Procedural Abstraction

CPSC 509: Programming Language Principles

Ronald Garcia[*]

3 February 2014

In this class we talk about a mechanism that all of you have used already: *procedures*. We often call them *functions*, but I hope to use the other word more often to distinguish them from the mathematical functions that we are working with in this class. This mechanism gives us a lot of power over how we can cleanly organize our programs and break up their behavior in well-organized ways, avoid duplication, and give meaningful names. We call this *procedural abstraction*. In particular, we will take a programming languages approach to studying procedures, to see what they can do, and what many of the outstanding issues are about programming with them.

Suppose you work for a promising startup company that develops software. After a long night of hacking, you put together this phenomenally useful program:

```
(if (zero? (- (+ 3 2) 1))
    (+ 3 2)
    (- (+ 3 2) 1))
```

Several of your colleagues think it's a great piece of code, and they want to use it, but they need to change it a bit to suit their purpose. Sure you could just give them the code and they could modify it for their particular needs, but then any bug fixes or enhancements you make to your code will not immediately show up in theirs. On the other hand, if your language supports *procedural abstraction*, the ability to write and call procedures/functions, then you can re-structure your program to support their needs as well as your own.

Racket, our example language, definitely supports defining procedures:

```
(define (f x) <procedure body>)
```

as well as calling them

```
(f 0)
```

Then you can take advantage of this abstraction mechanism to generalize your program for all of your friends. However, there's a very key issue to keep in mind: the way that you generalize your program depends on the needs of your users. Let's run through a few different scenarios:

Scenario 1: Different numbers Suppose your colleagues want to do the same computation but with different values in place of `3+2`. Well, piece of cake: replace every instance of `3+2` with a *variable reference* and make that *variable* the *formal parameter* to your procedure:

```
(define (when-one x)
  (if (zero? (- x 1))
      (+ 3 2)
      (- (+ 3 2) 1)))
```

Then you can get your previous results by calling it with the original expression as its *argument*:

```
(when-one (+ 3 2))
```

[*]

and your friends can use whatever other numeric expressions they want as their arguments to the procedure, e.g.

```
(when-one 0)
```

Scenario 2: Different operation Suppose, on the other hand, that your friends definitely want to use `3`, but they want to do something different from adding two (i.e. `(+ ... 2)`) to it in every spot. *No Problem!* Following the same recipe as above, replace every instance of `(+ ... 2)` with a procedure call, where the variable `g` *represents the procedure*, i.e. `(g ...)` and make that variable a formal parameter again:

```
(define (when-g3-is-one g)
  (if (zero? (- (g 3) 1))
      (g 3)
      (- (g 3) 1)))
```

Now you can get your behavior back by defining a procedure for plus-two, and passing *that* procedure in.

```
(define (add-2 x) (+ x 2))

(when-g3-is-one add-2)
```

Well, this might look a little unusual to you. Here we're *passing a procedure as an argument to another procedure and then using it*. Only some programming languages let you do that,[1] but as we see here, it can be mighty useful. Now, your friends can use your code by passing along their own operations as procedures, e.g.:

```
(define (times-2 x) (* x 2))

(when-g3-is-one times-2)
```

Scenario 3: Lots of double-calls Your friends are loving that they can pass in an operation to your innovative procedure `when-g3-is-one`, but perhaps they find that they must often repeat the same pattern over and over. For instance, maybe every procedure they pass in is an instance of repeating the same operation twice. For example, the expression `(+ x 2)` can be viewed as adding 1 twice (i.e. `(+ (+ x 1))`) and maybe someone else wants to multiply by four (i.e. `(* (* x 2)2)`) and yet some other poor fellow must frobnicate the argument twice (i.e. `(frobnicate (frobnicate x))`)! You could write *yet another* version of your function, of course called `when-gg3-is-one`, that applies its procedure argument twice, but that seems unsatisfying. Sure you could then pass into it the function `(define (g x)(+ x 1))` and `frobnicate`, but now you can't use `(define (g x)(* x 2))` with `when-gg3-is-one`, you need to use `when-g3-is-one`, which means that you need to keep two copies of the function around, and naturally whenever you fix a bug in one, you have to fix it in the other. Yuk! You'd be stuck maintaining two versions of your `f` procedure, and that's exactly what you don't want to do! Luckily there's a way to have your code and run it too: instead of rewriting your amazing procedure from scratch yet again, you can write a helper procedure that just captures the idea of "doing something twice":

```
(define (do-h-twice h)
  (define (r y) (h (h y))) ; define a procedure r that applies h twice
  r)                        ; and return that procedure
```

Here we define a procedure that:

1. takes a procedure (`h`)
2. in its body defines a *new* procedure `r` that uses `h`
3. returns the procedure `r` as its result.

Now given that procedure, you can define `when-gg3-is-one` in terms of `when-g3-is-one`!:

[1] FORTRAN definitely doesn't, but C does!

```
(define (when-gg3-is-one f)
  (when-g3-is-one (do-h-twice f)))
```

and you can get your behavior by calling

```
(define (add-1 x) (+ x 1))
(when-gg3-is-one add-1)
```

while your friend gets her behavior by calling `(when-gg3-is-one frobnicate)`.

