Assignment 2

Benjamin Jakubowski

June 17, 2016

1. Bindings and Nested Subprograms

Consider the program

```
program main:
  var a, b : integer;

  procedure sub1;
    var x , y : integer;
    begin {sub1}
    ...
    end; {sub1}

  procedure sub2;
    var x , y : integer;

    procedure sub3;
      var y, a: integer;
      begin {sub3}
      ...
      end; {sub3}
    begin {sub2}
    end; {sub2}

  begin {main}
  ...
  end {main}
```

Assuming static scoping is used, the table below lists all the variables, along with the program units where they are declared, that are visible in the bodies of sub1, sub2, and sub3.

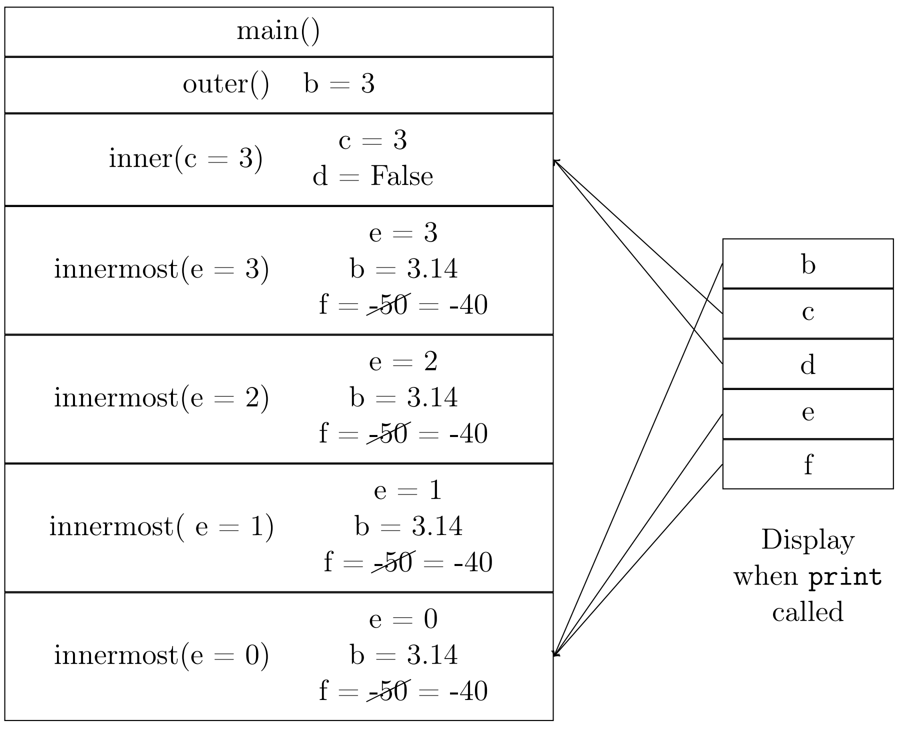
Unit	Var	Where Declared
sub1	x, y	sub1
	a, b	main
sub2	a	main
	x, b, t	sub2
sub3	x, b, t	sub2
	y, a	sub3

2. Bindings and Nested Subprograms

Consider the program

```
procedure outer ( ) is
  b : integer = 3;

  procedure inner ( c : integer ) is
    d : boolean = False;

    procedure innermost ( e : integer ) is
      b : real = 3.14;
      f : integer = -50 ;

    begin
      f = f + 10;

      if e == 0 then
        print b, c, d, f;
      else
        innermost(e-1);
      end if;

  begin
    innermost(c);
  end;

begin
  inner(b);
end
```

The runtime stack for this program (assuming a function `main` calls `outer()`) is shown below.


main()
outer() b = 3
inner(c = 3) c = 3 d = False
innermost(e = 3) e = 3 b = 3.14 f = ~~-50~~ = -40
innermost(e = 2) e = 2 b = 3.14 f = ~~-50~~ = -40
innermost(e = 1) e = 1 b = 3.14 f = ~~-50~~ = -40
innermost(e = 0) e = 0 b = 3.14 f = ~~-50~~ = -40
b
c
d
e
f
Display when print called

Call Stack

Thus, when the `innermost` base case is reached, the `print` statement will print

```
3.14, 3, False, 0, -40
```

3. Parameter Passing

Consider the program

```
 1  A[] = { 1, 3, 2, 7 };
 2  integer i = 0;
 3  mystery(i, A[i+1])
 4
 5  procedure mystery (a1, a2)
 6    integer tmp = 3;
 7
 8      for c from 1 to 3 do  // 1 to 3 inclusive
 9        for tmp = tmp + a2;
10        a1++;
11      end for;
12
13  end procedure;
```

We trace the code under (i) *call-by-name* and (ii) *call-by-value* semantics.

