Programming Languages: Application and Interpretation

Shriram Krishnamurthi
Brown University

This book is available free-of-cost from the author's Web site.
This version was generated on 2007-04-26.

Preface

The book is the textbook for the programming languages course at Brown University, which is taken primarily by third and fourth year undergraduates and beginning graduate (both MS and PhD) students. It seems very accessible to smart second year students too, and indeed those are some of my most successful students. The book has been used at over a dozen other universities as a primary or secondary text. The book's material is worth one undergraduate course worth of credit.

This book is the fruit of a vision for teaching programming languages by integrating the "two cultures" that have evolved in its pedagogy. One culture is based on interpreters, while the other emphasizes a survey of languages. Each approach has significant advantages but also huge drawbacks. The interpreter method writes *programs* to learn concepts, and has its heart the fundamental belief that by teaching the computer to execute a concept we more thoroughly learn it ourselves.

While this reasoning is internally consistent, it fails to recognize that understanding definitions does not imply we understand consequences of those definitions. For instance, the difference between strict and lazy evaluation, or between static and dynamic scope, is only a few lines of interpreter code, but the *consequences* of these choices is enormous. The survey of languages school is better suited to understand these consequences.

The text therefore melds these two approaches. Concretely, students program with a new set of features first, then try to distill those principles into an actual interpreter. This has the following benefits:

- By seeing the feature in the context of a real language, students can build something interesting with it first, so they understand that it isn't an entirely theoretical construct, and will actually *care* to build an interpreter for it. (Relatively few students are excited in interpreters for their own sake, and we have an obligation to appeal to the remainder too.)

- Students get at least fleeting exposure to multiple languages, which is an important educational attribute that is being crushed by the wide adoption of industrially fashionable languages. (Better still, by experimenting widely, they may come to appreciate that industrial fashions are just that, not the last word in technological progress.)

- Because they have already programmed with the feature, the explanations and discussions are much more interesting than when all students have seen is an abstract model.

- By first building a mental model for the feature through experience, students have a much better chance of actually discovering how the interpreter is supposed to work.

In short, many more humans learn by induction than by deduction, so a pedagogy that supports it is much more likely to succeed than one that suppresses it. The book currently reflects this design, though the survey parts are done better in lecture than in the book.

Separate from this vision is a goal. My goal is to not only teach students new material, but to also change the way they solve problems. I want to show students where languages come from, why we should regard languages as the ultimate form of abstraction, how to recognize such an evolving abstraction, and how to turn what they recognize into a language. The last section of the book, on domain-specific languages, is a growing step in this direction.

Design Principles

- Concepts like design, elegance and artistic sensibility are rarely manifest in computer science courses; in the name of not being judgmental, we may be running the risk of depriving our students of judgment itself. We should reverse this trend. Students must understand that artificial objects have their own aesthetic; the student must learn to debate the tradeoffs that lead to an aesthetic. Programming languages are some of the most thoroughly designed artifacts in computer science. Therefore, the study of programming languages offers a microcosm to study design itself.

- The best means we have to lead students to knowledge is through questions, not answers. The best education prepares them to assess new data by confronting it with questions, processing the responses, and iterating until they have formed a mental model of it. This book is therefore structured more like a discussion than a presentation. It leads the reader down wrong paths (so don't blindly copy code from it!). It allows readers to get comfortable with mistaken assumptions before breaking them down systematically.

- The programming languages course is one of the few places in the curriculum where we can tease out and correct our students' misconceptions about this material. They are often misled on topics such as efficiency and correctness. Therefore, material on compilation, type systems and memory management should directly confront their biases. For instance, a presentation of garbage collection that does not also discuss the trade-offs with manual memory management will fail to address the prejudices students bear.

Background and Prerequisite

This book assumes that students are comfortable reasoning informally about loop invariants, have modest mathematical maturity, and are familiar with the existence of the Halting Problem. At Brown, they have all been exposed to Java but not necessarily to any other languages (such as Scheme).

Supplementary Material

There is some material I use in my course that isn't (currently) in this book:

preparation in Scheme For the first week, I offer supplementary sessions that teach students Scheme. The material from these sessions is available from my course Web pages. In addition, I recommend the

use of a simple introduction to Scheme, such as the early sections of *The Little Schemer* or of *How to Design Programs*.

domain-specific languages I discuss instances of real-world domain-specific languages, such as the access-control language XACML. Students find the concepts easy to grasp, and can see why the language is significant. In addition, it is one they may themselves encounter (or even decide to use) in their programming tasks.

garbage collection I have provided only limited notes on garbage collection because I feel no need to offer my own alternative to Paul Wilson's classic survey, *Uniprocessor Garbage Collection Techniques*. I recommend choosing sections from this survey, depending on student maturity, as a supplement to this text.

model checking I supplement the discussion of types with a presentation on model checking, to show students that it is possible to go past the fixed set of theorems of traditional type systems to systems that permit developers to state theorems of interest. I have a pre-prepared talk on this topic, and would be happy to share those slides.

Web programming Before plunging into continuations, I discuss Web programming APIs and demonstrate how they mask important control operators. I have a pre-prepared talk on this topic, and would be happy to share those slides. I also wrap up the section on continuations with a presentation on programming in the PLT Scheme Web server, which natively supports continuations.

articles on design I hand out a variety of articles on the topic of design. I've found Dan Ingalls's dissection of Smalltalk, Richard Gabriel's on Lisp, and Paul Graham's on both programming and design the most useful. Graham has now collected his essays in the book *Hackers and Painters*.

logic programming The notes on logic programming are the least complete. Students are already familiar with unification from type inference by the time I arrive at logic programming. Therefore, I focus on the implementation of backtracking. I devote one lecture to the use of unification, the implications of the occurs-check, depth-first versus breadth-first search, and tabling. In another lecture, I present the implementation of backtracking through continuations. Concretely, I use the presentation in Dorai Sitaram's *Teach Yourself Scheme in Fixnum Days*. This presentation consolidates two prior topics, continuations and macros.

Exercises

Numerous exercises are sprinkled throughout the book. Several more, in the form of homework assignments and exams, are available from my course's Web pages (where ⟨*year*⟩ is one of `2000`, `2001`, `2002`, `2003`, `2004` and `2005`):

`http://www.cs.brown.edu/courses/cs173/`⟨*year*⟩`/`

In particular, in the book I do *not* implement garbage collectors and type checkers. These are instead homework assignments, ones that students generally find extremely valuable (and very challenging!).

Programs

This book asks students to implement language features using a combination of interpreters and little compilers. All the programming is done in Scheme, which has the added benefit of making students fairly comfortable in a language and paradigm they may not have employed before. End-of-semester surveys reveal that students are far more likely to consider using Scheme for projects in other courses after taking this course than they were before it (even when they had prior exposure to Scheme).

Though every line of code in this book has been tested and is executable, I purposely do not distribute the code associated with this book. While executable code greatly enhances the study of programming languages, it can also detract if students execute the code mindlessly. I therefore ask you, Dear Reader, to please type in this code as if you were writing it, paying close attention to every line. You may be surprised by how much many of them have to say.

Course Schedule

The course follows approximately the following schedule:

Weeks	*Topics*
1	Introduction, Scheme tutorials, Modeling Languages
2–3	Substitution and Functions
3	Laziness
4	Recursion
4	Representation Choices
4–5	State
5–7	Continuations
7–8	Memory Management
8–10	Semantics and Types
11	Programming by Searching
11–12	Domain-Specific Languages and Metaprogramming

Miscellaneous “culture lecture” topics such as model checking, extensibility and future directions consume another week.

An Invitation

I think the material in these pages is some of the most beautiful in all of human knowledge, and I hope any poverty of presentation here doesn’t detract from it. Enjoy!

Acknowledgments

This book has a long and humbling provenance. The conceptual foundation for this interpreter-based approach traces back to seminal work by John McCarthy. My own introduction to it was through two texts I read as an undergraduate, the first editions of *The Structure and Interpretation of Computer Programs* by Abelson and Sussman with Sussman and *Essentials of Programming Languages* by Friedman, Wand and Haynes. Please read those magnificent books even if you never read this one.

My graduate teaching assistants, Dave Tucker and Rob Hunter, wrote and helped edit lecture notes that helped preserve continuity through iterations of my course. Greg Cooper has greatly influenced my thinking on lazy evaluation. Six generations of students at Brown have endured drafts of this book.

Bruce Duba, Corky Cartwright, Andrew Wright, Cormac Flanagan, Matthew Flatt and Robby Findler have all significantly improved my understanding of this material. Matthew and Robby's work on DrScheme has greatly enriched my course's pedagogy. Christian Queinnec and Paul Graunke inspired the presentation of continuations through Web programming, and Greg Cooper created the approach to garbage collection, both of which are an infinite improvement over prior approaches.

Alan Zaring, John Lacey and Kathi Fisler recognized that I might like this material and introduced me to it (over a decade ago) before it was distributed through regular channels. Dan Friedman generously gave of his time as I navigated *Essentials*. Eli Barzilay, John Clements, Robby Findler, John Fiskio-Lasseter, Kathi Fisler, Cormac Flanagan, Matthew Flatt, Suresh Jagannathan, Gregor Kiczales, Mira Mezini, Prabhakar Ragde, Marc Smith, and Éric Tanter have provided valuable feedback after using prior versions of this text.

The book's accompanying software has benefited from support by several generations of graduate assistants, especially Greg Cooper and Guillaume Marceau. Eli Barzilay and Matthew Flatt have also made excellent, creative contributions to it.

My chairs at Brown, Tom Dean and Eli Upfal, have permitted me to keep teaching my course so I could develop this book. I can't imagine how many course staffing nightmares they've endured, and ensuing temptations they've suppressed, in the process.

My greatest debt is to Matthias Felleisen. An early version of this text grew out of my transcript of his course at Rice University. That experience made me realize that even the perfection embodied in the books I admired could be improved upon. This result is not more perfect, simply different—its outlook shaped by standing on the shoulders of giants.

Thanks

Several more people have made suggestions, asked questions, and identified errors:

Ian Barland, Hrvoje Blazevic, Daniel Brown, Greg Buchholz, Lee Butterman, Richard Cobbe, Bruce Duba, Stéphane Ducasse, Marco Ferrante, Dan Friedman, Mike Gennert, Arjun Guha, Roberto Ierusalimschy, Steven Jenkins, Eric Koskinen, Neel Krishnaswami, Benjamin Landon, Usman Latif, Dan Licata, Alice Liu, Paulo Matos, Grant Miner, Ravi Mohan, Jason Orendorff, Klaus Ostermann, Pupeno [sic], Manos Renieris, Morten Rhiger, Bill Richter, Peter Rosenbeck, Amr Sabry, Francisco Solsona, Anton van Straaten, Andre van Tonder, Michael Tschantz, Phil Wadler, Joel Weinberger, and Greg Woodhouse.

In addition, Michael Greenberg instructed me in the rudiments of classifying flora.

Contents

Part I

Prelude

Chapter 1

Modeling Languages

A student of programming languages who tries to study a new language can be overwhelmed by details. Virtually every language consists of

- a peculiar syntax,
- some behavior associated with each syntax,
- numerous useful libraries, and
- a collection of idioms that programmers of that language use.

All four of these attributes are important to a programmer who wants to adopt a language. To a scholar, however, one of these is profoundly significant, while the other three are of lesser importance.

The first insignificant attribute is the syntax. Syntaxes are highly sensitive topics,[1] but in the end, they don't tell us very much about a program's *behavior*. For instance, consider the following four fragments:

1. `a [25]`
2. `(vector-ref a 25)`
3. `a [25]`
4. `a [25]`

Which two are most alike? The first and second, obviously! Why? Because the first is in Java and the second is in Scheme, both of which signal an error if the vector associated with `a` has fewer than 25 entries; the third, in C, blithely ignores the vector's size, leading to unspecified behavior, even though its syntax is exactly the same as that of the Java code. The fourth, in ML or Haskell, is an application of `a` to the list containing just one element, `25`: that is, it's not an array dereference at all, it's a function appliction!

That said, syntax does matter, at least inasmuch as its brevity can help programmers express and understand more by saying less. For the purpose of our study, however, syntax will typically be a distraction, and

[1] Matthias Felleisen: "Syntax is the Viet Nam of programming languages."

will often get in the way of our understanding deeper similarities (as in the Java-Scheme-C example above). We will therefore use a uniform syntax for all the languages we implement.

The size of a language's library, while perhaps the most important characteristic to a programmer who wants to accomplish a task, is usually a distraction when studying a language. This is a slightly tricky contention, because the line between the core of a language and its library is fairly porous. Indeed, what one language considers an intrinsic primitive, another may regard as a potentially superfluous library operation. With experience, we can learn to distinguish between what must belong in the core and what need not. It is even possible to make this distinction quite rigorously using mathematics. Our supplementary materials will include literature on this distinction.

Finally, the idioms of a language are useful as a sociological exercise ("How do the natives of this linguistic terrain cook up a Web script?"), but it's dangerous to glean too much from them. Idioms are fundamentally human, and therefore bear all the perils of faulty, incomplete and sometimes even outlandish human understanding. If a community of Java programmers has never seen a particular programming technique—for instance, the principled use of objects as callbacks—they are likely to invent an idiom to take its place, but it will almost certainly be weaker, less robust, and less informative to use the idiom than to just use callbacks. In this case, and indeed in general, the idiom sometimes tells us more about the programmers than it does about the language. Therefore, we should be careful to not read too much into one.

In this course, therefore, we will focus on the behavior associated with syntax, namely the *semantics* of programming languages. In popular culture, people like to say "It's just semantics!", which is a kind of put-down: it implies that their correspondent is quibbling over minor details of meaning in a jesuitical way. But communication is all about meaning: even if you and I use different words to mean the same thing, we understand one another; but if we use the same word to mean different things, great confusion results. In this study, therefore, we will wear the phrase "It's just semantics!" as a badge of honor, because semantics leads to discourse which (we hope) leads to civilization.

Just semantics. That's all there is.

1.1 Modeling Meaning

So we want to study semantics. But how? To study meaning, we need a language for describing meaning. Human language is, however, notoriously slippery, and as such is a poor means for communicating what are very precise concepts. But what else can we use?

Computer scientists use a variety of techniques for capturing the meaning of a program, all of which rely on the following premise: the most precise language we have is that of mathematics (and logic). Traditionally, three mathematical techniques have been especially popular: *denotational*, *operational* and *axiomatic* semantics. Each of these is a rich and fascinating field of study in its own right, but these techniques are either too cumbersome or too advanced for our use. (We will only briefly gloss over these topics, in section 23.) We will instead use a method that is a first cousin of operational semantics, which some people call *interpreter* semantics.

The idea behind an interpreter semantics is simple: to explain a language, write an interpreter for it. The act of writing an interpreter forces us to understand the language, just as the act of writing a mathematical description of it does. But when we're done writing, the mathematics only resides on paper, whereas we can run the interpreter to study its effect on sample programs. We might incrementally modify the interpreter

if it makes a mistake. When we finally have what we think is the correct representation of a language's meaning, we can then use the interpreter to explore what the language does on interesting programs. We can even convert an interpreter into a compiler, thus leading to an efficient implementation that arises directly from the language's definition.

A careful reader should, however, be either confused or enraged (or both). We're going to describe the meaning of a language through an interpreter, which is a program. That program is written in some language. How do we know what *that* language means? Without establishing that first, our interpreters would appear to be mere scrawls in an undefined notation. What have we gained?

This is an important philosophical point, but it's not one we're going to worry about much in practice. We won't for the practical reason that the language in which we write the interpreter is one that we understand quite well: it's succint and simple, so it won't be too hard to hold it all in our heads. (Observe that dictionaries face this same quandary, and negotiate it successsfully in much the same manner.) The superior, theoretical, reason is this: others have already worked out the mathematical semantics of this simple language. Therefore, we really are building on rock. With that, enough of these philosophical questions for now. We'll see a few other ones later in the course.

1.2 Modeling Syntax

I've argued briefly that it is both futile and dangerous to vest too much emotion in syntax. In a platonic world, we might say

> Irrespective of whether we write
>
> - $3+4$ (infix),
> - `3 4 +` (postfix), or
> - (+ 3 4) (parenthesized prefix),
>
> we always mean the idealized action of adding the idealized numbers (represented by) "3" and "4".

Indeed, each of these notations is in use by at least one programming language.

If we ignore syntactic details, the *essence* of the input is a tree with the addition operation at the root and two leaves, the left leaf representing the number 3 and the right leaf the number 4. With the right data definition, we can describe this in Scheme as the expression

```
(add (num 3) (num 4))
```

and similarly, the expression

- $(3-4)+7$ (infix),
- `3 4 - 7 +` (postfix), or
- (+ (- 3 4) 7) (parenthesized prefix)

would be represented as

```
(add (sub (num 3) (num 4))
     (num 7))
```

One data definition that supports these representations is the following:

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?)
       (rhs AE?)]
  [sub (lhs AE?)
       (rhs AE?)])
```

where AE stands for "Arithmetic Expression".

Exercise 1.2.1 *Why are the lhs and rhs sub-expressions of type* AE *rather than of type* num*? Provide sample expressions permitted by the former and rejected by the latter, and argue that our choice is reasonable.*

1.3 A Primer on Parsers

Our interpreter should consume terms of type AE, thereby avoiding the syntactic details of the source language. For the user, however, it becomes onerous to construct terms of this type. Ideally, there should be a program that translates terms in concrete syntax into values of this type. We call such a program a *parser*.

In more formal terms, a parser is a program that converts *concrete syntax* (what a user might type) into *abstract syntax*. The word *abstract* signifies that the output of the parser is idealized, thereby divorced from physical, or syntactic, representation.

As we've seen, there are many concrete syntaxes that we could use for arithmetic expressions. We're going to pick one particular, slightly peculiar notation. We will use a prefix parenthetical syntax that, for arithmetic, will look just like that of Scheme. With one twist: we'll use {braces} instead of (parentheses), so we can distinguish concrete syntax from Scheme just by looking at the delimiters. Here are three programs employing this concrete syntax:

1. `3`
2. `{+ 3 4}`
3. `{+ {- 3 4} 7}`

Our choice is, admittedly, fueled by the presence of a convenient primitive in Scheme—the primitive that explains why so many languages built atop Lisp and Scheme *look* so much like Lisp and Scheme (i.e., they're parenthetical), even if they have entirely different meanings. That primitive is called *read*.

Here's how *read* works. It consumes an input port (or, given none, examines the standard input port). If it sees a sequence of characters that obey the syntax of a number, it converts them into the corresponding number in Scheme and returns that number. That is, the input stream

```
1 7 2 9 <eof>
```

(the spaces are merely for effect, not part of the stream) would result in the Scheme number 1729. If the sequence of characters obeys the syntax of a symbol (sans the leading quote), *read* returns that symbol: so

```
c s 1 7 3 <eof>
```

(again, the spaces are only for effect) evaluates to the Scheme symbol 'cs173. Likewise for other primitive types. Finally, if the input is wrapped in a matched pair of parenthetical delimiters—either (parentheses), [brackets] or {braces}—*read* returns a list of Scheme values, each the result of invoking *read* recursively. Thus, for instance, *read* applied to the stream

```
(1 a)
```

returns (list 1 'a), to

```
{+ 3 4}
```

returns (list '+ 3 4), and to

```
{+ {- 3 4} 7}
```

returns (list '+ (list '- 3 4) 7).

The *read* primitive is a crown jewel of Lisp and Scheme. It reduces what are conventionally two quite elaborate phases, called *tokenizing* (or *scanning*) and *parsing*, into three different phases: *tokenizing*, *reading* and *parsing*. Furthermore, it provides a single primitive that does the first and second, so all that's left to do is the third. *read* returns a value known as an *s-expression*.

The parser needs to identify what kind of program it's examining, and convert it to the appropriate abstract syntax. To do this, it needs a clear specification of the concrete syntax of the language. We'll use *Backus-Naur Form* (BNF), named for two early programming language pioneers. A BNF description of rudimentary arithmetic looks like this:

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

The <AE> in the BNF is called a *non-terminal*, which means we can rewrite it as one of the things on the right-hand side. Read ::= as "can be rewritten as". Each | presents one more choice, called a *production*. Everything in a production that isn't enclosed in <···> is literal syntax. (To keep the description simple, we assume that there's a corresponding definition for <num>, but leave its precise definition to your imagination.) The <AE>s in the productions are references back to the <AE> non-terminal.

Notice the strong similarity between the BNF and the abstract syntax representation. In one stroke, the BNF captures *both* the concrete syntax (the brackets, the operators representing addition and subtraction) and a default abstract syntax. Indeed, the only thing that the actual abstract syntax data definition contains that's not in the BNF is names for the fields. Because BNF tells the story of concrete and abstract syntax so succintly, it has been used in definitions of languages ever since Algol 60, where it first saw use.

Assuming all programs fed to the parser are syntactically valid, the result of reading must be either a number, or a list whose first value is a symbol (specifically, either '+ or '-) and whose second and third values are sub-expressions that need further parsing. Thus, the entire parser looks like this:[2]

```
;; parse : sexp ⟶ AE
;; to convert s-expressions into AEs

(define (parse sexp)
  (cond
    [(number? sexp) (num sexp)]
    [(list? sexp)
     (case (first sexp)
       [(+) (add (parse (second sexp))
                 (parse (third sexp)))]
       [(-) (sub (parse (second sexp))
                 (parse (third sexp)))])]))
```

Here's the parser at work. The first line after each invocation of (*parse* (*read*)) is what the user types; the second line after it is the result of parsing. This is followed by the next prompt.

```
Language: PLAI - Advanced Student.
> (parse (read))
3
(num 3)
> (parse (read))
{+ 3 4}
(add (num 3) (num 4))
> (parse (read))
{+ {- 3 4} 7}
(add (sub (num 3) (num 4)) (num 7))
```

This, however, raises a practical problem: we must type programs in concrete syntax manually every time we want to test our programs, or we must pre-convert the concrete syntax to abstract syntax. The problem arises because *read* demands manual input each time it runs. We might be tempted to use an intermediary such as a file, but fortunately, Scheme provides a handy notation that lets us avoid this problem entirely: we can use the quote notation to simulate *read*. That is, we can write

```
Language: PLAI - Advanced Student.
> (parse '3)
(num 3)
> (parse '{+ 3 4})
(add (num 3) (num 4))
```

[2]This is a parser for the whole language, but it is not a *complete* parser, because it performs very little error reporting: if a user provides the program `{+ 1 2 3}`, which is not syntactically legal according to our BNF specification, the parser silently ignores the `3` instead of signaling an error. You must write more robust parsers than this one.

```
> (parse '{+ {- 3 4} 7})
(add (sub (num 3) (num 4)) (num 7))
```

This is the last parser we will write in this book. From now on, you will be responsible for creating a parser from the concrete syntax to the abstract syntax. Extending the parser we have seen is generally straightforward because of the nature of syntax we use, which means it would be worthwhile to understand the syntax better.

1.4 Primus Inter Parsers

Most languages do not use this form of parenthesized syntax. Writing parsers for languages that don't is much more complex; to learn more about that, study a typical text from a compilers course. Before we drop the matter of syntax entirely, however, let's discuss our choice—parenthetical syntax—in a little more depth.

I said above that *read* is a crown jewel of Lisp and Scheme. In fact, I think it's actually one of the great ideas of computer science. It serves as the cog that helps decompose a fundamentally difficult process—generalized parsing of the input stream—into two very simple processes: reading the input stream into an intermediate representation, and parsing that intermediate representation. Writing a reader is relatively simple: when you see a opening bracket, read recursively until you hit a closing bracket, and return everything you saw as a list. That's it. Writing a parser using this list representation, as we've seen above, is also a snap.

I call these kinds of syntaxes *bicameral*,[3] which is a term usually used to describe legislatures such as that of the USA. No issue becomes law without passing muster in both houses. The lower house establishes a preliminary bar for entry, but allows some rabble to pass through knowing that the wisdom of the upper house will prevent excesses. In turn, the upper house can focus on a smaller and more important set of problems. In a bicameral syntax, the reader is, in American terms, the House of Representatives: it rejects the input

```
{+ 1 2)
```

(mismatched delimiters) but permits both of

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

the first of which is legal, the second of which isn't in our arithmetic language. It's the parser's (Senate's) job to eliminate the latter, more refined form of invalid input.

Exercise 1.4.1 *Based on this discussion, examine* XML. *What do the terms* well-formed *and* valid *mean, and how do they differ? How do these requirements relate to bicameral syntaxes such as that of Scheme?*

[3] Two houses.

Part II

Rudimentary Interpreters

Chapter 2

Interpreting Arithmetic

Having established a handle on parsing, which addresses syntax, we now begin to study semantics. We will study a language with only numbers, addition and subtraction, and further assume both these operations are binary. This is indeed a very rudimentary exercise, but that's the point. By picking something you know well, we can focus on the mechanics. Once you have a feel for the mechanics, we can use the same methods to explore languages you have never seen before.

The interpreter has the following contract and purpose:

```
;; calc : AE ⟶ number
;; consumes an AE and computes the corresponding number
```

which leads to these test cases:

```
(test (calc (parse '3)) 3)
(test (calc (parse '{+ 3 4})) 7)
(test (calc (parse '{+ {- 3 4} 7})) 6)
```

(notice that the tests must be consistent with the contract and purpose statement!) and this template:

```
(define (calc an-ae)
  (type-case AE an-ae
    [num (n) ...]
    [add (l r) ... (calc l) ... (calc r) ...]
    [sub (l r) ... (calc l) ... (calc r) ...]))
```

In this instance, we can convert the template into a function easily enough:

```
(define (calc an-ae)
  (type-case AE an-ae
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))]))
```

Running the test suite helps validate our interpreter.

What we have seen is actually quite remarkable, though its full power may not yet be apparent. We have shown that a programming language with just the ability to represent structured data can represent one of the most interesting forms of data, namely programs themselves. That is, we have just written a program that consumes programs; perhaps we can even write programs that generate programs. The former is the foundation for an interpreter semantics, while the latter is the foundation for a compiler. This same idea—but with a much more primitive language, namely arithmetic, and a much poorer collection of data, namely just numbers—is at the heart of the proof of Gödel's Theorem.

Chapter 3

Substitution

Even in a simple arithmetic language, we sometimes encounter repeated expressions. For instance, the Newtonian formula for the gravitational force between two objects has a squared term in the denominator. We'd like to avoid redundant expressions: they are annoying to repeat, we might make a mistake while repeating them, and evaluating them wastes computational cycles.

The normal way to avoid redundancy is to introduce an *identifier*.[1] As its name suggests, an identifier names, or identifies, (the value of) an expression. We can then use its name in place of the larger computation. Identifiers may sound exotic, but you're used to them in every programming language you've used so far: they're called *variables*. We choose not to call them that because the term "variable" is semantically charged: it implies that the value associated with the identifier can change (*vary*). Since our language initially won't offer any way of changing the associated value, we use the more conservative term "identifier". For now, they are therefore just names for computed constants.

Let's first write a few sample programs that use identifiers, inventing notation as we go along:

```
{with {x {+ 5 5}} {+ x x}}
```

We will want this to evaluate to `20`. Here's a more elaborate example:

```
  {with {x {+ 5 5}}
    {with {y {- x 3}}
      {+ y y}}}                                  [+ operation]
= {with {x 10} {with {y {- x 3}} {+ y y}}}       [substitution]
= {with {x 10} {with {y {- 10 3}} {+ y y}}}      [descent]
= {with {y {- 10 3}} {+ y y}}                    [- operation]
= {with {y 7} {+ y y}}                           [substitution]
= {with {y 7} {+ 7 7}}                           [descent]
= {+ 7 7}                                        [+ operation]
= 14
```

[1] As the authors of *Concrete Mathematics* say: "Name and conquer".

En passant, notice that the act of reducing an expression to a value requires more than just substitution; indeed, it is an interleaving of substitution and calcuation steps. Furthermore, when we have completed substition we "descend" into the inner expression to continue calculating.

Now let's define the language more formally. To honor the addition of identifiers, we'll give our language a new name: WAE, short for "`with` with arithmetic expressions". Its BNF is:

```
<WAE> ::= <num>
        | {+ <WAE> <WAE>}
        | {- <WAE> <WAE>}
        | {with {<id> <WAE>} <WAE>}
        | <id>
```

Notice that we've had to add *two* rules to the BNF: one for associating values with identifiers and another for actually using the identifiers. The nonterminal `<id>` stands for some suitable syntax for identifiers (usually a sequence of alphanumeric characters).

To write programs that process WAE terms, we need a data definition to represent those terms. Most of WAE carries over unchanged from AE, but we must pick some concrete representation for identifiers. Fortunately, Scheme has a primitive type called the symbol, which serves this role admirably.[2] Therefore, the data definition is

(**define-type** WAE
  [num (*n number?*)]
  [add (*lhs WAE?*) (*rhs WAE?*)]
  [sub (*lhs WAE?*) (*rhs WAE?*)]
  [with (*name symbol?*) (*named-expr WAE?*) (*body WAE?*)]
  [id (*name symbol?*)])

We'll call the expression in the *named-expr* field the *named expression*, since `with` lets the name in the id field stand in place of that expression.

3.1 Defining Substitution

Without fanfare, we used substitution to explain how `with` functions. We were able to do this because substitution is not unique to `with`: we've studied it for years in algebra courses, because that's what happens when we pass arguments to functions. For instance, let $f(x,y) = x^3 + y^3$. Then

$$f(12,1) = 12^3 + 1^3 = 1728 + 1 = 1729$$

$$f(10,9) = 10^3 + 9^3 = 1000 + 729 = 1729$$ [3]

Nevertheless, it's a good idea to pin down this operation precisely.

[2] In many languages, a string is a suitable representation for an identifier. Scheme does have strings, but symbols have the salutary property that they can be compared for equality in constant time.

[3] What's the next smallest such number?

Let's make sure we understand what we're trying to define. We want a crisp description of the process of *substitution*, namely what happens when we replace an identifier (such as x or `x`) with a value (such as 12 or `5`) in an expression (such as $x^3 + y^3$ or `{+ x x}`).

Recall from the sequence of reductions above that substitution is a part of, but not the same as, calculating an answer for an expression that has identifiers. Looking back at the sequence of steps in the evaluation example above, some of them invoke substitution while the rest are calcuation as defined for AE. For now, we're first going to pin down substitution. Once we've done that, we'll revisit the related question of calculation. But it'll take us a few tries to get substitution right!

Definition 1 (Substitution) *To* substitute *identifier i in e with expression v, replace all identifiers in e that have the name i with the expression v.*

Beginning with the program

```
{with {x 5} {+ x x}}
```

we will use substitution to replace the identifier `x` with the expression it is bound to, `5`. The definition of substitution above certainly does the trick: after substitution, we get

```
{with {x 5} {+ 5 5}}
```

as we would want. Likewise, it correctly substitutes when there are no instances of the identifier:

```
{with {x 5} {+ 10 4}}
```

to

```
{with {x 5} {+ 10 4}}
```

(since there are no instances of `x` in the expression, no substitutions occur). Now consider:

```
{with {x 5} {+ x {with {x 3} 10}}}
```

The rules reduce this to

```
{with {x 5} {+ 5 {with {5 3} 10}}}
```

Huh? Our substitution rule converted a perfectly reasonable program (whose value is `15`) into one that isn't even *syntactically* legal, i.e., it would be rejected by a parser, because the program contains `5` where the BNF tells us to expect an identifier. We definitely don't want substitution to have such an effect! It's obvious that the substitution algorithm is too naïve. To state the problem with the algorithm precisely, though, we need to introduce a little terminology.

Definition 2 (Binding Instance) *A* binding instance *of an identifier is the instance of the identifier that gives it its value. In* WAE*, the* `<id>` *position of a* `with` *is the only binding instance.*

Definition 3 (Scope) *The* scope *of a binding instance is the region of program text in which instances of the identifier refer to the value bound by the binding instance.*

Definition 4 (Bound Instance) *An identifier is* bound *if it is contained within the scope of a binding instance of its name.*

Definition 5 (Free Instance) *An identifier not contained in the scope of any binding instance of its name is said to be* free.

With this terminology in hand, we can now state the problem with the first definition of substitution more precisely: it failed to distinguish between bound instances (which should be substituted) and binding instances (which should not). This leads to a refined notion of substitution.

Definition 6 (Substitution, take 2) *To substitute identifier i in e with expression v, replace all identifiers in e which are not binding instances that have the name i with the expression v.*

A quick check reveals that this doesn't affect the outcome of the examples that the previous definition substituted correctly. In addition, this definition of substitution reduces

```
{with {x 5} {+ x {with {x 3} 10}}}
```

to

```
{with {x 5} {+ 5 {with {x 3} 10}}}
```

Let's consider a closely related expression:

```
{with {x 5} {+ x {with {x 3} x}}}
```

Hopefully we can agree that the value of this program is `8` (the left `x` in the addition evaluates to `5`, the right `x` is given the value `3` by the inner `with`, so the sum is `8`). The refined substitution algorithm, however, converts this expression into

```
{with {x 5} {+ 5 {with {x 3} 5}}}
```

which, when evaluated, yields `10`.

What went wrong here? Our substitution algorithm respected binding instances, but not their scope. In the sample expression, the `with` introduces a new scope for the inner `x`. The scope of the outer `x` is *shadowed* or *masked* by the inner binding. Because substitution doesn't recognize this possibility, it incorrectly substitutes the inner `x`.

Definition 7 (Substitution, take 3) *To substitute identifier i in e with expression v, replace all non-binding identifiers in e having the name i with the expression v, unless the identifier is in a scope different from that introduced by i.*

While this rule avoids the faulty substitution we've discussed earlier, it has the following effect: after substitution, the expression

```
{with {x 5} {+ x {with {y 3} x}}}
```

whose value should be that of `{+ 5 5}`, or `10`, becomes

```
{with {x 5} {+ 5 {with {y 3} x}}}
```

The inner expression should result in an error, because `x` has no value. Once again, substitution has changed a correct program into an incorrect one!

Let's understand what went wrong. Why didn't we substitute the inner `x`? Substitution halts at the `with` because, by definition, every `with` introduces a new scope, which we said should delimit substitution. But this `with` contains an instance of `x`, which we very much want substituted! So which is it—substitute within nested scopes or not? Actually, the two examples above should reveal that our latest definition for substitution, which may have seemed sensible at first blush, is too draconian: it rules out substitution within *any* nested scopes.

Definition 8 (Substitution, take 4) *To substitute identifier i in e with expression v, replace all non-binding identifiers in e having the name i with the expression v, except within nested scopes of i.*

Finally, we have a version of substitution that works. A different, more succint way of phrasing this definition is

Definition 9 (Substitution, take 5) *To substitute identifier i in e with expression v, replace all free instances of i in e with v.*

Recall that we're still defining substitution, not evaluation. Substitution is just an algorithm defined over expressions, independent of any use in an evaluator. It's the calculator's job to invoke substitution as many times as necessary to reduce a program down to an answer. That is, substitution simply converts

```
{with {x 5} {+ x {with {y 3} x}}}
```

into

```
{with {x 5} {+ 5 {with {y 3} 5}}}
```

Reducing this to an actual value is the task of the rest of the calculator.

Phew! Just to be sure we understand this, let's express it in the form of a function.

```
;; subst : WAE symbol WAE → WAE
;; substitutes second argument with third argument in first argument,
;; as per the rules of substitution; the resulting expression contains
;; no free instances of the second argument

(define (subst expr sub-id val)
  (type-case WAE expr
```

```
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              expr
              (with bound-id
                    named-expr
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

3.2 Calculating with `with`

We've finally defined substitution, but we still haven't specified how we'll use it to reduce expressions to answers. To do this, we must modify our calculator. Specifically, we must add rules for our two new source language syntactic constructs: `with` and identifiers.

- To evaluate `with` expressions, we calculate the named expression, then substitute its value in the body.
- How about identifiers? Well, any identifier that is in the scope of a `with` is replaced with a value when the calculator encounters that identifier's binding instance. Consequently, the purpose statement of *subst* said there would be no free instances of the identifier given as an argument left in the result. In other words, *subst* replaces identifiers with values before the calculator ever "sees" them. As a result, any as-yet-unsubstituted identifier must be free in the whole program. The calculator can't assign a value to a free identifier, so it halts with an error.

```
;; calc : WAE → number
;; evaluates WAE expressions by reducing them to numbers

(define (calc expr)
  (type-case WAE expr
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))]
    [with (bound-id named-expr bound-body)
          (calc (subst bound-body
                       bound-id
                       (num (calc named-expr))))]
    [id (v) (error 'calc "free identifier")]))
```

Observe that the step we earlier labeled "descend" is handled by the recursive invocation of *calc*. One subtlety: In the rule for `with`, the value returned by *calc* is a number, but *subst* is expecting a WAE expression. Therefore, we wrap the result in (num ···) so that substitution will work correctly.

Here are numerous test cases. Each one should pass:

(*test* (*calc* (*parse* '5)) 5)
(*test* (*calc* (*parse* '{+ 5 5})) 10)
(*test* (*calc* (*parse* '{with {*x* {+ 5 5}} {+ *x* *x*}})) 20)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* *x*}})) 10)
(*test* (*calc* (*parse* '{with {*x* {+ 5 5}} {*with* {*y* {- *x* 3}} {+ *y* *y*}}})) 14)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*y* {- *x* 3}} {+ *y* *y*}}})) 4)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*x* 3} 10}}})) 15)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*x* 3} *x*}}})) 8)
(*test* (*calc* (*parse* '{with {*x* 5} {+ *x* {*with* {*y* 3} *x*}}})) 10)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*y* *x*} *y*}})) 5)
(*test* (*calc* (*parse* '{with {*x* 5} {*with* {*x* *x*} *x*}})) 5)

3.3 The Scope of `with` Expressions

Just when we thought we were done, we find that several of the test cases above (can you determine which ones?) generate a free-identifier error. What gives?

Consider the program

```
{with {x 5}
  {with {y x}
    y}}
```

Common sense would dictate that its value is `5`. So why does the calculator halt with an error on this test case?

As defined, *subst* fails to correctly substitute in this program, because we did not account for the named sub-expressions in `with` expressions. To fix this problem, we simply need to make `subst` treat the named expressions as ordinary expressions, ripe for substitution. To wit:

```
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              expr
              (with bound-id
```

```
                  (subst named-expr sub-id val)
                  (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

The boxed expression shows what changed.

Actually, this isn't quite right either: consider

```
{with {x 5}
  {with {x x}
    x}}
```

This program should evaluate to `5`, but it too halts with an error. This is because we prematurely stopped substituting for `x`. We should substitute in the named expression of a `with` even if the `with` in question defines a new scope for the identifier being substituted, because its named expression is still in the scope of the enclosing binding of the identifier.

We finally get a valid programmatic definition of substitution (relative to the language we have so far):

```
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))
```

Observe how the different versions of *subst* have helped us refine the scope of `with` expressions. By focusing on the small handful of lines that change from one version to the next, and studying how they change, we progressively arrive at a better understanding of scope. This would be much harder to do through mere prose; indeed, our prose definition has not changed at all through these program changes, but translating the definition into a program and running it has helped us refine our intutition.

Exercise 3.3.1 *What's the value of*

```
{with {x x} x}
```

? What should *it be, and what does your calculator say it* is*? (These can be two different things!)*

3.4 What Kind of Redundancy do Identifiers Eliminate?

We began this material motivating the introduction of `with`: as a means for eliminating redundancy. Let's revisit this sequence of substitutions (skipping a few intermediate steps):

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {x 10} {with {y {- x 3}} {+ y y}}}
= {with {y {- 10 3}} {+ y y}}
= {with {y 7} {+ y y}}
= {+ 7 7}
= 14
```

Couldn't we have also written it this way?

```
  {with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}
= {with {y {- {+ 5 5} 3}} {+ y y}}
= {+ {- {+ 5 5} 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- {+ 5 5} 3}}
= {+ {- 10 3} {- 10 3}}
= {+ 7 {- 10 3}}
= {+ 7 7}
= 14
```

In the first sequence of reductions, we first reduce the named expression to a number, then substitute that number. In the second sequence, we perform a "textual" substitution, and only when we have no substitutions left to do do we begin to perform the arithmetic.

Notice that this shows there are really two interpretations of "redundancy" in force. One is a purely *static*[4] notion of redundancy: `with` exists solely to avoid writing an expression twice, even though it will be evaluated twice. This is the interpretation taken by the latter sequence of reductions. In contrast, the former sequence of reductions manifests both static and *dynamic*[5] redundancy elimination: it not only abbreviates the program, it also avoids re-computing the same value during execution.

Given these two sequences of reductions (which we will call *reduction regimes*, since each is governed by a different set of rules), which does our calculator do? Again, it would be hard to reason about this verbally, but because we've written a program, we have a concrete object to study. In particular, the lines we should focus on are those for `with`. Here they are again:

```
...
          [with (bound-id named-expr bound-body)
                (calc (subst bound-body
                             bound-id
                             (num (calc named-expr))))]
...
```

[4]Meaning, referring only to program text.

[5]Meaning, referring to program execution.

The boxed portion tells us the answer: we invoke *calc* before substitution (because the *result* of *calc* is what we supply as an argument to *subst*). This model of substitution is called *eager*: we "eagerly" reduce the named expression to a value before substituting it. This is in contrast to the second sequence of reductions above, which we call *lazy*, wherein we reduce the named expression to a value only when we need to (such as at the application of an arithmetic primitive).

At this point, it may seem like it doesn't make much difference which reduction regime we employ: both produce the same answer (though they may take a different number of steps). But do keep this distinction in mind, for we will see a good deal more on this topic in the course of our study.

Exercise 3.4.1 *Can you* prove *that the eager and lazy regimes will always produce the same answer for the* WAE *language?*

Exercise 3.4.2 *In the example above, the eager regime generated an answer in fewer steps than the lazy regime did. Either prove that that will always be the case, or provide a counterexample.*

Exercise 3.4.3 *At the beginning of this section, you'll find the phrase*

> *an identifier names, or identifies, (the value of) an expression*

Why the parenthetical phrase?

3.5 Are Names Necessary?

A lot of the trouble we've had with defining substitution is the result of having the same name be bound multiple times. To remedy this, a computer scientist named Nicolaas de Bruijn had a good idea.[6] He asked the following daring question: Who needs names at all? De Bruijn suggested that instead, we replace identifiers with numbers. The number dictates how many enclosing scopes away the identifier is bound. (Technically, we replace identifiers not with numbers but *indices* that indicate the binding depth. A number is just a convenient *representation* for an index. A more pictorially pleasing representation for an index is an arrow that leads from the bound to the binding instance, like the ones DrScheme's Check Syntax tool draws.)

The idea is easy to explain with an example: intead of writing

```
{with {x 5} {+ x x}}
```

we would write

```
{with 5 {+ <0> <0>}}
```

Notice that two things changed. First, we replaced the bound identifiers with indices (to keep indices separate from numbers, we wrap each index in pointy brackets). We've adopted the convention that the current scope is zero levels away. Thus, `x` becomes `<0>`. The second change is that, because we no longer care about the names of identifiers, we no longer need keep track of the `x` as the bound identifier. The presence of `with` indicates that we've entered a new scope; that's enough. Similarly, we convert

[6] De Bruijn had *many* great ideas, particularly in the area of using computers to solve math problems. The idea we present here was a small offshoot of that much bigger project, but as it so happens, this is the one many people know him for.

```
{with {x 5}
  {with {y 3}
    {+ x y}}}
```

into

```
{with 5
  {with 3
    {+ <1> <0>}}}
```

Let's consider one last example. If this looks incorrect, that would suggest you may have misunderstood the scope of a binding. Examining it carefully actually helps to clarify the scope of bindings. We convert

```
{with {x 5}
  {with {y {+ x 3}}
    {+ x y}}}
```

into

```
{with 5
  {with {+ <0> 3}
    {+ <1> <0>}}}
```

De Bruijn indices are useful in many contexts, and indeed the de Bruijn form of a program (that is, a program where all identifiers have been replaced by their de Bruijn indices) is employed by just about every compiler. You will sometimes find compiler texts refer to the indices as *static distance coordinates*. That name makes sense: the coordinates tell us how far away statically—i.e., in the program text— an identifier is bound. I prefer to use the less informative but more personal moniker as a form of tribute.

```
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)])

;; subst : WAE symbol WAE → WAE
(define (subst expr sub-id val)
  (type-case WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [sub (l r) (sub (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]))

;; calc : WAE → number
(define (calc expr)
  (type-case WAE expr
    [num (n) n]
    [add (l r) (+ (calc l) (calc r))]
    [sub (l r) (- (calc l) (calc r))]
    [with (bound-id named-expr bound-body)
          (calc (subst bound-body
                       bound-id
                       (num (calc named-expr))))]
    [id (v) (error 'calc "free identifier")]))
```

Figure 3.1: Calculator with `with`

Chapter 4

An Introduction to Functions

Through the agency of `with`, we have added identifiers and the ability to name expressions to the language. Much of the time, though, simply being able to name an expression isn't enough: the expression's value is going to depend on the context of its use. That means the expression needs to be parameterized; that is, it must be a *function*.

Dissecting a `with` expression is a useful exercise in helping us design functions. Consider the program

```
{with {x 5} {+ x 3}}
```

In this program, the expression `{+ x 3}` is parameterized over the value of `x`. In that sense, it's just like a function definition: in mathematical notation, we might write

$$f(x) = x + 3$$

Having named and defined f, what do we do with it? The WAE program introduces `x` and then immediately binds it to `5`. The way we bind a function's argument to a value is to apply it. Thus, it is as if we wrote

$$f(x) = x + 3; f(5)$$

In general, functions are useful entities to have in programming languages, and it would be instructive to model them.

4.1 Enriching the Language with Functions

We will initially model the DrScheme programming environment, which has separate windows for Definitions and Interactions. The Interactions window is DrScheme's "calculator", and the part we are trying to model with our calculators. The contents of the Definitions window are "taught" to this calculator by clicking the Run button. Our calculator should therefore consume an argument that reflects these definitions.

