Programming Languages

Evaluation

Dr Russ Ross

Utah Tech University—Department of Computing

Fall 2025

Evaluators

We're trying to implement a programming language: that is, to write an *evaluator* (i.e., something that "reduces programs to values"). It helps if we can first understand how evaluation works on paper, before we start dealing with computer complexities.

Before we get into the details, it's worth knowing that there are broadly speaking two kinds of evaluators (as well as many combinations of them). They follow very different strategies:

- An *interpreter* consumes a program and *simulates its execution*. That is, the interpreter does what we would expect "running the program" should do.
- A *compiler* consumes a program and *produces another program*. That output program must then be further evaluated.

Evaluators

That is, an interpreter maps programs in some language L to values:

interpreter :: Program$_L$ $\rightarrow$ Value

We leave open exactly what a *value* is for now, informally understanding it to be an answer the user would want to see—put differently, something that either cannot or does not need to be further e-valu-ated.

In contrast,

compiler :: Program$_L$ $\rightarrow$ Program$_T$

That is, a compiler from L to T (we use T for "target") consumes programs in L and produces programs in T. We aren't saying about how this T program must be evaluated.

- It may be interpreted directly, or it may be further compiled.
- For instance, one can compile a Scheme program to C. The C program may be interpreted directly, but it may very well be compiled to assembly.
- However, we can't keep compiling ad infinitum: at the bottom, there must be some kind of interpreter (e.g., in the computer's hardware) to provide answers.

Evaluators

Note that interpreters and compilers are themselves programs written in some language and must themselves run. Naturally, this can lead to interesting ideas and problems.

In our study, we will focus primarily on interpreters, but also see a very lightweight form of compilers. Interpreters are useful because:

1. A simple interpreter is often much easier to write than a compiler.
2. Debugging an interpreter can sometimes be much easier than debugging a compiler.

Therefore, they provide a useful "baseline" implementation technology that everyone can reach for. Compilers can often take an entire course of study.

Terminology

It is common, on the Web, to read people speak of "interpreted languages" and "compiled languages". These terms are **nonsense**.

- That isn't just a judgment; that's a literal statement: they do not make sense.
- Interpretation and compilation are techniques one uses to evaluate programs.
- A *language* (almost) never specifies how it should be evaluated.
- As a result, each implementer is free to choose whatever strategy they want.

Just as an example, C is often chosen as a canonically "compiled language", while Scheme is often presented as an "interpreted language".

- However, there have been (a handful of) interpreters for C; indeed, I used one when I first learned C.
- Likewise, there are numerous compilers for Scheme; I used one when I first learned Scheme.
- Python has several interpreters and compilers.

Terminology

Furthermore, this seemingly hard distinction is frequently broken down in practice.

- Many languages now have a "JIT", which stands for *just-in-time* compilation.
- That is, the evaluator starts out as an interpreter.
- If it finds itself interpreting the same code over and over, it compiles it and uses the compiled code instead.

When and how to do this is a complex and fascinating topic, but it makes clear that the distinction is not a bright line.

Some people are confused by the *interface* that an implementation presents. Many languages provide a *read-eval-print loop* (REPL), i.e., an interactive interface.

- It is often easier for an interpreter to do this.
- However, many systems with such an interface accept code at a prompt, compile it, run it, and present the answer back to the user; they mask all these steps.
- Therefore, the interface is not an indicator of what kind of implementation you are seeing.
- It is perhaps meaningful to refer to an *implementation* as "interactive" or "non-interactive", but that is not a reflection of the underlying language.

Terminology

In short, please remember:

- (Most) Languages do not dictate implementations. Different platforms and other considerations dictate what implementation to use.
- Implementations usually use one of two major strategies—interpretation and compilation—but many are also hybrids of these.
- A specific implementation may offer an interactive or non-interactive interface. However, this does *not* automatically reveal the underlying implementation strategy.
- Therefore, the terms “interpreted language” and “compiled language” are nonsensical.

Simulating an Interpreter by Hand

Since we have decided to write an interpreter, let's start by understanding *what* we are trying to get it to do, before we start to investigate *how* we will make it do it.

Let's consider the following program:

```
(define (f x) (+ x 1))
(f 2)
```

What does it produce? We can all guess that it produces 3. Now suppose we're asked, *why* does it produce 3? What might you say?

There's a good chance you'll say that it's because `x` gets replaced with 2 in the body of `f`, then we compute the body, and that's the answer:

```
(f 2)
→ (+ x 1) where x is replaced by 2
→ (+ 2 1)
→ 3
```

(These programs are written in Racket.)

Simulating an Interpreter by Hand

Now let's look at an extended version of the program:

```
;; f is the same as before
(define (g z)
  (f (+ z 4)))
(g 5)
```

We can use the same process:

```
(g 5)
→ (f (+ z 4)) where z is replaced by 5
→ (f (+ 5 4))
→ (f 9)
→ (+ x 1) where x is replaced by 9
→ (+ 9 1)
→ 10
```

Terminology: We call the variables in the function header the *formal parameters* and the expressions in the function call the *actual parameters*. So in `f`, `x` is the formal parameter, while 9 is an actual parameter. Some people also use *argument* in place of *parameter*, but there's no real difference between these terms.

Simulating an Interpreter by Hand

Observe that we had a choice: we could have gone either

```
→ (f (+ 5 4))
→ (f 9)
```

or

```
→ (f (+ 5 4))
→ (+ x 1) where x is replaced by (+ 5 4)
```

For now, both will produce the same *answer*, but this is actually a very consequential decision! It is in fact one of the most profound choices in programming language design.

Terminology:

- The former choice is called *eager* evaluation: think of it as "eagerly" reducing the actual parameter to a value before starting the function call.
- The latter choice is called *lazy* evaluation: think of it as not rushing to perform the evaluation.

SMoL is eager. There are good reasons for this, which we will explore later.

Simulating an Interpreter by Hand

Okay, so back to evaluation. Let's do one more step:

```
;; f is the same as before
;; g is the same as before
(define (h z w)
  (+ (g z) (g w)))
(h 6 7)
```

Simulating an Interpreter by Hand

Once again, we can look at the steps:

```
(h 6 7)
→ (+ (g z) (g w)) where z is replaced by 6 and w is replaced by 7
→ (+ (g 6) (g 7))
→ (+ (f (+ y 4)) (g 7)) where y is replaced by 6
→ (+ (f (+ 6 4)) (g 7))
→ (+ (f 10) (g 7))
→ (+ (+ x 1) (g 7)) where x is replaced by 10
→ (+ (+ 10 1) (g 7))
→ (+ 11 (g 7))
→ (+ 11 (f (+ y 4))) where y is replaced by 7
→ (+ 11 (f (+ 7 4)))
→ (+ 11 (f 11))
→ (+ 11 (+ x 1)) where x is replaced by 11
→ (+ 11 (+ 11 1))
→ (+ 11 12)
→ 23
```

Simulating an Interpreter by Hand

Observe that we again had some choices:

- Do we replace both calls at once, or do one at a time?
- If the latter, do we do the left or the right one first?

Languages have to make decisions about these, too! Above, we've again done what SMoL does:

- it finishes one call before starting the other, which makes SMoL *sequential*.
- Had we replaced both calls at once, we'd be exploring a *parallel* language.
- Conventionally, most languages choose a left-to-right order, so that's what we choose in SMoL.

Substitution

By the way, observe that you didn't need to know any computer programming to answer these questions.

- You did something similar in middle- and high-school algebra classes.
- You probably learned the phrase *substitution* for "replaced with". That's the same process we're following here.
- And indeed, we can think of programming as a natural outgrowth of algebra, except with much more interesting datatypes: not only numbers but also strings, images, lists, tables, vector fields, videos, and more.

Okay, so this gives us a way to implement an evaluator:

- Find a way to represent program source (e.g., a string or a tree).
- Look for the next expression to evaluate.
- Perform substitution (textually) to obtain a new program.
- Continue evaluating until there's nothing left but a value.

However, as you might have guessed, that's not how most programming languages *actually* work: in general it would be painfully slow. So we'll have to find a better way!

Representing Arithmetic

Let's start thinking about actually writing an evaluator. We'll start with a simple arithmetic language, and then build our way up from there.

So our language will have

- numbers
- some arithmetic operations: in fact, just addition
- and nothing more for now, so we can focus on the basics.

Over time we'll build this up.

Before we can think about the body of an evaluator, however, we need to figure out its type: in particular, what will it consume?