A. *Call-by-value*

Recall in *call-by-value* formal parameters are bound to the value of arguments. With this in mind, we trace the code (noting `i` and `A[i+1]` are bound when `mystery(i, A[i+1])` is called).

Line	State
3	Under *call-by-value*, the expressions `i, A[i+1]` in `mystery(i, A[i+1])` are immediately evaluated, so the formal parameters `a1` and `a2` are defined as `a1 = i = 0` and `a2 = A[i + 1] = A[0 + 1] = A[1] = 3`.
4	`temp = 3`.
7	`c = 1`
8	`temp = temp + a2 = 3 + 3 = 6`
9	`a1++ = a1 + 1 = 0 + 1 = 1`
7	`c = 2`
8	`temp = temp + a2 = 6 + 3 = 9`
9	`a1++ = a1 + 1 = 1 + 1 = 2`
7	`c = 3`
8	`temp = 9 + 3 = 5 + 1 = 12`
9	`a1++ = a1 + 1 = 2 + 1 = 3`

Thus, the final result is `temp = 12` and `i = 0`.

B. *Call-by-name*

Recall in *call-by-name* formal parameters are bound to the expressions passed to the subprogram, and these expressions are evaluated every time the parameter is referenced.

With this in mind, we trace the code.

Line	**State**
3	Under *call-by-name*, the formal parameters `a1` and `a2` are bound to the passed expression, such that `a1` is bound to the expression `"i"` and `a2` is bound to the expression `"A[i + 1]"`.
4	`temp = 3`.
7	`c = 1`
8	`temp = temp + a2 = 3 + A[i + 1] = 3 + A[0 + 1] = 3 + A[1] = 3 + 3 = 6`
9	`a1++ = i++ = i + 1 = 0 + 1 = 1`
7	`c = 2`
8	`temp = temp + a2 = 6 + A[i + 1] = 6 + A[1 + 1] = 6 + A[2] = 6 + 2 = 8`
9	`a1++ = i++ = i + 1 = 1 + 1 = 2`
7	`c = 3`
8	`temp = temp + a2 = 8 + A[i + 1] = 8 + A[2 + 1] = 8 + A[3] = 8 + 7 = 15`
9	`a1++ = i++ = 2 + 1 = 2 + 1 = 3`

Thus, the final result is `i = 3` and `temp = 15`.

4. Lambda Calculus

a. β reductions on SKK

Let S and K stand for the following lambda expressions:

$$S = \lambda x.\lambda y.\lambda z.(xz(yz))$$

and

$$K = \lambda x.\lambda y.x$$

We show the composition SKK is equivalent to $\lambda z.z$:

$$\begin{aligned} SKK &= \lambda x.\lambda y.\lambda z.(xz(yz))KK \\ &= \lambda y.\lambda z.(Kz(yz))K \\ &= \lambda z.(Kz(Kz)) \end{aligned}$$

Next, note

$$\begin{aligned} Kz &= (\lambda x.\lambda y.x)z \\ &= \lambda y.z \end{aligned}$$

Thus

$$\begin{aligned} SKK &= \lambda z.(Kz(Kz)) \\ &= \lambda z.\,(\lambda y.z(\lambda y.z)) \\ &= \lambda z.z \end{aligned}$$

since $(\lambda y.z(\lambda y.z)) = z$.