To add functions to WAE, we must define their concrete and abstract syntax. In particular, we must both describe a function *definition*, or *declaration*, and provide a means for its *use*, also known as an *application* or *invocation*. To do the latter, we must add a new kind of expression, resulting in the language F1WAE.[1]

[1] The reason for the "1" will become clear in Section 6.

We will presume, as a simplification, that functions consume only one argument. This expression language has the following BNF:

```
<F1WAE> ::= <num>
          | {+ <F1WAE> <F1WAE>}
          | {with {<id> <F1WAE>} <F1WAE>}
          | <id>
          | {<id> <F1WAE>}
```

(The expression representing the argument supplied to the function is known as the *actual parameter*.)

We have dropped subtraction from the language on the principle that it is similar enough to addition for us to determine its implementation from that of addition. To capture this new language, we employ terms of the following type:

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])
```

Convince yourself that this is an appropriate definition.

Now let's study function definitions. A function definition has three parts: the name of the function, the names of its arguments, known as the *formal parameters*, and the function's body. (The function's parameters may have types, which we will discuss in Chapter X.) For now, we will presume that functions consume only one argument. A simple data definition captures this:

```
(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])
```

Using this definition, we can represent a standard function for doubling its argument as

```
(fundef 'double
        'n
        (add (id 'n) (id 'n)))
```

Now we're ready to write the calculator, which we'll call *interp*—short for *interpreter*—rather than *calc* to reflect the fact that our language has grown beyond arithmetic. The interpreter must consume two arguments: the expression to evaluate, and the set of known function definitions. This corresponds to what the Interactions window of DrScheme works with. The rules present in the WAE interpreter remain the same, so we can focus on the one new rule.

```
;; interp : F1WAE listof(fundef) → number
;; evaluates F1WAE expressions by reducing them to their corresponding values

(define (interp expr fun-defs)
```

```
(type-case F1WAE expr
  [num (n) n]
  [add (l r) (+ (interp l fun-defs) (interp r fun-defs))]
  [with (bound-id named-expr bound-body)
        (interp (subst bound-body
                       bound-id
                       (num (interp named-expr fun-defs)))
                fun-defs)]
  [id (v) (error 'interp "free identifier")]
  [app (fun-name arg-expr)
       (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
         (interp (subst (fundef-body the-fun-def)
                        (fundef-arg-name the-fun-def)
                        (num (interp arg-expr fun-defs)))
                 fun-defs))])
```

The rule for an application first looks up the named function. If this access succeeds, then interpretation proceeds in the body of the function after first substituting its formal parameter with the (interpreted) value of the actual parameter. We see the result in DrScheme:

```
> (interp (parse '{double {double 5}})
          (list (fundef 'double
                        'n
                        (add (id 'n) (id 'n)))))
20
```

To make this interpreter function correctly, we must make several changes. First, we must adapt the parser to treat the relevant inputs (as identified by the BNF) as function applications. Second, we must modify the interpreter itself, changing the recursive calls to take an extra argument, and adding the implementation of app. Third, we must extend *subst* to handle the F1WAE language. Finally, we must write *lookup-fundef*, the helper routine that finds function definitions. The last two changes are shown in Figure 4.1.

Exercise 4.1.1 *Why is the argument expression of an application of type* F1WAE *rather than of type* WAE*? Provide sample programs permitted by the former and rejected by the latter, and argue that these programs are reasonable.*

Exercise 4.1.2 *Why is the body expression of a function definition of type* F1WAE *rather than of type* WAE*? Provide a sample definition permitted by using the former rather than the latter, and argue that it is reasonable.*

4.2 The Scope of Substitution

Suppose we ask our interpreter to evaluate the expression

```
(app 'f (num 10))
```

in the presence of the solitary function definition

```
(fundef 'f
        'n
        (app 'n (id 'n)))
```

What should happen? Should the interpreter try to substitute the *n* in the function position of the application with the number 10, then complain that no such function can be found (or even that function lookup fundamentally fails because the names of functions must be identifiers, not numbers)? Or should the interpreter decide that function names and function arguments live in two separate "spaces", and context determines in which space to look up a name? Languages like Scheme take the former approach: the name of a function can be bound to a value in a local scope, thereby rendering the function inaccessible through that name. This latter strategy is known as employing *namespaces* and languages such as Common Lisp adopt it.

4.3 The Scope of Function Definitions

Suppose our definition list contains multiple function definitions. How do these interact with one another? For instance, suppose we evaluate the following input:

```
(interp (parse '{f 5})
        (list (fundef 'f 'n (app 'g (add (id 'n) (num 5))))
              (fundef 'g 'm (sub (id 'm) (num 1)))))
```

What does this program do? The main expression applies `f` to `5`. The definition of `f`, in turn, invokes function `g`. Should `f` be able to invoke `g`? Should the invocation fail because `g` is defined after `f` in the list of definitions? What if there are multiple bindings for a given function's name?

We will expect this program to evaluate to `9`. We employ the natural interpretation that each function can "see" every function's definition, and the natural assumption that each name is bound at most once so we needn't disambiguate between definitions. Is is, however, possible to define more sophisticated scopes.

Exercise 4.3.1 *If a function can invoke every defined function, that means it can also invoke itself. This is currently of limited value because the language* F1WAE *lacks a harmonious way of terminating recursion. Consider adding a simple conditional construct (such as* `if0`*, which succeeds if the term in the first position evaluates to* `0`*) and writing interesting programs in this language.*

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])

(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])

;; lookup-fundef : symbol listof(FunDef) ⟶ FunDef
(define (lookup-fundef fun-name fundefs)
  (cond
    [(empty? fundefs) (error fun-name "function not found")]
    [else (if (symbol=? fun-name (fundef-fun-name (first fundefs)))
              (first fundefs)
              (lookup-fundef fun-name (rest fundefs)))]))

;; subst : F1WAE symbol F1WAE → F1WAE
(define (subst expr sub-id val)
  (type-case F1WAE expr
    [num (n) expr]
    [add (l r) (add (subst l sub-id val)
                    (subst r sub-id val))]
    [with (bound-id named-expr bound-body)
          (if (symbol=? bound-id sub-id)
              (with bound-id
                    (subst named-expr sub-id val)
                    bound-body)
              (with bound-id
                    (subst named-expr sub-id val)
                    (subst bound-body sub-id val)))]
    [id (v) (if (symbol=? v sub-id) val expr)]
    [app (fun-name arg-expr)
         (app fun-name (subst arg-expr sub-id val))]))
```

Figure 4.1: Implementation of Functions: Support Code

```
;; interp : F1WAE listof(fundef) → number
(define (interp expr fun-defs)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs) (interp r fun-defs))]
    [with (bound-id named-expr bound-body)
          (interp (subst bound-body
                         bound-id
                         (num (interp named-expr fun-defs)))
                  fun-defs)]
    [id (v) (error 'interp "free identifier")]
    [app (fun-name arg-expr)
         (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
           (interp (subst (fundef-body the-fun-def)
                          (fundef-arg-name the-fun-def)
                          (num (interp arg-expr fun-defs)))
                   fun-defs))]))
```

Figure 4.2: Implementation of Functions: Interpreter

Chapter 5

Deferring Substitution

Let's examine the process of interpreting the following small program. Consider the following sequence of evaluation steps:

```
  {with {x 3}
    {with {y 4}
      {with {z 5}
        {+ x {+ y z}}}}}
= {with {y 4}
    {with {z 5}
      {+ 3 {+ y z}}}}
= {with {z 5}
    {+ 3 {+ 4 z}}}
= {+ 3 {+ 4 5}}
```

at which point it reduces to an arithmetic problem. To reduce it, though, the interpreter had to apply substitution three times: once for each `with`. This is slow! How slow? Well, if the program has size n (measured in abstract syntax tree nodes), then each substitution sweeps the rest of the program once, making the complexity of this interpreter at least $O(n^2)$. That seems rather wasteful; surely we can do better.

How do we avoid this computational redundancy? We should create and use a *repository of deferred substitutions*. Concretely, here's the idea. Initially, we have no substitutions to perform, so the repository is empty. Every time we encounter a substitution (in the form of a `with` or application), we augment the repository with one more entry, recording the identifier's name and the value (if eager) or expression (if lazy) it should eventually be substituted with. We continue to evaluate without actually performing the substitution.

This strategy breaks a key invariant we had established earlier, which is that any identifier the interpreter encounters is of necessity free, for had it been bound, it would have been replaced by substitution. Because we're no longer using substitution, we will encounter bound identifiers during interpretation. How do we handle them? We must substitute them with by consulting the repository.

Exercise 5.0.2 *Can the complexity of substitution be worse than $O(n^2)$?*

5.1 The Substitution Repository

Let's provide a data definition for the repository:

```
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (ds DefrdSub?)])
```

where DefrdSub stands for "deferred substitutions". A DefrdSub has two forms: it's either empty (mtSub[1]) or non-empty (represented by an aSub structure). The latter contains a reference to the rest of the repository in its third field.

The interpreter obviously needs to consume a repository in addition to the expression to interpret. Therefore, its contract becomes

```
;; interp : F1WAE listof(fundef) DefrdSub → number
```

It will need a helper function that looks up the value of identifiers in the repository. Its code is:

```
;; lookup : symbol DefrdSub → F1WAE

(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-ds)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-ds))]))
```

With that introduction, we can now present the interpreter:

```
(define (interp expr fun-defs ds)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs ds) (interp r fun-defs ds))]
    [with (bound-id named-expr bound-body)
          (interp bound-body
                  fun-defs
                  (aSub bound-id
                        (interp named-expr
                                fun-defs
                                ds)
                        ds))]
    [id (v) (lookup v ds)]
    [app (fun-name arg-expr)
         (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
```

[1] "Empty sub"—get it?

```
(interp (fundef-body the-fun-def)
        fun-defs
        (aSub (fundef-arg-name the-fun-def)
              (interp arg-expr fun-defs ds)
              ds)))]))
```

Three clauses have changed: those for `with`, identifiers and applications. Applications must look up the value of an identifier in the repository. The rule for `with` evaluates the body in a repository that extends the current one (*ds*) with a binding for the `with`-bound identifier to its interpreted value. The rule for an application similarly evaluates the body of the function with the repository extended with the formal argument bound to the value of the actual argument.

To make sure this is correct, we recommend that you first study its behavior on programs that have no identifiers—i.e., verify that the arithmetic rules do the right thing—and only then proceed to the rules that involve identifiers.

5.2 Deferring Substitution Correctly

Consider the evaluation of the expression

```
{with {n 5}
  {f 10}}
```

in the following list of function definitions:

```
(list (fundef 'f 'p (id 'n)))
```

That is, `f` consumes an argument `p` and returns the value bound to `n`. This corresponds to the Scheme definition

```
(define (f p) n)
```

followed by the application

```
(local ([define n 5])
  (f 10))
```

What result does Scheme produce?

Our interpreter produces the value `5`. Is this the correct answer? Well, it's certainly *possible* that this is correct—after all, it's what the interpreter returns, and this could well be the interpreter for *some* language. But we do have a better way of answering this question.

Recall that the interpreter was using the repository to conduct substitution more efficiently. We hope that that's all it does—that is, it must not also change the meaning of a program! Our "reference implementation" is the one that performs explicit substitution. If we want to know what the value of the program really "is", we need to return to that implementation.

What does the substitution-based interpreter return for this program? It says the program has an unbound identifier (specifically, `n`). So we have to regard our interpreter with deferred substitutions as buggy.

While this new interpreter is clearly buggy relative to substitution, which it was supposed to represent, let's think for a moment about what we, as the human programmer, would *want* this program to evaluate to. It produces the value `5` because the identifier `n` gets the value it was bound to by the `with` expression, that is, from the scope in which the function `f` is *used*. Is this a reasonable way for the language to behave? A priori, is one interpretation better than the other? Before we tackle that, let's introduce some terminology to make it easier to refer to these two behaviors.

Definition 10 (Static Scope) *In a language with* static *scope, the scope of an identifier's binding is a syntactically delimited region.*

A typical region would be the body of a function or other binding construct. In contrast:

Definition 11 (Dynamic Scope) *In a language with* dynamic *scope, the scope of an identifier's binding is the entire remainder of the execution during which that binding is in effect.*

That is, in a language with dynamic scope, if a function `g` binds identifier `n` and then invokes `f`, then `f` can refer to `n`—and so can every other function invoked by `g` until it completes its execution—even though `f` has no locally visible binding for `n`.

Armed with this terminology, we claim that dynamic scope is entirely unreasonable. The problem is that we simply cannot determine what the value of a program will be without knowing everything about its execution history. If the function `f` were invoked by some other sequence of functions that did not bind a value for `n`, then that particular application of `f` would result in an error, even though a previous application of `f` in the very same program's execution completed successfully! In other words, simply by looking at the source text of `f`, it would be impossible to determine one of the most rudimentary properties of a program: whether or not a given identifier was bound. You can only imagine the mayhem this would cause in a large software system, especially with multiple developers and complex flows of control. We will therefore *regard dynamic scope as an error and reject its use* in the remainder of this text.

5.3 Fixing the Interpreter

Let's return to our interpreter. Our choice of static over dynamic scope has the benefit of confirming that the substituting interpreter did the right thing, so all we need do is make the new interpeter be a correct reimplementation of it. We only need to focus our attention on one rule, that for function application. This currently reads:

```
[app (fun-name arg-expr)
     (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
       (interp (fundef-body the-fun-def)
               fun-defs
               (aSub (fundef-arg-name the-fun-def)
                     (interp arg-expr fun-defs ds)
                     ds)))]
```

When the interpreter evaluates the body of the function definition, what deferred substitutions does it recognize? It recognizes all those already are in *ds*, with one more for the function's formal parameter to be replaced with the value of the actual parameter. But how many substitutions *should* be in effect in the function's body? In our substitution-based interpreter, we implemented application as follows:

```
[app (fun-name arg-expr)
     (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
       (interp (subst (fundef-body the-fun-def)
                      (fundef-arg-name the-fun-def)
                      (num (interp arg-expr fun-defs)))
               fun-defs))]
```

This performs only one substitution on the function's body: the formal parameter for its value. The code demonstrates that none of the substitutions applied to the *calling* function are in effect in the body of the *called* function. Therefore, at the point of invoking a function, our new interpeter must "forget" about the current substitutions. Put differently, at the beginning of every function's body, there is only one bound identifier—the function's formal parameter—independent of the invoking context.

How do we fix our implementation? We clearly need to create a substitution for the formal parameter (which we obtain using the expression (*fundef-arg-name the-fun-def*)). But the remaining substitutions must be empty, so as to not pick up the bindings of the calling context. Thus,

```
[app (fun-name arg-expr)
     (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
       (interp (fundef-body the-fun-def)
               fun-defs
               (aSub (fundef-arg-name the-fun-def)
                     (interp arg-expr fun-defs ds)
                     (mtSub))))]
```

That is, we use the empty repository to initiate evaluation of a function's body, extending it immediately with the formal parameter but no more. The difference between using *ds* and (mtSub) in the position of the box succintly captures the implementation distinction between dynamic and static scope, respectively—though the *consequences* of that distinction are far more profound that this small code change might suggest.

Exercise 5.3.1 *How come we never seem to "undo" additions to the repository? Doesn't this run the risk that one substitution might override another in a way that destroys static scoping?*

Exercise 5.3.2 *Why is the last ds in the interpretation of* `with` *also not replaced with (*mtSub*)? What would happen if we were to effect this replacement? Write a program that illustrates the difference, and argue whether the replacement is sound or not.*

Exercise 5.3.3 *Our implementation of lookup can take time linear in the size of the program to find some identifiers. Therefore, it's not clear we have really solved the time-complexity problem that motivated the use of a substitution repository. We could address this by using a better data structure and algorithm for lookup: a hash table, say. What changes do we need to make if we use a hash table?*
Hint: *This is tied closely to Exercise 5.3.1!*

```
(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (fun-name symbol?) (arg F1WAE?)])

(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])

(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (ds DefrdSub?)])

;; lookup-fundef : symbol listof(FunDef) ⟶ FunDef
(define (lookup-fundef fun-name fundefs)
  (cond
    [(empty? fundefs) (error fun-name "function not found")]
    [else (if (symbol=? fun-name (fundef-fun-name (first fundefs)))
              (first fundefs)
              (lookup-fundef fun-name (rest fundefs)))]))

;; lookup : symbol DefrdSub → F1WAE
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-ds)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-ds))]))
```

Figure 5.1: Functions with Deferred Substitutions: Support Code

```
;; interp : F1WAE listof(fundef) DefrdSub → number
(define (interp expr fun-defs ds)
  (type-case F1WAE expr
    [num (n) n]
    [add (l r) (+ (interp l fun-defs ds) (interp r fun-defs ds))]
    [with (bound-id named-expr bound-body)
          (interp bound-body
                  fun-defs
                  (aSub bound-id
                        (interp named-expr
                                fun-defs
                                ds)
                        ds))]
    [id (v) (lookup v ds)]
    [app (fun-name arg-expr)
         (local ([define the-fun-def (lookup-fundef fun-name fun-defs)])
           (interp (fundef-body the-fun-def)
                   fun-defs
                   (aSub (fundef-arg-name the-fun-def)
                         (interp arg-expr fun-defs ds)
                         (mtSub))))]))
```

Figure 5.2: Functions with Deferred Substitutions: Interpreter

Chapter 6

First-Class Functions

We began Section 4 by observing the similarity between a `with` expression and a function definition applied immediately to a value. Specifically, we observed that

```
{with {x 5} {+ x 3}}
```

is essentially the same as

$$f(x) = x + 3; f(5)$$

Actually, that's not *quite* right: in the math equation above, we give the function a name, f, whereas there is no identifier named `f` anywhere in the WAE program. We can, however, rewrite the mathematical formulation as

$$f = \lambda(x).x + 3; f(5)$$

which we can then rewrite as

$$(\lambda(x)x + 3)(5)$$

to get rid of the unnecessary name (f).

Notice, however, that our langugae F1WAE does not permit anonymous functions of the style we've used above. Because such functions are useful in their own right, we now extend our study of functions.

6.1 A Taxonomy of Functions

The translation of `with` into mathematical notation exploits two features of functions: the ability to create anonymous functions, and the ability to define functions anywhere in the program (in this case, in the function position of an application). Not every programming language offers one or both of these capabilities. There is, therefore, a taxonomy that governs these different features, which we can use when discussing what kind of functions a language provides:

first-order Functions are not values in the language. They can only be defined in a designated portion of the program, where they must be given names for use in the remainder of the program. The functions in F1WAE are of this nature, which explains the 1 in the name of the language.

higher-order Functions can return other functions as values.

first-class Functions are values with all the rights of other values. In particular, they can be supplied as the value of arguments to functions, returned by functions as answers, and stored in data structures.

We would like to extend F1WAE to have the full power of functions, to reflect the capability of Scheme. In fact, it will be easier to return to WAE and extend it with first-class functions.

6.2 Enriching the Language with Functions

To add functions to WAE, we must define their concrete and abstract syntax. First let's examine some concrete programs:

```
{{fun {x} {+ x 4}}
 5}
```

This program defines a function that adds `4` to its argument and immediately applies this function to `5`, resulting in the value `9`. This one

```
{with {double {fun {x} {+ x x}}}
  {+ {double 10}
     {double 5}}}
```

evaluates to `30`. The program defines a function, binds it to `double`, then uses that name twice in slightly different contexts (i.e., it instantiates the formal parameter with different actual parameters).

From these examples, it should be clear that we must introduce two new kinds of expressions: function applications (as before), as well as (anonymous) function definitions. Here's the revised BNF corresponding to these examples:

```
<FWAE> ::= <num>
         | {+ <FWAE> <FWAE>}
         | {with {<id> <FWAE>} <FWAE>}
         | <id>
         | {fun {<id>} <FWAE>}
         | {<FWAE> <FWAE>}
```

Note that F1WAE did not have function definitions as part of the expression language, since the definitions were assumed to reside outside the expression being evaluated. In this language, functions can be defined anywhere, including in the function position of an application (the last BNF production): instead of just the name of a function, programmers can write an arbitrary expression that must be evaluated to obtain the function to apply. The corresponding abstract syntax is:

(**define-type** FWAE
  [num (*n number?*)]
  [add (*lhs FWAE?*) (*rhs FWAE?*)]
  [with (*name symbol?*) (*named-expr FWAE?*) (*body FWAE?*)]

[id (*name symbol?*)]
[fun (*param symbol?*) (*body FWAE?*)]
[app (*fun-expr FWAE?*) (*arg-expr FWAE?*)])

To define our interpreter, we must think a little about what kinds of values it consumes and produces. Naturally, the interpreter consumes values of type FWAE. What does it produce? Clearly, a program that meets the WAE description must yield numbers. As we have seen above, some program that use functions and applications also evaluate to numbers. How about a program that consists solely of a function? That is, what is the value of the program

```
{fun {x} x}
```

? It clearly doesn't represent a number. It may be a function that, *when applied* to a numeric argument, produces a number, but it's not itself a number (if you think differently, you need to indicate which number it will be: 0? 1? 1729?). We instead realize from this that functions are also values that may be the result of a computation.

We could design an elaborate representation for function values, but for now, we'll remain modest. We'll let the function evaluate to its abstract syntax representation (i.e., a fun structure). (We will soon get more sophisticated than this.) For consistency, we'll also let numbers evaluate to num structures. Thus, the result of evaluating the program above would be the value

(fun 'x (id 'x))

Now we're ready to write the interpreter. We must pick a type for the value that *interp* returns. Since we've decided to represent function and number answers using the abstract syntax, it makes sense to use FWAE, with the caveat that only two kinds of FWAE terms can appear in the output: numbers and functions. Our first interpreter will use explicit substitution, to offer a direct comparison with the corresponding WAE and F1WAE interpreters.

```
;; interp : FWAE → FWAE
;; evaluates FWAE expressions by reducing them to their corresponding values
;; return values are either num or fun


(define (interp expr)
  (type-case FWAE expr
    [num (n) expr]
    [add (l r) (add-numbers (interp l) (interp r))]
    [with (bound-id named-expr bound-body)
          (interp (subst bound-body
                         bound-id
                         (interp named-expr)))]
    [id (v) (error 'interp "free identifier")]
```

```
[fun (bound-id bound-body)
     expr]
[app (fun-expr arg-expr)
     (local ([define fun-val (interp fun-expr)])
       (interp (subst (fun-body fun-val)
                      (fun-param fun-val)
                      (interp arg-expr))))]
```

))

(We've made a small change to the rule for add: it uses a helper named *add-numbers* because *interp* now returns an FWAE. As an exercise, define this helper function for yourself.)

The rule for a function says, simply, to return the function itself. (Notice the similarity to the rule for numbers, the other kind of constant in our language!) That leaves only the rule for applications to study. This rule first evaluates the function position of an application. This is because that position may itself contain a complex expression that needs to be reduced to an actual function. For instance, in the expression

```
{{{fun {x} x}
  {fun {x} {+ x 5}}}
 3}
```

the outer function position consists of the application of the identity function to a function that adds five to its argument.

When evaluated, the function position had better reduce to a function value, not a number (or anything else). For now, we implicitly assume that the programs fed to the interpreter have no errors. (Chapter X will study how we can detect erroneous programs.) Given a function, we need to evaluate its body after having substituted the formal argument with its actual value. That's what the rest of the program does: evaluate the expression that will become the bound value, bind it using substitution, and then interpet the resulting expression. The last few lines are very similar to the code for `with`.

To understand this interpreter better, consider what it produces in response to evaluating the following term:

```
{with {x 3}
  {fun {y}
    {+ x y}}}
```

DrScheme prints the following:

```
(fun 'y (add (num 3) (id 'y)))
```

Notice that the `x` inside the function body has been replaced by `3` as a result of substitution, so the function has no references to `x` left in it.

Exercise 6.2.1 *What induced the small change in the rule for* add*? Explain, with an example, what would go wrong if we did not make this change.*

Exercise 6.2.2 *Did you notice the small change in the interpretation of* `with`*?*

Exercise 6.2.3 *What goes wrong if the interpreter fails to evaluate the function position (by invoking the interpreter on it)? Write a program and present the expected and actual results.*

6.3 Making `with` Redundant

Now that we have functions and function invocation as two distinct primitives, we can combine them to recover the behavior of `with` as a special case. Every time we encounter an expression of the form

```
{with {var named} body}
```

we can replace it with

```
{{fun {var} body}
 named}
```

and obtain the same effect. The result of this translation does not use `with`, so it can be evaluated by a more primitive interpreter: one for AE enriched with functions.

Exercise 6.3.1 *Implement a pre-processor that performs this translation.*

We will assume the existence of such a pre-processor, and use the language FAE as our basis for subsequent exploration. Section 36 discusses such pre-processors in great detail.

6.4 Implementing Functions using Deferred Substitutions

As Section 5 described, our implementation will be more sprightly if we deferred substitutions instead of performing them greedily. Let's study how to adapt our interpreter to use this more efficient strategy.

First, we must provide a definition of the substitution repository. The repository associates identifiers with their values. Previously, the value had always been a number, but now our set of values is richer. We therefore use the following type, with the understanding that the value will always be a num or fun:

```
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value FAE?) (ds DefrdSub?)])
```

Relative to this, the definition of *lookup* remains the same (only it now returns values of type FAE).

Our first attempt at adapting the interpreter is

```
(define (interp expr ds)
  (type-case FAE expr
    [num (n) expr]
    [add (l r) (add-numbers (interp l ds) (interp r ds))]
    [id (v) (lookup v ds)]
    [fun (bound-id bound-body)
         expr]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr ds)])
           (interp (fun-body fun-val)
                   (aSub (fun-param fun-val)
```

(*interp arg-expr ds*)
ds)))]))

When we run a battery of tests on this interpreter, we find that the expression

```
{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {f 4}}}}
```

evaluates to `9`. This should be surprising, because we seem to again have introduced dynamic scope! (Notice that the value of `x` depends on the context of the application of `f`, not its definition.)

To understand the problem better, let's return to this example, which we examined in the context of the substitution interpreter: the result of interpreting

```
{with {x 3}
  {fun {y}
    {+ x y}}}
```

in the substitution interpreter is

(fun 'y (add (num 3) (id 'y)))

That is, it had substituted the `x` with `3` in the procedure. Now we are deferring substitutions, but we haven't modified how the interpreter evaluates a function definition. As a result, the above expression evaluates to

(fun 'y (add (id 'x) (id 'y)))

But what happened to the substitutions deferred in its body?

The moral is, to properly defer substitution, the value of a function should be not only its text, but also the substitutions that were due to be performed on it. We therefore define a new datatype for the interpreter's return value that attaches the definition-time repository to every function value:

(**define-type** FAE-Value
  [numV (*n number?*)]
  [closureV (*param symbol?*)
            (*body FAE?*)
            (*ds DefrdSub?*)])

Accordingly, we change the rule for fun in the interpreter to

[fun (*bound-id bound-body*)
  (closureV *bound-id bound-body ds*)]

We call this constructed value a *closure* because it "closes" the function body over the substitutions that are waiting to occur.

When the interpreter encounters a function application, it must ensure that the function's pending substitutions aren't forgotten. It must, however, ignore the substitutions pending at the location of the *invocation*, for that is precisely what led us to dynamic instead of static scope. It must instead use the substitutions of the

invocation location to convert the function and argument into values, hope that the function expression evaluated to a closure, then proceed with evaluating the body employing the repository of deferred substitutions stored in the closure.

```
[app (fun-expr arg-expr)
     (local ([define fun-val (interp fun-expr ds)])
       (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
                     (interp arg-expr ds)
                     (closureV-ds fun-val)))))]
```

That is, having evaluated *fun-expr* to yield *fun-val*, we obtain not only the actual function body from *fun-val*'s closure record but also the repository stored within it. Crucially, while we evaluate *arg-expr* in *ds*, the repository active at the invocation location, we evaluate the function's body *in its "remembered" repository*. Once again, the content of this boxed expression determines the difference between static and dynamic scope. Figure 6.2 presents the complete interpreter.

It is worth looking back at the interpreter in Figure 5.2. Observe that in the application rule, the repository used (in place of the boxed expression above) is (mtSub). This is just the special case when functions cannot be nested within one another (so there are no deferred substitutions to record). The C programming language implements a middle-ground by allowing functions (technically, pointers to functions) to serve as values but allowing them to be defined only at the top-level. Because they are defined at the top-level, they have no deferred substitutions at the point of declaration; this means the second field of a closure record is always empty, and can therefore be elided, making the function pointer a complete description of the function. This purchases some small implementation efficiency, at the cost of potentially great expressive power, as the examples in Section 6.5 illustrate.

Returning to the interpreter above, the input

```
{with {x 3}
  {fun {y}
    {+ x y}}}
```

results in the following output:

```
(closureV 'y
          (add (id 'x) (id 'y))
          (aSub 'x (numV 3) (mtSub)))
```

That is, the closure has the same effect as the output under substitution, namely

```
(fun 'y (add (num 3) (id 'y)))
```

except that the deferred substitution of `3` for `x` has been recorded in the closure's repository.

Exercise 6.4.1 *This interpreter does not check whether the function position evaluates to a closure value. Modify the interpreter to check and, if the expression fails to yield a closure, report an error.*

Exercise 6.4.2 *Suppose we explicitly implemented* `with` *in the interpreter. Given that* `with` *is just a shorthand for creating and applying a closure, would the changes we made to closure creation and function application have an effect on* `with` *too?*

6.5 Some Perspective on Scope

The example above that demonstrated the problem with our caching interpreter might not be a very convincing demonstration of the value of static scope. Indeed, you might be tempted to say, "If you know `x` is `3`, why not just use `3` instead of `x` inside the procedure declaration? That would avoid this problem entirely!" That's a legitimate response for that particular example, which was however meant only to *demonstrate the problem*, not to *motivate the need for the solution*. Lets's now consider three examples that do the latter.

6.5.1 Filtering and Sorting Lists

In the process of defining a sort routine like quicksort, we would want to define a helper procedure that filters all the elements less than the pivot:

```
;; filter-<-pivot: number list(number) → list(number)
(define (filter-<-pivot pivot l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (if (< (first l) pivot)
         (cons (first l) (filter-<-pivot pivot (rest l)))
         (filter-<-pivot pivot (rest l)))]))
```

Unfortunately, we need another procedure to filter all the elements greater than the pivot as well. It is poor programming practice to copy this code when we should, instead, be abstracting over it; furthermore, this code assumes that we will only want to sort lists of numbers (because it uses $<$, which compares only numbers), whereas we will want to sort lots of other types of lists, too. Therefore, a more sensible filtering routine would be this:

```
;; filter-pivot: X (X X → boolean) list(X) → list(X)
(define (filter-pivot pivot comp l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (if (comp (first l) pivot)
         (cons (first l) (filter-pivot pivot comp (rest l)))
         (filter-pivot pivot comp (rest l)))]))
```

We can then pass different procedures, such as $<$, $>=$, *string<=?*, and so on as the second argument, thereby using this procedure on lists of any comparable type.

While this is superior to the previous filter procedure, it is somehow unsatisfying because it does not properly reflect the *essence* of filtering. The essence, we would think, is a procedure with the following type:

```
;; filter-any: (X → boolean) list(X) → list(X)
```

That is, given a predicate that determines whether or not to keep an element in a list, and given a list, the filter procedure returns a list of just those elements that were chosen to be kept. The implementation of this is straightforward:

```
(define (filter-any comp l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (if (comp (first l))
         (cons (first l) (filter-any comp (rest l)))
         (filter-any comp (rest l)))]))
```

But now, consider the use of this procedure in a context such as quicksort: the comparator argument must *close* over the pivot. That is, the sorting routine has this form:

```
(define (qs comp l)
  (cond
    [(empty? l) empty]
    [(cons? l)
     (local ([define pivot (first l)])
       (append
        (qs comp (filter-any (lambda (x) (comp x pivot)) (rest l)))
        (list pivot)
        (qs comp (filter-any (lambda (x) (not (comp x pivot))) (rest l)))))]))
```

In a language with closures, it is entirely natural to write this program. If, however, the language did not have static scope, how would we be able to define the procedure given as an argument to *filter-any*? Examine the definition of sorting procedures in libraries for languages like C.

6.5.2 Differentiation

Let's implement (a simple version of) numeric differentiation in Scheme. The program is

```
(define H 0.0001)

(define (d/dx f)
  (lambda (x)
    (/ (- (f (+ x H)) (f x))
       H)))
```

In this example, in the algebraic expression, the identifier f is free relative to the inner function. However, we cannot do what we proposed earlier, namely to substitute the free variable with its value; this is because we don't know what values f will hold during execution, and in particular f will likely be bound to several different values over the course of the program's lifetime. If we run the inner procedure under dynamic scope, one of two things will happen: either the identifier f will not be bound to any value in the context of use, resulting in an unbound identifier error, or the procedure will use whatever f is bound to, which almost

certainly will not correspond to the value supplied to *d/dx*. That is, in a hypothetical dynamically-scoped Scheme, you would get

> (**define** *diff-of-square* (*d/dx* (**lambda** (*x*) (∗ *x x*))))
> (*diff-of-square* 10)
reference to undefined identifier: f
> (**define** *f* 'greg)
> (*diff-of-square* 10)
procedure application: expected procedure, *given: greg*; arguments were: 10.0001
> (**define** *f sqrt*)
> (*diff-of-square* 10)
0.15811348772487577

That is, *f* assumes whatever value it has at the point of *use*, ignoring the value given to it at the inner procedure's *definition*. In contrast, what we really get from Scheme is

> (*diff-of-square* 10)
20.000099999890608 ;; approximately $10 \times 2 = 20$

6.5.3 Callbacks

Let's consider another example, this one from Java. This program implements a *callback*, which is a common programming pattern employed in programming GUIs. In this instance, the callback is an object installed in a button; when the user presses the button, the GUI system invokes the callback, which brings up a message box displaying the number of times the user has pressed the button. This powerful paradigm lets the designer of the GUI system provide a generic library of graphical objects independent of the behavior each client wants to associate with that object.

```
// GUI library code
public class JButton {
    public void whenPressed(ActionEvent e) {
        for (int i = 0; i < listeners.length; ++i)
            listeners[i].actionPerformed(e);
    }
}

// User customization
public class GUIApp {
    private int count = 0;

    public class ButtonCallback implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            count = count + 1;
            JOptionPane.showMessageDialog(null,
```

```
                                                "Callback was invoked " +
                                                count + " times!");
        }
    }

    public Component createComponents() {
        JButton button = new JButton("Click me!");
        button.addActionListener(new ButtonCallback());
        return button;
    }
}
```

Stripped to its essence, the callback code is really no different from

```
;; GUI library code
(define (button callback)
  (local [(define (sleep-loop)
            (when button-pressed
              (begin
                (callback)
                (sleep-loop))))]
    (sleep-loop)))

;; User customization
(local [(define count 0)
        (define (my-callback)
          (begin
            (set! count (add1 count)) ;; increment counter
            (message-box
              (string-append "Callback was invoked "
                             (number→string count)
                             " times!"))))]
  (button my-callback))
```

That is, a callback is just a function passed to the GUI toolbox, which the toolbox invokes when it has an argument. But note that in the definition of *my-callback* (or `ButtonCallback`), the identifier *count* is not bound within the function (or object) itself. That is, it is *free* in the function. Therefore, whether it is scoped statically or dynamically makes a huge difference!

How do we want our callback to behave? Naturally, as the users of the GUI toolbox, we would be very upset if, the first time the user clicked on the button, the system halted with the message

```
error: identifier 'count' not bound
```

The bigger picture is this. As programmers, we hope that other people will use our functions, perhaps even in fantastic contexts that we cannot even imagine. Unfortunately, that means we can't possibly know

what the values of identifiers will be at the location of use, or whether they will even be bound. If we must rely on the locus of use, we will produce highly fragile programs: they will be useful only in very limited contexts, and their behavior will be unpredictable everywhere else.

Static scoping avoids this fear. In a language with static scope, the programmer has full power over choosing from the definition and use scopes. By default, free identifiers get their values from the definition scope. If the programmer wants to rely on a value from the use scope, they simply make the corresponding identifier a parameter. This has the added advantage of making very explicit in the function's interface which values from the use scope it relies on.

Dynamic scoping is primarily interesting as a historical mistake: it was in the earliest versions of Lisp, and persisted for well over a decade. Scheme was created as an experimental language in part to experiment with static scope. This was such a good idea that eventually, even Common Lisp adopted static scope. Most modern languages are statically scoped, but sometimes they make the mistake of recapitulating this phylogeny. So-called "scripting" languages, in particular, often make the mistake of implementing dynamic scope (or the lesser mistake of just failing to create closures), and must go through multiple iterations before they eventually implement static scope correctly.

6.6 Eagerness and Laziness

Recall that a lazy evaluator was one that did not reduce the named-expression of a `with` to a value at the time of binding it to an identifier. What is the corresponding notion of laziness in the presence of functions? Let's look at an example: in a lazy evaluator,

```
{with {x {+ 3 3}}
  {+ x x}}
```

would first reduce to

```
{+ {+ 3 3} {+ 3 3}}
```

But based on what we've just said in section 6.3 about reducing `with` to procedure application, the treatment of procedure arguments should match that of the named expression in a `with`. Therefore, a lazy language with procedures is one that does not reduce its argument to a value until necessary in the body. The following sequence of reduction rules illustrates this:

```
  {{fun {x} {+ x x}}
   {+ 3 3}}
= {+ {+ 3 3} {+ 3 3}}
= {+ 6 {+ 3 3}}
= {+ 6 6}
= 12
```

which is just an example of the `with` translation described above; a slightly more complex example is

```
{with {double {fun {x} {+ x x}}}
  {double {double 5}}}
```

```
= {{fun {x} {+ x x}}
   {{fun {x} {+ x x}}
    5}}
= {{fun {x} {+ x x}}
   {+ 5 5}}
= {+ {+ 5 5} {+ 5 5}}
= {+ 10 {+ 5 5}}
= {+ 10 10}
= 20
```

What do the corresponding reductions look like in an eager regime? Are there significant differences between the two?

Exercise 6.6.1 *Modify the interpreter with deferred substitution to handle a* lazy *language with first-class functions.*

6.7 Standardizing Terminology

We have thus far been using the terms "deferred substitution" and "repository". There is actually a standard term used for these, and we'll adopt this from now on.

Definition 12 (environment) *An* environment[1] *is a repository of deferred substitutions.*

[1] Cormac Flanagan: "Save the environment! Create a closure today!"

```
(define-type FAE
  [num (n number?)]
  [add (lhs FAE?) (rhs FAE?)]
  [id (name symbol?)]
  [fun (param symbol?) (body FAE?)]
  [app (fun-expr FAE?) (arg-expr FAE?)])

(define-type FAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body FAE?)
            (ds DefrdSub?)])

(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value FAE-Value?) (ds DefrdSub?)])

;; lookup : symbol DefrdSub → FAE-Value
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-ds)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-ds))]))

;; num+ : numV numV ⟶ numV
(define (num+ n1 n2)
  (numV (+ (numV-n n1) (numV-n n2))))
```

Figure 6.1: First-Class Functions with Deferred Substitutions: Support Code

```
;; interp : FAE DefrdSub → FAE-Value
(define (interp expr ds)
  (type-case FAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l ds) (interp r ds))]
    [id (v) (lookup v ds)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body ds)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr ds)])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr ds)
                         (closureV-ds fun-val))))]))
```

Figure 6.2: First-Class Functions with Deferred Substitutions: Interpreter

Part III

Laziness

Chapter 7

Programming with Laziness

We have seen several teasers about the difference between eager and lazy evaluation. As you may have guessed, once the language becomes sufficiently rich, this seemingly small difference does have a large impact on the language's behavior. We will now study these differences in two different contexts.

7.1 Haskell

The paradigmatic modern lazy programming language is called Haskell, in honor of Haskell Curry, who laid the foundation for a great deal of modern programming language theory. We will study the experience of programming in Haskell (using its own syntax) to get a feel for the benefits of laziness.

What follows is a partisan sample of Haskell's many wonders. It is colored by the fact that this text uses Haskell primarily to illustrate specific linguistic features, as opposed to providing a general introduction to Haskell. The Haskell language Web site[1] has references to several texts and tutorials that describe far more of the language and do so from several perspectives.

7.1.1 Expressions and Definitions

Like Scheme, simple Haskell programs do not need to be wreathed in scaffolding; and like most Scheme implementations, most Haskell implementations provide an interactive environment. These notes use one called Helium; others have a similar interface.

```
Prelude> 3
3
Prelude> True
True
```

(`Prelude>` is a Haskell prompt whose significance will soon become clear.) Haskell employs a traditional algebraic syntax for operations (with the corresponding order of precedence), with parentheses representing only grouping:

[1] http://www.haskell.org/

```
Prelude> 2*3+5
11
Prelude> 2+3*5
17
Prelude> mod 7 4
3
Prelude> mod (mod 7 4) 2
1
```

As in most programming languages other than Scheme, some built-in operators are written in infix notation while most others, including user-defined ones, are written in prefix. A prefix operator can always be used in an infix position, however, using a special syntactic convention (note that these are *back*quotes):

```
Prelude> 7 `mod` 4
3
Prelude> (7 `mod` 4) `mod` 2
1
```

and infix operators can, similarly, be treated as a procedural value, even used in a prefix position:

```
Prelude> ((<) 4 ((+) 2 3))
True
```

The latter is syntactically unwieldy; why would one need it?[2]

We have seen integers (`Int`) and booleans (`Bool`). Haskell also has characters (of type `Char`) that are written inside single-quotes: `'c'` (the character 'c'), `'3'` (the character '3'), `'\n'` (the newline character), and so on.

Based on what we've seen so far, we can begin to write Haskell functions. This one proposes a grading scale for a course:

```
scoreToLetter :: Int -> Char

scoreToLetter n
    | n > 90 = 'A'
    | n > 80 = 'B'
    | n > 70 = 'C'
    | otherwise = 'F'
```

The first line of this excerpt tells Haskell the type to expect for the corresponding definition (read `::` as "has the type"). The rest of the excerpt defines the actual function using a series of rules, akin to a Scheme conditional. Loading this definition into the Haskell evaluator makes it available to execute.[3] To test the function, we use it in the evaluator:

[2] Answer: Because we may want to use a traditionally infix operator, such as +, as an argument to another function. Think about what would happen without such a notation.

[3] In Helium, this definition must be saved in a file whose name begins with a capital letter. Helium's file functions can be accessed from the menus or, as in most other Haskell implementations, from the Haskell command-line: `:l` followed by a filename loads the definitions in the named file, `:r` reloads the file loaded most recently, and so on. The implementation manual will describe other short-cuts.

```
CS173> scoreToLetter 83
'B'
CS173> scoreToLetter 51
'F'
CS173> scoreToLetter 99
'A'
```

Note that in typical Haskell implementations, upon loading a file, the prompt changes from `Prelude` to the name of the file, indicating in which context the expression will be evaluated. The Prelude is the set of definitions built into Haskell.

7.1.2 Lists

Haskell naturally has more sophisticated types as well. As in Scheme, lists are inductively (or recursively) defined data-structures; the empty list is written `[]` and non-empty list constructor is written `:`. Haskell also offers a convenient abbreviation for lists. Thus:

```
Prelude> []
[]
Prelude> 1:[]
[1]
Prelude> 1:2:[]
[1,2]
Prelude> 1:[2,3,2+2]
[1,2,3,4]
```

Note, however, that lists must be *homogenous*: that is, all values in a list must be of the same type.

```
Prelude> [1,'a']
Type error in element of list
 expression       : [1, 'a']
 term             : 'a'
   type           : Char
   does not match : Int
```

(The exact syntax of the type error will depend on the specific Haskell implementation, but the gist should be the same. Here, Helium tells us that the second element of the list has type `Char`, whereas Helium was expecting a value of type `Int` based on the first list element.)

Haskell's Prelude has many useful functions already built in, including standard list manipulatives:

```
CS173> filter odd [1, 2, 3, 4, 5]
[1,3,5]
CS173> sum [1, 2, 3, 4, 5]
15
CS173> product [1, 2, 3, 4, 5]
120
```

We can, of course, use these in the context of our new definition:

```
CS173> map scoreToLetter [83, 51, 99]
"BFA"
CS173> length (map scoreToLetter [83, 51, 99])
3
```

It takes a little practice to know when one can safely leave out the parentheses around an expression. Eliding them in the last interaction above leads to this error:

```
CS173> length map scoreToLetter [83, 51, 99]
Type error in application
 expression       : length map scoreToLetter [83, 51, 99]
 term             : length
   type           : [a]                        -> Int
   does not match : ((b -> c) -> [b] -> [c]) -> (Int -> Char) -> [Int] -> d
 probable fix     : remove first and second argument
```

What?!? With practice (and patience), we realize that Haskell is effectively saying that `length` takes only one argument, while the use has three: `map`, `scoreToLetter` and `[83, 51, 99]`. In this case, Helium's suggestion is misleading: the fix is not to remove the arguments but rather to inform Haskell of our intent (first map the function, the determine the length of the result) with parentheses.

Suppose Haskell didn't have `length` built in. We could build it easily, using Haskell's pattern-matching notation:

```
len [] = 0
len (x:s) = 1 + len s
```

Here, the argument `(x:s)` automatically deconstructs the list, though Haskell also provides the operators `head` and `tail` for explicit manipulation. Notice, however, that we haven't written a type declaration for length. This brings us to two interesting aspects of Haskell.

7.1.3 Polymorphic Type Inference

We can ask Haskell for the type of any expression using the `:type` or `:t` directive. For instance:

```
CS173> :t 3
3 :: Int
CS173> :t True
True :: Bool
CS173> :t 3 + 4
3 + 4 :: Int
```

What should we expect when we ask Haskell for the type of `len`? Haskell responds with

```
CS173> :t len
len :: [a] -> Int
```

What does this type mean? It says that `len` consumes a list and returns an `Int`, but it says a little more. Specifically, it says that the list consumed by `len` must have elements (recall that lists are homogenous) of type... `a`. But `a` is not a concrete type like `Int` or `Bool`; rather, it is a *type variable*. Mathematically, we would write this as

$$\forall \alpha \,.\, \texttt{len} : [\alpha] \to \texttt{Int}$$

That is, α is bound to a concrete type, and remains bound to that type for a particular use of `len`; but different uses of `len` can bind α to different concrete types. We call such types *polymorphic*, and will study them in great detail in Section 29.

