# Chapter 2

# An imperative core

Contents

Your prior programming experience may be with a pure *imperative language*, such as Ada 83, Algol 60, C, Cobol, Fortran, Modula-2, or Pascal. A huge number of real-world programs are written in such "conventional" languages. Or you may have experience with a language that mixes imperative and object-oriented features, such as Ada 95, C#, C++, Eiffel, Java, Modula-3, or Objective C. These hybrid languages support an object-oriented style, but they are often used imperatively.

Without seeing alternatives, you may think of imperative programming as "just programming," but imperative programming is a well-developed style with identifiable characteristics:

- Programs process data one word at a time. A computation consists of many individual movements and computations of small items of data. Arrays are the most common high-level data structure, and they are processed one element at a time, not as a whole.
- Programs use many side effects. The computation proceeds by continually changing the "mutable state" of the machine—the values contained in the various machine words—by assignment.
- Most control flow happens in loops. Iteration, whether written using `goto` or using looping constructs such as `for` and `while`, is the primary method of control. Looping constructs fit well with element-by-element processing of arrays. Procedures, methods, and recursion take a back seat.
- Language structures for both data and control are fairly close to the underlying machine architecture. The `goto` is an unconditional jump; `if` and `while` give structure to conditional jumps. Arrays are contiguous blocks of memory, pointers are addresses, assignment is data movement, and so on. Debuggers for imperative languages are often simple adaptations of machine-level debuggers.

The imperative style can be embodied in a language. In this chapter, that language is *Impcore*. Impcore is comprised of standard imperative constructs—assignments, loops, conditionals, and procedures—which are ubiquitous in programming languages. They are found not only in "imperative" or "procedural" languages but also in many "functional" and "object-oriented" languages.

More important than the constructs, which should be familiar, are the intellectual tools we use to describe, understand, and implement the languages in this book.

- *Concrete syntax* is the form in which people write programs. Because we don't want to get into the *parsing* techniques used to recognize concrete syntax, we use a simple, parenthesized-prefix syntax, which is easy to recognize.
- *Abstract syntax* expresses the essential structure of programs.
- *Operational semantics* explains the meaning of the abstract syntax by explaining how it should be executed.
- I provide an interactive *interpreter*, which parses concrete syntax, converts it to abstract syntax, and executes it according to the operational semantics. Experimenting with interpreters helps build insights and skills that can't be obtained by mere discussion. The first few interpreters in this book are written in C; the rest in Standard ML.
- *Environments* determine the meanings of identifiers, both in the operational semantics and in the interpreter.

The first four of these tools—concrete syntax, abstract syntax, operational semantics, and an interpreter—are intimately connected, and the center of the connection is the abstract syntax. Abstract syntax is where we begin to learn the story about what a language is. We want all the languages in this book to be simple, and simplicity starts with simple syntax. To make syntax simple, one of our best tricks is to use as few *syntactic categories* as possible.

A syntactic category is a kind of a phrase in a language. Operationally, two phrases are in the same syntactic category if and only if they can be exchanged for each other without changing syntactic correctness. (That is, if a program parses, and you replace one phrase with a different phrase that is in the same syntactic category, the new program will also parse.) For example, in the C programming language, here are several phrases in the syntactic category of *expressions*:

```
x + 1          p->next          f(17)
```

Any of these phrases can appear in any context where an expression is expected, and the resulting C program will be grammatical.[1]

Most programming languages have at least four syntactic categories:

- A *declaration* introduces a new named thing without providing complete information about that thing. For example, the declaration of a variable might give its name and possibly its type, but not its initial value. The declaration of a function might give the types of its arguments and result, but not its body.
- A *definition* introduces a new named thing and provides all the information needed to program with that thing. For example, the definition of a variable might give its name and its initial value. The definition of a function might give its name, the names of its parameters, and its body.
- A *statement* is executed for *side effect*. For example, it might print output, it might change the value of a variable, or it might change some value in memory.
- An *expression* is evaluated to produce a value. In some languages, an expression can produce a value *and* have a side effect.

What does it mean, then, to have as few syntactic categories as possible? In this book,

- When possible, I avoid declarations: where a declaration could appear, I almost always require a definition. In late chapters, I make a couple of exceptions, allowing declarations of local variables or local procedures.
- I use a single syntactic category that encompasses both side-effecting phrases and value-producing phrases. Because the value-producing aspect is more important, I call that category *expressions*. This convention might surprise you: for example, we're not used to calling a while loop an expression! But it's easy to make a while loop an expression, and there's precedent; for example, in the programming language Icon (Griswold and Griswold 1996), *all* looping constructs are expressions.

  The real benefit of having only expressions, instead of both statements and expressions, is that there are fewer design decisions to make, and our languages end up being simpler. Having to choose one category or the other leads to gratuitous differences;

[1] "Grammatical" is not the same as "correct." For example, replacing `x + 1` with `p->next` will break a program if `x` is declared but `p` is not, or if `x + 1` and `p->next` have incompatible types.

for example, in Pascal an assignment is a statement, but in C an assignment is an expression. And if your language has two related syntactic categories, a construct can wind up in *both* of them: for example, C has both a conditional *statement*, written with `if` and `else`, and also a conditional *expression*, written with `?` and `:`. To program effectively in C, you have to know both. By having only expressions, we avoid this kind of duplication.

## 2.1 The Impcore language

An Impcore program is a sequence of *definitions*. Using the interpreter in this chapter, you can enter definitions interactively or load them from a file. There are two main kinds of definition: a definition of a variable, such as `(val n 5)`, and a definition of a function, such as `(define double (x) (+ x x))`. Because it's wonderfully convenient to be able to type in an expression, have it evaluated, and see the result, we also count an expression, such as `(+ 2 2)`, as a "definition"—in this case, a definition of the global variable `it`. The global variable `it` thereby "remembers the last expression typed in." The variable `it` has played this special role in interactive interpreters for over twenty years.

The interpreter "remembers" all definitions; if you are used to compiling programs at a command line, "remembering a definition" corresponds roughly to "compiling a program." And "evaluating an expression" (the special syntax for the definition of `it`) corresponds roughly to "running a program."

The rest of this section presents Impcore's syntax, Impcore's semantics, and examples.

### 2.1.1 Concrete syntax

The concrete syntax of Impcore is like that of Lisp. We define it using Extended Backus-Naur Form (EBNF). In this notation, text in `typewriter font` should be typed literally. Names in *italic font* stand for syntactic categories; below we write the name of a category on the left, a special symbol pronounced "produces" (::=), and the ways of producing phrases in the category. Within phrases, sequences of zero or more items are written in braces, as {item}. Alternatives are separated by vertical bars, as (one thing | another). Appendix K explains the notation more fully.

*def* ::= `(val` *variable-name exp*`)`
| *exp*
| `(define` *function-name* `(`*formals*`)` *exp*`)`
| `(use` *file-name*`)`

*exp* ::= *value*
| *variable-name*
| `(set` *variable-name exp*`)`
| `(if` *exp exp exp*`)`
| `(while` *exp exp*`)`
| `(begin` {*exp*}`)`
| `(`*function* {*exp*}`)`

*formals* ::= {*variable-name*}

*value* ::= *integer*

*function* ::= *function-name*
| *primitive*

*primitive* ::= `+` | `-` | `*` | `/` | `=` | `<` | `>` | `print`

*integer* ::= sequence of digits, possibly prefixed with a plus or minus sign

**-name* ::= sequence of characters not an *integer* and not containing `(`, `)`, `;`, or whitespace

It is not useful to define a *function-name* that is one of the "keywords" `define`, `if`, `while`, `begin`, or `set`. It is not wise to define a *function-name* that is one of the *primitives*. Aside from these restrictions, names can use any characters except parentheses, semicolon, whitespace, and nulls. A comment is introduced by a semicolon, and it continues to the end of the line; therefore a semicolon cannot occur within a name. The construct (`use` *file-name*) is not itself a definition, but it causes the interpreter to read the definitions in the named file as if they had been typed directly to the interpreter. The construct `(val x e)` defines a new global variable `x` and initializes it to the value of the expression `e`. Global variables must be defined before they are used or assigned to.

Expressions are fully parenthesized. We use *prefix* syntax, in which each operator precedes its arguments. Thus, translating the C assignment `i = 2*j + i - k/3` produces the Impcore expression `(set i (- (+ (* 2 j) i) (/ k 3)))`. The Impcore syntax is trivial to parse; C syntax is anything but. Readers accustomed to *infix* syntax, in which binary operators appear between their arguments, may find prefix syntax unattractive, especially in complex expressions. Luckily, expressions even this complex occur rarely.

### 2.1.2 Meaning

I present the meanings of Impcore programs using informal English, which is intuitive but not precise. Section 2.4 presents a precise, formal semantics.

In Impcore, all values are integers; as in C, `if` and `while` use their conditions by interpreting zero as false and nonzero as true.

`(if` $\mathtt{e}_1$ $\mathtt{e}_2$ $\mathtt{e}_3$`)` — Evaluate $\mathtt{e}_1$; if it is nonzero, evaluate $\mathtt{e}_2$ and return the result, otherwise evaluate $\mathtt{e}_3$ and return the result.

`(while` $\mathtt{e}_1$ $\mathtt{e}_2$`)` — Evaluate $\mathtt{e}_1$; if it is zero, return zero; otherwise, evaluate $\mathtt{e}_2$ and then re-evaluate $\mathtt{e}_1$; continue until $\mathtt{e}_1$ evaluates to zero. (A `while` expression always returns zero, but that doesn't matter since it is typically evaluated for its side effects.)

`(set x e)` — Evaluate `e`, assign its value to the variable `x`, and return its value. The variable `x` must be either a function parameter or a global variable defined with `val`.

`(begin` $\mathtt{e}_1$ ... $\mathtt{e}_n$`)` — Evaluate $\mathtt{e}_1, \ldots, \mathtt{e}_n$, in that order, and return the value of $\mathtt{e}_n$.

`(f` $\mathtt{e}_1$ ... $\mathtt{e}_n$`)` — Evaluate $\mathtt{e}_1, \ldots, \mathtt{e}_n$, in that order, calling the results $v_1, \ldots, v_n$. Apply function `f` to $v_1, \ldots, v_n$ and return the result. Function `f` may be *primitive* or user-defined; if `f` is user-defined, find its definition, let the names in the *formals* stand for $v_1, \ldots, v_n$, and return the result of evaluating `f`'s body.

Each *primitive* takes two arguments, except `print`, which takes one. The arithmetic operators `+`, `-`, `*`, and `/` do the obvious integer arithmetic. The comparison operators `<`, `>`, and `=` perform the indicated comparison and return either zero (for false) or one (for true). `print` prints and returns its argument.

Like C, Impcore has global variables and formal parameters. When a reference to a variable occurs in an expression at the top level (as opposed to in a function definition), it is

necessarily global. If it occurs within a function definition, then if the function has a formal parameter of that name, the variable is that parameter, otherwise it is a global variable. Like Awk, Impcore has no local variables *per se*, only formal parameters. (To provide local variables is the object of Exercise 26.)

### 2.1.3 Examples

As indicated above, you can type in function definitions and expressions interactively. The interpreter stores each function definition; it evaluates each expression, prints its value, and stores the value in the global variable `it`, which need not be defined in advance. Our first examples involve no function definitions. The arrow "`-> `" is the interpreter's prompt; an expression following a prompt is the user's input; and everything else is the interpreter's response:[2]

10a ⟨*transcript* 10a⟩≡ 10b▷

```
-> 3
3
-> (+ 4 7)
11
-> it
11
-> (val x 4)
4
-> (+ x x)
8
-> (print x)
4
4
```

Notice that `(print x)` first prints the value of `x`, then returns the value; the interpreter always prints the value of an expression, so in this case 4 is printed twice.

10b ⟨*transcript* 10a⟩+≡ ◁10a 10c▷

```
-> (val y 5)
5
-> (begin (print x) (print y) (* x y))
4
5
20
-> (if (> y 0) 5 10)
5
```

The body of a `while` expression (that is, its second argument) is often a `begin` expression.

10c ⟨*transcript* 10a⟩+≡ ◁10b 11a▷

```
-> (while (> y 0)
     (begin
       (set x (+ x x))
       (set y (- y 1))))
0
-> x
128
```

[2]Most of the examples in this book are written using the Noweb system for *literate programming*, which helps us extract the examples from the text and make sure they are consistent with the software. Marginal labels such as 10a identify chunks of examples or code, and pointers such as "10b▷" point to subsequent chunks. A fuller explanation of Noweb appears in Section 2.5 on page 23.

The interpreter allows expressions to spread over more than one line; if the interpreter sees an incomplete expression, it prompts with three spaces and waits for more input.[3]

Calls to user-defined functions work very much as in C: the arguments to the function are evaluated first, then the function's body is evaluated with its *formal parameters* (the variables occurring in the function's *formals*) "bound" to the *actual parameters* (the evaluated arguments of the application). After reading the definition of a function, the interpreter echoes its name.

11a ⟨*transcript* 10a⟩+≡ ◁10c 11b▷

```
-> (define add1 (x) (+ x 1))
add1
-> (add1 4)
5
-> (define double (x) (+ x x))
double
-> (double 4)
8
```

Just as in C, when the variable "x" occurs as a function's formal parameter, it is distinct from the "x" used earlier as a global variable. A reference to x *within* the function refers to the formal parameter; a reference outside the function refers to the global variable. Also as in C, Impcore passes all parameters by value; modifications to formal parameters do not affect the calling function. No assignment to a formal parameter can ever change the value of a global variable.

11b ⟨*transcript* 10a⟩+≡ ◁11a 12a▷

```
-> x
128
-> (define addx (x y) (set x (+ x y)))
addx
-> (addx x 1)
129
-> x
128
```

Most interesting functions require iteration or recursion. Here is an iterative version, in C, of Euclid's function to compute the greatest common denominator of two numbers:

11c ⟨*gcd.c* 11c⟩≡

```
int
gcd(int m, int n) {
    int r;

    while ((r = m%n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
```

[3] Prompts are defined in chunk 595a.

and here it is in Impcore:[4]

12a ⟨*transcript* 10a⟩+≡ ◁11b 12c▷

```
-> (define not (b) (if b 0 1))
-> (define != (x y) (not (= x y)))
-> (define mod (m n) (- m (* n (/ m n))))
-> (val r 0)
0
-> (define gcd (m n)
     (begin
       (while (!= (set r (mod m n)) 0)
         (begin
           (set m n)
           (set n r)))
       n))
-> (gcd 6 15)
3
```

A recursive version in C is:

12b ⟨*gcd-recur.c* 12b⟩≡

```
int
gcd(int m, int n) {
    if (n == 0)
        return m;
    else
        return gcd(n, m % n);
}
```

which would be rendered in Impcore as:

12c ⟨*transcript* 10a⟩+≡ ◁12a 15▷

```
-> (define gcd (m n)
     (if (= n 0)
       m
       (gcd n (mod m n))))
```

### 2.1.4 The Initial Basis

In language design, programmers and implementors have competing desires. Programmers want a rich set of data types and operations. Implementors want to keep primitive functionality as small and simple as possible. (So do semanticists!) Designers have found two strategies that help satisfy these competing desires: translation into a *core language* and definition of an *initial basis*.

Using a core language involves stratifying the language into two layers. The core language contains a relatively small set of constructs, which are implemented or defined directly. The full language contains additional constructs, which are defined (and sometimes implemented) by translation into the core language. For example, it would be possible to add a `for` expression to Impcore, and to define it by translation into `begin` and `while`. Constructs that are in the full language but not in the core are sometimes called *syntactic sugar*. I don't use syntactic sugar in this chapter, but in Chapter 3 I use it to implement a couple of constructs of μScheme.

