Computer Science 131, Spring 2001
Assignment 3: Formal Semantics

Out: Wednesday, February 7

Due: Wednesday, February 14, 11:00am

This assignment involves no programming. Your written answers must be given directly to the professor in class (or put under his office door, 1253 Olin). Typewritten solutions (e.g., using LaTeX or Framemaker) are strongly preferred. You may submit handwritten work only if it is clearly legible; *the graders have been instructed to mark as wrong any work they have trouble reading.*

If you submit a solution after the Wednesday morning deadline, be sure to mark it with the date and time of the handin so that the graders can count late days.

1 Mini-ML (50%)

1. **Dynamic Semantics**

   Show all of the state transitions required to evaluate the given programs to value. For each step, cite the number of the rule or rules needed to justify that step; the rules are shown in Appendix A. You may use "..." to avoid recopying chunks of code that do not change from one step to another, as long as your intent is completely clear.

   (a) Be careful with your substitutions!

```
let x be 3 in
  let f be (fix g(y) is x+y) in
    let x be 4 in
      f(x)
```

   (b)

```
let fact be
       fix g(y) is if y≤0 then 1 else y*g(y+(~1))
in
    fact 2
```

2. **Static Semantics**

   The inference rules defining the mini-ML type system are shown in Appendix B. Since the pieces of code used in this problem have no free variables, we can unambiguously talk about the type(s) of these expressions; if it helps, however, you can assume that the typing environment Γ is initially empty. You need not give the typing proofs for the first four parts.

   (a) What is the type of `fix f(x) is x+1` ?

   (b) What types can the expression `fix f(x) is x` be shown to have?

   (c) What types can the expression `fix f(x) is f(x+1)` be shown to have?

   (d) What types can the expression `fix f(x) is f(x)` be shown to have?

   (e) The following code evaluates to the value 3 according to the dynamic semantics.

```
let id be (fix f(x) is x) in
    if id(true) then id(3) else id(4)
```

   Can this be shown to have type `int` in the type system defined by the inference rules? How or why not?

2 SIL: A Simple Imperative Language (50%)

For this problem, you will be given a formal definition of execution for a very simple subset of a language like C.

The language is divided into *expressions*, which evaluate to a value as before, and *commands* which update variables and handle control flow. The only command that may not be self-explanatory is the `skip` command, which has no effect when executed. This is abstract syntax for the empty or null command usually represented the absence of a command, as when the C programmer writes `for (i=0; i<10; i++) ;` .

Values	v ::=	n | `tt` | `ff`
Expressions	e ::=	v | x | e`+`e | e`<=`e
Commands	c ::=	`skip`
		| x `:=` e
		| c`;` c
		| `if` e `then` c `else` c
		| `while` e `do` c

1. **Evaluation**

   Evaluation in this language is more complex than for mini-ML, because the state of the abstract machine must include the current values of each variable. (This was avoided in the case of mini-ML by substituting away variables as soon as their values were determined. This only worked because the values of a variable can never change.) We

do this by using what will be called a *memory* or a *store*. An arbitrary memory will be denoted M.

Memories are simply lookup tables that associate values with variable names. For example, a table stating that the variable `x` has value `5`, `y` has value `tt`, and `z` has value `ff` can be written as $(\texttt{x=5}, \texttt{y=tt}, \texttt{z=ff})$. The empty memory will be written $\emptyset$.

We don't care exactly how this lookup table is implemented; it doesn't matter as long as we can insert and lookup variables. (Memories are treated as an abstract data type.) If M is a memory than we will write $M(\texttt{x})$ to denote the value which M associates with the variable `x`. Also, we write $M, \texttt{x=}v$ to denote the memory that results by extening M with the information that `x` is now associated the value v.

We can now describe the program states in the dynamic sematics: a state is either a pair containing the command to be executed and the current memory, or else is just a memory. The final states are those containing just the memory.

The rules for the dynamic semantics are shown in Appendix C. One complication is that we must also define the evaluation of expressions, and the evaluation of an expression like `x+1` depends on the current memory (because you must know the current value of `x`). The first four rules define a relation $\langle e, M\rangle \Downarrow v$, to be read "the expression e in the memory M evaluates to the value v". This is what was called in class a "big-step" semantics for expressions, because the relation is directly between the initial expression and its final value. Convince yourself, for example, that $\langle \texttt{x} + 1, \texttt{x} = 5\rangle \Downarrow 6$. [We could have used a small-step semantics as in mini-ML for defining expressions; the big step semantics happens to be slightly more convenient here.]

