

CS 310 - Program 1

due Thursday, October 29th

1 Theory

1.1 Denotational Semantics

A number of systems have been devised to formalize the semantics of imperative programming languages. We have already seen how attribute grammars can be used to express the static semantics of programs. In a compiler the dynamic semantic analysis of a program takes place in the code generation phase. This process of converting syntactic structures to sequences of low-level instructions is the general idea behind *operational semantics*. The problem with operational semantics is that it is very hard to formalize mathematically. One alternative is *axiomatic semantics* which expresses the meaning of programs based on logical preconditions and postconditions. The third alternative is *Denotational semantics*.

As we learned in class, denotational semantics translates statements and expressions in an imperative programming language to a set of functions. These functions take a set of bindings that reflects the state of the variables *before* the execution of the statement as an argument, along with the statement to execute, and return a new set of bindings that reflects the state of the variables *after* the execution of the statement. Assignment statements, which have side-effects in an imperative language, simply result in a new set of bindings that includes the binding implied by the assignment. Variable references in expressions can be resolved by searching the set of bindings for an entry corresponding to the given variable. Statement lists and loops are implemented using recursion. Implementing local scope, break statements, and exceptions are considerably more complex and will not be part of this assignment.

1.2 Function Definitions

In this assignment you will implement functions for a `while` loop, an `if` statement, a `do` construct for statement lists, and an assignment statement. The functions are defined in pseudo-code as follows:

```
eval-assign( id = expr, bindings )
  return bindings.insert( id, eval-expr( expr, bindings ) )

eval-do( do stmt stmt-list, bindings )
  if expr-list is empty
    return eval-stmt( stmt, bindings )
  else
    return eval-do( do first-of(stmt-list) rest-of(stmt-list),
                    eval-stmt( stmt, bindings ) )

eval-if( if bool-expr then stmt, bindings )
  if eval-expr( bool-expr, bindings ) = true
    return eval-stmt( stmt, bindings )
  else
    return bindings

eval-while( while bool-expr stmt, bindings )
  if eval-expr( bool-expr, bindings ) = true
    return eval-while( while bool-expr stmt,
                      eval-stmt( stmt, bindings ) )
  else
    return bindings
```

2 Implementation

You will implement these functions in Scheme (which you will find is very well suited to such functions). I have already implemented the functions that correspond to expressions (which return values instead of bindings). The only one of these that you need to be concerned about is `eval-expr` which is needed by `eval-assign` to get a value to assign to a variable and by `eval-if` and `eval-while` to evaluate conditions. I have also implemented a general `eval-stmt` functions which is needed by `eval-do`, `eval-if`, and `eval-while` to evaluate the statements contained within conditional, repetition, and sequential control structures. The `eval-stmt` uses a set of predicate functions to determine what kind of statement it is evaluating, and based on the result it passes the statement to the corresponding function, namely one of the four functions you will be writing. Finally, I have implemented an `eval-program` function which takes a list of statements and a variable name, executes the statements, and returns the entry in the bindings for the variable.

2.1 Source Language

The source language that these functions will be interpreting is kind of a cross between C (infix operators, imperative control structures) and Scheme (parenthesized expressions). Here are some examples:

1. Literal Expressions: these can be numbers or Booleans such as 3, 4.5, or `true`.
2. Non-literal Expressions: these look like expressions in C or Java, except they must be fully parenthesized. Some examples:
 - `(x < 5)`
 - `((x < 20) and (y < 10))`
 - `(x + (y * z))`
3. Assignment Statements: these look like assignments in C or Java, except that instead of ending in a semicolon, they are contained in parentheses:
 - `(x = 5)`
 - `(y = (x + 1))`
 - `(i = (i + 1))`
4. Sequential Statements: A sequential statement is a parenthesized list whose first element is the `do` keyword and whose remaining elements are statements:
 - `(do (x = 1))`
 - `(do (x = 5) (y = (x + 1)) (z = (x + y)))`
5. Conditional Statements: A conditional statement is a parenthesized list with four elements: the `if` keyword, a Boolean expression, the `then` keyword, and a statement to be executed if the Boolean expression evaluates to `true`:
 - `(if (x < 5) then (y = 2))`
 - `(if (z >= 0) then (do (x = 5) (y = 7)))`
6. Repetition Statements: A repetition statement is a parenthesized list with three elements: the `while` keyword, a Boolean expression, and a statement to be repeatedly executed if the Boolean expression evaluates to `true`:
 - `(while (i < n) (do (total = (total + i)) (i = (i + 1))))`
 - `(while c (do (i = (i + 1)) (if (i >= 10) (c = false))))`

A program to implement an iterative factorial function might look like this:

```
(
  (i = 1)
  (n = 10)
  (total = 1)
  (while (i < n)
    (do
      (total = (total * i))
      (i = (i + 1))
    )
  )
)
```

Once you have your functions implemented you can test this program with the `eval-program` function:

```
> (eval-program '(i = 1)
                (n = 10)
                (total = 1)
                (while (i < n)
                  (do
                    (total = (total * i))
                    (i = (i + 1))
                  )
                )
                'total)
(total 362880)
```

Note that you must quote both the program and the variable you are looking up with a single quote. If you wish to test your individual functions (which you should) you will need to pass a set of bindings along with your expression:

```
> (eval-assign '(i = 1) '())
((i 1))

> (eval-if '(if true then (i = 1)) '())
((i 1))

> (eval-if '(if (x < 5) then (i = 1)) '((x 3)))
((x 3) (i 1))

> (eval-if '(if (x < 5) then (i = 1)) '((x 5)))
((x 5))
```

```

> (eval-do '(do
            (total = (total * i))
            (i = (i + 1))
          )
      '((i 1) (n 10) (total 1)))
((i 2) (n 10) (total 1))

> (eval-while '(while (i < n)
                    (do
                      (total = (total * i))
                      (i = (i + 1))
                    )
                )
      '((i 1) (n 10) (total 1)))
((i 10) (n 10) (total 362880))

```

2.2 Helper Functions

In order to make your job easier (and your code easier to read) I have implemented several helper functions:

- **first-of, rest-of, second-of, third-of, fourth-of:** These are just nicer names for the list deconstruction functions `car`, `cdr`, `cadr`, `caddr`, and `caddr` respectfully. They will be helpful in pulling out statements and expressions from the statements passed to your functions.
- **add-binding:** This will insert a binding of a variable to a value into a set of bindings (implemented as an a-list). It will replace the current binding of the variable if one exists or insert a new binding otherwise.
- **lookup-binding:** This will search the bindings for an entry matching a given variable. If it finds one, it returns the binding. If no binding matches, it returns `undef`. You shouldn't need this function, as it only comes up when a variable appears in an expression, which has already been implemented.

2.3 What to Hand In

E-mail the `denotational.scm` file with your statement evaluation functions and any other functions you write added to the end. Please also hand in a printed copy of `denotational.scm` in class on the due date.