Consider what's going on with `do-h-twice` though. Not only is it taking a procedure as an argument, but it's *returning a procedure as its result*. Taking it even further, the procedure that it's returning is defined in terms of the procedure that you passed in, using it along the way.[2] You can see here how this capability can be useful. A language that lets you create procedures on the fly, pass them into functions, return them as values, pretty much anywhere, is said to support *first-class procedures*.

In this example, though, isn't it annoying that you have to give a name to the procedure that adds one and *then* pass that name into the procedure? It's not like I have to always write:

```
(define three 3)
(do-something-to three)
```

to call `do-something-to` with the value `3`. The name `three` doesn't add anything. Same for the body of the procedure `do-h-twice` which defines a procedure named `r` and returns it. It would be nice if we could simply write something that means "the procedure that takes a value and adds 1 to it". Sounds like a great idea! We can use the keyword λ to define a procedure in place without giving it a name. We write `(λ (x)<procedure body>)` to define a procedure in-line without giving it a name. Then we can write `(when-gg3-is-one (λ(x)(+ x 1)))` to call `when-gg3-is-one` with the `add-1` procedure in-line, and we can go back and redefine the procedure `do-h-twice` as

```
(define (do-h-twice h) (λ (y) (h (h y))))
```

Languages that support features like λ are said to support *anonymous procedures*, since they don't need to be given a name.

Our goal here is to support procedural abstraction in a way that is general enough to support all of the examples above. As it turns out, it doesn't take a whole lot to do that.

1 Substitution

Before we add procedures, let's consider what it means to call a procedure like `(when-one 0)` in Scenario 1. Intuitively, we want to use `0` everywhere that the procedure refers to its formal parameter `x`. Formally we say that we want to *substitute* `0` for `x` in the body of the procedure.

Let's formalize this concept in the context of our language of Boolean expressions.

$$\begin{array}{lcl} n \in \mathbb{Z}, & & t \in \textsc{Term}, \quad x \in \textsc{Var} \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & \boxed{x} \end{array}$$

Recall that $\mathbb{Z}$ is the set of integers. The set VAR is some infinite set of identifiers. For our purposes we'll use variable names like x and y. The TERM x is a *variable reference*, which indicates where an *argument* will be eventually substituted.

We model substitution using functions from TERMs to TERMs. The strategy, though, is to define a different substitution function for *each* instance of substitution. The idea is that once you've seen one version of this function, you can see exactly how you would define any other (so you then know that they all exist and their equational properties).

For instance, let's define a function $[\mathsf{0}/\mathsf{x}] : \textsc{Term} \to \textsc{Term}$ that substitutes $\mathsf{0}$ for every instance of x in a term.[3] For instance, we expect the following equation to hold:

$$[\mathsf{0}/\mathsf{x}](\mathsf{if\ false\ then\ x\ else}\ (\mathsf{x} + \mathsf{1})) = \mathsf{if\ false\ then\ 0\ else}\ (\mathsf{0} + \mathsf{1}).$$

[2] You definitely can't do that in C!

[3] This style of name for the substitution function ($[\mathsf{0}/\mathsf{x}]$) is common in the literature

We first define this function in longhand style. Here is the principle of definition by recursion for the language augmented with variables:

Proposition 1 (Principle of Definition by Recursion on terms $t \in \textsc{Term}$)**.** *Let S be a set and $s_t, s_f \in S$ be two elements,*

$$H_{if} : S \times S \times S \to S$$

be a function on S, and

$$H_{var} : \textsc{Var} \to S.$$

Then there exists a unique function

$$F : \textsc{Term} \to S$$

such that

1. $F(\textsf{true}) = s_t$;
2. $F(\textsf{false}) = s_f$;
3. $F(\textsf{if}\, t_1 \;\textsf{then}\, t_2 \;\textsf{else}\, t_3) = H_{if}(F(t_1), F(t_2), F(t_3))$;
4. $F(x) = H_{var}(x)$.

The new components of the principle are marked in grey above. The H_{var} function on variables $\textsc{Var}$ is responsible for handling *all* possible variable references. Essentially this function is analogous to the s_t and s_f constants, which handle the two individual constants.

So here is our definition of $[\mathsf{0}/\mathsf{x}]$. We define the function by applying the principle of definition by recursion to the following components:

1. $S = \textsc{Term}$;
2. $s_t = \mathsf{true}$;
3. $s_f = \mathsf{false}$;
4. $H_{if}(t_1, t_2, t_3) = \mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3$;
5. $H_{var}(x) = \begin{cases} \mathsf{0} & \text{if } x = \mathsf{x} \\ x & \text{if } x \neq \mathsf{x}. \end{cases}$

Be careful not to be confused by the fact that we chose $S = \textsc{Term}$, which is necessary to define a function from $\textsc{Term}$s to $\textsc{Term}$s.

