COP-5555 Programming Language Principles

Practice Final Questions

PROBLEM 1.

Add procedures to the denotational description of Tiny. Don't panic. We will consider procedures with no parameters, and with no local variables, but the procedures are recursive. Two new constructs need to be handled: a procedure declaration, and a procedure call.

Complete the following:

CC[<proc I C>] =

CC[<call I>] =

Hints:

1) The solutions are one-liners.

2) Treat the procedure declaration as an asignment to the procedure name I.

PROBLEM 2.

Draw a display of the run-time environment, at the point marked "HERE" in the code shown below, for each of the following two techniques:

1) Static Links.
2) Displays.

```
program main;

    procedure A;
    begin {A}
        ...
    end;

    procedure B(procedure E);
    var x:integer;

        procedure C;
        begin {C}
            x := 5;          <---------- HERE !
        end;

    begin {B}
        E;
        B(C);
    end;

begin {main}
    B(A)
end;
```

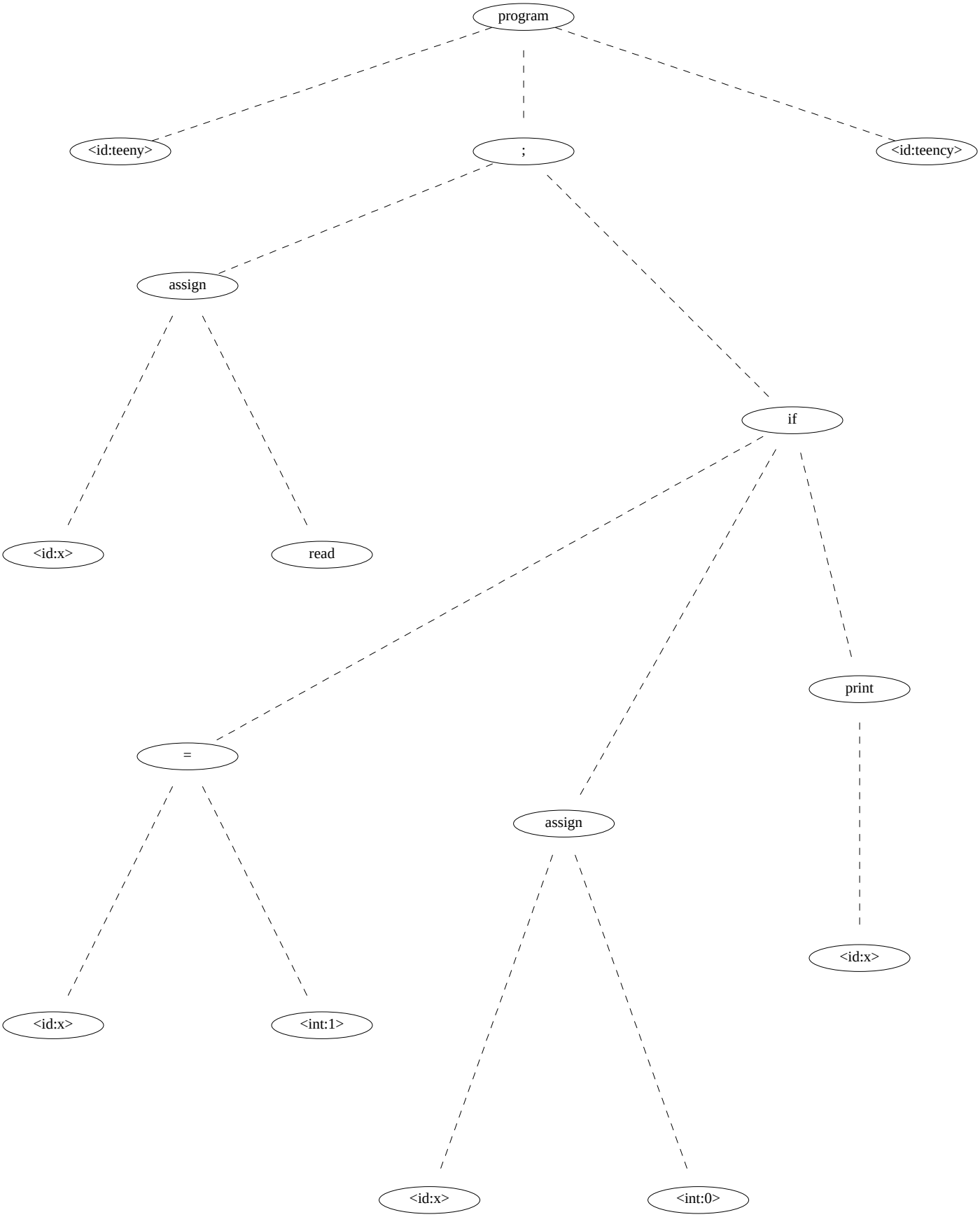
PROBLEM 3.