[4] From here on, we omit the names that interpreters print after function definitions.

The initial basis uses the same idea, except it works on predefined functions, not on language constructs. Some functions are "primitive" functions defined directly by C code in the interpreter. Other functions are also built into the interpreter, but are defined in terms of existing primitives and language constructs. All these functions are available to the programmer, and together they form the initial basis. Having a small set of primitives and a large initial basis makes things easy for everyone. Implementors can add to the initial basis just by writing ordinary code in the language, which is easier than defining new primitives. Figure 2.1 shows our additions to the initial basis of Impcore.

We write Boolean connectives using `if` expressions.

13a ⟨*additions to the Impcore initial basis* 13a⟩≡ 13b▷

```
(define and (b c) (if b c b))
(define or  (b c) (if b b c))
(define not (b)   (if b 0 1))
```

Unlike the similar constructs built into the syntax of many languages, these versions of `and` and `or` always evaluate both of their arguments.

We add new arithmetic comparisons.

13b ⟨*additions to the Impcore initial basis* 13a⟩+≡ ◁13a 13c▷

```
(define <= (x y) (not (> x y)))
(define >= (x y) (not (< x y)))
(define != (x y) (not (= x y)))
```

Finally, we define modulus in terms of division.

13c ⟨*additions to the Impcore initial basis* 13a⟩+≡ ◁13b

```
(define mod (m n) (- m (* n (/ m n))))
```

The C code to install the initial basis is shown in chunk ⟨*install the initial basis in* `functions` 46a⟩, which is continued in chunk 46c.

Figure 2.1: Additions to Impcore's initial basis

Before going on to the next sections, work some of the problems in the "Learning about the language" section of the exercises, which starts on page 54.

## 2.2 Abstract syntax

Throughout this book, we represent programs as *abstract-syntax trees* (ASTs), which are the best and simplest internal representation of source code. An abstract-syntax tree separates the internal representation of programs from the representation used to express the programs in written form. It reflects the structure of the original source code, without extraneous details. Abstract syntax is not just a great technique for implementing programming languages; it is the best way think about syntax.

As an example, a phrase in Impcore can be represented by an abstract-syntax tree that uses leaf nodes to represent basic elements of the language, such as variables and numbers, and uses interior nodes to represent more complex constructs, such as `if` and `set`.

We specify abstract syntax by giving the name of each kind of node, and by saying what kind of children each node has. For example, the abstract representation of *def* (definition) from Section 2.1.1 is

14a ⟨*simplified example of abstract syntax for Impcore* 14a⟩≡ 14b▷

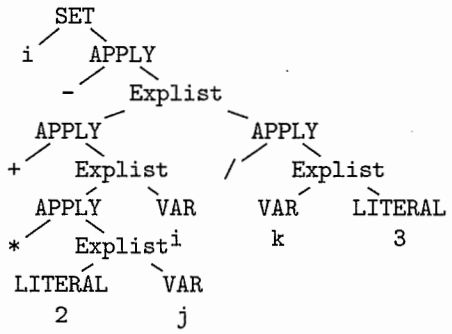
```
Def = VAL    (Name, Exp)
    | EXP    (Exp)
    | DEFINE (Name, Namelist, Exp)
    | USE    (Name)
```

Similarly, the abstract representation of *exp* is:

14b ⟨*simplified example of abstract syntax for Impcore* 14a⟩+≡ ◁14a

```
Exp = LITERAL (Value)
    | VAR     (Name)
    | SET     (Name, Exp)
    | IF      (Exp, Exp, Exp)
    | WHILE   (Exp, Exp)
    | BEGIN   (Explist)
    | APPLY   (Name, Explist)
```

As an example, the expression `(set i (- (+ (* 2 j) i) (/ k 3)))` has the following abstract-syntax tree.




Abstract syntax does not say anything about how a program is written in a source file; it specifies only the *structure* of a language. For example, all that matters about an `if` node is that it has three children: a conditional expression, an expression to execute when the condition is true, and an expression to execute when the condition is false. It doesn't matter how the expression is written using the concrete syntax; once we get to abstract syntax, the phrases "`(if` $e_1$ $e_2$ $e_3$`)`," "`if` $e_1$ `then` $e_2$ `else` $e_3$," and even "$e_1$ `?` $e_2$ `:` $e_3$" are indistinguishable.

The information in an abstract-syntax tree is also present in the (concrete) source, but the AST is much easier to analyze, manipulate, and interpret. ASTs focus our attention on structure and semantics. Concrete syntax becomes a separate concern; one could easily define a version of Impcore with C-like concrete syntax, but with identical abstract syntax.

The process of creating abstract syntax from concrete input is called *parsing*, and it is also a good way to find malformed phrases such as `(if x 0)` or `(val y)`. There is a large body of literature devoted to parsing; Appel (1998) and Aho, Sethi, and Ullman (1986) provide textbook treatments.

Our interpreters convert concrete syntax to ASTs, then manipulate the ASTs. When memory was scarce, some interpreters and compilers avoided representing ASTs explicitly; instead, they translated concrete syntax directly into a bytecode or into an intermediate form. Now that machines have many megabytes of memory, the simplicity of explicit ASTs is a good tradeoff, especially when using compiler-construction tools that automate the construction of ASTs.

## 2.3 Environments

An expression like `(* x 3)` cannot be evaluated by itself; we need to know what `x` is. Similarly, the effect of an expression such as `(set x 1)` depends on whether `x` is a formal parameter, a global variable, or something else entirely. In programming-language theory, the thing that defines the meanings of names is called an *environment*; it is usually considered to be a mapping from names to meanings. In implementations, environments are often realized as hash tables or search trees, and they are sometimes called *symbol tables*.

We write $\rho(x)$ to mean whatever is associated with the name $x$ in the environment $\rho$. We write $\rho\{x \mapsto v\}$ to mean the environment $\rho$ extended by a binding of the name $x$ to $v$, whatever $v$ is:

$$\rho\{x \mapsto v\}(y) = \begin{cases} v, & \text{when } y = x \\ \rho(y), & \text{when } y \neq x \end{cases}$$

We write $\{\}$ to mean the empty environment, i.e., the environment that does not bind any names.

Knowing what can be in an environment tells you what kinds of things names can stand for, which is a good first step in understanding a new programming language. Impcore uses three environments: the environment $\xi$ ("ksee") contains the values of global variables, the environment $\rho$ ("row", rhymes with "dough") contains the values of formal parameters, and the environment $\phi$ ("fee") contains the definitions of functions. The following transcript cleverly uses `x` in all three environments.

15 ⟨*transcript* 10a⟩+≡ ◁12c

```
-> (val x 2)
2
-> (define x (y) (+ x y))
-> (define z (x) (x x))
-> (z 4)
6
```

The first item defines a global variable `x` with value 2. The second item defines a function `x` that adds its argument to the global variable `x`. The third item defines `z`, which passes its formal parameter `x` to the function `x`. To understand this example, you must understand that Impcore has three distinct environments.

## 2.4 Operational semantics

Section 2.1.2 describes Impcore informally. Section 2.5 presents C code for an interpreter that implements Impcore. The informal description is short and readable but hard to make precise. The interpreter is precise, but it is much longer and harder to understand than an informal description. It also embodies many other decisions, including decisions about the *representations* of names, environments and abstract syntax. If we just want to know what an Impcore program means, these decisions are irrelevant and distracting—they are part of an implementation, not part of the language.

This section defines the meaning of Impcore using formal operational semantics, a description technique that is both short and precise. Operational descriptions specify the meanings of programs while ignoring details required for a running implementation. Although the notation can be intimidating at first, with practice you can read an operational description almost as easily as a description written in informal English.

An operational semantics is written by defining an abstract machine and rules for its execution. A good abstract machine is designed to show how to run programs written in the language being defined. We define the machine's states, including its start state and its acceptable final states, and we present rules for making transitions from one state to another. By applying these rules repeatedly, the machine can go from a start state like "I just turned on and have this program to execute" to an accepting state like "the answer is 42."

Sometimes a machine reaches a state from which it cannot make forward progress; for example a machine evaluating `(/ 1 0)` might not be able to make a transition. When a machine reaches such a state, we say it "gets stuck" or "goes wrong." (An implementation might indicate a run-time error.)

The state of the Impcore machine has four parts: a *definition* $d$ or *expression* $e$ being evaluated; a value environment $\xi$, which holds the values of global variables; a function-definition environment $\phi$; and a value environment $\rho$, which holds the values of formal parameters. When the machine is evaluating a definition $d$, we write its state as $\langle d, \xi, \phi \rangle$. When it is evaluating an expression $e$, we write its state $\langle e, \xi, \phi, \rho \rangle$. The state $\langle d, \xi, \phi \rangle$ does not include an environment $\rho$, because definitions do not appear inside functions, so while a definition is being evaluated, there are no formal parameters. When the machine is resting between evaluations, it remembers only the values of global variables and the definitions of functions, and we write its state $\langle \xi, \phi \rangle$.

In the initial state, $\xi = \{\}$, because there are no global variables defined, but $\phi = \phi_0$, where $\phi_0$ is preloaded with the definitions of primitive functions as well as the user-defined functions in the initial basis (see Figure 2.1 on page 13). We write a primitive function as PRIMITIVE($\oplus$), where $\oplus$ is the name of an operator like $+$, $=$, or $*$. We write a user-defined function as USER($\langle x_1, \ldots, x_n \rangle, e$), where the $x_i$'s are the formal parameters and $e$ is the body.

In Impcore, the values stored in $\xi$ and $\rho$ are integers.

### 2.4.1 Judgments and rules of inference

The transition rules of the abstract machine are written in the form of *judgments*. There is one kind of judgment for definitions and a different kind of judgment for expressions. We write the judgment $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$ to mean "evaluating expression $e$ produces value $v$." More precisely, the judgment means "in the environments $\xi$, $\phi$, and $\rho$, evaluating $e$ produces a value $v$, and it also produces new environments $\xi'$ and $\rho'$, while leaving $\phi$ unchanged."[5] We always use symbols $e$ and $e_i$ for expressions, $v$ and $v_i$ for values, and $x$ and $x_i$ for names.

Just by looking at the *form* of the judgment $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$, we can learn a few things:

- Evaluating an expression always produces a value, unless of course the machine gets stuck. Even expressions like SET and WHILE, which are typically evaluated only for side effects, must produce values.[6]
- Evaluating an expression might change the value of a global variable (from $\xi$) or a formal parameter (from $\rho$).
- Evaluating an expression never adds or changes a function definition (because $\phi$ is unchanged).

One thing we *can't* learn from the form of the judgment is whether evaluating an expression can introduce a new variable. In fact it can't, but to learn this requires that we study the full semantics and write an inductive proof (Exercise 15).

The heart of the interpreter in Section 2.5 is the function `eval`. Calling $\texttt{eval}(e, \xi, \phi, \rho)$ returns $v$ and has side effects on $\xi$ and $\rho$ such that $\langle e, \xi_{before}, \phi, \rho_{before}\rangle \Downarrow \langle v, \xi_{after}, \phi, \rho_{after}\rangle$. The rules in this section specify a straightforward, recursive implementation of `eval`.

The judgment for definitions takes a simpler form. We write $\langle d, \xi, \phi\rangle \rightarrow \langle \xi', \phi'\rangle$ to mean "evaluating definition $d$ in the environments $\xi$ and $\phi$ yields new environments $\xi'$ and $\phi'$." To help distinguish this judgment from an expression judgment, we use a different arrow.

We can't write just any judgment and expect it to be true. For example, it seems reasonable to claim $\langle \texttt{(+ 1 1)}, \xi, \phi, \rho\rangle \Downarrow \langle 2, \xi, \phi, \rho\rangle$, but unless some joker changes the meaning of `+`, $\langle \texttt{(+ 1 1)}, \xi, \phi, \rho\rangle \Downarrow \langle 4, \xi, \phi, \rho\rangle$ is clearly false. An operational semantics uses *rules of inference* to tell which judgments are valid. Each rule has the form

$$\frac{\text{premises}}{\text{conclusion}} \qquad \text{(NAME OF RULE)}$$

If we can prove each premise, we can use the rule to prove the conclusion.

[5] A judgment is a relation, not a function. In principle, it is possible to have $v_1 \neq v_2$ such that $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle$ and also $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v_2, \xi', \phi, \rho'\rangle$. A language that permits such ambiguity is *nondeterministic*. All the languages in this book are deterministic, but multithreaded languages, like Java or C#, can be nondeterministic. Programs written in such languages can produce different answers on different runs. Languages that do not specify the order in which expressions are evaluated, like C, can also be nondeterministic. Programs written in such languages can produce different answers when translated with different compilers. (See also Exercise 18.)

[6] This property distinguishes expression-oriented languages, like Impcore, ML, and Scheme, from statement-oriented languages like C. All these languages have imperative constructs that are evaluated only for side effects, but only in C do these constructs return no values.

For example, the rule

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}{\langle \text{IF}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle,} \qquad \text{(IFTRUE)}$$

which is part of the semantics of Impcore, says that whenever $\langle e_1, \xi, \phi, \rho\rangle$ evaluates to some nonzero value $v_1$, the expression $\text{IF}(e_1, e_2, e_3)$ evaluates to the result of evaluating $e_2$. Because $e_2$ is evaluated in the environment produced by evaluation of $e_1$, if $e_1$ contains side effects, such as assigning to a variable, the results of those side effects will be visible to $e_2$. (We can tell because the side effects of $e_1$ are captured in environments $\xi'$ and $\rho'$, and these are the environments used to evaluate $e_2$.) The premises don't even mention $e_3$ because if $v_1 \neq 0$, $e_3$ is never evaluated.

The recursive implementation of `eval` actually works *bottom-up* through rules. It is given a state $\langle e, \xi, \phi, \rho\rangle$, and it finds a new state $\langle v, \xi', \phi, \rho'\rangle$ such that $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$. It does so by looking at the *form* of $e$ and examining rules that have $e$'s of that form in their conclusions. For example, to evaluate an IF expression, it first makes a recursive call to itself to find $v_1$, $\xi'$, and $\rho'$ such that $\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle$. Then, if $v_1 \neq 0$, it can make another recursive call to find $v_2$, $\xi''$, and $\rho''$ such that $\langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle$. Having satisfied all the premises of rule IFTRUE, it can then return $v_2$ and the modified environments.

The rest of this section gives all the rules in the semantics of Impcore.

### 2.4.2 Values

Literal values evaluate to themselves without changing any environments.

$$\frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi, \phi, \rho\rangle} \qquad \text{(LITERAL)}$$

This rule has no premises, so at a LITERAL node, the recursive implementation of `eval` terminates.

### 2.4.3 Variables

If a name is bound in the parameter or global environment, then the variable with that name evaluates to the value associated with it by the environment. Otherwise, no rules apply, the machine gets stuck, and the computation does not continue.

Parameters hide global variables. The phrase $\operatorname{dom}\rho$ stands for the domain of $\rho$, i.e., the set of names bound by $\rho$.

$$\frac{x \in \operatorname{dom}\rho}{\langle \text{VAR}(x), \xi, \phi, \rho\rangle \Downarrow \langle \rho(x), \xi, \phi, \rho\rangle} \qquad \text{(FORMALVAR)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad x \in \operatorname{dom}\xi}{\langle \text{VAR}(x), \xi, \phi, \rho\rangle \Downarrow \langle \xi(x), \xi, \phi, \rho\rangle} \qquad \text{(GLOBALVAR)}$$

The premises of these rules involve only membership tests on environments, not other evaluation judgments, so at a VAR node, the recursive implementation of `eval` also terminates.