Representing Programs

Well, what *does* an evaluator consume? It consumes **programs**. So we need to figure out how to *represent programs*.

- Of course, computers represent programs all the time.
- When we're writing code, our text editor holds the program source.
- Every executable on disk and in memory is a representation of a program.
- When we visit a Web page, it sends down a JavaScript program.

These are all programs represented in the computer. But all these are a bit inconvenient for our needs, and we'll come up with a better representation in a moment.

Representing Programs

Before thinking about represent*ations*, let's think about what we're represent*ing*. Here are some example (arithmetic) programs:

```
1
0
-1
2.3
1 + 2
3 + 4
```

Already we have a question. How should we *write* our program? You can see where this is going: should we be writing the sum of 1 and 2 as

```
1 + 2
```

or as

```
(+ 1 2)
+ 1 2
1 2 +
```

and so on.

Representing Programs

(For that matter, we can even ask what numeral system to use for basic numbers: e..g, should we write 3 or III? You can program with the latter if you'd really like to.)

These are questions of what *surface syntax* to use. And they are very important! And interesting! And important!

- People get really attached to some surface syntaxes over the other (you may already be having some feelings about Racket's parenthetical syntax...I certainly do).
- You can even write that expression as a graphical block in Scratch and Snap!, and this syntax has been invaluable in getting young children to learn how to program without all the vagaries of textual syntax.

Thus, these are great human-factors considerations. But for now these are a distraction in terms of getting to understand the *models* underlying languages. Therefore, we need a way to represent all these different programs in a way that ignores these distinctions.

Abstract Syntax

This leads us to the first part of SImPl (the Standard Implementation Plan): the creation of what is called *abstract syntax*.

- In abstract syntax, we represent the essence of the input, ignoring the superficial syntactic details.
- Thus, in abstract syntax, all of the above programs will have the exact same representation.

An abstract syntax is an in-computer representation of programs. There are many kinds of data we can use as a representation, so let's think about the kinds of programs we might want to represent.

Abstract Syntax

For simplicity, we'll assume that our language has only numbers and addition; once we can handle that, it'll be easy to handle additional operations. Here are some sample (surface syntax) programs:

```
1
2.3
1 + 2
1 + 2 + 3
1 + 2 + 3 + 4
```

In conventional arithmetic notation, of course, we have to worry about the order of operations and what operations take precedence over what others.

- In abstract syntax, that's another detail we want to ignore;
- we'll instead assume that we are working internally with the equivalent of fully-parenthesized expressions, where all these issues have been resolved.

Abstract Syntax

Thus, it's as if the last two expressions above were written as

```
(1 + 2) + 3 or 1 + (2 + 3)
1 + ((2 + 3) + 4)
```

Observe, then, that each side of the addition operation can be a full-blown expression in its own right.

- This gives us a strong hint as to what kind of representation to use internally: a tree.
- Indeed, it's so common to use *abstract syntax trees* that the abbreviation, AST, is routinely used without explanation;
- you can expect to see it in books, papers, blog posts, etc. on this topic.

Abstract Syntax

You have quite possibly seen this idea before: it's called *sentence diagramming*. Here, for instance, is a diagram of the sentence "He studies linguistics at the university":

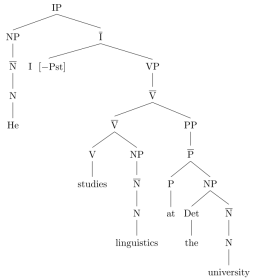

Figure 1: A sentence diagram for "He studies linguistics at the university".

An NP is a Noun Phrase, V is a Verb, and so on. Observe how the sentence diagram takes a *linear* sentence and turns it into a *tree-shaped* representation of the grammatical structure. We want to do the same for programs.

Representing Abstract Syntax

In the rest of this book, except where indicated otherwise, we will implement things in the `plait` language of Racket. Please make sure you have plait installed to follow along.

We will create a new tree datatype in `plait` to represent ASTs.

- In the sentence diagram above, the leaves of the tree are words, and the nodes are grammatical terms.
- In our AST, the leaves will be numbers, while the nodes will be operations on the trees representing each sub-expression.

For now, we have only one operation: addition. Here's how we can represent this in plait syntax:

```
(define-type Exp
  [num (n: Number)]
  [plus (left: Exp) (right: Exp)])
```

Representing Abstract Syntax

This:

```
(define-type Exp
  [num (n: Number)]
  [plus (left: Exp) (right: Exp)])
```

says:

- We are defining a new type, `Exp`
- There are two ways of making an `Exp`
- One way is through the constructor `num`:
  - A `num` takes one argument
  - That argument must be an actual number
- The other way is through the constructor `plus`:
  - A `plus` takes two arguments
  - Both arguments must be `Exp`s

Representing Abstract Syntax

If it helps as you read what follows, this

```
(define-type Exp
  [num (n: Number)]
  [plus (left: Exp) (right: Exp)])
```

is very analogous to the following Java pseudocode skeleton:

```
abstract class Exp {}

class num extends Exp {
    num (Number n) { }
}

class plus extends Exp {
    plus (Exp left, Exp right) { }
}
```

(or the analog with Python dataclasses)

Representing Abstract Syntax

Let's look at how some of the previous examples would be represented:

Surface Syntax	**AST**
`1`	`(num 1)`
`2.3`	`(num 2.3)`
`1 + 2`	`(plus (num 1) (num 2))`
`(1 + 2) + 3`	`(plus (plus (num 1) (num 2))` `(num 3))`
`1 + (2 + 3)`	`(plus (num 1)` `(plus (num 2) (num 3)))`

`1 + ((2 + 3) + 4)`	`(plus (num 1)`
	`(plus (plus (num 2)`
	`(num 3))`
	`(num 4)))`

Representing Abstract Syntax

Observe a few things about these examples:

- The datatype definition does not let us *directly* represent surface syntax terms such as `1 + 2 + 3 + 4`; any ambiguity has to be handled by the time we construct the corresponding AST term.
- The number representation might look a bit odd: we have a `num` constructor whose only job is to "wrap" a number. We do this for consistency of representation.
- Notice that every significant part of the expression went into its AST representation, though not always in the same way. In particular, the + of an addition is represented by the *constructor*; it is not part of the parameters.
- The AST really doesn't care what surface syntax was used. The last term could instead have been written as

```
(+ 1
   (+ (+ 2 3)
      4))
```

  and it would presumably produce the same AST.

Representing Abstract Syntax

In short, ASTs are tree-structured data that **represent programs in programs**. This is a profound idea! It's one of the great ideas of the 20th century, building on

- the brilliant work of Gödel (encoding),
- Turing (universal machine),
- von Neumann (stored program computer),
- and McCarthy (metacircular interpreter).

Aside: Not every part of the source program has been represented in the AST.

- For instance, presumably both `1 + 2` and `1  +  2` would be represented the same way, ignoring the spaces.
- In practice, a real language implementation does need to know something about the syntax:
  - for instance, to highlight pieces of the program source when there is an error, as DrRacket does.

Therefore, real-world implementations use abstract syntax but with metadata relating it back to the source.

Defining the Evaluator

Having seen how to represent arithmetic programs, we turn to writing an evaluator program that turns them into answers.

- What is the type of this evaluator?
- Clearly it consumes programs, which here are represented by `Exp`s.
- What does it produce? In this case, all these expressions are going to produce numbers.
- For this reason, we'll call this a calculator, or `calc` for short, for now.

We can thus give `calc` the type

```
(calc : (Exp -> Number))
```

Defining the Evaluator

Let's now try to define its body. Clearly we must have

```
(define (calc e)
  …)
```

In the body, given an `Exp`, we will want to take it apart using `type-case`, which tells us there are two options, each with some additional data (this is the moral equivalent of the method dispatch we'd have used in Python or C++):

```
(type-case Exp e
  [(num n) …]
  [(plus l r) …])
```

What happens in the case that the whole expression is already a number? Well, we have our answer, so we just return it. Otherwise, we have to add the two sides, giving an overall body of:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ l r)]))
```

Defining the Evaluator

Let's run it to...

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ l r)]))
```

...oops! We get a type error! It tells us that addition is expecting a number, but `l` is not a number: it's an `Exp`.

- Ah, that's because `l` and `r` still represent *expressions*, not the *answer* that the expressions evaluate to.
- To fix that, we need something that can turn an expression into a number...which is precisely what we're defining!

