MP 3 - Closures, References and Dynamic Scoping

CS421 Agha - Spring 2015

rev. 1.0

Introduction

Cooperation: You may work in pairs for this MP ***only***.

Assigned: February 28, 2015
Due: 11:59pm, March 14, 2015
Outline: This assignment covers material up to chapter 4 of the EOPL book, 3rd edition.
Grading: This assignment will account for 8% of your grade.

Submission and academic honesty instructions can be found here:

`https://wiki.cites.illinois.edu/wiki/display/cs421sp15/Assignments?src=contextnavchildmode`

Pay special attention to the policies regarding the use of other people's code.

Problems

In the grammars below, terminals are in courier font, for example, `lambda`. Nonterminals are italicized, as in *Expr*. Parentheses are literals (i.e. part of the specified grammar), while braces are part of the meta-language we use to describe the grammar. We use the * symbol to denote zero or more repetitions; for example, {(*Expr Var*)}* means zero or more occurrences of (*Expr Var*), including the parentheses. When it is clear from context, we omit the braces, so that *Var** means zero or more repetitions of *Var*.

Note: In the examples that follow, the input is assumed to be a string of characters. However, to make the examples cleaner, we omit the call to the interpreter. For example, we write

```
> x
'undefined
```

when we really mean

```
> (interpret "x")
'undefined
```

and (just another example) we write

```
> let x = 5 in
>   if =(x , 5) then 0 else 1
1
```

when we really mean

```
> (interpret "let x = 5 in if =(x , 5) then 0 else 1")
1
```

Assume the language given by the following grammar:

Expr	::=	`let` { *ID* = *Expr* }* `in` *Expr*
	|	`letrec` { *ID* = *Expr* }* `in` *Expr*
	|	`proc` ({*ID*}*) *Expr*
	|	(*Expr* {*Expr*}*)
	|	`newref` (*Expr*)
	|	`set` *ID* *Expr*
	|	`begin` *Expr* {; *Expr*}* `end`
	|	`if` *Expr* `then` *Expr* `else` *Expr*
	|	*ArithmeticOp* (*Expr* {, *Expr*}*)
	|	*ComparisonOp* (*Expr* , *Expr*)
	|	*Integer*
	|	*ID*
	|	`true`
	|	`false`
	|	`undefined`

Here, *ID* represents identifiers, that is, names of variables and procedures. You may assume they are simply strings of characters; *ArithmeticOp* denotes the usual +, -, *, / operators; *ComparisonOp* denotes comparison operators <, >, =.

Expressions between `begin` and `end` (and separated by semi-colons) are evaluated in sequence:

```
> let x = newref(1) in
>    begin
>      set x 2;
>      x
>    end

2
```

A `newref` expression creates a new reference with the specified value. A `set` changes the value of an already defined reference to the provided value. When used as an R-value, a reference is automatically dereferenced. In the example below, `y` is not bound to a reference (contrary to `x`). Instead, it is bound to the *value* provided by *dereferencing* `x`.

```
let x = newref(1) in
  ...
  let y = x in ...
```

Moreover, the value of a `let` expression is the value of its body (the expression following the `in` part). Also, the value of a `begin` ... `end` block is the value of the last expression in the sequence. Consider the example below, where the value of `x` in the end is 2:

```
> let x =
>   let y = newref(1) in
>     begin
>       set y 2;
>       y
>     end in x

2
```

References allow us to write functions with side-effects. In the example below, `x` is changed by `f` and this change is visible outside its scope:

```
> let x = newref(1) in
>   let f = proc (y) set y 2 in
>     begin
>       (f x);
>       x
>     end

2
```

Both `proc` and `let` accept multiple arguments, as in

```
> let f = proc (x y) +(x, y)
>     g = proc (x y z) +(x, y, z) in
>       (f (g 1 2 3) 1)

7
```

The semantics of the above is exactly the same as the following, using currying:

```
> let f = proc (x) proc(y) +(x, y) in
>   let g = proc (x) proc (y) proc (z) +(x, y, z) in
>     ((f (((g 1) 2) 3)) 1)

7
```

One can also use `set` to assign a process to a reference. For instance:

```
> let f = newref( proc (x y) +(x, y) ) in
>   begin
>     set f proc (x y) -(x, y);
>     (f 5 1)
>   end

4
```

What happens above is that initially, `f` is bound to a *reference* to an addition function, which is then changed to a subtraction function before it is finally called.

Because of the existence of side-effects, evaluation order matters. This language will employ left-to-right evaluation:

```
> let x = newref(1)
>     g = proc (x) begin set x 5; x end
>     h = proc (x) begin set x +(x, 7); x end
>     f = proc (x y) +(x, y) in
>       (f (h x) (g x))

13
```

Notice that `g` is *defined* before `h`. However, the *call* to `h` happens before `g`, and hence its side-effects take place first. As a result, `h` sets `x` to $1 + 7$, and returns 8. Then, `g` is called and sets `x` to 5, returning that value. So the values that the call to `f` sees are 8 and 5, hence the result, 13.