We can see the type parameter at work more clearly using the following (trivial) function:

```
listCopy [] = []
listCopy (x:s) = x : listCopy s
```

Haskell reports this type as

```
CS173> :t listCopy
listCopy :: [a] -> [a]
```

which is Haskell's notation for

$$\forall \alpha \,.\, \texttt{listCopy} : [\alpha] \to [\alpha]$$

When we apply `listCopy` to different argument list types, we see that it produces lists of the same type as the input each time:

```
CS173> :t listCopy [1,2,3]
listCopy [1,2,3] :: [Int]
CS173> :t listCopy ['a','b','c']
listCopy ['a','b','c'] :: [Char]
CS173> :t listCopy [[1], [1, 2], []]
listCopy [[1], [1, 2], []] :: [[Int]]
```

In the last instance, notice that we are applying `listCopy` to—and obtaining as a result—a list of type list of `Int` (i.e., a nested list of integers).

Why does Haskell assign the type parameter a name (`a`)? When there is only one parameter the name isn't necessary, but some functions are parameterized over multiple types. For instance, `map` is of this form:

```
CS173> :t map
map :: (a -> b) -> [a] -> [b]
```

which we might write with explicit quantifiers as

$$\forall \alpha, \beta \,.\, \texttt{map} : (\alpha \to \beta) \to [\alpha] \to [\beta]$$

Just from reading the type we can guess `map`'s behavior: it consumes a function that transforms each α into a corresponding β, so given a list of α's it generates the corresponding list of β's.

In the process of studying polymorphism, we may have overlooked something quite remarkable: that Haskell was able to generate types without our ever specifying them! This process is known as *type inference*. Indeed, not only is Haskell able to infer a type, it infers the *most general* type it can for each expression. We will study the machinery behind this remarkable power, too, in Section 29.

Exercise 7.1.1 *What would you expect is the type of the empty list? Check your guess using a Haskell implementation.*

Exercise 7.1.2 *Why does Haskell print the type of multi-argument functions with arrows between each adjacent pair of arguments? Experiment with Haskell by providing what appears to be a two-argument function (such as* `map`*) with only one argument.*

7.1.4 Laziness

Is Haskell eager or lazy? We can test this using a simple interaction:

```
CS173> head []
exception: Prelude.head: empty list.
```

This tells us that attempting to ask for the first element of the empty list will result in a run-time exception. Therefore, if Haskell used eager evaluation, the following expression should also result in an error:

```
CS173> (\ x -> 3) (head [])
3
```

The expression `(\ x -> 3)` uses Haskell's notation for defining an anonymous procedure: it is the syntactic analog of Scheme's (**lambda** (*x*) 3). Thus, the whole expression is equivalent to writing

((**lambda** (*x*) 3) (*first empty*))

which in Scheme would indeed result in an error. Instead, Haskell evaluates it to `3`. From this, we can posit that Haskell does not evaluate the argument until it is used, and therefore follows a lazy evaluation regime.

Why is laziness useful? Clearly, we rarely write a function that entirely ignores its argument. On the other hand, functions do frequently use different subsets of their arguments in different circumstances, based on some dynamic condition. Most programming languages offer a form of *short-circuited* evaluation for the branches of conditional (based on the value of the test expression, only one or the other branch evaluates) and for Boolean connectives (if the first branch of a disjunction yields true the second branch need not evaluate, and dually for conjunction). Haskell simply asks why this capability should not be lifted to function arguments also, and demonstrates what we get when we do.

In particular, since Haskell treats *all* function applications lazily, this also encompasses the use of most built-in constructors, such as the list constructor. As a result, when confronted with a definition such as

```
ones = 1 : ones
```

Haskell does not evaluate the second argument to `:` until necessary. When it does evaluate it, there is a definition available for `ones`: namely, a `1` followed by .... The result is therefore an infinite list, but only the act of examining the list actually constructs any prefix of it.

How do we examine an infinite list? Consider a function such as this:

```
front :: Int -> [a] -> [a]
front _ [] = []
front 0 (x:s) = []
front n (x:s) = x : front (n-1) s
```

When used, `front` causes as many list constructions of `ones` as necessary until the recursion terminates—

```
CS173> front 5 ones
[1,1,1,1,1]
CS173> front 10 ones
[1,1,1,1,1,1,1,1,1,1]
```

—*but no more*. Because the language does not force `front` to evaluate its arguments until necessary, Haskell does not construct any more of `ones` than is needed for `front` to terminate. That is, it is the act of pattern-matching that forces `ones` to grow, since the pattern-matcher must determine the form of the list to determine which branch of the function to evaluate.

Obtaining the prefix of a list of ones may not seem especially impressive, but there are many good uses for `front`. Suppose, for instance, we have a function that generates the eigenvalues of a matrix. Natural algorithms for this problem generate the values in decreasing order of magnitude, and in most applications, only the first few are meaningful. In a lazy language, we can pretend we have the entire sequence of eigenvalues, and use `front` to obtain just as many as the actual application needs; this in turn causes only that many to be computed. Indeed, any application can freely generate an infinite list of values, safe in the knowledge that a consumer can use operators such as `front` to inspect the prefix it cares about.

The function `front` is so useful when programming in Haskell that it is actually built into the Prelude, under the name `take`. Performing the same computation in an eager language is considerably more complex, because the computation that generates values and the one that consumes them must explicitly coordinate with each other: in particular, the generator must be programmed to explicitly expect requests from the consumer. This complicates the construction of the generator, which may already have complex domain-specific code; worse, if the generator was not written with such a use in mind, it is not easy to adapt it to behave accordingly.

Where else are infinite lists useful? Consider the process of generating a table of data whose rows cycle between a fixed set of colors. Haskell provides a function `cycle` that consumes a list and generates the corresponding cyclic list:

```
CS173> take 5 (cycle ["blue", "rondo"])
["blue","rondo","blue","rondo","blue"]
```

The procedure for displaying the data can consume the cyclic list and simply extract as many elements from it as necessary. The generator of the cyclic list doesn't need to know how many rows there will be in the table; laziness ensures that the entire infinite list does not get generated unless necessary. In other words, programmers often find it convenient to create cyclic data structure not so much to build a truly infinite data structure, but rather to produce one that is large enough for all possible consumers (none of which will ever examine more than a finite prefix, but each of which may want a different number of prefix elements).

Consider one more example. At the end of some stages of the Tour de France, the top finishers receive a "time bonus", which we can think of as a certain number of bonus points. Let us suppose that the top three finishers receive 20-, 12- and 8-second bonuses, respectively, while the others receive none. Given a list reflecting the order in which contestants completed a stage, we would like a list that pairs each name with the number of points that person received. That is, we would like a function `timeBonuses` such that

```
CS173> timeBonuses ["Lance", "Jan", "Tyler", "Roberto", "Iban"]
[("Lance",20),("Jan",12),("Tyler",8),("Roberto",0),("Iban",0)]
```

where `("Lance",20)` is an anonymous tuple of two elements, the first projection a string and the second a number. Note that the result is therefore a list of two-tuples (or pairs), where the heterogeneity of lists forces each tuple to be of the same type (a string in the first projection and a number in the second).

We can write `timeBonuses` by employing the following strategy. Observe that every position gets a fixed bonus (20, 12, and 8, followed by zero for everyone else), but we don't know how many finishers there will be. In fact, it isn't even clear there will be three finishers if the organizers run a particularly brutal stage! First let's create a list of all the bonuses:

```
[20, 12, 8] ++ cycle [0]
```

where ++ appends lists. We can check that this list's content matches our intuition:

```
Prelude> take 10 ([20, 12, 8] ++ cycle [0])
[20,12,8,0,0,0,0,0,0,0]
```

Now we need a helper function that will match up the list of finishers with the list of scores. Let's define this function in parts:

```
tB :: [String] -> [Int] -> [(String,Int)]
tB [] _ = []
```

Clearly, if there are no more finishers, the result must also be the empty list; we can ignore the second argument. In contrast, if there is a finisher, we want to assign him the next available time bonus:

```
tB (f:fs) (b:bs) = (f,b) : tB fs bs
```

The right-hand side of this definition says that we create an anonymous pair out of the first elements of each list (`(f,b)`), and construct a list (`:`) out of this pair and the natural recursion (`tB fs bs`).

At this point our helper function definition is complete. A Haskell implementation ought to complain that we haven't specified what should happen if the second argument is empty but the first is not:

```
(26,1): Warning: Missing pattern in function bindings:
  tBb (_ : _) [] = ...
```

This message says that the case where the first list is not empty (indicated by `(_ : _)`) and the second one is (`[]`) hasn't been covered. Since we know the second list is infinitely long, we can ignore this warning.

Given this definition of `tB`, it is now straightforward to define `timeBonuses`:

```
timeBonuses finishers =
    tB finishers ([20, 12, 8] ++ cycle [0])
```

This definition matches the test case above. We should also be sure to test it with fewer than three finishers:

```
CS173> timeBonuses ["Lance", "Jan"]
[("Lance",20),("Jan",12)]
```

The helper function `tB` is so helpful, it too (in a slightly different form) is built into the Haskell Prelude. This more general function, which terminates the recursion when the second list is empty, too, is called `zip`:

```
zip [] _ = []
zip _ [] = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

Notice that the type of `zip` is entirely polymorphic:

```
CS173> :type zip
zip :: [a] -> [b] -> [(a, b)]
```

Its name is suggestive of its behavior: think of the two lists as the two rows of teeth, and the function as the zipper that pairs them.

Haskell can equally comfortably accommodate non-cyclic infinite lists. To demonstrate this, let's first define the function `zipOp`. It generalizes `zip` by consuming an operator to apply to the pair of first elements:

```
zipOp :: (a -> b -> c) -> [a] -> [b] -> [c]
zipOp f [] _ = []
zipOp f _ [] = []
zipOp f (a:as) (b:bs) = (f a b) : zipOp f as bs
```

We can recover the `zip` operation from `zipOp` easily:[4]

```
myZip = zipOp (\ a b -> (a,b))
```

But we can also pass `zipOp` other operators, such as `(+)`:[5]

```
CS173> zipOp (+) [1, 1, 2, 3, 5] [1, 2, 3, 5, 8]
[2,3,5,8,13]
```

In fact, `zipOp` is also built into the Haskell Prelude, under the name `zipWith`.

In the sample interaction above, we are clearly beginning to build up the sequence of Fibonacci numbers. But there is an infinite number of these and, indeed, there is no reason the arguments to `zipOp` must be finite lists. Let us therefore generate the entire sequence. The code above is suggestive: clearly the first and second argument are the same list (the list of all Fibonacci numbers), but the second is the first list "shifted" by one, i.e., the tail of that list. We might therefore try to seed the process with the initial values, then use that seed to construct the remainder of the list:

```
seed = [1, 1]
output = zipOp (+) seed (tail seed)
```

[4] Recall that `(\ ···)` is Haskell's equivalent of (**lambda** ···).

[5] We have to enclose + to avoid parsing errors, since + is an infix operator. Without the parentheses, Haskell would try to add the value of `zipOp` to the list passed as the first argument.

But this produces only one more Fiboanacci number before running out of input values, i.e., `output` is bound to `[2]`. So we have made progress, but need to find a way to keep `seed` from exhausing itself. It appears that we want a way to make `seed` and `output` be the same, so that each new value computed triggers one more computation! Indeed,

```
fibs = 1 : 1 : zipOp (+) fibs (tail fibs)
```

We can test this in Haskell:

```
CS173> take 12 fibs
[1,1,2,3,5,8,13,21,34,55,89,144]
```

Sure enough `fibs` represents the entire infinite list of Fibonacci numbers, ready for further use.

Exercise 7.1.3 *Earlier, we saw the following interaction:*

```
Prelude> take 10 ([20, 12, 8] ++ cycle [0])
[20,12,8,0,0,0,0,0,0,0]
```

What happens if you instead write `take 10 [20, 12, 8] ++ cycle [0]`*? Does it result in a type error? If not, do you get the expected answer? If so, is it for the right reasons? Try this by hand before entering it into Haskell.*

Exercise 7.1.4 *The definition of the Fibonacci sequence raises the question of which "algorithm" Haskell is employing. Is it computing the* n^{th} *Fibonacci number in time linear in n (assuming constant-time arithmetic) or exponential in n?*

1. *First, try to determine this experimentally by asking for the* n^{th}*term for large values of n (though you may have trouble with arithmetic overflow).*
2. *Of course, even if you observe linear behavior, this is not* proof*; it may simply be that you didn't use a large enough value of n to observe the exponential. Therefore, try to reason about this deductively. What about Haskell will determine the computation time of the* n^{th} *Fibonacci?*

7.1.5 An Interpreter

Finally, we demonstrate an interpreter for WAE written in Haskell. First we define some type aliases,

```
type Identifier = String
type Value = Int
```

followed by the two important type definitions:

```
type Env = [(Identifier, Value)]
data WAE = Num Int
         | Add WAE WAE
         | Id Identifier
         | With Identifier WAE WAE
```

The core interpreter is defined by cases:

```
interp :: WAE -> Env -> Value
interp (Num n) env = n
interp (Add lhs rhs) env = interp lhs env + interp rhs env
interp (Id i) env = lookup i env
interp (With bound_id named_expr bound_body) env =
    interp bound_body
           (extend env bound_id (interp named_expr env))
```

The helper functions are equally straightforward:

```
lookup :: Identifier -> Env -> Value
lookup var ((i,v):r)
  | (eqString var i) = v
  | otherwise        = lookup var r

extend :: Env -> Identifier -> Value -> Env
extend env i v = (i,v):env
```

This definition of `lookup` uses Haskell's pattern-matching notation as an alternative to writing an explicit conditional. Finally, testing these yields the expected results:

```
CS173> interp (Add (Num 3) (Num 5)) []
8
CS173> interp (With "x" (Add (Num 3) (Num 5)) (Add (Id "x") (Id "x"))) []
16
```

We can comment out the type declaration for `interp` (a line beginning with two dashes (`--`) is treated as a comment), reload the file, and ask Haskell for the type of `interp`:

```
interp :: WAE -> Env -> Int
```

Remarkably, Haskell infers the same type as the one we ascribed, differing only in the use of the type alias.

Exercise 7.1.5 *Extend the Haskell interpreter to implement functions using Haskell functions to represent functions in the interpreted language. Does the resulting interpreted language have eager or lazy application? How would you make it take on the other semantics?*

7.2 Shell Scripting

While most programmers have never programmed in Haskell before, many *have* programmed in a lazy language: the language of most Unix shells. In this text we'll use the language of bash (the Bourne Again Shell), though most of these programs work identically or have very close counterparts in other popular shell languages.

The classical shell model assumes that all programs can potentially generate an infinite stream of output. The simplest such example is the program `yes`, which generates an infinite stream of `y`'s:

```
> yes
y
y
y
```

and so on, forever. (Don't try this at home without your fingers poised over the interrupt key!) To make it easier to browse the output of (potentially infinite) stream generators, Unix provides helpful applications such as `more` to page through the stream. In Haskell, function composition makes the output of one function (the equivalent of a stream-generating application) the input to another. In a shell, the `|` operator does the same. That is,

```
> yes | more
```

generates the same stream, but lets us view finite prefixes of its content in segments followed by a prompt. Quitting from `more` terminates `yes`.

What good is `yes`? Suppose you run the following command:

```
> rm -r Programs/Sources/Java
```

Say some of these files are write-protected. For each such file, the shell will generate the query

```
rm: remove write-protected file 'Programs/Sources/Java/frob.java'?
```

If you know for sure you want to delete all the files in the directory, you probably don't want to manually type `y` in response to each such question. How many can there be? Unfortunately, it's impossible to predict how many write-protected files will be in the directory. This is exactly where `yes` comes in:

```
> yes | rm -r Programs/Sources/Java
```

generates as many `y` inputs as necessary, satisfying all the queries, thereby deleting all the files.

We've seen that `more` is a useful way of examining part of a stream. But `more` is not directly analogous to Haskell's `take`. In fact, there is a Unix application that is: it's called `head`. `head` prints the first *n* entries in a stream, where *n* is given as an argument (defaulting to 10):

```
> yes | head -5
y
y
y
y
y
```

These examples already demonstrate the value of thinking of Unix programs as generators and consumers of potentially infinite streams, composed with `|`. Here are some more examples. The application `wc` counts the number of characters (`-c`), words (`-w`) and lines (`-l`) in its input stream. Thus, for instance,

```
> yes | head -5 | wc -l
       5
```

(not surprisingly). We can similarly count the number of files with suffix `.scm` in a directory:

```
> ls *.scm | wc -l
       2
```

We can compose these into longer chains. Say we have a file containing a list of grades, one on each line; say the grades (in any order in the file) are two `10`s, one `15`, one `17`, one `21`, three `5`s, one `2`, and ten `3`s. Suppose we want to determine which grades occur most frequently (and how often), in descending order.

The first thing we might do is sort the grades, using `sort`. This arranges all the grades in order. While sorting is not strictly necessary to solve this problem, it does enable us to use a very useful Unix application called `uniq`. This application eliminates adjacent lines that are identical. Furthermore, if supplied the `-c` ("count") flag, it prepends each line in the output with a count of how many adjacent lines there were. Thus,

```
> sort grades | uniq -c
       2 10
       1 15
       1 17
       1 2
       1 21
      10 3
       3 5
```

This almost accomplishes the task, except we don't get the frequencies in order. We need to sort one more time. Simply sorting doesn't do the right thing in two ways:

```
> sort grades | uniq -c | sort
       1 15
       1 17
       1 2
       1 21
       2 10
       3 5
      10 3
```

We want sorting to be numeric, not textual, and we want the sorting done in reverse (decreasing) order. Therefore:

```
> sort grades | uniq -c | sort -nr
      10 3
       3 5
       2 10
       1 21
       1 2
       1 17
       1 15
```

There is something fundamentally beautiful—and very powerful!—about the structure of the Unix shell. Virtually all Unix commands respect the stream convention, and so do even some programming languages built atop it: for instance, by default, Awk processes its input one-line-at-a-time, so the Awk program `{print $1}` prints the first field of each line, continuing until the input runs out of lines (if ever), at which point the output stream terminates. This great uniformity makes composing programs easy, thereby encouraging programmers to do it.

Alan Perlis recognized the wisdom of such a design in this epigram: "It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures" (the data structure here being the stream). The greatest shortcoming of the Unix shell is that is is so lacking in data-sub-structure, relying purely on strings, that every program has to repeatly parse, often doing so incorrectly. For example, if a directory holds a filename containing a newline, that newline will appear in the output of `ls`; a program like `wc` will then count the two lines as two different files. Unix shell scripts are notoriously fragile in these regards. Perlis recognized this too: "The string is a stark data structure and everywhere it is passed there is much duplication of process."

The heart of the problem is that the output of Unix shell commands have to do double duty: they must be readable by humans but also ready for processing by other programs. By choosing human readability as the default, the output is sub-optimal, even dangerous, for processing by programs: it's as if the addition procedure in a normal programming language always returned strings because you might *eventually* want to print an answer, instead of returning numbers (which are necessary to perform further arithmethic) and leaving conversion of numbers to strings to the appropriate input/output routine.[6]

In short, Unix shell languages are both a zenith and a nadir of programming language design. Please study their design very carefully, but also be sure to learn the right lessons from them!

[6]We might fantasize the following way of making shell scripts more robust: all Unix utilities are forced to support a `-xmlout` flag that forces them to generate output in a standard XML language that did no more than wrap tags around each record (usually, but not always, line) and each field, and a `-xmlin` flag that informs them to expect data in the same format. This would eliminate the ambiguity inherent in parsing text.

Chapter 8

Implementing Laziness

Now that we've seen Haskell and shell scripts at work, we're ready to study the implementation of laziness. That is, we will keep the *syntax* of our language unchanged, but alter the *semantics* of function application to be lazy.

8.1 Implementing Laziness

Consider the following expression:

```
  {with {x {+ 4 5}}
    {with {y {+ x x}}
      {with {z y}
        {with {x 4}
          z}}}}
```

Recall that in a lazy language, the argument to a function—which includes the named expression of a `with`—does not get evaluated until use. Therefore, we can naively think of the expression above reducing as follows:

```
  {with {x {+ 4 5}}
    {with {y {+ x x}}
      {with {z y}
        {with {x 4}
          z}}}}
= {with {y {+ x x}}
    {with {z y}
      {with {x 4}
        z}}}      [x -> {+ 4 5}]
= {with {z y}
    {with {x 4}
      z}}         [x -> {+ 4 5}, y -> {+ x x}]
```

```
= {with {x 4}
    z}            [x -> {+ 4 5}, y -> {+ x x}, z -> y]
= z               [x -> 4, y -> {+ x x}, z -> y]
= y               [x -> 4, y -> {+ x x}, z -> y]
= {+ x x}         [x -> 4, y -> {+ x x}, z -> y]
= {+ 4 4}
= 8
```

In contrast, suppose we used substitution instead of environments:

```
  {with {x {+ 4 5}}
    {with {y {+ x x}}
      {with {z y}
        {with {x 4}
          z}}}}
= {with {y {+ {+ 4 5} {+ 4 5}}}
    {with {z y}
      {with {x 4}
        z}}}
= {with {z {+ {+ 4 5} {+ 4 5}}}
    {with {x 4}
      z}}
= {with {x 4}
    {+ {+ 4 5} {+ 4 5}}}
= {+ {+ 4 5} {+ 4 5}}
= {+ 9 9}
= 18
```

We perform substitution, which means we replace identifiers whenever we encounter bindings for them, but we don't replace them only with values: sometimes we replace them with entire *expressions*. Those expressions have themselves already had all identifiers substituted.

This situation should look very familiar: this is the very same problem we encountered when switching from substitution to environments. Substitution *defines* a program's value; because environments merely defer substitution, they should not change that value.

We addressed this problem before using closures. That is, the text of a function was closed over (i.e., wrapped in a structure containing) its environment at the point of definition, which was then used when evaluating the function's body. The difference here is that we must create closures for *all* expressions that are not immediately reduced to values, so their environments can be used when the reduction to a value actually happens.

We shall refer to these new kinds of values as *expression closures*. Since they can be the result of evaluating an expression (as we will soon see), it makes sense to extend the set of values with this new kind of value. We will also assume that our language has conditionals (since they help illustrate some interesting points about laziness). Thus we will define the language CFAE/L (where the /L will denote "laziness") with the following grammar:

```
<CFAE/L> ::= <num>
           | {+ <CFAE/L> <CFAE/L>}
           | <id>
           | {fun {<id>} <CFAE/L>}
           | {<CFAE/L> <CFAE/L>}
```

Observe that the eager counterpart of this language would have the same *syntax*. The difference lies entirely in its interpretation. As before, we will continue to assume that `with` expressions are converted into immediate function applications by the parser or by a pre-processor.

For this language, we define an extended set of values:

```
(define-type CFAE/L-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAE/L?)
            (env Env?)]
  [exprV (expr CFAE/L?)
         (env Env?)])
```

That is, a exprV is just a wrapper that holds an expression and the environment of its definition.

What needs to change in the interpreter? Obviously, procedure application must change. By definition, we should not evaluate the argument expression; furthermore, to preserve static scope, we should close it over its environment:[1]

```
[app (fun-expr arg-expr)
     (local ([define fun-val (interp fun-expr env)]
             [define arg-val (exprV arg-expr env)])
       (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
                     arg-val
                     (closureV-env fun-val))))]
```

As a consequence, an expression such as

```
{with {x 3}
  x}
```

will evaluate to some expression closure value, such as

```
(exprV (num 3) (mtSub))
```

This says that the representation of the `3` is closed over the empty environment.

That may be an acceptable output for a particularly simple program, but what happens when we evaluate this one?

[1] The argument expression results in an expression closure, which we then bind to the function's formal parameter. Since parameters are bound to values, it becomes natural to regard the expression closure as a kind of value.

```
{with {x 3}
  {+ x x}}
```

The interpreter evaluates each `x` in the body to an expression closure (because that's what is bound to `x` in the environment), but the addition procedure cannot handle these: it (and similarly any other arithmetic primitive) needs to know exactly which number the expression closure corresponds to. The interpreter must therefore "force" the expression closure to reduce to an actual value. Indeed, we must do so in other positions as well: the function position of an application, for instance, needs to know which procedure to invoke. If we do not force evaluation at these points, then even a simple expression such as

```
{with {double {fun {x} {+ x x}}}
  {+ {double 5}
     {double 10}}}
```

cannot be evaluated (since at the points of application, `double` is bound to an *expression* closure, not a *procedural* closure with an identifiable parameter name and body).

Because we need to force expression closures to values in several places in the interpreter, it makes sense to write the code to do this only once:

```
;; strict : CFAE/L-Value → CFAE/L-Value [excluding exprV]
(define (strict e)
  (type-case CFAE/L-Value e
    [exprV (expr env)
           (strict (interp expr env))]
    [else e]))
```

Now we can use this for numbers,

```
(define (num+ n1 n2)
  (numV (+ (numV-n (strict n1)) (numV-n (strict n2)))))
```

and similarly in other arithmetic primitives, and also for applications:

```
[app (fun-expr arg-expr)
     (local ([define fun-val (strict (interp fun-expr env))]
             [define arg-val (exprV arg-expr env)])
       (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
                     arg-val
                     (closureV-env fun-val))))]
```

The points where the implementation of a lazy language forces an expression to reduce to a value (if any) are called the *strictness* points of the language; hence the perhaps odd name, *strict*, for the procedure that annotates these points of the interpreter.

Let's now exercise (so to speak) the interpreter's laziness. Consider the following simple example:

```
{with {f {undef x}}
  4}
```

Had the language been strict, it would have evaluated the named expression, halting with an error (that `undef` is not defined). In contrast, out interpreter yields the value `4`.

There is actually one more strictness point in our language: the evaluation of the conditional. It needs to know the precise value that the test expression evaluates to so it can determine which branch to proceed evaluating. This highlights a benefit of studying languages through interpreters: assuming we had good test cases, we would quickly discover this problem. (In practice, we might bury the strictness requirement in a helper function such as *num-zero?*, just as the arithmetic primitives' strictness is buried in procedures such as *num+*. We therefore need to trace which expression evaluations invoke *strict* primitives to truly understand the language's strictness positions.)

Figure 8.1 and Figure 8.2 present the heart of the interpreter.

Exercise 8.1.1 *Obtaining an* exprV *value as the result of evaluating a program isn't very useful, because it doesn't correspond to an answer as we traditionally know it. Modify the interpreter to always yield answers of the same sort as the eager interpreter.*
Hint: *You may find it useful to write a wrapper procedure instead of directly modifying the interpreter.*

Exercise 8.1.2 *Does laziness give us conditionals "for free"? A conditional serves two purposes: to make a decision about a value, and to avoid evaluating an unnecessary expression. Which of these does laziness encompass? Explain your thoughts with a modified interpreter.*

Exercise 8.1.3 *Interactive Haskell environments usually have one other, extra-lingual strictness point: the top-level of the Interaction window. Is this reflected here?*

8.2 Caching Computation

Evaluating an expression like

```
{with {x {+ 4 5}}
  {with {y {+ x x}}
    {with {z y}
      {with {x 4}
        z}}}}
```

can be rather wasteful: we see in the hand-evaluation, for instance, that we reduce the same expression, `{+ 4 5}`, to `9` two times. The waste arises because we bind identifiers to expressions, rather than to their values. So whereas one of our justifications for laziness was that it helped us avoid evaluating unnecessary expressions, laziness has had a very unfortunate (and unforeseen) effect: it has has forced the re-evaluation of *necessary* expressions.

Let's make a small change to the interpreter to study the effect of repeated evaluation. Concretely, we should modify *strict* to notify us every time it reduces an expression closure to a value:

```
(define (strict e)
  (type-case CFAL-Value e
    [exprV (expr env)
```

```
        (local ([define the-value (strict (interp expr env))])
          (begin
            (printf "Forcing exprV to ~a~n" the-value)
            the-value))]
    [else e]))
```

This will let us track the the amount of computation being performed by the interpreter on account of laziness. (How many times for our running example? Determine the answer by hand, then modify *strict* in the interpreter to check your answer!)

Can we do better? Of course: once we have computed the value of an identifier, instead of only using it, we can also *cache* it for future use. Where should we store it? The expression closure is a natural container: the next time we attempt to evaluate that closure, if we find a value in the cache, we can simply use that value instead.

To implement caching, we modify the interpreter as follows. First, we have to create a field for the value of the expression closure. What's the value of this field? Initially it needs to hold a dummy value, to eventually be replaced by the actual one. "Replaced" means its value needs to *change*; therefore, it needs to be a box. Concretely, we'll use the boolean value false as the initial value.

```
(define-type CFAE/L-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAE/L?)
            (env Env?)]
  [exprV (expr CFAE/L?)
         (env Env?)
         (cache boxed-boolean/CFAE/L-Value?)])
```

We define the cache's field predicate as follows:

```
(define (boxed-boolean/CFAL-Value? v)
  (and (box? v)
       (or (boolean? (unbox v))
           (numV? (unbox v))
           (closureV? (unbox v)))))
```

Notice that we carefully exclude exprV values from residing in the box. The box is meant to cache the result of strictness, which by definition and construction cannot result in a exprV. Therefore, this exclusion should never result in an error (and an indication to the contrary should be investigated).

Having changed the number of fields, we must modify all uses of the constructor. There's only one: in function application.

```
[app (fun-expr arg-expr)
     (local ([define fun-val (strict (interp fun-expr env))]
             [define arg-val (exprV arg-expr env (box false))])
       (interp (closureV-body fun-val)
               (aSub (closureV-param fun-val)
```

```
                    arg-val
                    (closureV-env fun-val))))]
```

That leaves only the definition of *strict*. This is where we actually use the cache:

```
(define (strict e)
  (type-case CFAE/L-Value e
    [exprV (expr env cache)
           (if (boolean? (unbox cache))
               (local [(define the-value (strict (interp expr env)))]
                 (begin
                   (printf "Forcing exprV ~a to ~a~n" expr the-value)
                   (set-box! cache the-value)
                   the-value))
               (begin
                 (printf "Using cached value~n")
                 (unbox cache)))]
    [else e]))
```

With these changes, we see that interpreting the running example needs to force an expression closure fewer times (how many?). The other instances reuse the value of a prior reduction. Figure 8.3 and Figure 8.4 present the heart of the interpreter. Haskell uses the value cache we have just studied, so it combines the benefit of laziness (not evaluating unnecessary arguments) with reasonable performance (evaluating the necessary ones only once).

Exercise 8.2.1 *An expression closure is extremely similar to a regular (function) closure. Indeed, if should be possible to replace the former with the latter. When doing so, however, we don't really need all the pieces of function closures: there are no arguments, so only the body and environment matter. Such a closure is called a* thunk*, a name borrowed from a reminiscent technique used in Algol 60. Implement laziness entirely using thunks, getting rid of expression closures.*

Exercise 8.2.2 *We could have achieved the same effect as using thunks (see Exercise 8.2.1) by simply using one-argument procedures with a dummy argument value. Why didn't we propose this? Put otherwise, what benefit do we derive by keeping expression closures as a different kind of value?*

Exercise 8.2.3 *Extend this language with recursion and list primitives so you can run the equivalent of the programs we saw in Section 7.1. In this extended language, implement Fibonacci, run it with and without value caching, and arrive at a conclusion about the time complexity of the two versions.*

8.3 Caching Computations Safely

Any language that caches computation (whether in an eager or lazy regime) is making a very strong tacit assumption: that an expression *computes the same value every time it evaluates*. If an expression can yield a different value in a later evaluation, then the value in the cache is corrupt, and using it in place of the correct value can cause the computation to go awry. So we must examine this evaluation decision of Haskell.

This assumption cannot be applied to most programs written in traditional languages, because of the use of side-effects. A method invocation in Java can, for instance, depend on the values of fields (directly, or indirectly via method accesses) in numerous other objects, any one of which may later change, which will almost certainly invalidate a cache of the method invocation's computed value. To avoid having to track this complex web of dependencies, languages like Java avoid caching values altogether in the general case (though an optimizing compiler may introduce a cache under certain circumstances, when it can ensure the cache's consistency).

Haskell implementations can cache values because Haskell does not provide explicit mutation operations. Haskell instead forces programmers to perform all computations by composing functions. While this may seem an onerous style to those unaccustomed to it, the resulting programs are in fact extremely elegant, and Haskell provides a powerful collection of primitives to enable their construction; we caught a glimpse of both the style and the primitives in Section 7.1. Furthermore, the lack of side-effects makes it possible for Haskell compilers to perform some very powerful optimizations not available to traditional language compilers, so what seems like an inefficient style on the surface (such as the creation of numerous intermediate tuples, lists and other data structures) often has little run-time impact.

Of course, no useful Haskell program is an island; programs must eventually interact with the world, which itself has true side-effects (at least in practice). Haskell therefore provides a set of "unsafe" operators that conduct input-output and other operations. Computations that depend on the results of unsafe operations cannot be cached. Haskell does, however, have a sophisticated type system (featuring quite a bit more, in fact, than we saw in Section 7.1) that makes it possible to distinguish between the unsafe and "safe" operations, thereby restoring the benefits of caching to at least portions of a program's computation. In practice, Haskell programmers exploit this by limiting unsafe computations to a small portion of a program, leaving the remainder in the pure style espoused by the language.

The absence of side-effects benefits not only the compiler but, for related reasons, the programmer also. It greatly simplifies reasoning about programs, because to understand what a particular function is doing a programmer doesn't need to be aware of the global flow of control of the program. In particular, programmers can study a program through *equational reasoning*, using the process of reduction we have studied in high-school algebra. The extent to which we can apply equational reasoning depends on the number of expressions we can reasonably substitute with other, equivalent expressions (including answers).

We have argued that caching computation is safe in the absence of side-effects. But the eager version of our interpreted language doesn't have side-effects either! We didn't need to cache computation in the same way we have just studied, because by definition an eager language associates identifiers with values in the environment, eliminating the possibility of re-computation on use. There is, however, a slightly different notion of caching that applies in an eager language called *memoization*.

Of course, to use memoization safely, the programmer or implementation would have to establish that the function's body does not depend on side-effects—or invalidate the cache when a relevant effect happens. Memoization is sometimes introduced automatically as a compiler optimization.

Exercise 8.3.1 *There are no lazy languages that permit mutation. Why not? Is there a deeper reason beyond the invaldation of several compiler optimizations?*

Exercise 8.3.2 *Why do you think there are no lazy languages without type systems?*
Hint: *This is related to Exercise 8.3.1.*

Referential Transparency

People sometimes refer to the lack of mutation as "referential transparency", as in, "Haskell is referentially transparent" (and, by implicit contrast, languages like Java, C++, Scheme and ML are not). What do they really mean?

Referential transparency is commonly translated as the ability to "replace equals with equals". For example, we can always replace $1+2$ with 3. Now think about that (very loose) definition for a moment: when can you *not* replace something with something else that the original thing is equal to? Never, of course—you always can. So by that definition, every language is "referentially transparent", and the term becomes meaningless.

Referential transparency really describes a *relation*: it relates pairs of terms exactly when they can be considered equivalent in all contexts. Thus, in most languages, $1+2$ is referentially transparent to 3 (assuming no overflow), and $\sqrt{4}$ (written in the appropriate notation) is referentially transparent to 2 (assuming the square root function returns only the positive root).

Given this understanding, we can now ask the following question: what is the *size of the referential transparency relation* for a program in a given language? While even a language like C subscribes a referential transparency relation, and some C programs have larger relations (because they minimize side-effects), the size of this relation is inherently larger for programs written in a language without mutation. This larger relation enables a much greater use of equational reasoning.

As a programmer, you should strive to make this relation as large as possible, no matter what language you program in: this has a positive impact on long-term program maintenance (for instance, when other programmers need to modify your code). As a student of programming languages, however, please use this term with care; in particular, always remember that it describes a relation between phrases in a program, and is rarely meaningful when applied to languages as a whole.

Memoization

Memoization associates a cache with each function. The cache tracks actual argument tuples and their corresponding return values. When the program invokes a "memoized" function, the evaluator first tries to find the function's value in the cache, and only invokes the function proper if that argument tuple hadn't been cached before. If the function is recursive, the recursive calls might also go through the memoized version. Memoization in this instance reduces the exponential number of calls in computing Fibonacci numbers to a linear number, without altering the natural recursive definition.

It's important to note that what we have implemented for lazy languages is *not* memoization. While we do cache the value of each expression closure, this is different from caching the value of all expression closures that contain the same expression closed over the same environment. In our implementation, if a program contains the same source expression (such as a function invocation) twice, each use of that expression results in a separate evaluation.

8.4 Scope and Evaluation Regimes

Students of programming languages often confuse the notions of scope (static versus dynamic) and evaluation regimes (eager versus lazy). In particular, readers often engage in the following fallacious reasoning:

> *Because* lazy evaluation substitutes expressions, not values, and *because* substituting expressions (naively) results in variables getting their values from the point of use rather than the point of definition, *therefore* lazy evaluation must result in dynamic scope.

It is very important to not be trapped by this line of thought. The scoping rules of a language are determined a priori by the language designer. (For the reasons we have discussed in Section 6.5, this should almost always be *static* scope.) It is up to the language implementor to faithfully enforce them. Likewise, the language designer determines the reduction regime, perhaps based on some domain constraints. Again, the implementor must determine how to correctly implement the chosen regime. We have seen how the use of appropriate closure values can properly enforce static scope in both eager and lazy evaluation regimes.

```
(define-type CFAE/L
  [num (n number?)]
  [add (lhs CFAE/L?) (rhs CFAE/L?)]
  [id (name symbol?)]
  [fun (param symbol?) (body CFAE/L?)]
  [app (fun-expr CFAE/L?) (arg-expr CFAE/L?)])

(define-type CFAE/L-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAE/L?)
            (env Env?)]
  [exprV (expr CFAE/L?)
         (env Env?)])

(define-type Env
  [mtSub]
  [aSub (name symbol?) (value CFAE/L-Value?) (env Env?)])

;; num+ : CFAE/L-Value CFAE/L-Value ⟶ numV
(define (num+ n1 n2)
  (numV (+ (numV-n (strict n1)) (numV-n (strict n2)))))

;; num-zero? : CFAE/L-Value → boolean
(define (num-zero? n)
  (zero? (numV-n (strict n))))

;; strict : CFAE/L-Value → CFAE/L-Value [excluding exprV]
(define (strict e)
  (type-case CFAE/L-Value e
    [exprV (expr env)
           (local ([define the-value (strict (interp expr env))])
             (begin
               (printf "Forcing exprV to ~a~n" the-value)
               the-value))]
    [else e]))
```

Figure 8.1: Implementation of Laziness: Support Code

```
;; interp : CFAE/L Env → CFAE/L-Value
(define (interp expr env)
  (type-case CFAE/L expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (strict (interp fun-expr env))]
                 [define arg-val (exprV arg-expr env)])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         arg-val
                         (closureV-env fun-val))))]))
```

Figure 8.2: Implementation of Laziness: Interpreter

```
(define-type CFAE/L-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body CFAE/L?)
            (env Env?)]
  [exprV (expr CFAE/L?)
         (env Env?)
         (cache boxed-boolean/CFAE/L-Value?)])

(define (boxed-boolean/CFAE/L-Value? v)
  (and (box? v)
       (or (boolean? (unbox v))
           (numV? (unbox v))
           (closureV? (unbox v)))))

;; strict : CFAE/L-Value → CFAE/L-Value [excluding exprV]
(define (strict e)
  (type-case CFAE/L-Value e
    [exprV (expr env cache)
           (if (boolean? (unbox cache))
               (local [(define the-value (strict (interp expr env)))]
                 (begin
                   (printf "Forcing exprV ~a to ~a~n" expr the-value)
                   (set-box! cache the-value)
                   the-value))
               (begin
                 (printf "Using cached value~n")
                 (unbox cache)))]
    [else e]))
```

Figure 8.3: Implementation of Laziness with Caching: Support Code

```
;; interp : CFAE/L Env → CFAE/L-Value
(define (interp expr env)
  (type-case CFAE/L expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (strict (interp fun-expr env))]
                 [define arg-val (exprV arg-expr env (box false))])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         arg-val
                         (closureV-env fun-val))))]))
```

Figure 8.4: Implementation of Laziness with Caching: Interpreter

Part IV

Recursion

Chapter 9

Understanding Recursion

Can we write the factorial function in FAE? We currently don't have subtraction or multiplication, or a way of making choices in our FAE code. But those two are easy to address. At this point adding subtraction and multiplication is trivial, while to make choices, we can add a simple conditional construct, leading to this language:

```
<CFAE> ::= <num>
         | {+ <CFAE> <CFAE>}
         | {* <CFAE> <CFAE>}
         | <id>
         | {fun {<id>} <CFAE>}
         | {<CFAE> <CFAE>}
         | {if0 <CFAE> <CFAE> <CFAE>}
```

An `if0` evaluates its first sub-expression. If this yields the value `0` it evaluates the second, otherwise it evaluates the third. For example,

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

evaluates to 1.

Given CFAE, we're ready to write factorial (recall from Section 6.3 that `with` can be handled by a pre-processor or by the parser):

```
{with {fac {fun {n}
             {if0 n
                  1
                  {* n {fac {+ n -1}}}}}}}
  {fac 5}}
```

What does this evaluate to? 120? No. Consider the following simpler expression, which you were asked to contemplate when we studied substitution:

```
{with {x x} x}
```

In this program, the `x` in the named expression position of the `with` has no binding. Similarly, the environment in the closure bound to `fac` binds no identifiers. Therefore, only the environment of the first invocation (the body of the `with`) has a binding for `fac`. When `fac` is applied to `5`, the interpreter evaluates the body of `fac` in the closure environment, which has no bound identifiers, so interpretation stops on the intended recursive call to `fac` with an unbound identifier error. (As an aside, notice that this problem disappears with *dynamic* scope! This is why dynamic scope persisted for as long as it did.)

Before you continue reading, please pause for a moment, study the program carefully, write down the environments at each stage, step by hand through the interpreter, even run the program if you wish, to convince yourself that this error will occur. Understanding the error thoroughly will be essential to following the remainder of theis section.

9.1 A Recursion Construct

It's clear that the problem arises from the scope rules of `with`: it makes the new binding available only in its body. In contrast, we need a construct that will make the new binding available to the named expression also. Different intents, so different names: Rather than change `with`, let's add a new construct to our language, `rec`.