Thus, we instead write

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]))
```

The type-checker is happy now.

Defining the Evaluator

And sure enough, we can confirm that our examples produce what we expect. For instance:

```
(calc (num 1))
```

produces 1,

```
(calc (plus (num 1) (num 2)))
```

produces 3, and

```
(calc (plus (num 1)
            (plus (num 2) (num 3))))
```

produces 6.

Defining the Evaluator

Aside: We've glossed over a detail: we've assumed that + always means numeric addition (which was already implicit in calling it "plus" in the AST).

- But some languages allow any number of different types to be "added": e.g., it can also concatenate strings.
- In such languages, the name in the AST might be something more generic, and the evaluator would need to handle the different possible behaviors.

In fact, we've glossed over something even more basic: what numeric addition means, or for that matter, even what numbers are.

- In our calculator, we have adopted numbers from `plait` (in `num`) and addition from `plait` (by using +).
- Those places in `calc` also tell us where we would go to change those choices.

Testing the Evaluator

The examples above are fine, but we should write these in the syntax of tests, so that the computer checks them for us automatically:

```
(test (calc (num 1)) 1)
(test (calc (num 2.3)) 2.3)
(test (calc (plus (num 1) (num 2))) 3)
(test (calc (plus (plus (num 1) (num 2))
                  (num 3)))
      6)
(test (calc (plus (num 1)
                  (plus (num 2) (num 3))))
      6)
(test (calc (plus (num 1)
                  (plus (plus (num 2)
                              (num 3))
                        (num 4))))
      10)
```

Sure enough, when we run this, Racket confirms that all these tests pass.

Testing the Evaluator

Pro Tip: It can get annoying to scan through all this testing output to see whether any of the tests failed. Simply add

```
(print-only-errors #true)
```

before your tests and Racket will suppress reporting on the passing tests, so you can focus on the ones that failed: in other words, no news is good news.

In general, test early, often, and extensively.

- Programming language evaluators translate our thoughts into computer actions.
- Therefore, it's critical that they do so precisely.
- This is why language implementations are some of the most tested software you can imagine (when's the last time you were stopped by a bug in your language implementation?), and people who will tolerate bugs in just about any other software are much less forgiving of bugs in implementations.

Some Subtler Tests

Try the following test:

```
(test (calc (plus (num 0.1) (num 0.2))) 0.3)
```

It succeeds! Are we happy? Suppose we instead write it as:

```
(test (calc (plus (num 0.1) (num 0.2))) 1/3)
```

As expected, it fails:

- but the error message reveals that the left-hand side evaluated to `0.30000000000000004`.
- This should be a cue that we have actually gotten *floating point* addition.
- This is because `plait` treats numbers written with a decimal point, like `0.1`, as floating point bitstrings.
- However, floating point bitstrings cannot precisely represent the number 0.3.

In fact, plait's test allows a little bit of numeric slack so that the passing test above works. (This is because in plait, `0.3` really does precisely represent the number 0.3, because it was written literally and not the result of a floating-point computation.)

Some Subtler Tests

This reinforces a point we made in passing above and was therefore easy to miss:

- By adopting `plait`'s primitives, we have also inherited its semantics.
- This may or may not be what we wanted!
- Therefore, when writing an evaluator using a host language, we have to make sure that its semantics are the one we want, otherwise we could be in for an unpleasant surprise.
- If we want different behavior, we have to implement it explicitly.

Conclusion

This concludes our first look at SImPl:

- We have *represented* a program in a program.
- We have *processed* that represented program in a program.
- We have just written our first program that processes programs—now we're off to the races!

The Problem

Earlier we went through the basic steps of the SImPl, but we left open a big question: how do we get programs *into* the AST representation?

Of course, the simplest way is what we already did: to write the AST constructors directly, e.g.,

```
(num 1)
(plus (num 1) (num 2))
(plus (num 1) (plus (num 2) (num 3)))
```

which, as we noted, has the virtue of also ignoring exactly how the program source was written.

However, this can get very tedious.

- We don't want to have to write `(num ...)` every time we want to write a number, for instance!
- In particular, the more tedious it is the less likely we are to write many or complex tests, and that would be especially unfortunate.

The Problem

Therefore, we'd like a more convenient surface syntax, along with a *program* to translate that into ASTs.

- As we have already seen, there is a large number of surface syntaxes we can use,
- And we aren't even limited to textual syntax: it could be graphical; spoken; gestural (imagine you're in a virtual reality environment); and so on.
- As we have noted, this wide range of modalities is important—especially so if the programmer has physical constraints—but it's outside the range of our current study.

The Problem

Even with textual syntax, we have to deal with issues like ambiguity (e.g., order of operations in arithmetic).

- In general, the process of converting the input syntax into ASTs is called *parsing*.
- We could write a whole booklet just on parsing...so we won't.
- Instead, we're going to pick one syntax that strikes a reasonable balance between convenience and simplicity, which is the parenthetical syntax of Racket, and has special support in `plait`.

The Problem

That is, we will write the above examples as

```
1
(+ 1 2)
(+ 1 (+ 2 3))
```

and see how Racket can help us make these convenient to work with.

In fact, in this book we will follow a convention (that Racket doesn't care about, because it treats `()`, `[]`, and `{}` interchangeably): we'll write programs to be represented using `{}` instead of `()`.

Thus, the above three programs become

```
1
{+ 1 2}
{+ 1 {+ 2 3}}
```

S-Expressions

There is a name for this syntax: these are called *s-expressions* (the *s-* is for historical reasons).

- In `plait`, we will write these expressions *preceded by a back-tick* (`` ` ``).
- A back-tick followed by a Racket term is of type `S-Exp`. Here are examples of s-expressions:

```
`1
`2.3
`-40
```

- These are all numeric s-expressions. We can also write

```
`{+ 1 2}
`{+ 1 {+ 2 3}}
```

S-Expressions

It's not obvious, but these are actually list s-expressions. We can tell by asking

```
> (s-exp-list? `1)
- Boolean
#f
> (s-exp-list? `{+ 1 2})
- Boolean
#t
> (s-exp-list? `{+ 1 {+ 2 3}})
- Boolean
#t
```

So the first is not but the second two are; similarly,

```
> (s-exp-number? `1)
- Boolean
#t
> (s-exp-number? `{+ 1 {+ 2 3}})
- Boolean
#f
```

S-Expressions

The `S-Exp` type is a container around the actual number or list, which we can extract:

```
> (s-exp->number `1)
- Number
1
> (s-exp->list `{+ 1 2})
- (Listof S-Exp)
(list `+ `1 `2)
```

Do Now:

- What happens if you apply `s-exp->number` to a list s-exp
- or `s-exp->list` to a number s-expression?
- Or either to something that isn't an s-expression at all?

Try it right now and find out! Do you get somewhat different results?

S-Expressions

Let's look at that last output above a bit more closely.

The resulting list has three elements, two of which are numbers, but the third is something else:

`+

is a symbol s-expression.

- Symbols are like strings but somewhat different in operations and performance.
- Whereas there are numerous string operations (like substrings), symbols are treated atomically;
- other than being converted to strings, the only other operation they support is equality.
- But in return, symbols can be checked for equality in *constant* time.

S-Expressions

Symbols have the same syntax as Racket variables, and hence are perfect for representing variable-like things. Thus

```
> (s-exp-symbol? `+)
- Boolean
#t
> (s-exp->symbol `+)
- Symbol
'+
```

This output shows how symbols are written in Racket: with a single-quote (`'`).

There are other kinds of s-expressions as well, but this is all we need for now! With this, we can write our first parser!

Primus Inter Parsers

Do Now: Think about what type we want for our parser.

- What does our parser need to produce? Whatever the calculator consumes, i.e., `Exp`.
- What does it consume? Program source expressions written in a "convenient" syntax, i.e., `S-Exp`.
- Hence, its type must be

  ```
  (parse : (S-Exp -> Exp))
  ```

- That is, it converts the human-friendly(ier) syntax into the computer's internal representation.

Primus Inter Parsers

Writing this requires a certain degree of pedantry. First, we need a conditional to check what kind of s-exp we were given:

```
(define (parse s)
  (cond
    [(s-exp-number? s) ...]
    [(s-exp-list? s) ...]))
```

If it's a numeric s-exp, then we need to extract the number and pass it to the `num` constructor:

```
(num (s-exp->number s))
```

Primus Inter Parsers

Otherwise, we need to extract the list and check whether the first thing in the list is an addition symbol. If it is not, we signal an error:

```
(let ([l (s-exp->list s)])
  (if (symbol=? '+
                (s-exp->symbol (first l)))
    ...
    (error 'parse "list not an addition")))
```

Otherwise, we create a `plus` term by recurring on the two sub-pieces.