### 2.4.4 Assignment

Assignment follows the same rules as variable lookup: if the name is known in the parameter environment, we change the binding there; otherwise we look in the global environment.

$$\frac{x \in \operatorname{dom}\rho \qquad \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \textsc{set}(x, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\{x \mapsto v\}\rangle} \qquad \text{(FormalAssign)}$$

$$\frac{x \notin \operatorname{dom}\rho \qquad x \in \operatorname{dom}\xi \qquad \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \textsc{set}(x, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho'\rangle} \qquad \text{(GlobalAssign)}$$

The premises of both SET rules involve evaluation judgments on the right-hand side $e$, so at a SET node, the recursive implementation of `eval` always makes a recursive call.

In Impcore, it is possible to assign only to previously defined variables; given a SET node where $x \notin \operatorname{dom}\rho$ and $x \notin \operatorname{dom}\xi$, the machine gets stuck. In many languages, like Awk for example, assignment to an undefined and undeclared variable creates a new global variable (Aho, Kernighan, and Weinberger 1988). The following rule might be used in an operational semantics for Awk:

$$\frac{x \notin \operatorname{dom}\rho \qquad \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \textsc{set}(x, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho'\rangle} \qquad \text{(GlobalAssign for Awk)}$$

To spot such subtle differences, you have to read inference rules carefully. In Impcore, it is possible to create new global variables only by means of a VAL definition, as shown in rule DefineGlobal in Section 2.4.7.

### 2.4.5 Control Flow

#### Conditional evaluation

The expression $\textsc{if}(e_1, e_2, e_3)$ first evaluates the expression $e_1$ to produce value $v_1$. If $v_1$ is nonzero, the result of the IF expression is the result of evaluating $e_2$, otherwise, it's the result of evaluating $e_3$. It is simplest to have different rules for the two cases.

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle}{\langle \textsc{if}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle} \qquad \text{(IfTrue)}$$

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 = 0 \qquad \langle e_3, \xi', \phi, \rho'\rangle \Downarrow \langle v_3, \xi'', \phi, \rho''\rangle}{\langle \textsc{if}(e_1, e_2, e_3), \xi, \phi, \rho\rangle \Downarrow \langle v_3, \xi'', \phi, \rho''\rangle} \qquad \text{(IfFalse)}$$

Even though there are two rules with IF in the conclusion, only one can apply at one time, because the premises $v_1 \neq 0$ and $v_1 = 0$ are mutually exclusive. This property keeps the evaluation of Impcore programs deterministic.

### Loops

The WHILE$(e_1, e_2)$ loop continues to evaluate $e_2$ while $e_1$ evaluates to a nonzero value.

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2, \xi', \phi, \rho'\rangle \Downarrow \langle v_2, \xi'', \phi, \rho''\rangle \qquad \langle \text{WHILE}(e_1, e_2), \xi'', \phi, \rho''\rangle \Downarrow \langle v_3, \xi''', \phi, \rho'''\rangle}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho\rangle \Downarrow \langle v_3, \xi''', \phi, \rho'''\rangle} \quad \text{(WHILEITERATE)}$$

Because the value $v_2$ is not used, it should be clear that the body $e_2$ is evaluated only for its side effects, i.e., for the new environments $\xi''$ and $\rho''$.

If a WHILE loop terminates, it evaluates to zero.

$$\frac{\langle e_1, \xi, \phi, \rho\rangle \Downarrow \langle v_1, \xi', \phi, \rho'\rangle \qquad v_1 = 0}{\langle \text{WHILE}(e_1, e_2), \xi, \phi, \rho\rangle \Downarrow \langle 0, \xi', \phi, \rho'\rangle} \quad \text{(WHILEEND)}$$

The semantics show that a WHILE expression is executed only for its side effects; it is easy to prove that if a WHILE expression evaluates to a value $v$, then $v = 0$.

### Sequential execution

BEGIN requires two rules: one for the normal case and one for the empty BEGIN. (We permit Impcore programs to include empty BEGIN expressions because this choice simplifies the implementation.)

The empty BEGIN evaluates to zero.

$$\frac{}{\langle \text{BEGIN}(), \xi, \phi, \rho\rangle \Downarrow \langle 0, \xi, \phi, \rho\rangle} \quad \text{(EMPTYBEGIN)}$$

A nonempty BEGIN evaluates its expressions left to right.

$$\frac{\begin{array}{c}\langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \langle e_2, \xi_1, \phi, \rho_1\rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2\rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1}\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle\end{array}}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle} \quad \text{(BEGIN)}$$

As expected, the values $v_1, \ldots, v_{n-1}$ are ignored, but the environments from evaluating $e_1$ are used when evaluating $e_2$, and so on. By seeing how the environment from one expression is used to evaluate the next, we can understand the order of evaluation of expressions (see also Exercise 21). The order of the premises themselves is irrelevant. For example, the BEGIN rule might equally well have been written this way:

$$\frac{\begin{array}{c}\langle e_n, \xi_{n-1}, \phi, \rho_{n-1}\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle \\ \langle e_{n-1}, \xi_{n-2}, \phi, \rho_{n-2}\rangle \Downarrow \langle v_{n-1}, \xi_{n-1}, \phi, \rho_{n-1}\rangle \\ \vdots \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle\end{array}}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle} \quad \text{(equivalent BEGIN)}$$

This equivalent rule still specifies that $e_1$ is evaluated before $e_2$, etc., but when the rule is written this way, it is not as easy to understand.

### 2.4.6 Function Application

#### User-defined functions

The description of user-defined functions begins to show one of the advantages of an operational semantics: it is much more concise than an implementation.

$$\frac{\begin{array}{c}\phi(f) = \textsc{user}(\langle x_1, \ldots, x_n\rangle, e) \\ x_1, \ldots, x_n \text{ all distinct} \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1}\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle \\ \langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle\end{array}}{\langle \textsc{apply}(f, e_1, \ldots, e_n), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v, \xi', \phi, \rho_n\rangle} \quad (\textsc{ApplyUser})$$

As in the BEGIN rule, expressions $e_1$ through $e_n$ are evaluated in order. We then create a new, unnamed formal-parameter environment that maps the formal parameter names $x_1, \ldots, x_n$ to the results of evaluating the expressions, and we evaluate the body of the function in this new environment. By reading this rule carefully, we can draw several conclusions:

- The behavior of a function doesn't depend on the function's name, but only on the definition to which the name is bound.
- The body of a function can't get at the formal parameters of its caller, since the body $e$ is evaluated in a state that does not contain $\rho_0, \ldots, \rho_n$.
- If a function assigns to its own formal parameters, its caller can't see the new values because the caller has no access to the environment $\rho'$.
- After the body of a function is evaluated, the environment $\rho'$ containing the values of its formal parameters is thrown away. This fact is very important to implementors of programming languages, who can use temporary space (in registers and on the stack) to implement formal-parameter environments.

Taken together, these facts mean that the formal parameters of a function are private to that function—neither its caller nor its callees can see them or modify them. The privacy of formal parameters is an essential part of what language designers call "functional abstraction," which both programmers and implementors rely on.

#### Primitive functions

Evaluation of primitives is very similar to evaluation of user-defined functions. We evaluate the arguments and then perform the operation.

As a representative of the arithmetic primitives, here is addition:

$$\frac{\begin{array}{c}\phi(f) = \textsc{primitive}(+) \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \langle e_2, \xi_1, \phi, \rho_1\rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2\rangle\end{array}}{\langle \textsc{apply}(f, e_1, e_2), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1 + v_2, \xi_2, \phi, \rho_2\rangle} \quad (\textsc{ApplyAdd})$$

As a representative of the comparison primitives, here is the test for equality. As with the `if` expression, we use two rules with mutually exclusive premises ($v_1 = v_2$ and $v_1 \neq v_2$):

$$\frac{\begin{array}{c}\phi(f) = \text{PRIMITIVE}(\texttt{=}) \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \langle e_2, \xi_1, \phi, \rho_1\rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2\rangle \\ v_1 = v_2\end{array}}{\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0\rangle \Downarrow \langle 1, \xi_2, \phi, \rho_2\rangle} \qquad \text{(APPLYEQTRUE)}$$

$$\frac{\begin{array}{c}\phi(f) = \text{PRIMITIVE}(\texttt{=}) \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \langle e_2, \xi_1, \phi, \rho_1\rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2\rangle \\ v_1 \neq v_2\end{array}}{\langle \text{APPLY}(f, e_1, e_2), \xi_0, \phi, \rho_0\rangle \Downarrow \langle 0, \xi_2, \phi, \rho_2\rangle} \qquad \text{(APPLYEQFALSE)}$$

The other arithmetic and comparison primitives are so similar to addition and equality that they are not worth including here.

Because our formal model does not include output, the final primitive, `print`, appears to be the identity function.

$$\frac{\begin{array}{c}\phi(f) = \text{PRIMITIVE}(\texttt{print}) \\ \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle\end{array}}{\langle \text{APPLY}(f, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle \quad \text{while printing } v} \qquad \text{(APPLYPRINT)}$$

Modeling `print` formally would not be difficult; we could extend the program state to include a list of all values ever printed. We prefer, however, not to clutter our state and rules with such a list. It is OK to leave the specification of printing informal because our operational semantics is intended to convey understanding, not to nail down every last detail.

### 2.4.7 Rules for evaluating definitions

The rules above apply to the evaluation of expressions. The remaining rules apply to evaluation of definitions. These rules use the judgment $\langle d, \xi, \phi\rangle \to \langle \xi', \phi'\rangle$, and they are implemented by the `evaldef` function in Section 2.5.2.

#### Variable definition

Each global variable must be defined before use. $\text{VAL}(x, e)$ adds the binding $x \mapsto v$ to the global environment $\xi'$, where $v$ is the result of evaluating $e$. When an expression is evaluated at top level, there are no formal parameters, so $e$ is evaluated with an empty $\rho$. When $x$ is already a global variable, $\text{VAL}(x, e)$ behaves just like $\text{SET}(x, e)$. The DEFINEGLOBAL rule is almost identical to the GLOBALASSIGN rule, but the DEFINEGLOBAL rule does not require $x \in \operatorname{dom} \xi$.

$$\frac{\langle e, \xi, \phi, \{\}\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \text{VAL}(x, e), \xi, \phi\rangle \to \langle \xi'\{x \mapsto v\}, \phi\rangle} \qquad \text{(DEFINEGLOBAL)}$$

**Function definition**

To process a function definition, the interpreter packages the formal parameters and body as $\textsc{user}(\langle x_1, \ldots, x_n \rangle, e)$, then binds the package into the function-definition environment $\phi$. The DEFINEFUNCTION rule enforces the invariant that the names of formal parameters are not duplicated.

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \textsc{define}(f, \langle x_1, \ldots, x_n \rangle, e), \xi, \phi \rangle \to \langle \xi, \phi\{f \mapsto \textsc{user}(\langle x_1, \ldots, x_n \rangle, e)\} \rangle} \quad (\textsc{DefineFunction})$$

**Top-level expression**

Evaluating an expression can modify the global-variable environment $\xi$ but not the function environment $\phi$. In fact, evaluating a top-level expression $e$ *always* modifies the global environment $\xi$: in addition to whatever is modified during the evaluation of $e$, in the final environment, the special variable `it` is bound to $v$, the result of evaluating $e$.

$$\frac{\langle e, \xi, \phi, \{\} \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle}{\langle \textsc{exp}(e), \xi, \phi \rangle \to \langle \xi'\{\texttt{it} \mapsto v\}, \phi \rangle} \quad (\textsc{EvalExp})$$

If you read the rules carefully, you should see that evaluating the "definition" $\textsc{exp}(e)$ has exactly the same effects as evaluating the more conventional definition $\textsc{val}(\texttt{it}, e)$.

## 2.5 The interpreter

The rest of this chapter presents C code for our interpreter. We've written the code (and this book) using the Noweb system for *literate programming*, which extracts both book and code from a single source. This source contains the text of the book interleaved with named "code chunks." The code chunks are written in the order best suited to explaining the interpreter, not the order dictated by a C compiler. Chunks contain source code and references to other chunks. The names of chunks appear italicized and in angle brackets, as in ⟨*evaluate* `e->u.ifx` *and return the result* 38a⟩. The label "38a" shows what page the definition is on. At the definition, the label appears in the left margin. When multiple definitions appear on the same page, each one gets a distinct lower-case letter, which is appended to the page number. Each definition is shown using an ≡ sign. A definition can be continued in a later chunk; Noweb concatenates the contents of all definitions of the same chunk. A definition that continues a previous definition is identified by a +≡ sign in place of the ≡ sign. When a chunk's definition is continued, Noweb includes pointers to the previous and next definitions, written "◁35e" and "42b▷;" these pointers appear at the right margin. The notation "(36a)" shows where a chunk is used.

To help you make connections between chunks that appear on different pages, the Noweb system does some *identifier cross-reference*. For example, if you look on page 44, on the bottom outside margin you'll see an alphabetical list of identifiers: a *mini-index*. The mini-index tells you, for example, that function `filereader` is declared in chunk 31c on page 31. It also tells you that functions `fopen` and `fclose` are not defined anywhere in the book; they are part of the *initial basis* of the C language. (In practice, they are defined by the include file `<stdio.h>`, but we refer to any built-in function as a basis function.) In any mini-index, $\mathcal{B}$ stands for a basis function, and $\mathcal{A}$ stands for an automatically generated function. Finally, $\mathcal{P}$ stands for a *primitive* function that is defined in an interpreter, in which case a chunk number is also given. Examples of primitives appear starting in Chapter 3.

In addition to the mini-indices, there are two large indices at the back of the book. The "Author and Concept Index" on page ?? provides the sort of index you might find in any textbook. The "Code Index" on page 753 allows you to look up the definition—or in the case of C code, the declaration—of any function in any interpreter.

Once you find the code you're looking for, you may have an easier time reading it if you know our programming conventions. For example, when we introduce a new type, we use `typedef` to give it a name that begins with a capital letter, like `Name`, or `Exp`, or `Def`. The representation of such a type might be *exposed*, in which case you get to see the entire definition of the type, and you can get at fields of structures and so on. `Exp` and `Def` are exposed types. The representation of such a type might also be *abstract*, in which case you *can't* get at the representation. In C, the only way to make a type abstract is to make it a pointer to a named `struct`, but not to give the fields of the struct. `Name` is an abstract type; you can store a `Name` in a field or a variable, and you can pass a `Name` to a function, but you can't look inside a `Name` to see how it is represented.

We write the names of functions using lowercase letters only, except for some automatically generated functions used to build lists.

When possible, we initialize each variable where it is declared, just like a `val` definition in Impcore.

Although we sometimes write simply `x` instead of `x!=NULL`, we usually avoid writing `!x`; at 3:00AM, it is easier to see `x==NULL` or `x==0`.

To the degree that C allows, we are careful to distinguish interfaces from implementations. You should be able to use a module after seeing only its interface, without worrying about how the interface is implemented.

The code we present here shows how to apply the ideas of abstract syntax, environments, and operational semantics to implement a language. We don't care about the details of how to get from concrete syntax to abstract syntax (lexing and parsing). We have therefore divided the interpreter into two parts, and we have put the less interesting part in Appendix A. The more interesting part includes these interfaces:

| | |
|---|---|
| `ast` | Abstract syntax |
| `env` | Environments |
| `error` | Error reporting |
| `eval` | The evaluator—core of the language semantics |
| `interp` | The read-eval-print loop and error functions |
| `list` | Routines for constructing and using lists |
| `name` | Name management |
| `print` | An extensible version of `printf` |
| `reader` | Reading abstract syntax from various inputs |

This chapter also presents the implementations of some of these interfaces. The rest of the implementations are in Appendix A, along with these *uninteresting* interfaces and their implementations:

| | |
|---|---|
| `lex` | The lexical analyzer (or scanner) |
| `parse` | The parser |

In order to understand any of the code, you have to understand the interfaces in Section 2.5.1, but the interesting stuff—the *programming-language* content—is in the implementation in Section 2.5.2, which starts on page 35.