The remaining rules define the (small-step) state transitions for commands. For example, we have

$$
\begin{array}{lll}
 & \langle \texttt{(n := 3; if n<=5 then n:=5 else skip)}, \emptyset\rangle & \\
\rightarrow & \langle \texttt{(if n<=5 then n:=5 else skip)}, \texttt{(n=3)}\rangle & \text{(by Rules 38, 37, 32)} \\
\rightarrow & \langle \texttt{n:=5}, \texttt{(n=3)}\rangle & \text{(by Rules 40, 35, 33, 32)} \\
\rightarrow & (\texttt{n} = 5) & \text{(by Rules 37, 32)}
\end{array}
$$

Show the steps (and cite the rules justifying each step) to completely evaluate

$$\langle \texttt{x := 0; while x<=0 do (skip; x := x+1)}, \emptyset\rangle.$$

2. **Concurrency**

Suppose we want to extend the language to allow multitasking, so that evaluation can work on more than one command at once. One way to do this is to add a new command form $c_1 \| c_2$, with evaluation rules as follows:

$$\frac{\langle c_1, M\rangle \rightarrow \langle c_1', M'\rangle}{\langle (c_1\texttt{;}\ c_2), M\rangle \rightarrow \langle c_1'\texttt{;}\ c_2, M'\rangle} \tag{1}$$

$$\frac{\langle c_2, M\rangle \to \langle c_2', M'\rangle}{\langle (c_1\texttt{;}\ c_2), M\rangle \to \langle c_1\texttt{;}\ c_2', M'\rangle} \tag{2}$$

$$\frac{\langle c_1, M\rangle \to M'}{\langle (c_1\texttt{;}\ c_2), M\rangle \to \langle c_2, M'\rangle} \tag{3}$$

$$\frac{\langle c_2, M\rangle \to M'}{\langle (c_1\texttt{;}\ c_2), M\rangle \to \langle c_1, M'\rangle} \tag{4}$$

This extention makes evaluation of commands non-deterministic; a single state may step to several possible states. What final states can the following command end up in? Briefly justify your answer.

$$\langle \texttt{(x:=0; x:=x+1)} \parallel \texttt{(x:=3; x:=x+5)}, \emptyset\rangle$$

3. **loop-while-repeat**

Suppose we wanted to replace the `do-while` loop in the SIL language with a new sort of looping construct that allows the exit test to occur anywhere within the loop:

$$\begin{array}{rcl} c & ::= & \ldots \\ & | & \texttt{loop}\ c\ \texttt{while}\ e\ c\ \texttt{repeat} \end{array}$$

The idea is that code like

```
loop
  c1
while e
  c2
repeat
```

will start by executing the command c_1. If e then evaluates to `ff` the entire loop terminates. If the expression evaluates to `tt`, then command c_2 is executed and the loop starts over again (executing c_1, followed by another loop test, etc.).

One nice property of this construct is that it is more general than the separate "while" and "do" loops found in languages like C or Java. Recall that in the concrete syntax that a programmer would use, it is common to express the nop-command `skip` with the "empty" command. Then by letting c_1 be the empty command we get the readable code

```
loop while e
  c2
repeat
```

which has the same behavior as the SIL command `while` e `do` c_2 and the C command `while (`e`)` c_2. Conversely, by letting c_2 be the empty command we get the readable code

```
loop
   c1
while e repeat
```

which has the behavior of the C command `do` c_1 `while (`e`)`. It is also occasionally convenient to have the case where c_1 and c_2 are both empty.

Show some inference rule or rules which define evaluation of this `loop-while-repeat` construct, matching the informal description given above. (There are several possible formulations.)

A Dynamic Semantics for Mini-ML

$$\frac{e_1 \longrightarrow e_1'}{e_1 \texttt{+} e_2 \longrightarrow e_1' \texttt{+} e_2} \tag{5}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1 \texttt{+} e_2 \longrightarrow v_1 \texttt{+} e_2'} \tag{6}$$

$$\frac{}{n_1 \texttt{+} n_2 \longrightarrow n_1 + n_2} \tag{7}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 \texttt{*} e_2 \longrightarrow e_1' \texttt{*} e_2} \tag{8}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1 \texttt{*} e_2 \longrightarrow v_1 \texttt{*} e_2'} \tag{9}$$

$$\frac{}{n_1 \texttt{*} n_2 \longrightarrow n_1 n_2} \tag{10}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1 \texttt{<=} e_2 \longrightarrow e_1' \texttt{<=} e_2} \tag{11}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1 \texttt{<=} e_2 \longrightarrow v_1 \texttt{<=} e_2'} \tag{12}$$