```
(plus (parse (second l))
      (parse (third l)))
```

Primus Inter Parsers

Putting it all together:

```
(define (parse s)
  (cond
    [(s-exp-number? s)
     (num (s-exp->number s))]
    [(s-exp-list? s)
     (let ([l (s-exp->list s)])
       (if (symbol=? '+
                     (s-exp->symbol (first l)))
         (plus (parse (second l))
               (parse (third l)))
         (error 'parse "list not an addition")))]))
```

It's all a bit much, but fortunately this is about as hard as parsing will get in this book!

Everything you see from now on will basically be this same sort of pattern, which you can freely copy.

Primus Inter Parsers

We should, of course, make sure we've got good tests for our parser. For instance:

```
(test (parse `1) (num 1))
(test (parse `2.3) (num 2.3))
(test (parse `{+ 1 2}) (plus (num 1) (num 2)))
(test (parse `{+ 1
                 {+ {+ 2 3}
                    4}})
      (plus (num 1)
            (plus (plus (num 2)
                        (num 3))
                  (num 4))))
```

Primus Inter Parsers

Do Now: Are there other kinds of tests we should have written?

We have only written *positive* tests. We can also write *negative* tests for situations where we expect errors:

```
(test/exn (parse `{1 + 2}) "")
```

- `test/exn` takes a string that must be a substring of the error message.
- You might be surprised that the test above uses the empty string rather than, say, "addition".
- Try out this example to investigate why. How can you improve your parser to address this?

Other situations we should check for include there being too few or too many sub-parts.

- Addition, for instance, is defined to take exactly two sub-expressions.
- What if a source program contains none, one, three, four, …?
- This is the kind of pedantry that parsing calls for.

Primus Inter Parsers

Once we have considered these situations, we're in a happy place, because `parse` produces output that `calc` can consume.

- We can therefore compose the two functions!
- Better still, we can write a helper function that does it for us:

```
(run : (S-Exp -> Number))

(define (run s)
  (calc (parse s)))
```

Primus Inter Parsers

So we can now rewrite our old evaluator tests in a much more convenient way:

```
(test (run `1) 1)
(test (run `2.3) 2.3)
(test (run `{+ 1 2}) 3)
(test (run `{+ {+ 1 2} 3})
      6)
(test (run `{+ 1 {+ 2 3}})
      6)
(test (run `{+ 1 {+ {+ 2 3} 4}})
      10)
```

Compare this against the `calc` tests we had earlier!

Extending the AST

So far our language has had only arithmetic. We will now examine how to extend our language to also support conditionals.

In SImPl, we have to do at least two things:

1. Extend the datatype representing expressions to include conditionals.
2. Extend the evaluator to handle (the representation of) these new expressions.

Optionally, if we have a parser, we should also

3. Extend the parser to produce these new representations.

Extending the AST

Because we have fixed our conditionals to have three parts, we just need to represent that in the AST. This is straightforward:

```
(define-type Exp
  [num (n: Number)]
  [plus (left: Exp) (right: Exp)]
  [cnd (test: Exp) (then: Exp) (else: Exp)])
```

The real work will happen in the evaluator.

Extending the Calculator

Clearly, adding conditionals doesn't change what our calculator previously did, we can leave that intact, and just focus on the handling of `if`:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) ...]))
```

Extending the Calculator

Indeed, we can recursively evaluate each term, in case it's useful:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) ... (calc c) ... (calc t) ... (calc e) ...]))
```

But now we run into a problem. What is the result of calling `(calc c)`?

- We expect it to be some kind of Boolean value. But we don't have Boolean values in the language!
- That's not all. Above, we have written both `(calc t)` and `(calc e)`. However, the whole point of a conditional is that we don't want to evaluate both, only one.

The Design Space of Conditionals

Even the simplest conditional exposes us to many variations in language design. The intent is that *test-expression* is evaluated first; if it results in a true value then (only) the *then-expression* is evaluated, else (only) the *else-expression* is evaluated. However, this simple construct results in at least three different, mostly independent design decisions:

1. What kind of values can the *test-expression* be?
2. What kind of terms are the branches?
3. If the branches are expressions and hence allowed to evaluate to values, how do the values relate?

The Design Space of Conditionals

Initially, it may seem attractive to design a language with several *truthy* and *falsy* values: after all, this appears to give the programmer more convenience. However, this can lead to bewildering inconsistencies across languages:

Value	**JavaScript**	**Python**	**Ruby**
-1	truthy	truthy	truthy
0	falsy	falsy	truthy
“”	falsy	falsy	truthy
“0”	truthy	truthy	truthy
NaN	falsy	truthy	truthy
nil, null, None, undefined	falsy	falsy	falsy
[]	truthy	falsy	truthy
empty map or object	truthy	falsy	truthy

The Design Space of Conditionals

Of course, it need not be so complex.

- Scheme, for instance, has only one value that is falsy: `false` itself (written as `#false`). Every other value is truthy.
- For those who value allowing non-Boolean values in conditionals, this represents an elegant trade-off: it means a function need not worry that a type-consistent value resulting from a computation might cause a conditional to reverse itself.
- Note that Ruby, which is inspired in part by Scheme, adopted this simple model.
- Lua, another Scheme-inspired language, is also spartan in its falsy values.

Using Truthy-Falsy Values

Some languages use truthy-falsy values to handle partial functions. Instead of signaling an error, they return a falsy value when the argument cannot be handled. Consider this Racket example:

```
(define (g s)
  (+ 1 (or (string->number s) 0)))
```

This function accepts a string that may or may not represent a number. If it does, it returns one bigger number; otherwise it returns 1. This works because `string->number` returns a number or, if the string is not legal, `#false`. These therefore serve as a rough-and-ready option types.

Implementing Conditionals

Okay, so we have many decisions to make! To first get a working evaluator, without having to go beyond numbers, we can use a slightly different conditional construct: one that checks whether the conditional evaluates to a special numeric value, such as 0. We can just reuse `plait`'s conditional:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) (if
                  (zero? (calc c))
                  (calc t)
                  (calc e))]))
```

Implementing Conditionals

Observe that the semantics of the conditional—that 0 is true, and everything else is false—is now made manifest in the body of `calc`. If we want a different semantics, that's the part of the program to change.

This solution might feel a bit… disappointing?

- This is true!
- This is not entirely true. We have made some conscious decisions, like the handling of conditionals.
- We just happened to defer those to `plait`, but we could have made other decisions if we wanted.
- This reuse is actually part of the power of an interpreter: it lets you exploit features that have already been built instead of having to re-implement all of them from scratch.
- By reusing the host language, we can zero in on the differences.

Adding Booleans

Okay, so what if we wanted proper Booleans? To employ SImPl, we need to alter the AST, the evaluator, and the parser. We can add Booleans much like we did numbers: with a constructor that wraps a `plait` representation of the Boolean.

```
(define-type Exp
  [num (n: Number)]
  [bool (b: Boolean)]
  [plus (left: Exp) (right: Exp)]
  [cnd (test: Exp) (then: Exp) (else: Exp)])
```

Adding Booleans

It's very important to keep in mind what the `num` and `bool` constructors stand for.

- This is abstract syntax: we are just (abstractly) representing the program that the user wrote, not the result of its evaluation.
- Therefore, these constructors are capturing syntactic constants in the source program: values like 3.14 and -1 for the former and `#true` and `#false` for the latter.
- They do not represent compound expressions that will evaluate to numbers or Booleans.
- **Aside**: The abstract syntax does not dictate what concrete syntax we use. We might write Boolean values as `#t`, `#true`, `true`, `True`, etc. This is precisely the abstraction that abstract syntax provides!

Adding Booleans

Easy peasy! This naturally suggests what we should do in the evaluator:

```
(define (calc e)
  (type-case Exp e
    [(num n) n]
    [(bool b) b]
    [(plus l r) (+ (calc l) (calc r))]
    [(cnd c t e) (if
                  (zero? (calc c))
                  (calc t)
                  (calc e))]))
```

Oh…oops. This version of `calc` doesn't type-check, because our calculator is supposed to return only numbers, not Booleans! We want to build full-fledged programming languages. They have a wide range of values: numbers, Boolean, strings, images, functions, and more.

The Value Datatype

Therefore, we first need to define a datatype that reflects the different kinds of values that an evaluator can produce. We will follow a convention and call the return type constructors `...V` and the input expression constructors `...E`.