### 2.5.1 Interfaces

C provides poor support for separating interfaces from implementations. The best a programmer can do is put interfaces in `.h` files and use the C preprocessor to `#include` those `.h` files where they are needed. It is up to the programmer to ensure that the right files are included, that they are included in the right order, and that no file is included more than once; the C language and preprocessor don't help. C programmers have developed conventions to deal with these problems, but these conventions are better suited to large software projects than to small interpreters. We have therefore chosen simply to put all the interfaces into one header file, `all.h`.

The header file is split into three parts: type definitions, then structure definitions, then function prototypes. These splits make it easy for functions or structures declared in one interface to use types defined in another; because declarations and definitions of types always precede the function prototypes that use those types, we need not worry about getting things in the right order. So we can reuse some of the code in this interpreter in later interpreters, we also distinguish between shared and unshared definitions; a definition is "shared" if it is used in another interpreter later in the book.

25 ⟨`all.h` *for Impcore* 25⟩≡

```
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
#include <setjmp.h>
#include <ctype.h>
⟨shared type definitions 28c⟩
⟨type definitions for Impcore 28a⟩
⟨shared structure definitions 588c⟩
⟨structure definitions for Impcore 32b⟩
⟨shared function prototypes 28d⟩
⟨function prototypes for Impcore 27b⟩
```

Each interface not only declares types, structures, and functions but also explains how to use them. The interface also explains what happens when a function is used incorrectly; in particular, it explains who is responsible for detecting or avoiding an error. A *checked run-time error* is a mistake that the implementation guarantees to detect; passing a `NULL` pointer to a function is a typical example of a checked error. The implementation need not *recover* from a checked error, and indeed, many of our implementations simply print an error message and halt.

An *unchecked* run-time error is more insidious; this is a mistake that it is up to the C programmer to avoid. If a C programmer makes an unchecked run-time error, the implementation provides no guarantees; anything can happen. Unchecked run-time errors are part of the price we pay for programming in C.

**Sidebar: Safety**

A language in which all errors are checked is called *safe*. Safety is usually implemented by a combination of compile-time and run-time checking. Popular safe languages include Awk, C#, Haskell, Java, Lua, ML, Perl, Python, Ruby, Scheme, and Smalltalk. Another way to characterize a safe language is that "there are no unexplained core dumps;" a program that halts always issues an informative error message.

A language that permits unchecked errors is called *unsafe*. Unsafe languages put an extra burden on the programmer, but they provide extra expressive power. This extra power is needed to write things like garbage collectors and device drivers; systems programming languages, like Bliss and C, have historically been unsafe. C++ is an anomaly: it is ostensibly intended for high-level problem-solving, but it is nevertheless unsafe.

A few very interesting systems-programming languages are safe by default, but have unsafe features that can be turned on explicitly at need, usually by a keyword `UNSAFE`. Among these languages, the best known may be Cedar and Modula-3.

### Abstract Syntax

The type of an abstract-syntax tree is a *sum type*, also known as a *discriminated-union type*. Any value of such a type is one of a list of alternatives, which are given explicitly in the definition of the sum type. Although sum types are extremely useful in symbolic computing, they are not directly supported in C, which provides only an *undiscriminated* union, sometimes also called an "unsafe" union. A C union provides a list of alternatives, but there is no way to tell which alternative was last assigned to the union. Accordingly, to represent a sum type in C, we require both a union, to hold the alternatives, and a *discriminant* (or "tag"), to show which alternative is actually in use. For convenience, we put the union and the tag together in a structure, calling the union `u` and the tag `alt`. Also for convenience, we name each alternative, and we use the names both in an enumeration, which lists the possible values of `alt`, and to identify the members of the union. In keeping with a common C practice, we write enumeration literals using all capital letters; for members of the union, we write the lowercase equivalents.

The notation required to use values of sum types can get unwieldy. For example, to refer to the name of a function in a definition `d`, we need `d->u.define.name`. Although something like `d->name` might be more convenient, using `define` lets the compiler prevent us from accidentally using fields in other parts of the sum. Other languages provide better notation. For example, C++ and some dialects of C provide anonymous unions and structures, which can provide less unwieldy notation for the same representation of sum types. ML provides direct support for sum types, including a convenient pattern-matching notation.

Using our conventions, here are the definitions used to implement `Def`, a simplified version of which appears in chunk ⟨*simplified example of abstract syntax for Impcore* 14a⟩.

27a ⟨*type and structure definitions for Impcore* 27a⟩≡ 29e▷

```
typedef struct Userfun Userfun;
struct Userfun { Namelist formals; Exp body; };

typedef struct Def *Def;
typedef enum { VAL, EXP, DEFINE, USE } Defalt;
struct Def {
    Defalt alt;
    union {
        struct { Name name; Exp exp; } val;
        Exp exp;
        struct { Name name; Userfun userfun; } define;
        Name use;
    } u;
};
```

To make these structures easy to create, we define a creator function for each alternative in the sum, as well as for `Userfun`.

27b ⟨*function prototypes for Impcore* 27b⟩≡ (25) 28b▷

```
Userfun mkUserfun(Namelist formals, Exp body);
Def mkVal(Name name, Exp exp);
Def mkExp(Exp exp);
Def mkDefine(Name name, Userfun userfun);
Def mkUse(Name use);
struct Def mkValStruct(Name name, Exp exp);
struct Def mkExpStruct(Exp exp);
struct Def mkDefineStruct(Name name, Userfun userfun);
struct Def mkUseStruct(Name use);
```

Writing such type definitions and creator functions by hand is tedious and prone to error, especially when type definitions change. We therefore generate them automatically. Appendix L shows an ML program that generates such code from the following descriptions.

27c ⟨*definition.t* 27c⟩≡

```
Userfun = (Namelist formals, Exp body)
Def*    = VAL    (Name name, Exp exp)
        | EXP    (Exp)
        | DEFINE (Name name, Userfun userfun)
        | USE    (Name)
```

We maintain the invariant that the names in `formals` are all distinct.

Here is the corresponding description for `Exp`:

27d ⟨*exp.t* 27d⟩≡

```
Exp* = LITERAL (Value)
     | VAR     (Name)
     | SET     (Name name, Exp exp)
     | IFX     (Exp cond, Exp true, Exp false)
     | WHILEX  (Exp cond, Exp exp)
     | BEGIN   (Explist)
     | APPLY   (Name name, Explist actuals)
```

The descriptions above are slightly elaborated versions of ⟨*simplified example of abstract syntax for Impcore* 14a⟩. We use similar descriptions for much of the C code in this book.

We also define a type for lists of `Exp`s.

28a ⟨*type definitions for Impcore* 28a⟩≡ (25) 29a▷
```
typedef struct Explist *Explist; /* list of Exp */
```

An `Explist` is list of pointers of type `Exp`. We use this naming convention throughout our C code. The definitions of our list types are in the lists interface, in chunk 32b.

For debugging, it's convenient to be able to print abstract-syntax trees, and we provide functions to do so. These functions are set up to work with our extensible replacement for `printf` (Section 2.5.1); the type `Printer` is defined in chunk 34a.

28b ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁27b 29b▷
```
Printer printexp, printdef;
```

### Names

Programming languages are full of names. To make it easy to compare names and look them up in tables, we define an abstract data type to represent them. The essential feature of this abstract type is that a name has no internal structure; names are atomic objects which can efficiently be compared for equality.

For real implementations, it is convenient to build names from strings. Unlike C strings, names are immutable, and they can be compared using pointer equality.

28c ⟨*shared type definitions* 28c⟩≡ (25) 31a▷
```
typedef struct Name *Name;
typedef struct Namelist *Namelist;   /* list of Name */
```

Pointer comparison is built into C, but we provide two other operations on names.

28d ⟨*shared function prototypes* 28d⟩≡ (25) 28e▷
```
Name strtoname(const char *s);
const char *nametostr(Name n);
```

These functions satisfy the following algebraic laws:

```
strcmp(s, nametostr(strtoname(s))) == 0
strcmp(s, t) == 0 if and only if strtoname(s) == strtoname(t)
```

Informally, the first law says if you build a name from a string, `nametostr` returns a copy of your original string. The second law says you can compare names using pointer equality.

Because `nametostr` returns a string of type `const char*`, a client of `nametostr` cannot modify that string without subverting the type system. Modification of the string is an unchecked runtime error.

The `Name` type is abstract; the interface does not give the members of `struct Name`. A client should create new values of type `Name*` only by calling `strtoname`; to do so by casting other pointers is a violation of the type system and an unchecked run-time error.

Function `printname` prints names.

28e ⟨*shared function prototypes* 28d⟩+≡ (25) ◁28d 31b▷
```
Printer printname;
```

### Values

The value interface defines the type of value that our expressions evaluate to. Impcore supports only integers. A `Valuelist` is a list of `Values`.

29a ⟨*type definitions for Impcore* 28a⟩+≡ (25) ◁28a 29c▷
```
typedef int Value;
typedef struct Valuelist *Valuelist;     /* list of Value */
```

29b ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁28b 29f▷
```
Printer printvalue;
```

### Functions

In the Impcore interpreter, the type "function" is another discriminated-union type. There are two alternatives: user-defined functions and primitive functions. Just like the operational semantics, which represents a user-defined function as $\textsc{user}(\langle x_1, \ldots, x_n \rangle, e)$, the interpreter represents a user-defined function as a pair containing formals and body. The interpreter represents each primitive by its name.

29c ⟨*type definitions for Impcore* 28a⟩+≡ (25) ◁29a 30a▷
```
typedef struct Funlist *Funlist; /* list of Fun */
```

29d ⟨*fun.t* 29d⟩≡
```
Fun = USERDEF   (Userfun)
    | PRIMITIVE (Name)
```

We automatically generate these type and structure definitions.

29e ⟨*type and structure definitions for Impcore* 27a⟩+≡ ◁27a
```
typedef struct Fun Fun;
typedef enum { USERDEF, PRIMITIVE } Funalt;
struct Fun { Funalt alt; union { Userfun userdef; Name primitive; } u; };
```

We also generate prototypes for creator functions.

29f ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁29b 29g▷
```
Fun mkUserdef(Userfun userdef);
Fun mkPrimitive(Name primitive);
```

Finally, the module exports a print function.

29g ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁29f 30b▷
```
Printer printfun;
```

### Environments

In the operational semantics, the environments $\rho$ and $\xi$ hold values, and the environment $\phi$ holds functions. To represent these two kinds of environments, C offers these choices:

- We can define one C type for environments that hold a `Value` and another for environments that hold a `Fun`, and we can define two versions of each function. This choice guarantees type safety, but requires duplication of code.
- We can define a single C type for environments that hold a `void*` pointer, define a single version of each function, and use type casting to convert a `void*` to a `Value*` or `Fun*` as needed. This choice duplicates no code, but it is unsafe; if we accidentally put a `Value*` in an environment intended to hold a `Fun*`, it is an error that neither the C compiler nor the run-time system can detect.

In the interests of safety, we duplicate code. Chapter 5 shows how in another implementation language, ML, we can use *polymorphism* to achieve type safety without duplicating code.

30a ⟨*type definitions for Impcore* 28a⟩+≡ (25) ◁29c
```
typedef struct Valenv *Valenv;
typedef struct Funenv *Funenv;
```

A new environment may be created by passing a list of names and a list of associated values or function definitions. For example, `mkValenv`$(\langle x_1, \ldots, x_n \rangle, \langle v_1, \ldots, v_n \rangle)$ returns the environment $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. If the two lists are not the same length, it is a checked runtime error.

30b ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁29g 30c▷
```
Valenv mkValenv(Namelist vars, Valuelist vals);
Funenv mkFunenv(Namelist vars, Funlist   defs);
```

To retrieve a value or function definition, we use `fetchval` or `fetchfun`. In the operational semantics, we write the lookup `fetchval`$(x, \rho)$ simply as $\rho(x)$.

30c ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁30b 30d▷
```
Value fetchval(Name name, Valenv env);
Fun   fetchfun(Name name, Funenv env);
```

If the given name does not appear in the environment, it is a checked runtime error. To avoid such errors, we can call `isvalbound` or `isfunbound`; they return `1` if the given name is in the environment, and `0` otherwise. Formally, `isvalbound`$(x, \rho)$ is written $x \in \operatorname{dom} \rho$.

30d ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁30c 30e▷
```
int isvalbound(Name name, Valenv env);
int isfunbound(Name name, Funenv env);
```

To add new bindings to an environment, use `bindval` and `bindfun`. Unlike previous operations on environments, `bindval` and `bindfun` cannot be specified as pure functions. Instead, `bindval` and `bindfun` *mutate* their environments, replacing the old bindings with new ones. Calling `bindval`$(x, v, \rho)$ is equivalent to performing the assignment $\rho := \rho\{x \mapsto v\}$. Because $\rho$ is a *mutable* abstraction, the caller can see the modifications to the environment.

30e ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁30d 31g▷
```
void bindval(Name name, Value val, Valenv env);
void bindfun(Name name, Fun   fun, Funenv env);
```

These functions can be used to replace existing bindings or to add new ones.

### Readers for characters and definitions

The details of reading characters and converting them to abstract syntax are interesting, but they are more relevant to study of compiler construction than to study of programming languages. We relegate the implementations to Appendix A, giving here only the interfaces.

**Reading characters** A `Reader` encapsulates a source of characters: a string or a file. The `readline` function prints a prompt, reads the next line of input from the source, and returns a pointer to the line. The caller needn't worry about how long the line is; `readline` always allocates enough memory to hold it. Because `readline` reuses the same memory to hold successive lines, it is an unchecked run-time error to retain a pointer returned by `readline` after a subsequent call to `readline`. A client that needs to save input characters must copy the result of `readline` before calling `readline` again.

31a ⟨*shared type definitions* 28c⟩+≡ (25) ◁28c 31d▷
```
typedef struct Reader *Reader;
```

31b ⟨*shared function prototypes* 28d⟩+≡ (25) ◁28e 31c▷
```
char *readline(Reader r, char *prompt);
```

Each of our interpreters leaves the prompt empty when not reading interactively.

To create a reader, we need a string or a file. A reader also gets a name, which we can use in error messages.

31c ⟨*shared function prototypes* 28d⟩+≡ (25) ◁31b 31e▷
```
Reader stringreader(const char *stringname, const char *s);
Reader filereader  (const char *filename,   FILE *fin);
```

**Reading definitions** A `Defreader` encapsulates a source of definitions. The `readdef` function returns the next definition. It may also return `NULL`, if there are no more definitions in the source, or call `error` (page 35), if there is some problem converting the source to abstract syntax.

31d ⟨*shared type definitions* 28c⟩+≡ (25) ◁31a 34a▷
```
typedef struct Defreader *Defreader;
```

31e ⟨*shared function prototypes* 28d⟩+≡ (25) ◁31c 31f▷
```
Def readdef(Defreader r);
```

To create a definition reader, we need a source of characters.

31f ⟨*shared function prototypes* 28d⟩+≡ (25) ◁31e 33f▷
```
Defreader defreader(Reader r, int doprompt);
```

The `doprompt` flag controls whether the resulting `Defreader` prompts when reading input.

