Programming Languages CSCI-GA.2110.001 Spring 2015

Homework 1 Due Sunday, March 8

You should write the answers using Word, latex, etc., and upload them as a PDF document. No implementation is required. Since there are drawings, if you prefer, you can hand write the answers and scan them to a PDF.

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).

   Answer: [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 (rep- resenting 243.867 × 1011). There must be at least one digit before the decimal point and one digit after the decimal point (before the "E").

   Answer: [0-9] + [.] [0-9] + [e E] [+ -] ? [0-9]+

   (c) Procedure names that: must start with a letter; may contain letters, digits, and (un- derscore); and must be no more than 15 characters.

   Answer: [a-z] [a-z 0-9 _]{0,14}

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).

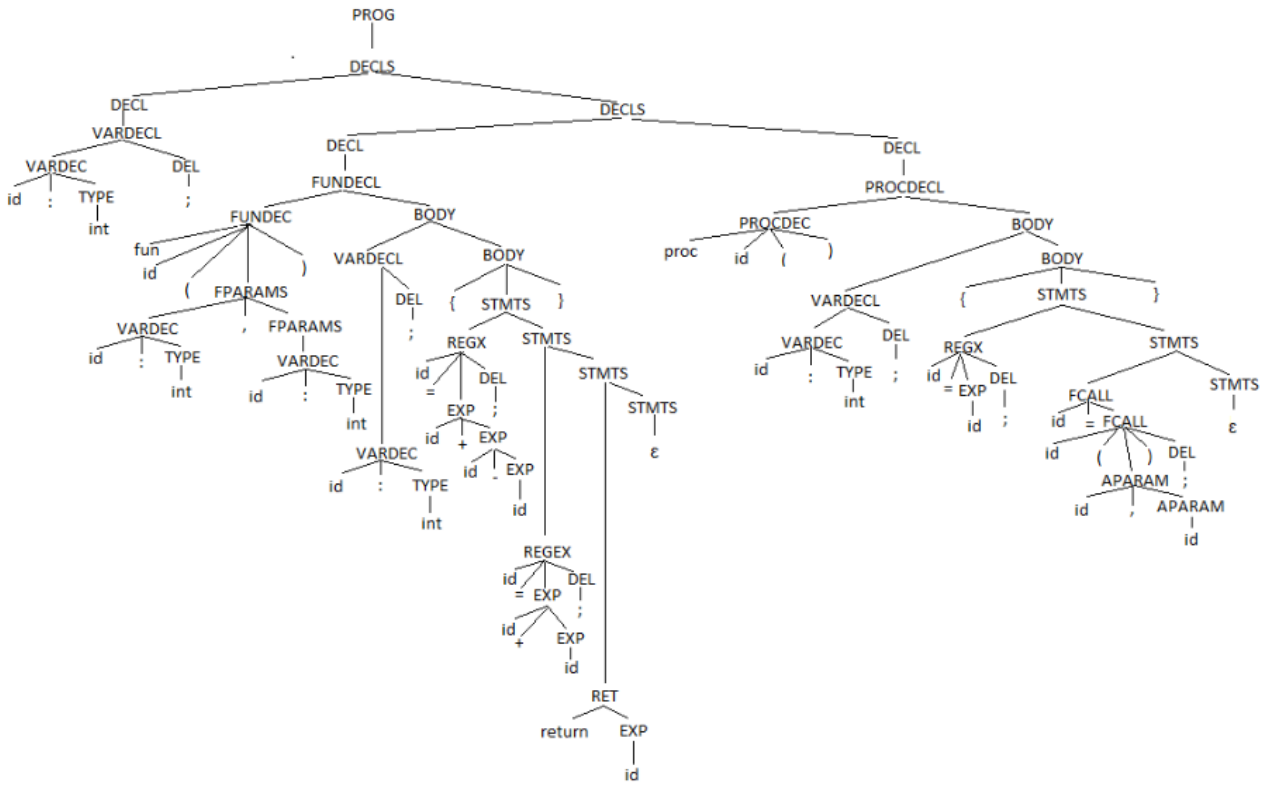
```
x: int;

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 pro- gram should be able to be derived from PROG.

Answer:

PROG	-> DECLS
DECLS	-> DECL DECLS | DECL
DECL	-> VARDECL | FUNDECL | PROCDECL
VARDECL	-> VARDEC DEL
VARDEC	-> Id : TYPE
TYPE	-> int | float
FUNDECL	-> FUNDEC BODY | ε
PROCDECL	-> PROCDEC BODY
FUNDEC	-> fun id (FPARAMS)
FPARAMS	-> VARDEC, FPARAMS | VARDEC
PROCDEC	-> proc g ( )
BODY	-> VARDECL BODY | { STMTS }
STMTS	-> REGX STMTS | RET STMTS | FCALL STMTS | ε
REGX	-> id = EXP DEL
EXP	-> id + EXP | id – EXP | id
RET	-> return EXP DEL
FCALL	-> id = FCALL | id (APARAM) DEL
APARAM	-> id , APARAM | id

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



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

Answer:

Static scoping: Static scoping is a convention used with many programming languages in which the scope of a variable is resolved in the scope of the function definition, so that it may only be called (referenced) from within the block of code in which it is defined.

Dynamic scoping: Dynamic scoping means that when a symbol is referenced, the compiler/interpreter will walk up the symbol-table stack to find the correct instance of the variable to use. In this convention the variables are resolved according to the scope of the 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.

Answer:

**********Program which gives different results on static and dynamic scoping**********