Substituting these components into the principle itself (and breaking the cases into separate equations as is standard practice), we get the following shorthand definition:[4]

$$\begin{aligned}
&[\mathsf{0}/\mathsf{x}]\mathsf{x} = \mathsf{0} \\
&[\mathsf{0}/\mathsf{x}]x = x \text{ if } x \neq \mathsf{x} \\
&[\mathsf{0}/\mathsf{x}]\mathsf{true} = \mathsf{true} \\
&[\mathsf{0}/\mathsf{x}]\mathsf{false} = \mathsf{false} \\
&[\mathsf{0}/\mathsf{x}]\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 = \mathsf{if}\ ([\mathsf{0}/\mathsf{x}]t_1)\ \mathsf{then}\ ([\mathsf{0}/\mathsf{x}]t_2)\ \mathsf{else}\ ([\mathsf{0}/\mathsf{x}]t_3).
\end{aligned}$$

If we extend our language to include arithmetic expressions, and follow the same recipe, then we can show that the example above exactly fits this model.

One thing that we can immediately see is that given any $\textsc{Value}\ v$ and any $\textsc{Var}\ x$, we can define $[v/x]$ by simply producing a new $H_{var}(x)$ function.

Finally, now that we know how to define substitution for particular functions, we can treat substitution generically with respect to which term is being substituted, and which variable it is being substituted for.

[4] It is a common notational style to not wrap the substitution argument in parentheses.

Finally, we can define a generic substitution function

$$\{\cdot/\cdot\}\cdot : \text{TERM} \times \text{VAR} \times \text{TERM} \rightarrow \text{TERM}$$
$$\{t_1/x\}t_2 = [t_1/x]t_2$$

For this definition, we're using slightly different notation (braces rather than brackets) just to make it clear that we are defining a separate general substitution function using the original individual functions, but in practice we use square brackets for both.

At this point in the course we have to define the general substitution function this way because we can't justify defining this function directly, (we have no recursion principle for TERM × VAR × TERM so we have to settle for defining the individual substitution functions and only then defining the general function in turn. Later in the course we will learn how to justify a direct definition of general substitution. However, the individual substitution functions will suffice for us in most cases.

2 Lambda Abstractions as Procedures

Now that we have a model for what it means to call procedures, we need to actually add procedures to our language! Based on the scenarios above, we want to be able to do a number of things, including passing procedures to procedures and producing procedures as the results of other procedures. Amazingly, getting this much power out of procedures, at least mathematically, takes very little additional machinery.

First, let's introduce our notation for procedures.

$$t ::= \ldots \mid \lambda x.t$$

The symbol λ is the Greek character "lambda", and the notation $\lambda x.t$ is called a *lambda abstraction*, because it stands for *abstracting* the term t with respect to the *parameter* x. Referring back to tree form, this is the common notation used for what is really under the hood a TREE of the form $\lambda(x, t)$

Comparing this to our earlier notation, a procedure like `add-1` defined as

```
(define (add-1 x) (+ x 1))
```

which is the procedure that maps `x` to `x+1` corresponds to the lambda abstraction

$$\lambda \mathsf{x}.\mathsf{x} + 1.$$

Notice that we didn't even have to give this procedure a name. Now, our language already has a set of terms that immediately stand for performing some operation on a term, like $t = 0$. Now we are going to be able to write terms that produce new procedures and then we want to *apply* a computed procedure to an argument. We also add new notation for this.

$$t ::= \ldots \mid t\ t$$

The notation $t_1\ t_2$[5]

means to evaluate t_1 to produce some procedure $\lambda x.t_{11}$, evaluate t_2 to produce some argument v_2, and then *apply* the procedure to the argument. Under the hood, the notation $t_1\ t_2$ corresponds to a TREE like $\mathsf{apply}(t_1, t_2)$: for some odd reason, standard convention is to use juxtaposition of TERMs to stand for procedure application.

Notational Conventions The "dot" in a lambda abstraction $\lambda x.t$ acts kind of like a parenthesis, in that it helps to determine what the body of a procedure is. One important aspect of dot is that it *eats everything to its right*. For example, the expression $\lambda x.y\ x$ corresponds to the TREE $\lambda(x, \mathsf{apply}(y, x))$. Similarly, the expression $\lambda x.y\ \lambda z.x$ corresponds to the TREE $\lambda(x, \mathsf{apply}(y, \lambda(z, x)))$. If you want to restrict the reach of a lambda abstraction's body, then you should place the entire expression in parentheses: for example, the expression $(\lambda x.y)\ \lambda z.x$ corresponds to the tree $\mathsf{apply}(\lambda(x, y), \lambda(z, x))$

[5] This is written in LaTeX as `t_1\;t_2` where the `\;` gets the spacing right.

By convention, application associates to the *left*, so

$$t_1\ t_2\ t_3 \equiv (t_1\ t_2)\ t_3 \equiv \mathsf{apply}(\mathsf{apply}(t_1, t_2), t_3).$$

If you want one of the arguments to an application to be itself an application, then you must explicitly parenthesize it, e.g., $t_1\ (t_2\ t_3)$.