### Evaluation

The evaluator's interface is not very interesting: it is the *implementation*, which starts on page 35, that is interesting. The function `evaldef` corresponds to the $\rightarrow$ relation in our operational semantics, while `eval` corresponds to the $\Downarrow$ relation. For example, `eval`$(e, \xi, \phi, \rho)$ finds a $v$, $\xi'$, and $\rho'$ such that $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$, assigns $\rho := \rho'$ and $\xi := \xi'$, and returns $v$.

31g ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁30e 32a▷
```
void  evaldef(Def d, Valenv globals, Funenv functions, int echo);
Value eval   (Exp e, Valenv globals, Funenv functions, Valenv formals);
```

If the `echo` parameter to `evaldef` is nonzero, `evaldef` prints the values and names of top-level expressions and functions. As with the formal rules of the operational semantics, we can see from the result types that evaluating a `Def` does not produce a `Value`, but evaluating an `Exp` does produce a value. Both kind of evaluations can have side effects on environments.

The function `readevalprint` consumes all of an input source, evaluating each definition. As for `evaldef`, the `echo` parameter controls whether `readevalprint` prints the values and names of top-level expressions and functions.

32a ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁31g 33a▷

```
void readevalprint(Defreader r, Valenv globals, Funenv functions, int echo);
```

### Lists

The interpreter uses lists of names, values, functions, top-level expressions, and expressions. As with environments, C offers two choices. We can define five separate list types, getting safety but slightly bloating our code, or we can use lists of `void*`, getting compactness but losing safety. Again, we have chosen safety.[7]

Lists are recursive data types. Either a list is empty, or it is a pointer to a pair (`hd`, `tl`), where `hd` is the first element of the list and `tl` is the rest of the list. We use the null pointer to represent the empty list.

32b ⟨*structure definitions for Impcore* 32b⟩≡ (25)

```
struct Explist {
    Exp hd;
    Explist tl;
};

struct Parlist {
    Par hd;
    Parlist tl;
};

struct Valuelist {
    Value hd;
    Valuelist tl;
};

struct Funlist {
    Fun hd;
    Funlist tl;
};

struct Namelist {
    Name hd;
    Namelist tl;
};
```

[7] Programmers who use C++ templates have made the same choice. Early implementations of templates were notorious for code bloat, not only because they had a separate implementation for each type, but also because they might have a dozen duplicate implementations of `List<Exp>` alone, one for every module in which `List<Exp>` is used.

These definitions are generated by an Awk script, which searches header files for lines of the form

```
typedef struct Foo *Foo; /* list of Quux */
```

The script also generates a length function, an extractor, a creator function, and a printer for each type of list. These functions are named `length`$N$`L`, `nth`$N$`L`, `mk`$N$`L`, and `print`$N$`L`, where $N$ is the first letter of the list type. The length of the `NULL` list is zero; the length of other lists is the number of elements. Elements are numbered from zero, and asking for `nth`$N$`L(l, n)` when `n` $\geq$ `length`$N$`L(l)` is a checked run-time error. Calling `mk`$N$`L` creates a fresh list with the new element at the head; it does not mutate the old list.

Here are the prototypes:

33a ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁32a 33b▷

```
int     lengthEL (Explist l);
Exp     nthEL    (Explist l, unsigned n);
Explist mkEL     (Exp hd, Explist tl);
Printer printexplist;
```

33b ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁33a 33c▷

```
int     lengthPL (Parlist l);
Par     nthPL    (Parlist l, unsigned n);
Parlist mkPL     (Par hd, Parlist tl);
Printer printparlist;
```

33c ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁33b 33d▷

```
int       lengthVL (Valuelist l);
Value     nthVL    (Valuelist l, unsigned n);
Valuelist mkVL     (Value hd, Valuelist tl);
Printer   printvaluelist;
```

33d ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁33c 33e▷

```
int     lengthFL (Funlist l);
Fun     nthFL    (Funlist l, unsigned n);
Funlist mkFL     (Fun hd, Funlist tl);
Printer printfunlist;
```

33e ⟨*function prototypes for Impcore* 27b⟩+≡ (25) ◁33d

```
int      lengthNL (Namelist l);
Name     nthNL    (Namelist l, unsigned n);
Namelist mkNL     (Name hd, Namelist tl);
Printer  printnamelist;
```

Generating all this code automatically makes the fivefold replication bearable. As shown in Chapter 5, ML's polymorphism enables a simpler solution.

### Printing

The standard C function `printf` and its siblings `fprintf` and `vfprintf` convert internal C values to sequences of characters in an output file. Unfortunately, they convert only numbers, characters, and strings. It would be a lot more convenient if `printf` also converted `Name`s, `Exp`s, and so on. We can't change `printf`, so instead we define new functions. Functions `print` and `fprint` are a bit like `printf` and `fprintf`.

33f ⟨*shared function prototypes* 28d⟩+≡ (25) ◁31f 34b▷

```
void print (const char *fmt, ...);  /* print to standard output */
void fprint(FILE *output, const char *fmt, ...);  /* print to given file */
```

The advantage of our versions is that they are *extensible*; if we want to print new kinds of values, like Impcore expressions, we can add new printing functions.

To keep our code simple, we made `print` and `fprint` less powerful than their counterparts in the standard C library. Our "conversion specifications" are two characters: the first is always `%`, and the second is a character like `d` or `s`. Unlike conversion specifications in standard C, ours may not contain minus signs, numbers, periods, etc. The Impcore interpreter uses the following conversion specifications, which its `main` procedure installs (see ⟨*install conversion specifications in print module* 45c⟩).

- `%d` Print an integer in decimal format
- `%e` Print an `Exp`
- `%f` Print a `Fun`
- `%E` Print an `Explist` (list of `Exp`)
- `%n` Print a `Name`
- `%N` Print a `Namelist` (list of `Name`)
- `%p` Print a `Par` (see Appendix A)
- `%P` Print a `Parlist` (list of `Par`)
- `%s` Print a `char*` (string)
- `%t` Print a `Def` (`t` stands for "top-level", which is where definitions appear)
- `%v` Print a `Value`
- `%V` Print a `Valuelist` (list of `Value`)

By convention, we use lowercase letters to print individual values and uppercase letters to print lists. To print a `Def`, we cannot use `%d`, because `%d` is universally used to print decimal integers. Instead we use `%t` for "top-level," which is where a `Def` appears.

If you just want to use the functions, without learning how to extend them, you can use only `print` and `eprint`—ignore `vprint` and the details of `Printer` and `installprinter` that follow.

An extension to `print` or `eprint` knows how to print one type of C value, like an Impcore expression or value. Such a function has type `Printer`; a printer consumes and writes one argument from a `va_list_box`, which holds a list of arguments.

34a ⟨*shared type definitions* 28c⟩+≡ (25) ◁31d
```
⟨definition of va_list_box 35a⟩
typedef void Printer(FILE *output, va_list_box*);
```

To tell `print` and `eprint` about a new extension, we announce the extension by calling `installprinter` with the extension and with a character that is used for the extension's "conversion specification."

34b ⟨*shared function prototypes* 28d⟩+≡ (25) ◁33f 34c▷
```
void installprinter(unsigned char c, Printer *fmt);
```

The printer interface provides printers to format percent signs, strings, and decimal integers.

34c ⟨*shared function prototypes* 28d⟩+≡ (25) ◁34b 34d▷
```
Printer printpercent, printstring, printdecimal;
```

If you are a sophisticated C programmer, you may want to use our analog of `vfprintf`:

34d ⟨*shared function prototypes* 28d⟩+≡ (25) ◁34c 35b▷
```
void vprint(FILE *output, const char *fmt, va_list_box *box);
```

Extensible printers are popular with sophisticated C programmers; Hanson (1996, Chapter 14) presents an especially well-crafted example. Unfortunately, some widely used compilers make the construction of extensible printers more difficult than necessary. In particular, not all versions of the GNU C compiler work correctly when values of type `va_list` are passed as arguments on the AMD64 platform.[8] A workaround for this problem is to place the `va_list` in a structure and pass a pointer to the structure:

35a ⟨*definition of* `va_list_box` 35a⟩≡ (34a)
```
typedef struct va_list_box {
  va_list ap;
} va_list_box;
```

If you are not accustomed to variadic functions and `stdarg.h`, you may wish to consult Sections 7.2 and 7.3 of Kernighan and Ritchie 1988.

### Error handling

Any module can signal an error by calling `error`. When called, `error` acts just like `eprint` except that it does not return after printing; instead it `longjmp`s to `errorjmp`. It is an unchecked run-time error to call `error` except while a suitable `setjmp` is active on the C call stack.

35b ⟨*shared function prototypes* 28d⟩+≡ (25) ◁34d 35c▷
```
void error(const char *fmt, ...);
extern jmp_buf errorjmp;        /* longjmp here on error */
```

Function `checkargc` is used to check the number of arguments to both user-defined and primitive functions. The first argument is an abstract-syntax tree representing the application being checked; if `expected` ≠ `actual`, `checkargc` calls `error`, passing a message that contains `e`.

35c ⟨*shared function prototypes* 28d⟩+≡ (25) ◁35b 35d▷
```
void checkargc(Exp e, int expected, int actual);
```

The `duplicatename` function finds a duplicate name on a `Namelist` if such a name exists. It is used to check that formal parameters to user-defined functions all have different names. If the name list `names` contains a duplicate occurrence of any name, the function returns such a name; otherwise it returns `NULL`.

35d ⟨*shared function prototypes* 28d⟩+≡ (25) ◁35c
```
Name duplicatename(Namelist names);
```

## 2.5.2 Implementation of the evaluator

The most interesting parts of the implementation are the evaluator (functions `eval` and `evaldef`) and the `readevalprint` loop.

35e ⟨*eval.c* 35e⟩≡ 36a▷
```
#include "all.h"

⟨eval helpers 40a⟩
```

[8] Library functions such as `vfprintf` itself are grandfathered; only *users* cannot write functions that take `va_list` arguments. Feh.

### Evaluating expressions

The function `eval` implements the $\Downarrow$ relation from the operational semantics. Calling `eval`$(e, \xi, \phi, \rho)$ finds a $v$, $\xi'$, and $\rho'$ such that $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$, assigns $\rho := \rho'$ and $\xi := \xi'$, and returns $v$. Because it is unusual to use Greek letters in C code, we have chosen the following names:

| | |
|---|---|
| $\xi$ | `globals` |
| $\phi$ | `functions` |
| $\rho$ | `formals` |

Like any other recursive interpreter, `eval` implements the operational semantics by examining the inference rules from the bottom up. The first step in evaluating `e` is to discover what the syntactic form of `e` is, by looking at the tag (the `alt` field) in the discriminated union.

36a ⟨*eval.c* 35e⟩+≡ ◁35e 42b▷

```
Value eval(Exp e, Valenv globals, Funenv functions, Valenv formals) {
    switch (e->alt) {
    case LITERAL:
        ⟨evaluate e->u.literal and return the result 36b⟩
    case VAR:
        ⟨evaluate e->u.var and return the result 37a⟩
    case SET:
        ⟨evaluate e->u.set and return the result 37b⟩
    case IFX:
        ⟨evaluate e->u.ifx and return the result 38a⟩
    case WHILEX:
        ⟨evaluate e->u.whilex and return the result 38b⟩
    case BEGIN:
        ⟨evaluate e->u.begin and return the result 39a⟩
    case APPLY:
        ⟨evaluate e->u.apply and return the result 39b⟩
    default:
        assert(0);
        return 0;   /* not reached */
    }
}
```

For each syntactic form, we consult the operational semantics to find the rules that have the form on the left-hand sides of their *conclusions.*

Only one rule has LITERAL in its conclusion.

$$\frac{}{\langle \text{LITERAL}(v), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi, \phi, \rho\rangle} \quad \text{(LITERAL)}$$

checkoverflow 591c

The implementation simply returns the literal value.

36b ⟨*evaluate* `e->u.literal` *and return the result* 36b⟩≡ (36a)

```
return e->u.literal;
```

Two rules have variables in their conclusions.

$$\frac{x \in \mathrm{dom}\,\rho}{\langle \mathrm{VAR}(x), \xi, \phi, \rho\rangle \Downarrow \langle \rho(x), \xi, \phi, \rho\rangle} \quad \text{(FORMALVAR)}$$

$$\frac{x \notin \mathrm{dom}\,\rho \qquad x \in \mathrm{dom}\,\xi}{\langle \mathrm{VAR}(x), \xi, \phi, \rho\rangle \Downarrow \langle \xi(x), \xi, \phi, \rho\rangle} \quad \text{(GLOBALVAR)}$$

We know which rule to use by checking $x \in \mathrm{dom}\,\rho$, which in C is `isvalbound(e->u.var, formals)`. If $x \notin \mathrm{dom}\,\rho$ and $x \notin \mathrm{dom}\,\xi$, the operational semantics gets stuck—so the interpreter issues an error message. Less formally, we look up the variable by checking first the local environment and then the global environment.

37a ⟨*evaluate* `e->u.var` *and return the result* 37a⟩≡ (36a)

```
if (isvalbound(e->u.var, formals))
    return fetchval(e->u.var, formals);
else if (isvalbound(e->u.var, globals))
    return fetchval(e->u.var, globals);
else
    error("unbound variable %n", e->u.var);
assert(0);   /* not reached */
return 0;
```

The call to `error` illustrates the convenience of an extensible printer; we use `%n` to print a `Name` directly, without first needing to convert it to a string.

Setting a variable is very similar. Again there are two rules, and again we distinguish by looking at the domain of $\rho$ (`formals`).

$$\frac{x \in \mathrm{dom}\,\rho \qquad \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \mathrm{SET}(x, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\{x \mapsto v\}\rangle} \quad \text{(FORMALASSIGN)}$$

$$\frac{x \notin \mathrm{dom}\,\rho \qquad x \in \mathrm{dom}\,\xi \qquad \langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \mathrm{SET}(x, e), \xi, \phi, \rho\rangle \Downarrow \langle v, \xi'\{x \mapsto v\}, \phi, \rho'\rangle} \quad \text{(GLOBALASSIGN)}$$

Because both rules require the premise $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle$, we evaluate the right-hand side first and put the result in `v`.

37b ⟨*evaluate* `e->u.set` *and return the result* 37b⟩≡ (36a)

```
{
    Value v = eval(e->u.set.exp, globals, functions, formals);

    if (isvalbound(e->u.set.name, formals))
        bindval(e->u.set.name, v, formals);
    else if (isvalbound(e->u.set.name, globals))
        bindval(e->u.set.name, v, globals);
    else
        error("set: unbound variable %n", e->u.set.name);
    return v;
}
```

| | |
|---|---|
| bindval | 30e |
| error | 35b |
| eval | 36a |
| fetchval | 30c |
| formals | 36a |
| functions | 36a |
| globals | 36a |
| isvalbound | 30d |

To evaluate `ifx`, we again have two rules.

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2,\xi',\phi,\rho'\rangle \Downarrow \langle v_2,\xi'',\phi,\rho''\rangle}{\langle \text{IF}(e_1,e_2,e_3),\xi,\phi,\rho\rangle \Downarrow \langle v_2,\xi'',\phi,\rho''\rangle} \quad \text{(IfTrue)}$$

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 = 0 \qquad \langle e_3,\xi',\phi,\rho'\rangle \Downarrow \langle v_3,\xi'',\phi,\rho''\rangle}{\langle \text{IF}(e_1,e_2,e_3),\xi,\phi,\rho\rangle \Downarrow \langle v_3,\xi'',\phi,\rho''\rangle} \quad \text{(IfFalse)}$$

Both rules have the same first premise: $\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle$. We can find $v_1$, $\xi'$, and $\rho'$ by making the recursive call `eval(e->u.ifx.cond, globals, functions, formals)`. This call is safe only because the new environments $\xi'$ and $\rho'$ are used in the third premises of *both* rules. If $\xi'$ and $\rho'$ were not always used, we would have had to make copies and pass the copies to the recursive call.