First we'll rename our expressions:

```
(define-type Exp
  [numE (n: Number)]
  [boolE (b: Boolean)]
  [plusE (left: Exp) (right: Exp)]
  [cndE (test: Exp) (then: Exp) (else: Exp)])
```

The Value Datatype

Now we introduce a `Value` datatype to represent the types of answers our evaluator can produce:

```
(define-type Value
  [numV (the-number: Number)]
  [boolV (the-boolean: Boolean)])
```

We update the type of our evaluator:

```
(calc : (Exp -> Value))
```

and the early parts are easy:

```
(define (calc e)
  (type-case Exp e
    [(numE n) (numV n)]
    [(boolE b) (boolV b)]
    ...))
```

Updating the Evaluator

Now suppose we try to use our existing code:

```
[(plusE l r) (+ (calc l) (calc r))]
```

This has two problems.

- The first is we can't return a number; we have to return a `numV`:

  ```
  [(plusE l r) (numV (+ (calc l) (calc r)))]
  ```

- But now we run into a subtler problem. The type-checker is not happy because the result of `calc` is a `Value`, and `+` consumes only `Numbers`. The type checker is forcing us to make a decision here: what happens if one of the sides of `+` does not evaluate to a number?

Updating the Evaluator

First, let's build an abstraction to handle this, so that we can keep the core of the interpreter relatively clean:

```
[(plusE l r) (add (calc l) (calc r))]
```

Now we can defer all the logic of evaluating + to add. The least-non-standard policy is to require both branches to evaluate to numbers:

```
(define (add v1 v2)
  (type-case Value v1
    [(numV n1)
     (type-case Value v2
       [(numV n2) (numV (+ n1 n2))]
       [else (error '+ "expects RHS to be a number")])]
    [else (error '+ "expects LHS to be a number")]))
```

Updating the Evaluator

Pro Tip: You've just added a complex chunk of code. Now would be a very good time to test your evaluator. Here are two things to consider:

- Right now the code for conditionals also does not type-check. You may find it convenient to replace the entire RHS with something semantically incorrect but type-correct, like `(numV 0)`, to restore your working evaluator.
- Don't forget to test for the error cases! You would do so using `test/exn`. For instance:

```
(test/exn (calc (plusE (numE 4) (boolE #false))) "RHS")
```

Updating the Evaluator

Let's now turn our attention to the conditional. The core logic must clearly be similar: check something about the condition, and based on it, evaluate only one of the other two clauses. We can again defer the decision about truthiness to a helper function:

```
[(cndE c t e) (if (boolean-decision (calc c))
                  (calc t)
                  (calc e))]
```

Updating the Evaluator

Again, the least non-standard policy is to be strict about requiring a Boolean:

```
(define (boolean-decision v)
  (type-case Value v
    [(boolV b) b]
    [else (error 'if "expects conditional to evaluate to a boolean")]))
```

But again, starting from a strict interpretation, we can see where we can give in to any urges we feel to design a more liberal semantics: by replacing the `else` clause. Observe that we did something different here than for addition: the entire point of a conditional is to *not* evaluate one of the branches!

Evaluating Local Binding

Most programming languages have some notion of *local binding*. There are two words there, which we'll tease apart:

- *Binding* means to associate names with values. For instance, when we call a function, the act of calling associates ("binds") the formal parameters with the actual values.
- *Local* means they are limited to some region of the program, and not available outside that region.

For instance, in many languages we can write something like

```
fun f(x):
  y = 2
  x + y
```

Evaluating Local Binding

This seems clear enough. But here is a more subtle program:

```
fun f(x):
  for(y from 0 to 10):
    ...
  print(x + y)
```

Is that legal?

- It depends on whether the `y` is still "alive" or "active" or "visible" or whatever other phrase you would like;
- formally, we would say, it depends on whether `y` is *in scope*.
- Specifically, we'd ask whether the last `y` is a *bound* instance of the *binding* that takes place in the `for`.

This is complicated!

- Many languages do rather odd, complicated, and certainly unintuitive things.
- These odd things are not really part of SMoL;
- if anything, they are a violation of it.

A Syntax for Local Binding

Part of the problem is actually syntactic.

- When we write a program like the above, there's no clear beginning or ending of the scope of `y`.
- This is actually a great virtue of parenthetical syntax: it suggests a clear region (between the parentheses).

Following the syntax of Racket, we'll add a new construct to our language. It's getting a bit tricky to keep track of the full syntax, so we'll use a notation called BNF (short for Backus-Naur Form).

Let's start with our arithmetic language:

```
<expr> ::= <num>
         | {+ <expr> <expr>}
```

which reads as "define (`::=`) `expr` (short for expression) to be either

- a number
- or (`|`) the surface syntax consisting of `{`, `+`, an `expr`, another `expr`, and `}`".

A Syntax for Local Binding

BNF gives us a convenient notation for the grammar of a language through its concrete syntax, and

- our abstract syntax will usually correspond very directly to the BNF in a very natural manner.
- (Observe, however, that in the BNF, we simply say that each sub-expression is an expr, because that's all we need to know to properly form programs.
- However, in the AST, we give the parts different names to tell them apart.)

A Syntax for Local Binding

Notation: BNF is divided into *terminals* and *non-terminals*.

Non-terminals are placeholders like `expr` and `num` above:

- they stand for many more possibilities
  - (an expr above can be replaced with one of two possibilities (for now), while
  - there are many possible ways to write `nums`).
- They are given this name because the grammar doesn't "terminate" here:
- the name is a place-holder that can (and must) be further expanded.
- The convention is to write non-terminals inside `<pointy brackets>`.

Terminals, in contrast, are concrete syntax, like `{`, `}`, and `+` above.

- They are so-called because they stand for themselves and can't be expanded further.
- They are sometimes also called *literals*, because they must be written literally as shown.
- For this reason, they are not surrounded by any decorative symbols.
- Everything is written literally unless it's a non-terminal, in which case it's replaced by something according to the definition of the non-terminal.

A Syntax for Local Binding

Now we can define an extended language:

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
```

That is, we're adding a new language construct, `let1`, which has three parts:

- a variable (`var`) and
- two expressions (the two `expr`'s).

The Meaning of Local Binding

Do Now: Here are some examples of this new construct; what do you *expect* each one to produce?

```
{let1 {x 1}
  {+ x x}}

{let1 {x 1}
  {let1 {y 2}
    {+ x y}}}

{let1 {x 1}
  {let1 {y 2}
    {let1 {x 3}
      {+ x y}}}}
```

The Meaning of Local Binding

What do you *expect* each one to produce?

```
{let1 {x 1}
  {+ x
     {let1 {x 2} x}}}

{let1 {x 1}
  {+ {let1 {x 2} x}
     x}}

x
```

The Meaning of Local Binding

Do Now: Oh, did you notice something? None of the above programs is syntactically legal! Why?

It's because there is no syntax yet for variables. Our syntax permits us to *bind* variables but not to *use* them. So we have to fix that:

```
<expr> ::= <num>
         | {+ <expr> <expr>}
         | {let1 {<var> <expr>} <expr>}
         | <var>
```

Now the above terms are all syntactically valid, so we can go back to the question of what they should evaluate to.

The Meaning of Local Binding

The first two programs are pretty obvious:

```
{let1 {x 1}
  {+ x x}}
```

should evaluate to 2, and

```
{let1 {x 1}
  {let1 {y 2}
    {+ x y}}}
```

should evaluate to 3.

The Meaning of Local Binding

How about this program?

```
{let1 {x 1}
  {let1 {y 2}
    {let1 {x 3}
      {+ x y}}}}
```

Here we see the advantage of the parenthetical notation. In a more conventional syntax, this might correspond to

```
x = 1
y = 2
x = 3
x + y
```

where any number of things could happen:

- we might have two different `x`'s;
- we might have an `x` bound and then modified;
- and in some languages, an introduction of `x` could be "lifted" so that it's no longer clear which `x` is most recent.

The Meaning of Local Binding

With our parenthetical syntax, though, it's pretty clear what scopes we want.