To clarify this idea, if one were to dereference `x` after the above code finishes, the value we would get is going to be 5. But if we reverse the order of calls:

```
> let x = newref(1)
>     g = proc (x) begin set x 5; x end
>     h = proc (x) begin set x +(x, 7); x end
>     f = proc (x y) +(x, y) in
>       (f (g x) (h x))

17
```

then the value of `x` after execution will be 12 instead, because `g` is called before `h`.

Procedures can be returned as values, as in the following example:

```
> let x =
>   let inc = proc (x) +(1, x) in inc
>   in
>     (x 5)

6
```

What happens in this example is: the `let inc ... in inc` expression returns `inc` which is bound to the defined procedure. The outermost `let x = ...` receives this value, hence binding `x` to the same procedure. The last `in` part uses this binding and calls the procedure with 5 as the argument. Do not be confused by the use of `x` in two places, the above is exactly equivalent to

```
> let f =
>   let inc = proc (x) +(1, x) in inc
>   in
>     (f 5)

6
```

Another example on the same topic:

```
> let g =
>   let counter = newref(0) in
>     proc (dummy)
>       begin
>         set counter +(counter, 1);
>         counter
>       end
>    in
>     let a = (g 11) in
>       let b = (g 11) in -(a, b)

-1
```

Now `g` is bound to the value returned by `let counter =...` which returns a procedure. The body of this procedure ignores its argument (`dummy`), increases `counter` by one, and returns the new value. Now the first time `g` is called, it returns 1 which gets bound to `a`. The second time `g` is called, the counter is already at 1, which means it gets increased to 2, then this value is returned and bound to `b`. Hence the result is `-(1, 2)`. Notice that in both calls, 11 is bound to `dummy` and hence ignored.

The symbol `undefined` is allowed as the value of an expression, with the intention to be used in case of errors:

```
> x
'undefined

> if 5 then 0 else 1
'undefined

> let x = undefined in x
'undefined
```

The value of a `set` expression is also `undefined`.

```
> let x = let y = set x 1 in y in x
'undefined
```

The above example works in a recursive manner: first, variable `x` is created, and to compute its value, we recursively evaluate `let y = set x 1 in y`. That results in the creation of variable `y`, which is bound to the value of `set x 1`. Although this sets `x` to 1, the value *returned* is actually `undefined`. As a result, the `let y ... in y` part evaluates to `undefined` and this gets bound to the outermost `let x` variable, making the value of the whole expression (the `let x ... in x` part) also `undefined`.

The value of boolean expressions is `true` or `false`, which are terminal symbols in the grammar:

```
> let x = 5 in =(x, 5)
'true

> let x = newref(1) in =(x, 1)
'true
```

Notice how in the second example above, `x` appears as an R-value, and thus is dereferenced before compared to the value 1.

Furthermore, we allow mutually recursive procedures:

```
> letrec factorial = proc (x) if =(x , 0) then 1 else *(x , (factorial -(x , 1))) in
>   (f 5)

120

> letrec f = proc (x) if =(x , 0) then 1 else *(x, (g -(x, 1)))
>        g = proc (x) if =(x , 0) then 2 else *(x, (f -(x, 1))) in
>   (f 3)

12
```

1. Write an interpreter for this language as specified above, with *static* scoping and *eager* evaluation:

   ```
   > let x = 1 in
   >   let f = proc (y) +(x, y) in
   >     let x = 2 in
   >       (f 5)

   6
   ```

   In this example, the value of `x` at the time the `proc` expression is evaluated is 1, hence the body of the procedure is always bound to `+(1, y)`. That is true even when calling `f`, i.e. at the evaluation of `(f 5)`. Even though the previous line binds `x` to 2, that is a different `x` (in a different scope).

   Note that all of the examples in the description of the language indeed use static scoping and eager evaluation. Name this interpreter `static-interpreter`.

2. Write an interpreter for the same language, again with eager evaluation, but *dynamic* scoping:

   ```
   > let x = 1 in
   >   let f = proc (y) +(x, y) in
   >     let x = 2 in
   >       (f 5)

   7
   ```

   Here, when `f` is called, the value of `x` in its body is 2 because of the preceding `let` expression. As a result, the computed expression is `+(2,5)`.

   Name this interpreter `dynamic-interpreter`.

3. Build *lazy* evaluation into `static-intepreter`, such that the program below does not enter an infinite loop:

   ```
   > letrec ill = proc (x) (ill x) in
   >   let f = proc (y) 5 in
   >     (f (ill 2))

   5
   ```

   Name this interpreter `lazy-interpreter`.

Notes: Use SLLGEN to produce the parser. Also, contrary to Scheme, the language we ask you to implement uses the semi-colon to denote sequencing of actions, not comments.