It takes practice to get the notation for lambda abstractions and applications right. *Make sure that you can read and write these expressions by taking advantage of these conventions.*

One thing to keep in mind is the relationship between applying our new procedures compared to the old operators that we had in the language. If our language still had the term $\mathsf{zero?}(t)$, then it would *not* be legal in our language to write: $\mathsf{zero?}\ t$ in the sense of $\mathsf{apply}(\mathsf{zero?}, t)$, because $\mathsf{zero?}$ by itself is not a TERM in the language, only expressions of the form $\mathsf{zero?}(t)$ are. *However*, the expression $\lambda\mathsf{x}.\mathsf{zero?}(\mathsf{x})$ is a perfectly fine TERM, so we can always write $(\lambda\mathsf{x}.\mathsf{zero?}(\mathsf{x}))\ \mathsf{0}$.

Let's write down the rest of the formal semantics of our new language. We call it TFL because it's a Tiny Functional programming Language. It is much like the Boolean and Arithmetic Language, but we replace unary arithmetic with proper integers, for greater convenience.[6]

$$\begin{array}{lll} n \in \mathbb{Z}, & t \in \textsc{Term}, & x \in \textsc{Var} \\ t & ::= & n \mid t\ \mathsf{=}\ t \mid t\ \mathsf{+}\ t \mid t\ \mathsf{-}\ t \mid t \ast t \\ & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & x \mid \lambda x.t \mid t\ t \end{array}$$

Our big-step relation for Boolean and Arithmetic Expressions is defined by the following rules:

$$\text{(elam)}\frac{}{\lambda x.t \Downarrow \lambda x.t} \qquad \text{(eapp)}\frac{t_1 \Downarrow \lambda x.t_{11} \quad t_2 \Downarrow v_2 \quad [v_2/x]t_{11} \Downarrow v}{t_1\ t_2 \Downarrow v} \qquad \text{(etrue)}\frac{}{\mathsf{true} \Downarrow \mathsf{true}}$$

$$\text{(efalse)}\frac{}{\mathsf{false} \Downarrow \mathsf{false}} \qquad \text{(eif-t)}\frac{t_1 \Downarrow \mathsf{true} \quad t_2 \Downarrow v_2}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_2} \qquad \text{(eif-f)}\frac{t_1 \Downarrow \mathsf{false} \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_3}$$

$$\text{(en)}\frac{}{\mathsf{n} \Downarrow \mathsf{n}} \qquad \text{(eqt)}\frac{t_1 \Downarrow n \quad t_2 \Downarrow n}{t_1\ \mathsf{=}\ t_2 \Downarrow \mathsf{true}} \qquad \text{(eqf)}\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1\ \mathsf{=}\ t_2 \Downarrow \mathsf{false}} \quad n_1 \neq n_2$$

$$\text{(plus)}\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1\ \mathsf{+}\ t_2 \Downarrow n_3} \quad n_3 = n_1 + n_2 \qquad \text{(minus)}\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1\ \mathsf{-}\ t_2 \Downarrow n_3} \quad n_3 = n_1 - n_2$$

$$\text{(times)}\frac{t_1 \Downarrow n_1 \quad t_2 \Downarrow n_2}{t_1 \ast t_2 \Downarrow n_3} \quad n_3 = n_1 \times n_2$$

Notice that there is no rule for evaluating variables. We expect that their meaning is solely determined by substitution for procedure parameters. Nonetheless, some of the rules can be instantiated to have variables in them.

Substitution for this larger language needs to account for the new arithmetic operations as well as pro-

[6] Note that we've seen a stripped down version of this language in class.

cedure abstraction and application.

$$
\begin{aligned}
&[t/x] : \textsc{Term} \to \textsc{Term} \\
&\qquad\qquad [t/x]\mathsf{true} = \mathsf{true} \\
&\qquad\qquad [t/x]\mathsf{false} = \mathsf{false} \\
&\qquad\qquad [t/x]x = t \\
&\qquad\qquad [t/x]x_0 = x_0 \text{ if } x_0 \neq x \\
&[t/x](\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{then}\ t_3) = \mathsf{if}\ ([t/x]t_1)\ \mathsf{then}\ ([t/x]t_2)\ \mathsf{else}\ ([t/x]t_3) \\
&\qquad\qquad [t/x]n = ??? \\
&\qquad\qquad [t/x]t_1 \mathbin{\mathsf{=}} t_2 = ??? \\
&\qquad\qquad [t/x]t_1 \mathbin{\mathsf{+}} t_2 = ??? \\
&\qquad\qquad [t/x]t_1 \mathbin{\mathsf{-}} t_2 = ??? \\
&\qquad\qquad [t/x]t_1 * t_2 = ??? \\
&\qquad\qquad [t/x]t_1\ t_2 = ([t/x]t_1 \cup [t/x]t_2 \\
&\qquad\qquad [t/x]\lambda x_0.t_0 = ???
\end{aligned}
$$

I leave the cases for arithmetic expressions as an exercise to the reader. We can see above that substitution for application is straightforward. Extending substitution to deal with lambda abstractions, however, is not so straightforward. We deal with that next.

3 Substitution and Lambda Abstractions

In this section, we discuss how substitution and lambda abstractions interact. You might think that it follows obviously based on what you've seen before, and you would be mostly right, but there is some subtlety to getting substitution "right," by which I mean behaving in a manner that actually matches your intuitions. Historically programming languages like LISP (and to this day Emacs Lisp) got this "wrong." In this section, we'll talk about some of the things that can go wrong with substitution into procedures, then give a general-purpose definition of substitution.