```
{let1 {x 1}
  {let1 {y 2}
    {let1 {x 3}
      {+ x y}}}}
```

To determine the value, we can rely on our old friend, substitution.

- However, when we substitute the outer `x`, we expect that to stop at the point where the inner `x` begins:
- that is, the inner `x` shadows the outer one.

Hence, the result should be 5.

The Meaning of Local Binding

Do Now:

- The example above is uninteresting in that the outer `x` never sees any use.
- What kind of program might we write that has two let bindings of `x` that lets us clearly see that there are two `x`'s?

That's what this program shows:

```
{let1 {x 1}
  {+ x
     {let1 {x 2} x}}}
```

It seems fairly clear that

- the left `x` in the addition should be `1`, while
- the `x` in the right expression should be shadowed
- and hence should evaluate to `2`.

The sum should therefore be `3`.

The Meaning of Local Binding

Now for a more complex example:

```
{let1 {x 1}
  {+ {let1 {x 2} x}
     x}}
```

Here, it's especially useful to turn to substitution to determine the answer. Again,

- It seems clear that x in the left expression is shadowed and hence should be 2. The big question, of course, is
- What about the x on the right hand side of the addition (i.e., on the last line)?

Here, again, conventional textual syntax is fraught with ambiguity: is

```
x = 2
```

on the left a binding of a new x or a modification of the outer x? Those are two very different things!

The Meaning of Local Binding

A more complex example (continued):

```
{let1 {x 1}
  {+ {let1 {x 2} x}
     x}}
```

But with our syntax it's much clearer that it should be the former, not the latter.

Thus, by substitution, the outer `x` is replaced by `1`, giving

```
{+ {let1 {x 2} x}
   1}
```

in which we perform one more substitution, producing

```
{+ 2
   1}
```

and hence 3.

Static Scoping

The program

```
{let1 {x 1}
  {+ {let1 {x 2} x}
     x}}
```

introduces us to a very important concept: indeed, one of the central ideas behind SMoL. This is that

- A variable's binding is determined by *its position in the source program*, and **not** by *the order of the program's execution*.
- That is, the x on the last line is bound by the same place—and hence obtains the same value—irrespective of other bindings that took place before it was evaluated.

Static Scoping

To understand this better, let's see a progression of programs.

```
{let1 {x 1}
  {+ {let1 {x 2} x}
     x}}
```

You might think it's okay whether this produces 3 or 4. How about this?

```
{let1 {x 1}
  {+ {if true
         {let1 {x 2} x}
         4}
     x}}
```

You should expect the same out of this:

- the conditional is always true, so clearly we are always going to evaluate the inner binding, so
- its answer should be the same as for the previous program.

Static Scoping

But how about this?

```
{let1 {x 1}
  {+ {if true
         4
         {let1 {x 2} x}}
     x}}
```

Now you might not be so sure.

- Since the conditional is never taken, you probably don't want the inner binding to have an influence.
- That is, you are willing to *let the program's control flow influence the bindings*.

On its face that sounds reasonable.

Static Scoping

But now how about this program?

```
{let1 {x 1}
  {+ {if {random}
         4
         {let1 {x 2} x}}
     x}}
```

or

```
{let1 {x 1}
  {+ {if {moon-is-currently-full}
         4
         {let1 {x 2} x}}
     x}}
```

Are you okay with the binding structure changing every two weeks?

Static Scoping

What about this version:

```
{let1 {x 1}
  {+ {if {moon-is-currently-full}
         4
         {let1 {y 2} x}}
     y}}
```

Then, depending on the phase of the moon, the program either produces an answer or results in an unbound-variable error.

Static Scoping

The decision to let control flow determine binding is called *dynamic scope*.

- It is the one **unambiguously wrong** design decision in programming languages.
- It has a long and sordid history: the original Lisp had it, and it was not until over a decade later that Scheme fixed it.
- Unfortunately, those who don't know history are doomed to repeat it: early versions of Python and JavaScript also had dynamic scope. Taking it back out has been a herculean effort.

Dynamic scope means:

- We can't be sure about the binding structure of our programs.
- The evaluator can't be sure, either.
- Nor can programmer tools.

For instance, a program refactoring tool needs to know binding structure:

- even a simple "variable renaming" tool needs to know which variables to rename.
- In DrRacket, there is no ambiguity, so variable renaming works correctly.
- This is not true in other languages: see, for instance, Appendix 2 of this paper on the semantics of Python.

Static Scoping

The opposite of dynamic scope—where we can determine the binding by following the structure of the AST—is called *static scope*.

Static scope is a defining characteristic of SMoL.

Dynamic scope occurred in early implementations because

- it was easy to obtain: it was the default behavior.
- We have to work a bit harder to obtain static scope, as we will see.

An Evaluator for Local Binding

Now that we've seen what behavior we want, we should implement it.

- That is, we'll extend our calculator to handle local binding (a feature you may well have wished your calculator had).
- To reflect that our calculator is growing up, from now on we'll call it an *interpreter*, abbreviated in code to `interp`.

Let's start with the new AST. For simplicity, we'll ignore conditionals, which are anyway orthogonal to our goal of handling local binding.

Recall that we added two new branches to the BNF, so we'll want two new corresponding branches to the AST:

```
(define-type Exp
  [numE (n: Number)]
  [plusE (left: Exp) (right: Exp)]
  [varE (name: Symbol)]
  [let1E (var: Symbol)
         (value: Exp)
         (body: Exp)])
```

An Evaluator for Local Binding

We can also copy over our previous calculator, but we pretty quickly run into trouble:

```
(define (interp e)
  (type-case Exp e
    [(numE n) n]
    [(varE s) ...]
    [(plusE l r) (+ (interp l) (interp r))]
    [(let1E var val body) ...]))
```

What do we do when we encounter a `let1E`? For that matter, what do we do when we encounter a variable?

In fact, these two should be intimately connected:

- the variable binding introduced by the former
- should substitute the variable use in the latter.

Caching Substitution

We repeatedly—and rightly—refer back to substitution to understand how programs should work, and indeed will do so again later. But substitution as an *evaluation* technique is messy.

- This requires us to constantly keep rewriting the program text,
- which takes time linear in the size of the program (which can get quite large)
- for *every* variable binding.

Most real language implementations do not work this way.

Caching Substitution

Instead, we might think of employing a space-time tradeoff:

- we'll use a little extra space to save ourselves a whole lot of time.
- That is, we'll *cache* the substitution in a data structure called the *environment*.
- An environment records names and their corresponding values: that is,
- it's a collection of key-value pairs.

Thus,

- whenever we encounter a binding we remember its value, and
- when we encounter a variable, we look up its value.

Aside: As with all caches,

- we want them to only improve performance along a dimension,
- not change the meaning. That is,
- we no longer want substitution to define how we produce an answer. But,
- we still want it to tell us what answer to produce.

This will become important below.

Caching Substitution

We will use a hash table to represent the environment:

```
(define-type-alias Env (Hashof Symbol Value))
(define mt-env (hash-empty)) ;; "empty environment"
```

We will need the interpreter to actually take an environment as a formal parameter, to use in place of substitution. Thus:

```
(interp : (Exp Env -> Value))
(define (interp e nv) ...)
```

Caching Substitution

Now what happens when we encounter a variable?

- We try to look it up in the environment.
- That may succeed or, in the case of our last example above, fail.

We will use `hash-ref`, which

- looks up keys in hash tables, and
- returns an `Optionof` type to account for the possibility of failure.

We can encapsulate it in a function that we will repeatedly find useful:

```
(define (lookup (s : Symbol) (n : Env))
  (type-case (Optionof Value) (hash-ref n s)
    [(none) (error s "not bound")]
    [(some v) v]))
```

In the event the lookup succeeds, then we want the value found, which is wrapped in `some`.

Caching Substitution

This function enables our interpreter to stay very clean and readable:

```
[(varE s) (lookup s nv)]
```

Finally, we are ready to tackle `let1`. What happens here? We must

- evaluate the body of the expression, in
- an environment that has been extended, with
- the new name
- bound to its value.

Phew!

Caching Substitution

Fortunately, this isn't as bad as it sounds. Again, a function will help a lot:

```
(extend : (Env Symbol Value -> Env))
(define (extend old-env new-name value)
  (hash-set old-env new-name value))
```

With this, we can see the structure clearly:

```
[(let1E var val body)
 (let ([new-env (extend nv
                        var
                        (interp val nv))])
```

(Observe that we used `let` in plait to define `let1`. We'll see more of this...)

Caching Substitution

In sum, our core interpreter is now:

```
(define (interp e nv)
  (type-case Exp e
    [(numE n) (numV n)]
    [(varE s) (lookup s nv)]
    [(plusE l r) (+ (interp l nv) (interp r nv))]
    [(let1E var val body)
     (let ([new-env (extend nv
                            var
                            (interp val nv))])
       (interp body new-env))]))
```

Exercise:

1. What if we had not called `(interp val nv)` above?
2. What if we'd used `nv` instead of `new-env` in the call to `interp`?
3. Are there any other errors in the interpreter based on copying what we had before?
4. We seem to extend the environment but never remove anything from it. Is that okay?

This concludes our first interesting "programming language". We have already been forced to deal with some fairly subtle questions of scope, and with how to interpret them. Things will only get more interesting from here!

Functions in the Language

Now that we have arithmetic and conditionals, let's proceed to creating a full-fledged programming language by adding functions.

There are many ways to think about adding functions to the language. Many languages, for instance, have top-level functions; e.g.:

```
fun f(x):
  x + x
```

Indeed, some languages (such as C) *only* have top-level functions.

Functions in the Language

Most modern languages, however, have the ability to write functions outside the top-level: e.g.,

```
fun f(x):
  fun sq(y):
    y * y
  sq(x) + sq(x)
```

and even

- to *return* those functions, and even
- to allow them to be written *anonymously*.

Since just about every modern language supports it, we'll think of this as a component of SMoL.

Indeed, with such a facility, we don't really need a named function construct per se: we could instead have written

```
fun f(x):
  sq = lam(y): y * y
  sq(x) + sq(x)
```

And in turn we can replace `f` with a name-binding and `lam`, too.

Extending the Representation

Therefore, let's think about what it takes to evaluate functions-as-values to SMoL.

- We don't need functions to inherently have a name, because naming can be done by `let1`.
- We'll assume, for simplicity, that all functions take only one argument; extending this to multiple arguments is left as an exercise.
- **Exercise**: What issues might we have to deal with when we extend functions from having one argument only to having multiple arguments?

First, we need to extend our abstract syntax.

Do Now: How many new constructs do we need to add to the abstract syntax?

Extending the Representation

When we added `let1`, you may recall that it didn't suffice to add one construct; we needed two: one for variable *binding* and one for variable *use*.

- You'll often see this pattern when adding values to the language.
- For any new kind of value, you can expect to see one or more ways to *make* it and one or more ways to *use* it.
- (Even arithmetic: numeric constants were a way to make them, arithmetic operations consumed them—but also made them.)

Likewise with functions, we need a way to represent both

```
lam(x): x * x
```

for defining new functions, and

```
sq(3)
```

to use them.

Extending the Representation

Terminology: In more advanced texts, you will sometimes see the (formally correct, but perhaps slightly confusing) terms *introduction* and *elimination*:

- introduction brings the new concept in,
- elimination uses them.

Thus,

- the `lam` introduces new functions, and
- an application eliminates them.

We therefore add

```
[lamE (var: Symbol) (body: Exp)]
[appE (fun: Exp) (arg: Exp)]
```

to our AST.

Extending the Representation

Let's assume we've already extended our parser, so that programs like the following are legal:

```
{let1 {f {lam x {+ x x}}}
      {f 3}}

{let1 {x 3}
      {let1 {f {lam y {+ x y}}}
            {f 3}}}
```

These parse, respectively, into

```
(let1E 'f (lamE 'x (plusE (varE 'x) (varE 'x)))
       (appE (varE 'f) (numE 3)))

(let1E 'x (numE 3)
       (let1E 'f (lamE 'y (plusE (varE 'x) (varE 'y)))
              (appE (varE 'f) (numE 3))))
```

and should both evaluate to 6.

Evaluating Functions

Now let's think about the evaluator, which by now we can think of as turning into a full-blown interpreter. Let's start with the (almost) simplest kind of new program:

```
{lam x {+ x x}}
```

which is represented as

```
(lamE 'x (plusE (varE 'x) (varE 'x)))
```

Do Now: What do we want this program to evaluate to? Think in terms of types!

Evaluating Functions

Remember that `calc` produces numbers. What *number* does the above expression evaluate to? What number do you *expect* it to produce?

- If we really want to stretch our credibility, we could either make up an encoding of it in a number, or use a number in memory.
- But neither of these is what we would *expect*!

Let's look at what some other languages do:

```
> (lambda (x) (+ x x))
#<procedure>
> (number? (lambda (x) (+ x x)))
#f
```

```
>>> lambda x: x + x
<function <lambda> at 0x108fd16a8>
>>> isinstance(lambda x: x + x, numbers.Number)
False
```

Evaluating Functions

Both Racket and Python agree: the result of creating an anonymous function is a function-kind of value, not a number. What this says is that we have to broaden the kinds of values that `interp` can produce.

- **Terminology**: A *side-effect* is a change to the system that is visible from outside the body of a function.
  - Typical side-effects are modifications to variables that are defined outside the function, communication with a network, changes to files, and so on.
- **Terminology**: A function is *pure* if, for a given input, it always produces the same output, and has no side-effects.
  - In reality, a computation always has *some* side-effects, such as the consumption of energy and production of heat, but we usually overlook these because they are universal.
  - In a few settings, however, they can matter: e.g., if a cryptographic key can be stolen by measuring these side-effects.
- **Terminology**: Traditionally, some languages have used the terms *procedure* and *function* for similar but not identical concepts.
  - Both are function-like entities that encapsulate a body of code and can be applied (or "called").
  - A procedure is an encapsulation that does not produce a value; therefore, it must have side-effects to be of any use.
  - In contrast, a function always produces a value (and may be expected to not have any side-effects).
  - This terminology has gotten completely scrambled over the years and people now use the terms interchangeably, but if someone seems to be making a distinction between the two, they probably mean something like the above.

Extending Values

What happens when evaluating a function? Both Racket and Python seem to suggest that we return a function. We could have no additional information about the function:

```
(define-type Value
  [numV (the-number: Number)]
  [boolV (the-boolean: Boolean)]
  [funV])
```

(That syntax means `funV` is a constructor of no parameters.

- It conveys no information at all other than the fact that it's a `funV`;
- because we can't mix types, it says, in particular, that a value is not numeric or a Boolean—and nothing more.)

But now think about a program like this (assuming `x` is bound):

```
{{if0 x
      {lam x {+ x 1}}
      {lam x {- x 2}}}
 5}
```

In both cases we're going to get a `funV` value with no additional information, so when we try to perform the application, we…can't.

Extending Values

Instead, it's clear that the function value needs to tell us about the function.

- We need to know the body, because that's what we need to evaluate;
- but the body can (and very likely does) reference the name of the formal parameter, so we need that too.

Therefore, what we really need is

```
(define-type Value
  [numV (the-number: Number)]
  [boolV (the-boolean: Boolean)]
  [funV (var: Symbol) (body: Exp)])
```

At this point, it seems like we've gone to a lot of trouble for nothing. We take numeric and Boolean values and simply re-wrap them in new constructors, and now we're doing the same thing for functions. A certain Shakespeareian play's title comes to mind.

Patience.

Extending Values

With what we have, we can already have a functioning interpreter. The `lam` case is obviously very simple:

```
[(lamE v b) (funV v b)]
```

The application case is a bit more detailed. We need to:

1. Evaluate the function position, to figure out what kind of value it is.
2. Evaluate the argument position, since we've agreed that's what happens in SMoL.
3. Check that the function position really does evaluate to a function. If it does not, raise an error.
4. Evaluate the body of the function. But because the body can refer to the formal parameter…
5. …first make sure the formal is bound to the actual value of the argument.

Extending Values

Codifying this, in stages:

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              ...)]
```

Extending Values

Codifying this, in stages:

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b) ...]
                [else (error 'app "didn't get a function")]))]
```

Extending Values

Codifying this, in stages:

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b)
                 (interp b ...)]
                [else (error 'app "didn't get a function")]))]
```

Extending Values

Codifying this, in stages:

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b)
                 (interp b (extend nv v av))]
                [else (error 'app "didn't get a function")]))]
```

Stepping Back

Putting it all together, we get the following interpreter:

```
(interp : (Exp Env -> Value))

(define (interp e nv)
  (type-case Exp e
    [(numE n) (numV n)]
    [(varE s) (lookup s nv)]
    [(plusE l r) (add (interp l nv) (interp r nv))]
    [(lamE v b) (funV v b)]
    [(appE f a) (let ([fv (interp f nv)]
                      [av (interp a nv)])
                  (type-case Value fv
                    [(funV v b)
                     (interp b (extend nv v av))]
                    [else (error 'app "didn't get a function")]))]
    [(let1E var val body)
     (let ([new-env (extend nv
                            var
                            (interp val nv))])
       (interp body new-env))]))
```

Stepping Back

Exercise: We wrote down a particular ordering above, which we put into practice in the code.

- But is that the same ordering that actual languages use?
- In particular, are non-function errors reported after or before evaluating the argument?
- Experiment and find out!

Stepping Back

Since we've taken several steps to get here, it's easy to lose sight of what we've just done.

- In just 20 lines of code (with a few helper functions), we have described the implementation of a *full programming language*.
- Not only that, a language that can express *all computations*.

When Turing Award winner Alan Kay first saw the equivalent program, he says,

> *Yes, that was the big revelation to me when I was in graduate school—when I finally understood that the half page of code on the bottom of page 13 of the Lisp 1.5 manual was Lisp in itself. These were "Maxwell's Equations of Software!" This is the whole world of programming in a few lines that I can put my hand over.*

> *I realized that anytime I want to know what I'm doing, I can just write down the kernel of this thing in a half page and it's not going to lose any power. In fact, it's going to gain power by being able to reenter itself much more readily than most systems done the other way can possibly do.*

Stepping Back

If you want to see the original, here's that manual (by McCarthy, Abrahams, Edwards, Hart, Levin). Here it is, copied:

evalquote is defined by using two main functions, called eval and apply. apply handles a function and its arguments, while eval handles forms. Each of these functions also has another argument that is used as an association list for storing the values of bound variables and function names.