```
<RCFAE> ::= <num>
          | {+ <RCFAE> <RCFAE>}
          | {* <RCFAE> <RCFAE>}
          | <id>
          | {fun {<id>} <RCFAE>}
          | {<RCFAE> <RCFAE>}
          | {if0 <RCFAE> <RCFAE> <RCFAE>}
          | {rec {<id> <RCFAE>} <RCFAE>}
```

RCFAE is CFAE extended with a construct for recursive binding. We can use `rec` to write a description of factorial as follows:

```
{rec {fac {fun {n}
             {if0 n
                  1
                  {* n {fac {+ n -1}}}}}}}
  {fac 5}}
```

Simply defining a new syntactic construct isn't enough; we must also describe what it means. Indeed, notice that syntactically, there is nothing but the keyword distinguishing `with` from `rec`. The interesting work lies in the interpreter. But before we get there, we'll first need to think hard about the semantics at a more abstract level.

9.2 Environments for Recursion

It's clear from the analysis of our failed attempt at writing `fac` using `with` that the problem has something to do with environments. Let's try to make this intuition more precise.

One way to think about constructs such as `with` is as *environment transformers*. That is, they are functions that consume an environment and transform it into one for each of their sub-expressions. We will call the environment they consume—which is the one active outside the use of the construct—the *ambient* environment.

There are two transformers associated with `with`: one for its named expression, and the other for its body. Let's write them both explicitly.

$\rho_{\texttt{with},named}(e) = e$

In other words, whatever the ambient environment for a `with`, that's the environment used for the named expression. In contrast,

$\rho_{\texttt{with},body}(e) =$ (aSub *bound-id*
bound-value
e)

where *bound-id* and *bound-value* have to be replaced with the corresponding identifier name and value, respectively.

Now let's try to construct the intended transformers for `rec` in the factorial definition above. Since `rec` has two sub-expressions, just like `with`, we will need to describe two transformers. The body seems easier to tackle, so let's try it first. At first blush, we might assume that the body transformer is the same as it was for `with`, so:

$\rho_{\texttt{rec},body}(e) =$ (aSub 'fac
(closureV $\cdots$)
e)

Actually, we should be a bit more specific than that: we must specify the environment contained in the closure. Once again, if we had a `with` instead of a `rec`, the closure would close over the ambient environment:

$\rho_{\texttt{rec},body}(e) =$ (aSub 'fac
(closureV 'n ;; bound id
(if0 $\cdots$) ;; body
e)
e)

But this is no good! When the `fac` procedure is invoked, the interpreter is going to evaluate its body in the environment bound to *e*, which doesn't have a binding for `fac`. So this environment is only good for the first invocation of `fac`; it leads to an error on subsequent invocations.

Let's understand how this closure came to have that environment. The closure simply closes over whatever environment was active at the point of the procedure's definition. Therefore, the real problem is making sure we have the right environment for the named-expression portion of the `rec`. If we can do that, then the procedure in that position would close over the right environment, and everything would be set up right when we get to the body.

We must therefore shift our attention to the environment transformer for the named expression. If we evaluate an invocation of `fac` in the ambient environment, it fails immediately because `fac` isn't bound. If we evaluate it in (aSub 'fac (closureV $\cdots$ *e*) *e*), we can perform one function application before we halt with an error. What if we wanted to be able to perform two function calls (i.e., one to initiate the computation, and one recursive call)? Then the following environment would suffice:

```
ρ_rec,named(e) =
  (aSub 'fac
        (closureV 'n
                  (if0 ···) ;; body
                  (aSub 'fac
                        (closureV 'n
                                  (if0 ···) ;; body
                                  e)
                        e))
        e)
```

That is, when the body of `fac` begins to evaluate, it does so in the environment

```
(aSub 'fac
      (closureV 'n
                (if0 ···) ;; body
                e)
      e)
```

which contains the "seed" for one more invocation of `fac`. That second invocation, however, evaluates its body in the environment bound to *e*, which has no bindings for `fac`, so any further invocations would halt with an error.

Let's try this one more time: the following environment will suffice for one initial and *two* recursive invocations of `fac`:

```
ρ_rec,named(e) =
  (aSub 'fac
        (closureV 'n
                  (if0 ···) ;; body
                  (aSub 'fac
                        (closureV 'n
                                  (if0 ···) ;; body
                                  (aSub 'fac
                                        (closureV 'n
                                                  (if0 ···) ;; body
                                                  e)
                                        e))
                        e))
        e)
```

There's a pattern forming here. To get true recursion, we need to not "bottom out", which happens when we run out of extended environments. This would not happen if the "bottom-most" environment were somehow to refer back to the one enclosing it. If it could do so, we wouldn't even need to go to three levels; we only need one level of environment extension: the place of the boxed e in

$\rho_{\texttt{rec},named}(e) =$
  (aSub 'fac
    (closureV 'n
      (if0 ···) ;; body
      $\boxed{e}$)
    e)

should instead be a reference to the entire right-hand side of that definition. The environment must be a *cyclic* data structure, or one that refers back to itself.

We don't seem to have an easy way to represent this environment transformer, because we can't formally just draw an arrow from the box to the right-hand side. However, in such cases we can use a variable to name the box's location, and specify the constraint externally (that is, once again, name and conquer). Concretely, we can write

$\rho_{\texttt{rec},named}(e) =$
  (aSub 'fac
    (closureV 'n
      (if0 ···) ;; body
      E)
    e)

But this has introduced an unbound (free) identifier, E. It's easy to make this bound, by introducing a new (helper) function:

$\rho'(e) =$
  $\lambda\ E$.
    (aSub 'fac
      (closureV 'n
        (if0 ···) ;; body
        E)
      e)

We'll call this function, ρ', a *pre-transformer*, because it consumes both e, the ambient environment, and E, the environment to put in the closure. For some ambient environment e_0, let's set

$$F_{e_0} = \rho'(e_0)$$

Observe that F_{e_0} is a procedure ready to consume an E, the environment to put in the closure. What does it return? It returns an environment that extends the ambient environment. If we feed the right environment for E, then recursion can proceed forever. What E will enable this?

Whatever we get by feeding some E_0 to F_{e_0}—that is, $F_{e_0}(E_0)$—is precisely the environment that will be bound in the closure in the named expression of the `rec`, by definition of F_{e_0}. We also want that the

environment we get back one that extends the ambient environment with a suitable binding for `fac`, i.e., the *same* environment. In short, we want

$$E_0 = F_{e_0}(E_0)$$

That is, the environment E_0 we need to feed F_{e_0} needs to be the same as the environment we will get from applying F_{e_0} to it. We're being asked to supply the very answer we want to produce!

We call such a value—one such that the function's output is the same as its input—a *fixed-point* of a function. In this particular case, the fixed-point of ρ' is an environment that extends the ambient environment with a binding of a name to a closure whose environment is ... itself.

Exercise 9.2.1 *This discussion about recursion has taken place in the context of environments, i.e., deferred substitutions. How would it differ if we were performing explicit substitution (i.e., without deferral)?*

Fixed-Points

Consider functions over the real numbers. The function $f(x) = 0$ has exactly one fixed point, because $f(n) = n$ only when $n = 0$. But not all functions over the reals have fixed points: consider $f(x) = x + 1$. A function can have two fixed points: $f(x) = x^2$ has fixed points at 0 and 1 (but not, say, at -1). And because a fixed point occurs whenever the graph of the function intersects the line $y = x$, the function $f(x) = x$ has infinitely many fixed points.

The study of fixed points over topological spaces is fascinating, and yields many rich and surprising theorems. One of these is the Brouwer fixed point theorem. The theorem says that every continuous function from the unit n-ball to itself must have a fixed point. A famous consequence of this theorem is the following result. Take two instances of the same map, align them, and lay them flat on a table. Now crumple the upper copy, and lay it atop the smooth map any way you like (but entirely fitting within it). No matter how you place it, at least one point of the crumpled map lies directly above its equivalent point on the smooth map!

The mathematics we must use to carefully define this fixed-point is not trivial. Fortunately for us, we're using programming instead of mathematics! In the world of programming, the solution will be to generate a *cyclic* environment.

Recursiveness and Cyclicity

It is important to distinguish between *recursive* and *cyclic* data. A recursive object contains references to instances of objects of the same kind as it. A cyclic object doesn't just contains references to objects of the same *kind* as itself: it contains references to *itself*.

To easily recall the distinction, think of a typical family tree as a canonical recursive structure. Each person in the tree refers to two more family trees, one each representing the lineage of their mother and father. However, nobody is (usually) their own ancestor, so a family tree is never cyclic. Therefore, structural recursion over a family tree will always terminate. In contrast, the Web is not merely recursive, it's cyclic: a Web page can refer to another page which can refer back to the first one (or, a page can refer to itself). Naïve recursion over a cyclic datum will potentially not terminate: the recursor needs to either not try traversing the entire object, or must track which nodes it has already visited and accordingly prune its traversal. Web search engines face the challenge of doing this efficiently.

9.3 An Environmental Hazard

When programming with `rec`, we have to be extremely careful to avoid using bound values prematurely. Specifically, consider this program:

```
{rec {f f}
  f}
```

What should this evaluate to? The `f` in the body has whatever value the `f` did in the named expression of the `rec`—whose value is unclear, because we're in the midst of a recursive definition. An implementation could give you an internal value that indicates that a recursive definition is in progress; it could even go into an infinite loop, as `f` tries to look up the definition of `f`, which depends on the definition of `f`, which....

There is a safe way out of this pickle. The problem arises because the named expression can be any complex expression, including the identifier we are trying to bind. But recall that we went to all this trouble to create recursive *procedures*. If we merely wanted to bind, say, a number, we have no need to write

```
{rec {n 5}
  {+ n 10}}
```

when we could write

```
{with {n 5}
  {+ n 10}}
```

just as well instead. Therefore, instead of the liberal syntax for RCFAE above, we could use a more conservative syntax that restricts the named expression in a `rec` to *syntactically* be a procedure (i.e., the programmer may only write named expressions of the form `{proc ...}`). Then, interpretation of the named expression immediately constructs a closure, and the closure can't be applied until we interpret the body—by which time the environment is in a stable state.

Exercise 9.3.1 *Are there any other expressions we can allow, beyond just syntactic procedures, that would not compromise the safety of this conservative recursion regime?*

Exercise 9.3.2 *Can you write a useful or reasonable program that is permitted in the liberal syntax, and that safely evaluates to a legitimate value, that the conservative syntax prevents?*

Chapter 10

Implementing Recursion

We have now reduced the problem of creating recursive functions to that of creating cyclic environments.

The interpreter's rule for `with` in Figure 5.2 was as follows:

```
[with (bound-id named-expr bound-body)
     (interp bound-body
           (aSub bound-id
                 (interp named-expr
                       ds)
                 ds))]
```

It is tempting to write something similar for `rec`, perhaps making a concession for the recursive environment by using a different constructor:

```
[rec (bound-id named-expr bound-body)
     (interp bound-body
           (recSub bound-id
                 (interp named-expr
                       ds)
                 ds))]
```

Unfortunately, this suffers from a fatal flaw. The problem is that it interprets the named expression in the environment *ds*. We have decided in Section 9.3 that the named expression must syntactically be a `fun` (using, say, the parser to enforce this restriction), which means its value is going to be a closure. That closure is going to capture its environment, which in this case will be *ds*, the ambient environment. But *ds* doesn't have a binding for the identifier being bound by the `rec` expression, which means the function won't be recursive. So this attempt cannot succeed.

Rather than hasten to evaluate the named expression, we could pass the pieces of the function to the procedure that will create the recursive environment. When it creates the recursive environment, it can generate a closure for the named expression that closes over this recursive environment. In code,

```
[rec (bound-id named-expr bound-body)
     (interp bound-body
```

```
(cyclically-bind-and-interp bound-id
                            named-expr
                            env))]
```

(Recall that *ds* is the old name for *env*.) This puts the onus on *cyclically-bind-and-interp*, but hopefully also gives it the pieces it needs to address the problem. That procedure is expected to create and return the appropriate environment, which associates the bound identifier with a closure whose environment is the containing environment.

Let's turn our attention to *cyclically-bind-and-interp*. First, let's make a note of its contract:

```
;; cyclically-bind-and-interp : symbol fun env → env
```

(Section 9.3 explains why the second argument should be a fun and not any other kind of expression.)

Before we can create a cyclic environment, we must first extend it with a binding for the new identifier. At this point we know the identifier's name but not necessarily the value bound to it, so we'll place a dummy value in the environment:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (numV 1729)]
          [define new-env (aSub bound-id value-holder env)])
    ...))
```

If the program uses the identifier being bound before it has its real value, it'll get the dummy value as the result. But because we have assumed that the named expression is syntactically a function, this can't happen.

Now that we have this extended environment, we can interpret the named expression in it:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (numV 1729)]
          [define new-env (aSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    ...))
```

The named expression evaluates to a closure, which will close over the extended environment (*new-env*). Notice that this environment is half-right and half-wrong: it has the right names bound, but the newest addition is bound to the wrong (dummy) value.

Now comes the critical step. The value we get from evaluating the named expression is the same value we want to get on all subsequent references to the name being bound. (We didn't have this value before, which is why we had to place a dummy value in the environment.) Therefore, the dummy value—the one bound to the identifier named in the `rec`—needs to be *replaced* with the new value.

To perform this replacement, we need to ensure that the environment is *mutable*. To make it mutable, we must use a different kind of value in the environment, ideally a Scheme *box*.[1] Unfortunately, using a box rather than a *RCFAE-Value?* would violate the contract for aSub. Therefore, we will need to add a new variant to the datatype for environments; let's call it an aRecSub, to commemorate the construct that motivated its introduction. An aRecSub is just like an aSub, except its *value* must satisfy this predicate:

[1] A Scheme box is a mutable cell. Boxes have three operations: *box* : *Value* → *box*, which creates a fresh cell containing the argument value; *unbox* : *box* → *Value*, which returns the value stored in a box; and *set-box!* : *box Value* → *void*, which changes the value held in a box but returns no value of interest.

```
(define (boxed-RCFAE-Value? v)
  (and (box? v)
       (RCFAE-Value? (unbox v))))
```

Consequently, the proper definition of an environment is

```
(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (value RCFAE-Value?)
        (env Env?)]
  [aRecSub (name symbol?)
           (value boxed-RCFAE-Value?)
           (env Env?)])
```

Now we have to rewrite the code we've written so far, using aRecSub instead of aSub, and boxing the dummy value:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    ...))
```

Now that we have a box in the environment, it's ripe for mutation:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    (set-box! value-holder named-expr-val)))
```

Since any closures in the value expression *share the same binding*, they automatically avail of this update. Finally, we must remember that *cyclically-bind-and-interp* has to actually return the updated environment for the interpreter to use when evaluating the body:

```
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    (begin
      (set-box! value-holder named-expr-val)
      new-env)))
```

There's one last thing we need to do. Because we have introduced a new kind of environment, we must update the environment lookup procedure to recognize it.

```
[aRecSub (bound-name boxed-bound-value rest-env)
```

```
      (if (symbol=? bound-name name)
          (unbox boxed-bound-value)
          (lookup name rest-env))]
```

This only differs from the rule for aSub in that we must remember that the actual value is encapsulated within a box. Figure 10.1 and Figure 10.2 present the resulting interpreter.

Working through our factorial example from earlier, the ambient environment is (mtSub), so the value bound to *new-env* in *cyclically-bind-and-interp* is

```
(aRecSub 'fac
         (box (numV 1729))
         (mtSub))
```

Next, *named-expr-val* is bound to

```
(closureV 'n
          (if0 ...)
          (aRecSub 'fac
                   (box (numV 1729))
                   (mtSub)))
```

Now the mutation happens. This has the effect of changing the value bound to 'fac in the environment:

```
(aRecSub 'fac
         (box (closureV ...))
         (mtSub))
```

But we really should be writing the closure out in full. Now recall that this is the *same* environment contained in the closure bound to 'fac. So the environment is really

```
(aRecSub 'fac
         (box (closureV 'n
                        (if0 ...)
                        □))
         (mtSub))
```

where □ is a reference back to this very same environment! In other words, we have a cyclic environment that addresses the needs of recursion. The cyclicity ensures that there is always "one more binding" for `fac` when we need it.

Exercise 10.0.3 *Were we able to implement recursive definitions in* F1WAE *(Section 4)? If so, how was that possible without all this machinery?*

Exercise 10.0.4 *Lift this restriction on the named expression. Introduce a special kind of value that designates "there's no value here (yet)"; when a computation produces that value, the evaluator should halt with an error.*

```
(define-type RCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body RCFAE?)
            (env Env?)])

(define (boxed-RCFAE-Value? v)
  (and (box? v)
       (RCFAE-Value? (unbox v))))

(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (value RCFAE-Value?)
        (env Env?)]
  [aRecSub (name symbol?)
           (value boxed-RCFAE-Value?)
           (env Env?)])

;; lookup : symbol env → RCFAE-Value
(define (lookup name env)
  (type-case Env env
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-env))]
    [aRecSub (bound-name boxed-bound-value rest-env)
             (if (symbol=? bound-name name)
                 (unbox boxed-bound-value)
                 (lookup name rest-env))]))

;; cyclically-bind-and-interp : symbol RCFAE env → env
(define (cyclically-bind-and-interp bound-id named-expr env)
  (local ([define value-holder (box (numV 1729))]
          [define new-env (aRecSub bound-id value-holder env)]
          [define named-expr-val (interp named-expr new-env)])
    (begin
      (set-box! value-holder named-expr-val)
      new-env)))
```

Figure 10.1: Recursion: Support Code

```
;; interp : RCFAE env → RCFAE-Value
(define (interp expr env)
  (type-case RCFAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env) (interp r env))]
    [mult (l r) (num* (interp l env) (interp r env))]
    [if0 (test truth falsity)
         (if (num-zero? (interp test env))
             (interp truth env)
             (interp falsity env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV bound-id bound-body env)]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)])
           (interp (closureV-body fun-val)
                   (aSub (closureV-param fun-val)
                         (interp arg-expr env)
                         (closureV-env fun-val))))]
    [rec (bound-id named-expr bound-body)
         (interp bound-body
                 (cyclically-bind-and-interp bound-id
                                             named-expr
                                             env))]))
```

Figure 10.2: Recursion: Interpreter

Part V

Intermezzo

Chapter 11

Representation Choices

Having grown comfortable with environments, let's try to get to the essence of what an environment is. This will have an interesting (and perhaps surprising) implication for their representation, which in turn will lead us to a deeper investigation of representations in general.

11.1 Representing Environments

We have seen one way of implementing the environment, which is as a list- or stack-like datatype:

```
(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (value FAE-Value?)
        (env Env?)])
```

The environment, as we've noted, is a mapping from identifiers to values. But it's a particular kind of mapping: whenever we look up the value of an identifier, we want to get at most one value for it. That is, the environment is just a (partial) function.

In a language like Scheme, we can implement a partial function directly using Scheme's functions, without having to go through a data structure representation. First, let's define a predicate for our new representation:

```
(define (Env? x)
  (procedure? x))
```

This predicate is not exact, but it'll suffice for our purposes. Using this representation, we have a different way of implementing aSub (the contract stays the same):

```
(define (aSub bound-name bound-value env)
  (lambda (want-name)
    (cond
      [(symbol=? want-name bound-name) bound-value]
      [else (lookup want-name env)])))
```

The function aSub must return an environment and, since we've chosen to represent environments by Scheme functions (**lambda**s), aSub must return a function. This explains the **lambda**.

An environment is a function that consumes one argument, a *want-name*, the name we're trying to look up. It checks whether the name that name is the one bound by the current procedure. If it is, it returns the bound value, otherwise it continues the lookup process. How does that work?

```
(define (lookup name env)
  (env name))
```

A environment is just a procedure expecting an identifier's name, so to look up a name, we simply apply it to the name we're looking for.

The implementation of the initial value, mtSub, is simply a function that consumes an identifier name and halts in error:

```
(define (mtSub)
  (lambda (name)
    (error 'lookup "no binding for identifier")))
```

These changes are summarized in Figure 11.1. Given these changes, the core interpreter remains *unchanged* from Figure 6.2.

11.2 Representing Numbers

Let's consider our representation of numbers. We made the decision to represent FAE numbers as Scheme numbers. Scheme numbers handle overflow automatically by growing as large as necessary. If we want to have FAE numbers behave differently—for instance, by overflowing like Java's numbers do—we would need to use modular arithmetic that captures our desired overflow modes, instead of directly mapping the operators in FAE to those in Scheme.

Because numbers are not as interesting as some of the other features we'll be studying, we won't be conducting such an exercise. The relevant point is that when writing an interpreter, we get the power to make these kinds of choices. A related choice, which *is* relevant to this text, is the representation of functions.

11.3 Representing Functions

What other representations are available for FAE functions (i.e., `fun` expressions)? Currently, our interpreter uses a datatype. We might try to use strings or vectors; vectors would gain little over a datatype, and it's not quite clear how to use a string. One Scheme type that *ought* to be useful, however, is Scheme's own procedure mechanism, **lambda**. Let's consider how that might work.

First, we need to change our representation of function values. We will continue to use a datatype, but only to serve as a wrapper of the actual function representation (just like the numV clause only wraps the actual number). That is,

```
(define-type FAE-Value
  [numV (n number?)]
  [closureV (p procedure?)])
```

We will need to modify the fun clause of the interpreter. When we implemented environments with procedures, we embedded a variant of the original lookup code in the redefined aSub. Here we do a similar thing: we want FAE function application to be implemented with Scheme procedure application, so we embed the original app code inside the Scheme procedure representing a FAE function.

```
[fun (bound-id bound-body)
     (closureV (lambda (arg-val)
                       (interp bound-body
                               (aSub bound-id arg-val env))))]
```

That is, we construct a closureV that wraps a real Scheme closure. That closure takes a single value, which is the value of the actual parameter. It then interprets the body in an extended environment that binds the parameter to the argument's value.

These changes should immediately provoke two important questions:

1. *Which environment will the interpreter extend when evaluating the body?* Because Scheme itself obeys static scoping, the interpreter will automatically employ the environment active at procedure *creation*. That is, Scheme's **lambda** does the hard work so we can be sure to get the right cache.

2. *Doesn't the body get interpreted when we define the function?* No, it doesn't. It only gets evaluated when something—hopefully the application clause of the interpreter—extracts the Scheme procedure from the closureV value and applies it to the value of the actual parameter.

Correspondingly, application becomes

```
[app (fun-expr arg-expr)
     (local ([define fun-val (interp fun-expr env)]
             [define arg-val (interp arg-expr env)])
       ( (closureV-p fun-val)
         arg-val))]
```

Having reduced the function and argument positions to values, the interpreter extracts the Scheme procedure that represents the function (the boxed expression), and applies it to the argument value.

In short, a `fun` expression now evaluates to a Scheme procedure that takes a FAE value as its argument. Function application in FAE is now just procedure application in Scheme. Figure 11.2 presents the entire revised interpreter.

11.4 Types of Interpreters

We have seen a few different implementations of interpreters that are quite different in flavor. They suggest the following taxonomy.

Definition 13 (syntactic interpreter) *A syntactic interpreter is one that uses the interpreting language to represent only* terms *of the interpreted language, implementing all the corresponding behavior explicitly.*

Definition 14 (meta interpreter) *A meta interpreter is an interpreter that uses language features of the interpreting language to* directly implement behavior *of the interpreted language.*

While our substitution-based FAE interpreter was very nearly a syntactic interpreter, we haven't written any purely syntactic interpreters so far: even that interpreter directly relied on Scheme's implementation of numbers. The interpreter in Figure 11.2, which employs Scheme's **lambda** (and its attendant static scope) to represent `fun`, is distinctly a meta interpreter.

With a good match between the interpreted language and the interpreting language, writing a meta interpreter can be very easy. With a bad match, though, it can be very hard. With a syntactic interpreter, implementing *each* semantic feature will be somewhat hard,[1] but in return you don't have to worry as much about how well the interpreting and interpreted languages correspond. In particular, if there is a particularly strong *mismatch* between the interpreting and interpreted language, it may take less effort to write a syntactic interpreter than a meta interpreter.

As an example, consider the implementation of laziness. Suppose we use Scheme closures as the representation of functions, as in Figure 11.2. Function application in this language automatically becomes eager, "inheriting" this behavior from Scheme's eager evaluation semantics. If we instead wanted lazy evaluation, we would have to expend some effort to "undo" the behavior inherited from Scheme and make application lazy. Worse, we would be inheriting any subtleties that Scheme's closure and application semantics might posess.[2] In contrast, the relatively syntactic interpreter given for laziness in Section 8 does not suffer from this peril.

Based on this discussion, we can now understand when it is and isn't reasonable to exploit Scheme's representations, which may have seemed arbitrary until now. It is reasonable for features not under study (such as numbers), but unreasonable for features directly under examination (such as function application, when we're studying functions—whether eager or lazy). Once we've provided a syntactic interpreter to explain a feature, such as application and recursion, we can then exploit that feature in Scheme to build the next level of complexity.

As an exercise, we can build upon our latest interpreter to remove the encapsulation of the interpreter's response in the FAE-Value type. The resulting interpreter is shown in Figure 11.3. This is a true meta interpreter: it uses Scheme closures to implement FAE closures, Scheme procedure application for FAE function application, Scheme numbers for FAE numbers, and Scheme arithmetic for FAE arithmetic. In fact, ignoring some small syntactic differences between Scheme and FAE, this latest interpreter can be classified as something more specific than a meta interpreter:

Definition 15 (meta-circular interpreter) *A meta-circular interpreter is a meta interpreter in which the interpreting and interpreted language are the same.*

(Put differently, the trivial nature of the interpreter clues us in to the deep connection between the two languages, whatever their syntactic differences may be.)

[1] Though a poor choice of meta language can make this much harder than necessary! We choose Scheme in part because it has so many powerful features to draw upon in a meta interpreter.

[2] While Scheme has few dark corners, some versions of popular scripting languages have non-standard semantics for standard constructs such as first-class functions. A meta interpreter that used these constructs directly would then inherit these dark corners, probably inadvertently.

Meta-circular interpreters do very little to promote understanding. To gain an understanding of the language being interpreted—one of the reasons we set out to write interpreters—we must already thoroughly understand that language so we can write or understand the interpreter! Therefore, meta-circular interpreters are elegant but not very effective educational tools (except, perhaps, when teaching one particular language). In this text, therefore, we will write interpreters that balance between syntactic and meta elements, using only those meta features that we have already understood well (such as Scheme closures). This is the only meta-circular interpreter we will write.

That said, meta-circular interpreters do serve in one very important role: they're good at finding weaknesses in language definitions! For instance, if you define a new scripting language, no doubt you will put great effort into the design of its domain-specific features, such as those to parse data files or communicate over a network. But will you get the domain-independent parts—procedures, scope, etc.—right also? And how can you be sure? One good way is to try and write a meta, then meta-circular interpreter using the language. You will probably soon discover all sorts of deficiencies in the core language. The failure to apply this simple but effective experiment is partially responsible for the messy state of so many scripting languages (Tcl, Perl, JavaScript, Python, etc.) for so long; only now are they getting powerful enough to actually support effective meta-circular interpreters.

In short, by writing a meta-circular interpreter, you are likely to find problems, inconsistencies and, in particular, weaknesses that you hadn't considered before. In fact, some people would argue that *a truly powerful language is one that makes it easy to write its meta-circular interpreter*.

Exercise 11.4.1 *It is instructive to extend the Haskell interpreter given in Section 7.1 to implement recursion. Use the data structure representation of the environment. In Section 10, this required mutation. Haskell does not provide a mutation operation. Without it, are you able to implement recursion?*

11.5 Procedural Representation of Recursive Environments

Section 11.1 introduced a second, procedural, representation of environments. Section 10 discussed the implementation of recursion using a data structure representation of environments. We should therefore consider whether we can implement recursion using the procedural representation.

Figure 10.2 presents a core interpreter that is relatively independent of the representation of environments. To enable recursion, we simply need to provide an implementation of the key helper function:

;; cyclically-bind-and-interp : symbol fun env → env

which must begin as follows:

(**define** (*cyclically-bind-and-interp bound-name named-expr env*)
  ⋯)

We know that the following code pattern must exist because of the nature of the procedural representation of environments:

(**define** (*cyclically-bind-and-interp bound-name named-expr env*)
  ⋮
  (**lambda** (*want-name*)

```
    (cond
      [(symbol=? want-name bound-name)
       ...]
      [else (lookup want-name env)]))
  ...)
```

If the symbols match, what do we want to return? Looking up an identifier in an environment produces values. Recall that the named expression must be a function, so its value must be a closure. Thus, the response if the symbols match must yield a closure:

```
(define (cyclically-bind-and-interp bound-name named-expr env)
  ⋮
  (lambda (want-name)
    (cond
      [(symbol=? want-name bound-name)
       (closureV (fun-param named-expr)
                 (fun-body named-expr)
                 ...)]
      [else (lookup want-name env)]))
  ...)
```

What's not yet clear is what environment to close over. It clearly can't be just *env*; it must also contain this additional binding. So how about we give a name to this new environment that knows about the binding for *bound-name*?

```
(define (cyclically-bind-and-interp bound-name named-expr env)
  (local ([define rec-ext-env
            (lambda (want-name)
              (cond
                [(symbol=? want-name bound-name)
                 (closureV (fun-param named-expr)
                           (fun-body named-expr)
                           ...)]
                [else (lookup want-name env)]))])
    ...))
```

Having named it, it's now easy to fill in the two ellipses. What environment do we want to close over in the closure? One that binds the function named in *bound-name* to the appropriate closure. This is the environment *rec-ext-env*. What do we want to return from this procedure? The recursively extended environment. This is also *rec-ext-env*. Thus, ignoring the box momentarily,

```
(define (cyclically-bind-and-interp bound-name named-expr env)
  (local ([define rec-ext-env
            (lambda (want-name)
              (cond
                [(symbol=? want-name bound-name)
```

```
                (closureV (fun-param named-expr)
                          (fun-body named-expr)
                          [rec-ext-env])]
               [else (lookup want-name env)]))])
    rec-ext-env))
```

The relevant portions of the interpreter are in Figure 10.2 and Figure 11.4. Notice that all the code common to Figure 11.4 and Figure 11.1 is identical.

Exercise 11.5.1 *One difference between Figure 11.1 and Figure 11.4 is the addition of recursive environment bindings. When we added support for recursive bindings in Section 10, we had to modify lookup. Why didn't lookup change between Figure 11.1 and Figure 11.4?*

This definition raises two natural questions:

1. *Is this really a recursive environment?* Yes it is, though you'll just have to take the word of the authors of DrScheme that **local** does indeed define *rec-ext-env* as a *recursive* procedure, so references to that name in the procedure's body will indeed refer back to the same procedure.

2. *Doesn't the boxed reference to rec-ext-env have the same problem we were trying to avoid with expressions such as* `{rec {x x} x}`*?* Actually, it doesn't. The reference here is "under a lambda", that is, it is separated from the binding instance by a procedure declaration. Therefore, when the named expression portion of the **local** is evaluated, it associates a closure with *rec-ext-env* that doesn't get invoked until much later—by which time the recursive environment of the **local** is safely defined. This is the same issue we discussed in Section 9.3.

Reassuring as these responses may be, there is still something deeply unsatisfying about this solution. We set out to add recursive functions to RCFAE. We reduced this to the problem of defining recursive environments, which is legitimate (and, arguably, recursive environments are easier to think about than recursive functions themselves). But we then *implemented* recursive environments by falling right back on Scheme's recursive functions: an abuse of meta-interpretive power, if ever there was any! For this reason, the syntactic interpreter given in Section 10 is superior: it doesn't rely on advanced knowledge of Scheme (or, at least, no knowledge of features that we don't also find in more mainstream programming languages).

As an aside, this discussion highlights both a power and peril of meta-interpretive choices. The power of choosing the procedural representation is that we can add recursion to the language very easily. If our goal is to add it as quickly as possible, while minimizing error, it makes sense to exploit the effort put into implementing recursion for Scheme. But the peril is that this implementation does not hold descriptive power: it still begs the question of how to implement recursion from scratch.

Exercise 11.5.2 *Is it possible to implement recursive environments using the procedural representation* without *employing Scheme's constructs for creating recursive procedures? That is, can* FAE *alone express recursive functions?*

Exercise 11.5.3 *The two implementations differ slightly in the way they treat illegal named expressions (i.e., ones that are not syntactic procedures). Do you see why? How would you make them behave identically?*

```
(define (Env? x)
  (procedure? x))

;; mtSub : () → Env
(define (mtSub)
  (lambda (name)
    (error 'lookup "no binding for identifier")))

;; aSub: symbol FAE-Value Env → Env
(define (aSub bound-name bound-value env)
  (lambda (want-name)
    (cond
      [(symbol=? want-name bound-name) bound-value]
      [else (lookup want-name env)])))

;; lookup : symbol Env → FAE-Value
(define (lookup name env)
  (env name))
```

Figure 11.1: Procedural Representation of Environments

```
(define-type FAE-Value
  [numV (n number?)]
  [closureV (p procedure?)])

;; interp : FAE Env → FAE-Value
(define (interp expr env)
  (type-case FAE expr
    [num (n) (numV n)]
    [add (l r) (num+ (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (closureV (lambda (arg-val)
                     (interp bound-body
                             (aSub bound-id arg-val env))))]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)]
                 [define arg-val (interp arg-expr env)])
           ((closureV-p fun-val)
            arg-val))]))
```

Figure 11.2: Procedural Representation of Functions

```
(define (number-or-procedure? v)
  (or (number? v)
      (procedure? v)))

(define-type Env
  [mtSub]
  [aSub (name symbol?) (value number-or-procedure?) (env Env?)])

;; lookup : symbol Env → number-or-procedure
(define (lookup name env)
  (type-case Env env
    [mtSub () (error 'lookup "no binding for identifier")]
    [aSub (bound-name bound-value rest-env)
          (if (symbol=? bound-name name)
              bound-value
              (lookup name rest-env))]))

;; interp : FAE Env → number-or-procedure
(define (interp expr env)
  (type-case FAE expr
    [num (n) n]
    [add (l r) (+ (interp l env) (interp r env))]
    [id (v) (lookup v env)]
    [fun (bound-id bound-body)
         (lambda (arg-val)
           (interp bound-body
                   (aSub bound-id arg-val env)))]
    [app (fun-expr arg-expr)
         (local ([define fun-val (interp fun-expr env)]
                 [define arg-val (interp arg-expr env)])
           (fun-val arg-val))]))
```

Figure 11.3: Meta-Circular Interpreter

```
(define-type RCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body RCFAE?)
            (env Env?)])

(define (Env? x)
  (procedure? x))

(define (mtSub)
  (lambda (name)
    (error 'lookup "no binding for identifier")))

(define (aSub bound-name bound-value env)
  (lambda (want-name)
    (cond
      [(symbol=? want-name bound-name) bound-value]
      [else (lookup want-name env)])))

(define (cyclically-bind-and-interp bound-name named-expr env)
  (local ([define rec-ext-env
            (lambda (want-name)
              (cond
                [(symbol=? want-name bound-name)
                 (closureV (fun-param named-expr)
                           (fun-body named-expr)
                           rec-ext-env)]
                [else (lookup want-name env)]))])
    rec-ext-env))

(define (lookup name env)
  (env name))
```

Figure 11.4: Recursion: Support Code with Procedural Representation of Environments

Part VI

State

Chapter 12

Church and State

In Section 10 we freely employed Scheme boxes, but we haven't yet given an account of how they work (beyond an informal intuition). We therefore begin an investigation of boxes and other forms of *mutation*—the changing of values associated with names—to endow a language with *state*.

Mutation is a standard feature in most programming languages. The programs we have written in Scheme have, however, been largely devoid of state. Indeed, Haskell has no mutation operations at all. It is, therefore, possible to design and use languages—even quite powerful ones—that have no explicit notion of state. Simply because the idea that one can program without state hasn't caught on in the mainstream is no reason to reject it.

That said, state does have its place in computation. If we create programs to model the real world, then some of those programs are going to have to accommodate the fact that there the real world has events that truly alter it. For instance, cars really do consume fuel as they run, so a program that models a fuel tank *might* best use state to record changes in fuel level.

Despite that, it makes sense to eschew state where possible because state makes it harder to reason about programs. Once a language has mutable entities, it becomes necessary to talk about the program *before* each mutation happens and similarly *after* that mutation (i.e., the different "states" of the program). Consequently, it becomes much harder to determine what a program actually *does*, because any such answer becomes dependent on *when* one is asking: that is, it becomes dependent on time.

Because of this complexity, programmers should use care when introducing state into their programs. A legitimate use of state is when it models a real world entity that really is itself changing: that is, it models a *temporal* or *time-variant* entity. In contrast, many uses of state to, for instance, manage the counter of a loop, are inessential, and these invariably lead to errors when programs are used in a multi-threaded context. The moral is that it's important to understand what state means and how to evaluate programs that use it (which is what we're about to do), but it's equally important to use it with great care, especially in an increasingly concurrent world.

Chapter 13

Mutable Data Structures

Let's extend our source language to support boxes. Once again, we'll rewind to a simple language so we can study the effect of adding boxes without too much else in the way. That is, we'll define BCFAE, the combination of boxes, conditionals, functions and arithmetic expressions. We'll continue to use `with` expressions with the assumption that the parser converts these into function applications. In particular, we will introduce four new constructs:

```
<BCFAE> ::= ...
          | {newbox <BCFAE>}
          | {setbox <BCFAE> <BCFAE>}
          | {openbox <BCFAE>}
          | {seqn <BCFAE> <BCFAE>}
```

We can implement BCFAE by exploiting boxes in Scheme. This would, however, sheds little light on the nature of boxes. We should instead try to model boxes more explicitly.

What other means have we? If we can't use boxes, or any other notion of state, then we'll have to stick to mutation-free programs to define boxes. Well! It seems clear that this won't be straightforward.

Let's first understand boxes better. Suppose we write

(**define** *b1* (*box* 5))
(**define** *b2* (*box* 5))
(*set-box! b1* 6)
(*unbox b2*)

What response do we get?

This suggests that whatever is bound to *b1* and to *b2* must inherently be different. That is, we can think of each value being held in a different place, so changes to one don't affect the other.[1] The natural representation of a "place" in a modern computer is, of course, a memory cell.

[1] Here's a parable adapted from one I've heard ascribed to Guy Steele. Say you and I have gone on a trip. Over dinner, you say, "You know, I have a Thomas Jefferson $2 note at home!" That's funny, I say; so do I! We wonder whether it's actually the *same* $2 bill that we both think is ours alone. When I get home that night, I call my spouse and ask her to tear my $2 bill in half. You then call your spouse and ask, "Is our $2 bill intact?" Guy Steele is Solomonic.

13.1 Implementation Constraints

Before we get into the details of memory, let's first better understand the operational behavior of boxes. Examine this program:

```
{with {b {newbox 0}}
  {seqn {setbox b {+ 1 {openbox b}}}
        {openbox b}}}
```

which is intended to be equivalent to this Scheme program:

```
(local ([define b (box 0)])
  (begin
    (set-box! b (+ 1 (unbox b)))
    (unbox b)))
```

which evaluates to 1, that is, the mutation in the first operation in the sequence has an effect on the output of the second (which would otherwise have evaluated to 0). Now let's consider a naive interpreter for `seqn` statements. It's going to interpret the first term in the sequence in the environment given to the interpreter, then evaluate the second term in the same environment:

```
[seqn (e1 e2)
      (begin
        (interp e1 env)
        (interp e2 env))]
```

Besides the fact that this simply punts to Scheme's **begin** form, this *can't possibly be correct*! Why not? Because the environment is the only term common to the interpretation of *e1* and *e2*. If the environment is immutable—that is, it doesn't contain boxes—and if we don't employ any global mutation, then the outcome of interpreting the first sub-expression can't possibly have any effect on interpreting the second![2] Therefore, something more complex needs to happen.

One possibility is that we update the environment, and the interpreter always returns both the value of an expression *and* the updated environment. The updated environment can then reflect the changes wrought by mutation. The interpretation of `seqn` would then use the environment resulting from evaluating the first sequent to interpret the second.

While this is tempting, it can significantly alter the intended meaning of a program. For instance, consider this expression:

```
{with {a {newbox 1}}
  {seqn {with {b 3}
          b}
        b}}
```

[2]Depends on what we mean by "effect". The first branch of the sequence could, of course, fail to terminate or could result in an error, which are observable effects. But they are not effects that permit the evaluation of the second branch of the sequence.

This program should halt with an error, because static scope dictates that the second sequent (`b`) contains an unbound identifier. But passing the environment from the first sequent to the second would bind `b`. In other words, this strategy destroys static scope.

Even if we were to devise a sophisticated form of this environment-passing strategy (such as removing all new bindings introduced in a sub-expression), it still wouldn't be satisfactory. Consider this example:

```
{with {a {newbox 1}}
  {with {f {fun {x} {+ x {openbox a}}}}
    {seqn
      {setbox a 2}
      {f 5}}}}
```

We want the mutation to affect the box stored in the closure bound to `f`. But that closure already closes over the environment present at the time of evaluating the named expression—an environment that still reflects that `a` is bound to `1`. Even if we update the environment after the `setbox` operation, we cannot use the updated environment to evaluate the closure's body, at least not without (again!) introducing the potential to violate static scope.

As an aside, notice that in the program fragment above, changing the value of `a` is *not a violation of static scope*! The scoping rule only tells us where each identifier is bound; it does not (in the presence of mutation) fix the value bound to that identifier. To be pedantic, the value bound to the identifier does in fact remain the same: it's the same box for all time. The content of the box can, however, change over time.

We thus face an implementation quandary. There are two possible evaluation strategies for this last code fragment, both flawed:

- Use the environment (which maps `a` to `1`) stored in the closure for `f` when evaluating `{f 5}`. This will, however, ignore the mutation in the sequencing statement. The program will evaluate to `6` rather than `7`.
- Use the environment present at the time of procedure invocation: `{f 5}`. This will certainly record the change to `a` (assuming a reasonable adaptation of the environment), but this reintroduces dynamic scope.

To see the latter, we don't even need a program that uses mutation or sequencing statements. Even a program such as

```
{with {x 3}
  {with {f {fun {y} {+ x y}}}
    {with {x 5}
      {f 10}}}}
```

which should evaluate to `13` evaluates to `15` instead.

13.2 Insight

The preceding discussion does, however, give us some insight into a solution. It tells us that we need to have *two* repositories of information. One, the environment, is the guardian of static scope. The other

will become responsible for tracking dynamic changes. This latter entity is known in the parlance as the *store*. Determining the value bound to an identifier will become a two-step process: we will first use the environment to map the identifier to something that the store will then map to a value. What kind of thing is this intermediary? It's the index that identifies a mutable cell of memory—that is, it's a *memory location*.

Using this insight, we slightly alter our environments:

```
(define-type Env
  [mtSub]
  [aSub (name symbol?)
        (location number?)
        (env Env?)])
```

The store is really a partial function from address locations to values. This, too, we shall implement as a data structure.

```
(define-type Store
  [mtSto]
  [aSto (location number?)
        (value BCFAE-Value?)
        (store Store?)])
```

Correspondingly, we need two lookup procedures:

```
;; env-lookup : symbol Env → location

(define (env-lookup name env)
  (type-case Env env
    [mtSub () (error 'env-lookup "no binding for identifier")]
    [aSub (bound-name bound-location rest-env)
          (if (symbol=? bound-name name)
              bound-location
              (env-lookup name rest-env))]))

;; store-lookup : location Store → BCFAE-Value

(define (store-lookup loc-index sto)
  (type-case Store sto
    [mtSto () (error 'store-lookup "no value at location")]
    [aSto (location value rest-store)
          (if (= location loc-index)
              value
              (store-lookup loc-index rest-store))]))
```

Notice that the types of the two procedures compose to yield a mapping from identifiers to values, just like the erstwhile environment.

Let's now dive into the terms of the interpreter. We'll assume that two identifiers, *env* and *store*, are bound to values of the appropriate type. Some cases are easy: for instance,

[num (*n*) (numV *n*)]
[id (*v*) (*store-lookup* (*env-lookup v env*) *store*)]
[fun (*bound-id bound-body*)
  (closureV *bound-id bound-body env*)]

would all appear to be unchanged. Now consider the conditional:

[if0 (*test truth falsity*)
  (**if** (*numV-zero?* (*interp test env store*))
    (*interp truth env store*)
    (*interp falsity env store*))]

Suppose, with this implementation, we evaluate the following program:

```
{with {b {newbox 0}}
  {if0 {seqn {setbox b 5}
             {openbox b}}
       1
       {openbox b}}}
```

We would want this to evaluate to `5`. However, the implementation does not accomplish this, because mutations performed while evaluating the *test* expression are not propagated to the conditional branches.

In short, what we really want is a *(potentially) modified store* to result from evaluating the condition's test expression. It is this store that we must use to evaluate the branches of the conditional. But the ultimate goal of the interpreter is to produce answers, not just stores. What this means is that the interpreter must now return two results: the value corresponding to the expression, and a store that reflects modifications made in the course of evaluating that expression.

13.3 An Interpreter for Mutable Boxes

To implement state without relying on Scheme's mutation operations, we have seen that we must modify the interpreter significantly. The environment must map names to locations in memory, while the store maps these locations to the values they contain. Furthermore, we have seen that we must force the interpreter to return not only the value of each expression but also an updated store that reflects mutations made in the process of computing that value.

To capture this pair of values, we introduce a new datatype,[3]

(**define-type** Value×Store
  [v×s (*value BCFAE-Value?*) (*store Store?*)])

and reflect it in the type of the interpreter:

;; interp : BCFAE Env Store → Value×Store

Before defining the interpreter, let's look at how evaluation proceeds on a simple program involving boxes.

[3] In Scheme source programs, we would write Value×Store as `Value*Store` and v×s as `v*s`.

13.3.1 The Evaluation Pattern

Now that we have boxes in our language, we can model objects that have state. For example, let's look at a simple stateful object: a light switch. We'll use number to represent the state of the light switch, where `0` means off and `1` means on. The identifier `switch` is bound to a box initially containing `0`; the function `toggle` flips the light switch by mutating the value inside this box:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    ...}}
```

(Since `toggle` doesn't require a useful argument, we call its parameter `dum`.) The interesting property of `toggle` is that it can have different behavior on two invocations with the same input. In other words, the function has memory. That is, if we apply the function twice to the *same* (dummy) argument, it produces different values:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    {+ {toggle 1729}
       {toggle 1729}}}}
```

This expression should return `1`—the first application of `toggle` returns `1`, and the second returns `0`. To see why, let's write down the environment and store at each step.

The first `with` expression:

```
{with {switch {newbox 0}}
  ...}
```

does two things: it allocates the number `0` at some store location (say 100), and then binds the identifier `switch` to this location. We'll assume locations are represented using the boxV constructor, defined below. This gives the following environment and store:

env = [*switch* → 101]
store = [101 → (boxV 100), 100 → (numV 0)]

Notice that composing the environment and store maps `switch` to a box containing `0`.

After the second `with` expression:

```
{with {switch {newbox 0}}
  {with {toggle {fun {dum}
                  {if0 {openbox switch}
                       {seqn
                         {setbox switch 1}
                         1}
                       {seqn
                         {setbox switch 0}
                         0}}}}
    ...}}
```

the environment and store are:

env = [*toggle* → 102, *switch* → 101]
store = [102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 0)]

Now we come to the two applications of `toggle`. Let's examine the first call. Recall the type of *interp*: it consumes an expression, an environment, and a store. Thus, the interpretation of the first application is:

(*interp* '{toggle 1729}
[*toggle* → 102, *switch* → 101]
[102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 0)])

Interpreting `switch` results in the value (boxV 100), so interpreting `{openbox switch}` reduces to a store dereference of location 100, yielding the value (numV 0).

The successful branch of the `if0` expression:

```
{seqn
  {setbox switch 1}
  1}}
```

modifies the store; after the `setbox`, the environment and store are:

env = [*toggle* → 102, *switch* → 101]
store = [102 → (closureV '{fun ...} [*switch* → 101]),
101 → (boxV 100),
100 → (numV 1)]

For the first application of `toggle`, the interpreter returns a Value×Store where the value is (numV 1) and the store is as above.

Now consider the second application of *toggle*. It uses the store returned from the first application, so its interpretation is:

(*interp* '{toggle 1729}
  [*toggle* → 102, *switch* → 101]
  [102 → (closureV '{fun ...} [*switch* → 101]),
   101 → (boxV 100),
   100 → (numV 1)])

This time, in the body of `toggle`, the expression `{openbox switch}` evaluates to `1`, so we follow the failing branch of the conditional. The interpreter returns the value (numV 0) and a store whose location 100 maps to (numV 0).

Look carefully at the two (*interp* ···) lines above that evaluate the two invocations of `toggle`. Although both invocations took the *same* expression and environment, they were evaluated in different *stores*; that is the difference that led to the different results. Notice how the interpreter passed the store through the computation: it passed the original store from addition to the first `toggle` application, which return a modified store; it then passed the modified store to the second `toggle` application, which returned yet another store. The interpreter returned this final store with the sum of `0` and `1`. Therefore, the result of the entire expression is `1`.

13.3.2 The Interpreter

This style of passing the current store in and updated store out of every expression's evaluation is called *store-passing style*. We must now update the CFAE interpreter to use this style, and then extend it to support the operations on boxes.

Terms that are already syntactically values do not affect the store (since they require no further evaluation). Therefore, they return the store unaltered:

```
[num (n) (v×s (numV n) store)]
[id (v) (v×s (store-lookup (env-lookup v env) store) store)]
[fun (bound-id bound-body)
     (v×s (closureV bound-id bound-body env) store)]
```

The interpreter for conditionals reflects a pattern that will soon become very familiar:

```
[if0 (test truth falsity)
    (type-case Value×Store (interp test env store)
      [v×s (test-value test-store)
           (if (num-zero? test-value)
               (interp truth env test-store)
               (interp falsity env test-store))])]
```

In particular, note the store used to interpret the branches: It's the store that *results from evaluating the condition*. The store bound to *test-store* is "newer" than that bound to *store*, because it reflects mutations made while evaluating the test expression.

Exercise 13.3.1 *Modify this interpreter to use the wrong store—in this case, store rather than test-store in the success and failure branches—and then write a program that actually catches the interpreter producing faulty output. Until you can do this, you have not truly understood how programs that use state should evaluate!*

When we get to arithmetic expressions and function evaluation, we have a choice to make: in which order do we evaluate the sub-expressions? Given the program

```
{with {b {newbox 4}}
  {+ {openbox b}
     {with {dummy {setbox b 5}}
       {openbox b}}}}
```

evaluating from left-to-right yields `9` while evaluating from right-to-left produces `10`! We'll fix a left-to-right order for binary operations, and function-before-argument (also a kind of "left-to-right") for applications. Thus, the rule for addition is

```
[add (l r)
     (type-case Value×Store (interp l env store)
       [v×s (l-value l-store)
            (type-case Value×Store (interp r env l-store)
              [v×s (r-value r-store)
                   (v×s (num+ l-value r-value)
                        r-store)])])]
```

Carefully observe the stores used in the two invocations of the interpreter as well as the one returned with the resulting value. It's easy to make a mistake!

To upgrade a CFAE interpreter to store-passing style, we must also adapt the rule for applications. This looks more complex, but for the most part it's really just the same pattern carried through:

```
[app (fun-expr arg-expr)
     (type-case Value×Store (interp fun-expr env store)
       [v×s (fun-value fun-store)
            (type-case Value×Store (interp arg-expr env fun-store)
              [v×s (arg-value arg-store)
                   (local ([define new-loc (next-location arg-store)])
                     (interp (closureV-body fun-value)
                             (aSub (closureV-param fun-value)
                                   new-loc
                                   (closureV-env fun-value))
                             (aSto new-loc
                                   arg-value
                                   arg-store)))])])]
```

Notice that every time we extend the environment, we map the binding to a new location (using *next-location*, defined below). This new location is then bound to the result of evaluating the argument. As a result, tracing the formal parameter through both the environment and the store still yields the same result.

Finally, we need to demonstrate the interpretation of boxes. First, we must extend our notion of values:

```
(define-type BCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body BCFAE?)
            (env Env?)]
  [boxV (location number?)])
```

Given this new kind of value, let's study the interpretation of the four new constructs.

Sequences are easy. The interpreter evaluates the first sub-expression, ignores the resulting value (thus having evaluated the sub-expression only for the effect it might have on the store)—notice that *e1-value* is bound but never used—and returns the result of evaluating the second expression in the store (potentially) modified by evaluating the first sub-expression:

```
[seqn (e1 e2)
      (type-case Value×Store (interp e1 env store)
        [v×s (e1-value e1-store)
             (interp e2 env e1-store)])]
```

The essence of `newbox` is to obtain a new storage location, wrap its address in a boxV, and return the boxV as the value portion of the response accompanied by an extended store.