Substitution for applications is quite easy, similar to the case for if:

$$[t/x]t_1\ t_2 = ([t/x]t_1)\ ([t/x]t_2). \tag{$\star$}$$

The case for lambda abstractions is a bit more subtle though! You might be tempted to do the "obvious" thing, proceed by recursion on the structure of TERMs.

$$[t/x]\lambda x_0.t_0 = \lambda x_0.[t/x]t_0$$

But this equation implies results that we do not desire. To make this concrete consider the following two examples:

$$
\begin{aligned}
[\mathsf{0}/\mathsf{x}]\lambda\mathsf{y}.\mathsf{x} &= ??? \\
[\mathsf{0}/\mathsf{x}]\lambda\mathsf{x}.\mathsf{x} &= ???
\end{aligned}
$$

The first case seems pretty straightforward: the x becomes the value 0 as a result of substituting for x:

$$[\mathsf{0}/\mathsf{x}]\lambda\mathsf{y}.\mathsf{x} = \lambda\mathsf{y}.[\mathsf{0}/\mathsf{x}]\mathsf{x} = \lambda\mathsf{y}.\mathsf{0}.$$

The second case is a little more odd. Substitution is applied to the identity procedure, $\lambda\mathsf{x}.\mathsf{x}$, which takes an argument and immediately yields that argument as its result. That means that the reference to x is referring specifically to the x bound by the lambda abstraction. We do not want substitution to "break" this procedure: after substitution, the same variable should refer to the same binding site, so something like

$$[\mathsf{0}/\mathsf{x}]\lambda\mathsf{x}.\mathsf{x} = \lambda\mathsf{x}.\mathsf{0}.$$

Would be painfully broken! Looking at it from yet another angle, consider the following example:

$$[\mathsf{y}/\mathsf{x}]\lambda\mathsf{y}.\mathsf{y}\ \mathsf{x} = ???$$

The procedure body has a variable reference x that ostensibly points to outside of the procedure somewhere, but if we use equation ($\star$):

$$[\mathsf{y}/\mathsf{x}]\lambda\mathsf{y}.\mathsf{y}\ \mathsf{x} = \lambda\mathsf{y}.\mathsf{y}\ \mathsf{y}$$

Then that variable reference to x, which pointed outside the procedure is now *captured* by the procedure's parameter. What's worse is that if we keep this behavior but change the name of the variable, then the behavior of the procedure changes!

$$[\mathsf{y}/\mathsf{x}]\lambda\mathsf{w}.\mathsf{w}\ \mathsf{x} = \lambda\mathsf{w}.\mathsf{w}\ \mathsf{y}$$

Now the variable reference y that is substituted in is no longer captured. Intuitively, *the meaning of the program should not depend on which function parameter names you choose inside a procedure*: that kind of dependency breaks the abstraction barrier that we are trying to create (hence the name "procedural abstraction"). This was exactly something that the LISP programming language got wrong in its early days (and that Emacs LISP still has wrong).

The lesson to be learned here is that we must use care when defining substitution for lambda abstractions. To do so, we must first introduce a few new concepts.

3.1 Free Variables

First, we must consider the status of variables in programs. Variables in this language stand for references to procedure parameters that will eventually be replaced with values when the corresponding procedure is applied. For this reason, we had better make sure that each variable is associated with a procedure parameter. Consider the expression $\mathsf{x} + 1$. By itself, this expression does not associate x with any procedure parameter. We say that the variable x *occurs free* in the term $\mathsf{x} + 1$. On the other hand, in the expression $\lambda\mathsf{x}.\mathsf{x} + 1$, the argument to $\mathsf{x} + 1$ is *bound* by the surrounding lambda abstraction. That is to say, there are no free variables in the term $\lambda\mathsf{x}.\mathsf{x} + 1$. We say that this term is *closed*. We formalize this concept by defining a function that gives the *free variables* of any TERM.[7]

$$
\begin{aligned}
FV &: \textsc{Term} \to \mathcal{P}(\textsc{Var})\\
FV(\mathsf{true}) &= \emptyset\\
FV(\mathsf{false}) &= \emptyset\\
FV(x) &= \{\,x\,\}\\
FV(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{then}\ t_3) &= FV(t_1) \cup FV(t_2) \cup FV(t_3)\\
FV(n) &= ???\\
FV(t_1 \mathbin{\mathsf{=}} t_2) &= ???\\
FV(t_1 \mathbin{\mathsf{+}} t_2) &= ???\\
FV(t_1 \mathbin{\mathsf{-}} t_2) &= ???\\
FV(t_1 * t_2) &= ???\\
FV(t_1\ t_2) &= FV(t_1) \cup FV(t_2)\\
FV(\lambda x.t) &= FV(t) \setminus \{\,x\,\}
\end{aligned}
$$

[7] Once again, I leave it as an exercise for you to extended it to handle arithmetic expressions.