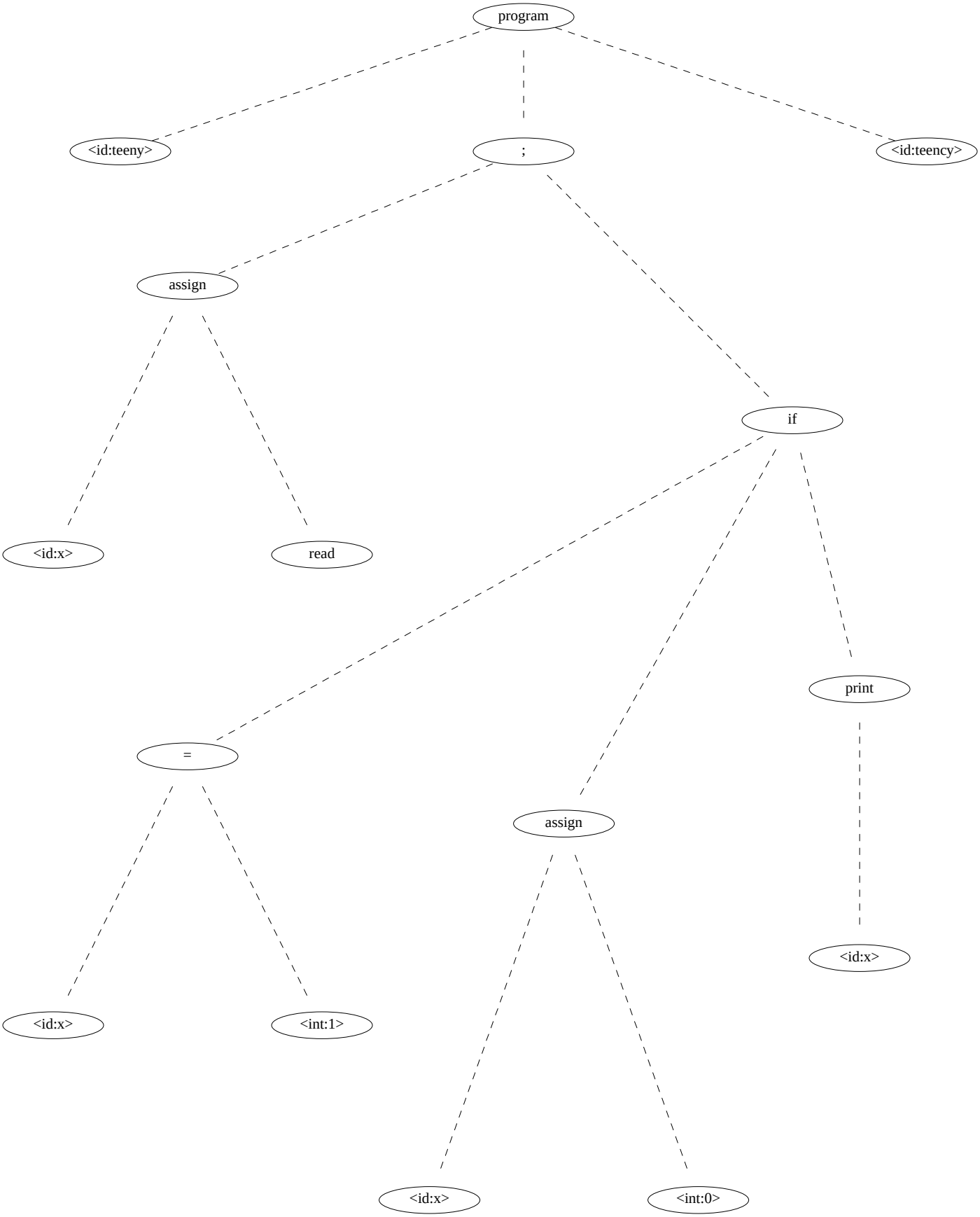
Define each of the following terms; using examples to illustrate them. Dynamic Type Binding, Operator Overloading, Dangling Pointer, Dynamic Scoping, Activation Record, Information Hiding, Object-Oriented Programming Language.

