HW3
50pts
Posted Friday, October 3
Due Thursday, October 16

Submit written part in `HW3Solutions.pdf` and code in `predictive_rec_descent.py` and `predictive_rec_descent.pl`. Submission size limit is 2.5MB.

Problem 1 (20pts). Consider the pseudocode with nested subroutines:

```
procedure main
     g : integer

     procedure B(a : integer)
          x : integer

          procedure A(x : integer)
               g := x

          procedure R(m : integer)
               write_integer(x)
               x /:= 2 -- integer division
               if x > 1
                   R(m + 2)
               else
                   A(m + 1)

          -- body of B
          x := a * a
          R(1)

     -- body of main
     B(3)
     write_integer(g)
```

a) (5pts) What does the program print under static scoping?

```
Output:  9
         4
         2
         6
```

b) (5pts) Show the frames on the stack when A has just been called assuming static scoping rules. Show the static and dynamic links of each frame, as well as the local variables and their values right after the assignment `g := x`. Explain how `A` finds `g`.

Start →

Procedure main	Procedure B	Procedure R	Procedure R	Procedure R	Procedure A
Static Link main	Static Link main	Static Link B	Static Link B	Static Link B	Static Link B
Dynamic Link main	Dynamic Link main	Dynamic Link B	Dynamic Link R	Dynamic Link R	Dynamic Link R
g= ?	a=3 x=9	m=1 x=4	m=3 x=2	m=5 x=1	x=6

c) (5pts) Now, what does this program print under dynamic scoping?

```
Output:  9
         4
         2
         1
```

d) (5pts) Explain how `R` finds `x` under dynamic scoping rules.

Under dynamic scoping, when `R` executes write_integer(`x`), it looks for `x` in this order up the call chain:

1. It's own frame - `R`'s local vars - if not found, then move onto
2. The caller's frame - `B`'s local vars - if not found, then onto
3. The caller's caller's frame - `main`'s local vars, and so on

Here, R does not have its own x. So it searches up the call chain and finds x in the B frame, which it then uses continuously to reference, modify, and print out.

Problem 2 (30pts). The grammar below generates boolean expressions in prefix form:

$$\begin{array}{lll} \textit{start} & \rightarrow & \textit{expr}\ \texttt{\$\$} \\ \textit{expr} & \rightarrow & \texttt{or}\ \textit{expr}\ \textit{expr} \mid \texttt{and}\ \textit{expr}\ \textit{expr} \mid \texttt{not}\ \textit{expr} \mid \texttt{id} \end{array}$$

a) (5pts) Write an attribute grammar (in pseudocode) to translate expressions into fully parenthesized infix form. For example, expression `and and a or b c d` turns into the following fully parenthesized expression `((a and (b or c)) and d)`.

1.	expr → or expr_1 expr_2	expr.val := "(" + expr_1.val + " or " + expr_2.val + ")"
2.	expr → and expr_1 expr_2	expr.val := "(" + expr_1.val + " and " + expr_2.val + ")"
3.	expr → not expr_1	expr.val := "(not " + expr_1.val + ")"
4.	expr → id	expr.val := id.s

b) (5pts) Now write an attribute grammar (in pseudocode again) to translate the expressions into *parenthesized* expressions in infix form *without redundant parentheses* assuming the standard convention: unary `not` has highest precedence, followed by `and`, followed by `or`, and `and` and `or` are left-associative. For example, the above expression turns into `a and (b or c) and d`. *Hint:* Assign a precedence attribute *prec* to operators and expressions. In part c) and part d) you will code your solution respectively in Python and in Prolog.

First, let's assign two synthesized attributes to the expressions/terminals:

1. `val` - the resulting infix string
2. `prec` - it's precendence (higher number means higher precedence)

Then, synthesize precedences to the operators/terminals(prec(operator))(Higher number means higher precedence):

$$\text{prec(id)} = 4 \quad \text{prec(not)} = 3 \quad \text{prec(and)} = 2 \quad \text{prec(or)} = 1$$

Now, let's write the attribute grammar with helper functions paren_L(E, p), paren_R(E, p), and paren_{un}(E, p) that adds parentheses to the left, right, or unary expression E if its precedence is lower than p.:

Helper functions:

$$\text{paren}_L(E, p) = \begin{cases} (E.\text{val}) & \text{if } E.\text{prec} < p, \\ E.\text{val} & \text{otherwise.} \end{cases}$$

$$\text{paren}_R(E, p) = \begin{cases} (E.\text{val}) & \text{if } E.\text{prec} \leq p, \\ E.\text{val} & \text{otherwise.} \end{cases}$$

$$\text{paren}_{\text{un}}(E, p) = \begin{cases} (E.\text{val}) & \text{if } E.\text{prec} < p, \\ E.\text{val} & \text{otherwise.} \end{cases}$$

CONTINUED ON NEXT PAGE

Attribute Grammer:

1.	expr $\rightarrow$ or expr_1 expr_2	expr.prec := 1 expr.val := $\text{paren}_L(\text{expr}_1, 1)$ + " or " + $\text{paren}_R(\text{expr}_2, 1)$
2.	expr $\rightarrow$ and expr_1 expr_2	expr.prec := 2 expr.val := $\text{paren}_L(\text{expr}_1, 2)$ + " and " + $\text{paren}_R(\text{expr}_2, 2)$
3.	expr $\rightarrow$ not expr_1	expr.prec := 3 expr.val := "not " + $\text{paren}_{un}(\text{expr}_1, 3)$
4.	expr $\rightarrow$ id	expr.prec := 4 expr.val := id.s