```
[newbox (value-expr)
        (type-case Value×Store (interp value-expr env store)
          [v×s (expr-value expr-store)
               (local ([define new-loc (next-location expr-store)])
                 (v×s (boxV new-loc)
                      (aSto new-loc expr-value expr-store)))])]
```

To modify the content of a box, the interpreter first evaluates the first sub-expression to a location, then updates the store with a *new value for the same location*. Because all expressions must return a value, `setbox` chooses to return the new value put in the box as the value of the entire expression.

```
[setbox (box-expr value-expr)
        (type-case Value×Store (interp box-expr env store)
          [v×s (box-value box-store)
               (type-case Value×Store (interp value-expr env box-store)
                 [v×s (value-value value-store)
                      (v×s value-value
                           (aSto (boxV-location box-value)
                                 value-value
                                 value-store))])])]
```

Opening a box is straightforward: get a location, look it up in the store, and return the resulting value.

```
[openbox (box-expr)
         (type-case Value×Store (interp box-expr env store)
```

```
[v×s (box-value box-store)
     (v×s (store-lookup (boxV-location box-value)
                        box-store)
          box-store)])]
```

Of course, the term "the store" is ambiguous, because there are different stores before and after evaluating the sub-expression. Which store should we use? Does this interpreter do the right thing?

All that remains is to implement *next-location*. Here's one implementation:

```
(define next-location
  (local ([define last-loc (box -1)])
    (lambda (store)
      (begin
        (set-box! last-loc (+ 1 (unbox last-loc)))
        (unbox last-loc)))))
```

This is an extremely unsatisfying way to implement *next-location*, because it ultimately relies on a box! However, this box is not essential. Can you get rid of it?

The core of the interpreter is in Figure 13.1 and Figure 13.2.

Exercise 13.3.2 *Define next-location so it does not have side-effects.*
Hint: *You may need to modify the interpreter to do this.*

13.4 Scope versus Extent

Notice that while closures refer to the environment of definition, they do not refer to the corresponding store. The store is therefore a global record of changes made during execution. As a result, stores and environments have different *patterns of flow*. Whereas the interpreter employs the same environment for both arms of an addition, for instance, it cascades the store from one arm to the next and then back out alongside the resulting value. This latter kind of flow is sometimes called *threading*, since it resembles the action of a needle through cloth.

These two flows of values through the interpreter correspond to a deep difference between names and values. A value persists in the store long after the name that introduced it has disappeared from the environment. This is not inherently a problem, because the value may have been the result of a computation, and some other name may have since come to be associated with that value. In other words, identifiers have *lexical scope*; values themselves, however, potentially have indefinite, *dynamic extent*.

Some languages confuse these two ideas. As a result, when an identifier ceases to be in scope, they remove the value corresponding to the identifier. That value may be the result of the computation, however, and some other identifier may still have a reference to it. This premature removal of the value will, therefore, inevitably lead to a system crash. Depending on how the implementation "removes" the value, however, the system may crash later instead of sooner, leading to extremely vexing bugs. This is a common problem in languages like C and C++.

In most cases, garbage collection (Section 21) lets languages dissociate scope from the reclamation of space consumed by values. The performance of garbage collectors is often far better than we might naïvely imagine (especially in comparison to the alternative).

In terms of language design, there are many reasons why C and C++ have adopted the broken policy of not distinguishing between scope and extent. These reasons roughly fall into the following categories:

- Justified concerns about fine-grained performance control.
- Mistakes arising from misconceptions about performance.
- History (we understand things better now than we did then).
- Ignorance of concepts that were known even at that time.

Whatever their reasons, these language design flaws have genuine and expensive consequences: they cause both errors and poor performance in programs. These errors, in particular, can lead to serious security problems, which have serious financial and social consequences. Therefore, the questions we raise here are not merely academic.

While programmers who are experts in these languages have evolved a series of ad hoc techniques for contending with these problems, we students of programming languages should know better. We should recognize their techniques for what they are, namely symptoms of a broken programming language design rather than proper solutions to a problem. Serious students of languages and related computer science technologies take these flaws as a starting point for exploring new and better designs.

Exercise 13.4.1 *Modify the interpreter to evaluate addition from right to left instead of left-to-right. Construct a test case that should yield different answers in the two cases, and show that your implementation returns the right value on your test case.*

Exercise 13.4.2 *Modify* `seqn` *to permit an arbitrary number of sub-expressions, not just two. They should evaluate in left-to-right order.*

Exercise 13.4.3 *New assignments to a location currently mask the old thanks to the way we've defined store-lookup, but the data structure still has a record of the old assignments. Modify the implementation of stores so that they have at most one assignment for each location.*

Exercise 13.4.4 *Use Scheme procedures to implement the store as a partial function.*

```
;; interp : BCFAE Env Store → Value×Store
(define (interp expr env store)
  (type-case BCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
         (type-case Value×Store (interp l env store)
           [v×s (l-value l-store)
                (type-case Value×Store (interp r env l-store)
                  [v×s (r-value r-store)
                       (v×s (num+ l-value r-value)
                            r-store)])])]
    [id (v) (v×s (store-lookup (env-lookup v env) store) store)]
    [fun (bound-id bound-body)
         (v×s (closureV bound-id bound-body env) store)]
    [app (fun-expr arg-expr)
         (type-case Value×Store (interp fun-expr env store)
           [v×s (fun-value fun-store)
                (type-case Value×Store (interp arg-expr env fun-store)
                  [v×s (arg-value arg-store)
                       (local ([define new-loc (next-location arg-store)])
                         (interp (closureV-body fun-value)
                                 (aSub (closureV-param fun-value)
                                       new-loc
                                       (closureV-env fun-value))
                                 (aSto new-loc
                                       arg-value
                                       arg-store)))])])]
    [if0 (test truth falsity)
         (type-case Value×Store (interp test env store)
           [v×s (test-value test-store)
                (if (num-zero? test-value)
                    (interp truth env test-store)
                    (interp falsity env test-store))])]
    ⋮
```

Figure 13.1: Implementing Mutable Data Structures , Part 1

```
⋮
[newbox (value-expr)
        (type-case Value×Store (interp value-expr env store)
          [v×s (expr-value expr-store)
               (local ([define new-loc (next-location expr-store)])
                 (v×s (boxV new-loc)
                      (aSto new-loc expr-value expr-store)))])]
[setbox (box-expr value-expr)
        (type-case Value×Store (interp box-expr env store)
          [v×s (box-value box-store)
               (type-case Value×Store (interp value-expr env box-store)
                 [v×s (value-value value-store)
                      (v×s value-value
                           (aSto (boxV-location box-value)
                                 value-value
                                 value-store))])])]
[openbox (box-expr)
         (type-case Value×Store (interp box-expr env store)
           [v×s (box-value box-store)
                (v×s (store-lookup (boxV-location box-value)
                                   box-store)
                     box-store)])]
[seqn (e1 e2)
      (type-case Value×Store (interp e1 env store)
        [v×s (e1-value e1-store)
             (interp e2 env e1-store)])]))
```

Figure 13.2: Implementing Mutable Data Structures, Part 2

Chapter 14

Variables

In Section 13 we studied the implementation of mutable data structures. The boxes we studied there could just as well have been vectors or other *container* types, such as objects with fields.

In traditional languages like C and Java, there are actually two forms of mutation. One is mutating the value in a container, such as an object (in Java). The expression

```
o.f = e
```

evaluates `o` to an object, `e` to some value, and changes the content of field `f` of `o` to hold the value of `e`. Note that `o` can be an arbitrary expression (for instance, it can look up an object in some other data structure) that is *evaluated* to a value. In contrast, a programmer can also write a method such as

```
void m (int i) {
  i = 4;
}
```

Here, `i` must literally be an identifier; it cannot be an arbitrary expression that evaluates to an identifier. That is, we are not mutating the value contained within a box (or position in a vector, or a field); rather, we are mutating the value bound to an identifier itself. That makes the identifier a *variable*. A more interesting example would be a pattern that repeatedly occurs in object-oriented programming:

```
  private int x;
  void set_x (int y) {
    x = y;
  }
  void get_x () {
    return x;
  }
```

Here, the variable `x` is private to the object, and can be accessed only through the getter and setter methods. The setter assigns a new value to `x`.

14.1 Implementing Variables

First, let's extend our language to include variables:

```
<VCFAE> ::= ...
          | {set <id> <VCFAE>}
          | {seqn <VCFAE> <VCFAE>}
```

Observe that the `set` expression expects a literal identifier after the keyword.

Implementing variables is a little different from implementing boxes. In the latter case, we first evaluate the position that identifies the box:

```
[setbox (box-expr value-expr)
        (type-case Value×Store (interp box-expr env store)
          [v×s (box-value box-store)
               ⋮])]
```

In contrast, in a language with variables, identifiers do not represent boxes. Therefore, the corresponding code:

```
[set (var value)
     (type-case Value×Store (interp var env store)
       [v×s (var-value var-store)
            ⋮])]
```

would be counter-productive. Evaluating the identifier would result in a value, which we cannot mutate, as opposed to a location, whose content we can modify by updating the store. This immediately suggests a slightly different evaluation strategy:

```
[set (var value)
     (type-case Value×Store (interp value env store)
       [v×s (value-value value-store)
            (local ([define the-loc (env-lookup var env)])
              ⋮)])]
```

That is, we evaluate the expression that represents the new value to be stored in the interpreter. Instead of evaluating the identifier, however, we only look it up in the environment. This results in a location where the new value should be stored. In particular, *notice an unusual pattern*: the interpreter dereferences the identifier in the environment, but does *not* dereference the result (the identifier's location) in the store. We have not seen this pattern before, and will never see it again after this material.

Many languages make a distinction between mutable data structures and mutable identifiers. When a mutable identifier appears in the assignment position of an assignment operator (many languages use the same syntactic operator, = or `:=`, to represent both operations), the language implementation only partially resolves the identifier. This special kind of value—the location of an identifier—is traditionally known as an *l-value* (pronounced "ell-value").

Whence this unusual name? Consider the following two statements in C:

```
x = 2;
y = x;
```

In the second statement, `x` must be reduced to a value—i.e., *store-lookup* and *env-lookup* must be composed and applied to its content—whereas in the first statement, `x` must only be reduced to a location, not to a value. In languages where locations are not values (more on that below), this odd kind of "value" is known as an "l-value", since it appears *only* on the left-hand-side of assignment statements.

Given this insight, we can now easily complete the definition of the assignment statement:

```
[set (var value)
     (type-case Value×Store (interp value env store)
       [v×s (value-value value-store)
            (local ([define the-loc (env-lookup var env)])
              (v×s value-value
                   (aSto the-loc value-value value-store)))])]
```

The rest of the interpreter remains unchanged. Note, in particular, that it still employs store-passing style. Figure 14.1 and Figure 14.2 present the core of the interpreter.

14.2 Interaction Between Variables and Function Application

Variables and function application appear to be two independent language features, but perhaps they are not. Consider the following program:

```
{with {v 0}
  {with {f {fun {y}
                {set y 5}}}
    {seqn {f v}
          v}}}
```

What do we expect it to evaluate to? There are two different, reasonable answers: `0` and `5`. The first assumes that the mutation is to the formal variable, `y`, and does not affect the actual argument, `v`; the second assumes that this mutation does have the effect of modifying the actual argument.

Our current implementation yields the value `0`. This is because the act of invoking the function binds the formal parameter to a new location:

```
(local ([define new-loc (next-location arg-store)])
  (interp (closureV-body fun-value)
          (aSub (closureV-param fun-value)
                new-loc
                (closureV-env fun-value))
          (aSto new-loc
                arg-value
                arg-store)))
```

The evaluated argument is held in this new location. Therefore, changes to the content of that location in the store do not affect the actual parameter. This is the standard form of eager evaluation, traditionally called *call-by-value*.

Let's now explore the alternative. This form of evaluation is called *call-by-reference*. This new technique gets its name because we will pass a *reference* to the actual argument, rather than merely its value. Thus, updates to the reference within the called procedure will become visible to the calling context, too.

To explore this design, let's extend our language further so we have two kinds of procedures: call-by-value (`fun`) and call-by-reference (`refun`):

```
<RVCFAE> ::= ...
           | {refun {<id>} <RVCFAE>}
           | {set <id> <RVCFAE>}
           | {seqn <RVCFAE> <RVCFAE>}
```

That is, syntactically a call-by-reference procedure looks the same as a call-by-value procedure other than the distinguishing keyword. It is their interpretation that will distinguish them.

All the code we have developed thus far remains the same for call-by-value procedure invocation. In particular, `with` expressions should continue to expand into immediate applications of `fun`-defined procedures. Let us proceed to defining the interpretation of call-by-reference procedures.

The first step is to evaluate a reference procedure definition. This is straightforward:

```
[refun (bound-id bound-body)
  (v×s (refclosV bound-id bound-body env) store)]
```

We create a new kind of closure so we can later distinguish what kind of procedure we are about to apply, but its fields are the same:

```
(define-type RVCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body RVCFAE?)
            (sc SubCache?)]
  [refclosV (param symbol?)
            (body RVCFAE?)
            (sc SubCache?)])
```

Now let us study the interpretation of application. After evaluating the procedure position, we must check which kind of procedure it is before evaluating the argument. If it's a call-by-value procedure, we proceed as before:

```
[app (fun-expr arg-expr)
     (type-case Value×Store (interp fun-expr env store)
       [v×s (fun-value fun-store)
            (type-case RVCFAE-Value fun-value
              [closureV (cl-param cl-body cl-env)
                        (type-case Value×Store (interp arg-expr env fun-store)
```

```
                    [v×s (arg-value arg-store)
                         (local ([define new-loc (next-location arg-store)])
                           (interp cl-body
                                   (aSub cl-param
                                         new-loc
                                         cl-env)
                                   (aSto new-loc
                                         arg-value
                                         arg-store)))])]
          [refclosV (cl-param cl-body cl-env)
                    ⋮]
          [numV (_) (error 'interp "trying to apply a number")])])]
```

We can thus focus our attention on the interpretation of function applications where the function position evaluates to a reference procedure closure.

When applying a call-by-reference procedure, we must supply it with the location of the actual argument. This presupposes that the actual argument will be a variable. We put this down to an implicit constraint of the language, namely that whenever applying a reference procedure, we assume that the argument expression is *syntactically* a variable. Given this, we can easily determine its location, and extend the closure's environment with the formal parameter bound to this location:

```
[refclosV (cl-param cl-body cl-env)
          (local ([define arg-loc (env-lookup (id-name arg-expr) env)])
            (interp cl-body
                    (aSub cl-param
                          arg-loc
                          cl-env)
                    fun-store))]
```

Notice the recurrence of the l-value pattern: an environment lookup without a corresponding store lookup. (This is why we dispatched on the type of the closure without first evaluating the argument: had we not done so, the argument expression would have been reduced to a value, which would be either useless or incorrect in the case of reference procedures.) As a result, any mutations to the formal parameter are now changes to the same location as the actual parameter, and are thus effectively mutations to the actual parameter also. Thus, the example that inaugurated this section will yield the result `5`.

Figure 14.3 and Figure 14.4 present the core of the interpreter.

14.3 Perspective

Should languages have reference procedures? Passing references to procedures has the following dangerous property: the formal parameter becomes an *alias* of the actual parameter, as all changes to the formal manifest as changes to the actual also. This is especially insidiuous because the programmer may not know he is about to apply a reference procedure: Some languages like C offer the ability to mark specific

parameters of multi-parameter procedures with keywords such as `&` and `ref`, meaning they alone should be passed by reference (these are known as *reference parameters*). The client of such a procedure may thus find that, mysteriously, the act of invoking this procedure has changed the value of his identifiers. This *aliasing* effect can lead to errors that are particularly difficult to detect and diagnose.

This phenomenon cannot occur with call-by-value: changes to the variable in the called procedure do not affect the caller. There is, therefore, nearly universal agreement in modern languages that arguments should be passed by value. If the called procedure intends to mutate a value, it must consume a box (or other container data structure); the caller must, in turn, signal acceptance of this behavior by passing a box as the actual argument. The caller then has the freedom to inspect the content of the (possibly mutated) box and determine whether to accept this mutation in the remainder of the program, or to reject it by ignoring the altered content of the box.

Why did languages introduce reference parameters? For one thing, they are "cheaper": they do not require additional allocation. (We can see this difference clearly when we contrast the two kinds of procedure application.) However, the problems they introduce arguably far outweigh this small savings in memory.

Reference parameters do, however, also confer a small expressiveness benefit. Without reference parameters, we cannot define a procedure that swaps the content of two variables. In the following code,

```
{with {swap {fun {x}
               {fun {y}
                 {with {z x}
                   {seqn {set x y}
                         {set y z}}}}}}
  {with {a 3}
    {with {b 2}
      {seqn {{swap a} b}
            b}}}}
```

the result of the computation is still `2`, because the mutations to `x` and `y` inside the procedure do not affect `a` and `b`. In contrast,

```
{with {swap {refun {x}
               {refun {y}
                 {with {z x}
                   {seqn {set x y}
                         {set y z}}}}}}
  {with {a 3}
    {with {b 2}
      {seqn {{swap a} b}
            b}}}}
```

results in the value `3`: since `x` and `y` are just aliases to `a` and `b`, mutations to the former are reflected as mutations to the latter. (Note that both procedures must be `refun`s and not `fun`s, else the swap is at best partial.)

This example also, however, illustrates why aliasing can cause problems. The implementor of the procedure may have used mutation accidentally, without meaning to affect the caller. The procedure boundary

abstraction has, however, been compromised by the aliasing, and accidental side-effects can leak into the calling contexts, exposing unnecessary implementation details of the procedure.

In the early days of programming language design, before programs were particularly sophisticated, the ability to write simple abstractions such as `swap` was considered valuable (since it is used, for instance, in the implementation of some sorting algorithms). Today, however, we recognize that such abstractions are rather meager in the face of the needs of modern systems. We pay greater attention, instead, to the need for creating useful abstraction boundaries between units of modularity such as procedures: the fewer hidden interactions they have, and the less they interfere with one another, the more easily we can reason about their behavior in isolation.

Exercise 14.3.1 *While call-by-value preserves the value of* variables *in the calling context, it does not protect all values. In particular, in many call-by-value languages, a composite data structure (such as a vector) passed as an argument may be mutated by the callee, with the effects visible to the caller.*

1. *Does this behavior contradict the claim that the language is passing "values" as arguments? Use our investigation of mutable data structures in Section 13 to make your argument* rigorous.
   Hint: *Implement an interpreter for a language with both boxes and call-by-reference application, then argue about similarities and differences.*

2. *Languages like ML tackle this problem by forcing programmers to annotate all mutable data structures using references, the ML counterpart to boxes. Any data structure not so mutated is considered immutable. What trade-offs does ML's design introduce?*

Exercise 14.3.2 *There appears to be a neutral ground between call-by-value and call-by-reference. Consider the following proposed syntax:*

```
{with {swap {fun {x}
               {fun {y}
                 {with {z x}
                   {seqn {set x y}
                         {set y z}}}}}}}
  {with {a 3}
    {with {b 2}
      {seqn {{swap {ref a}} {ref b}}
            b}}}}
```

The `ref` *notation is an indicator to the interpreter to pass the variable's location rather than its value; that is, by using* `{ref a}` *and* `{ref b}`*, the invoker of the procedure indicates his willingness to have his variables be aliased and thus, potentially, be mutated.*

1. *Modify the interpreter to support the use of* `ref` *for procedure arguments.*

2. *Does this proposal result in a procedural abstraction of the process of swapping the values of two variables? If it does, this would reconcile the design tension between the two invocation techniques: it avoids the difficulty of call-by-value (the inability to write a swap procedure) as well as that of call-by-reference (aliasing of parameters without the caller's knowledge). Discuss.*

3. *Suppose programmers are allowed to apply* `ref` *to variables elsewhere in the program. What type should the interpreter use to represent the resulting value? How does this compare to an l-value? Does this introduce the need for additional operators in the language? How does this relate to the* `&` *operator in C?*

```
(define-type VCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body VCFAE?)
            (env Env?)])

;; interp : VCFAE Env Store → Value×Store
(define (interp expr env store)
  (type-case VCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
         (type-case Value×Store (interp l env store)
           [v×s (l-value l-store)
                (type-case Value×Store (interp r env l-store)
                  [v×s (r-value r-store)
                       (v×s (num+ l-value r-value)
                            r-store)])])]
    [id (v) (v×s (store-lookup (env-lookup v env) store) store)]
    [fun (bound-id bound-body)
         (v×s (closureV bound-id bound-body env) store)]
    ⋮
```

Figure 14.1: Implementing Variables, Part 1

```
⋮
[app (fun-expr arg-expr)
     (type-case Value×Store (interp fun-expr env store)
       [v×s (fun-value fun-store)
            (type-case Value×Store (interp arg-expr env fun-store)
              [v×s (arg-value arg-store)
                   (local ([define new-loc (next-location arg-store)])
                     (interp (closureV-body fun-value)
                             (aSub (closureV-param fun-value)
                                   new-loc
                                   (closureV-env fun-value))
                             (aSto new-loc
                                   arg-value
                                   arg-store)))])])]
[if0 (test truth falsity)
    (type-case Value×Store (interp test env store)
      [v×s (test-value test-store)
           (if (num-zero? test-value)
               (interp truth env test-store)
               (interp falsity env test-store))])]
[set (var value)
     (type-case Value×Store (interp value env store)
       [v×s (value-value value-store)
            (local ([define the-loc (env-lookup var env)])
              (v×s value-value
                   (aSto the-loc value-value value-store)))])]
[seqn (e1 e2)
      (type-case Value×Store (interp e1 env store)
        [v×s (e1-value e1-store)
             (interp e2 env e1-store)])]))
```

Figure 14.2: Implementing Variables, Part 2

```
(define-type RVCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
            (body RVCFAE?)
            (env Env?)]
  [refclosV (param symbol?)
            (body RVCFAE?)
            (env Env?)])

;; interp : RVCFAE Env Store → Value×Store
(define (interp expr env store)
  (type-case RVCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
         (type-case Value×Store (interp l env store)
           [v×s (l-value l-store)
                (type-case Value×Store (interp r env l-store)
                  [v×s (r-value r-store)
                       (v×s (num+ l-value r-value)
                            r-store)])])]
    [id (v) (v×s (store-lookup (env-lookup v env) store) store)]
    [if0 (test pass fail)
         (type-case Value×Store (interp test env store)
           [v×s (test-value test-store)
                (if (num-zero? test-value)
                    (interp pass env test-store)
                    (interp fail env test-store))])]
    [fun (bound-id bound-body)
         (v×s (closureV bound-id bound-body env) store)]
    [refun (bound-id bound-body)
           (v×s (refclosV bound-id bound-body env) store)]
    ⋮
```

Figure 14.3: Implementing Call-by-Reference, Part 1

```
⋮
[app (fun-expr arg-expr)
     (type-case Value×Store (interp fun-expr env store)
       [v×s (fun-value fun-store)
            (type-case RVCFAE-Value fun-value
              [closureV (cl-param cl-body cl-env)
                        (type-case Value×Store (interp arg-expr env fun-store)
                          [v×s (arg-value arg-store)
                               (local ([define new-loc (next-location arg-store)])
                                 (interp cl-body
                                         (aSub cl-param
                                               new-loc
                                               cl-env)
                                         (aSto new-loc
                                               arg-value
                                               arg-store)))])]
              [refclosV (cl-param cl-body cl-env)
                        (local ([define arg-loc (env-lookup (id-name arg-expr) env)])
                          (interp cl-body
                                  (aSub cl-param
                                        arg-loc
                                        cl-env)
                                  fun-store))]
              [numV (_) (error 'interp "trying to apply a number")])])]
[set (var value)
     (type-case Value×Store (interp value env store)
       [v×s (value-value value-store)
            (local ([define the-loc (env-lookup var env)])
              (v×s value-value
                   (aSto the-loc value-value value-store)))])]
[seqn (e1 e2)
      (type-case Value×Store (interp e1 env store)
        [v×s (e1-value e1-store)
             (interp e2 env e1-store)])]))
```

Figure 14.4: Implementing Call-by-Reference, Part 2

Part VII

Continuations

Chapter 15

Some Problems with Web Programs

Web programs can be awfully buggy. For instance, consider the following interaction with a popular commercial travel Web site.

1. Choose the option to search for a hotel room, and enter the corresponding information.
2. Suppose the Web site response with two hotels, options *A* and *B*.
3. Using your browser's interactive facilities, open the link to hotel *A* in a separate window.
4. Suppose you find *A* reasonable, but are curious about the details of *B*. You therefore return to the window listing all the hotels and open the details for *B* in a separate window.
5. Having scanned the details of *B*, you find *A* a more attractive option. Since the window for *A* is still on-screen, you switch to it and click the reservation link.
6. The travel site makes your reservation at hotel *B*.

If an error like this were isolated to a single Web page, or even to a single site, we can put it down to programmer error. But when such errors occur on numerous sites, as indeed they do, it forces us to systematically investigate their cause and to more carefully consider the design and implementation of Web programs.

Before we investigate the problem in general, it helps to understand its breadth. The following is an uncontroversial property that we would expect of a travel reservation site:

> The user should receive a reservation at the hotel that was displayed on the page he submitted.

Can we generalize this? That is, should a user receive information based strictly on the information displayed on the page on which the user clicked a button?

Consider an on-line bookstore. Conduct the same sequence of interactions as above, except with books instead of hotels. Upon examining choice *B*, suppose you clicked to add it to your "shopping cart". Now when you go to the page for book *A* and add it, too, to your shopping cart, what do you expect to find in it? Certainly, the bookseller hopes you have both *A and B* in the cart (since, after all, they are in the business of selling as many books as possible). This is a clear violation of the property we elucidated above.

The problem is compounded by the number of interaction operations supported by modern Web browsers. In addition to opening Web pages in new windows, browsers offer the ability to clone the currently-visible page, to go back to a previous page, to go forward to a page (from which the user had previously gone back), to create bookmarks, and so on. Worse, most of these operations are *silent*: the browser does not notify the Web application that they have been executed, so the application must reconstruct events based on the submissions it receives.

Many of the problems with Web programs trace back to their structure. The Web's architecture dictates that every time a Web program sends an Web page to a user, it is forced to terminate; this is because the Web implements a *stateless* protocol. If and when the user chooses to resume the computation (by clicking on a link or button), some other program must resume the computation. This forces a rather perverse program structure on the programmer. We will now study the implications of this structure in some detail.

Stateful and Stateless Protocols

Suppose a client-server computation performs multiple interactions. In a stateful protocol, the server maintains some state information recording its context in the dialog. A well-known example of a stateful protocol is FTP, the Internet file-transfer protocol. In an FTP session, the user can enter multiple commands, and the interpretation of each command is relative to the history of past commands. That is, two invocations of `ls` (to list files in a directory) will produce different answers if the user has invoked `cd` (to change the directory) betwixt. (The context information here is the current directory.)

In contrast to many traditional Internet protocols, the Web implements a stateless protocol, meaning it does not retain any record of prior communication. As a result, in principle the Web application is responsible for completely restoring the state of the computation on each interaction. By analogy, suppose you were executing an editor such as Emacs within an SSH session (which is also stateful: this state is lost when the connection dies). In a stateless SSH, after every unit of output the connection would close. When the user entered another keystroke, the communication would have to carry with it the entire state of running applications (indeed, the entire *history*, to enable Undo operations), the server would have to invoke Emacs afresh, run all the commnands entered so far and, having restored the application to its past state…enter one new keystroke. (In practice, therefore, Web applications are instead designed to communicate with coarse granularity.)

Stateful protocols are easier to program, because the developer is not responsible for setup and breakdown of the state at each interaction. So why use a stateless protocol? They confer the advantage that the server can tolerate far higher loads. If a server can maintain only 1,000 connections at any instant, a stateful protocol that keeps connections open until the transaction terminates would not be able to service more than 1,000 users at a time. Worse, it would need a policy for determining when to terminate connections that appear to no longer be active (e.g., users who have neither logged out nor completed a purchase). A stateless protocol avoids this; the server can serve many more clients in rapid order, and can ignore clients who are not interested in completing a computation. It pays the price of transmitting enough data to resume the compuation.

Chapter 16

The Structure of Web Programs

Suppose we are trying to implement the following simple Web program. The program presents the user with a prompt for a number. Given an input, it presents a prompt for a second number. Given a second input, it displays the sum of the two numbers in a Web page:[1]

```
(web-display
  (+ (web-read "First number: ")
     (web-read "Second number: ")))
```

While this is an extremely simple application, it is sufficient for demonstrating many concepts. Furthermore, it is a microcosm of a Web application that accepts information in multiple stages, such as the outward flight choice on one page and the return choice on the next.

Even this "addition server" is difficult to implement:

1. The Web developer must turn this application into three programs:
   (a) The first program displays the first form.
   (b) The second program consumes the form values from the first form, and generates the second form.
   (c) The third program consumes the form values from the second form, computes the output, and generates the result.
2. Because the value entered in the first form is needed by the third program to compute its output, this value must somehow be transmitted between from the first program to the third. This is typically done by using the hidden field mechanism of HTML.
3. Suppose, instead of using a hidden field, the application developer used a Java Servlet session object, or a database field, to store the first number. (Application developers are often pushed to do this because that is the feature most conveniently supported by the language and API they are employing.) Then, if the developer were to exploratorily open multiple windows, as we discussed in Section 15, the application can compute the wrong answer.

[1] We are assuming the existence of some simple primitives that mask the necessary but, for now, irrelevant complexity of generating HTML forms, and so on.

In particular, the program we have written above, which runs perfectly well on a display console, cannot run on the Web: the moment *web-read* dispatches its Web form to the user, the Web protocol forces the computation to terminate, taking with it all memory of what had to happen next, i.e., the pending computation.

Where is this pending computation specified? The system resumes execution at the URL specified in the "action" field of the generated form. The developer is therefore responsible for making sure that the application that resides at that URL is capable of resuming the computation in its entirety. We have entirely neglected this problem by assuming the existence of a *web-read* procedure, but in fact the entire problem is that we cannot implement it without a more explicit handle on the pending computation.

16.1 Explicating the Pending Computation

For our motivating example, what is the pending computation at the point of the first interaction? In words, it is to consume the result from the form (the first number), generate a form for the second number, add them, then display their result. Since natural language is unwieldy, we would benefit from writing this pending computation in code instead:

```
(web-display
  (+ •
    (web-read "Second number: ")))
```

where we use • to represent the result from the user's form submission. What is •, exactly? It appears to be an invented notation that we must then explain formally. Instead, we can treat it as an identifier, binding it in the traditional way:

```
(lambda (•)
  (web-display
    (+ •
      (web-read "Second number: "))))
```

This procedure, then, represents the computation pending at the point of the first interaction. Applying this procedure to the result of that interaction (the user's input) resumes the computation. Similarly, the pending computation at the point of the second interaction is

```
(lambda (•2)
  (web-display
    (+ •
      •2)))
```

where • is the user's response to the first prompt, which should be in the closure of this procedure.

16.2 A Better Server Primitive

Suppose, therefore, that we had a modified version of *web-read* that we'll call *web-read/k*. This new procedure takes two arguments. The first is a string that it converts into a form, as before. The second is a procedure of one argument representing the pending computation, which we'll henceforth call the *receiver*.

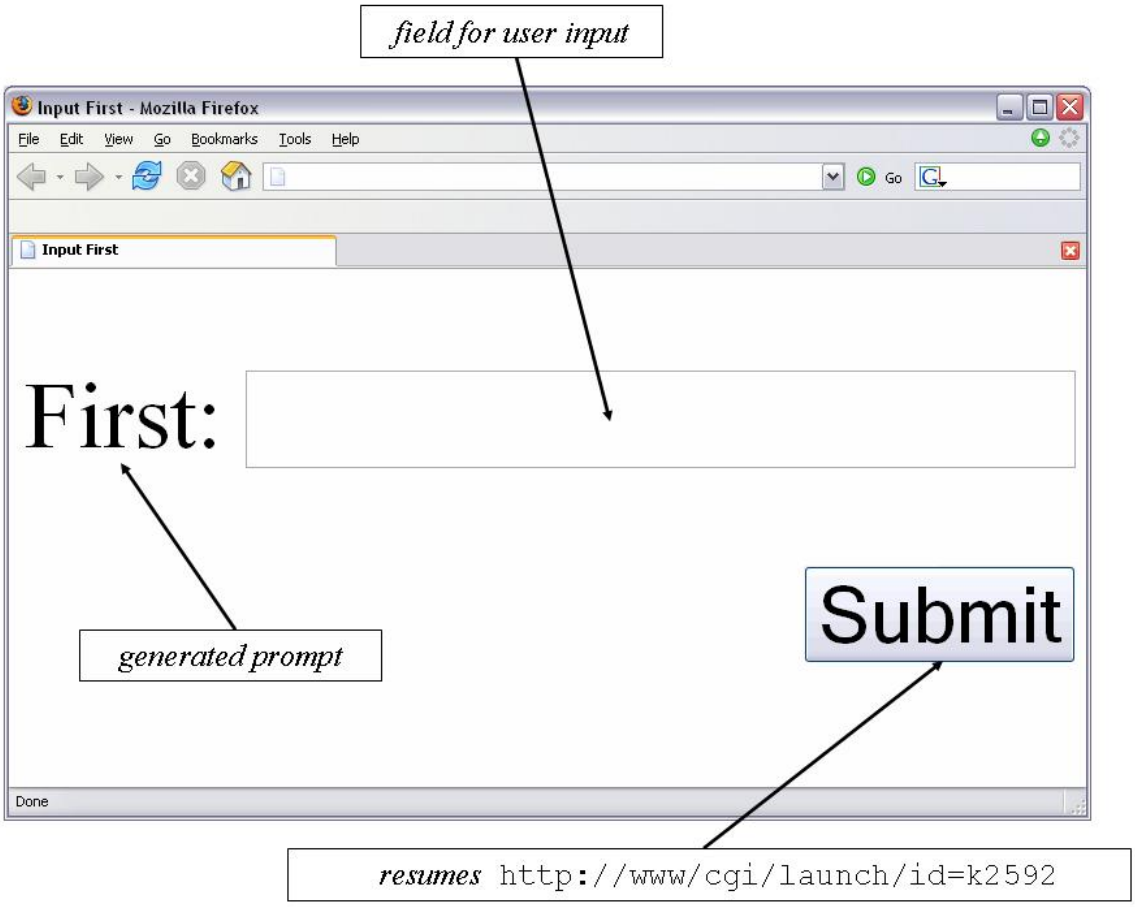

Figure 16.1: Form generated by *web-read/k*

Every time *web-read/k* is invoked, it creates a fresh entry in a hash table. It stores the receiver in this entry, and generates a form action URL that contains the hash table key for this procedure. The hash table is kept in memory by the Web server (which, we'll assume, doesn't terminate). *web-read/k* generates a page and then terminates the Web application's execution, in conformance with the Web protocol.

This generated page is shown in Figure 16.1. The image shows, in outline, the Web page generated by invoking *web-read/k* with the first argument reading "First". This string becomes the prompt. Next to the prompt is a text box where the user can provide input. The action field of the HTML form has a reference to the hash table key of the corresponding fresh entry (in this instance, `k2592`).

When the user submits a response, the server invokes the application named `launch`. This application does two things. First, it uses the key associated with the `id` argument to obtain a receiver closure from the hash table. Second, it extracts the input typed by the user in the text box. The receiver is then applied to this extracted value. This resumes the computation.

Assuming the existence of such a primitive, we might try to rewrite our running application as

```
(web-display
  (+ (web-read/k "First number: "
                 (lambda (•)
                   •))
     (web-read "Second number: ")))
```

but this won't work! Recall that at every Web interaction, the Web application entirely terminates. That means, any computation that has not been included in the receiver is lost forever. As a consequence, when this application resumes, the only "remembered" computation is that in the receiver, which is just the identity procedure: the second Web input, as well as the ensuing computation and the display of the result, have all been irretrievably lost.

In other words, *any computation that isn't explicitly mentioned in the receiver simply never gets performed*, because of the program's termination after each interaction. This forces us to move *all* pending computation into the receiver. Here's what we might try:

```
(web-read/k "First number: "
            (lambda (•)
              (web-display
                (+ •
                   (web-read "Second number: ")))))
```

This, however, is subject to the same analysis: it still uses the hypothetical *web-read* procedure, which we've conceded we don't quite know how to implement. We must, therefore, instead employ *web-read/k* again, as follows:

```
(web-read/k "First number: "
            (lambda (•)
              (web-read/k "Second number: "
                          (lambda (•)
                            (web-display
                              (+ • •)))))))
```

Oh, not quite: we want to add the first number to the second, not just compute twice the second number. Therefore:

```
(web-read/k "First number: "
            (lambda (•1)
              (web-read/k "Second number: "
                          (lambda (•2)
                            (web-display
                              (+ •1 •2)))))))
```

Now, when the program finally generates the sum, it can safely halt without having registered any receivers, because there aren't any computations left to perform. Relative to the original source program, however, the structure of this application is considerably more intricate.

Exercise 16.2.1 *To be entirely pedantic, there is one thing left to do, which is to explicitly halt the program. Extend the program to do this, then transform it to correctly employ web-read/k.*

16.3 Testing Web Transformations

One of the subtle problems with transforming interactive programs for the Web is that they are difficult to test. This difficulty has at least two facets. First, the use of HTML makes programs unwieldy, so we would rather defer its use until the end, but without it we cannot interact with a Web browser. Second, testing a program at the console can be misleading: a computation may not have been properly moved into a receiver but, because Scheme programs do not terminate after every interaction, we would never notice this problem until we ran the program on the Web.

Fortunately, it is easy to simulate the Web's behavior at the console with the following code. The following implementation of *web-read/k* stores the receiver and prompt in a box, and terminates the program's execution using *error*:

```
(define the-receiver (box 'dummy-value))
(define receiver-prompt (box 'dummy-value))

(define (web-display n)
  (printf "Web output: ~a~n" n))

(define (web-read/k p k)
  (begin
    (set-box! receiver-prompt p)
    (set-box! the-receiver k)
    (error 'web-read/k "run (resume) to enter number and simulate clicking Submit")))
```

The procedure *resume* uses the values in these boxes to resume the computation:

```
(define (resume)
  (begin
    (display (unbox receiver-prompt))
    ((unbox the-receiver) (read))))
```

We can therefore test a program such as the addition application as follows:

```
Language: PLAI - Advanced Student.
web-read/k: run (resume) to enter number and simulate clicking Submit
>
```

This means the program has arrived at *web-read/k* for the first time. We run

```
> (resume)
```

which prompts us for the first input. Providing an input results in the same terminating "error" message, corresponding to the next interaction point. Running (*resume*) prompts for the second input. When we provide the second, we see the sum of the two numbers printed to the console.

Exercise 16.3.1 *Use this testing harness to execute the incorrectly transformed versions of the addition program to explore their behavior.*

16.4 Executing Programs on a Traditional Server

Suppose we must run our Web application on a traditional Web server, which does not provide support for the hash table used by *web-read/k*. This doesn't mean we must waste the effort we expended transforming the program: that effort was a direct consequence of the Web's protocol, which the traditional server also obeys (even more slavishly!).

What's the problem with executing this program on a traditional server?

```
(web-read/k "First number: "
            (lambda (•1)
              (web-read/k "Second number: "
                          (lambda (•2)
                            (web-display
                              (+ •1 •2))))))
```

If *web-read/k* cannot behave in a privileged fashion, then its receiver argument will not be invoked automatically by the server. Instead, the entire computation will terminate with the first interaction.

To reflect this problem, let us use a different primitive, *web-read/r* in place of *web-read/k*. The suffix indicates that it will be given the *name* of a **r**eceiver as a second argument. *web-read/r* uses this name in the URL inserted in the action field of the generated form. To do so, however, each receiver must be a named Web application that the server can invoke directly (whereas the receivers are currently anonymous procedures nested within other procedures).

The process of making nested procedures into top-level ones is known as *lifting*. That is, each anonymous procedure is moved to the top-level and given an unique name. In the example program above, the innermost procedure might become

```
(define (f2 •2)
  (web-display
    (+ •1
       •2)))
```

which the outer procedure can refer to:

```
(define (f1 •1)
  (web-read/r "Second number: "
              "f2"))
```

The main program then becomes

```
(web-read/r "First number: "
            "f1")
```

Because of this transformation, *web-read/r* can safely terminate after using the procedure named in the second argument in the action field's URL. All the remaining work must be completed by this named top-level procedure. Each top-level procedure consumes one argument, which is the data provided by the user.

Unfortunately, by sloppily lifting the procedures to the top-level, we've created a problem: $\bullet_1$ is a free identifier in *f2*! The problem is that we were simplistic in the way we lifted the procedures. (A different simplistic method—failing to keep the two instances of • separate—would, of course, have created a different problem, namely that *f2* would have just added • to itself, ignoring the user's first input.)

In general, when lifting we must add parameters for all the free variables of the procedure being lifted, then pass along the values for parameters from the point of closure creation. In general, procedure lifting requires the computation of a *transitive closure* (because lifting one procedure may render another procedure's previously-bound identifiers free). That is, the Web program ought to become:

```
(define (f2 •1 •2)
  (web-display
    (+ •1
       •2)))
(define (f1 •1)
  (web-read/r "Second number: "
              "f2"))
(web-read/r "First number: "
            "f1")
```

But how is *f2* to get this extra argument? Recall that each top-level procedure takes only one argument: the user's input. The (traditional) Web server can't know that it has to hold on to this value and communicate it.

In practice, these values become the source of values to be stored in hidden fields. Every top-level receiver has to be sensitive to creating and extracting these form values. Specifically, the converted Web application has the following form:

```
(define (f2 user-input)
  (local ([define •1 (get-form-field user-input 'n1)]
          [define •2 (get-form-field user-input 'n2)])
    (web-display
      (+ •1
         •2))))
(define (f1 user-input)
  (web-read/r/fields "Second number: "
                     "f2"
                     user-input
                     (list 'n1)))
(web-read/r "First number: "
            "f1")
```

where n_1 and n_2 are the names used in the form. The procedure *web-read/r/fields* takes the same first two arguments as *web-read/r*. The third argument is the data structure representing the user's input. This is

followed by a list of field names; these fields are extracted from the user input and inserted into the generated HTML form using hidden fields.

How would *f1* know which parameters must be passed to *f2* using the hidden fields? These are precisely those identifiers that are free in the receivers.

Exercise 16.4.1 Automate *this transformation, i.e., write a program that implements it without the need for human intervention.*

Chapter 17

More Web Transformation

We have already seen how the application

```
(web-display
  (+ (web-read "First number: ")
     (web-read "Second number: ")))
```

must be transformed into

```
(web-read/k "First number: "
            (lambda (•1)
              (web-read/k "Second number: "
                          (lambda (•2)
                            (web-display
                              (+ •1 •2))))))
```

to execute on the Web. Let us now examine some more applications of a more complex flavor.

17.1 Transforming Library and Recursive Code

Suppose we have the procedure *tally*. It consumes a list of items and prompts the user for the cost of each item. When done, it generates the sum of these items, which a programmer could display on the Web. The code for *tally* is as follows:

```
(define (tally item-list)
  (if (empty? item-list)
      0
      (+ (web-read (generate-item-cost-prompt (first item-list)))
         (tally (rest item-list)))))
```

This version of *tally* is clearly not Web-friendly, due to the use of *web-read*, which we do not know how to implement. We must therefore transform this code.

The first thing to observe is that on its own, *tally* is not a complete program: it doesn't do anything! Instead, it is a library procedure that may be used in many different contexts. Because it has a Web interaction, however, there is the danger that at the point of interaction, the rest of the computation—i.e., the computation that invoked *tally*—will be lost. To prevent this, *tally* must consume an extra argument, a receiver, that represents the rest of the pending computation. To signify this change in contract, we will use the convention of appending */k* to the name of the procedure and *k* to name the receiver parameter.