PROBLEM 4.

On the next two pages are two copies of the AST for a small Tiny program. Place next to each tree node the necessary attributes. For most nodes, this means four inherited attributes (on the node's left), and five synthesized attributes (on the right). Fill in the actual values of ALL the attributes, according to the attribute grammar in your class notes, i.e. show the result of propagating the values through the functional graph. DO NOT depict the functional graph itself. Please adher to the convention adopted in class: the attributes are, from left to right: code, error, next, top, type.

Note: The first copy of the tree is for you to scratch up; the second is there in case you wish to produce a final clean copy.


program
<id:teeny>
;
<id:teency>
assign
if
<id:x>
read
=
assign
print
<id:x>
<id:x>
<int:1>
<id:x>
<int:0>


program
<id:teeny>
;
<id:teency>
assign
if
<id:x>
read
=
assign
print
<id:x>
<id:x>
<int:1>
<id:x>
<int:0>

PROBLEM 5.

Add the Algol ’for’ loop to the the denotational description of Tiny. The Algol for loop has the following syntax:

Statement	-> ’for’ ’<identifier>’ ’:=’ List ’do’ Statement	=> ’for’
List	-> IntExp list ’,’	
IntExp	-> Expression ’..’ Expression	=> ’..’
	-> Expression	

For example:

for i:= 0, 8, 9, 8..19, 7..3 do output(i)

Note: ranges are intended to count ’upwards’ only. Complete the following:

CC[<’for’ <id:x> $IE_1 \ldots IE_n$ C>] =

PROBLEM 6.

Write an attribute grammar for Roman numerals. In case you don't remember how roman numerals work, the following rules should refresh your memory:

1) Roman numerals are composed of symbols I (one), V (five), X (ten), L (fifty), C (one hundred), D (five hundred), and M (one thousand).

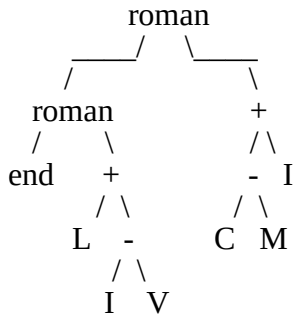
2) A smaller number placed in front of a larger number is subtracted from it. (e.g. IV means four). Only one such smaller number can be subtracted at a time (e.g. IIV does not mean 3).

3) A number placed after a greater or equal number is added to it (e.g. DC means 600, XIII means 13, and MMM means 3000).

Below is an AST grammar for roman numerals.

```
R → < 'roman' R E >
  →  'end'
E → < '+' E E >
  → < '-' E E >
  → 'M'              # 1000
  → 'D'              # 500
  → 'C'              # 100
  → 'L'              # 50
  → 'X'              # 10
  → 'V'              # 5
  → 'I'              # 1
```

Below is a sample AST, produced from the input string 'LIV , CMI'. Note that there are *two* roman numerals: 50+(5-1)=54, and (1000-100)+1=901. Notice the reversed order of the subtraction.



The ultimate goal is produce one attribute, a 'file' attribute, which will contain the values of all the roman numerals in the tree. Determine the following: ATT (attributes), SATT (synthesized attributes), IATT (inherited attributes).

Determine the attributes that are attached to each tree node, by specifying the functions S and I, as in the class notes. Then write the complete attribute grammar.

PROBLEM 7.

Show that the fixed-point finder $Y=\lambda F.(\lambda f.\, f\ f)(\lambda g.\, F(g\ g))$ is a fixed-point of the function $H = \lambda y.\, \lambda f.\, f(y\ f)$.

PROBLEM 8.

Suppose that you are given an AST and its standardized version ST.

a) Is it possible to uniquely destandardize the ST, and reproduce the AST? If so then write a destandardizer in pseudocode format for destandardizing a lambda expression to an RPAL AST. If it can't be done, explain why not.

b) Is it possible to uniquely deparse the AST, and produce the RPAL source ? If not, explain why not. If it can be done, discuss a strategy for producing the RPAL source. Illustrate it with pseudo-code if appropriate.

PROBLEM 9.