B. Reducing an expression two ways

We reduce the following expression two ways:

$$(\lambda x. *\ x\ x)(+\ 2\ 3)$$

B.1. Reduction in applicative-order

To reduce in applicative order, we first evaluate the arguments, then pass them to the function. Specifically, note $(+23) = 5$. Thus

$$\begin{aligned}(\lambda x. *\ x\ x)(+\ 2\ 3)(+\ 2\ 3) &= (\lambda x. *\ x\ x)(5)(5)\\ &= (*\ 5\ 5)\\ &= 25\end{aligned}$$

B.2. Reduction in normal-order

To reduce in normal order, we pass the expressions directly to the function before they're evaluated.

$$\begin{aligned}(\lambda x. *\ x\ x)(+\ 2\ 3)(+\ 2\ 3) &= (*(+\ 2\ 3)(+\ 2\ 3))\\ &= (*\ 5\ 5)\\ &= 25\end{aligned}$$

C. Need for α-conversions

In this section we reduce three lambda expressions.

C.1. $(\lambda xy.yx)(\lambda x.xy)$

Consider the expression $(\lambda xy.yx)(\lambda x.xy)$. Note y is unbound in $(\lambda x.xy)$ but bound in $(\lambda xy.yx)$; hence α-conversion is required:

$$\begin{aligned}&(\lambda xy.yx)(\lambda x.xy)\\ &\qquad \to_\alpha (\lambda xw.wx)(\lambda x.xy)\\ &\qquad \to_\beta (\lambda w.w(\lambda x.xy))\end{aligned}$$

Now, if we did not use an α-conversion, we would incorrectly obtain the reduced expression:

$$\begin{aligned}&(\lambda xy.yx)(\lambda x.xy)\\ &\qquad \to_\beta (\lambda y.y(\lambda x.xy))\end{aligned}$$

and $(\lambda y.y(\lambda x.xy)) \neq (\lambda w.w(\lambda x.xy))$.

C.2. $(\lambda x.xz)(\lambda xz.xy)$

Consider the expression $(\lambda x.xz)(\lambda xz.xy)$. Note initially no α-conversion is required, since there are no free variables in $(\lambda xz.xy)$ which are bound in $(\lambda x.xz)$.
Thus we proceed with a β-conversion

$$\begin{aligned}&(\lambda x.xz)(\lambda xz.xy)\\ &\qquad \to_\beta (\lambda xz.xy)z\end{aligned}$$

Now we require an α conversion, since the second z is free while z is bound in $(\lambda xz.xy)$.

$$\begin{aligned}&\to_\alpha (\lambda xw.xy)z\\ &\to_\beta (\lambda w.zy)\end{aligned}$$

If we did not use an *alpha*-conversion, we would incorrectly obtain the reduced expression:

$$\begin{aligned}&(\lambda x.xz)(\lambda xz.xy)\\ &\qquad \to_\beta (\lambda xz.xy)z\\ &\qquad \to_\beta (\lambda z.zy)\end{aligned}$$

and $(\lambda w.zy) \neq (\lambda z.zy)$.

C.3. $(\lambda x.xz)(\lambda x.x)$

Consider the expression$(\lambda x.xz)(\lambda x.x)$. No α-conversion is required, since there are no free variables in $(\lambda x.x)$.
Thus we proceed with a β-conversion

$$\begin{aligned}&(\lambda x.xz)(\lambda x.x)\\ &\qquad \to_\beta (\lambda x.x)y\\ &\qquad \to_\beta y\end{aligned}$$

D. $\texttt{PLUS}\ \ulcorner 1\urcorner\ulcorner 1\urcorner$

Recall

$$\texttt{PLUS} = \lambda mnfx.mf(nfx)$$

$$\ulcorner 1\urcorner = \lambda fx.fx$$

So

$$\begin{aligned}
\texttt{PLUS}\ulcorner 1\urcorner\ulcorner 1\urcorner &= (\lambda mnfx.mf(nfx))\,(\lambda fx.fx)\,(\lambda fx.fx)\\
&\to_\beta (\lambda nfx.\,(\lambda fx.fx)\,f(nfx))\,(\lambda fx.fx)\\
&\to_\beta (\lambda fx.\,(\lambda fx.fx)\,f((\lambda fx.fx)\,fx))\\
&\to_\beta (\lambda fx.\,(\lambda fx.fx)\,f(fx))\\
&= (\lambda fx.\,((\lambda fx.fx)\,f(fx)))\\
&\to_\beta (\lambda fx.\,((\lambda x.fx)\,(fx)))\\
&\to_\beta (\lambda fx.\,(f(fx)))
\end{aligned}$$