Programming Languages
CSCI-GA.2110.001 Fall 2015

Homework 1
ANSWERS

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

   (a) Passwords consisting of letters and digits that contain at least two upper case letters and one digit. They can be of any length (obviously at least three characters).

   ```
   ([A-Za-z0-9]*[A-Z][A-Za-z0-9]*[A-Z][A-Za-z0-9]*[0-9][A-Za-z0-9]*)|
   ([A-Za-z0-9]*[A-Z][A-Za-z0-9]*[0-9][A-Za-z0-9]*[A-Z][A-Za-z0-9]*)|
   ([A-Za-z0-9]*[0-9][A-Za-z0-9]*[A-Z][A-Za-z0-9]*[A-Z][A-Za-z0-9]*)
   ```

   (b) Floating point literals that specify an exponent, such as the following: 2.43876E13 (representing 2.43867×10^{13}).

   ```
   [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 10 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)
   ```

   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).

   ```
   program one;

     var x;

     function f(var x, var y)
       var z;
     begin
       z := x+y-1;
       return z*2;
     end f;

     procedure g()
       var a;
     begin
       a := 3;
       x := a;
     end g;
   ```

```
begin
 g();
 print(f(x));
end one;
```

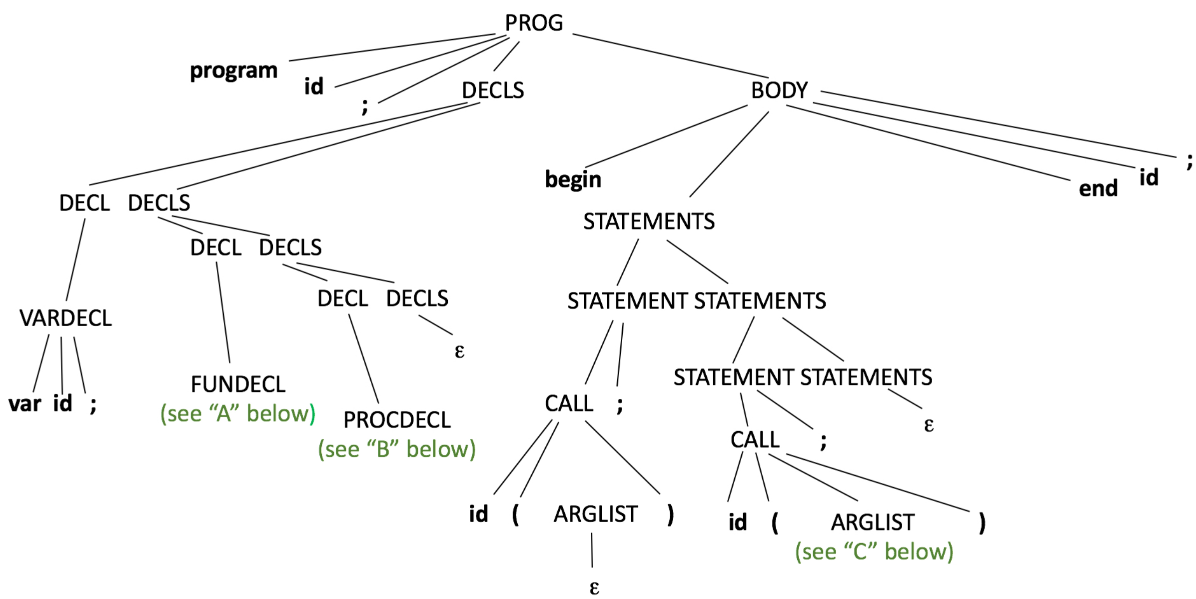
You only have to create grammar rules that are sufficient to parse the above program.