Describe the addition of the ‘‘loop--while’’ construct to your Tiny compiler. Describe the changes you would make to your parser by giving the pseudo-code of the relevant portion of your recursive descent parser. Show pseudo-code describing how your constrainer would analyze this construct, and show pseudo-code describing how your code generator would generate code for it.

The ‘‘loop--while’’ construct has the following form:

loop S while B : T repeat

Both S and T are statement; B is a boolean expression. S is executed at least once. After each execution of S, expression B is tested: if true, T and then S are executed, and B is examined again; if false, the iteration stops.

PROBLEM 10.

What is the output of the following RPAL program ? Explain the program's behavior in general. Hints: try the Is_Element function first; then the mystery function F. All the variable names are intended to be descriptive.

```
let Is_Element Number Tuple = Rec_Is_Element Number Tuple (Order Tuple)
        where rec Rec_Is_Element Number Tuple Index =
                    Index eq 0 -> false
                  | (Tuple Index) eq Number -> true
                  | Rec_Is_Element Number Tuple (Index-1)
in
let rec Rec_F Tuple Index =
        Index eq 0 -> nil
       | (let Result = Rec_F Tuple (Index-1) in
          (Is_Element (Tuple Index) Result -> Result
         | (Result aug (Tuple Index)))
         )
in
let F Tuple = Rec_F Tuple (Order Tuple)
in
Print ( F (1,2,3,2,4,5,4) )
```

PROBLEM 11.

Choose, among the two statements below, the one you believe is closest to the truth, and complete the statement you've chosen.

1) Attribute grammars are closer to the denotational approach than the operational approach, because . . .

2) Attribute grammars are closer to the operational approach than the denotational approach, because . . .

PROBLEM 12.

Suppose we were to change the description of the `binop` node in the denotational description of the CSE machine to each of the two descriptions shown below. In each case determine whether the new description gives different results from the description given in the course notes, and if so, give a sample RPAL program (in AST form) whose output will be altered by the change. Show your RPAL program's output before and after the change.

$$EE[\langle binop\ E_1\ E_2\rangle] = \lambda(e,o).(e,o) \Rightarrow EE[E_2] \Rightarrow (\lambda(v_2,o_2).(e,o_2) \Rightarrow EE[E_1] \Rightarrow (\lambda(v_1,o_1).(PE\ binop)\ v_1\ v_2\ (e,o_2)))$$

$$EE[\langle binop\ E_1\ E_2\rangle] = \lambda(e,o).(e,o) \Rightarrow EE[E_2] \Rightarrow (\lambda(v_2,o_2).(e,o_2) \Rightarrow EE[E_1] \Rightarrow (\lambda(v_1,o_1).(PE\ binop)\ v_1\ v_2\ (e,o)))$$

PROBLEM 13.

Extend the attribute grammar given in the class notes (for Tiny) by adding the ‘‘loop--while’’ construct. The ‘‘loop--while’’ construct has the following form:

```
loop S while B : T repeat
```

Both S and T are statements; B is a boolean expression. S is executed at least once. After each execution of S, expression B is tested: if true, T and then S are executed, and B is examined again; if false, the iteration stops.

PROBLEM 14.

Augment the denotational semantic description of Tiny with conditional expressions. Conditional expressions can be used anywhere ordinary expressions are used, and they are used to choose between two expressions, rather than between two statements. The syntax is that of the conditional expression in the C language. For example:

```
(y <= 12) ? 3 : 4
```

In this case, the value of the conditional expression will be either 3 or 4, depending on whether or not y is less or equal to 12. Whichever value is the case, either 3 or 4, is the resulting value of the expression. Conditional expressions can be nested arbitrarily. The ":" clause is required.

PROBLEM 15.

Draw a picture of the run-time environment, at the point marked "HERE" in the code shown below, for each of the following two techniques:

1) Static Links.

2) Displays.

```
program main;

       procedure A;
       begin {A}
             ...
       end;

       procedure B(procedure E);
       var x:integer;

             procedure C;
             begin {C}
                   x := 5;    <----------- HERE !
             end;

       begin {B}
             E;
             B(C);
       end;

begin {main}
       B(A)
end;
```

PROBLEM 16.

For each of the following two scenarios, produce an RPAL program whose CSE Machine evaluation has exactly what is given in the ENVIRONMENT column (no more, no less, and in the order given). For each scenario, you are to show:

(a) the RPAL program,

(b) its STANDARDIZED tree,

(c) its control structures.

