Programming Languages
CSCI-GA.2110.001 Spring 2015

Midterm Exam
ANSWERS

Please write all answers in the blue book. Keep your answers brief!

1. (a) For each of the following sets, indicate if the set can be specified using a regular expression. If so, provide the regular expression. If not specify why not.

   i. The set of all strings representing the positive numbers less than 100, i.e. {`1, 2, 3, ..`}.
   Trivially, it could be expressed as:
   `1 | 2 | ... | 99`
   A better regular expression would be:
   `[1-9]([0-9] |` ϵ`)`
   where `[0-9]` represents `(0 | 1 | ... | 9)`.

   ii. The set of all strings containing only parentheses, i.e. "(" and ")", such that the parentheses are balanced.
   This cannot be expressed using a regular expression. Regular expressions cannot be used, in general, to express nesting.

   iii. The set of all strings containing containing equal numbers of `x`'s and `y`'s.
   This cannot be expressed using a regular expression. Regular expressions cannot be used, in general, to count occurrences of characters (as would be needed in this case)

   (b) For each of the above sets, indicate if the set can be specified using a context free grammar. If so, provide the CFG. If not, explain why not.

   i. Trivially, it could be expressed as:
   `S` $\rightarrow$ `1 | 2 | ... | 99`
   Here is a better CFG :
   `S` $\rightarrow$ `POSDIGIT | POSDIGIT DIGIT`
   `DIGIT` $\rightarrow$ `0 | POSDIGIT`
   `POSDIGIT` $\rightarrow$ `1 | 2 | ... | 9`

   ii. `S` $\rightarrow$ `(S) | SS |` ϵ

   iii. $S \rightarrow$ `x`S`y`S `|` `y`S`x`S `|` ϵ

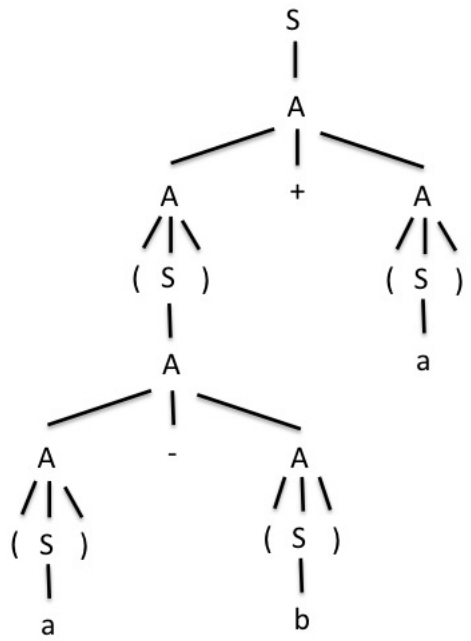
   (c) Consider the following grammar in which S and A are non-terminals and `a`, `b`, `+`, `-`, `(`, and `)` are terminals.

   $S \rightarrow$ `a` | `b` | A
   $A \rightarrow A$ `+` A | A `-` A | `(` S `)`

   Using this grammar, generate a short string containing every one of the terminals and show the corresponding parse tree. Assume S is the starting non-terminal.

   Here is a derivation that generates the string `((a) - (b)) + (a)`:
   $S \Rightarrow A \Rightarrow A$ `+` $A \Rightarrow$ `(`S`)` `+` $A \Rightarrow$ `(`A `-` A`)` `+` $A \Rightarrow$
   `((`S`)` `-` A`)` `+` $A \Rightarrow$ `((a)` `-` A`)` `+` $A \Rightarrow$ `((a)` `-` `(`S`))` `+` $A \Rightarrow$
   `((a) - (b))` `+` $A \Rightarrow$ `((a) - (b))` `+` `(`S`)` $\Rightarrow$ `((a) - (b)) + (a)`

   The corresponding parse tree would be:



2. Write a very short program that would print a different result if pass-by-reference or pass-by-name parameter passing were used. Show the result for each parameter passing mechanism.

   Here is such a program:

```
program main;
   i: integer;
   a: array[1..2] of integer;

   procedure f(x:integer, y:integer)
   begin
     x := x + 1;
     y := y + 1;
   end

begin (* main *)
  i := 1;
  a[1] := 1; a[2] := 2;
  f(i,a[i]);
  print(a[1],a[2]);
end;
```

   The output for pass-by-reference would be: 2 2
   The output for pass-by-name would be: 1 3

3. (a) What purpose does an Ada package specification serve?

