Lecture 15: Assignment

Assignment Operator

• Corresponds to EOPL §3.7
• Assignment is a fundamental operation in practical PLs
• Even “purely” functional languages need mechanisms for saving mutable state
• In this lecture, we will extend our PL to include the assignment operator for variables

Referential Transparency

• The fragment of Scheme we studied is purely functional.
• Definition: A function is referentially transparent if, given the same arguments, it always returns the same value.
• Referential transparent programs are easy to reason about.
• Side effect changes the global state of the system, and thus destroys referential transparency.
• e.g., variable assignment
• Purely functional programming enforces referential transparency by forbidding variable assignment.

• Once a variable receives its binding, its value never changes.

➡ Binding vs Assignment

• Definition: a **binding** is an association of a name to a value
• e.g., lambda, let, cases, . . .
• Assignment: changes the value of an existing binding
• e.g., set!, . . .
• Assignment operator in Scheme

  (set! var exp)

  1. Evaluate expression exp.
  2. Assign the value of exp to variable var.

• Examples:

  (set! x (+ x 1))
  (set! y (* y x))
Sequencing Operator

• NOTE: We will implement later but need functionality now!
• Suppose we are to execute the following in sequence:
  (set! x (+ x 1))
  (set! y (* y x))
• and then return the value of \( y \).
• Given the fragment of Scheme we have learned so far, we are forced to write:
  (let ((dummy (set! x (+ x 1))))
   (let ((dummy (set! y (* y x))))
    y))
• where \texttt{dummy} is a dummy variable

 ✦ Very Ugly!

• Instead we need a sequencing operator which evaluates expressions in a predictable (left-to-right) order
• Essential for programs with side-effects (assignment)
• Unnecessary for purely functional PLs. Why?
Sequencing Operator in Scheme

\(\text{(begin \exp1 \exp2 \ldots \expn)}\)

- Evaluate expressions \(\exp1, \exp2, \ldots, \expn\) in the order they appear, returning the value of \(\expn\).
- Example again:

\[
\text{(begin}
\quad \text{(set! x (+ x 1))}
\quad \text{(set! y (* y x))}
\quad y
\text{)}
\]
Iteration Operator

Introduction

• Another useful side-effect operator in Scheme
  (for-each proc list )

• Arguments:
  1. proc : a one-argument procedure
  2. list : a list of arguments

• Semantics: Procedure proc is applied to each member of list in the order the arguments appear in list.

• The result of for-each procedure application is undefined.

• Example: Revisiting list-sum operator
  (define list-sum
   (lambda (L)
    (let ((sum 0))
     (begin
      (for-each (lambda (x) (set! sum (+ sum x))) L)
     sum)))

• Like map, for-each can be applied to multiple list arguments.
Applications of Variable Assignment

- Sharing of variables is useful in practical applications
- Reduction of parameter passing between modules
- Useful examples:
  1. I/O procedures (cin/cout) in C
  2. Encapsulated state: Protection of data integrity
  3. Destructive update: Circular data structures
- Example: counters again

> (define counter
   (let ((x 0))
     (lambda ()
       (begin
         (set! x (+ x 1))
         x))))

> (counter)
1
> (counter)
2
Denoted versus Expressed values (revisited)

► Review

• See EOPL §3.1 (p.71)

• *Expressed values*: values that can be returned from expressions.

• *Denoted values*: bindings of variables in environments.

• So far the set of expressed values and the set of denoted values are the same.

• When we first built the interpreter, we had only numbers:

  Expressed Values = Number
  Denoted Values   = Expressed Values

• When you added lists, these became new expressed values:

  Expressed Values = Number + List
  Denoted Values   = Expressed Values

• When we added functions, they too became new expressed values:

  Expressed Values = Number + List + Procval
  Denoted Values   = Expressed Values
Variable Assignment

- We will need 3 steps:
  1. A new value type, reference, that represents the address of a memory store.
  2. Function apply-env will retrieve references rather than stored values.
  3. Interpreter extensions so that:
     - Variables evaluate to values stored at the locations addressed by references.
     - Assignment stores values into locations addressed by references.

New Denotation

Expressed Value = Number + ProcVal
Denoted Value = Ref(Expressed Value)
Syntax for Assignment Operator

- The BNF for variable assignment is
  \[ \text{<expression>} := \text{set <identifier> = <expression>} \]
- We add the following line to our grammatical specification:
  \[(\text{expression} ("set" \text{identifier} ":=" \text{expression}) \text{varassign-exp})\]
- Examples
  
  \[
  \begin{align*}
  \text{set x = add1(x)} \\
  \text{let z = 20} \\
  \quad \text{in let temp = set z = 19} \\
  \quad \text{in z}
  \end{align*}
  \]
  
  - Here's how to read this let expression:
    
    Let the variable "z" be 20 in evaluating the following expression:
    Let the variable "temp" be the result of setting "z" to 19 in the following expression:
    Evaluate z
Abstract Datatype for Set operator

- The abstract syntax generated by SLLGEN for variable assignment is a variant of expression:

  (varassign-exp
   (id symbol?)
   (rhs-exp expression?))

- Approach:
  1. Evaluate the rhs-exp to get the expressed value (called an \textit{R-value}).
  2. Find the denoted value for the variable id in the environment (called an \textit{L-value}).
  3. Put the result of the rhs-exp into the reference location given by the denoted value.
Implementing References

Implementation

• Remember that environments are implemented using vectors to store expressed values
• We can build a reference type that points to a particular vector and contains an index into that vector.
• In C, we would simply use a pointer to type (eg- *<type>)
  
  (define-datatype reference reference? 
   (a-ref 
    (position integer?) 
    (vec vector?)))

• Dereferencing a variable involves looking up the expressed value in the specified vector at a particular position

  (define deref 
   (lambda (ref) 
    (cases reference ref 
      (a-ref (pos vec) (vector-ref vec pos))))))
• Assignment to a variable reference involves looking up the memory location of the expressed value and changing it

(define setref!
  (lambda (ref val)
    (cases reference ref
      (a-ref (pos vec) (vector-set! vec pos val))))))
Changes to Environment code

- We need to add a new function apply-env-ref to our environment implementation
- Returns a reference to where the variable is stored in memory (denoted value).
- Our current apply-env returns the expressed value

```
(define apply-env-ref
  (lambda (env sym)
    (cases environment env
      (empty-env-record ()
        (eopl:error 'apply-env-ref "No binding for ~s" sym))
      (extended-env-record (syms vals env)
        (let ((pos (rib-find-position sym syms)))
          (if (number? pos)
            (a-ref pos vals)
            (apply-env-ref env sym))))))
```

-
Implementing Assignment

• Now that we have the necessary machinery in place for references
• Let’s extend eval-expression for our new varassign-exp variant

(define eval-expression
 (lambda (exp env)
   (cases expression exp
     (lit-exp (datum) datum)
     (var-exp (id) (apply-env env id))
     (varassign-exp (id rhs-exp)
       (begin
         (setref!
           (apply-env-ref env id)
           (eval-expression rhs-exp env))
         1))
   ...))

• Notice that we call apply-env-ref to get a reference to the id, then we evaluate the right-hand side of the assignment (rhs-exp), then we use setref! to change the referenced location to contain the new expressed value.