Scenario A:

CONTROL	STACK	ENVIRONMENT
$e_0\ \gamma\ \lambda_1^x\ 1$	---	e_0= PE
$e_0\ \gamma$	${}^0\lambda_1^x\ 1$	
$e_0\ e_1\ \gamma\ \lambda_2^y\ 2$	---	$e_1=[1/x]e_0$
$e_0\ e_1\ \gamma$	${}^1\lambda_2^y\ 2$	
$e_0\ e_1\ e_2\ \gamma\ \lambda_3^z\ 3$	---	$e_2=[2/y]e_1$
$e_0\ e_1\ e_2\ \gamma$	${}^2\lambda_3^z\ 2$	
$e_0\ e_1\ e_2\ e_3$ nil	---	$e_3=[3/z]e_2$
$e_0\ e_1\ e_2\ e_3$	nil	
---	nil	

Scenario B:

CONTROL	STACK	ENVIRONMENT
$e_0\ \gamma\ \lambda_1^z\ \gamma\ \lambda_2^y\ \gamma\ \lambda_3^x\ 1$	---	e_0=PE
$e_0\ \gamma\ \lambda_1^z\ \gamma\ \lambda_2^y\ \gamma$	${}^0\lambda_3^x\ 1$	
$e_0\ \gamma\ \lambda_1^z\ \gamma\ \lambda_2^y\ e_1\ 2$	---	$e_1=[1/x]e_0$
$e_0\ \gamma\ \lambda_1^z\ \gamma\ \lambda_2^y$	2	
$e_0\ \gamma\ \lambda_1^z\ \gamma$	${}^0\lambda_2^y\ 2$	
$e_0\ \gamma\ \lambda_1^z\ e_2\ 3$	---	$e_2=[2/y]e_0$
$e_0\ \gamma\ \lambda_1^z$	3	
$e_0\ \gamma$	${}^0\lambda_1^z\ 3$	
$e_0\ e_3$ nil	---	$e_3=[3/z]e_0$
$e_0\ e_3$	nil	
---	nil	

PROBLEM 17.

Add a swap statement to the denotational description of Tiny. The swap statement is a double assignment among two variables (not expressions). For example, 'a :=: b' swaps the values of variables a and b. Specifically, complete the following:

```
CC[<:=: I1 I2>] =
```

"Implement" the swap statement, i.e. describe the changes necessary to the RPAL version of the denotational description of Tiny, in order to add the swap statement to the language.

PROBLEM 18.

Consider the following Tiny program:

```
program sum:
   assign s:=0;
   assign i:=1;
   while not(i=10) do
     assign s := s + read;
     assign i := i + 1;
   od;
   output s
end sum.
```

Using the grammar on page 90 of your notes, build the AST for this program. Place next to each tree node the necessary attributes. For most nodes, this means four inherited attributes (on the node's left), and five synthesized attributes (on the right). Fill in the actual values of ALL the attributes, according to the attribute grammar in your class notes, i.e. show the result of propagating the values through the functional graph. DO NOT depict the functional graph itself. Please adher to the convention adopted in class: the attributes are, from left to right: code, error, next, top, type.

PROBLEM 19.

Consider the following RPAL program:

```
let rec f x n i =
  (i eq 0) -> (x+n) |
  (n eq 0) -> (i ls 3) -> (i -1) | x
        |  (f x (f x (n-1) i) (i-1))
in f 2 3 1
```

The corresponding λ-expression is:

(λ f.f 2 3 1)(Y F), where
F = λ f.λ x.λ n.λ i.
(i eq 0) -> (x+n) |
(n eq 0) -> (i ls 3) -> (i -1) | x
| (f x (f x (n-1) i) (i-1))

Evaluate the λ-expression using β reductions and the fixed-point identity.

PROBLEM 20.

Consider the denotational semantics description of the CSE Machine. Make the necessary changes in the specification, to specify normal order of evaluation, instead of programming language order. You may disregard the situations involving recursion.

PROBLEM 21.

Add the ‘‘for’’ statement to the attribute grammar of Tiny. Model the ‘‘for’’ statement after the one in the C language. The AST grammar production is given below:

$$E \rightarrow \langle \text{'for'}\ E_1\ E_2\ E_3\ E_4 \rangle$$

If needed, specify any extensions to the instruction set of the toy stack machine for which we generate code. Develop first a sketch of the code you are generating for this construct.