   It declares the elements of a package (types, procedures, variables, etc.) that are visible outside of the package.

   (b) The following Ada code was written with the intent of printing the numbers from 10 to 20, but there is a bug. State what the program will print as it is currently written and why.

```
task taskOne is
   entry E1;
end taskOne;

task body taskOne is
begin
   for I in 10..20 loop
      taskTwo.E2;
      accept E1;
   end loop;
end taskOne;
```

```
task taskTwo is
   entry E2;
end taskTwo;

task body taskTwo is
begin
   for I in 10..20 loop
      accept E2 do
         put(I); new_line;
         taskOne.E1;
      end E2;
   end loop;
end taskTwo;
```

The program will print 10 and then hang (deadlock). When task `taskOne` makes the entry call `taskTwo.E2`, task `taskOne` will wait until `accept` block in task `taskTwo` is done. The bug is that, because the entry call `taskOne.E1` in task `taskTwo` is within the `accept` block, that entry call will never complete because task `taskOne` is still waiting for the accept block to complete.

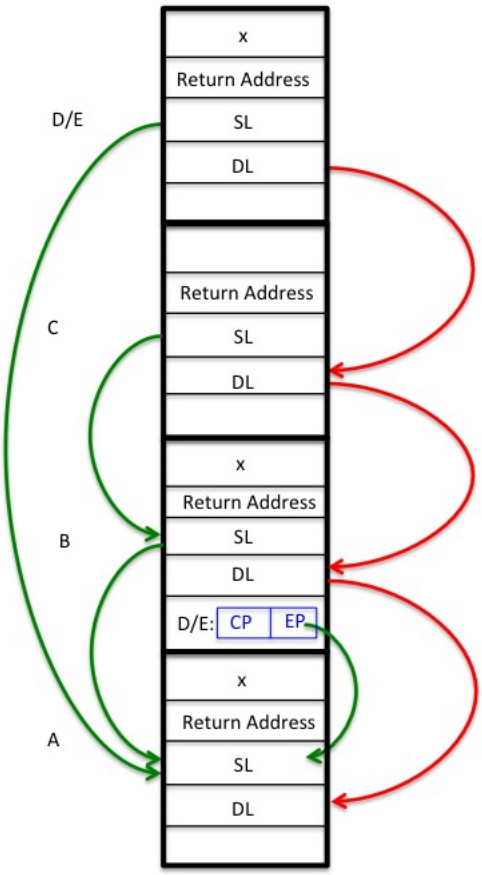
(c) What is a small change that can be made to the above code that would cause it to behave as intended? Just describe the change, you don't need to rewrite the code. The change should be made to only one of the tasks.

In task `taskTwo`, move the entry call `taskOne.E1` to below the end of the `accept` block.

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.

```
procedure A()
   x: integer := 7;

   procedure B(procedure D)
      x: integer := 5;

      procedure C()
      begin (* C *)
         D();
      end; (* C *)

   begin (* B *)
      C();
   end (* B *)

   procedure  E()
   begin (* E *)
      print(x);
   end; (* E *)

begin (* A *)
  B(E);
```

```
end; (* A *)
```



(b) What would the program print?

It would print 7, since the `x` referenced in `E` is the `x` declared in `A`.

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

It would print 5, since the first `x` encountered along the dynamic chain, starting at `E`'s stack frame is the `x` declared in `B`.

5. (a) In Scheme, write the function `(max-all L)`, where the elements of the list L are numbers or arbitrarily nested lists of numbers, that returns the largest number found at any depth within L. For example,

```
> (max-all '((10 2) 30 (45 (15 16 (73 58) 9) 13)))
73
```

Answer:

```
;; Assumes L has at least one element, since (max '()) makes no sense

(define (max-all L)
  (let ((max-car  (if (pair? (car L)) (car L) (max-all (car L)))))
    (cond
     ((null? (cdr L)) max-car)
```

```
      (else (let ((max-cdr (max-all (cdr L))))
              (if (> max-car max-cdr) max-car max-cdr))) )))
```

(b) In Scheme, write the function (every-other L) that, given a list L, returns the list containing every other element of L, i.e. the first element, the third element, the fifth element, etc. For example,

```
> (every-other '(1 2 (3 4) 5 6 (7 8) 9))
(1 (3 4) 6 9)
```

Answer:

```
(define (every-other L)
  (cond ((or (null? (cdr L)) (null? (cddr L))) (list (car L)))
        (else (cons (car L) (every-other (cddr L))))))
```