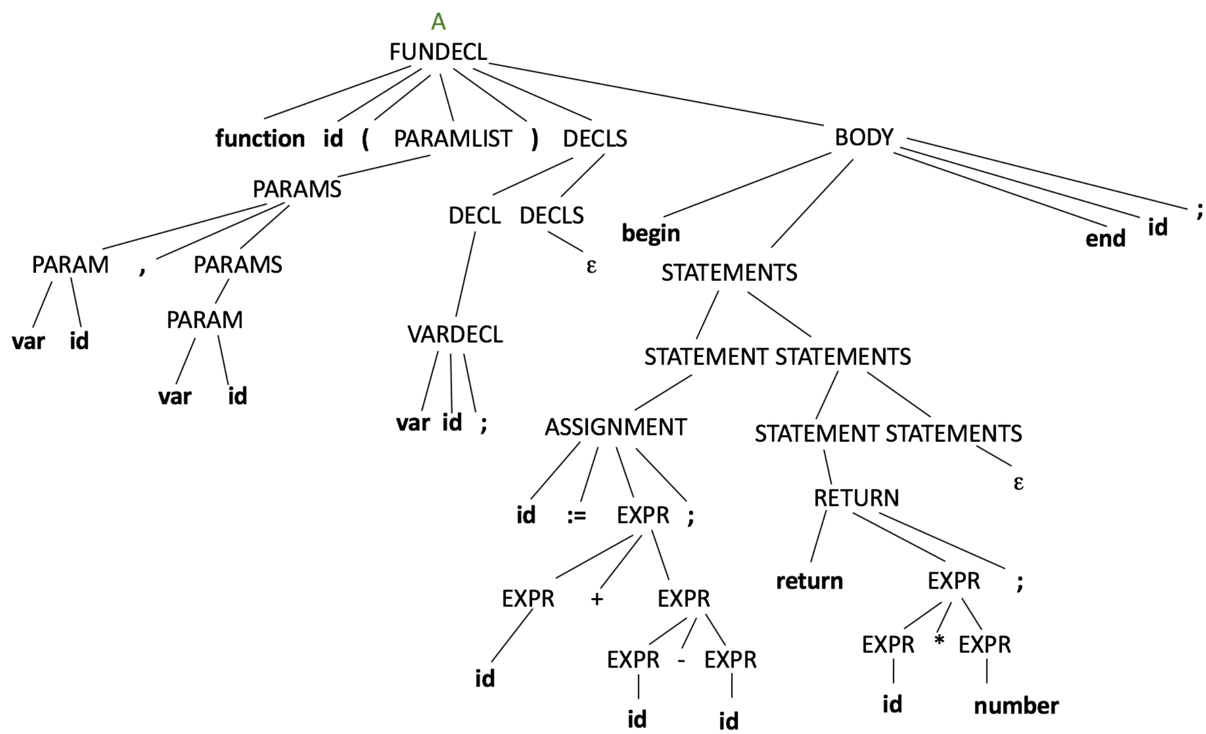
Here is one such grammar - though other ones are certainly possible.

