* A *)
  if (x = 3) then
    return B;
  else
    return C;
end
```

In this case, the function A returns either the procedure B or the procedure C. Therefore, the closure used to represent the return value, B or C, must be heap-allocated (and must preserve access to x).

5. For each of these parameter passing mechanims,

   (a) pass by value
   (b) pass by reference
   (c) pass by value-result
   (d) pass by name

   state what the following program (in some Pascal-like language) would print if that parameter passing mechanism was used:

```
program foo;
  var i,j: integer;
      a: array[1..5] of integer;

  procedure f(x,y:integer)
  begin
    x := x * 2;
    i := i + 1;
    y := a[i] + 1;
  end

begin
  for j := 1 to 5 do a[j] = j*2;
  i := 2;
  f(i,a[i]);
  for j := 1 to 5 do print(a[j]);
end.
```

Pass by Value: 2 4 6 8 10
Pass by Reference: 2 11 6 8 10
Pass by Value Result: 2 7 6 8 10 (assuming the location of the parameter a[i] is only computed once).
Pass by Name: 2 4 6 8 11

6. (a) In Ada, define a procedure containing two tasks, each of which contains a single loop. The loop in the first task prints the numbers from 1 to 1000, the loop in the second task prints the numbers from 2001 to 3000. The execution of the procedure should cause the tasks to alternate printing one hundred numbers at a time, so that the user would be guaranteed to see:
`1 2...100 2001 2002...2100 101 102...200 2101 2102...2200 201...`
Be sure there is only one loop in each task.

```
procedure Hw1 is

   task T1 is
      entry Go;
   end T1;

   task T2 is
      entry Go;
   end T2;

   task body T1 is
   begin
      for I in 1..1000 loop
 Put(I);
 if (I mod 100) = 0 then
    T2.Go;
    accept Go;
 end if;
      end loop;
   end T1;

   task body T2 is
   begin
      for I in 2001..3000 loop
 if (I mod 100) = 1 then
    accept Go;
 end if;
 Put(I);
 if (I mod 100) = 0 then
    T1.Go;
 end if;
      end loop;
   end T2;

begin
   null;
end Hw1;
```

(b) Looking at the code you wrote for part (a), are the printing of any of the numbers occurring concurrently? Justify your answer by describing what concurrency is and why

these events do or do not occur concurrently.

The printing of the numbers are not occurring concurrently. Concurrency is the absence of a predetermined ordering among events (i.e. no assumptions can be made about the order in which the events will occur), but in this case, the order in which the numbers are printed (and by whom) is completely predetermined.