```
(define (tally/k item-list k)
  (if (empty? item-list)
      0
      (+ (web-read (generate-item-cost-prompt (first item-list)))
         (tally (rest item-list)))))
```

What is the first thing this procedure does? It checks for whether the list is empty. Does this involve any Web interaction? Clearly not; all the data and primitives are available locally. If the list is not empty, then *tally* prompts for a user input through the Web. This must happen through *web-read/k*. What is the receiver of this Web invocation? That is, what computation remains to be done? Clearly the recursive invocation of *tally*; but there is also the receiver, *k*, which represents the rest of the waiting computation. Therefore, the Web-friendly version of *tally* appears to become

```
(define (tally/k item-list k)
  (if (empty? item-list)
      0
      (web-read/k (generate-item-cost-prompt (first item-list))
                  (lambda (v)
                    (+ v
                       (tally/k (rest item-list)
                                k))))))
```

We can read the second argument to *web-read/k* as saying: "Consume the value provided by the user and add it to the value generated by the recursion. The receiver in the recursive invocation is the same *k* as before, because the computation pending outside the procedure has not changed."

This may look reasonable, but it suffers from an immediate problem. When the recursive call occurs, if the list had two or more elements, then there will immediately be another Web interaction. Because this will terminate the program, the pending addition will be lost! Therefore, the addition of *v* has to move *into the receiver fed to tally/k*. In code:

```
(define (tally/k item-list k)
  (if (empty? item-list)
      0
      (web-read/k (generate-item-cost-prompt (first item-list))
                  (lambda (first-item-cost)
                    (tally/k (rest item-list)
                             (lambda (tally-of-remaining-costs)
                               (k (+ first-item-cost
```

tally-of-remaining-costs))))

That is, the receiver of the Web interaction is invoked with the cost of the first item. When *tally/k* is invoked recursively, it is applied to the rest of the list. Its receiver must therefore receive the tally of costs of the remaining items. That explains the pattern in the receiver.

The only problem is, where does a receiver ever get a value? We create larger-and-larger receivers on each recursive invocation, but what ever invokes them?

Here is the same problem from a different angle (that also answers the question above). Notice that each recursive invocation of *tally/k* takes place in the aftermath of a Web interaction. We have already seen how the act of Web interaction terminates the pending computation. Therefore, when the list empties, where is the value 0 going? Presumably to the pending computation—but there is none. Any computation that *would* have been pending has now been recorded in *k*, which is expecting a value. Therefore, the correct transformation of this procedure is

```
(define (tally/k item-list k)
  (if (empty? item-list)
      (k 0)
      (web-read/k (generate-item-cost-prompt (first item-list))
                  (lambda (first-item-cost)
                    (tally/k (rest item-list)
                             (lambda (tally-of-remaining-costs)
                               (k (+ first-item-cost
                                     tally-of-remaining-costs))))))))
```

Now we have a truly reusable abstraction. Whatever the computation pending outside the invocation of *tally/k*, its proper Web transformation yields a receiver. If this receiver is fed as the second parameter to *tally/k*, then it is guaranteed to be invoked with the value that *tally* would have produced in a non-Web (e.g., console) interaction. The pattern of receiver creation within *tally/k* ensures that no pending computation gets lost due to the behavior of the Web protocol.

Exercise 17.1.1 *There is a strong formal claim hidden behind this manual transformation: that the value given to the initial k fed to tally/k is* the same *as that returned by tally in the non-Web version. Prove this.*

17.2 Transforming Multiple Procedures

Suppose we have the procedure *total+s&h*. It consumes a list of items to purchase, queries the user for the cost of each item, then generates another prompt for the corresponding shipping-and-handling cost,[1] and finally prints the result of adding these together. The procedure *total+s&h* relies on *tally* to compute the sum of the goods alone.

```
(define (total+s&h item-list)
```

[1]The term *shipping and handling* refers to a cost levied in the USA by companies that handle long-distance product orders placed by the mail, phone and Internet. It is ostensibly the price of materials to package and labor to dispatch the ordered goods. This rate is usually a (step) procedure of the cost of items ordered, and must hence be calculated at the end of the transaction.

```
(local ([define total (tally item-list)])
  (+ (web-read (generate-s&h-prompt total))
     total)))
```

Just as we argued in the transformation of *tally*, this procedure alone does not constitute a computation. It must therefore consume an extra parameter, representing a receiver that will consume its result. Likewise, it cannot invoke *tally*, because the latter performs a Web interaction; it must instead invoke *tally/k*, passing along a suitable receiver to ensure no computation is lost.

```
(define (total+s&h/k item-list k)
  (local ([define total (tally/k item-list ???)])
    (+ (web-read (generate-s&h-prompt total))
       total)))
```

Reasoning as before, what is the first thing *total+s&h/k* does? It invokes a procedure to compute the tally. Because this procedure involves a Web interaction, it must be invoked appropriately. That is, the transformed procedure must take the form

```
(define (total+s&h/k item-list k)
  (tally/k item-list
           (lambda (tally-of-items)
             ???)))
```

What is the pending computation? It is to bind the resulting value to *total*, then perform another Web interaction:

```
(define (total+s&h/k item-list k)
  (tally/k item-list
           (lambda (tally-of-items)
             (local ([define total tally-of-items])
               ???))))
```

(Notice that the Web transformation has forced us to give names to intermediate results, thereby rendering the name *total* unnecessary. We will, however, leave it in the transformed program so that the transformation appears as mechanical as possible.) With the pending computation, this is

```
(define (total+s&h/k item-list k)
  (tally/k item-list
           (lambda (tally-of-items)
             (local ([define total tally-of-items])
               (web-read/k (generate-s&h-prompt total)
                           (lambda (s&h-amount)
                             (k (+ s&h-amount
                                   total)))))))))
```

Notice how *total+s&h/k* had to create a receiver to pass to *tally/k*, the transformed version of *tally*. Reading this receiver, it says to consume the value computed by *tally/k* (which it binds to *tally-of-items*), ask the user

to enter the shipping-and-handling amount, compute the final total, and convey this amount to the initial receiver.

It's easy to forget this last step: to apply *k*, the initial receiver supplied to *total+s&h/k*, to the final value. Doing so would effectively "forget" all the computation that was waiting for the result of *total+s&h/k*, i.e., the computation awaiting the result of *total+s&h* in the original program. This is obviously undesirable.

You might worry that the **local** might be "forgotten" by the *web-read/k* that follows. But all we care about is that the name *total* be associated with its value, and the receiver will take care of that (since it is a closure, it must be closed over the value of *total*).

17.3 Transforming State

Suppose we want to write a program that keeps track of an account's balance. On every invocation it presents the current balance and asks the user for a change (i.e., deposit or withdrawal, represented respectively by positive and negative numbers). We would like the Web application to look like this:

```
(define account
  (local ([define balance 0])
    (lambda ()
      (begin
        (set! balance (+ balance
                         (web-read
                           (format "Balance: ~a; Change" balance))))
        (account)))))
```

Note that *account* is bound to a closure, which holds a reference to *balance*. Recall that mutable variables introduce a distinction between their location and the value at that location. The closure closes over the location, while the store is free to mutate underneath. Thus, even though *balance* always refers to the same location, its value (the actual account balance) changes with each interaction.

How do we transform this program? Clearly the procedure bound to *account* must take an additional argument to represent the remainder of the computation:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (set! balance (+ balance
                         (web-read
                           (format "Balance: ~a; Change" balance))))
        (account/k ???)))))
```

More importantly, we must move the *web-read* to be the first action in the procedure:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
```

```
(begin
  (web-read/k (format "Balance: ~a; Change" balance)
              (lambda (v)
                (begin
                  (set! balance (+ balance v))
                  (account/k ???)))))))
```

What's left is to determine what argument to pass as the receiver in the recursive call. What new pending activity have we created? The only thing the procedure does on each recursion is to mutate *balance*, which is already being done in the receiver to the Web interaction primitive. Therefore, the only pending work is whatever was waiting to be done before invoking *account/k*. This results in the following code:

```
(define account/k
  (local ([define balance 0])
    (lambda (k)
      (begin
        (web-read/k (format "Balance: ~a; Change" balance)
                    (lambda (v)
                      (begin
                        (set! balance (+ balance v))
                        (account/k k))))))))
```

The closure created as the receiver for the Web interaction has a key property: it closes over the location of *balance*, not the value. The value itself is stored in the heap memory that is kept alive by the Web server.

Exercise 17.3.1 *If we wanted to run this application without any reliance on a custom server (Section 16.4), we would have to put these heap data somewhere else. Can we put them in hidden fields, as we discussed in Section 16? If so, test and make sure this works correctly, even in the face of the user interactions we discussed in Section 15. If it fails, find an alternative to hidden fields that does work appropriately!*

17.4 The Essence of the Transformation

By now we have performed the Web transformation often enough that we should begin to see a clear pattern between the original and transformed procedure. Essentially, given a procedure f and its transformed version *f/k*, we have that if the application

$(f\ v_1 \ldots v_m)$

would have computed the answer a, then executing

$(f/k\ v_1 \ldots v_m\ k)$

results in k being applied to the value a. This invariant ensures that, provided k is properly stored somewhere, applying it to the value when it becomes available will leave the result of the computation unaffected.

17.5 Transforming Higher-Order Procedures

Suppose our Web program were the following:

```
(define (get-one-temp c)
  (web-read (format "Temperature in city ~a" c)))

(web-display
  (average
    (map get-one-temp
         (list "Bangalore" "Budapest" "Houston" "Providence"))))
```

(Assume we've define *average* elsewhere, and that it performs no Web input-output.) In principle, converting this program is merely an application of what we studied in Section 17.1 and Section 17.2, but we'll work through the details to reinforced what you read earlier.

Transforming *get-one-temp* is straightforward:

```
(define (get-one-temp/k c k)
  (web-read/k (format "Temperature in city ~a" c)
              k))
```

This means we must invoke this modified procedure in the *map*. We might thus try

```
(web-display
  (average
    (map get-one-temp/k
         (list "Bangalore" "Budapest" "Houston" "Providence"))))
```

Unfortunately, *map* is expecting its first argument, the procedure, to consume only the elements of the list; it does not provide the second argument that *get-one-temp/k* needs. So Scheme reports

```
map: arity mismatch for procedure get-one-temp/k: expects 2 arguments, given 1
```

It therefore becomes clear that we must modify *map* also. Let's first write *map* in full:

```
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l))
            (map f (rest l)))))
```

Clearly we must somehow modify the invocation of f. What can we pass as a second argument? Here's one attempt:

```
(define (map f l)
  (if (empty? l)
      empty
      (cons (f (first l) (lambda (x) x))
            (map f (rest l)))))
```

That is, we'll pass along the identity procedure. Does that work? Think about this for a moment.

Let's try testing it. We get the following interaction:

```
Language: PLAI - Advanced Student.
web-read/k: run (resume) to enter number and simulate clicking Submit
>
```

This means the program has arrived at *web-read/k* for the first time. We run

```
> (resume)
```

which prompts us for an input. Suppose we enter `25`. We then see

```
Temperature in city Bangalore: 25
25
>
```

It stopped: the program terminated without ever giving us a second Web prompt and asking us for the temperature in another city!

Why? Because the value of the receiver stored in the hash table or box is the identity procedure. When computation resumes (on the user's submission), we expect to find the closure representing the *rest of the computation*. Since the stored closure is instead just the identity procedure, the program terminates thinking its task is done.

This gives us a pretty strong hint: the receiver we pass had better make some reference to *map*, and indeed, had better continue the iteration. In fact, let's think about where we get the first value for *cons*. This value is the temperature for a city. It must therefore come from *web-read/k*. But that is exactly the value that *web-read/k* supplies to its receiver. Therefore, everything starting with the *cons* onward must move to within the closure:

```
(define (map f/k l)
  (if (empty? l)
      empty
      (f/k (first l)
           (lambda (v)
             (cons v
                   (map f (rest l)))))))
```

This version is still not quite okay. This is because the recursive call invokes *map*, which suffers from the same problem we have just discussed above. Indeed, running this version teminates after reading the temperature for the second city, and returns just a list containing the second city's temperature! Instead, it must invoke a modified version of *map*, namely *map/k*, with an appropriate additional argument:

```
(define (map/k f/k l k)
  (if (empty? l)
      empty
      (f/k (first l)
```

```
(lambda (v)
  (cons v
        (map/k f/k (rest l) ???)))))
```

We must determine what to pass as an argument in the recursive call. But before we do that, let's study what we have written carefully. When the first Web interaction results in a response, the server will invoke the (**lambda** (*v*) ···). This *cons*es the input temperature to the value of the recursive call. The recursive call will, however, eventually result in an invocation of *web-read/k*. That invocation will halt the program. Once the program halts, we lose record of the *cons*. So this program can't work either! We must instead move the *cons* inside the receiver, where it won't be "forgotten".

Given what we've learned in Section 17.4, it makes sense to think of the value given on invoking *map/k* on the rest of the list as the list of temperatures for the remaining cities. Therefore, we simply need to *cons* the temperature for the first city onto this result:

```
(define (map/k f/k l k)
  (if (empty? l)
      empty
      (f/k (first l)
           (lambda (v)
             (map/k f/k (rest l)
                    (lambda (v-rest)
                      (cons v v-rest)))))))
```

Now we're ready to modify the main program. We had previously written

```
(web-display
  (average
    (map get-one-temp/k
         (list "Bangalore" "Budapest" "Houston" "Providence"))))
```

We have to convert the invocation of *map* to one of *map/k* and, in turn, determine what to pass as the second argument to *map/k*. Using the same reasoning we have employed before (in particular that, as written, the *web-display* and *average* procedures will never execute, since they will be forgotten when the server terminates the program), we know to write this:

```
(map/k get-one-temp/k
       (list "Bangalore" "Budapest" "Houston" "Providence")
       (lambda (v)
         (web-display
           (average v))))
```

This program now runs through the four cities, accepts the temperatures in order, and produces ... the empty list.

What went wrong here? We can reason this way. The empty *list* cannot result from *average* (which must produce a number), so we can reason that the initial receiver must never have been invoked at all. (We can verify this by commenting out the definition of *average* and noticing that this doesn't cause a problem: the

procedure is never invoked.) So it must be the case that the receiver supplied to *map/k* never made it any further.

Studying *map/k*, we see the problem. Though the procedure consumes a receiver, that receiver never gets used anywhere in its body. In fact, we should be passing the result of the *cons* to this procedure:

```
(define (map/k f/k l k)
  (if (empty? l)
      empty
      (f/k (first l)
           (lambda (v)
             (map/k f/k (rest l)
                    (lambda (v-rest)
                      (k (cons v v-rest)))))))))
```

Everything now looks hunky-dory, so we run the program, enter the four temperatures, and still get ... the empty list!

Since there is really only one place in the program where we explicitly mention the empty list, we might suspect it now. Indeed, the first branch in the conditional of *map/k* is indeed the culprit. When a value becomes available, we should not return it. Why not? Because we know no procedure is awaiting it directly. Why not? Because according to the Web protocol, any waiting procedures would have terminated when the whole program terminated at the previous interaction point! Therefore, to *return* a value, a procedure must instead *hand the value to the receiver*. (Of course, this merely reiterates the invariant we established in Section 17.4.) That is, we want

```
(define (map/k f/k l k)
  (if (empty? l)
      (k empty)
      (f/k (first l)
           (lambda (v)
             (map/k f/k (rest l)
                    (lambda (v-rest)
                      (k (cons v v-rest)))))))))
```

The moral of this lengthy story is that, to make a program Web-ready, we must (a) generate receivers that capture pending computations, and (b) pass values to receivers instead of returning them. In rare cases, a procedure will neither return a value nor generate additional pending computation—*get-one-temp* is a good example—in which case, its transformed version will consume a receiver and pass along *the same* receiver to other computations (as *get-one-temp/k* does).

Exercise 17.5.1 *Why did we not transform average? In general, what principle guides whether or not we transform a given procedure? (Make sure your principle also applies to map!)*

17.6 Perspective on the Web Transformation

Notice three implications of the transformation the Web forces us to employ:

1. We have had to make decisions about the order of evaluation. That is, we had to choose whether to evaluate the left or the right argument of addition first. This was an issue we had specified only implicitly earlier; if our evaluator had chosen to evaluate arguments right-to-left, the Web program at the beginning of this document would have asked for the second argument before the first! We have made this left-to-right order of evaluation explicit in our transformation.

2. The transformation we use is global, namely it (potentially) affects all the procedures in the program by forcing them all to consume an extra receiver as an argument. We usually don't have a choice as to whether or not to transform a procedure. Suppose f invokes g and g invokes h, and we transform f to f/k but don't transform g or h. Now when f/k invokes g and g invokes h, suppose h consumes input from the Web. At this point the program terminates, but the last receiver procedure (necessary to resume the computation when the user supplies an input) is the one given to f/k, with all record of g and h erased.[2]

3. This transformation sequentializes the program. Given a nested expression, it forces the programmer to choose which sub-expression to evaluate first (a consequence of the first point above); further, every subsequent operation lies in the receiver, which in turn picks the first expression to evaluate, pushing all other operations into its receiver; and so forth. The net result is a program that looks an awful lot like a traditional procedural program. This suggests that this series of transformations can be used to compile a program in a language like Scheme into one in a language like C!

Exercise 17.6.1 *This presentation has intentionally left out the contracts on the procedures. Add contracts to all the procedures—both the original programs and the Web versions.*

Exercise 17.6.2 *Adding contracts to the Web versions (Exercise 17.6.1) reveals a very interesting pattern in the types of the receivers. Do you see a connection between this pattern and the behavior of the Web?*

[2]Indeed, we would have encountered an error even earlier, when the transformed version of f, namely f/k, tried to invoke g with an extra receiver argument that g was not transformed to accept. In this even simpler way, therefore, the transformation process has a cascading effect.

Chapter 18

Conversion into Continuation-Passing Style

Given how much the Web distorts the structure of an application, we would benefit from automating the Web transformation. Then we could program in the lucid style of the initial versions of the programs, and leave it to a "Web compiler" to transform these into their corresponding Web forms. This compiler would be responsible for creating the receivers automatically. With a few further steps we could also implement lifting and translation into the Web protocol (e.g., employing hidden fields).

To build such a compiler, we must better understand the receiver. Each expression's receiver is a procedure representing the *rest of the computation to be performed when this expression completes*. Furthermore, we have seen that once we have converted applications use the receiver, it is useless to return values in the traditional fashion, because all the code that would have otherwise received this return value is now encapsulated in the receiver. Traditionally, the return value propagates using the *control stack*, which represents the remainder of the computation. The receiver is therefore a *procedural representation of the stack*.

Programs that obey this stylized form of generating and using receivers are said to be in *continuation-passing style*, or CPS. Note that CPS is a *style of program*; many programs can be in CPS. Any program that satisfies the programming pattern that its stack is represented procedurally can be said to be in CPS. More to the point, any program that is not in CPS can be converted into CPS. This is in fact precisely the essence of the Web compiler: it is a program that converts an arbitrary source program into CPS.[1]

18.1 The Transformation, Informally

The translations we have done by hand have been rather informal in nature. If we wish to automate this transformation, we must adhere to more rigid rules. For instance, we have to agree upon a uniform representation for all CPS terms, so any program processing them knows what to both generate and expect.

Consider our running example of addition. Given this term,

```
(+ (web-read "First number: ")
   (web-read "Second number: "))
```

[1] We will often find it convenient to have a verb form to represent the act of performing this transformation. It is common, *par abus de langage*, to use CPS itself as a verb (as in, "CPS the following program"). Remember, though, that in proper usage, CPS itself names a form of program, not an algorithm or procedure used to convert programs into that form.

we hand-translated it to the following:

```
(web-read/k "First number: "
            (lambda (l-val)
              (web-read/k "Second number: "
                          (lambda (r-val)
                            (+ l-val r-val)))))
```

This hand-translation is sufficient if this expression is the entire program. If, however, we wish to use it in a broader context (e.g., as a sub-expression in a larger program), this does not suffice, since it does not recognize that there may be a pending computation outside its own evaluation. How can we make this translated expression reflect that fact? We have to introduce a procedure that consumes a receiver, and uses that receiver to communicate the computed value:

```
(lambda (k)
  (web-read/k "First number: "
              (lambda (l-val)
                (web-read/k "Second number: "
                            (lambda (r-val)
                              (k (+ l-val r-val)))))))
```

In particular, if a transformer were (recursively) invoked upon the sub-expression

```
(+ (web-read "First number: ")
   (web-read "Second number: "))
```

it would need to return

```
(lambda (k)
  (web-read/k "First number: "
              (lambda (l-val)
                (web-read/k "Second number: "
                            (lambda (r-val)
                              (k (+ l-val r-val)))))))
```

which can then be employed in the transformation of a larger program. (Observe that in the special case where this is the entire program, applying this transformed term to the identity procedure results in the same result as our original manual transformation.)

The following convention is therefore particularly useful: Every term, when converted to CPS, will be represented as a procedure. This procedure will consume one argument, its receiver. The converted body will communicate any value it computes to this receiver; if the body requires further computation, this will be done using an augmented version of the receiver (i.e., no prior computation will be lost).

Let us similarly consider the transformation of a procedure application. For simplicity, we will assume that procedures have only one parameter. Recall that not only the parameter position, but even the procedure position, of an application can be an arbitrary expression and must therefore be reduced to a value. For example, this expression could be an excerpt from the core of a parser:

```
((extract-from-table next-char)
```

(*get-lookahead LOOKAHEAD-CONSTANT*))

The next character is used as an index into a table, and the procedure obtained is applied to a constant number of lookahead characters. The transformation of the procedure position should be

```
(extract-from-table/k next-char
                     (lambda (f-val)
                       ???))
```

Similarly, the transformation of the argument position would be

```
(get-lookahead/k LOOKAHEAD-CONSTANT
                 (lambda (a-val)
                   ???))
```

Given these two values (*f-val*, the procedure, and *a-val*, the actual parameter), we can now perform the application. The following looks reasonable:

(*k* (*f-val a-val*))

Unfortunately, if the procedure itself performs a Web interaction, then that will halt the computation, erasing any record of returning the value to *k*. Instead, therefore, *k* must be given as an argument to the procedure, which can then use it appropriately. Putting together the pieces, the transformation into CPS of the procedure application above is

```
(lambda (k)
  (extract-from-table/k next-char
                       (lambda (f-val)
                         (get-lookahead/k LOOKAHEAD-CONSTANT
                                          (lambda (a-val)
                                            (f-val a-val k))))))
```

Reading this sequentially, it says to evaluate the procedure expression, and store its value in *f-val*; then evaluate the argument, and store its value in *a-val*; and finally invoke the procedure on the argument. This procedure's receiver is the same as that of the procedure application itself.

What do we do with variables and simple constants? Recall that every term in CPS must be a procedure that consumes a receiver. Since there is no computation to perform, the constant is simply communicated to the receiver. Thus, the CPS representation of 3 is just

```
(lambda (k)
  (k 3))
```

Suppose we are given the procedure (**lambda** (*x*) *x*). What does it look like in CPS? Since every **lambda** expression is also a constant, it is tempting to use the same rule above for these expressions also, i.e.:

```
(lambda (k)
  (k (lambda (x) x)))
```

However, the transformation is more subtle than that. Observe that a procedure application invokes the procedure on two arguments, not one, whereas the procedure (**lambda** (*x*) *x*) consumes only one. What is

the second argument? It's the *dynamic* receiver: i.e., the receiver at the time of the procedure *application* (as opposed to its definition). Furthermore, we don't want to ignore this receiver: it's the stack active at the point of procedure invocation, so we want to preserve it. This is in direct contrast to what we did with environments—there we wanted the static environment—and more in line with our treatment of the store, where we wanted the current store at the time of procedure application, and therefore did not close over the store. Similarly, we do not close over the receiver at the point of procedure creation. The transformed version instead reads

```
(lambda (k)
  (k (lambda (x dyn-k)
       ( (lambda (k) (k x))
         dyn-k))))
```

where the boxed expression is the result of transforming the body, namely, *x*. This is equivalent (when the inner application finally happens) to

```
(lambda (k)
  (k (lambda (x dyn-k)
       (dyn-k x))))
```

That is, it's a procedure that accepts a value and a dynamic receiver and sends the value to that receiver. This, of course, is precisely the behavior we would expect of the identity function.

That leaves only one kind of expression to translate, namely *web-read*. We are in a happy position: to translate *web-read* into *web-read/k* we need access to the receiver, which is precisely what our transformation has given us access to. Therefore, an expression such as

```
(web-read "First number: ")
```

simply becomes

```
(lambda (k)
  (web-read/k "First number: "
              k))
```

18.2 The Transformation, Formally

For this material, please switch to the PLAI - Pretty Big language level.

We are now ready to write this transformation formally.

In general, every program transformer is a kind of compiler. The transformer of a program into CPS is, however, a peculiar kind of compiler: it is a *source-to-source* transformation. That is, the transformer consumes a program in a source language and generates a new program in the same language (assuming the language has closures and some other properties copacetic to the use of CPS).

Scheme provides a particularly convenient mechanism, called a *macro*, for writing source-to-source transformers. In this section, we present the transformer into CPS using macros but without much elaboration of the macro mechanism. We will discuss macros in considerably more detail in Section 36.

Macros are triggered by a keyword that signals the transformation. We will use the keyword **cps**, so that for every term of the form (**cps** *e*), the expression *e* will be transformed into CPS. Every macro definition consists of a preamble naming the new keyword being defined and a list of rules dictating how to transform source expressions. The rules employ pattern-matching to extract portions of the input expression for insertion into the output.

The **cps** macro's header has the following form:

```
(define-syntax cps
  (syntax-rules (+ lambda web-read)
```

This says we are defining a macro for the keyword **cps**, and that this macro will treat +, **lambda** and **web-read** specially: they must match exactly against an input for the corresponding rule to apply. The macro's rules are as follows. Note that these are simply the generalization of the instances we studied above.

First, the rule for addition:

```
[(cps (+ e1 e2))
 (lambda (k)
   ((cps e1) (lambda (l-val)
               ((cps e2) (lambda (r-val)
                           (k (+ l-val r-val)))))))]
```

This says that whenever the term has the form

```
(cps (+ e1 e2))
```

where + is expected to match precisely, the sub-expressions (which can be arbitrary code) are named *e1* and *e2* respectively. The corresponding transformed term has the form

```
(lambda (k)
  ((cps e1) (lambda (l-val)
              ((cps e2) (lambda (r-val)
                          (k (+ l-val r-val)))))))
```

where *e1* and *e2* are inserted into the transformed term. Since they are within a (**cps** ·) expression, they will be transformed into CPS recursively by the same macro.

The transformer for a procedure application is as follows:

```
[(cps (f a))
 (lambda (k)
   ((cps f) (lambda (f-val)
              ((cps a) (lambda (a-val)
                         (f-val a-val k))))))]
```

following the pattern: convert the procedure expression and apply it to a receiver expecting the procedure value; do likewise for the argument; and when both values are handy, apply the procedure to the current receiver and the argument value.

The corresponding transformation for a procedure is therefore

```
[(cps (lambda (a) body))
```

```
(lambda (k)
  (k (lambda (a dyn-k)
       ((cps body) dyn-k))))]
```

Recall that every user procedure must now consume the dynamic receiver in addition to its formal parameter.

As we have discussed, the transformation of **web-read** is especially simple:

```
[(cps (web-read prompt))
 (lambda (k)
   (web-read/k prompt k))]
```

Finally, if all other terms fail to match, we assume that the source program is a simple value (namely an identifier or a non-procedural constant such as a string or number). The corresponding term in CPS consumes a receiver (to maintain the consistency of the interface) and immediately sends the value to the receiver:

```
[(cps v)
 (lambda (k) (k v))]
```

Figure 18.1 presents the entire CPS macro.

Example Given these rules, our transformer will convert

```
(+ (web-read "First number: ")
   (web-read "Second number: "))
```

into

```
(lambda (k1)
  ((lambda (k2)
     (web-read/k "First number: " k2))
   (lambda (l-val)
     ((lambda (k3)
        (web-read/k "Second number: " k3))
      (lambda (r-val)
        (k1 (+ l-val r-val)))))))
```

This may look rather more complicated than we are used to. However, we merely need to perform the inner procedure applications, substituting the known receivers for *k2* and *k3*. Doing this yields:

```
(lambda (k1)
  (web-read/k "First number: "
              (lambda (l-val)
                (web-read/k "Second number: "
                            (lambda (r-val)
                              (k1 (+ l-val r-val)))))))
```

which is exactly what we expect!

The Fischer CPS Transformation

The CPS transformer we have studied here is one of the very oldest, due to Michael Fischer. In the three decades since Fischer defined this transformation, there has been considerable research into building a better CPS transformation. Why? This version, while easy to understand, introduces a considerable amount of overhead: look at all the procedures in the output that weren't in the source program! In principle, we would like a transformer that approximates the cleanliness of hand-translation: this would be useful both to humans who must read the output and to programs that must process it. This is what the better transformers strive to achieve.

18.3 Testing

The code in Figure 18.1 also defines **define-cps** so you can write top-level definitions that are translated into CPS. Thus, you can write the following program, reminiscent of the cascading transformation we discussed in Section 17.6:

(**define-cps** (*g x*) (+ *x x*))
(**define-cps** (*h f*) (**lambda** (*x*) (*f x*)))
(**define-cps** (*dummy x*) ((*h g*) 10))

It also defines the **run** construct, which is useful for testing code converted to CPS. You can use it to execute

(**run** (*dummy* 1729))

and observe the computation terminate with the value 20.

Exercise 18.3.1 *Suppose, instead, the* CPS *rule for a procedure were*

```
[(cps (lambda (a) body))
 (lambda (k)
   (k (lambda (a dyn-k)
        ((cps body) k)))))]
```

i.e., the transformed procedure ignored the dynamic receiver and used the static one instead. What impact would this have on program behavior? Predict, then run and check!

```
(define-syntax define-cps
  (syntax-rules ()
    [(define-cps (f arg) body)
     (define-cps f (lambda (arg) body))]
    [(define-cps v val)
     (define v ((cps val) (lambda (x) x)))]))

(define-syntax cps
  (syntax-rules (+ lambda web-read)
    [(cps (+ e1 e2))
     (lambda (k)
       ((cps e1) (lambda (l-val)
                   ((cps e2) (lambda (r-val)
                               (k (+ l-val r-val)))))))]
    [(cps (lambda (a) body))
     (lambda (k)
       (k (lambda (a dyn-k)
            ((cps body) dyn-k))))]
    [(cps (web-read prompt))
     (lambda (k)
       (web-read/k prompt k))]
    [(cps (f a))
     (lambda (k)
       ((cps f) (lambda (f-val)
                  ((cps a) (lambda (a-val)
                             (f-val a-val k)))))))]
    [(cps v)
     (lambda (k) (k v))]))

(define-syntax run
  (syntax-rules ()
    [(run e) ((cps e)
              (lambda (x)
                (error "terminating with value" x)))]))
```

Figure 18.1: Implementation of CPS Converter

Chapter 19

Programming with Continuations

For this material, please switch to the PLAI - Pretty Big language level.

In Section 18 we saw how conversion to CPS restores the Web programming interface we desire: programmers can use **web-read** and not have to "invert" the structure of the program. While in principle this accomplishes the task, in practice conversion to CPS has several disadvantages:

1. It requires access to the source of the entire program. If a procedure is defined in a library for which we don't have access to the source, or is perhaps written in a different language (as *map* often is), then the CPS translator will either fail to run or will produce potentially erroneous output (i.e., code that does not properly restore the state of the computation).

2. By replacing the machine's stack with an explicit representation in the form of receivers, it inhibits optimizations built into compilers and microprocessor architectures.

3. As we will see in Section 20.4, executing a program in CPS also assumes that the run-time system will not needlessly create stack frames (since the stack is entirely represented by the receiver). Since many languages (such as C and Java) do anyway, the program consumes memory unnecessarily. In an extreme case, a Java or C program that would have executed without exhausting memory will run out of memory after conversion into CPS.

The first of these problems is particularly compelling, since it affects not only performance but even correctness. We would benefit from an operation that automatically constructs the receiver at any point during the program's execution, instead of expecting it to have already been created through a static compilation process.

Some programming languages, notably Scheme, have such an operation. This operation creates a representation of the "rest of the computation" (which is what the receiver represented) as a procedure of one argument. Giving that procedure a value causes the remaining computation to resume with that value. This procedure is called a *continuation*. This explains where CPS obtains its name, but note that the program does not need to be transformed a priori; the continuation is created *automatically* and *on-the-fly*. As you might imagine, creating (or "capturing") a continuation simply involves copying the stack, though there are less and more efficient ways of obtaining the same effect.

Adding continuations to a language makes it easy to create a better Web programming protocol, as we shall see. But just as laziness—which was already present in the shell but was essentially an extra-lingual feature (since programmers could not explicitly control it)—, once exposed as a feature in a programming language, gave programmers immense power in numerous contexts, so do continuations. We will explore this power in greater detail.

19.1 Capturing Continuations

In Scheme, we create a value representing the continuation using one of two related constructs. The traditional form is called *call/cc*, short for "call with current continuation". *call/cc* consumes a procedure as an argument, and invokes this procedure with a continuation. That is, uses of *call/cc* typically look like

(*call/cc*
  (**lambda** (*k*)
    $\cdots$ *k* $\cdots$)) ;; *k* is the continuation

Because the extra **lambda** is extremely tiresome, however, Scheme provides a nicer interface to capturing the current continuation: you may instead *equivalently* write

(**let/cc** *k*
  $\cdots$ *k* $\cdots$)) ;; *k* is bound to the continuation

Note that **let/cc** is a *binding* construct: it introduces a new scope, binding the named identifier (*k*, above) in that context. For the rest of this material, we'll use **let/cc** rather than *call/cc*.[1]

19.2 Escapers

Let's write some programs using continuations. What is the value of this program?

(**let/cc** *k*
  (*k* 3))

We must first determine the continuation bound to `k`. This is the same procedure as the value of the receiver in CPS. Since in this case, there is no computation waiting to be done outside the **let/cc** expression, the receiver would be the initial receiver, namely the identity function. Therefore, this receiver is

(**lambda** (•)
  •)

Applying this to 3 produces the answer 3.

Consider this program:

(+ 1
  (**let/cc** *k*

[1] So why does Scheme offer *call/cc*, ghastly as it looks? Historically, the original standards writers were loath to add new binding forms, and the use of **lambda** meant they didn't need to create one. Also, *call/cc* lets us create some incredibly clever programming puzzles that we can't write quite as nicely with **let/cc** alone! Ask us for some.

```
    (k 3)))
```

What is the continuation this time? It's

```
(lambda (•)
  (+ 1 •))
```

(all the code "outside the parens"). This procedure, when invoked, yields the value 4. Because the continuation is *the rest of the computation*, we want to halt with the value 4. But this looks confusing, because substitution gives

```
   (+ 1
      (k 3)) ;; where k is bound to (lambda (•) (+ 1 •))
= (+ 1
      ((lambda (•)
         (+ 1 •))
       3))
= (+ 1
      (+ 1 3))
```

which performs the addition twice, producing the answer 5. The problem is that we're effectively applying the continuation *twice*, whereas computation should halt after it has been applied once. We will use a special notation to reflect this: **lambda**↑ will represent a procedure that, when its body finishes computing, halts the entire computation. We'll call these *escaper* procedures, for obvious reasons. That is, the continuation is really

```
(lambda↑ (•)
  (+ 1 •))
```

so the expression

```
(+ 1
   ((lambda↑ (•)
      (+ 1 •))
    3))
```

evaluates to 4, with the outermost addition ignored (because we invoked an escaper).

19.3 Exceptions

Let's consider a similar, but slightly more involved, example. Suppose we are deep in the midst of a computation when we realize we are about to divide by zero. At that point, we realize that we want the value of the entire expression to be one. We can use continuations to represent this pattern of code:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
```

```
        (/ 1 n)))))
(+ 3 (f 0))
```

The continuation bound to *k* is

```
(lambda↑ (•)
  (+ 3
     (+ 10
        (* 5
           •))))
```

but oops, we're about to divide by zero! Instead, we want the entire division expression to evaluate to one. Here's how we can do it:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc k
          (/ 1 (if (zero? n)
                   (k 1)
                   n))))))
```

so that • in the continuation is substituted with 1, we bypass the division entirely, and the program can continue to evaluate.

Have you seen such a pattern of programming before? But of course: *k* here is acting as an *exception handler*, and the *invocation* of *k* is *raising the exception*. A better name for *k* might be *esc*:

```
(define (f n)
  (+ 10
     (* 5
        (let/cc esc
          (/ 1 (if (zero? n)
                   (esc 1)
                   n))))))
```

which makes pretty clear what's happening: *when you invoke the continuation*, it's as if the entire **let/cc** expression that binds *esc* should be cut out of the program and replaced with the value passed to *esc*, i.e., its as if the actual code for *f* is really this:

```
(define (f n)
  (+ 10
     (* 5
        1)))
```

In general, this "cut-and-paste" semantics for continuations is the simplest way (in conjunction with escaper procedures) of understanding a program that uses continuations.

There was, incidentally, something sneaky about the program above: it featured an expression in the body of a **let/cc** that did *not* invoke the continuation. That is, if you can be sure the user of *f* will never pass an argument of 0, it's as if the body of the procedure is really

```
(define (f n)
  (+ 10
     (* 5
        (let/cc esc
          (/ 1 n)))))
```

but we haven't talked about what happens in programs that don't invoke the continuation. In fact, that's quite easy: the value of the entire **let/cc** expression is exactly that of the value of its body, just as when you don't actually raise an exception.

19.4 Web Programming

Now that we're getting a handle on continuations, it's easy to see how they apply to the Web. We no longer need the procedure *web-read/k*; now, **web-read** can be implemented directly to do the same thing that *web-read/k* was expected to do. **web-read** captures the current continuation, which corresponds to the second argument supplied to *web-read/k* (except the continuation is now captured automatically, instead of having to be supplied as an explicit second argument).

The rest of the implementation is just as before: it stores the continuation in a fresh hash table entry, and generates a URL containing that hash table entry's key. The launcher extracts the continuation from the hash table and applies it to the user's input. As a result, all the programs we have written using **web-read** are now directly executable, without the need for the CPS transformation.

19.5 Producers and Consumers

A number of programs follow a *producer-consumer* metaphor: one process generates (possibly an infinite number of) values, while another consumes them as it needs new ones. We saw several examples of this form in Haskell, for instance. Many client-server programs are like this. Web programs have this form (we, the user, are the supplier of inputs—and on some Web sites, the number really does seem quite close to infinite ...). I/O, in general, works this way. So it's worth understanding these processes at a deeper level.

To avoid wrangling with the complexities of these APIs, we'll reduce this to a simpler problem. We'd like to define a producer of symbols (representing a sequence of cities to visit) that takes one argument, *send*, which masks the details of the API, and sends a sequence of city names on demand:

```
(define (route-producer send)
  (begin
    (send 'providence)
    (send 'houston)
    (send 'bangalore)))
```

That is, when we first invoke *route-producer*, it invokes *send* with the value 'providence—and halts. When we invoke it again, it sends 'houston. The third time we invoke it, it sends the value 'bangalore. (For now, let's not worry about what happens if we invoke it additional times.)

What do we supply as a parameter to elicit this behavior? Clearly, we can't simply supply an argument such as (**lambda** (*x*) *x*). This would cause all three *send* operations to happen in succession, returning the

value of the third one, namely 'bangalore. Not only is this the wrong value on the first invocation, it also produces the same answer no matter how many times we invoke *route-producer*—no good.

In fact, if we want to somehow suspend the computation, it's clear we need one of these exception-like operations so that the computation halts prematurely. So a simple thing to try would be this:

(**let/cc** *k* (*route-producer k*))

What does this do? The continuation bound to *k* is

(**lambda**↑ (•) •)

Substituting it in the body results in the following program:

```
(begin
  ((lambda↑ (•) •)
   'providence)
  ((lambda↑ (•) •)
   'houston)
  ((lambda↑ (•) •)
   'bangalore))
```

(all we've done is substitute *send* three times). When we evaluate the first escaper, the entire computation reduces to 'providence—and computation halts! In other words, we see the following interaction:

```
> (let/cc k (route-producer k))
'providence
```

This is great—we've managed to get the program to suspend after the first *send*! So let's try doing this a few more times:

```
> (let/cc k (route-producer k))
'providence
> (let/cc k (route-producer k))
'providence
> (let/cc k (route-producer k))
'providence
```

Hmm—that should dampen our euphoria a little.

What's the problem? We really want *route-producer* to "remember" where it was when it sent a value so it can send us the *next* value when we invoke it again. But to resume, we must first capture a snapshot of our computation before we sent a value. That sounds familiar. . . .

Going back to *route-producer*, let's think about what the continuation is at the point of invoking *send* the first time. The continuation is

```
(lambda↑ (•)
  (begin
    •
    (send 'houston)
    (send 'bangalore)))
```

(with *send* substituted appropriately). Well, this looks promising! If we could somehow hang on to this continuation, we could use it to resume the producer by sending 'houston, and so forth.

To hang on to the computation, we need to store it somewhere (say in a box), and invoke it when we run the procedure the next time. What is the initial value to place in the box? Well, initially we don't know what the continuation is going to be, and anyway we don't need to worry because we know how to get a value out the first time (we just saw how, above). So we'll make the box initially contain a flag value, such as false. Examining what's in the box will let us decide whether we're coming through the first time or not. If we are we proceed as before, otherwise we need to capture continuations and what-not.

Based on this, we can already tell that the procedure is going to look roughly like this:

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (send)
      (if (unbox resume)
          ;; then
          there's a continuation present—do something!
          ;; else
          (begin
            (send 'providence)
            (send 'houston)
            (send 'bangalore)))))))
```

where the box bound to *resume* stores the continuation. Note that *resume* is outside the **lambda** but in its scope, so the identifier is defined only once and all invocations of the procedure share it.

