Unfortunately, this desguaring transformation won't work at all! Do you see why? If you don't, try to run it.

2. The second is that we are implicitly depending on exactly what `bminusS` means; if its meaning changes, so will that of `uminusS`, even if we don't want it to. In contrast, defining a functional abstraction that consumes two terms and generates one representing the addition of the first to -1 times the second, and using this to define the desugaring of both `uminusS` and `bminusS`, is a little more fault-tolerant.

   You might say that the meaning of subtraction is never going to change, so why bother? Yes and no. Yes, it's *meaning* is unlikely to change; but no, its *implementation* might. For instance, the developer may decide to log all uses of binary subtraction. In the macro expansion, all uses of unary negation would also get logged, but they would not in the second expansion.

Fortunately, in this particular case we have a much simpler option, which is to define $-b = -1 \times b$. This expansion works with the primitives we have, and follows structural recursion. The reason we took the above detour, however, is to alert you to these problems, and warn that you might not always be so fortunate.

5 Adding Functions to the Language

Let's start turning this into a real programming language. We could add intermediate features such as conditionals, but to do almost anything interesting we're going to need functions or their moral equivalent, so let's get to it.

Exercise

Add conditionals to your language. You can either add boolean datatypes or, if you want to do something quicker, add a conditional that treats `0` as false and everything else as true.

What are the important test cases you should write?

Imagine, therefore, that we're modeling a system like DrRacket. The developer defines functions in the definitions window, and uses them in the interactions window. For now, let's assume all definitions go in the definitions window only (we'll relax this soon [REF]), and all expressions in the interactions window only. Thus, running a program simply loads definitions. Because our interpreter corresponds to the interactions window prompt, we'll therefore assume it is supplied with a set of definitions.

A *set* of definitions suggests no ordering, which means, presumably, any definition can refer to any other. That's what I intend here, but when you are designing your own language, be sure to think about this.

5.1 Defining Data Representations

To keep things simple, let's just consider functions of one argument. Here are some Racket examples:

```
(define (double x) (+ x x))
```

```
(define (quadruple x) (double (double x)))

(define (const5 _) 5)
```

Exercise

When a function has multiple arguments, what simple but important criterion governs the names of those arguments?

What are the parts of a function definition? It has a name (above, `double`, `quadruple`, and `const5`), which we'll represent as a symbol (`'double`, etc.); its *formal parameter* or *argument* has a name (e.g., `x`), which too we can model as a symbol (`'x`); and it has a body. We'll determine the body's representation in stages, but let's start to lay out a datatype for function definitions:

<***fundef***> **::=**

```
(define-type FunDefC
  [fdC (name : symbol) (arg : symbol) (body : ExprC)])
```

What is the body? Clearly, it has the form of an arithmetic expression, and sometimes it can even be represented using the existing `ArithC` language: for instance, the body of `const5` can be represented as `(numC 5)`. But representing the body of `double` requires something more: not just addition (which we have), but also "`x`". You are probably used to calling this a *variable*, but we will *not* use that term for now. Instead, we will call it an *identifier*.

I promise we'll return to this issue of nomenclature later [REF].

Do Now!

Anything else?

Finally, let's look at the body of `quadruple`. It has yet another new construct: a function *application*. Be very careful to distinguish between a function *definition*, which describes what the function is, and an *application*, which uses it. These are uses. The *argument* (or *actual parameter*) in the inner application of `double` is `x`; the argument in the outer application is `(double x)`. Thus, the argument can be any complex expression.

Let's commit all this to a crisp datatype. Clearly we're extending what we had before (because we still want all of arithmetic). We'll give a new name to our datatype to signify that it's growing up:

<***exprC***> **::=**

```
(define-type ExprC
  [numC (n : number)]
  <idC-def>
  <app-def>
  [plusC (l : ExprC) (r : ExprC)]
  [multC (l : ExprC) (r : ExprC)])
```

Identifiers are closely related to formal parameters. When we apply a function by giving it a value for its parameter, we are in effect asking it to replace all instances of that formal parameter in the body—i.e., the identifiers with the same name as the formal parameter—with that value. To simplify this process of search-and-replace, we might as well use the same datatype to represent both. We've already chosen symbols to represent formal parameters, so:

Observe that we are being coy about a few issues: what kind of "value" [REF] and when to replace [REF].

<***idC-def***> **::=**

```
[idC (s : symbol)]
```

Finally, applications. They have two parts: the function's name, and its argument. We've already agreed that the argument can be any full-fledged expression (including identifiers and other applications). As for the function name, it again makes sense to use the same datatype as we did when giving the function its name in a function definition. Thus:

<***app-def***> **::=**

```
[appC (fun : symbol) (arg : ExprC)]
```

identifying which function to apply, and providing its argument.