Once we have $v_1$, testing it against zero tells us which rule to use, and the third premises of both rules require recursive calls to `eval`. The expression $e_2$ is `e->u.ifx.true`; $e_3$ is `e->u.ifx.false`.

38a ⟨*evaluate* `e->u.ifx` *and return the result* 38a⟩≡ (36a)

```
if (eval(e->u.ifx.cond, globals, functions, formals) != 0)
    return eval(e->u.ifx.true, globals, functions, formals);
else
    return eval(e->u.ifx.false, globals, functions, formals);
```

There are also two rules for while loops.

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 \neq 0 \qquad \langle e_2,\xi',\phi,\rho'\rangle \Downarrow \langle v_2,\xi'',\phi,\rho''\rangle \qquad \langle \text{WHILE}(e_1,e_2),\xi'',\phi,\rho''\rangle \Downarrow \langle v_3,\xi''',\phi,\rho'''\rangle}{\langle \text{WHILE}(e_1,e_2),\xi,\phi,\rho\rangle \Downarrow \langle v_3,\xi''',\phi,\rho'''\rangle} \quad \text{(WhileIterate)}$$

$$\frac{\langle e_1,\xi,\phi,\rho\rangle \Downarrow \langle v_1,\xi',\phi,\rho'\rangle \qquad v_1 = 0}{\langle \text{WHILE}(e_1,e_2),\xi,\phi,\rho\rangle \Downarrow \langle 0,\xi',\phi,\rho'\rangle} \quad \text{(WhileEnd)}$$

A simple translation of $\langle \text{WHILE}(e_1,e_2),\xi'',\phi,\rho''\rangle \Downarrow \langle v_3,\xi''',\phi,\rho'''\rangle$ where it appears as a premise of the first rule would require a recursive call to `eval(e, ...)`. Because we know `e` is a while loop, however, we can turn the recursion into iteration, writing the interpretation of Impcore's while loop as a `while` loop in C. Such an optimization is valuable because it can keep the C stack from overflowing during the execution of a long while loop in Impcore.

38b ⟨*evaluate* `e->u.whilex` *and return the result* 38b⟩≡ (36a)

```
while (eval(e->u.whilex.cond, globals, functions, formals) != 0)
    eval(e->u.whilex.exp, globals, functions, formals);
return 0;
```


eval 36a
formals 36a
functions 36a
globals 36a

For the `begin` expression, we use a `for` loop to evaluate all the premises in turn. In the pointless case where the `begin` doesn't contain any expressions, we have crafted the operational semantics to match the implementation.

$$\frac{}{\langle \text{BEGIN}(), \xi, \phi, \rho\rangle \Downarrow \langle 0, \xi, \phi, \rho\rangle} \quad \text{(EMPTYBEGIN)}$$

$$\frac{\begin{array}{c}\langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \langle e_2, \xi_1, \phi, \rho_1\rangle \Downarrow \langle v_2, \xi_2, \phi, \rho_2\rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1}\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle\end{array}}{\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle} \quad \text{(BEGIN)}$$

39a ⟨*evaluate* `e->u.begin` *and return the result* 39a⟩≡ (36a)

```
{
    Explist el;
    Value v = 0;
    for (el=e->u.begin; el; el=el->tl)
        v = eval(el->hd, globals, functions, formals);
    return v;
}
```

There are many rules for applying functions, but we divide them into two classes. One class contains only the rule for user-defined functions; the other class contains the rules for primitives. To apply function $f$, the interpreter looks at the form of $\phi(f)$.

39b ⟨*evaluate* `e->u.apply` *and return the result* 39b⟩≡ (36a)

```
{
    Fun f;
    ⟨make f the function denoted by e->u.apply.name, or call error 39c⟩
    switch (f.alt) {
    case USERDEF:
        ⟨apply f.u.userdef and return the result 40b⟩
    case PRIMITIVE:
        ⟨apply f.u.primitive and return the result 41a⟩
    default:
        assert(0);
        return 0;   /* not reached */
    }
}
```

If the lookup $\phi(f)$ fails, we call `error`.

39c ⟨*make* `f` *the function denoted by* `e->u.apply.name`, *or call* `error` 39c⟩≡ (39b)

```
if (!isfunbound(e->u.apply.name, functions))
    error("call to undefined function %n", e->u.apply.name);
f = fetchfun(e->u.apply.name, functions);
```

Applying a user-defined function has something in common with `begin`, because arguments $e_1, \ldots, e_n$ have to be evaluated. The difference is that `begin` keeps only result $v_n$ (in variable `v` in chunk 39a), where function evaluation keeps all the result values, to bind into a new environment.

$$\frac{\begin{array}{c}\phi(f) = \textsc{user}(\langle x_1, \ldots, x_n\rangle, e) \\ x_1, \ldots, x_n \text{ all distinct} \\ \langle e_1, \xi_0, \phi, \rho_0\rangle \Downarrow \langle v_1, \xi_1, \phi, \rho_1\rangle \\ \vdots \\ \langle e_n, \xi_{n-1}, \phi, \rho_{n-1}\rangle \Downarrow \langle v_n, \xi_n, \phi, \rho_n\rangle \\ \langle e, \xi_n, \phi, \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle\end{array}}{\langle \textsc{apply}(f, e_1, \ldots, e_n), \xi_0, \phi, \rho_0\rangle \Downarrow \langle v, \xi', \phi, \rho_n\rangle} \quad (\textsc{ApplyUser})$$

We use the auxiliary function `evallist`, which is given $e_1, \ldots, e_n$ along with $\xi_0$, $\phi$, and $\rho_0$. It evaluates $e_1, \ldots, e_n$ in order, mutates the environments so that when it is finished, $\xi = \xi_n$ and $\rho = \rho_n$. Finally, `evallist` returns the list $v_1, \ldots, v_n$.

40a ⟨*eval helpers* 40a⟩≡ (35e)

```
static Valuelist evallist(Explist el, Valenv globals, Funenv functions, Valenv formals) {
    if (el == NULL) {
        return NULL;
    } else {
        Value v = eval(el->hd, globals, functions, formals);
        return mkVL(v, evallist(el->tl, globals, functions, formals));
    }
}
```

The rules of Impcore require that `el->hd` be evaluated before `el->tl`. To ensure the correct order of evaluation, we must call `eval(el->hd, ...)` and `evallist(el->tl, ...)` in separate C statements. Writing both calls as parameters to `mkVL` would be a mistake because C makes no guarantees about the order in which the actual parameters of a function are evaluated.

The premises of the ApplyUser rule require that the list of formal parameters to $f$ be the same length as the list of actual parameters in the call. We let `nl` represent the formals, `vl` the actuals. If the formals and actuals are the same length, we use `mkValenv(nl, vl)` to create a fresh environment $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ in which to evaluate the body.

checkargc 35c
eval 36a
formals 36a
functions 36a
globals 36a
lengthNL 33e
lengthVL 33c
mkValenv 30b
mkVL 33c

⟨*apply* `f.u.userdef` *and return the result* 40b⟩≡ (39b)

```
{
    Namelist  nl = f.u.userdef.formals;
    Valuelist vl = evallist(e->u.apply.actuals, globals, functions, formals);
    checkargc(e, lengthNL(nl), lengthVL(vl));
    return eval(f.u.userdef.body, globals, functions, mkValenv(nl, vl));
}
```

Impcore has few primitive operators, and they are simple. We handle `print` separately from the arithmetic primitives. More general techniques for implementing primitives, which are appropriate for larger languages, are shown as part of the implementation of $\mu$Scheme, in Section 3.14.1.

41a ⟨*apply* `f.u.primitive` *and return the result* 41a⟩≡ (39b)

```
{
    Valuelist vl = evallist(e->u.apply.actuals, globals, functions, formals);
    if (f.u.primitive == strtoname("print"))
        ⟨apply print to vl and return 41b⟩
    else
        ⟨apply arithmetic primitive to vl and return 42a⟩
}
```

41b ⟨*apply* `print` *to* `vl` *and return* 41b⟩≡ (41a)

```
{
    Value v;
    checkargc(e, 1, lengthVL(vl));
    v = nthVL(vl, 0);
    print("%v\n", v);
    return v;
}
```

Because all the arithmetic primitives are single letters, we can use a switch on the first character of the name. This technique is reasonable only because the set of primitives is small and fixed. The code also depends on the fact that Impcore shares C's rules for the values of comparison expressions.

42a ⟨*apply arithmetic primitive to* vl *and return* 42a⟩≡ (41a)

```
{
    const char *s;
    Value v, w;

    checkargc(e, 2, lengthVL(vl));
    v = nthVL(vl, 0);
    w = nthVL(vl, 1);

    s = nametostr(f.u.primitive);
    assert(strlen(s) == 1);
    switch (s[0]) {
    case '<':
        return v < w;
    case '>':
        return v > w;
    case '=':
        return v == w;
    case '+':
        return v + w;
    case '-':
        return v - w;
    case '*':
        return v * w;
    case '/':
        if (w == 0)
            error("division by zero in %e", e);
        return v / w;
    default:
        assert(0);
        return 0;   /* not reached */
    }
}
```

checkargc 35c
error 35b
evaldef 43a
lengthVL 33c
nametostr 28d
nthVL 33c
readdef 31e
strlen B
vl 41a
42b

## Evaluating definitions

The `readevalprint` function processes a whole source of input, evaluating each definition in turn.

⟨*eval.c* 35e⟩+≡ ◁36a 43a▷

```
void readevalprint(Defreader reader, Valenv globals, Funenv functions, int echo) {
    Def d;
    while ((d = readdef(reader)))
        evaldef(d, globals, functions, echo);
}
```

The function `evaldef` implements the $\rightarrow$ relation from the operational semantics. That is, calling `evaldef`$(d, \xi, \phi,$ `echo`$)$ finds a $\xi'$ and $\phi'$ such that $\langle d, \xi, \phi\rangle \rightarrow \langle \xi', \phi'\rangle$, and `evaldef` mutates the C representation of the environments so the global-variable environment becomes $\xi'$ and the function environment becomes $\phi'$. If `echo` is nonzero, `evaldef` also prints the interpreter's response to the user's input. Printing the response is `evaldef`'s job because only `evaldef` can tell whether to print a value (for `exp` and `val`), a name (for `define`), or nothing at all (for `use`).

Just like `eval`, `evaldef` looks at the conclusions of rules, and it discriminates on the syntactic form of `t`.

43a ⟨*eval.c* 35e⟩+≡ ◁42b

```
void evaldef(Def d, Valenv globals, Funenv functions, int echo) {
    switch (d->alt) {
    case VAL:
        ⟨evaluate d->u.val, mutating globals 43b⟩
        break;
    case EXP:
        ⟨evaluate d->u.exp and possibly print the result 44a⟩
        break;
    case DEFINE:
        ⟨evaluate d->u.define, mutating functions 44b⟩
        break;
    case USE:
        ⟨evaluate d->u.use, possibly mutating globals and functions 44d⟩
        break;
    default:
        assert(0);
    }
}
```

The operational semantics dictates the cases for `VAL`, `EXP`, and `DEFINE`.

A variable definition updates $\xi$.

$$\frac{\langle e, \xi, \phi, \{\}\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \textsc{val}(x, e), \xi, \phi\rangle \rightarrow \langle \xi'\{x \mapsto v\}, \phi\rangle} \qquad (\textsc{DefineGlobal})$$

The premise shows we must call `eval` to get value $v$ and environment $\xi'$. When we call `eval`, we must use an empty environment as $\rho$. The rule says the new environment $\xi'$ is retained, and the value of the expression, $v$, is bound to $x$ in it. The implementation may also print $v$.

43b ⟨*evaluate* `d->u.val`, *mutating* `globals` 43b⟩≡ (43a)

```
{
    Value v = eval(d->u.val.exp, globals, functions, mkValenv(NULL, NULL));
    bindval(d->u.val.name, v, globals);
    if (echo)
        print("%v\n", v);
}
```

bindval 30e
mkValenv 30b
print 33f

Evaluating a top-level expression is just like evaluating a definition of `it`.

$$\frac{\langle e, \xi, \phi, \{\}\rangle \Downarrow \langle v, \xi', \phi, \rho'\rangle}{\langle \text{EXP}(e), \xi, \phi\rangle \to \langle \xi'\{\texttt{it} \mapsto v\}, \phi\rangle} \quad \text{(EVALEXP)}$$

44a ⟨*evaluate* `d->u.exp` *and possibly print the result* 44a⟩≡ (43a)

```
{
    Value v = eval(d->u.exp, globals, functions, mkValenv(NULL, NULL));
    bindval(strtoname("it"), v, globals);
    if (echo)
        print("%v\n", v);
}
```

A function definition updates $\phi$. Our implementation also prints the name of the function.

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{DEFINE}(f, \langle x_1, \ldots, x_n\rangle, e), \xi, \phi\rangle \to \langle \xi, \phi\{f \mapsto \text{USER}(\langle x_1, \ldots, x_n\rangle, e)\}\rangle} \quad \text{(DEFINEFUNCTION)}$$

44b ⟨*evaluate* `d->u.define`, *mutating* `functions` 44b⟩≡ (43a)

```
⟨fail if d->u.define has duplicate formal parameters 44c⟩
bindfun(d->u.define.name, mkUserdef(d->u.define.userfun), functions);
if (echo)
    print("%n\n", d->u.define.name);
```

As happens too often, the error checking is more work than the normal case.

44c ⟨*fail if* `d->u.define` *has duplicate formal parameters* 44c⟩≡ (44b)

```
if (duplicatename(d->u.define.userfun.formals) != NULL)
    error("Formal parameter named %n appears twice in definition of function %n",
          duplicatename(d->u.define.userfun.formals), d->u.define.name);
```

We don't have a formal definition of `use`. We open the file named by `use`, build a top-level reader, and through `readevalprint`, we recursively call `evaldef` on all the definitions in that file.

⟨*evaluate* `d->u.use`, *possibly mutating* `globals` *and* `functions` 44d⟩≡ (43a)

```
{
    const char *filename = nametostr(d->u.use);
    FILE *fin = fopen(filename, "r");
    if (fin == NULL)
        error("cannot open file \"%s\"", filename);
    readevalprint(defreader(filereader(filename, fin), 0), globals, functions, 0);
    fclose(fin);
}
```

As noted in Exercise 29, this code can leak open file descriptors.


bindfun 30e
bindval 30e
defreader 31f
duplicatename 35d
echo 43a
error 35b
fclose $\mathcal{B}$
filereader 31c
fopen $\mathcal{B}$
functions 43a
globals 43a
mkUserdef 29f
mkValenv 30b
nametostr 28d
print 33f
strtoname 28d

### 2.5.3 Implementing `main`

The `main` program recognizes the `-q` option ("quiet," to suppress prompts), initializes the interpreter, and runs the read-eval-print loop.

45a ⟨*impcore.c* 45a⟩≡

```
#include "all.h"

int main(int argc, char *argv[]) {
    Funenv functions = mkFunenv(NULL, NULL);
    Valenv globals   = mkValenv(NULL, NULL);
    int doprompt = (argc <= 1) || (strcmp(argv[1], "-q") != 0);
    Defreader input = defreader(filereader("standard input", stdin), doprompt);
    ⟨install conversion specifications in print module 45c⟩
    ⟨install the initial basis in functions 46a⟩
    while (setjmp(errorjmp))
        ;
    readevalprint(input, globals, functions, 1);
    return 0;
}
```

The interpreter is initialized in three phases, all of which are simple. First, we initialize empty environments and a definition reader. Then, we initialize the printers. Finally, we populate the `functions` environment with functions from the initial basis.