```
Procedure A()
        x : integer;
        Procedure B()
                x : integer;
        Begin
                x := 1;
                D();
        End;
        Procedure C()
                x : integer;
        Begin
                x := 3;
                D();
        End;
        Procedure D()
        Begin
                Put_Line(x);
                New_Line;
        End;
Begin
        x := 10;
        B();
        C();
End;
```

Output with Static scoping:

10

10

Output with dynamic scoping:

1

3

(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)?

Answer: In static scoping the scope of variable within a block is resolved in the scope of block definition. Static links are used to resolve a non-local variable reference in a block structured statically scoped programming language. Static link points to the stack frame of the defining procedure. In case of nested blocks the compiler follows the static chain to resolve variable references.

In Programming languages which has first-class functions, the static link for the current stack is copied from the 'Closure' which is passed by the calling procedure. The Closure has the environment pointer for the current stack which is the static link.

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

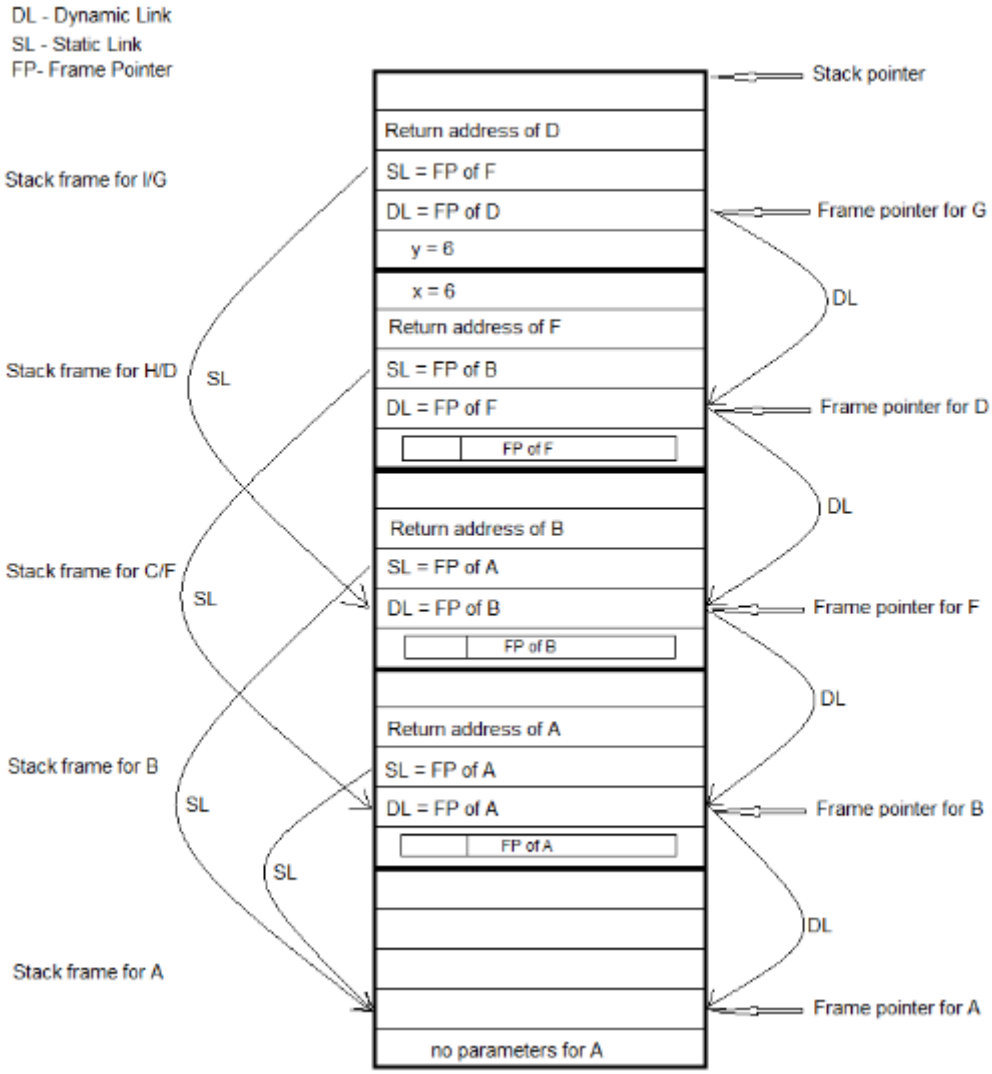
Answer: In dynamically scoped language, the variable references are resolved according to the scope of the call. Dynamic links are used to resolve a non-local variable reference. Dynamic link points to the stack frame of the calling procedure.

In case of nested blocks the compiler follows the dynamic chain to resolve variable references.

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;
```