Using these definitions, it's instructive to write out the representations of the examples we defined above:

- `(fdC 'double 'x (plusC (idC 'x) (idC 'x)))`
- `(fdC 'quadruple 'x (appC 'double (appC 'double (idC 'x))))`
- `(fdC 'const5 '_ (numC 5))`

We also need to choose a representation for a set of function definitions. It's convenient to represent these by a list.

Look out! Did you notice that we spoke of a *set* of function definitions, but chose a *list* representation? That means we're using an ordered collection of data to represent an unordered entity. At the very least, then, when testing, we should use any and all permutations of definitions to ensure we haven't subtly built in a dependence on the order.

5.2 Growing the Interpreter

Now we're ready to tackle the interpreter proper. First, let's remind ourselves of what it needs to consume. Previously, it consumed only an expression to evaluate. Now it also needs to take a list of function definitions:

<***interp***> **::=**

```
(define (interp [e : ExprC] [fds : (listof FunDefC)]) : number
  <interp-body>)
```

Let's revisit our old interpreter (section 3). In the case of numbers, clearly we still return the number as the answer. In the addition and multiplication case, we still need to recur (because the sub-expressions might be complex), but which set of function definitions do we use? Because the act of evaluating an expression neither adds nor removes function *definitions*, the set of definitions remains the same, and should just be passed along unchanged in the recursive calls.

<***interp-body***> **::=**

```
(type-case ExprC e
  [numC (n) n]
  <idC-interp-case>
  <appC-interp-case>
  [plusC (l r) (+ (interp l fds) (interp r fds))]
  [multC (l r) (* (interp l fds) (interp r fds))])
```

Now let's tackle application. First we have to look up the function definition, for which we'll assume we have a helper function of this type available:

```
; get-fundef : symbol * (listof FunDefC) -> FunDefC
```

Assuming we find a function of the given name, we need to evaluate its body. However, remember what we said about identifiers and parameters? We must "search-and-replace", a process you have seen before in school algebra called *substitution*. This is sufficiently important that we should talk first about substitution before returning to the interpreter (section 5.4).

5.3 Substitution

Substitution is the act of replacing a name (in this case, that of the formal parameter) in an expression (in this case, the body of the function) with another expression (in this case, the actual parameter). Let's define its type:

```
; subst : ExprC * symbol * ExprC -> ExprC
```

It helps to also give its parameters informative names:
<***subst***> **::=**

```
(define (subst [what : ExprC] [for : symbol] [in : ExprC]) : ExprC
  <subst-body>)
```

The first argument is what we want to replace the name with; the second is for what name we want to perform substitution; and the third is in which expression we want to do it.

Do Now!

> Suppose we want to substitute `3` for the identifier `x` in the bodies of the three example functions above. What should it produce?

In `double`, this should produce `(+ 3 3)`; in `quadruple`, it should produce `(double (double 3))`; and in `const5`, it should produce `5` (i.e., no substitution happens because there are no instances of `x` in the body).

These examples already tell us what to do in almost all the cases. Given a number, there's nothing to substitute. If it's an identifier, we haven't seen an example with a *different* identifier but you've guessed what should happen: it stays unchanged. In the other cases, descend into the sub-expressions, performing substitution.

A common mistake is to assume that the result of substituting, e.g., `3` for `x` in `double` is `(define (double x) (+ 3 3))`. This is incorrect. We only substitute *at the point when we apply the function*, at which point the function's invocation is replaced by its body. The header enables us to find the function and ascertain the name of its parameter; but

Before we turn this into code, there's an important case to consider. Suppose the name we are substituting happens to be the name of a function. Then what should happen?

Do Now!

What, indeed, should happen?

There are many ways to approach this question. One is from a design perspective: function names live in their own "world", distinct from ordinary program identifiers. Some languages (such as C and Common Lisp, in slightly different ways) take this perspective, and partition identifiers into different *namespaces* depending on how they are used. In other languages, there is no such distinction; indeed, we will examine such languages soon [REF].

For now, we will take a pragmatic viewpoint. Because expressions evaluate to numbers, that means a function name could turn into a number. However, numbers cannot name functions, only symbols can. Therefore, it makes no sense to substitute in that position, and we should leave the function name unmolested irrespective of its relationship to the variable being substituted. (Thus, a function could have a parameter named `x` as well as refer to another *function* called `x`, and these would be kept distinct.)

Now we've made all our decisions, and we can provide the body code:

<***subst-body***> **::=**

```
(type-case ExprC in
  [numC (n) in]
  [idC (s) (cond
             [(symbol=? s for) what]
             [else in])]
  [appC (f a) (appC f (subst what for a))]
  [plusC (l r) (plusC (subst what for l)
                      (subst what for r))]
  [multC (l r) (multC (subst what for l)
                      (subst what for r))])
```

Exercise

Observe that, whereas in the `numC` case the interpreter returned `n`, substitution returns `in` (i.e., the original expression, equivalent at that point to writing `(numC n)`. Why?