The code

45b ⟨*idiomatic error handler* 45b⟩≡

```
while (setjmp(errorjmp)) {
    ⟨recover from an error (never defined)⟩
}
```

is an idiom that is commonly seen when C programmers use `setjmp` and `longjmp` to deal with errors. On the first loop test, `setjmp` initializes `errorjmp` and returns zero, so the code in ⟨*recover from an error* (never defined)⟩ is not executed, and control continues following the `while` loop. If an error occurs later, the error routine calls `longjmp(errorjmp, 1)`, which returns control to the `setjmp` again, this time returning `1`. At this point the body of the `while` is executed. Since we don't have to do any work to recover from an error, we leave the body of the `while` empty. On the next iteration through the `while` statement, the process starts over from the beginning, because `setjmp` resets the jump buffer and returns zero again.

Implementations of printers are distributed throughout the interpreter, but all the printers are installed here.

45c ⟨*install conversion specifications in print module* 45c⟩≡ (45a)

```
installprinter('d', printdecimal);
installprinter('e', printexp);
installprinter('E', printexplist);
installprinter('f', printfun);
installprinter('n', printname);
installprinter('N', printnamelist);
installprinter('p', printpar);
installprinter('P', printparlist);
installprinter('s', printstring);
installprinter('t', printdef);
installprinter('v', printvalue);
installprinter('V', printvaluelist);
```

The initial basis includes both primitives and user-defined functions. We install the primitives first.

46a ⟨*install the initial basis in* `functions` 46a⟩≡ (45a) 46c▷
```
{
    static char *prims[] = { "+", "-", "*", "/", "<", ">", "=", "print", 0 };
    char **p;
    for (p=prims; *p; p++) {
        Name n = strtoname(*p);
        bindfun(n, mkPrimitive(n), functions);
    }
}
```

We represent the user-defined part of the initial basis as a single string, which is interpreted by `readevalprint`. These functions also appear in Figure 2.1 on page 13, from which this code is derived automatically.

46b ⟨*C representation of initial basis for Impcore* 46b⟩≡ (46c)
```
const char *basis=
    "(define and (b c) (if b c b))\n"
    "(define or  (b c) (if b b c))\n"
    "(define not (b)   (if b 0 1))\n"
    "(define <= (x y) (not (> x y)))\n"
    "(define >= (x y) (not (< x y)))\n"
    "(define != (x y) (not (= x y)))\n"
    "(define mod (m n) (- m (* n (/ m n))))\n";
```

46c ⟨*install the initial basis in* `functions` 46a⟩+≡ (45a) ◁46a
```
{
    ⟨C representation of initial basis for Impcore 46b⟩
    if (setjmp(errorjmp))
        assert(0); /* fail if error in basis */
    readevalprint(defreader(stringreader("initial basis", basis), 0),
                  globals, functions, 0);
}
```


_setjmp B
bindfun 30e
defreader 31f
errorjmp 35b
functions 45a
globals 45a
mkPrimitive 29f
readevalprint 32a
stringreader 31c
strtoname 28d


## 2.5.4 Implementing names

We include the implementations of names and environments because these data types are fundamental. The implementations are straightforward, and the techniques they use should be familiar.

Each name is associated with a string. We use a very simple representation in which the name simply contains the string.

⟨*name.c* 46d⟩≡ 47a▷
```
#include "all.h"

struct Name {
    const char *s;
};
```

Returning the string associated with a name is trivial.

47a ⟨*name.c* 46d⟩+≡ ◁46d 47b▷
```
const char* nametostr(Name n) {
    assert(n != NULL);
    return n->s;
}
```

Finding the name associated with a string is harder. We must return the same name for any copy of a string we have seen before. We implement this requirement very simply, by keeping `names`, a list of all names we have ever returned. A simple linear search gives us the name already associated with `s`, if any.

47b ⟨*name.c* 46d⟩+≡ ◁47a 47d▷
```
Name strtoname(const char *s) {
    static Namelist names;
    Namelist nl;

    assert(s != NULL);
    for (nl=names; nl; nl=nl->tl)
        if (strcmp(s, nl->hd->s) == 0)
            return nl->hd;

    ⟨allocate a new name, add it to names, and return it 47c⟩
}
```

A faster implementation might use a search tree or a hash table, not a simple list. Hanson (1996) shows such an implementation.

If the name is not on the list, we make a new one and add it.

47c ⟨*allocate a new name, add it to* `names`*, and return it* 47c⟩≡ (47b)
```
{
  Name n = malloc(sizeof(*n));
  assert(n != NULL);
  n->s = malloc(strlen(s) + 1);
  assert(n->s != NULL);
  strcpy((char*)n->s, s);
  names = mkNL(n, names);
  return n;
}
```

We print a name by printing its string.

47d ⟨*name.c* 46d⟩+≡ ◁47b
```
void printname(FILE *output, va_list_box *box) {
    Name n = va_arg(box->ap, Name);
    fputs(n == NULL ? "<null>" : nametostr(n), output);
}
```

fputs *B*
malloc *B*
mkNL,
in Impcore 33e
in μScheme *A*
strcmp *B*
strcpy *B*
strlen *B*

## 2.5.5 Implementing environments

In the interest of type safety, all the environment code is implemented twice: once for value environments and once for function environments. We show only the value-environment code here; the function-environment code appears in Appendix A.

47e ⟨*env.c* 47e⟩≡ 48a▷
```
#include "all.h"
```

We represent an environment as two parallel lists: one holding names and one holding values. It is an invariant of our representation that these lists have the same length.

48a ⟨*env.c* 47e⟩+≡ ◁47e 48b▷
```
struct Valenv {
    Namelist  nl;
    Valuelist vl;
};
```

Given the representation, creating an environment is simple. The assertion ensures that our invariant holds at the outset.

48b ⟨*env.c* 47e⟩+≡ ◁48a 48c▷
```
Valenv mkValenv(Namelist nl, Valuelist vl) {
    Valenv e = malloc(sizeof(*e));
    assert(e != NULL);
    assert(lengthNL(nl) == lengthVL(vl));
    e->nl = nl;
    e->vl = vl;
    return e;
}
```

To generalize searching a list for a value, we define function `findval`. It returns a pointer to the value corresponding to `name`, or it returns `NULL` if there is no binding for `name`. The pointer can be used to test for binding (`isvalbound`), to fetch a bound value (`fetchval`), or to change an existing binding (`bindval`). Code for the corresponding functions `findfun`, `isfunbound`, `fetchfun`, and `bindfun` appears in Appendix A.

Like `strtoname`, the `findval` function uses linear search. Hash tables or search trees would be faster but more complicated.

48c ⟨*env.c* 47e⟩+≡ ◁48b 48d▷
```
static Value* findval(Name name, Valenv env) {
    Namelist  nl;
    Valuelist vl;

    for (nl=env->nl, vl=env->vl; nl && vl; nl=nl->tl, vl=vl->tl)
        if (name == nl->hd)
            return &vl->hd;
    return NULL;
}
```

A name is bound if the find function found it.

48d ⟨*env.c* 47e⟩+≡ ◁48c 48e▷
```
int isvalbound(Name name, Valenv env) {
    return findval(name, env) != NULL;
}
```

lengthNL 33e
lengthVL 33c
malloc *B*

We fetch a value through the pointer returned by `findval`, if any.

48e ⟨*env.c* 47e⟩+≡ ◁48d 49▷
```
Value fetchval(Name name, Valenv env) {
    Value *vp = findval(name, env);
    assert(vp != NULL);
    return *vp;
}
```

You might think that to add a new binding to an environment, we would always have to insert a new binding at the beginning of the lists. But we can get away with an optimization. If $x \in \operatorname{dom} \rho$, instead of extending $\rho$ by making $\rho\{x \mapsto v\}$, we can overwrite the old binding of $x$. This optimization is safe only because no program written in Impcore can tell the difference. We can prove this by examining the rules of the operational semantics, which show that in any context where $\rho\{x \mapsto v\}$ appears, there is no way to get to the old $\rho(x)$. (See Exercise 20.)

49 ⟨*env.c* 47e⟩+≡ ◁48e

```
void bindval(Name name, Value val, Valenv env) {
    Value *vp = findval(name, env);
    if (vp != NULL)
        *vp = val;              /* safe optimization */
    else {
        env->nl = mkNL(name, env->nl);
        env->vl = mkVL(val,  env->vl);
    }
}
```

## 2.6 Operational semantics revisited: Proofs

If the interpreter in Section 2.5 is correct, then calling `eval`$(e, \xi, \phi, \rho)$ returns a value $v$ if and only if there is a *proof* of the judgment $\langle e, \xi, \phi, \rho\rangle \Downarrow \langle v, \xi, \phi, \rho\rangle$ using the rules from Section 2.4. Proofs in programming languages are not very interesting; or rather, they are interesting only when they are wrong. A wrong proof, or the failure to find a proof, tells you that you got something wrong in your language design—just like a bug in a program.

But there's more to proofs than being interesting. What a proof in programming languages *can* do for you is convey *certainty*. A proof provides a way of being utterly sure that your program does what you think it does, or your language does what you think it does.

### 2.6.1 Proofs about evaluation: Theory

If we are to draw certain conclusions from a proof, we must be very sure that we know what a proof is. Programming languages borrows from formal logic the representation of a proof as a *derivation* of a judgment. In a derivation, inference rules are instantiated and composed into a *derivation tree*. Just as in a single rule, the conclusion of the derivation appears on the bottom, below a horizontal line. Above the line are the premises needed to prove that conclusion. The premises must be related to the conclusion by some inference rule. Finally, each of the premises must be the conclusion of a derivation of its own.

A derivation is also called a *proof tree*; the root contains the conclusion, and each subtree is also a derivation. A leaf node represents the application of an inference rule that has no premises, like the LITERAL rule on page 18. Every time a call to the `eval` function terminates, that call corresponds to a leaf node in a derivation.

mkNL 33e
mkVL 33c

Unusually for computer scientists, we write derivation trees with the root at the bottom. Here is part of a derivation tree for evaluating the sum of two squares,

```
(+ (* x x) (* y y))
```

in an environment where $\rho$ binds $x$ to 3 and $y$ to 4.

$$\textsc{ApplyAdd}\ \dfrac{\textsc{ApplyMul}\ \dfrac{\textsc{FormalVar}\ \dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \textsc{var}(\mathtt{x}), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle} \qquad \dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \textsc{var}(\mathtt{x}), \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \textsc{FormalVar}}{\langle \textsc{apply}(*, \textsc{var}(\mathtt{x}), \textsc{var}(\mathtt{x})), \xi, \phi, \rho\rangle \Downarrow \langle 9, \xi, \phi, \rho\rangle} \quad \cdots}{\langle \textsc{apply}(+, \textsc{apply}(*, \textsc{var}(\mathtt{x}), \textsc{var}(\mathtt{x})), \textsc{apply}(*, \textsc{var}(\mathtt{y}), \textsc{var}(\mathtt{y}))), \xi, \phi, \rho\rangle \Downarrow \langle 25, \xi, \phi, \rho\rangle}$$

Each node in the tree is labelled with the name of the rule to which it corresponds. Because derivation trees take so much space, we've left out the subtree that proves

$$\langle \textsc{apply}(*, \textsc{var}(\mathtt{y}), \textsc{var}(\mathtt{y})), \xi, \phi, \rho\rangle \Downarrow \langle 16, \xi, \phi, \rho\rangle.$$

To save space, we can take some liberties with the notation. In particular, instead of writing abstract syntax like $\textsc{apply}(*, \textsc{var}(\mathtt{y}), \textsc{var}(\mathtt{y}))$, we can instead write the concrete syntax `(* y y)`. The resulting derivation is easier to digest, but it is harder to correlate with the rules of the operational semantics:

$$\textsc{ApplyAdd}\ \dfrac{\textsc{ApplyMul}\ \dfrac{\textsc{FormalVar}\ \dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \mathtt{x}, \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle} \qquad \dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \mathtt{x}, \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}\ \textsc{FormalVar}}{\langle \texttt{(* x x)}, \xi, \phi, \rho\rangle \Downarrow \langle 9, \xi, \phi, \rho\rangle} \quad \cdots}{\langle \texttt{(+ (* x x) (* y y))}, \xi, \phi, \rho\rangle \Downarrow \langle 25, \xi, \phi, \rho\rangle}$$

If we don't label the nodes, we can squeeze in the full derivation tree:

$$\dfrac{\dfrac{\dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \mathtt{x}, \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle} \qquad \dfrac{\mathtt{x} \in \operatorname{dom}\rho \qquad \rho(\mathtt{x}) = 3}{\langle \mathtt{x}, \xi, \phi, \rho\rangle \Downarrow \langle 3, \xi, \phi, \rho\rangle}}{\langle \texttt{(* x x)}, \xi, \phi, \rho\rangle \Downarrow \langle 9, \xi, \phi, \rho\rangle} \qquad \dfrac{\dfrac{\mathtt{y} \in \operatorname{dom}\rho \qquad \rho(\mathtt{y}) = 4}{\langle \mathtt{y}, \xi, \phi, \rho\rangle \Downarrow \langle 4, \xi, \phi, \rho\rangle} \qquad \dfrac{\mathtt{y} \in \operatorname{dom}\rho \qquad \rho(\mathtt{y}) = 4}{\langle \mathtt{y}, \xi, \phi, \rho\rangle \Downarrow \langle 4, \xi, \phi, \rho\rangle}}{\langle \texttt{(* y y)}, \xi, \phi, \rho\rangle \Downarrow \langle 16, \xi, \phi, \rho\rangle}}{\langle \texttt{(+ (* x x) (* y y))}, \xi, \phi, \rho\rangle \Downarrow \langle 25, \xi, \phi, \rho\rangle}$$

If we know something about the environments, we can use derivation trees to answer questions about the evaluation of expressions and definitions. One example is the evaluation of the expression in Exercise 10. This kind of application of the language semantics is called the *theory* of the language. But we can answer much more interesting questions if we *prove facts about derivations*. For example, Exercise 11 asks you to show that in Impcore, the expresssion `(if x x 0)` is always equivalant to `x`. Reasoning about derivations is called *metatheory*.

### 2.6.2 Proofs about derivations: Metatheory

If you already know that you want to study metatheory, you're better off with another book. In this book, our goal is to give you an idea of what kinds of things metatheory is good for, so that when you decide if you want to study metatheory, you'll know a little bit about it.

Here's a claim about Impcore programs: in any program, we can replace the expression `(+ x 0)` by the expression `x`, and this replacement doesn't change the meaning of the program. This claim is not very interesting to a programmer, but it's important to a compiler writer. It's exactly this sort of claim that compiler writers use to create "optimizations" that improve the performance of programs. And if you're going to write an optimization, you must be utterly certain that the optimization is correct. Providing certainty is where proofs—and proofs about proofs—are useful.

What do we know about derivations that refer to the expression `(+ x 0)`? We know that if there is a derivation at all, the derivation is going to contain a judgment of the form

$$\langle (+\ x\ 0), \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle.$$

Let's consider just the sub-derivation rooted in that judgment. What do we know about it? We know that it must apply a rule that permits the application of the + operator on the left-hand side of its conclusion. *This kind of reasoning is exactly the kind of reasoning used to write the interpreter code in chunk 36a.* In both cases, we look for inference rules with APPLY(+, ...) in their conclusions.