$$\frac{}{n_1 \texttt{<=} n_2 \longrightarrow n_1 \leq n_2} \tag{13}$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \longrightarrow \texttt{if } e_1' \texttt{ then } e_2 \texttt{ else } e_3} \tag{14}$$

$$\frac{}{\texttt{if tt then } e_2 \texttt{ else } e_3 \longrightarrow e_2} \tag{15}$$

$$\frac{}{\texttt{if ff then } e_2 \texttt{ else } e_3 \longrightarrow e_3} \tag{16}$$

$$\frac{e_1 \longrightarrow e_1'}{\texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 \longrightarrow \texttt{let } x \texttt{ be } e_1' \texttt{ in } e_2} \tag{17}$$

$$\frac{}{\texttt{let } x \texttt{ be } v_1 \texttt{ in } e_2 \longrightarrow e_2[x \rightarrow v_1]} \tag{18}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\, e_2 \longrightarrow e_1'\, e_2} \tag{19}$$

$$\frac{e_2 \longrightarrow e_2'}{v_1\, e_2 \longrightarrow v_1\, e_2'} \tag{20}$$

$$\frac{}{(\texttt{fix } f(x) \texttt{ is } e)\, v \longrightarrow (e[x \rightarrow v])[f \rightarrow (\texttt{fix } f(x) \texttt{ is } e)]} \tag{21}$$

B Static Semantics for Mini-ML

$$\frac{}{\Gamma \vdash n : \texttt{int}} \tag{22}$$

$$\frac{}{\Gamma \vdash \texttt{tt} : \texttt{bool}} \tag{23}$$

$$\frac{}{\Gamma \vdash \texttt{ff} : \texttt{bool}} \tag{24}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{+} e_2 : \texttt{int}} \tag{25}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{*} e_2 : \texttt{int}} \tag{26}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{<=} e_2 : \texttt{bool}} \tag{27}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : t} \tag{28}$$

$$\frac{}{\Gamma \vdash x : \Gamma(x)} \tag{29}$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x{:}t_1 \vdash e_2 : t_2}{\Gamma \vdash \texttt{let } x \texttt{ be } e_1 \texttt{ in } e_2 : t_2} \tag{30}$$

$$\frac{\Gamma, f{:}t_1 \rightarrow t_2, x{:}t_1 \vdash e : t_2}{\Gamma \vdash \texttt{fix } f(x) \texttt{ is } e : t_1 \rightarrow t_2} \tag{31}$$

C Dynamic Semantics for SIL

$$\frac{}{\langle v, M\rangle \Downarrow v} \tag{32}$$

$$\frac{}{\langle x, M\rangle \Downarrow M(x)} \tag{33}$$

$$\frac{\langle e_1, M\rangle \Downarrow n_1 \qquad \langle e_2, M\rangle \Downarrow n_2}{\langle e_1 \texttt{+} e_2, M\rangle \Downarrow n_1 + n_2} \tag{34}$$

$$\frac{\langle e_1, M\rangle \Downarrow n_1 \qquad \langle e_2, M\rangle \Downarrow n_2}{\langle e_1 \texttt{<=} e_2, M\rangle \Downarrow n_1 \leq n_2} \tag{35}$$

$$\frac{}{\langle \texttt{skip}, M\rangle \rightarrow M} \tag{36}$$

$$\frac{\langle e, M\rangle \Downarrow v}{\langle x \texttt{ := } e, M\rangle \rightarrow (M, x{=}v)} \tag{37}$$

$$\frac{\langle c_1, M\rangle \rightarrow \langle c_1', M'\rangle}{\langle (c_1 \texttt{;}\ c_2), M\rangle \rightarrow \langle c_1' \texttt{;}\ c_2, M'\rangle} \tag{38}$$

$$\frac{\langle c_1, M\rangle \rightarrow M'}{\langle (c_1 \texttt{;}\ c_2), M\rangle \rightarrow \langle c_2, M'\rangle} \tag{39}$$

$$\frac{\langle e, M\rangle \Downarrow \texttt{tt}}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, M\rangle \rightarrow \langle c_1, M\rangle} \tag{40}$$

$$\frac{\langle e, M\rangle \Downarrow \texttt{ff}}{\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, M\rangle \rightarrow \langle c_2, M\rangle} \tag{41}$$

$$\frac{\langle e, M\rangle \Downarrow \texttt{tt}}{\langle \texttt{while } e \texttt{ do } c, M\rangle \rightarrow \langle (c\texttt{; while } e \texttt{ do } c), M\rangle} \tag{42}$$

$$\frac{\langle e, M\rangle \Downarrow \texttt{ff}}{\langle \texttt{while } e \texttt{ do } c, M\rangle \rightarrow M} \tag{43}$$