5.4 The Interpreter, Resumed

Phew! Now that we've completed the definition of substitution (or so we think), let's complete the interpreter. Substitution was a heavyweight step, but it also does much of the work involved in applying a function. It is tempting to write

<***appC-interp-case-take-1***> **::=**

```
[appC (f a) (local ([define fd (get-fundef f fds)])
              (subst a
```

```
                    (fdC-arg fd)
                    (fdC-body fd)))]
```

Tempting, but wrong.

Do Now!

Do you see why?

Reason from the types. What does the interpreter return? Numbers. What does substitution return? Oh, that's right, expressions! For instance, when we substituted in the body of `double`, we got back the representation of `(+ 5 5)`. This is not a valid answer for the interpreter. Instead, it must be reduced to an answer. That, of course, is precisely what the interpreter does:

<***appC-interp-case***> **::=**

```
[appC (f a) (local ([define fd (get-fundef f fds)])
              (interp (subst a
                             (fdC-arg fd)
                             (fdC-body fd))
                      fds))]
```

Okay, that leaves only one case: identifiers. What could possibly be complicated about them? They should be just about as simple as numbers! And yet we've put them off to the very end, suggesting something subtle or complex is afoot.

Do Now!

Work through some examples to understand what the interpreter should do in the identifier case.

Let's suppose we had defined `double` as follows:

```
(define (double x) (+ x y))
```

When we substitute `5` for `x`, this produces the expression `(+ 5 y)`. So far so good, but what is left to substitute `y`? As a matter of fact, it should be clear from the very outset that this definition of `double` is *erroneous*. The identifier `y` is said to be *free*, an adjective that in this setting has negative connotations.

In other words, the interpreter should never confront an identifier. All identifiers ought to be parameters that have already been substituted (known as *bound* identifiers—here, a positive connotation) before the interpreter ever sees them. As a result, there is only one possible response given an identifier:

<***idC-interp-case***> **::=**

```
[idC (_) (error 'interp "shouldn't get here")]
```

And that's it!

Finally, to complete our interpreter, we should define `get-fundef`:

```
(define (get-fundef [n : symbol] [fds : (listof FunDefC)]) : FunDefC
  (cond
    [(empty? fds) (error 'get-fundef "reference to undefined function")]
    [(cons? fds) (cond
                   [(equal? n (fdC-name (first fds))) (first fds)]
                   [else (get-fundef n (rest fds))])]))
```

5.5 Oh Wait, There's More!

Earlier, we gave the following type to `subst`:

```
; subst : ExprC * symbol * ExprC -> ExprC
```

Sticking to surface syntax for brevity, suppose we apply `double` to `(+ 1 2)`. This would substitute `(+ 1 2)` for each `x`, resulting in the following expression—`(+ (+ 1 2) (+ 1 2))`—for interpretation. Is this necessarily what we want?

When you learned algebra in school, you may have been taught to do this differently: first reduce the argument to an answer (in this case, `3`), then substitute the answer for the parameter. This notion of substitution might have the following type instead:

```
; subst : number * symbol * ExprC -> ExprC
```

Careful now: we can't put raw numbers inside expressions, so we'd have to constantly wrap the number in an invocation of `numC`. Thus, it would make sense for `subst` to have a helper that it invokes after wrapping the first parameter. (In fact, our existing `subst` would be a perfectly good candidate: because it accepts any `ExprC` in the first parameter, it will certainly work just fine with a `numC`.)

Exercise

Modify your interpreter to substitute names with answers, not expressions.

We've actually stumbled on a profound distinction in programming languages. The act of evaluating arguments before substituting them in functions is called *eager* application, while that of deferring evaluation is called *lazy*—and has some variations. For now, we will actually prefer the eager semantics, because this is what most mainstream languages adopt. Later [REF], we will return to talking about the lazy application semantics and its implications.

In fact, we don't even have substitution quite right! The version of substitution we have doesn't scale past this language due to a subtle problem known as "name capture". Fixing substitution is complex, subtle, and an exciting intellectual endeavor, but it's not the direction I want to go in here. We'll instead sidestep this problem in this book. If you're interested, however, read about the *lambda calculus*, which provides the tools for defining substitution correctly.

6 From Substitution to Environments

Though we have a working definition of functions, you may feel a slight unease about it. When the interpreter sees an identifier, you might have had a sense that it needs to "look it up". Not only did it not look up anything, we defined its behavior to be an error! While absolutely correct, this is also a little surprising. More importantly, we write interpreters to *understand* and *explain* languages, and this implementation might strike you as not doing that, because it doesn't match our intuition.