Analogously, the following function formalizes the notion of *bound variables*:

$$
\begin{aligned}
&BV : \textsc{Term} \to \mathcal{P}(\textsc{Var})\\
&\qquad\qquad BV(\mathsf{true}) = \emptyset\\
&\qquad\qquad BV(\mathsf{false}) = \emptyset\\
&\qquad\qquad BV(x) = \emptyset\\
&BV(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{then}\ t_3) = BV(t_1) \cup BV(t_2) \cup BV(t_3)\\
&\qquad\qquad BV(n) = ???\\
&\qquad\qquad BV(t_1 \mathsf{=} t_2) = ???\\
&\qquad\qquad BV(t_1 \mathsf{+} t_2) = ???\\
&\qquad\qquad BV(t_1 \mathsf{-} t_2) = ???\\
&\qquad\qquad BV(t_1 * t_2) = ???\\
&\qquad\qquad BV(t_1\ t_2) = BV(t_1) \cup BV(t_2)\\
&\qquad\qquad BV(\lambda x.t) = BV(t) \cup \{\, x \,\}
\end{aligned}
$$

Note that just as a variable may appear free multiple times in a single expression, a variable may be bound more than once in an expression (e.g., $\lambda \mathsf{x}.\lambda \mathsf{x}.\mathsf{x}$). Furthermore, a variable may appear both free and bound in the same expression (e.g. $(\lambda \mathsf{x}.0)\ \mathsf{x}$)).

3.2 Substitution for Procedures

Now that you have a formal understanding of free variables, we can complete our definition of substitution.

$$
\begin{array}{ll}
[t/x]\lambda x_0.t_0 = \lambda x_0.t_0 & \text{if } x = x_0\\
[t/x]\lambda x_0.t_0 = \lambda x_0.t_0 & \text{if } x \neq x_0 \text{ and } x \notin FV(t_0)\\
[t/x]\lambda x_0.t_0 = \lambda x_0.[t/x]t_0 & \text{if } x \neq x_0 \text{ and } x_0 \notin FV(t)\\
[t/x]\lambda x_0.t_0 = \lambda x_1.[t/x][x_1/x_0]t_0 & \text{if } x \neq x_0,\ x \in FV(t_0),\ x_0 \in FV(t), \text{ and } x_1 \notin FV(t) \cup FV(t_0)
\end{array}
$$

There are a few things worth mentioning about this particular definition. In the literature, a number of subtle points are typically glossed over because you can get pretty far with approximately the right definition, but it's good to know the finer details so that they don't come back to bite you.

Overlapping equations Take a close look at the second and third equations above. Notice that the side-conditions do not keep either equation from being applicable in cases that apply to the other equation (i.e. suppose that both $x \notin FV(t_0)$ *and* $\mathsf{x}_0 \notin FV(t)$. If you look closely, though, you'll find that that's okay: both equations are true in the cases where they overlap. This is one of the differences between math and programs. If I wrote the above as a program it may run less efficiently depending on how I order those two cases. Lucky for us mathematics doesn't run, it just singles out the function from TERMs to TERMs that satisfies these equations. There's nothing wrong with two equations being true at the same time.

This doesn't really define a function The definition above is not quite a function definition. Why? Well, notice the instance of x_1 in the last equation. The function doesn't specify *which* x_1 we're using, just that it is not free in t_1 or t_2. So how to deal with this? Well, most authors don't bother: they just say that x_1 is "fresh", which typically means that it doesn't appear in any of the terms currently under discussion), but technically the definition is broken because there are many VARs that satisfy that property, so this is not a single function but many functions. One fix, due to the logician Haskell Curry[8], is the following: Fix some total ordering on all $x \in X$, meaning that $X = \{\, \mathsf{x}_0, \mathsf{x}_1, \dots \}$. Then amend the above definition to say that:

$$x_1 \text{ is the least } \textsc{Var} \text{ such that } x_1 \notin FV(t_1) \cup FV(t_2).$$

[8] Haskell Curry has had two programming languages named after him: Haskell and Curry.

Yes, this is super-contrived, but to be honest there are some programming language implementations that essentially do something like this to deterministically choose new variable names. We may talk about that later in the course as we implement language features.

Not-so-obvious Induction Principle If you try to rewrite this function definition in "longhand" style, say by recursion over the structure of TERMs, you will likely run into some small hickups that don't seem to fit. For instance, how do you fit the renaming in from the last equation? Furthermore, renaming seems to be another use of "the same" function, or at least a related one. You probably weren't losing much sleep over this bit anyway, but let me assure you that this definition is technically fine. Nonetheless, the principle of recursion on TERMs isn't precisely sufficient to justify it[9].

4 Alpha-Equivalence (or Parameter Names Don't Matter)

One of the common threads in the discussion so far is that the parameter names in procedures shouldn't matter: for instance, the procedure $\lambda(\mathsf{x}, \mathsf{x})$ should be the same as $\lambda(\mathsf{y}, \mathsf{y})$. Only the internals of the procedure should care about which parameter name is used: the rest of the program should not depend on it. The alternative is a programming language where you have to worry all the time about what local variable names are used within each function. That makes it hard to build correct modules independently.

