COMP 301 : PROGRAMMING LANGUAGES
PROBLEM SET 5

Due date: Dec 3, 2009 Friday 09:30 am

Please note:

- In this course, there is a 1 day late policy: You can still submit your homework within 24 hours past the deadline, but will lose 25% of your grade. Anything submitted after that will not be graded.
- Name your file as homework#_novellname.doc (you can submit docx files as well)
  - Example: hw1_bdevrim.docx
- Submit your homework to: F:\COURSES\UGRADS\COMP\COMP301\HOMEWORK
- For each problem set, your grade will be converted to a plus, half a plus or a minus, based on the following scheme:
  - 100 ≥ grade ≥ 75 ⇒ plus
  - 75 > grade ≥ 50 ⇒ half a plus
  - 50 > grade ≥ 0 ⇒ minus
- Following each problem set, there will be an in class quiz that covers the same material. You can bump up your homework grade if you manage to get a better score in the quiz. Grading for quizzes is the same as the problem sets.
- Any instance of cheating/plagiarism will be referred to the disciplinary committee. All involved parties (e.g., recipient AND the receiver of assistance) will receive an F as their final grade.

Problem 1:\nExtend the language so that a let declaration can declare an arbitrary number of variables, using the grammar

Expression ::= let (Identifier = Expression)* in Expression

As in Scheme’s let, each of the right-hand sides is evaluated in the current environment, and the body is evaluated with each new variable bound to the value of its associated right-hand side. For example,

```
let x = 30
in let x = -(x,1)
y = -(x,2)
in -(x,y)
```

should evaluate to 1.

Problem 2:\nIn PROC, procedures have only one argument, but one can get the effect of multiple argument procedures by using procedures that return other procedures. For example, one might write code like

```
let f = proc (x) proc (y) ...
in ((f 3) 4)
```

1 See EOPL, p.80, Exercise 3.20
2 See EOPL, p.72, Exercise 3.6
This trick is called *Currying*, and the procedure is said to be *Curried*. Write a Curried procedure that takes two arguments and returns their sum. You can write \( x + y \) in our language by writing \(- (x, - (0, y))\).

**Problem 3**: *Dynamic binding* (or *dynamic scoping*) is an alternative design for procedures, in which the procedure body is evaluated in an environment obtained by extending the environment at the point of call. For example in

```plaintext
let a = 3
in let p = proc (x) -(x, a)
a = 5
in -(a, (p 2))
```

the \( a \) in the procedure body would be bound to 5, not 3. Modify the language to use dynamic binding. Do this twice, once using a procedural representation for procedures, and once using a data-structure representation.

**Problem 4**: Extend the language above to allow the declaration of any number of mutually recursive unary procedures, for example:

```plaintext
letrec
  even(x) = if zero?(x) then 1 else (odd -(x,1))
  odd(x) = if zero?(x) then 0 else (even -(x,1))
in (odd 13)
```

\[
= (-
  (apply-procedure
   (value-of
    <<<double>>>
     [x=6_] (extend-env-rec
       double x <<<if zero?(x) ...>> ρ0))
   (value-of
    <<<-(x,1)>>
     [x=6_] (extend-env-rec
       double x <<<if zero?(x) ...>> ρ0)))
-2)
= (-
  (apply-procedure
   (procedure x <<<if zero?(x) ...>>>
     (extend-env-rec double x <<<if zero?(x) ...>> ρ0))
5_)
-2)
= ...
```

---

3 See EOPL, p.82, Exercise 3.28
4 See EOPL, p.84-85, Exercise 3.32