If we consult Section 2.4.6, we see there are two such rules: APPLYUSER and APPLYADD. And now we see that the claim is false! If $\phi$(+) refers to a user-defined function, the APPLYUSER rule kicks in, and it is *not* safe to replace `(+ x 0)` with `x`. Here's a demonstration:

51 ⟨*terrifying transcript* 51⟩≡

```
-> (define + (x y) y)
-> (define addzero (x) (+ x 0))
-> (addzero 99)
0
-> (define addzero2 (x) x)
-> (addzero2 99)
99
```

If a compiler writer wants to be able to replace an instance of `(+ x 0)` with `x`, he or she will first have to prove that the environment $\phi$ in which `(+ x 0)` is evaluated *never* binds + to a user-defined function. (Compilers typically include lots of infrastructure for proving facts about environments, but such infrastructure is well beyond the scope of this book.)

Section 2.8 includes several exercises that you can solve using metatheory:

- In Exercises 11 and 12 you can investigate under what circumstances `(if x x 0)` is equivalent to `x`.
- In Exercise 15 you can prove that evaluting an expression can't create a new variable.
- In Exercise 18 you can prove that in Impcore, evaluating an expression always produces the same result. (In languages that support parallel execution, this property usually doesn't hold.)
- In Exercise 20 you can prove that Impcore programs can be evaluated using a stack of mutable environments, as the implementation in Section 2.5 does.

A little metatheory goes a long way.

## 2.7 Further reading

Knuth (1984) describes literate programming; Ramsey (1994) describes Noweb.

The term "initial basis" is taken from ML, where it refers to a collection of environments binding not only values but types, "signatures," and other entities.

Plotkin (1981) started the modern movement toward operational semantics. This paper describes a form of operational semantics that is better suited to proving properties of programs; our form is better suited to specifying evaluators.

Ken Thompson was the first we know of to create an extensible printer for C programs; his implementation first appeared in Ninth Edition Research Unix. Hanson (1996) describes another implementation.

## 2.8 Exercises

### Learning about language design

1. Pick your favorite programming language and identify the syntactic categories. (If you're not sure exactly what a syntactic category is, you're not alone: see Exercise 2.)

   - What syntactic categories are there?
   - How are the different syntactic categories related? In particular, when can a phrase in one syntactic category be a direct part of a phrase in another (or the same) syntactic category?
   - Do all the phrases in each individual syntactic category share a similar job? What is that job?
   - Which syntactic categories do you have to know about to understand how the language is structured? Which categories are details that apply only to one corner of the language, and aren't important for overall understanding?

   In Impcore, we might answer these questions as follows:

   - The syntactic categories of Impcore are definitions and expressions.
   - A definition may contain an expression, and an expression may contain another expression, but an expression may not contain a definition.
   - In Impcore, the job of an expression is to be evaluated to produce a value. The job of a definition is to introduce a new name into an environment.

### The difficulties of theory

2. As far as we are aware, the term *syntactic category* does not have a precise, technical definition. One interpretation is to say that a syntactic category is a nonterminal in a grammar for which more than one kind of phrase can be produced. If we look at the grammar for Impcore on page 8, we see that this interpretation allows *def* and *exp*, but it rules out *function* and *primitive* on the grounds that even though *function* and *primitives* are nonterminals that have multiple productions, each production of *function* or *primitive* always produces exactly the same kind of phrase: a single name.

   But if you have studied formal language theory, you will know that any language as complex as Impcore can be described by a plethora of different grammars. We would prefer, if possible, that syntactic categories be properties of a *language*, not of the *grammar* used to describe the language.

   In this Exercise, please develop a precise, formal definition of "syntactic category."

   - One starting point might be a substitution principle: if two phrases are in the same syntactic category, then in any grammatical program, if you interchange one phrase for the other, the new program is still grammatical. For example, in a C program, replacing `x - 1` with `*p++` cannot introduce a syntax error. If there are maximal sets of interchangeable phrases, these might be related to syntactic categories.
   - Another possible approach is to define syntactic categories not in terms of *concrete* syntax but instead to use *abstract* syntax, as in Section 2.2. A study of well-formed trees, or of interchangeable trees, might lead to a satisfying definition.

In one sentence, here is the problem: develop a definition of the vague term "syntactic category" that you find precise, satisfying, and meaningful independent of how a language's concrete syntax is expressed.

## Learning about the language

Exercises 3–8 specify some number-theoretic functions for you to program. If you can, use recursion; it is good practice for Chapter 3:

3. Define a function `sigma` satisfying (`sigma` $m$ $n$)$= m+(m+1)+\cdots+n$. When $m > n$, the behavior of `sigma` is unspecified; your implementation may do anything you like.

4. Define functions `exp` and `log` which (`exp` $m$ $n$)$= m^n$ $(m, n \geq 0)$, and (`log` $m$ $n$) is the least integer $l$ such that $m^{l+1} > n$. The behavior of (`log` $m$ $n$) is specified only when $m > 1$ and $n > 0$; on other inputs, your your implementation may do anything you like.

5. Define a function `choose` such that (`choose` $n$ $k$) is the number of ways of selecting $k$ items from a collection of $n$ items, without repetitions, where $n$ and $k$ are nonnegative integers. This quantity is called a *binomial coefficient*, and it is notated $\binom{n}{k}$. It can be defined as $\frac{n!}{k!(n-k)!}$, but the following identities are more helpful computationally:

$$\begin{array}{ll} \binom{n}{0} = 1 & \text{when } n \geq 0 \\ \binom{n}{n} = 1 & \text{when } n \geq 0 \\ \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1} & \text{when } n > 0 \text{ and } k > 0 \end{array}$$

6. Define a function `fib` such that (`fib` $m$) is the $m$th Fibonacci number. The Fibonacci numbers are defined by identities: (`fib 0`) $= 0$, (`fib 1`) $= 1$, and for $m > 1$, (`fib` $m$) $=$ (`fib (-` $m$ `1))` $+$ (`fib (-` $m$ `2))`.

7. Define functions `prime?`, `nthprime`, `sumprimes`, and `relprime?` meeting these specifications: (`prime?` $n$) is true (1) if $n$ is prime, false (0) otherwise. (`nthprime` $n$) is the $n$th prime number. (Consider that 2 is the first prime number, so (`nthprime 1`) $= 2$, so (`nthprime 2`) $= 3$, and so on.) (`sumprimes` $n$) is the sum of the first $n$ primes. (`relprime?` $m$ $n$) is true if $m$ and $n$ are relatively prime (have no common divisors except 1), false otherwise.

8. Define a function `binary` such that (`binary` $m$) is the number whose decimal representation is the binary representation of $m$. For example (`binary 12`) $= 1100$, since $1100_2 = 12_{10}$. An ideal implementation of `binary` will work on any integer input, including negative ones. For example, (`binary -5`) $= -101$.

9. The scope rules of Impcore are identical to that of C. Consider this C program:

55 ⟨*mystery.c* 55⟩≡

```
int x;

void R(int y) {
    x = y;
}

void Q(int x) {
    R(x + 1);
    printf("%d\n", x);
}

void main(void) {
    x = 2;
    Q(4);
    printf("%d\n", x);
}
```

What is its output? Give the analogous program in Impcore, where a "program" is a sequence of definitions.

## Learning about the semantics

10. Use the operational semantics to prove that if you evaluate `(begin (set x 3) x)` in an environment where $\rho(\mathtt{x}) = 99$, then the result of the evaluation is 3. In your proof, use a formal derivation tree like the example on page 50.

11. Use the operational semantics to show that if there exist environments $\xi$, $\phi$, and $\rho$ such that

$$\langle \text{IF}(\text{VAR}(\mathtt{x}), \text{VAR}(\mathtt{x}), \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$$

and

$$\langle \text{VAR}(\mathtt{x}), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle$$

then $v_1 = v_2$.

12. Use the operational semantics to show that there exist environments $\xi$, $\phi$, $\rho$, $\xi'$, and $\rho'$ and a value $v_1$ such that

$$\langle \text{IF}(\text{VAR}(\mathtt{x}), \text{VAR}(\mathtt{x}), \text{LITERAL}(0)), \xi, \phi, \rho \rangle \Downarrow \langle v_1, \xi', \phi, \rho' \rangle$$

if *and only if* there exist environments $\xi$, $\phi$, $\rho$, $\xi''$, and $\rho''$ and a value $v_2$ such that

$$\langle \text{VAR}(\mathtt{x}), \xi, \phi, \rho \rangle \Downarrow \langle v_2, \xi'', \phi, \rho'' \rangle.$$

Give necessary and sufficient conditions on the environments $\xi$, $\phi$, and $\rho$ such that both expressions evaluate successfully.

13. In Impcore, it is an error to refer to a variable that is not bound in any environment. In Awk, the first use of such a variable (either for its value or as the target of an assignment) implicitly creates a new global variable with value 0. In Icon, the rule is similar, except the implicitly created variable is a local variable, whose scope is the entire procedure in which the assignment appears.

    (a) Change the rules of Impcore as needed, and add as many new rules as needed, to give Impcore Awk-like semantics for unbound variables.

    (b) Change the rules of Impcore as needed, and add as many new rules as needed, to give Impcore Icon-like semantics for unbound variables.[9]

    (c) Which of the two changes do you prefer, and why?

14. Write a program in Impcore that causes an error when run by the Impcore interpreter, prints 1 when run with the Awk-like extension from problem 13, and prints 0 when run with the Icon-like extension from problem 13. In Impcore, a "program" is simply a sequence of definitions.

15. Prove that the execution of an Impcore expression does not change the set of variables bound in the environment. That is, prove that if $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$, then $\operatorname{dom} \xi = \operatorname{dom} \xi'$.

16. Is is true or false that evaluating an expression without a SET node does not change any environment? Use metatheory to justify your answer.

17. Give operational semantics for a C-like FOR$(e_1, e_2, e_3, e_4)$. Like `while`, the value it returns is unimportant, since a `for` expression is evaluated for its side effects; you may choose any final value you find convenient.

18. Prove that Impcore is deterministic. That is, for any $e$ and any environments, there is at most one $v$ such that $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$.

19. Suppose you want extend Impcore by adding a FLIP expression, which models the flip of a coin by returning either 0 or 1 with equal probability. What rules would you add to the operational semantics for FLIP? Would the resulting language still be deterministic?

20. The operational semantics never mutates an environment; when a change in environment is needed, the semantics always creates a new environment that is equal to an existing environment with one or more new bindings. But our code uses mutable environments—and for formal parameters $\rho$, our code uses a *stack* of mutable environments. (The stack is implicit in the C call stack.) Prove that using a stack of mutable environments is safe. (In this question, we don't consider the environment of global variables $\xi$.)

    Although proving properties of an implementation is a best practice in programming languages, it is also a bit tricky, because you save the most work by proving properties of an implementation that you have not yet built. Here we're asking you to *imagine* an implementation that uses an *explicit*, global stack of environments.

[9] Impcore has top-level expressions whereas Icon does not. For purposes of this problem, assume that every top-level expression is evaluated in its own, anonymous procedure.

We suggest that you reimagine procedure `eval`—the implementation of the evaluation judgment $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$—as one that takes $\rho$ off of the stack, does a bunch of computation, and just before it terminates, pushes $\rho'$ onto the stack. The "bunch of computation" in the middle may include pushes, pops, and recursive calls to `eval`.

Prove that the following properties hold.

(a) In `eval`, the implementation of every proof rule that ends in the judgment form $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$ can be changed to pop $\rho$ off the stack and push $\rho'$ onto the stack. (It is possible that $\rho' = \rho$.)

(b) If $\rho' = \rho$, then the only copy of $\rho$ is the one on top of the stack. If $\rho' \neq \rho$, then once $\rho$ is popped off the stack, it is thrown away and never used again. In particular, no environment ever needs to be copied anywhere except on the stack; that is, the stack holds all the environments that will ever be used in any future evaluation.

From this lemma, it is an easy step to say that the operation "pop $\rho$; push $\rho'$" can be replaced by the operation "mutate $\rho$ in place to become $\rho'$." In particular, it is safe to implement $\rho\{x \mapsto v\}$ by mutating an existing binding, rather than by constructing a new binding. Part (b) of the lemma is crucial; it is safe to mutate $\rho$ *only* because the sole copy is on top of the stack.

Your proof of the lemma should be by induction on the height of a derivation of $\langle e, \xi, \phi, \rho \rangle \Downarrow \langle v, \xi', \phi, \rho' \rangle$. The base cases are the rules that have no evaluation judgments in the premises, such as the LITERAL or FORMALVAR rules. The induction steps are the rules that *do* have evaluation judgments as premises, such as FORMALASSIGN.

This theorem is the justification for our implementation of `bindval`, as referred to in Section 2.5.5.

21. In some programming languages, like C, function parameters may be evaluated in any order, and the choice of order is up to the implementation. How would you write this specification in a formal semantics? What if the order is unspecified, but must be the same each time? What if the order can change each time?

## Learning about the interpreter

22. Add the primitive `read` to the Impcore interpreter and the initial basis. `Read` is a function of no arguments, which reads a number from standard input and returns it.

23. Implement FOR, as describe in Exercise 17.

24. Write an Impcore program that takes a long time to execute. Profile the interpreter.

    (a) Approximately what fraction of time is spent in linear search? Approximately how much faster might the interpreter run if you used search trees? What about hash tables?

    (b) Download code from Hanson 1996[10] and use it to implement names and environments. How much speedup do you actually get?

    (c) What other "hot spots" can you find? What is the best way to make the interpreter run faster?

    As in other programs an Impcore "program" is simply a sequence of definitions.

25. Change the Impcore interpreter to pass parameters by *reference* instead of by value. For example, if a variable `x` is passed to a function `f`, function `f` can modify `x` by assigning to a formal parameter. If non-variable expression is passed as an argument to a function, assignments to formal parameters should have no effect outside the function. (In particular, it should not be possible to change the value of an integer literal by assignment to a formal parameter.)

    To implement this change, change the return type of `eval` and `fetchval` to be `Value*`, and make `Valuelists` hold `Value*`s rather than `Values`. Type checking in your C compiler should help you find the other parts that need to change. No change in syntax is needed.

    (a) Is the `bindval` function in the environment interface still necessary?

    (b) Write a function that makes use of call by reference (e.g., reference parameters can be used to return multiple values from a function).

    (c) How does call by reference affect the truth of the assertion (page 11): "no assignment to a formal parameter can ever change the value of a global variable."?

    (d) Discuss the advantages and disadvantages of reference parameters. How would this discussion change if Impcore had arrays?

26. Add local variables to function definitions. That is, change the concrete and abstract syntax of definitions to:

    `(define` *function-name* `(`*formals*`) (`*locals*`)` *expression*`)`
    `Userfun = (Namelist formals, Namelist locals, Exp body)`

    where *locals*, having the same syntax as *formals*, names those variables that are local to the function. If a local variable has the same name as a formal parameter, then in the body of the function, that name refers to the local variable. And when a user-defined function is applied, the local variables should be initialized to zero.

    The affected code is mainly in `parse.c` (section A.2.7) and `eval.c`. Make sure not to lose the check that the number of actual parameters is correct.

27. Implement your solutions to Exercise 13. Use your implementation to test your solution to Exercise 14.

[10] See URL `http://www.cs.princeton.edu/software/cii`.

28. Add floating-point numbers to the language. Let `Value` be a sum type containing either an integer or a floating-point number:

```
Value = INT   (int)
      | FLOAT (float)
```

The arithmetic primitives should apply integer or floating-point operations, depending on the types of their arguments; given arguments of mixed types, these functions should promote integers to floats. Add primitives `trunc` and `round` to convert floating-point values to integers; `trunc` should round toward minus infinity and `round` should round to nearest (to even if halfway between two integers). Support floating-point literals using `strtod`.

29. The implementation of `use` in chunk ⟨*evaluate* `d->u.use`, *possibly mutating* `globals` *and* `functions` 44d⟩ leaks open file descriptors when files have bugs. Explain how you would fix the problem.