In the language of procedures we have described so far, especially now that we have a workable definition of substitution, this property holds, and we can formalize it as an equivalence relation.[10]

Definition 1 (Alpha Equivalence). *Let $\sim_a$: $\text{TERM} \times \text{TERM}$ be defined by the following rules:*

$$\frac{}{x \sim_a x} \qquad \frac{t_{11} \sim_a t_{21} \quad t_{12} \sim_a t_{22}}{t_{11}\ t_{12} \sim_a t_{21}\ t_{22}} \qquad \frac{t_1 \sim_a t_2}{\lambda(x, t_1) \sim_a \lambda(x, t_2)}$$

$$\frac{\lambda(x_3, [x_3/x_1]t_1) \sim_a \lambda(x_3, [x_3/x_2]t_2)}{\lambda(x_1, t_1) \sim_a \lambda(x_2, t_2)} \quad \begin{array}{l} x_1 \neq x_2, \\ x_3 \notin FV(t_1) \cup FV(t_2) \end{array}$$

Most of the cases that define this relation look exactly like equality: If we throw out the last rule, then the definition is in fact that for equality. This makes it clear that identical terms are alpha-convertible. The last rule, though is the only interesting one. Essentially it says that two lambda abstractions are alpha-convertible if *renaming their parameters* to a common parameter results in alpha-convertible terms.

Given this relationship between TERMs, we can formalize the sense in which parameter names don't matter:

Proposition 2. *If $t_1 \sim_a t_2$ and $t_3 \sim_a t_4$ then $[t_3/x]t_1 \sim_a [t_4/x]t_2$.*

Proof. By induction over $t_1 \sim_a t_2$.[11] □

Proposition 3. *If $t_1 \sim_a t_2$ and $t_1 \Downarrow v_1$ then $t_2 \Downarrow v_2$ and $v_1 \sim_a v_2$.*

Proof. An exercise. □

In essence, these propositions say that if two programs differ only in their choice of bound variables, then their results also only differ in their bound variable names.

We call $\sim_a$ "alpha-equivalence" because it forms an equivalence relation.

Proposition 4 ($\sim_a$ is an equivalence relation). *Given $t_1, t_2, t_3 \in \text{TERM}$, the following are true:*

1. $t_1 \sim_a t_1$;
2. *If $t_1 \sim_a t_2$ then $t_2 \sim_a t_1$;*

[9] ...despite what a number of textbooks say (sigh).

[10] We just define it for the pure language of lambda abstractions, but you can extend it to other language features yourself.

[11] I recommend working through this.

3. *If* $t_1 \sim_a t_2$ *and* $t_2 \sim_a t_3$ *then* $t_1 \sim_a t_3$.

The "alpha" bit is a painfully cute reference to "alphabet", implying that the programs are equivalent up to the choice of alphabet for bound variables. Equivalence relations show up all the time in mathematics. They are a mathematical way of characterizing groups of stuff that are "like one another" (i.e. are equivalent) so long as you ignore some uninteresting differences. Literal equality is an equivalence relation, but it is the one that ignores nothing: two entities are only equal if they are one and the same. In our case here, two TERMs are alpha-equivalent if they are exactly the same except for the choice of parameters used as binders in lambda abstractions. We want to believe that this choice doesn't matter, expecially when it comes lambda abstractions. As it turns out, we can prove this.

4.1 Equivalence Classes and the Variable Convention

What makes equivalence relations special is that each one *partitions* a set into a collection of non-overlapping subsets: the sets of all items that are equivalent to one another. This is one way of interpreting a group of things as though they were conceptually one (big) thing: collect them all. In our particular case, alpha equivalence $\sim_a$ partitions the set TERM into the set:

$$\text{TERM}/{\sim_a} = \{\, T \in \mathcal{P}(\text{TERM}) \mid \forall t_1, t_2 \in \text{TERM}.\, t_1 \in T \wedge t_2 \in T \iff t_1 \sim_a t_2 \,\}$$

of all α*-equivalence* classes. We'll refer to these classes by giving one of the members of the class and enclosing it in brackets. For example,

$$[\lambda x_3.x_3]_{\sim_a} = \{\, (\lambda x_0.x_0), (\lambda x_1.x_1), (\lambda x_2.x_2), \ldots \}$$
$$[\lambda x_3.x_1\ x_3]_{\sim_a} = \{\, (\lambda x_0.x_1\ x_0), (\lambda x_2.x_1\ x_2), (\lambda x_3.x_1\ x_3), \ldots \}$$

These notations are pretty standard: Given some set A and some equivalence relation $\approx$, the name $A/{\approx}$ refers to the set of equivalence classes, and for each element $a \in A$, the name $[a]_{\approx}$ refers to the set of all elements equivalent to a. Note that if $a_1 \approx a_2$, then $[a_1]_{\approx} = [a_2]_{\approx}$: they are just two different names for the same set. Often if the equivalence class is obvious, then the subscript is dropped for succinctness, i.e., $[a_1] = [a_2]$

Equivalence classes give us a relatively clean way of brushing all this variable-renaming nonsense under the rug. The strategy essentially boils down to this: Rather than take the set TERM as the abstract syntax of our language, *take the alpha equivalence classes* TERM$/{\sim_a}$ *to be the abstract syntax*.