Answer:



(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.

Answer: Closures on heap are needed in some languages as procedures can create objects that outlive the procedure i.e. the lifetime of an object exceeds that of its scope.

Example:

1. Imaginary language:

```
Vehicle foo()
 {
        Vehicle V = new Vehicle ();
        return V;
 }

 Main(){

 Vehicle W = foo();
 }
```

2. Javascript:

```
<html>
<body>
<button type="button" onclick="call()">Click Me</button>
<p>Counter is : </p>
<p id="1">0</p>

<script>
var add = (function () {
   var counter = 0;
   return function () {return counter += 1;}
})();

function call(){

   document.getElementById("1").innerHTML = add();
}
</script>
</body>
</html>
```

3. Java:

```
class square {
     int length;

     square(int l) {
       length = l;
     }

    square get_new_square() {
       square sq = new square(10);
       return sq;
     }
}

class demo {
     public static void main(String args[]) {
       square ob1 = new square(40);
       square ob2;
       ob2 = ob1.get_new_square();
       }
}
```

5. For each of these parameter passing mechanism,

   (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.
```

Answer:

(a) pass by value:

Prints: 2 4 6 8 10

Explanation:

```
program foo;
        var i,j: integer;
        a: array[1..5] of integer;
        procedure f(x,y:integer)          // f(2, 4)
        begin
                x := x * 2;               // x = 2* 2 = 4
                i := i + 1;               // i = 2 + 1 = 3
                y := a[i] + 1;            // y = a[3] + 1 = 6 + 1 = 7
        end
begin
        for j := 1 to 5 do a[j] = j*2;  // sets the array to {2,4,6,8,10}
        i := 2;
        f(i,a[i]);                      // f(2, a[2]) = f(2, 4) //no value is passed back
        for j := 1 to 5 do print(a[j]); // prints 2 4 6 8 10
end
```

(b) pass by reference:

Prints: 2 11 6 8 10

Explanation:

```
program foo;
        var i,j: integer;
```

```
        a: array[1..5] of integer;
        procedure f(x,y:integer)        // f(pointer to i, pointer to a[2])
        begin
                x := x * 2;             // i = i* 2 = 4
                i := i + 1;             // i = i + 1 = 4 + 1 = 5
                y := a[i] + 1;          // a[2] = a[5] + 1 = 10 + 1 = 11
        end
    begin
        for j := 1 to 5 do a[j] = j*2;  // sets the array to {2,4,6,8,10}
        i := 2;
        f(i,a[i]);                      // f(pointer to i, pointer to a[2])  i = 5 a[2] = 11
        for j := 1 to 5 do print(a[j]); // prints 2 11 6 8 10
    end
```

(c) pass by value-result:

Prints: 2 7 6 8 10

Explanation:

```
    program foo;
        var i,j: integer;
        a: array[1..5] of integer;
        procedure f(x,y:integer)        // f(2,4])
        begin
                x := x * 2;             // x = 2 * 2 = 4
                i := i + 1;             // i = i + 1 = 2 + 1 = 3
                y := a[i] + 1;          // y = a[3] + 1 = 6 + 1 = 7
        end
    begin
        for j := 1 to 5 do a[j] = j*2;  // sets the array to {2,4,6,8,10}
        i := 2;
        f(i,a[i]);                      // f(2, a[2])  returns i = 4 a[2] = 7
        for j := 1 to 5 do print(a[j]); // prints 2 7 6 8 10
    end
```

(d) pass by name:

Prints: 2 4 6 8 11

Explanation:

```
    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;  // sets the array to {2,4,6,8,10}
        i := 2;
        f(i,a[i]);
    //replace function call by the body of the function replacing the formal parameters by actual
    //parameters
        i := i * 2;                     // i = 4
        i := i + 1;                     // i = 5
        a[i] := a[i] + 1;               // a[5] = a[5] + 1 = 11
        for j := 1 to 5 do print(a[j]); // prints 2 4 6 8 11
    end
```

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.

Answer:

```
--**********************************Program***********************************************
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Integer_Text_IO; use Ada.Integer_Text_IO;

Procedure Print is

    task type Print1 is
        entry run( x: in integer);
    end Print1;

    task type Print2 is
        entry run( x: in integer);
    end Print2;

    task body Print1 is
        counter : integer;
        start   : integer;
        stop    : integer;
```

```
    Begin
            For j in 1 .. 10 loop
            Accept run ( x : in integer ) do
                    counter := x;
                    counter := counter - 1;
                    start := (counter * 100) + 1;
                    stop := start + 99;

            for count in start .. stop loop
                    Ada.Integer_Text_IO.put(count);
            End loop;
            end;
            end loop;
    end Print1;
    task body Print2 is
            counter : integer;
            start   : integer;
            stop    : integer;
    Begin
            for k in 1 .. 10 loop
            Accept run ( x : in integer ) do
                    counter := x;
                    counter := counter - 1;
                    start := (counter * 100) + 1 +2000;
                    stop := start + 99;

            for count in start .. stop loop
                    Ada.Integer_Text_IO.put(count);
            End loop;
            end;
            end loop;
    end Print2;

Prnt1 : Print1;
Prnt2 : Print2;

Begin
for i in 1 .. 10 loop
Prnt1.run(i);
Prnt2.run(i);
end loop;
```

```
end;
--*****************************************************************************************************
```

(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.

Answer:

Concurrency is achieved when different independent piece of logic are executed simultaneously or in interleaved manner by the process in which the relative order of execution cannot be determined.

In Ada concurrency is achieved through tasks which run concurrently with the procedure in which they are defined.

In this code the printing does not occur concurrently because the order execution of the tasks are being controlled by the entries to the task.

When a task reaches an entry point it waits for call to the entry by the main procedure (rendezvous). Once rendezvous occurs the tasks continues it execution once the logic in accept block has been executed.

In the above program the logic to print the numbers is within the accept block of the task hence it does not get executed concurrently.