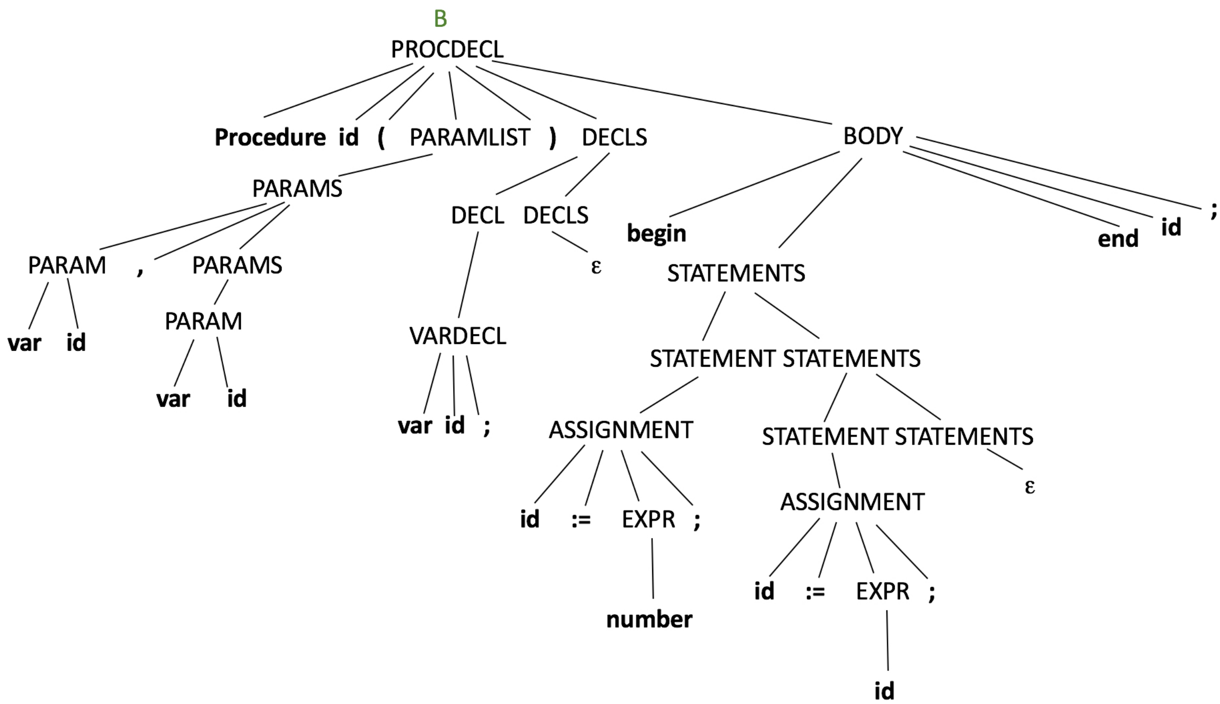
```
PROG → program id ; DECLS BODY
DECLS → DECL DECLS | ϵ
DECL → VARDECL | FUNDECL | PROCDECL
VARDECL → var id ;
FUNDECL → function id ( PARAMLIST ) DECLS BODY
PROCDECL → procedure id ( PARAMLIST ) DECLS BODY
PARAMLIST → PARAMS | ϵ
PARAMS → PARAM | PARAM , PARAMS
PARAM → var id
BODY → begin STATEMENTS end id ;
STATEMENTS → STATEMENT STATEMENTS | ϵ
STATEMENT → ASSIGNMENT | CALL ; | 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
```

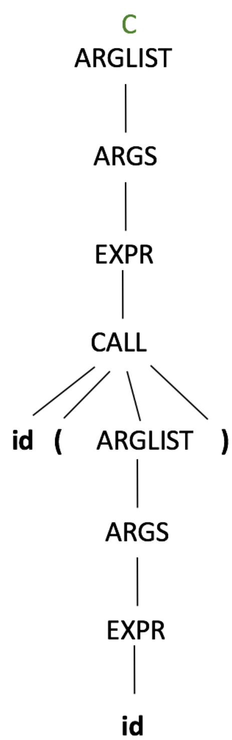
Note that the above grammar is ambiguous, which can be fixed by modifying the production for EXPR, as discussed in class (but not required for this assignment).

(b) Draw the parse tree for the above program.


PROG
program
id
;
DECLS
BODY
DECL DECLS
DECL DECLS
DECL DECLS
ε
VARDECL
var id ;
FUNDECL
(see "A" below)
PROCDECL
(see "B" below)
begin
STATEMENTS
STATEMENT STATEMENTS
CALL ;
id (ARGLIST)
ε
STATEMENT STATEMENTS
CALL ;
id (ARGLIST)
(see "C" below)
ε
end
id
;


A
FUNDECL
function id (PARAMLIST) DECLS
BODY
PARAMS
PARAM , PARAMS
var id
PARAM
var id
DECL DECLS
VARDECL
var id ;
ε
begin
STATEMENTS
end
id
;
STATEMENT STATEMENTS
ASSIGNMENT
id := EXPR ;
EXPR + EXPR
id
EXPR - EXPR
id id
STATEMENT STATEMENTS
RETURN
ε
return EXPR ;
EXPR * EXPR
id number





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

Static scoping means that a procedure body is evaluated in the environment (mapping of variables to values or locations) of the procedure definition. Dynamic scoping means that a procedure body is evaluated in the environment

of the procedure 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 scoping 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.

