DUE: Monday 27 March 2006 in class

PROBLEMS

Some problems require you to describe syntax or semantics of a language. Your syntactic/semantic descriptions should include thorough comments that explain the ideas behind each specification in clear technical English. The existence of a formalism to describe semantics does not relieve you of the responsibility to explain, any more than it does in writing code. When precise semantics is called for, divide your solution into parts in the following order: (1) syntax changes (2) semantic domain changes (3) semantic function changes. Under (1), describe any *contextual constraints* (static semantics) informally—there is no need to formalize these. Under (3), give signatures for all semantic functions and auxiliary functions introduced.

Normally, it should only be necessary to give the *changes* in (1), (2) and (3) above. Do not repeat syntax and semantics that has been given in the lecture or text, and that remains the same in describing your solution.

1. Increment and Decrement

Extend the imperative language IMP discussed in class (Example 3.6, p. 66 of the Watt text) to allow expressions to have side-effects via the following productions. These productions incorporate the post-increment and post-decrement operators, which should be taken to have the same meaning as they do in `C`.

```
Expression ::= Numeral
      | Expression + Expression
        ...
      | Identifier ++
      | Identifier --
```

HINT: Since expression evaluation now has side-effects, the signature of *evaluate* has now changed to

$$evaluate : \texttt{Expression} \rightarrow (\mathsf{Environ} \rightarrow (\mathsf{Store} \rightarrow (\mathsf{Store} \times \mathsf{Value})))$$

Extend the denotational semantics of this language to accommodate this change. You may assume some fixed evaluation order on subexpressions if you like, but state what that order is in your solution. (Be sure to study how your changes propagate through the rest of the semantics for this example, and include all changes that are necessary as a consequence.)

2. Extending IMP with Multiple Assignment

Watt text, Exercise 3.16, page 95, **part (a)**.

Organize your semantic functions for maximum clarity and economy, using **let** $\cdots$ **in**.

HINT: To approach semantics like this, involving a list of syntactic objects, one approach is to construct an abstract syntax that "breaks up" the list of Identifiers, using (a) left recursion or (b) right recursion on a new nonterminal like "`MultAssign ::= ... MultAssign ... | Assign`" Auxiliary semantic functions that are driven by this recursion may become necessary.

To take the semantics of `Numeral`, for example, we would have with this approach

```
Numeral ::= Numeral Digit | Digit
```

$$val[\![ND]\!] = 10 * val[\![N]\!] + val[\![D]\!]$$
$$val[\![0]\!] = 0$$
$$\cdots$$

Do not use constructs with ellipses in the list, like $val[\![d_1, d_2, \cdots, d_n]\!] = \cdots$ since they do not indicate how the list syntax drives the semantics as the parse tree is walked.

3. Extending IMP with Multiple Assignment—Reprise

Watt text, Exercise 3.16, page 95, **part (b)**.

Organize your semantic functions for maximum clarity and economy, using **let** $\cdots$ **in**.

Same comments as previous problem.

4. Extending IMP with Sequential Declarations

Watt text, Exercise 3.17, page 95.

5. A Fixpoint Combinator in Scheme

The power function $pow = \lambda n \,.\, 2^n$ may be thought of as the fixed point ("fixpoint") $pow = \tau\, pow$ of the functional

$$\tau = \lambda f \,.\, \lambda n \,.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}f(n-1)\ .$$

(Recall we often write function application as $f\ n$ or $(f\ n)$ instead of $f(n)$.)

The "call-by-value" or "applicative order" fixpoint combinator (fixpoint functional) can be defined by

$$\mathbf{Y}_{cbv} = \lambda f \,.\, (\lambda x \,.\, \lambda y \,.\, f(xx)y)(\lambda x \,.\, \lambda y \,.\, f(xx)y)$$

$\mathbf{Y}_{cbv}$ is much more than a theoretical artifact—it is capable of actually producing the fixpoint function of *any* functional like τ above. To see this, let us *evaluate* $(\mathbf{Y}_{cbv}\tau)$ at the value 2. In other words, we perform reductions until an irreducible (normal) form is reached. To save repetition, it is helpful to write Z for the string $(\lambda x \,.\, \lambda y \,.\, \tau(xx)y)$. Also remember that PQR is short for $(PQ)R$.

$$\begin{aligned}
(\mathbf{Y}_{cbv}\tau)2 &\rightarrow\ (ZZ)2\\
&\rightarrow\ (\ (\lambda x \,.\, \lambda y \,.\, \tau(xx)y)Z\ )2\\
&\rightarrow\ (\lambda y \,.\, \tau(ZZ)y)2\\
&\rightarrow\ \tau(ZZ)2\\
&\rightarrow\ (\lambda f \,.\, \lambda n \,.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}f(n-1)(ZZ))2\\
&\rightarrow\ (\lambda n \,.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}(ZZ)(n-1))2\\
&\rightarrow\ \textbf{if}\ 2 = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}(ZZ)(2-1)\\
&\rightarrow\ 2{*}(ZZ)1\\
&\rightarrow\ \vdots\\
&\rightarrow\ 2{*}2{*}(ZZ)0\\
&\rightarrow\ 2{*}2{*}(\ (\lambda x \,.\, \lambda y \,.\, \tau(xx)y)Z\ )0\\
&\rightarrow\ 2{*}2{*}(\lambda y \,.\, \tau(ZZ)y)0\\
&\rightarrow\ 2{*}2{*}\tau(ZZ)0\\
&\rightarrow\ 2{*}2{*}(\lambda f \,.\, \lambda n \,.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}f(n-1)(ZZ))0\\
&\rightarrow\ 2{*}2{*}(\lambda n \,.\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}(ZZ)(n-1))0\\
&\rightarrow\ 2{*}2{*}\ \textbf{if}\ 0 = 0\ \textbf{then}\ 1\ \textbf{else}\ 2{*}(ZZ)(0-1)\\
&\rightarrow\ 2{*}2{*}1\\
&\rightarrow\ 4
\end{aligned}$$

This certainly provides evidence that, in fact, $\mathbf{Y}_{cbv}\tau \equiv pow$, the power function, aka $\lambda n \,.\, 2^n$.

(a) Since Scheme is not strongly typed, you can *define* $\mathbf{Y}_{cbv}$ in Scheme. Of course, τ can also be defined, e.g.,

```
(define tau
  (lambda (f)
    (lambda (n)
      (if (= n 0)
          1
          (* 2 (f (- n 1)))
      )
    )
  )
)
```

Define $\mathbf{Y}_{cbv}$ in Scheme and show a test of

```
((Ycbv tau) 5)
```

Trace evaluations of `Ycbv` and `tau` to see if they agree with what would be expected by hand computation. *HINT:* the `(trace ... )` facility is useful here.

(b) Pick another simple recursive definition besides the power function. Make it define a function on some type other than the integers. Construct the functional τ for this application from the right-hand-side of the function definition. Test `((Ycbv tau) arg)` to verify that the fixpoint function is being correctly computed.

(c) Does the following fixpoint combinator, often called the "normal order fixpoint combinator", compute the correct fixpoint function in Scheme?

$$\mathbf{Y}_{cbn} = \lambda f . (\lambda x . f(xx))(\lambda x . f(xx))$$

Construct it and experiment on the `tau` of part (a). If it does, demonstrate that it works by thorough testing. If it does not, find out why not. Hand computations may be helpful along with flowtracing the interpreter. State your conclusions clearly.