If (*unbox resume*) doesn't evaluate to false,[2] that means the box contains a continuation. What can we do with continuations? Well, really only one thing: invoke them. So we must have something like this if the test succeeds:

((*unbox resume*) ···)

But what is that continuation? It's one that goes into the midst of the **begin** (recall we wrote it down above). Since we really don't care about the value, we may as well pass the continuation some kind of dummy value. Better to pass something like 'dummy, which can't be confused for the name of a city. So the procedure now looks like

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (send)
      (if (unbox resume)
          ((unbox resume) 'dummy)
          (begin
            (send 'providence)
            (send 'houston)
            (send 'bangalore))))))
```

[2] In Scheme, only false fails a conditional test; all other values succeed.

Of course, we haven't actually done any of the hard work yet. Recall that we said we want to capture the continuation of *send*, but because *send* is given as a parameter by the user, we can't be sure it'll do the right thing. Instead, we'll locally define a version of *send* that does the right thing. That is, we'll rename the *send* given by the user to *real-send*, and define a *send* of our own that invokes *real-send*.

What does our *send* need to do? Obviously it needs to capture its continuation. Thus:

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send (lambda (value-to-send)
                             (let/cc k
                               ···))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
              (send 'providence)
              (send 'houston)
              (send 'bangalore)))))))
```

What do we do with *k*? It's the continuation that we want to store in *resume*:

```
(set-box! resume k)
```

But we also want to pass a value off to *real-send*:

```
(real-send value-to-send)
```

So we just want to do this in sequence (observe that we can't do these in the other order):

```
(begin
  (set-box! resume k)
  (real-send value-to-send))
```

When the client next invokes *route-producer*, the continuation stored in the box bound to *resume* is that of invoking (our locally defined) *send* within the body... which is exactly where we want to resume computation! Here's our entire procedure:

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send (lambda (value-to-send)
                             (let/cc k
                               (begin
                                 (set-box! resume k)
                                 (real-send value-to-send))))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
```

(*send* 'providence)
(*send* 'houston)
(*send* 'bangalore)))))))

Here's the interaction we wanted:[3]

```
> (let/cc k (route-producer k))
'providence
> (let/cc k (route-producer k))
'houston
> (let/cc k (route-producer k))
'bangalore
```

It's a bit unwieldy to keep writing these **let/cc**s, so we can make our lives easier as follows:

(**define** (*get producer*)
  (**let/cc** *k* (*producer k*)))

(we could equivalently just define this as (**define** *get call/cc*)) so that

```
> (get route-producer)
'providence
> (get route-producer)
'houston
> (get route-producer)
'bangalore
```

Exercise 19.5.1 *How would you define the same operators in Haskell?*

Exercise 19.5.2 *Before you read further: do you see the subtle bug lurking in the definitions above?*
Hint: *We would expect to be able to invoke (get route-producer) three times and combine the result into a list.*

[3]It's critical that you work through the actual continuations by hand.

Continuations and "Goto" Statements

What we're doing here has already gone far beyond the simple exception pattern we saw earlier. There, we used continuations only to ignore partial computations. Now we're doing something much richer. We're first executing part of a computation in a producer. At some point, we're binding a continuation and storing it in a persistent data structure. Then we switch to performing some completely different computation (in this case, the top-level requests for the next city). At this point, the partial computation of the producer has seemingly gone away, because we invoked an escaper to return to the top-level. But by accessing the continuation in the data structure, we are able to *resume a prior computation*: that is, we can not only jump "out", as in exceptions, but we can even jump back "in"! We do "jump back in" in the Web computations, but that's a much tamer form of resumption. What we're doing here is resuming between two separate computations!

Some people compare continuations to "goto" statements, but we should think this through. Goto statements can usually jump to an arbitrary line, which means they may continue execution with nonsensical state (e.g., variables may not be properly initialized). In contrast, a continuation only allows you to resume a computational state you have visited before. By being more controlled in that sense, continuations avoid the worst perils of gotos.

On the other hand, continuations are far more powerful than typical goto statements. Usually, a goto statement can only transfer control to a lexically proximate statement (due to how compilers work). In contrast, a computation can represent any arbitrary prior computation and, irrespective of lexical proximity, the programmer can resurrect this computation. In short, continuations are more structured, yet much more powerful, than goto statements.

19.6 A Better Producer

The producer shown above is pretty neat—indeed, we'd like to be able to use this in more complex computations. For instance, we'd like to initialize the producer, then write

```
(list (get route-producer)
      (get route-producer)
      (get route-producer))
```

Evaluating this in DrScheme produces. . . *an infinite loop*!

What went wrong? It's revealing to use this version of the program instead:

```
(list (let/cc k (route-producer k))
      (let/cc k (route-producer k))
      (let/cc k (route-producer k)))
```

After a while, click on Break in DrScheme. If you try it a few times, sooner or later you'll find that the break happens at the *second* **let/cc**—but never the third! In fact, if you ran this program instead:

```
(list (let/cc k (route-producer k))
      (begin
        (printf "got here!~n")
```

```
    (let/cc k (route-producer k)))
  (begin
    (printf "here too!~n")
    (let/cc k (route-producer k))))
```

you'd find several got heres in the in the Interactions window, but no here toos. That's revealing!

What's going on? The first continuation bound to *real-send* in *route-producer* is

```
(lambda↑ (•)
  (list •
    (let/cc k (route-producer k))
    (let/cc k (route-producer k))))
```

Since we are invoking *route-producer* for the first time, in its body we eventually invoke *send* on 'providence. What's the computation that we capture and store in *resume*? Let's compute this step-by-step. The top-level computation is

```
(list (let/cc k (route-producer k))
    (let/cc k (route-producer k))
    (let/cc k (route-producer k)))
```

Substituting the body of *route-producer* in place of the first invocation, and evaluating the conditional, we get

```
(list (begin
        (send 'providence)
        (send 'houston)
        (send 'bangalore))
    (let/cc k (route-producer k))
    (let/cc k (route-producer k)))
```

We're playing fast-and-loose with scope—technically, we should write the entire **local** that binds *send* between the first **let/cc** and **begin**, as well as all the values bound in the closure—but let's be a bit sloppy for the sake of readability. Just remember what *real-send* is bound to: it'll be relevant in a little while!

When *send* now captures its continuation, what it really captures is

```
(lambda↑ (•)
  (list (begin
          •
          (send 'houston)
          (send 'bangalore))
    (let/cc k (route-producer k))
    (let/cc k (route-producer k))))
```

send stores this continuation in the box, then supplies 'providence to the continuation passed as *real-send*. This reduces the entire computation to

```
(list 'providence
```

(**let/cc** *k* (*route-producer k*))
(**let/cc** *k* (*route-producer k*)))

The second continuation therefore becomes

```
(lambda↑ (•)
  (list 'providence
        •
        (let/cc k (route-producer k))))
```

which becomes the new value of *real-send*. The second time into *route-producer*, however, we do have a value in the box bound to *resume*, so we have to extract and invoke it. We do, resulting in

```
((lambda↑ (•)
   (list (begin
           •
           (send 'houston)
           (send 'bangalore))
         (let/cc k (route-producer k))
         (let/cc k (route-producer k))))
 'dummy)
```

which makes

```
(list (begin
        'dummy
        (send 'houston)
        (send 'bangalore))
      (let/cc k (route-producer k))
      (let/cc k (route-producer k))))
```

the entire computation (because we invoked a **lambda**↑). This looks like it should work fine. When we invoke *send* on 'houston, we capture the second computation, and march down the line.

Unfortunately, a very subtle bug is lurking here. The problem is that the *send* captured in the continuation is closed over the *old* value of *real-send*, because the *real-send* that the continuation closes over is from the previous invocation of *route-producer*. Executing the **begin** inside *send* first stores the new continuation in the box bound to *resume*:

```
(lambda↑ (•)
  (list (begin
          •
          (send 'bangalore))
        (let/cc k (route-producer k))
        (let/cc k (route-producer k))))
```

but then executes the old *real-send*:

```
((lambda↑ (•)
```

```
  (list •
      (let/cc k (route-producer k))
      (let/cc k (route-producer k))))
 'houston)
```

In other words, the entire computation now becomes

```
(list 'houston
    (let/cc k (route-producer k))
    (let/cc k (route-producer k)))
```

which is basically the same thing we had before! (We've got 'houston instead of 'providence, but the important part is what follows it.) As you can see, we're now stuck in a vicious cycle. Now we see why it's the *second* sub-expression in the list where the user break occurs—it's the one that keeps evaluating over and over (the first is over too quickly to notice, while the computation never gets to the third).

This analysis gives us a good idea of what's going wrong. Even though we're passing in a fresh, correct value for *real-send*, the closure still holds the old and, by now, wrong value. We need the new value to somehow replace the value in the old closure.

This sounds like the task of an assignment statement, to mutate the value bound to *real-send*. So we'll first box and refer to the value:

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send-to (box real-send)]
              [define send (lambda (value-to-send)
                             (let/cc k
                               (begin
                                 (set-box! resume k)
                                 ((unbox send-to) value-to-send))))])
        (if (unbox resume)
            ((unbox resume) 'dummy)
            (begin
              (send 'providence)
              (send 'houston)
              (send 'bangalore)))))))
```

Now we have a box whose content we can replace. We should replace it with the new value for *real-send*. Where is that new value available? It's available at the time of invoking the continuation—that is, we could pass that new value along instead of passing 'dummy. Where does this value go? Using our "replacing text" semantics, it replaces the entire (**let/cc** ···) expression in the definition of *send*. Therefore, that expression evaluates to the new value to be put into the box. All we need to do is actually update the box:

```
(define route-producer
  (local ([define resume (box false)])
    (lambda (real-send)
```

```
(local ([define send-to (box real-send)]
        [define send (lambda (value-to-send)
                       (set-box! send-to
                                 (let/cc k
                                   (begin
                                     (set-box! resume k)
                                     ((unbox send-to) value-to-send)))))])
  (if (unbox resume)
      ((unbox resume) real-send)
      (begin
        (send 'providence)
        (send 'houston)
        (send 'bangalore)))))))
```

This time, even though *send* invokes the old closure, the box in that closure's environment now has the continuation for the new resumption point. Therefore,

```
> (list (get route-producer)
        (get route-producer)
        (get route-producer))
(providence houston bangalore)
```

There's just one problem left with this code: it deeply intertwines two very separate concerns, one of which is sending out the actual values (which is specific to whatever domain we are computing in) and the other of which is erecting and managing the continuation scaffold. It would be nice to separate these two.

This is actually a lot easier than it looks. Simply factor out the body into a separate procedure:

```
(define (route-producer-body send)
  (begin
    (send 'providence)
    (send 'houston)
    (send 'bangalore)))
```

(This, you recall, is what we set out to write in the first place!) Everything else remains in a general procedure that simply consumes (what used to be) "the body":

```
(define (general-producer body)
  (local ([define resume (box false)])
    (lambda (real-send)
      (local ([define send-to (box real-send)]
              [define send (lambda (value-to-send)
                             (set-box! send-to
                                       (let/cc k
                                         (begin
                                           (set-box! resume k)
                                           ((unbox send-to) value-to-send)))))])
```

```
        (if (unbox resume)
            ((unbox resume) real-send)
            (body send))))))
```

Introducing this separation makes it easy to write a host of other value generators. They can even generate a potentially infinite number of values! For instance, here's one that generates all the odd positive intgers:

```
(define (odds-producer-body send)
  (local ([define (loop n)
            (begin
              (send n)
              (loop (+ n 2)))])
    (loop 1)))
```

We can make an actual generator out of this as follows:

```
(define odds-producer (general-producer odds-producer-body))
```

and then invoke it in a number of different ways:

```
> (get odds-producer)
1
> (get odds-producer)
3
> (get odds-producer)
5
> (get odds-producer)
7
> (get odds-producer)
9
```

or (assuming an initialized program)

```
> (+ (get odds-producer)
     (get odds-producer)
     (get odds-producer)
     (get odds-producer)
     (get odds-producer))
25
```

Exercise 19.6.1 *Are you sure we're done? If we try to extract more items than a producer has (for instance, invoking (get route-producer) a fourth time), what happens?*

19.7 Why Continuations Matter

Continuations are valuable because they enable programmers to create new control operators. That is, if a language did not already have (say) a producer-consumer construct, continuations make it possible for the programmer to build them manually and concisely. More importantly, these additions can be *encapsulated* in a library, so the rest of the program does not need to be aware of them and can use them as if they were built into the language. If the language did not offer continuations, defining some of these libraries would require a whole-program conversion into CPS. This would not only place an onerous burden on the users of these operations, in some cases this may not even be possible (for instance, when some of the source is not available). This is, of course, the same argument we made for not wanting to rely on CPS for the Web, but the other examples illustrate that this argument applies in several other contexts as well.

Chapter 20

Implementing Continuations

Now that we've seen how continuations work, let's study how to implement them in an interpreter.

The first thing we'll do is change our representation of closures. Instead of using a structure to hold the pieces, we'll use Scheme procedures instead. This will make the rest of this implementation a lot easier.[1]

First, we modify the datatype of values. Only two rules in the interpreter need to change: that which creates procedure values and that which consumes them. These are the fun and app cases, respectively. Both rules are very straightforward: one creates a procedure and wraps it in an instance of closureV, while the other extracts the procedure in the closureV structure and applies it to the argument. The code is in Figure 11.2. Note that this change is so easy only because functions in the interpreted language closely match those of Scheme: both are eager, and both obey static scope. (As we've discussed before, this is our the usual benefit of using meta interpreters that match the interpreted language.)

Recall that *any* program can be converted to CPS—therefore, so can the interpreter. Performing this conversion gives us access to the continuation at every stage, which we can then make available to the programmer through a new language construct. We will therefore approach the implementation of continuations in two steps: first making them explicit in the interpreter, then providing access to them in an extended language.

20.1 Representing Continuations

We'll assume that the interpreter takes an extra argument k, a receiver. The receiver expects the answer from each expression's interpretation. Thus, if the interpreter already has a value handy, it supplies that value to the receiver, otherwise it passes a (possibly augmented) receiver along to eventually receive the value of that expression. The cardinal rule is this: *We never want to use an invocation of interp as a sub-expression of some bigger expression*. Instead, we want *interp* to communicate its answer by passing it to the given k. (Recall the Web situation: every time you invoke a procedure that interacts, the program terminates. Suppose *interp* were executing on the Web. Then any non-trivial context that invokes *interp* will disappear when the interpreter halts. Therefore, all pending computation must be bundled into the receiver.)

[1] Yes, we're switching to a more meta interpreter, but this is acceptable for two reasons: (1) by now, we understand procedures well, and (2) the purpose of this lecture is to implement continuations, and so long as we accomplish this without using Scheme continuations, we won't have cheated.

Let's consider the simplest case, namely numbers. A number needs no further evaluation: it is already a value. Therefore, we can feed the number to the awaiting receiver.

```
[num (n) (k (numV n))]
```

Identifiers and closures, already being values, look equally easy:

```
[id (v) (k (lookup v env))]
[fun (param body)
     (k (closureV (lambda (arg-val)
                    (interp body (aSub param arg-val env)))))]
```

Now let's tackle addition. The rule traditionally looks like this:

```
[add (l r) (numV+ (interp l env) (interp r env))]
```

The naive solution might be to transform it as follows:

```
[add (l r) (k (numV+ (interp l env) (interp r env)))]
```

but do we have a value immediately handy to pass off to k? We will after interpreting both sub-expressions and adding their result, but we don't just yet. Recall that we can't invoke *interp* in the midst of some larger computation. Therefore, we need to bundle that remaining computation into a receiver. What is that remaining computation?

We can calculate the remaining computation as follows. In the naive version, what's the first thing the interpreter needs to do? It must evaluate the left sub-expression.[2] So we compute that first, and move all the remaining computation into the receiver of that invocation of the interpreter:

```
[add (l r) (interp l env
                  (lambda (lv)
                    (k (num+ lv (interp r env)))))]
```

In other words, in the new receiver, we record the computation waiting to complete after reducing the left sub-expression to a value. However, this receiver is not quite right either. It has two problems: the invocation of *interp* on r has the wrong arity (it supplies only two arguments, while the interpreter now consumes three), and we still have an invocation of the interpreter in a sub-expression position. We can eliminate both problems by performing the same transformation again:

```
[add (l r) (interp l env
                  (lambda (lv)
                    (interp r env
                            (lambda (rv)
                              (k (num+ lv rv)))))))]
```

That is, the first thing to do in the receiver of the value of the left sub-expression is to interpret the right sub-expression; the first thing to do with its value is to add them, and so on.

Can we stop transforming now? It is true that *interp* is no longer in a sub-expression—it's always the first thing that happens in a receiver. What about the invocation of *numV+*, though? Do we have to transform it the same way?

[2] Notice that once again, we've been forced to choose an order of evaluation, just as we had to do to implement state.

It depends. When we perform this transformation, we have to decide which procedures are *primitive* and which ones are not. The interpreter clearly isn't. Usually, we treat simple, built-in procedures such as arithmetic operators as primitive, so that's what we'll do here (since *num+* is just a wrapper around addition).[3] Had we chosen to transform its invocation also, we'd have to add another argument to it, and so on. As an exercise, you should consider performing this transformation.

Now let's tackle the conditional. Clearly the interpretation of the test expression takes place in a sub-expression position, so we'll need to lift it out. An initial transformation would yield this:

```
[if0 (test truth falsity)
    (interp test env
          (lambda (tv)
            (if (num-zero? tv)
               (interp truth env ???)
               (interp falsity env ???))))]
```

Do we need to transform the subsequent invocations of *interp*? No we don't! Once we perform the test, we interpret one branch or the other, but no code in this rule is awaiting the result of interpretation to perform any further computation—the result of the rule is the same as the result of interpreting the chosen branch.

Okay, so what receivers do they use? The computation they should invoke is the same computation that was awaiting the result of evaluating the conditional. The receiver *k* represents exactly this computation. Therefore, we can replace both sets of ellipses with *k*:

```
[if0 (test truth falsity)
    (interp test env
          (lambda (tv)
            (if (num-zero? tv)
               (interp truth env k)
               (interp falsity env k))))]
```

That leaves only the rule for application. The first few lines of the transformed version will look familiar, since we applied the same transformation in the case of addition:

```
[app (fun-expr arg-expr)
     (interp fun-expr env
           (lambda (fun-val)
             (interp arg-expr env
                   (lambda (arg-val)
                     ((closureV-p fun-val) arg-val)))))]
```

All we have to determine is what to write in place of the box.

Is the code in the box still valid? Well, the reason we write interpreters is so that we can experiment! How about we just try it on a few expressions and see what happens?

[3] Using our Web analogy, the question is which primitives might arguably invoke a Web interaction. Ones that use the Web must be transformed and be given receivers to stash on the server, while ones that don't can remain unmolested. Arithmetic, clearly, computes entirely locally.

```
>  (interp-test '5 5)
#t
> (interp-test '{+ 5 5} 10)
#t
> (interp-test '{with {x {+ 5 5}} {+ x x}} 20)
procedure interp: expects 3 arguments, given 2 ...
```

Oops! DrScheme highlights the interpretation of the body in the rule for fun.

Well, of course! The interpreter expects three arguments, and we're supplying it only two. What should the third argument be? It needs to be a receiver, but which one? In fact, it has to be whatever receiver is active at the time of the procedure invocation. This is eerily reminiscent of the store: while the environment stays static, we have to pass this extra value that reflects the current state of the dynamic computation. That is, we really want the rule for functions to read

```
[fun (param body)
     (k (closureV (lambda (arg-val dyn-k)
                    (interp body (aSub param arg-val env) dyn-k))))]
```

(What happens if we use *k* instead of *dyn-k* in the invocation of the interpreter? Try it and find out!) Correspondingly, application becomes

```
[app (fun-expr arg-expr)
     (interp fun-expr env
             (lambda (fun-val)
               (interp arg-expr env
                       (lambda (arg-val)
                         ((closureV-p fun-val)
                          arg-val k)))))]
```

The core of the interpreter is in Figure 20.1.

What, incidentally, is the type of the interpreter? Obviously it now has one extra argument. More interestingly, what is its return type? It used to return values, but now…it doesn't return! That's right: whenever the interpreter has a value, it passes the value off to a receiver.

20.2 Adding Continuations to the Language

At this point, we have most of the machinery we need to add continuations explicitly as values in the language. The receivers we have been implementing are quite similar to the actual continuations we need. They appear to differ in two ways:

1. They capture what's left to be done in the interpreter, not in the user's program.

2. They are regular Scheme procedures, not **lambda**↑ procedures.

However,

1. *They capture what's left to be done in the interpreter, not in the user's program.* Because the interpreter closes over the expressions of the user's program, invocations of the interpreter simulate the execution of the user's program. Therefore, the receivers effectively do capture the continuations of the user's program.

2. *They are regular Scheme procedures, not* **lambda**↑ *procedures.* We have taken care of this through the judicious use of a programming pattern. Recall our discussion of the type of the revised interpreter? The interpreter never returns—thereby making sure that no computation awaits the value of the receiver, which is *effectively* the same as computation terminating when the receiver is done.

In other words, we've very carefully set up the interpreter to truly represent the continuations, making it easy to add continuations to the language.

Adding Continuations to Languages

Different languages take different approaches to adding continuations. Scheme's is the most spartan. It add just one primitive procedure, *call/cc*. The resulting continuations can be treated as if they were procedures, so that procedure application does double duty. DrScheme slightly enriches the language by also providing **let/cc**, which is a binding construct, but it continues to overload procedure application.
The language SML uses **callcc** (which is not a binding construct) to capture continuations, and adds a **throw** construct to invoke continuations. Consequently, in SML, procedure application invokes only procedures, and **throw** invokes only continuations, making it possible for a type-checker to distinguish between the two cases.
It's possible that a language could have both a binding construct like **let/cc** and a separate **throw**-like construct for continuation invocation, but there don't appear to be any.

To implement continuations, we will take the DrScheme approach of adding a binding construct but overloading procedural application:

```
<KCFAE> ::= ...
          | {<KCFAE> <KCFAE>}
          | {bindcc <id> <KCFAE>}
```

(This grammar is just for purposes of illustration. It would be easy to add a `throw` construct instead of overloading application.)

To implement continuations, we need to add one new rule to the interpreter, and update the existing rule for application. We'll also add a new kind of type, called contV.

How does bindcc evaluate? Clearly, we must interpret the body in an extended environment:

```
[bindcc (cont-var body)
        (interp body
                (aSub cont-var
                      (contV ???)
                      env)
                k)]
```

The receiver used for the body is the same as the receiver of the `bindcc` expression. This captures the intended behavior when the continuation is not used: namely, that evaluation proceeds as if the continuation were never captured.

What kind of value should represent a continuation? Clearly it needs to be a Scheme procedure, so we can apply it later. Functions are represented by procedures of two values: the parameter and the receiver of the application. Clearly a continuation must also take the value of the parameter. However, the whole point of having continuations in the language is to ignore the receiver at the point of invocation and instead employ the one stored in the continuation value. Therefore, we can just ignore the receiver at the point of application. The invoked continuation instead uses, as its own receiver, that captured at the point of its creation:

```
[bindcc (cont-var body)
        (interp body
                (aSub cont-var
                      (contV (lambda (val)
                                     (k val)))
                      env)
                k)]
```

(Note again the reliance on Scheme's static scope to close over the value of k.) This makes the modification to the application clause very easy:

```
[app (fun-expr arg-expr)
     (interp fun-expr env
             (lambda (fun-val)
               (interp arg-expr env
                       (lambda (arg-val)
                         (type-case KCFAE-Value fun-val
                           [closureV (c) (c arg-val k)]
                           [contV (c) (c arg-val)]
                           [else (error "not an applicable value")])))))]
```

Notice the very clear contrast between function and continuation application: function application employs the receiver at the point of *application*, whereas continuation application employs the receiver at the point of *creation*. This difference is dramatically highlighted by this code.

One last matter: what is the initial value of k? If we want to be utterly pedantic, it should be all the computation we want to perform with the result of interpretation—i.e., a representation of "the rest of DrScheme". In practice, it's perfectly okay to use the identity function. Then, when the interpreter finishes its task, it invokes the identity function, which returns the value to DrScheme. For the purpose of testing, it's even better to use a procedure that prints a value and then halts the program entirely (as we discussed in Section 16.3)—that lets us test whether we converted the interpreter into CPS properly.

And that's it! In these few lines, we have captured the essence of continuations. (The heart of the interpreter is in Figure 20.2.) Note in particular two properties of continuations that are reflected by, but perhaps not obvious from, this implementation:

- To reiterate: we ignore the continuation at the point of application, and instead use the continuation from the point of creation. This is the semantic representation of the intuition we gave earlier for understanding continuation programs: "replace the entire **let/cc** expression with the value supplied to the continuation". Note, however, that the captured continuation is itself a dynamic object—it depends on the entire history of calls—and thus cannot be computed purely from the program source without evaluation. In this sense, it is different from the environment in a closure, which can partially be determined entirely statically (that is, we can determine which identifiers are in the environment, though it is undecidable what their values will be.)
- The continuation closes over the environment; in particular, its body is scoped statically, not dynamically.

Exercise 20.2.1 *Add* **web-read** *to* KCFAE.

20.3 On Stacks

Let's re-examine the procedure application rule in the interpreter:

```
[app (fun-expr arg-expr)
     (interp fun-expr env
             (lambda (fun-val)
               (interp arg-expr env
                       (lambda (arg-val)
                         ((closureV-p fun-val)
                          arg-val k)))))]
```

Observe how the receiver of the value of the function position, and that of the value of the argument position, are both new procedures, whereas when the procedure application finally happens (in the latter receiver), the third argument is just *k*.

Given that the receivers represent the stack at every point, this should seem rather strange: wasn't the stack meant to hold a record of procedure invocations? And if so, why is the receiver "growing" when evaluating the function and the argument, but "shrinking" back to its original size when the application actually happens?

We have said that the receiver corresponds directly to the stack. In particular, the receiver is a procedure that may refer to another procedure (that it closes over), which may refer to another procedure (that *it* closes over), and so on. Each of these procedures represents one stack frame (sometimes called an *activation record*, because it records an extant activation of a procedure in the running program). Returning a value "pops" the stack; since we have made the stack explicit, the equivalent operation is to pass the value to be returned to the receiver.

We therefore see, from this pattern, that the stack is used solely to store intermediate results. It plays no part in the actual invocation of a function. This probably sets on its head everything you have been taught about stacks until now. This is an important and, perhaps, startling point:

Stacks are not necessary for invoking functions.

The stack only plays a role in *evaluating the argument to the function* (and, in a language with first-class functions, evaluating the function position itself); once that argument has been reduced to a value (in an eager language), the stack has no further role with respect to invoking a function. The actual invocation is merely a jump to an address holding the code of the function: it's a goto.

Converting a program to CPS thus accomplishes two things. First, it exposes something—the stack—normally hidden during the evaluation process; this is an instance of *reflection*. The transformation also makes this a value that a programmer can manipulate directly (even changing the meaning of the program as a result); this is known as *reification*.

Reflection and Reification

Reflection and reification are very powerful programming concepts. Most programmers encounter them only very informally or in a fairly weak setting. For instance, Java offers a very limited form of reflection (a programmer can, for instance, query the names of methods of an object), and some languages reify very low-level implementation details (such as memory addresses in C). Few languages reify truly powerful computational features; the ones that do enable entirely new programming patterns that programmers accustomed only to more traditional languages usually can't imagine. Truly smart programmers sometimes create their own languages with the express purpose of reifying some useful hidden element, and implement their solution in the new language, to create a very powerful kind of reusable abstraction. A classic instance of this is Web programmers who have reified stacks to enable a more direct programming style.

20.4 Tail Calls

Converting the program to CPS helps us clearly see which calls are just gotos, and which ones need stack build-up. The ones that are just gotos are those invocations that use the same receiver argument as the one they received. Those that build a more complex receiver are relying on the stack.

Procedure calls that do not place any burden on the stack are called *tail calls*. Converting a program to CPS helps us identify tail calls, though it's possible to identify them from the program source itself. An invocation of g in a procedure f is a tail call if, in the control path that leads to the invocation of g, the value of f is determined by the invocation of g. In that case, g can send its value directly to whoever is expecting f's value; this verbal description is captured precisely in the CPSed version (since f passes along its receiver to g, which sends its value to that receiver). This insight is employed by compilers to perform *tail call optimization* (abbreviated TCO, and sometimes referred to as *last call optimization*), whereby they ensure that tail calls incur no stack growth.

Here are some consequences of TCO:

- With TCO, it no longer becomes necessary for a language to provide looping constructs. Whatever was previously written using a custom-purpose loop can now be written as a recursive procedure. So long as all recursive calls are tail calls, the compiler will convert the calls into gotos, accomplishing the same efficiency as the loop version. For instance, here's a very simple version of a `for` loop, written using tail calls:

```
(define (for init condition change body result)
  (if (condition init)
      (for (change init)
           condition
           change
           body
           (body init result))
      result))
```

By factoring out the invariant arguments, we can write this more readably as

```
(define (for init condition change body result)
  (local [(define (loop init result)
            (if (condition init)
                (loop (change init)
                      (body init result))
                result))]
    (loop init result)))
```

To use this as a loop, write

```
(for 10 positive? sub1 + 0)
```

which evaluates to 55. It's possible to make this look more like a traditional `for` loop using macros, which we will discuss in Section 36. That aside, notice how similar this is to a *fold* operator! Indeed, *foldl* employs a tail call in its recursion, meaning it is just as efficient as looping constructs in more traditional languages.

- Put differently, thanks to TCO, the set of looping constructs is extensible, not limited by the imagination of the language designer. In particular, with care it becomes easy to create loops (or *iterators*) over new data structures without suffering an undue performance penalty.

- While TCO is traditionally associated with languages such as Scheme and ML, there's no reason they must be. It's perfectly possible to have TCO in any language. Indeed, as our analysis above has demonstrated, TCO is the *natural consequence of understanding the true meaning of function calls*. A languages that deprives you of TCO is cheating you of what is rightfully yours—stand up for your rights! Because so many language designers and implementors habitually mistreat their users by failing to support TCO, however, programmers have become conditioned to think of all function calls as inherently expensive, even when they are not.

- A special case of a tail call is known as *tail recursion*, which occurs when the tail call within a procedure is to the same procedure. This is the behavior we see in the procedure *for* above. Keep in mind, however, that tail recursion optimization is only a special case. While it is an *important* special case (since it enables the implementation of linear loops), it is neither the most interesting case nor, more importantly, the only useful one.

Sometimes, programmers will find it natural to split a computation across two procedures, and use tail calls to communicate between them.[4] This leads to very natural program structures. A programmer using a language like Java is, however, forced into an unpleasant decision. If they split code across methods, they pay the penalty of method invocations that use the stack needlessly. But even if they combine the code into a single procedure, it's not clear that they can easily turn the two code bodies into a single loop. Even if they do, the structure of the code has now been altered irrevocably. Consider the following example:

```
(define (even? n)
  (if (zero? n)
      true
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      false
      (even? (sub1 n))))
```

Try writing this entirely through loops and compare the resulting structure.

Therefore, even if a language gives you tail recursion optimization, remember that you are getting less than you deserve. Indeed, it sometimes suggests an implementor who realized that the true nature of function calls permitted calls that consumed no new stack space but, due to ignorance or a failure of imagination, restricted this power to tail recursion only. The primitive you really want a language to support is tail *call* optimization. With it, you can express solutions more naturally, and can also build very interesting abstractions of control flow patterns.

- Note that CPS converts every program into a form where every call is a tail call!

Exercise 20.4.1 *If, in* CPS*, every call is a tail call, and the underlying language supports* TCO *(as Scheme does), does the* CPS *version of a program run in constant stack space even if the original does not? Discuss.*

Exercise 20.4.2 *Java does not support* TCO*. Investigate why not.*

20.5 Testing

You might think, from last time's extended example of continuation use, that it's absolutely necessary to have state to write any interesting continuation programs. While it's true that most *practical* uses of the full power of continuations (as opposed to merely exceptions, say) do use state, it's possible to write some fairly complicated continuation programs without state for the purposes of testing our interpreter. Here are some

[4] They may not even communicate mutually. In the second version of the loop above, *for* invokes *loop* to initiate the loop. That call is a tail call, and well it should be, otherwise the entire loop will have consumed stack space. Because Scheme has tail calls, notice how effortlessly we were able to create this abstraction. If the language supprted only tail *recursion* optimization, the latter version of the loop, which is more pleasant to read and maintain, would actually consume stack space against our will.

such programs. You should, of course, first determine their value by hand (by writing the continuations in the form of **lambda**↑ procedures, substituting, and evaluating).

First, a few old favorites, just to make sure the easy cases work correctly:

1. `{bindcc k 3}`
2. `{bindcc k {k 3}}`
3. `{bindcc k {+ 1 {k 3}}}`
4. `{+ 1 {bindcc k {+ 1 {k 3}}}}`

And now for some classic examples from the continuations lore:

1. ```
   {{bindcc k
              {k {fun {dummy}
                      3}}}
    1729}
   ```
2. ```
   {bindcc k
            {k
             {k
              {k 3}}}}
   ```
3. ```
   {{{bindcc k k}
     {fun {x} x}}
    3}
   ```
4. ```
   {{{{bindcc k k}
       {fun {x} x}}
      {fun {x} x}}
     3}
   ```

The answer in each case is fairly obvious, but you would be cheating yourself if you didn't hand-evaluate each of these first. This is painful, but there's no royal road to understanding! (If in doubt, of course, run their counterpart in Scheme.)

```
(define-type CFAE-Value
  [numV (n number?)]
  [closureV (p procedure?)])

;; interp : CFAE Env receiver → doesn't return
(define (interp expr env k)
  (type-case CFAE expr
    [num (n) (k (numV n))]
    [add (l r) (interp l env
                       (lambda (lv)
                         (interp r env
                                 (lambda (rv)
                                   (k (num+ lv rv))))))]
    [if0 (test truth falsity)
         (interp test env
                 (lambda (tv)
                   (if (num-zero? tv)
                       (interp truth env k)
                       (interp falsity env k))))]
    [id (v) (k (lookup v env))]
    [fun (param body)
         (k (closureV (lambda (arg-val dyn-k)
                        (interp body (aSub param arg-val env) dyn-k))))]
    [app (fun-expr arg-expr)
         (interp fun-expr env
                 (lambda (fun-val)
                   (interp arg-expr env
                           (lambda (arg-val)
                             ((closureV-p fun-val)
                              arg-val k)))))]))
```

Figure 20.1: Making Continuations Explicit

```
(define-type KCFAE-Value
  [numV (n number?)]
  [closureV (p procedure?)]
  [contV (c procedure?)])

;; interp : KCFAE Env receiver → doesn't return
(define (interp expr env k)
  (type-case KCFAE expr
    [num (n) (k (numV n))]
    [add (l r) (interp l env
                       (lambda (lv)
                         (interp r env
                                 (lambda (rv)
                                   (k (num+ lv rv))))))]
    [if0 (test truth falsity)
         (interp test env
                 (lambda (tv)
                   (if (num-zero? tv)
                       (interp truth env k)
                       (interp falsity env k))))]
    [id (v) (k (lookup v env))]
    [fun (param body)
         (k (closureV (lambda (arg-val dyn-k)
                        (interp body (aSub param arg-val env) dyn-k))))]
    [app (fun-expr arg-expr)
         (interp fun-expr env
                 (lambda (fun-val)
                   (interp arg-expr env
                           (lambda (arg-val)
                             (type-case KCFAE-Value fun-val
                               [closureV (c) (c arg-val k)]
                               [contV (c) (c arg-val)]
                               [else (error "not an applicable value")])))))]
    [bindcc (cont-var body)
            (interp body
                    (aSub cont-var
                          (contV (lambda (val)
                                   (k val)))
                          env)
                    k)]))
```

Figure 20.2: Adding Continuations as Language Constructs

Part VIII

Memory Management

Chapter 21

Automatic Memory Management

21.1 Motivation

We have seen in Section 13 that, when making data structures explicit in the store, the store (or *heap*) has allocated data that are no longer necessary. Ideally, we would like this space reclaimed automatically. This would enable us to program *as if we had an infinite amount of memory* yet, so long as we never exceed the actual virtual memory capacity at any instant, the program would never halt with an out-of-memory error.

Whose responsibility is it to reclaim these locations? It can't be the responsibility of the allocating procedure. For instance, suppose we had written a procedure *filter-pos* that filters out the positive numbers in a given list. This procedure cannot know know whether or not its caller needs the argument list again. That procedure may pass the same list to some other procedure, and so on.

Even if it the chain of responsibility is clear, memory reclamation is often frustrating because it interferes with the flow of control in the program, mixing high-level algorithmic description with low-level resource management. Let's say we knew for sure that the input list would not be used any longer. The procedure *filter-pos* could then attempt to reclaim the list's elements as follows:

```
(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
     (begin
       (reclaim-memory! (first l))
       (if (> (first l) 0)
           (cons (first l) (filter-pos (rest l)))
           (filter-pos (rest l))))]))
```

There is a subtle bug in this program, but let's focus on a simpler problem with it: while this reclaims each first element, it doesn't reclaim the *cons*es that constitute the input list. We might therefore try

```
(define (filter-pos l)
  (cond
    [(empty? l) empty]
```

```
    [else
     (begin
       (reclaim-memory! (first l))
       (reclaim-memory! (rest l))
       (if (> (first l) 0)
           (cons (first l) (filter-pos (rest l)))
           (filter-pos (rest l))))]))
```

Unfortunately, this version duplicates the bug! Once we reclaim the *first* and *rest* of the list, we can no longer refer to those elements. In particular, in a concurrent system (and most software today *is* concurrent), the moment we reclaim the memory, another process might write into it, so if we access the memory we might get nonsensical output. And even otherwise, in general, if we reclaim and then perform a procedure call (in this case, a recursive one), when we return (as we do in the first branch, to perform the *cons*) that heap location may have since been overridden with other values. So this is a problem even in the absence of concurrency. We must therefore instead write

```
(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
     (local ([define result
               (if (> (first l) 0)
                   (cons (first l) (filter-pos (rest l)))
                   (filter-pos (rest l)))])
       (begin
         (reclaim-memory! (first l))
         (reclaim-memory! (rest l))
         result))]))
```

While this version is no longer succeptible to the problems we discussed earlier, it has introduced a significant new problem. Whereas earlier *filter-pos* was tail-recursive in cases when the list element not positive, now *filter-pos* is *never* tail recursive. In fact, the problem we see here is a common problem with loop-like programs: we must hold on to the value being passed in the recursive call so we can reclaim it after the call completes, which forces us to destroy any potential for tail-call optimizations.

In short, even when we know who is responsible for reclaiming data, we face several problems:

- The program structure may be altered significantly.
- Concurrency, or even just other function calls, can expose very subtle reclamation errors.
- Loops often lose tail-calling behavior.
- It becomes much harder to define simple abstractions. For example, we would need two versions of a *filter-pos* procedure, one that does and one that doesn't reclaim its argument list. In turn, every procedure that wants to invoke *filter-pos* must choose which version to invoke. And so on up the

abstraction hierarchy. (Can you see how the number of possible options can grow exponentially in the number of arguments?)

These problems, furthermore, assume we can even know which procedure is responsible for managing every datum, which is a very strong assumption. Sometimes, two procedures may share a common resource (think of the pasteboard in a typical windowing system, which is shared between multiple applications), which means it's no single unit's responsibility in particular.

At any rate, reasoning about these chains of ownership is hard, and making the wrong decisions leads to numerous insidious errors. Therefore, it would be better if we could make this the responsibility of the run-time system: that is, whatever is responsible for allocating memory should also be responsible for reclaiming memory when it is no longer necessary. That is, we usually prefer to program with *automated memory management*, colloquially referred to by the much more colorful term, *garbage collection*.

21.2 Truth and Provability

In the previous paragraph, we have given the garbage collector the responsibility of reclaiming memory "when it is no longer necessary". This puts a very significant pressure on the garbage collector: the collector must know whether or not a programmer is going to use a datum again or not. In other words, garbage collection becomes an artificial intelligence problem.

This highlights a common tension that arises in computer science, and especially in programming language design: that between *truth* and *provability*. This might sound like a very profound philosophical issue—and it is—but you are already very familiar with it from math courses, where a professor asked you to prove something she knew to be true, but you were unable to construct an actual line of reasoning for it! We see this tension in several other places, too: for example, the type checker may not know whether or not a given operation will succeed, while the programmer has a complex line of reasoning that justifies it; and the optimizer in a compiler might not be able to prove that an expensive expression is equivalent to a less expensive one (you might notice that this goes back to our discussion about referential transparency).

Anyway, the garbage collector obviously cannot know a programmer's intent, so it needs to *approximate* her intent as best as possible. Furthermore, this approximation must meet a few tightly intertwined properties. To understand these, let us consider a few extreme implementations of collectors.

The first collector reclaims absolutely no garbage. Obviously it runs very quickly, and it never accidentally reclaims something that it should not reclaim. However, this is obviously useless. This suggests that a collector must demonstrate

utility The collector's approximation must identify enough garbage to actually help computation continue.

Another collector avoids this problem by reclaiming *all* data in memory, irrespective of whether or not they are necessary for future computation. This, too, is obviously not very useful, because the computation would soon crash. Therefore, a collector must exhibit

soundness The collector must never reclaim a datum that is used by a subsequent computation.

A third collector, wanting to avoid both of the above perils, halts at every datum and computes a very complex simulation of the program's execution to determine whether or not the program will access this

datum again. It has to consider all execution paths, both branches of each conditional, and so on. This, too, would be unacceptable: a collector must manifest

efficiency The collector should run sufficiently quickly so that programmers do not get frustrated (and therefore turn back to manual memory management).

In practice, garbage collectors reconcile these demands thanks to a very useful approximation of truth: *reachability*. That is, a collector begins with a set of memory locations called the *root set*; this typically includes all the heap references on the stack and in the current registers. From the root set, the collector sweeps the heap to determine which objects are reachable: if object *o* is reachable, then all objects that *o* refers to in its fields are also reachable—and so on, recursively. All reachable objects are called *live*, and survive garbage collection; the collector reclaims the storage allocated to all other objects.

With a little reflection, we realize that reachability is an excellent approximation of truth. If an object is reachable, then there is (in effect) some sequence of field dereferences and function or method invocations that can use its value. Since the programmer may have written exactly such a sequence of invocations, the collector should not reclaim the object. If, on the other hand, an object is not reachable in this fashion, *no* sequence of dereferences and invocations can use its value.[1] Therefore, the garbage collector can safely reclaim its space.

Reachability is, of course, not always a strong enough approximation to truth. For instance, consider the following program fragment:

```
(define v (make-vector 1000))
(define k (vector-length v))
;; rest of program
```

Suppose the rest of the program never references *v*.[2] In that case, after *k* has been given its value the space consumed by the vector bound to *v* should be reclaimed; but since *v* is a global variable, it is always reachable, so the collector cannot reclaim it. In general, large data structures bound to global variables are invariably candidates for *space leakage*, which is what we call the phenomenon of a collector not reclaiming space that we know is no longer necessary. (Notice the difference between truth and provability coming into play very strongly.) Tracing space leaks is sometimes subtle, but it is often as simple as looking at values bound to global and static variables and, when those values are no longer necessary, mutating the variable to a value like `null` (in Java) or (*void*) (in Scheme).

Notice, by the way, the asymmetry in our justification for why tracing is a reasonable approximation to truth. Unreachable objects *will* not be used so they can always be reclaimed safely, whereas reachable objects *may* be used again so we must allow them to persist. In fact, a collector can sometimes reason about these "maybe" cases. For instance, consider the following program:

```
(local ([define v (make-vector 1000)]
        [define k (vector-length v)])
  ...)
```

[1] This claim makes a certain important assumption about the underlying programming language that is not always valid: it applies to languages like Java and Scheme but not to C and C++. Do you see it?

[2] This assumes that the rest of the program text is known. Modern languages support features such as *dynamic loading*, which is the ability to extend the program during its execution.

Now suppose the body of this expression doesn't reference *v*. Because *v* is not global, as soon as the value of *k* has been computed, the implementation can safely set the value of *v* to a null or void value, thus making the vector formerly bound to *v* a candidate for reclamation immediately. Many implementations for languages that employ garbage collection do in fact perform such "safe for space" optimizations.

Part IX

Semantics

Chapter 22

Shrinking the Language

How small can a programming language be? We've already seen that Scheme is sufficient for expressing a large number of computations quite concisely. The version of Scheme we've seen so far is nevertheless quite small; here are most of the features we've used:

x y z ⋯ ;; variables
'a 'b 'c ⋯ ;; symbols
0 1 2 ⋯ ;; numbers
+ - ∗ ⋯ ;; arithmetic operators
define-type type-case
cons first rest
cond if
true false *zero?* **and or**
() ;; function application
local
let/cc *call/cc*
define lambda

We have seen in Section 18 that we can express continuations using just procedures and applications, in the form of CPS. Similarly, it is easy to see that we can express various type constructors with lists by being disciplined about their use. (To wit, we would put a tag at the beginning of the list indicating what type we are representing, and check the tag before every access to avoid ill-typed operations.) We have also seen in Section 6.3 that local definitions can be expressed in terms of function definitions and application.

That still leaves several constructs, which we can organize into groups related by purpose:

- variables, procedure definitions, applications
- numbers and arithmetic
- Boolean constants and operations
- lists and other aggregate data structures

In what follows, we will methodically eliminate most of these features until we are left with a minimal set that is surprisingly small. In particular, we will demonstrate that the first group—variables, procedures and application—can encode all the remaining language constructs.

As a motivating example, consider the following definition of the factorial function:

```
(define (fact n)
  (if (zero? n)
      1
      (* n (fact (sub1 n)))))
```

This contains most of the features under discussion.

22.1 Encoding Lists

Let's first consider lists. Though we don't need lists to define factorial, they are a useful indicator of how we would handle compound data structures.

We will consider a data structure even simpler than lists, namely pairs. Once we have pairs, we can construct lists quite easily: each pair's second element is another pair, representing the next "link" in the list. Therefore, we must demonstrate how to implement *pair*, *left* and *right* without using any Scheme data structure primitives (such as lists or new types).

How can we do this? The pair constructor must consume two values and return *something*. What can we return? Since we are not trying to eliminate procedures, perhaps it can return a procedure of some sort. That is, every instance of

(*pair A B*)

in the program becomes

(**lambda** ...)

We will write this as

(*pair A B*) ≡ (**lambda** ...)

where ≡ represents textual substitution. So what should the argument and body of the procedure be? Should it return *A*? Sometimes, yes, if the program wants the first value in the pair; at other times, it should return *B*. In other words, the response needs to be parameterized to depend on the selector. We can express this neatly thus:

(*pair A B*) ≡ (**lambda** (*selector*) (*selector A B*))

This defers the problem to defining the selectors, but there is hope now. The operator that chooses the left is simply

left ≡ (**lambda** (*A B*) *A*)

and analogously,

right ≡ (**lambda** (*A B*) *B*)

Finally, we must rewrite every use of these primitives to be in *object-verb* rather than in *verb-object* form: that is, because the pair is itself a procedure that consumes the selector as an argument, we must write

(*e left*)

in place of

(*first e*)

for every *e* that will evaluate to a pair, and analogously replacing *rest* with *right*.[1]

Exercise 22.1.1 *Our knowledge of computer science tells us that the left and right fields must consume space* somewhere*; yet in this encoding, they appear to consume no space at all. Is that true (and, if so, how is that possible)?*

22.2 Encoding Boolean Constants and Operations

What is a conditional? At its simplest, it makes a choice between two computations based on some criterion. We are used to a construct, such as **if**, as the operator that makes that choice, based on some Boolean value. But what if, instead, the choice were made by the Boolean value itself?

Here's an analogous situation. In a *pure* object-oriented language, every value is an object, including the Boolean values. That is, we can think of values `true` and `false` as subtypes of a Boolean type, each endowed with a `choose` method. The `choose` method takes two objects as arguments, each with a `run` method that represents the rest of the computation. A `true` object invokes the `run` method of its first argument object, while a `false` object invokes `run` in its second argument. Thus, effectively, the conditional is implemented by the process of dynamic dispatch (which chooses which object to run when `choose` is invoked on a Boolean value).

We can apply this insight into encoding conditionals entirely in terms of procedures. Every instance of

(**if** *C T F*)

is rewritten as

(*C T F*)

which we will henceforth write as

(**if** *C T F*) ≡ (*C T F*)

(once again, read ≡ as textual replacement). where we assume that *C* will always evaluate to one of the two Boolean values. We therefore reduce the problem to one of defining Boolean values that correctly implement the choice operation.