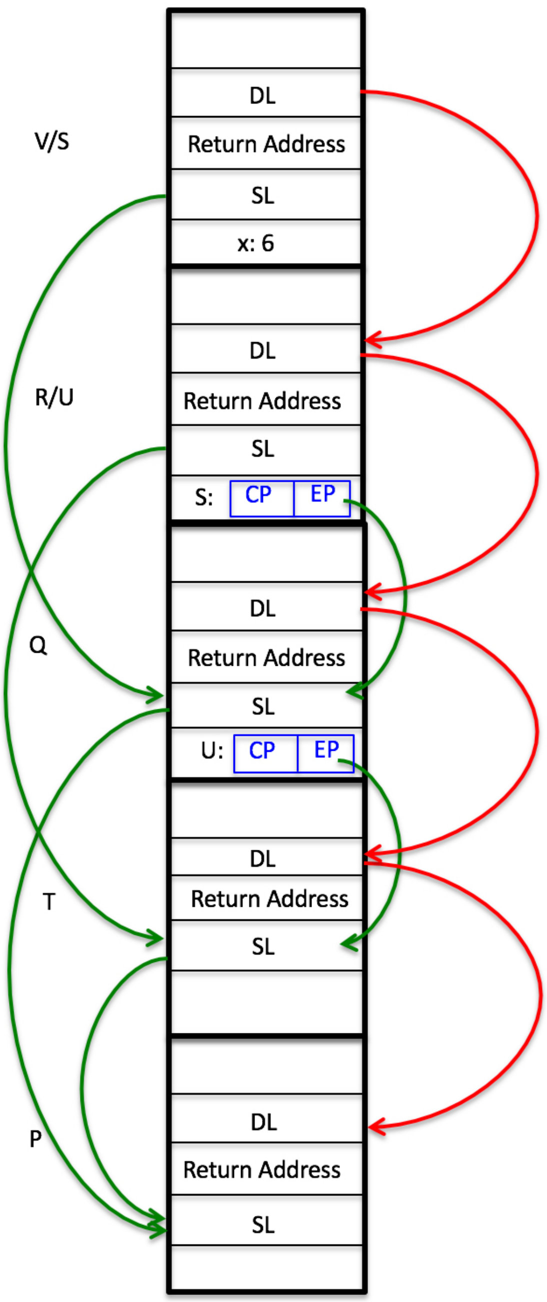
(d) In a block structured but dynamically scoped language, what would the rule for resolving variable references be?

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 address, dynamic link, static link), during the execution of procedure S in the following program.

```
procedure P;
  procedure Q(procedure R)
    procedure  S(x:integer);
    begin
```

```
      writeln(x);
    end;
 begin (* Q *)
   R(S);
 end;
 procedure T;
   procedure U(procedure V);
    begin
       V(6);
    end;
  begin (* T *)
      Q(U);
  end;
begin (* P *)
 T;
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..6] of integer;

  procedure f(x,y:integer)
  begin
    x := x * 3;
    i := i + 1;
    y := a[i] + 2;
  end

begin
  for j := 1 to 6 do a[j] = j;
  i := 1;
  f(i,a[i]);
  for j := 1 to 6 do print(a[j]);
end.
```

Pass by Value: **1 2 3 4 5 6**

Pass by Reference: **6 2 3 4 5 6**

Pass by Value-Result: **4 2 3 4 5 6**

Note: The above answer assumes the location of the parameter a[i] is only computed once, i.e. when f is called. If the location of a[i] is computed twice, i.e. when f is called and then again when f returns, the result would be either 1 2 4 4 5 6 or 1 4 3 4 5 6, depending on the order in which the formal parameters are copied back to the actual parameters when f returns.

Pass by Name: **1 2 3 6 5 6**

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 500, the loop in the second task prints the numbers from 501 to 1000. The execution of the procedure should cause the tasks to alternate printing fifty numbers at a time, so that the user would be guaranteed to see:

   `1 2...50 501 502 ...  550 51 52 ...  100 551 552 ...  600 ...`

   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..500 loop
         Put(I);
         if (I mod 50) = 0 then
            T2.Go;
            accept Go;
         end if;
      end loop;
   end T1;

   task body T2 is
   begin
      for I in 501..1000 loop
         if (I mod 50) = 1 then
```

```
            accept Go;
         end if;
         Put(I);
         if (I mod 50) = 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.