COMP 280 : Extra Credit Project
due: Friday, April 28, 2000

This extra credit project considers the value-flow analysis (and type checker) for Scheme programs that we discussed in class during the exam week. There are two stages to the project. If you want extra credit, you must complete at least the first stage (and have the code mostly working). You may earn additional extra credit points for completing the second stage. All work on this project must be submitted by the last day of classes.

Warning: This project is non-trivial. Don't expect to start the night before and make enough headway to earn any points. You will receive no points unless you have at least the entire first stage mostly working.

Honor Code Policy

The honor code policy for this project is similar to that for homeworks: you may discuss the project with other students, but you must write the final programs on your own. However, there are two additional requirements for the project:

1. You must clearly document who else you worked with on the project, and on what aspects (for example, "X and I developed our data definitions for programs together", or "Y helped me debug the following error in rule R"). Attach this information to the front of your project writeup. If you did the project entirely on your own, say so. Failure to mention who you worked with may result in no credit for the project.
2. Since this is for extra credit, I may ask you to come in and explain parts of your solution to me. These sessions will be individual, not with groups of students who worked together. Thus, make sure you understand how your program works before turning it in. If I'm not convinced that you understand what you turned in, you may receive little or no credit for the project.

What to Turn In

You should turn in printouts of all programs, including test cases and descriptive comments describing the rules in the program. You should turn in a clear description of the data representation that you use for programs as well. Your work should be neat and readable. You should also turn in a description of who else you worked with on the project (see the section on the honor code policy for this project).

Problem Definition

In this project, you will implement (in Prolog) the value-flow analysis that we discussed in class. This section gives the data definition for programs and describes the relations that your program must generate. First, the data definition:

A program is either

- a variable, or
- (**let** (*x exp*) *body*), where *x* is a variable, *body* is a program, and *exp* is one of
  - a constant (number or *empty*),
  - (*cons variable variable*),
  - (**lambda** *variable program*),

- (*first variable*),
- (*rest variable*),
- (*variable variable*) (function application),
- (+ *variable variable*), or
- (optional) (**if** (= *variable variable*) *program program*), where = operates on numbers only (not on lists).

Thus, example programs are

```
(let (a 3)
  (let (b 4)
    (let (c (cons a b))
      (let (d (first c))
        d))))
```

```
(let (f (lambda x x))
  (let (z 3)
    (let y (f z)
      y)))
```

The value-flow graph for a program indicates how values flow between variables. In the first example, 3 flows to a, 4 flows to b, (*cons a b*) flows to c, and a flows to d. In the second example, 3 flows to z, z flows to x and x flows to y. Each "flows" statement represents an edge in the value-flow graph.

A value-flow solver computes the transitive closure of a value-flow graph. Thus, the result of the solver is a mapping (also called an *environment*) from every variable in the input program to the values that the variable may take on during program execution. For the first example, the solver would produce mapping

$$\langle a, 3\rangle, \langle b, 4\rangle, \langle c, (cons\ a\ b)\rangle, \langle d, 3\rangle.$$

For the second example, the solver would produce mapping

$$\langle f, (\textbf{lambda}\ x\ x)\rangle, \langle z, 3\rangle, \langle y, 3\rangle, \langle x, 3\rangle.$$

Stage 1

Implement a value-flow solver in Prolog. Your program should define a rule *in_E(Var, Value)*, which returns true if the value Value can flow to variable Var (in_E stands for "in environment").

I suggest you proceed in the following steps:

1. Develop a Prolog representation for programs in the language defined earlier. I suggest you use lists as your main structuring mechanism. You may wish to show me your representation before proceeding, so you don't lose time due to a poor representation. If you want me to look at your representation, bring me a description of the representation and both example programs written in your representation.
2. Define a helper rule *is_prog(Program)*, which is true of all representations of programs in the program you wish to analyze. For example, if you wish to analyze the first example given earlier, the following statements should all evaluate to true (though with the programs written in your Prolog representation of programs).
   - is_prog((let (a 3) (let (b 4) (let (c (cons a b)) (let (d (first c)) d))))).

- is_prog((let (b 4) (let (c (cons a b)) (let (d (first c)) d)))).
- is_prog((let (c (cons a b)) (let (d (first c)) d))).
- is_prog((let (d (first c)) d)).
- is_prog(d).

3. On paper, develop the implications that determine when a variable and value should satisfy in_E. For example, in class we discussed the implication governing binding variables to numbers:

   $$\text{is\_prog((let (a 3) Body))} \rightarrow \text{in\_E(a, 3)}.$$

   You should develop one implication for each condition under which a value flows to a variable. Use the data definition for programs to guide which rules to develop. Each rule generally concerns one case of the program data definition. If your rules are spanning multiple cases, you're likely headed down the wrong track.

   Do NOT try to use the Scheme code/implementation that we discussed in class to figure out these rules. You will get horribly confused if you do this. The solver is far easier to implement in Prolog than in Scheme. You should only consider the part of the lecture where we talked about the implications defining where values flow in the program. As a guideline, my Prolog implementation of this stage requires roughly 50 lines of actual code (including the is_prog rule, but not including blank lines, comments, etc).

   Note: This is the hardest part of this stage. Writing the Prolog program should be reasonably easy once you figure out these rules.

4. Write and test the Prolog program for in_E. Your program does not need to do any error checking (*i.e.*, it can assume that it always receives correct inputs).

Stage 2

Use the value-flow generator to implement a type-checker for our language of programs. There are two subproblems to this stage. The first detects type errors. If you want a bit of a challenge, also try the second subproblem which detects the source of type errors.

1. Implement a rule *type_error(Var, Value, Exp)* which returns true if the value Value yields a type error for variable Var at expression Exp. For example, given the following program, *type_error(y, 3, (first y))* should return true.

```
(let (f (lambda x x))
  (let (z 3)
    (let y (f z)
      (let (a (first y))
        a))))
```

2. (Challenge) Implement a rule *type_error_source(Var1, Value, Var2)* which returns true if the program contains a type error for value Value of variable Var1, and Value *originated* at variable Var2. For the above example, *type_error_source(y, 3, z)* should be true, but *type_error_source(y, 3, x)* should not.