Above, Prop. 2 showed that substitution respects alpha-equivalence classes. One consequence of this fact is that we can define *a substitution function over equivalence classes* $[t]_{\sim_a}$!¹²

$$\{\cdot/\cdot\}\cdot : (\text{TERM}/{\sim_a}) \times \text{VAR} \times (\text{TERM}/{\sim_a}) \to (\text{TERM}/{\sim_a})$$
$$\{[t]_{\sim_a}/x\}[t_0]_{\sim_a} = [[t/x]t_0]_{\sim_a}.$$

Substitution on alpha-equivalence classes is defined in terms of substitution on plain ole' terms. Can you see why Prop. 2 makes this a well-defined function? At this point, we can redefine the big-step semantics to operate on equivalence classes of terms rather than particular Terms.

Well, this seems like a big mess. How did it help us at all? The actual answer is that it hasn't yet. Our goal is to sweep some details under the carpet, but so far we've been dwelling on the details themselves. It's now time to do some sweeping.

In the literature, you will typically not see this kind of detail. Instead, what happens is this: The BNF for TERM is given, followed by a statement of the form "we consider terms *up to* alpha-equivalence" or "we consider terms *modulo* alpha-equivalence." These magic words draw in much of the incantation above. From this point, any reference to TERM is really talking about TERM$/{\sim_a}$. Furthermore, any reference to a metavariable like t is really a reference to the alpha-equivalence class $[t]_{\sim_a}$.

Here's where all this work shines through. First, a textual convention is established regarding metavariables. Remember that t really stands for $[t]_{ae}$, and that there are many possible t's that one can consider as a

[12] We use curly braces for substitution here just so the definition looks clear. In practice, one would really use square brackets to talk about this substitution function.

name for the class (any TERM that is alpha-equivalent). The following convention is taken: "If $t_1, t_2, \ldots, t_n$ appear in a certain mathematical context, then in these terms all bound variables are chosen to be different from the free variables." Since we are operating on alpha-equivalence classes, it's possible to make this assumption because we can always find equivalence class representatives t_i that together satisfy this convention.

This approach, when treated informally, is called Barendregt's Variable Convention (or just "The Variable Convention."). In fact, most texts don't bother mentioning that they assume the variable convention: instead they just note that they treat terms up to alpha-equivalence, and assume the variable convention without comment.

In practice, the variable convention means that whenever you write down terms or expressions that contain metavariables that stand for terms, you can safely assume that the bound variables of any subterm are distinct from the free variables of any subterm (because you could always alpha-convert first to make that so). Under this convention, the definition for just the procedural part of substitution becomes very short:

$$\begin{aligned}
[t/x]x &= t \\
[t/x]x_1 &= x_1 \text{ if } x_1 \neq x \\
[t/x]t_1\ t_2 &= ([t/x]t_1)\ ([t/x]t_2) \\
[t/x]\lambda x_1.t_1 &= \lambda x_1.[t/x]t_1
\end{aligned}$$

The variable convention only comes into play in the last rule, where you silently assume that $x \neq x_1$, and $x_1 \notin FV(t)$. Thus the relevant rule from substitution over actual TERMs applies.

This convention then carries over to definition of the big-step semantics for the language, as well as any other function or relation definitions. The result is a very concise definition on paper, but it's important to know what underlying hidden details are implied.

5 Defining an Evaluator

We're not done yet! Our language isn't fully defined until we have produced an evaluator for it. We have a few considerations to take into account in defining our language.

6 Legal Programs

What will count as programs in this language? Well, recall that variable references are given meaning by substitution, and our semantics only performs substitution for bound procedure variables when the procedure is called. It naturally follows, then that our programs should be such that all variables are bound, i.e., that there are no free variables.

Formally, we define two sets, the set of *closed terms* and the set of *closed values*:

$$\begin{aligned}
\text{TERM}^0 &= \{\, t \in \text{TERM} \mid FV(t) = \emptyset \,\}. \\
\text{VALUE}^0 &= \{\, v \in \text{VALUE} \mid FV(v) = \emptyset \,\}.
\end{aligned}$$

These will be our legal programs and our expected values.

6.1 Procedure results

We now have to consider what to do if a program results in a procedure. We *could* produce the procedure as the result itself, but that's a bit strange, and we don't want to depend on the internals of the procedure that we get back (if only because the language implementation may play some tricks with procedures that you don't want the user to see). So for this reason, we will simply report if a program results in a procedure, without reporting *which* procedure. To do that, we introduce the new atom procedure as an observable result of the evaluator.

6.2 The evaluator

Tying these concepts together, we get the evaluator for our language:

$$\begin{array}{l}
\textsc{Pgm} = \textsc{Term}^0 \\
k \in \textsc{Const} = \textsc{Bool} \cup \mathbb{Z} \\
\textsc{Obs} = \textsc{Const} \cup \{\, \textsf{procedure} \,\} \\
eval : \textsc{Pgm} \rightharpoonup \textsc{Obs} \\
eval(t) = \begin{cases} k & t \Downarrow k \\ \textsf{procedure} & t \Downarrow \lambda x_0.t_0. \end{cases}
\end{array}$$

First we distinguish the set of observable constants $\textsc{Const}$, and define evaluation to yield those constants, but evaluation yields the atom procedure if the result is some procedure $\lambda x.t$.