```
evalquote[fn;x] = apply[fn;x;NIL]
```

where

```
apply[fn;x;a] =
    [atom[fn] → [eq[fn;CAR] → caar[x];
                 eq[fn;CDR] → cdar[x];
                 eq[fn;CONS] → cons[car[x];cadr[x]];
                 eq[fn;ATOM] → atom[car[x]];
                 eq[fn;EQ] → eq[car[x];cadr[x]];
                 T → apply[eval[fn;a];x;a]];
     eq[car[fn];LAMBDA] → eval[caddr[fn];pairlis[cadr[fn];x;a]];
     eq[car[fn];LABEL] → apply[caddr[fn];x;cons[cons[cadr[fn];
                                                caddr[fn]];a]]]
eval[e;a] = [atom[e] → cdr[assoc[e;a]];
     atom[car[e]] →
          [eq[car[e];QUOTE] → cadr[e];
           eq[car[e];COND] → evcon[cdr[e];a];
           T → apply[car[e];evlis[cdr[e];a];a]];
     T → apply[car[e];evlis[cdr[e];a];a]]
```

pairlis and assoc have been previously defined.

```
evcon[c;a] = [eval[caar[c];a] → eval[cadar[c];a];
              T → evcon[cdr[c];a]]
and
evlis[m;a] = [null[m] → NIL;
              T → cons[eval[car[m];a];evlis[cdr[m];a]]]
```

Stepping Back

Alright, so we now have a working interpreter for a full-fledged language.

But before we can feel sure of that, we should try a few more examples to confirm that we're happy with what we have.

Extending Tests

Well, actually, we shouldn't be too happy. Consider the following examples:

```
(let1E 'x (numE 1)
       (let1E 'f (lamE 'y (varE 'x))
              (let1E 'x (numE 2)
                     (appE (varE 'f) (numE 10)))))
```

What do we expect it to produce? If in doubt, we can write the same thing as a Racket program:

```
(let ([x 1])
  (let ([f (lambda (y) x)])
    (let ([x 2])
      (f 10))))
```

- What we see is that in *Racket*, the inner binding of `x` does *not* override the outer one, the one that was present at the time the function bound to `f` was defined.
- Therefore, this produces `1` in Racket.

Extending Tests

We should want this! Otherwise, consider this program:

```
(let1E 'f (lamE 'y (varE 'x))
       (let1E 'x (numE 1)
              (appE (varE 'f) (numE 10))))
```

This corresponds to

```
(let ([f (lambda (y) x)])
  (let ([x 1])
    (f 3)))
```

which has an unbound identifier (`x`) error.

But our interpreter produces `1` instead of halting with an error, which leads us right back to **dynamic scope**!

Return to Static Scope

So how do we fix this?

- The examples above actually give us a clue, but there is another source of inspiration as well.
- Do you remember that we started with substitution?
- We'll walk through these examples in Racket, so that you can run each of them directly and check that they produce the same answer.

Return to Static Scope

Consider again this program:

```
(let ([x 1])
  (let ([f (lambda (y) x)])
    (let ([x 2])
      (f 10))))
```

Substituting `1` for `x` produces:

```
(let ([f (lambda (y) 1)])
  (let ([x 2])
    (f 10)))
```

Substituting `f` produces:

```
(let ([x 2])
  ((lambda (y) 1) 10))
```

Finally, substituting `x` with 2 produces (note that there are no xs left in the program!):

```
((lambda (y) 1) 10)
```

Return to Static Scope

When you see it this way, it's clear *why* the later binding of `x` should have no impact:

- it's a different `x`, and
- the earlier `x` has effectively already been substituted.

Since we have agreed that substitution is how we want our programs to work, our job now is to make sure that the environment actually implements that *correctly*.

The way to do it is to recognize that

- the environment represents the substitutions waiting to happen,
- and just *remembers* them.

That is, our representation of a function needs to also keep track of the environment at the moment of function creation:

```
(define-type Value
  [numV (the-number: Number)]
  [boolV (the-boolean: Boolean)]
  [funV (var: Symbol) (body: Exp) (nv: Env)])
```

Return to Static Scope

```
(define-type Value
  [numV (the-number: Number)]
  [boolV (the-boolean: Boolean)]
  [funV (var: Symbol) (body: Exp) (nv: Env)])
```

This new and richer kind of `funV` value has a special name: it's called a *closure*. That's because the expression is "closed" over the environment in which it was defined.

- **Terminology**: A *closed* term is one that has no unbound variables. The body of a function may have unbound variables—like `x` above—but the closure makes sure that they aren't *really* unbound, because they can get their values from the stored environment.
- **Quote**: "Save the environment! Create a closure today!"
  —Cormac Flanagan
- **Quote**: "Lambdas are relegated to relative obscurity until Java makes them popular by not having them."
  —James Iry, A Brief, Incomplete, and Mostly Wrong History of Programming Languages

Return to Static Scope

That means, when we create a closure, we have to record the environment at the time of its creation:

```
[(lamE v b) (funV v b nv)]
```

Finally, when we use a function (represented by a closure), we have to make sure we use the *stored* environment, not the one present at the point of calling the function, which is the *dynamic* one:

```
[(appE f a) (let ([fv (interp f nv)]
                  [av (interp a nv)])
              (type-case Value fv
                [(funV v b nv)
                 (interp b (extend nv v av))]
                [else (error 'app "didn't get a function")]))]
```

Just to be clear: in the code above,

- the `nv` in the `funV` case *intentionally shadows* the `nv` bound at the top of the interpreter. Thus,
- the call to `extend` extends the environment *from the closure*,
- rather than the one present at the point of the call.

Return to Static Scope

Exercise: Notice that the function and argument expressions (`f` and `a`, respectively) are evaluated in the environment given to the interpreter, not the one inside the closure.

- Is this correct?
- Or should they be using the closure's environment?

You can do two things: argue from first principles or argue with examples.

- In the latter case, you would modify the interpreter to make the other choice.
- You would then use a sample input that produces different answers depending on which environment is used,
- indicate which one is correct (showing what the equivalent Racket program would produce can be a good argument),
- and use that to justify the chosen environment.

Hint: One of these you will need to argue from first principles, the other you should be able to argue using a program.

A Subtle Test

In the examples above, we always use the closure in the scope in which it was defined. However, our language is actually more powerful than that: we can

- return a closure and
- use it outside the scope in which it was defined.

Here's a sample Racket program:

```
((let ([x 3])
   (lambda (y) (+ x y)))
 4)
```

Do Now: Take a moment to read it carefully. What should it produce?

A Subtle Test

First we bind the `x`, then we evaluate the `lambda`. This creates a closure that remembers the binding to `x`.

This closure is the value returned by this expression:

```
(let ([x 3])
  (lambda (y) (+ x y)))
```

This value is now applied to 4.

- It's legal to do this, because the value returned is a function.
- When we apply it to 4, that evaluates the sum of 4 and 3, producing 7.

Sure enough, translating this and sending it to our interpreter produces 7:

```
(test (interp (appE (let1E 'x (numE 3)
                           (lamE 'y (plusE (varE 'x) (varE 'y))))
                    (numE 4))
              mt-env)
      (numV 7))
```

A Subtle Test

Exercise: Here's another test to try out, written as a Racket program:

```
((let ([y 3])
   (lambda (y) (+ y 1)))
 5)
```

What does it produce in Racket? Translate it and try it in your interpreter.