PROBLEM 22.

Consider the denotational semantics description of Tiny, in its original, mathematical (non-RPAL) form. We added simple, parameter-less procedures to this language, as follows:

CC[<proc p C>] = λ(m,i.o).(Replace m p C, i, o)
CC[<call p>] = λ(m,i,o). m p eq $\underline{\perp} \rightarrow$ error | CC[m p] (m,i,o)

Describe the semantics of single-parameter functions, i.e.

CC[<fcn f x C>] =
EE[<call f E>] =

PROBLEM 23.

Add the unary auto-increment operators (++, from the C language) to the RPAL version of Tiny's denotational semantics. We'll need two names for them: 'prefix++' and 'postfix++'. Assume the operand of the auto-increment operator is a single name, i.e. the parser disallows auto-increment of anything but an identifier. To get you started,

```
...

| (E 1) @EQ 'prefix++' -> ...

...

| (E 1) @EQ 'postfix++' -> ...
```

PROBLEM 24.

A severe system crash takes place on your computer system, leaving you with an implementation of a Pascal-like language in executable form. The source code is gone forever. You remember that before the crash there were three executable compilers. One used copy-in,copy-out as the parameter passing mechanism; another other used pass by reference, and the third used pass-by-value. You don't know which compiler survived. The only difference between the three compilers was the parameter passing mechanism.

Write a (short) program in Pascal/C-like pseudo-code that will allow us to determine which compiler survived the crash. Specifically, write a program whose output will be 0 if copy-in,copy-out is used, 1 if pass by reference is used, and 2 if pass-by-value is used. Show a legible trace of the program's execution.

PROBLEM 25.

In the denotational semantics description of the CSE machine, we have the following definition of BB.

BB: ST $\rightarrow$ State $\rightarrow$ Result

BB = λast.λ(e,o).(e,o) => EE[ast] => (λ(v,o).v$\in$Boolean$\rightarrow$(v,o)|error)

Suppose we were to change the definition of BB above to the following:

BB = λast.λ(e,o).(e,o) => EE[ast] => (λ R.R 1$\in$Boolean$\rightarrow$(R 1,o)|error)

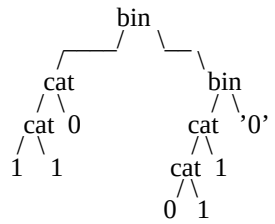
Describe the effect of this change in the denotational description, upon the semantics of the CSE machine. Give a sample RPAL program, in AST form, whose output will be altered by this change. Show your RPAL program's output before and after the change.

PROBLEM 26.

You are about to partially specify the semantics of the language of integer (no fractions) binary numbers. Below is the AST grammar for binary numbers.

```
B → < 'bin' E B >
  →  '0'
E → < 'cat' E D >
  →  D
D → '0'
  → '1'
```

Below is a sample AST, produced from the input string '110,011'. The numbers are 4+2=6, and 2+1=3.



The following are the semantic domains:

```
Num    = { 0, 1, 2, . . . }
Output = Num *   (tuples of numbers)
```

The following are the functionalities of the semantic functions:

```
EE: E→Num
BB:B→Output→Output
```

Fill in the following lines to complete the denotational semantic description of the language. Hints: the meaning of a binary number tree X, such as the one above, is BB [X] nil. The cascade of "bin" nodes is intended for BB to take the current output, augment it with value of the current binary number, and pass the new output along to the next "bin" node. A cascade of "cat" nodes is intended for EE to calculate the value of the binary number, from left to right.

```
EE ['1']              =
EE ['0']              =
EE [< cat E D > ]     =
BB [< 'bin' E B > ]   =
BB ['0']              =
```

PROBLEM 27.

In this course the in-depth topic has been specification of syntax and of semantics of programming languages. Write a short essay (maximum half a page) on how the study of this in-depth topic has changed (if at all!) the way you think about how programming language constructs work. Specifically, various chapters in the textbook contain a fairly large number of facts about programming language constructs. Explain how (or whether) your new knowledge of specification mechanisms (context-free grammars, operational semantics, denotational semantics, attribute gramamrs) affected your understanding of the issues as you read those chapters. List specific topics in those chapters, which you might have understood differently had you not become an expert in specification this semester.

PROBLEM 28.

Describe, with an example, the two principal methods of handling non-local variables in an imperative programming language. Cite one advantage, and one disadvantage, of each method.