Defining the Boolean values is quite easy given our preceding discussion of objects. The value representing truth must consume the two options, ignore (by convention) the second and use only the first:

yes ≡ (**lambda** (*T F*) *T*)

[1] This shift from algebra's verb-object convention is, of course, familiar to object-oriented programmers.

Likewise,

no ≡ (**lambda** (*T F*) *F*)

Exercise 22.2.1 *This encoding of Booleans assume the use of lazy evaluation. Provide an example that illustrates this, and demonstrate how we can remove this dependency.*

Exercise 22.2.2 *Define* **and** *and* **or** *in terms of our limited set of primitives. Do your definitions perform short-circuiting?*

Exercise 22.2.3 *Is it purely coincidental that left and yes have the same definition?*

22.3 Encoding Numbers and Arithmetic

Having dispatched of lists and Booleans, we are now ready to tackle numbers. Let's agree to limit our attention to the natural numbers (an integer no smaller than zero).

What is the essence of a natural number? It is a counting object: it tells us how many instances there are of some discrete entity. While it is conventional to use "Arabic" numerals (0, 1, 2, 3, ...) to represent these numbers, there are many other representations available (for instance, the whole numbers—natural numbers strictly bigger than zero—can be represented using Roman numerals: I, II, III, IV, ...). Even compared with this variety of notations, though, the representation we will define here is truly striking.

Let's think about "one-ness". The number one represents many things. It captures the number objects in a collection of one book, of one maple leaf, of one walrus, of one cricket ball. It also represents the act of applying a function to a value.

Which function? Which value? Any function and any value will do just fine; indeed, to avoid having to decide, we can simply make them parameters. That is, we can represent one as

one ≡ (**lambda** (*f*) (**lambda** (*x*) (*f x*)))

This is the most abstract way we have of saying "the act of applying some function to some argument once". If we supply the concrete arguments *add1* and 0 for *f* and *x*, respectively, we get the expected numeral 1 from Scheme. But we can also supply *square* and 5, respectively, to get the numeral 25, and so on.

If that's what represents one-ness, what represents two-ness and three-ness? Why, the same kind of thing: respectively, two applications and three applications of some function to some argument:

two ≡ (**lambda** (*f*) (**lambda** (*x*) (*f* (*f x*))))
three ≡ (**lambda** (*f*) (**lambda** (*x*) (*f* (*f* (*f x*)))))

and so on. Indeed, supplying *add1* and 0 to each of these numerals yields expected Scheme numeral. We should therefore intuitively think of *f* as an "add one" operation and *x* as the "zero" constant, but in fact we can supply any operation and base constant we want (and, in what follows, we will in fact exploit this abstract representation).

Since we want to represent the natural numbers, we must be able to represent zero, too. The pattern above suggests the following numeral:

zero ≡ (**lambda** (*f*) (**lambda** (*x*) *x*))

and indeed, that is a numeral that represents zero applications of f to x. (As we said, think of x as "zero", so a procedure that returns x unmolested is a good representative for 0.)

Now that we have numerals to represent the natural numbers, it's time to define operations on them. Let's begin with incrementing a number by one. That is, we expect

(*succ one*) ≡ (*succ* (**lambda** (*f*) (**lambda** (*x*) (*f x*))))

to yield the equivalent of

(**lambda** (*f*) (**lambda** (*x*) (*f* (*f x*))))

This looks nasty: it appears we must perform "surgery" on the procedure to "insert" another application of f. This is impossible since the procedures are opaque objects (in a computer, represented by the implementation as some sequence of bits we may not even understand).

It's important to note that what we want is the *equivalent* of the representation of two: that is, we want a numeral that represents two-ness. Here is another term that has the same effect:

```
(lambda (f)
  (lambda (x)
    (f ((one f) x))))
```

That is, it applies *one* to f and x, obtaining the effect of applying f to x once. It then applies f again to the result. This has the same *net effect* as the more concise representation of *two*. By the same line of reasoning, we can see that this pattern always represents the act of incrementing a number:

```
succ ≡
(lambda (n)
  (lambda (f)
    (lambda (x)
      (f ((n f) x)))))
```

Now we can tackle addition. Observe the following arithmetic result:

$$m+n = \overbrace{1+1+\cdots+1}^{n \text{ times}}+m$$

Let's try putting this in words. To add m and n, we add one n times to m. That is, we apply an operation that adds one, n times, to a base value of m. How do we iterate anything n times? That's exactly what the numeral for n represents: the act of performing n applications. The numeral expects two values: the operation to apply, and the base value. The operation we want is the addition of one, which we've just defined; the base value is the other addend. Therefore:

```
sum ≡
(lambda (m)
  (lambda (n)
    ((n succ) m)))
```

A similar insight gives us multiplication:

$$m \times n = \overbrace{m + m + \cdots + m}^{n \text{ times}} + 0$$

From this, and employing a similar line of reasoning, we can define

```
prod ≡
(lambda (m)
  (lambda (n)
    ((n (sum m)) zero)))
```

(We can see from this definition the wisdom of having the binary operators accept one argument at a time.)

It's easy to see that we can define other additive operators inductively. But how about subtraction? This seems to create an entirely new level of difficulty. Addition seemed to need the ability to modify the numeral to apply the first argument one more time, but we found a clever way of applying it "from the outside". Subtraction, on the other hand, requires that a procedure *not* be applied, which seems harder still.

The solution to this problem is to make the following observation. Consider the pair $\langle 0,0 \rangle$. Now apply the following algorithm. Given such a pair of numbers, create a new pair. The new pair's left component is the old pair's right component; the new pair's right component is the old pair's right component incremented by one. Visually,

$$\begin{aligned} \text{initial value} &: \langle 0,0 \rangle \\ \text{after 1 iteration} &: \langle 0,1 \rangle \\ \text{after 2 iterations} &: \langle 1,2 \rangle \\ \text{after 3 iterations} &: \langle 2,3 \rangle \\ \text{after 4 iterations} &: \langle 3,4 \rangle \end{aligned}$$

and so on.

You might find this procedure rather strange, in that it entirely ignores the left component of each preceding pair to create the next one in the sequence. Notice, however, that after n iterations, the left component holds the value $n-1$. Furthermore, observe the operations that we used to obtain these pairs: creation of an initial pair, pair deconstruction, increment by one, and new pair construction. That is, the following procedure represents the algorithm applied at each step:

```
(lambda (p)
  (pair (right p)
        (succ (right p))))
```

The following represents the initial value:

```
(pair zero zero)
```

If we apply this n times, then read out the left value of the resulting pair, we get... $n-1$!

```
pred ≡
(lambda (n)
```

(*left*
((*n* (**lambda** (*p*)
(*pair* (*right p*)
(*succ* (*right p*))))))
(*pair zero zero*)))

Once we have subtraction by one, we can implement regular subtraction and other such operations.

That leaves only one arithmetic-related primitive we need to implement factorial, namely Scheme's *zero?*. What does this operator do? Given a representation for zero, it returns true, otherwise false. What is the one characteristic that distinguishes the numeral for zero from that for all non-zero numbers? The latter all apply their first argument to their second at least once, while the former does not. Therefore, the following defines *zero?*:

iszero ≡
(**lambda** (*n*)
((*n* (**lambda** (*ignore*) *no*)) *yes*))

If the first argument is applied at all, no matter how many times it's applied, it returns the representation of false; if it never is, then the "zero" value, the representation of true, results.

Historical aside: These numerals are known as the Church numerals, in honor of their inventor, Alonzo Church.

Exercise 22.3.1 *Can you extend this encoding to other kinds of numbers: negative integers, rationals, reals, complex numbers, . . . ?*
Hint: *Some of these are easy to encode using pairs of other kinds of numbers.*

Exercise 22.3.2 *Here are two alternate representations of the exponentiation operation. Which one is faster?*

```
(define exp1
  (lambda (m)
    (lambda (n)
      ((n (prod m)) one))))

(define exp2
  (lambda (m)
    (lambda (n)
      (n m))))
```

22.4 Eliminating Recursion

The **define** construct of Scheme is surprisingly powerful. It not only assigns values to names, it also enables the construction of recursive procedures (otherwise the definition of factorial given above would not function). To eliminate **define**, therefore, we must create a way of defining recursive procedures. . . without recursion!

As we do this, we will sometimes encounter expressions that we're not yet sure how to write. We will use the symbol $\underline{\bullet}$ to represent a special value: if the computation ever tries to apply it, it halts immediately. Think of it as a landmine that the computation should never apply.[2]

Let's now study the recursion in factorial. Let's begin with the following skeletal definition:

```
fact ≡
(lambda (n)
  (if (zero? n)
      1
      (* n (• (sub1 n)))))
```

This definition is not entirely useless. Indeed, it correctly computes the factorial on the input 0. On any input greater than 0, however, the computation terminates uselessly.

We can make a more useful definition by including a copy of *fact* as follows:

```
fact ≡
(lambda (n)
  (if (zero? n)
      1
      (* n (  (lambda (n)
                (if (zero? n)
                    1
                    (* n (• (sub1 n)))))
              (sub1 n)))))
```

This definition works perfectly well on inputs of 0 and 1, but not greater. We can repeat this process endlessly—a process, not at all coincidentally, reminiscent of creating the proper environment for recursion in Section 9—but obviously, we will never get the *true* definition of factorial. We'll have to do better.

While we're trying to generate a spark of insight, let's try to clean up the code above. Instead of relying on this unspecified $\underline{\bullet}$ operation, we might as well just parameterize over that location in the program:

```
mk-fact ≡
(lambda (f)
  (lambda (n)
    (if (zero? n)
        1
        (* n (f (sub1 n))))))
```

The resulting procedure isn't quite factorial itself, but rather a factorial-*maker*: given the right value for the parameter, it will yield the proper factorial procedure. That still begs the question, however, of what to supply as a parameter.

Let's go back to our doomed attempt to nest copies of the factorial procedure. This has the advantage that, until the copies run out, there is always *another copy available*. So we have a clearer handle on the problem now: we need to provide as a parameter something that will *create another copy upon demand*.

[2] A good approximation of $\underline{\bullet}$ is the Scheme procedure *exit*.

It would seem that *mk-fact* is just such a creator of copies. So what happens if we feed *mk-fact* as the argument to *mk-fact*—

(*mk-fact mk-fact*)

—to fill the hole where we need a factorial generator? This application results in the procedure

```
(lambda (f)
  (lambda (n)
    (if (zero? n)
        1
        (* n (mk-fact (sub1 n))))))
```

(We've just substituted *mk-fact* for *f* in the body of *mk-fact*.) We can safely apply this procedure to 0 and obtain 1, but if we apply it to any larger input, we get an error: *mk-fact* is expecting a procedure as its argument, but here we're applying it to a number.

Okay, so we cannot apply *mk-fact* to a number. To gain some insight into what we *can* apply it to, let's apply it to •, so the new definition of *mk-fact* is:

```
(lambda (f)
  (lambda (n)
    (if (zero? n)
        1
        (* n ((f •) (sub1 n))))))
```

and apply this to *mk-fact*. Upon substitution, this evaluates to

```
(lambda (n)
  (if (zero? n)
      1
      (* n ((mk-fact •) (sub1 n))))))
```

This procedure clearly works correctly on the argument value 0. For argument 1, evaluation results in:

```
(* 1
  (if (zero? 0)
      1
      (* 0 ((• •) (sub1 n)))))
```

which also works correctly. For initial arguments of 2 and greater, however, evaluation halts on trying to apply •. In short, we have a definition of *mk-fact* that, when applied to itself, works correctly for values of 0 and 1, but no higher.

By itself, this does not seem like any kind of progress at all. In fact, however, we have come most of the way to a solution. Recall that earlier, to define a factorial procedure for values of 0 and 1, we had to *copy* the definition of the procedure, resulting in a definition that was roughly twice as large as the one we wanted. Now we have one that is roughly the same size as the original, i.e., it involves no copying.

It is not surprising that the computation eventually terminated, when we supplied • as the argument. But what else could we have supplied? Observe that when we supplied *mk-fact* as an argument, the term

in the "recursive" position evaluated to (*mk-fact* $\underline{\bullet}$), which continued computation correctly (for one more step); whereas supplying $\underline{\bullet}$ made that term ($\underline{\bullet}$ $\underline{\bullet}$), which terminated it. Therefore, we should ensure that the function position of that application is always *mk-fact*. This means precisely that instead of supplying $\underline{\bullet}$ as the argument, we should supply *mk-fact*! This should not be surprising: just as applying *mk-fact* to itself outside the body of *mk-fact* was useful in initiating the computation, doing the same inside is useful in continuing it.

That is, we must define

```
mk-fact ≡
(lambda (f)
  (lambda (n)
    (if (zero? n)
        1
        (* n ((f f) (sub1 n))))))
```

such that factorial is ostensibly defined by

```
(mk-fact mk-fact)
```

Does that work? Substituting as before, we get

```
(lambda (n)
  (if (zero? n)
      1
      (* n ( (mk-fact mk-fact) (sub1 n)))))
```

This of course means we can substitute the inner application also:

```
(lambda (n)
  (if (zero? n)
      1
      (* n (  (lambda (n)
                (if (zero? n)
                    1
                    (* n ( (mk-fact mk-fact) (sub1 n)))))) (sub1 n)))))
```

and now we can see the recursion unwind: as we need another copy of factorial, the application of *mk-fact* to itself generates a fresh copy. Thus, we have a satifactory solution to the problem of defining the "recursive" factorial function without any use of recursion!

To summarize, we have the following definition,

```
mk-fact ≡
(lambda (f)
  (lambda (n)
    (if (zero? n)
        1
        (* n ((f f) (sub1 n))))))
```

with factorial defined by

fact ≡
(*mk-fact mk-fact*)

That is, factorial is

```
((lambda (mk-fact)
   (mk-fact mk-fact))
 (lambda (f)
   (lambda (n)
     (if (zero? n)
         1
         (* n ((f f) (sub1 n)))))))
```

(Test this! In a fresh Scheme session, apply this expression directly to numeric values and make sure you get the factorial of the input as a result. Pretty amazing, huh?)

While this is a correct implementation of factorial, we seem to be writing a lot of code relative to the recursive version defined using **define**. Furthermore, we would like to know how much of this solution can apply to other functions also. With that in mind, let's try to refactor this code a little. What would we *like* to write? As programmers, we would rather not have to keep track of the self-application in the body; that is, we would rather write

```
(make-recursive-procedure
  (lambda (fact)
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (sub1 n)))))))
```

which looks almost exactly like the definition created using **define**. So, how do we get the self-application out?

Observe that the definition of factorial above is equivalent to this one:

```
((lambda (mk-fact)
   (mk-fact mk-fact))
 (lambda (f)
   (
    (lambda (g)
      (lambda (n)
        (if (zero? n)
            1
            (* n (g (sub1 n))))))
    (f f))))
```

All we have done is introduce a new level of abstraction, binding g to the self-application of f. Note, however, that the boxed expression is precisely the definition of factorial that we wanted to write, except

that *fact* is called *g*. One more level of abstraction separates the factorial-specific code from the recursive function generator:

```
make-recursive-procedure ≡
(lambda (p)
  ((lambda (mk-fact)
     (mk-fact mk-fact))
   (lambda (f)
     (p
      (f f)))))
```

In fact, because this code has nothing to do with defining factorial at all, we can rename *mk-fact*:

```
make-recursive-procedure ≡
(lambda (p)
  ((lambda (f)
     (f f))
   (lambda (f)
     (p (f f)))))
```

This is now a generic procedure that creates recursive procedures out of its argument! It is remarkable that such a procedure even exists; its structure is daunting at first sight, but relatively easy to understand once you grasp the need for "copies" of a procedure, and that self-application generates as many of these copies as necessary.

In the literature, the procedure *make-recursive-procedure* is known as the *Y combinator*. It is sometimes also known as a "fixed-point combinator", because it computes the fixed-point of its argument procedure.

The Lambda Calculus

With the definition of the Y combinator, we have reduced all of Scheme to just three primitives: procedure definition, procedure application, and variables. With just those three, we have provided an encoding of all of the rest of the language. This compact little language is the core of what is known, also for historical reasons, as the *lambda calculus*. The "calculus" part of the language is beyond the scope of our study.

In the 1930s, several mathematicians were asking fundamental questions about what could be computed procedurally, and about the relative power of different formalisms. While Alan Turing was defining his Turing machine formalism, Alonzo Church and several others created an alternate formalism: the lambda calculus. These mathematicians were able to demonstrate that several of their formalisms—particularly these two—were equivalent in expressive power, so theoreticians could choose one or the other based on convenience and suitability, without worrying about expressive constraints. (To this day, many choose to use the lambda calculus and its variants since it offers so much more expressive power than a Turing machine.) Indeed, the fact that so many independently-derived formalisms had the same expressive power led to the formulation of the Church-Turing thesis: that *no* formal language is more powerful than those defined by Church and Turing (the lambda calculus and the Turing machine, respectively).

Exercise 22.4.1 *Type the definition of make-recursive-procedure into Scheme and use it to create a recursive factorial procedure:*

```
(make-recursive-procedure
  (lambda (fact)
    (lambda (n)
      (if (zero? n)
          1
          (* n (fact (sub1 n)))))))
```

What do you observe? Explain and correct.

Chapter 23

Semantics

We have been writing interpreters in Scheme in order to understand various features of programming languages. What if we want to explain our interpreter to someone else? If that person doesn't know Scheme, we can't communicate how our interpreter works. It would be convenient to have some common language for explaining interpreters. We already have one: math!

Let's try some simple examples. If our program is a number n, it just evaluates to some mathematical representation of n. We'll use a $\widehat{n}$ to represent this *number*, whereas n itself will hold the *numeral*. For instance, the numeral 5 is represented by the number $\widehat{5}$ (note the subtle differences in typesetting!). In other words, we will write

$$n \Rightarrow \widehat{n}$$

where we read $\Rightarrow$ as "reduces to". Numbers are already values, so they don't need further reduction.

How about addition? We might be tempted to write

$$\{+ \;\; l \;\; r\} \Rightarrow \widehat{l + r}$$

In particular, the addition to the left of the $\Rightarrow$ is in the programming language, while the one on the right happens in mathematics and results in a number. That is, the addition symbol on the left is *syntactic*. It could map to any mathematical operation. A particularly perverse language might map it to multiplication, but more realistically, it is likely to map to addition modulo some base to reflect fixed-precision arithmetic. It is the expression on the right that gives it meaning, and in this case it assigns the meaning we would expect (corresponding, say, to DrScheme's use of unlimited-precision numbers for integers and rationals).

That said, this definition is unsatisfactory. Mathematical addition only works on numbers, but l and r might each be complex expressions in need of reduction to a value (in particular, a number) so they can be added together. We denote this as follows:

$$\frac{l \Rightarrow \widehat{l_v} \qquad r \Rightarrow \widehat{r_v}}{\{+\; l\; r\} \Rightarrow \widehat{l_v + r_v}}$$

The terms above the bar are called the *antecedents*, and those below are the *consequents*. This rule is just a convenient way of writing an "if ... then" expression: it says that *if* the conditions in the antecedent hold, *then* those in the consequent hold. If there are multiple conditions in the antecedent, they must all hold for

the rule to hold. So we read the rule above as: *if* l reduces to l_v, *and if* r reduces to r_v, *then* adding the respective expressions results in the sum of their values. (In particular, it makes sense to add l_v and r_v, since each is now a number.) A rule of this form is called a *judgment*, because based on the truth of the conditions in the antecedent, it issues a judgment in the consequent (in this case, that the sum will be a particular value).

These rules subtly also *bind* names to values. That is, a different way of reading the rule is not as an "if ... then" but rather as an imperative: it says "reduce l, call the result l_v; reduce r, call its result r_v; if these two succeed, then add l_v and r_v, and declare the sum the result for the entire expression". Seen this way, l and r are bound in the consequent to the sub-expressions of the addition term, while l_v and r_v are bound in the antecedent to the results of evaluation (or reduction). This representation truly is an abstract description of the interpreter.

Let's turn our attention to functions. We want them to evaluate to closures, which consist of a name, a body and an environment. How do we represent a structure in mathematics? A structure is simply a tuple, in this case a triple. (If we had multiple kinds of tuples, we might use tags to distinguish between them, but for now that won't be necessary.) We would like to write

$$\texttt{\{fun \{}i\texttt{\}}\ b\texttt{\}} \Rightarrow \langle i, b, ??? \rangle$$

but the problem is we don't have a value for the environment to store in the closure. So we'll have to make the environment explicit. From now on, $\Rightarrow$ will always have a term and an environment on the left, and a value on the right. We first rewrite our two existing reduction rules:

$$n, \mathscr{E} \Rightarrow \widehat{n}$$

$$\frac{l, \mathscr{E} \Rightarrow \widehat{l_v} \qquad r, \mathscr{E} \Rightarrow \widehat{r_v}}{\{+\ l\ r\}, \mathscr{E} \Rightarrow \widehat{l_v + r_v}}$$

Now we can define a reduction rule for functions:

$$\texttt{\{fun \{}i\texttt{\}}\ b\texttt{\}}, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E} \rangle$$

Given an environment, we can also look up the value of identifiers:

$$i, \mathscr{E} \Rightarrow \mathscr{E}(i)$$

All that remains is application. As with addition, application must first evaluate its subexpressions, so the general form of an application must be as follows:

$$\frac{f, \mathscr{E} \Rightarrow ??? \qquad a, \mathscr{E} \Rightarrow ???}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

What kind of value must f reduce to? A closure, naturally:

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow ???}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

(We'll use $\mathscr{E}'$ to represent to closure environment to make clear that it may be different from $\mathscr{E}$.) We don't particularly care what kind of value a reduces to; we're just going to substitute it:

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v}{\{f\ a\}, \mathscr{E} \Rightarrow ???}$$

But what do we write below? We have to evaluate the body, b, in the extended environment; whatever value it returns is the value of the application. So the evaluation of b also moves into the antecedent:

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v \qquad b, ??? \Rightarrow b_v}{\{f\ a\}, \mathscr{E} \Rightarrow b_v}$$

In what environment do we reduce the body? It has to be the environment in the closure; if we use $\mathscr{E}$ instead of $\mathscr{E}'$, we introduce dynamic rather than static scoping! But additionally, we must extend $\mathscr{E}'$ with a binding for the identifier named by i; in particular, it must be bound to the value of the argument. We can write all this concisely as

$$\frac{f, \mathscr{E} \Rightarrow \langle i, b, \mathscr{E}' \rangle \qquad a, \mathscr{E} \Rightarrow a_v \qquad b, \mathscr{E}'[i \leftarrow a_v] \Rightarrow b_v}{\{f\ a\}, \mathscr{E} \Rightarrow b_v}$$

where $\mathscr{E}'[i \leftarrow a_v]$ means "the environment $\mathscr{E}'$ extended with the identifier i bound to the value a_v". If $\mathscr{E}'$ already has a binding for i, this extension shadows that binding.

The judicious use of names conveys information here. We're demanding that the value used to extend the environment must be the same as that resulting from evaluating a: the use of a_v in both places indicates that. It also places an ordering on operations: clearly the environment can't be extended until a_v is available, so the argument must evaluate before application can proceed with the function's body. The choice of two different names for environments—$\mathscr{E}$ and $\mathscr{E}'$—denotes that the two environments need not be the same.

We call this a *big-step operational semantics*. It's a *semantics* because it ascribes meanings to programs. (We can see how a small change can result in dynamic instead of static scope and, more mundanely, that the meaning of + is given to be addition, not some other binary operation.) It's *operational* because evaluation largely proceeds in a mechanical fashion; we aren't compiling the entire program into a mathematical object and using fancy math to reduce it to an answer. Finally, it's *big-step* because $\Rightarrow$ reduces expressions down to irreducible answers. In contrast, a *small-step* semantics performs one atomic reduction at a time, rather like a stepper in a programming environment.

Exercise 23.0.2 *Extend the semantics to capture conditionals.*

Exercise 23.0.3 *Extend the semantics to capture lists.*
Hint: *You may want to consider tagging tuples.*

Exercise 23.0.4 *Extend the semantics to capture recursion.*

Exercise 23.0.5 *Alter the semantics to reflect lazy evaluation instead.*

Part X

Types

Chapter 24

Introduction

Until now, we've largely ignored the problem of program errors. We haven't done so entirely: if a programmer writes

```
{fun {x}}
```

we do reject this program, because it isn't syntactically legal—every function must have a body. But what if, instead, he were to write

```
{+ 3
   {fun {x} x}}
```

? Right now, our interpreter might produce an error such as

```
num-n: not a number
```

A check deep in the bowels of our interpreter is flagging the use of a non-numeric value in a position expecting a number.

At this point, we can make the same distinction between the syntactic and meta levels about *errors* as we did about representations. The error above is an error at the *syntactic* level,[1] because the interpreter is checking for the correct use of its internal representation. Suppose we had division in the interpreted language, and the corresponding *num/* procedure failed to check that the denominator was non-zero; then the interpreter's behavior would be that of Scheme's on division-by-zero. If we had expected an error and Scheme did not flag one (or vice versa), then the interpreter would be unfaithful to the intent of the interpreted language.

Of course, this discussion about the source of error messages somewhat misses the point: we really ought to reject this program without ever executing it. But rejecting it is difficult because this program is legitimate from the perspective of the parser. It's only illegal from the *semantic* viewpoint, it is the meaning, as opposed to the syntax, of + that does not accept functions as arguments. Therefore, we clearly need a more sophisticated layer that checks for the validity of programs.

[1] Not to be confused with a syntax error!

How hard is this? Rejecting the example above seems pretty trivial: indeed, it's so easy, we could almost build this into the parser (to not accept programs that have *syntactic* functions as arguments to arithmetic primitives). Let's think more broadly. Sometimes it does not seem much harder: for instance,

```
{with {f {fun {x} {+ x 1}}}
  {+ 3
     {f 5}}}
```

is clearly legal, whereas

```
{with {f {fun {x}
           {fun {y} {+ x y}}}}
  {+ 3
     {f 5}}}
```

is not. Here, simply substituting `f` in the body seems to be enough. The problem does not quite reduce to the parsing problem that we had earlier—a function application is necessary to determine the program's validity. But consider this program:

```
{fun {f}
  {+ 3
     {f 5}}}
```

Is this program valid? Clearly, it depends on whether or not `f`, when applied to `5`, evaluates to a number. Since this expression may be used in many different contexts, we cannot know whether or not this is legal without examining each application, which in turn may depend on other substitutions, and so on. In short, it appears that we will need to run the program just to determine whether `f` is always bound to a function, and one that can accept numbers—but running the program is precisely what we're trying to avoid!

We now commence the study of *types* and *type systems*, which are designed to identify the abuse of types before executing a program. First, we need to build an intuition for the problems that types can address, and the obstacles that they face. Consider the following program:

```
{+ 3
   {if0 mystery
        5
        {fun {x} x}}}
```

This program executes successfully (and evaluates to `8`) if `mystery` is bound to `0`, otherwise it results in an error. The value of `mystery` might arise from any number of sources. For instance, it may be bound to `0` only if some mathematical statement, such as the Collatz conjecture, is true.[2] In fact, we don't even need to explore something quite so exotic: our program may simply be

[2]Consider the function $f(n)$ defined as follows: If n is even, divide n by 2; if odd, compute $3n+1$. The Collatz conjecture posits that, for every positive integer n, there exists some k such that $f^k(n) = 1$. (The sequences demonstrating convergence to 1 are often quite long, even for small numbers! For instance: $7 \to 22 \to 11 \to 34 \to 17 \to 52 \to 26 \to 13 \to 40 \to 20 \to 10 \to 5 \to 16 \to 8 \to 4 \to 2 \to 1$.)

```
{+ 3
   {if0 {read-number}
        5
        {fun {x} x}}}
```

Unless we can read the user's mind, we have no way of knowing whether this program will execute without error. In general, even without involving the mystery of mathematical conjectures or the vicissitudes of users, we cannot statically determine whether a program will halt with an error, because of the Halting Problem.

This highlights an important moral:

> Type systems are always prey to the Halting Problem. Consequently, a type system for a general-purpose language must always either over- or under-approximate: either it must reject programs that might have run without an error, or it must accept programs that will error when executed.

While this is a problem in theory, what impact does this have on practice? Quite a bit, it turns out. In languages like Java, programmers *think* they have the benefit of a type system, but in fact many common programming patterns force programmers to employ *casts* instead. Casts intentionally subvert the type system and leave checking for execution time. This indicates that Java's evolution is far from complete. In contrast, most of the type problems of Java are not manifest in a language like ML, but its type systems still holds a few (subtler) lurking problems. In short, there is still much to do before we can consider type system design a solved problem.

24.1 What Are Types?

A *type* is any property of a program that we can establish without executing the program. In particular, types capture the intuition above that we would like to predict a program's behavior without executing it. Of course, given a general-purpose programming language, we cannot predict its behavior entirely without execution (think of the user input example, for instance). So any static prediction of behavior must necessarily be an approximation of what happens. People conventionally use the term *type* to refer not just to any approximation, but one that is an abstraction of the set of values.

A type labels every expression in the language, recording what kind of value evaluating that expression will yield. That is, types describe invariants that hold for all executions of a program. They approximate this information in that they typically record only what *kind* of value the expression yields, not the precise value itself. For instance, types for the language we have seen so far might include `number` and `function`. The operator + consumes only values of type `number`, thereby rejecting a program of the form

```
{+ 3
   {fun {x} x}}
```

To reject this program, we did not need to know precisely which function was the second argument to +, be it `{fun {x} x}` or `{fun {x} {fun {y} {+ x y}}}`. Since we can easily infer that `3` has

type `number` and `{fun {x} x}` has type `function`, we have all the information we need to reject the program without executing it.

Note that we are careful to refer to *valid* programs, but never *correct* ones. Types do not ensure the correctness of a program. They only guarantee that the program does not make certain kinds of errors. Many errors lie beyond the ambit of a type system, however, and are therefore not caught by it. Many type systems will not, for instance, distinguish between a program that sorts values in ascending order from one that sorts them in descending order, yet the difference between those two is usually critical for a program's overall correctness.

24.2 Type System Design Forces

Designing a type system involves finding a careful balance between two competing forces:

1. Having more information makes it possible to draw richer conclusions about a program's behavior, thereby rejecting fewer valid programs or permitting fewer buggy ones.

2. Acquiring more information is difficult:

   - It may place unacceptable restrictions on the programming language.
   - It may incur greater computational expense.
   - It may force the user to annotate parts of a program. Many programmers (sometimes unfairly) balk at writing anything beyond executable code, and may thus view the annotations as onerous.
   - It may ultimately hit the limits of computability, an unsurpassable barrier. (Often, designers can surpass this barrier by changing the problem slightly, though this usually moves the task into one of the three categories above.)

24.3 Why Types?

Types form a very valuable first line of defense against program errors. Of course, a poorly-designed type system can be quite frustrating: Java programming sometimes has this flavor. A powerful type system such as that of ML, however, is a pleasure to use. ML programmers, for instance, claim that programs that type correctly often work correctly within very few development iterations.

Types that have not been subverted (by, for instance, casts in Java) perform several valuable roles:

- When type systems detect legitimate program errors, they help reduce the time spent debugging.

- Type systems catch errors in code that is not executed by the programmer. This matters because if a programmer constructs a weak test suite, many parts of the system may receive no testing. The system may thus fail after deployment rather than during the testing stage. (Dually, however, passing a type checker makes many programmers construct poorer test suites—a most undesirable and unfortunate consequence!)

- Types help document the program. As we discussed above, a type is an abstraction of the values that an expression will hold. Explicit type declarations therefore provide an approximate description of code's behavior.
- Compilers can exploit types to make programs execute faster, consume less space, spend less time in garbage collection, and so on.
- While no language can eliminate arbitrarily ugly code, a type system imposes a baseline of order that prevents at least a few truly impenetrable programs—or, at least, prohibits *certain kinds* of terrible coding styles.

Chapter 25

Type Judgments

25.1 What They Are

First, we must agree on a language of types. Recall that types need to abstract over sets of values; earlier, we suggested two possible types, `number` and `function`. Since those are the only kinds of values we have for now, let's use those as our types.

We present a type system as a collection of rules, known formally as *type judgments*, which describe how to determine the type of an expression.[1] There must be at least one type rule for every kind of syntactic construct so that, given a program, at least one type rule applies to every sub-term. Judgments are often recursive, determining an expression's type from the types of its parts.

The type of any numeral is `number`:

$$n : \texttt{number}$$

(read this as saying "any numeral n has type `number`") and of any function is `function`:

$$\{\texttt{fun } \{i\} \ b\} : \texttt{function}$$

but what is the type of an identifier? Clearly, we need a *type environment* (a mapping from identifiers to *types*). It's conventional to use Γ (the upper-case Greek "gamma") for the type environment. As with the value environment, the type environment must appear on the left of every type judgment. All type judgments will have the following form:

$$\Gamma \vdash e : t$$

where e is an expression and t is a type, which we read as "Γ proves that e has type t". Thus,

$$\Gamma \vdash n : \texttt{number}$$

$$\{\texttt{fun } \{i\} \ b\} : \texttt{function}$$

$$\Gamma \vdash i : \Gamma(i)$$

[1] A *type system* for us is a collection of types, the corresponding judgments that ascribe types to expressions, *and* an algorithm for performing this ascription. For many languages a simple algorithm suffices, but as languages get more sophisticated, devising this algorithm can become quite difficult, as we will see in Section 30.

The last rule simply says that the type of identifier i is whatever type it is bound to in the environment.

This leaves only addition and application. Addition is quite easy:

$$\frac{\Gamma \vdash l : \texttt{number} \qquad \Gamma \vdash r : \texttt{number}}{\Gamma \vdash \{+\ \ l\ \ r\} : \texttt{number}}$$

All this leaves is the rule for application. We know it must have roughly the following form:

$$\frac{\Gamma \vdash f : \texttt{function} \qquad \Gamma \vdash a : \tau_a \qquad \cdots}{\Gamma \vdash \{f\ \ a\} : ???}$$

where τ_a is the type of the expression a (we will often use τ to name an unknown type).

What's missing? Compare this against the semantic rule for applications. There, the representation of a function held an environment to ensure we implemented static scoping. Do we need to do something similar here?

For now, we'll take a much simpler route. We'll demand that the programmer *annotate* each function with the type it consumes and the type it returns. This will become part of a modified function syntax. That is, the language becomes

```
<TFWAE> ::= ...
          | {fun {<id> : <type>} : <type> <TFWAE>}
```

(the "T", naturally, stands for "typed") where the two type annotations are now required: the one immediately after the argument dictates what type of value the function consumes, while that after the argument but before the body dictates what type it returns. An example of a function definition in this language is

```
{fun {x : number} : number
  {+ x x}}
```

We must also change our type grammar; to represent function types we conventionally use an arrow, where the type at the tail of the arrow represents the type of the argument and that at the arrow's head represents the type of the function's return value:

```
<type> ::= number
         | (<type> -> <type>)
```

(notice that we have dropped the overly naive type `function` from our type language). Thus, the type of the function above would be `(number -> number)`. The type of the outer function below

```
{fun {x : number} : (number -> number)
  {fun {y : number} : number
    {+ x y}}}
```

is `(number -> (number -> number))`, while the inner function has type `(number -> number)`.

Equipped with these types, the problem of checking applications becomes easy:

$$\frac{\Gamma \vdash f : (\tau_1 \rightarrow \tau_2) \qquad \Gamma \vdash a : \tau_1}{\Gamma \vdash \{f\ \ a\} : \tau_2}$$

That is, if you provide an argument of the type the function is expecting, it will provide a value of the type it promises. Notice how the judicious use of the same type name τ_1 and τ_2 accurately captures the sharing we desire.

There is one final bit to the introductory type puzzle: how can we be sure the programmer will not lie? That is, a programmer might annotate a function with a type that is completely wrong (or even malicious). (A different way to look at this is, having rid ourselves of the type `function`, we must revisit the typing rule for a function declaration.) Fortunately, we can guard against cheating and mistakes quite easily: instead of blindly accepting the programmer's type annotation, we check it:

$$\frac{\Gamma[i \leftarrow \tau_1] \vdash b : \tau_2}{\Gamma \vdash \texttt{fun}\ \{i\ \texttt{:}\ \tau_1\}\ \texttt{:}\ \tau_2\ b\} : (\tau_1 \rightarrow \tau_2)}$$

This rule says that we will believe the programmer's annotation if the body has type τ_2 when we extend the environment with i bound to τ_1.

There is an important relationship between the type judgments for function declaration and for application:

- When typing the function declaration, we *assume* the argument will have the right type and *guarantee* that the body, or result, will have the promised type.

- When typing a function application, we *guarantee* the argument has the type the function demands, and *assume* the result will have the type the function promises.

This interplay between assumptions and guarantees is quite crucial to typing functions. The two "sides" are carefully balanced against each other to avoid fallacious reasoning about program behavior. In addition, just as `number` does not specify which number will be used, a function type does not limit which of many functions will be used. If, for instance, the type of a function is `(number -> number)`, the function could be either increment or decrement (or a lot else, besides). The type checker is able to reject misuse of *any* function that has this type without needing to know which actual function the programmer will use.

By the way, it would help to understand the status of terms like i and b and n in these judgments. They are "variable" in the sense that they will be replaced by some program term: for instance, $\{\texttt{fun}\ \{i\ \texttt{:}\ \tau_1\}\ \texttt{:}\ \tau_2\ b\}$ may be instantiated to `{fun {x : number} : number x}`, with i replaced by `x`, and so forth. But they are not program variables; rather, they are variables that stand for program text (including program variables). They are therefore called *metavariables*.

Exercise 25.1.1 *It's possible to elide the return type annotation on a function declaration, leaving only the argument type annotation. Do you see how?*

Exercise 25.1.2 *Because functions can be nested within each other, a function body may not be closed at the time of type-checking it. But we don't seem to capture the definition environment for types the way we did for procedures. So how does such a function definition type check? For instance, how does the second example of a typed procedure above pass this type system?*

25.2 How Type Judgments Work

Let's see how the set of type judgments described above accepts and rejects programs.

1. Let's take a simple program,

```
{+ 2
   {+ 5 7}}
```

   We stack type judgments for this term as follows:

$$\dfrac{\emptyset \vdash \texttt{2} : \texttt{number} \qquad \dfrac{\emptyset \vdash \texttt{5} : \texttt{number} \qquad \emptyset \vdash \texttt{7} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 5 7\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{+ 2 \{+ 5 7\}\}} : \texttt{number}}$$

   This is a *type judgment tree*.[2] Each node in the tree uses one of the type judgments to determine the type of an expression. At the leaves (the "tops") are, obviously, the judgments that do not have an antecedent (technically known as the *axioms*); in this program, we only use the axiom that judges numbers. The other two nodes in the tree both use the judgment on addition. The metavariables in the judgments (such as *l* and *r* for addition) are replaced here by actual expressions (such as `2`, `5`, `7` and `{+ 5 7}`): we can employ a judgment only when the pattern matches consistently. Just as we begin evaluation in the empty environment, we begin type checking in the empty *type* environment; hence we have $\emptyset$ in place of the generic Γ.

   Observe that at the end, the result is the type `number`, not the value 14.

2. Now let's examine a program that contains a function:

```
{{fun {x : number} : number
   {+ x 3}}
 5}
```

   The type judgment tree looks as follows:

$$\dfrac{\dfrac{\dfrac{[\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{x} : \texttt{number} \qquad [\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{3} : \texttt{number}}{[\texttt{x}\leftarrow\texttt{number}] \vdash \texttt{\{+ x 3\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{fun \{x : number\} : number \{+ x 3\}\}} : (\texttt{number}\rightarrow\texttt{number})} \qquad \emptyset \vdash \texttt{5} : \texttt{number}}{\emptyset \vdash \texttt{\{\{fun \{x : number\} : number \{+ x 3\}\} 5\}} : \texttt{number}}$$

   When matching the sub-tree at the top-left, where we have just Γ in the type judgment, we have the extended environment in the actual derivation tree. We must use the same (extended) environment consistently, otherwise the type judgment for addition cannot be applied. The set of judgments used

[2] If it doesn't look like a tree to you, it's because you've been in computer science too long and have forgotten that real trees grow upward, not downward. Botanically, however, most of these "trees" are really shrubs.

to assign this type is quite different from the set of judgments we would use to evaluate the program: in particular, we type "under the `fun`", i.e., we go into the body of the `fun` even if the function is never applied. In contrast, we would never evaluate the body of a function unless and until the function was applied to an actual parameter.

3. Finally, let's see what the type judgments do with a program that we know to contain a type error:

```
{+ 3
   {fun {x : number} : number
     x}}
```

The type judgment tree begins as follows:

$$\frac{???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

We don't yet know what type (if any) we will be able to ascribe to the program, but let's forge on: hopefully it'll become clear soon. Since the expression is an addition, we should discharge the obligation that each sub-expression must have numeric type. First for the left child:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad ???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

Now for the right sub-expression. First let's write out the sub-expression, then determine its type:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : ???}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

As per the judgments we have defined, any function expression must have an arrow type:

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : (??? \rightarrow ???)}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

This does the type checker no good, however, because arrow types are distinct from numeric types, so the resulting tree above does not match the form of the addition judgment (no matter what goes in place of the two ???'s). To match the addition judgment the tree must have the form

$$\frac{\emptyset \vdash \texttt{3} : \texttt{number} \quad \emptyset \vdash \texttt{\{fun \{x : number\} : number x\}} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 3 \{fun \{x : number\} : number x\}\}} : ???}$$

Unfortunately, we do not have any judgments that let us conclude that a syntactic function term can have a numeric type. So this doesn't work either.

In short, we cannot construct a legal type derivation tree for the original term. Notice that this is not the same as saying that the tree directly identifies an error: it does not. A type error occurs when we are *unable to construct a type judgment tree*.

This is subtle enough to bear repeating: To flag a program as erroneous, we must *prove* that no type derivation tree can possibly exist for that term. But perhaps some sequence of judgments that we haven't thought of exists that (a) is legal and (b) correctly ascribes a type to the term! To avoid this we may need to employ quite a sophisticated proof technique, even human knowledge. (In the third example above, for instance, we say, "we do not have any judgments that let us conclude that a syntactic function term can have a numeric type". But how do we know this is true? We can only conclude this by carefully studying the structure of the judgments. A computer program might not be so lucky, and in fact may get stuck endlessly trying judgments!)

This is why a set of type judgments alone does not suffice: what we're really interested in is a type system that includes an algorithm for type-checking. For the set of judgments we've written here, and indeed for the ones we'll study initially, a simple top-down, syntax-directed algorithm suffices for (a) determining the type of each expression, and (b) concluding that some expressions manifest type errors. As our type judgments get more sophisticated, we will need to develop more complex algorithms to continue producing tractable and useful type systems.

Chapter 26

Typing Control

26.1 Conditionals

Let's expand our language with a conditional construct. We can use `if0` like before, but for generality it's going to be more convenient to have a proper conditional and a language of predicates. The type judgment for the conditional must have the following form:

$$\frac{\Gamma \vdash c : ??? \qquad \Gamma \vdash t : ??? \qquad \Gamma \vdash e : ???}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

where c is the conditional, t the "then"-expression, and e the "else"-expression.

Let's begin with the type for c. What should it be? In a language like Scheme we permit any value, but in a stricter, typed language, we might demand that the expression always evaluate to a boolean. (After all, if the point is to detect errors sooner, then it does us no good to be overly lax in our type rules.) However, we don't yet have such a type in our type language, so we must first extend that language:

```
<type> ::= number
         | boolean
         | (<type> -> <type>)
```

Armed with the new type, we can now ascribe a type to the conditional expression:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : ??? \qquad \Gamma \vdash e : ???}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

Now what of the other two, and of the result of the expression? One option is, naturally, to allow both arms of the conditional to have whatever types the programmer wants:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : \tau_1 \qquad \Gamma \vdash e : \tau_2}{\Gamma \vdash \{\texttt{if}\ \ c\ \ t\ \ e\} : ???}$$

By using two distinct type variables, we do not demand any conformity between the actual types of the arms. By permitting this flexibility, however, we encounter two problems. The first is that it isn't clear what

type to ascribe to the expression overall.[1] Second, it reduces our ability to trap program errors. Consider a program like this:

```
{+ 3
   {if {is-zero mystery}
       5
       {fun {x} x}}}
```

Because we know nothing about `mystery`, we must conservatively conclude that it *might* be non-zero, which means eventually we are going to see a type error that we only catch at run-time. But why permit the programmer to write such a program at all? We might as well prevent it from ever executing. Therefore, we use the following rule to type conditionals:

$$\frac{\Gamma \vdash c : \texttt{boolean} \qquad \Gamma \vdash t : \tau \qquad \Gamma \vdash e : \tau}{\Gamma \vdash \{\texttt{if } c \ t \ e\} : \tau}$$

Notice that by forcing the two arms to have the same type, we can assign that common type to the entire expression, so the type system does not need to know which branch was chosen on a given execution: the type remains the same.

Having added conditionals and the type `boolean` isn't very useful yet, because we haven't yet introduced predicates to use in the test position of the conditional. Indeed, we can easily see that this is true becuase we have not yet written a function type with `boolean` on the right-hand side of the arrow. You can, however, easily imagine adding procedures such as `is-zero`, with type `number -> boolean`.

26.2 Recursion

Now that we have conditionals, if we can also implement recursion, we would have a Turing-complete language (that could, for instance, with a little more arithmetic support, enable writing factorial). So the next major piece of the puzzle is typing recursion.

Given the language TFAE (typed FAE), can we write a recursive program? Let's try to write an infinite loop. Our first attempt might be this FAE program

```
{with {f {fun {i}
              {f i}}}
  {f 10}}
```

which, expanded out, becomes

```
{{fun {f}
   {f 10}}
 {fun {i}
   {f i}}}
```

[1] It's tempting to create a new kind of type, a *union* type, so that the type of the expression is $\tau_1 \cup \tau_2$. This has far-reaching consequences, however, including a significant reduction in type-based guarantee of program reliability.

When we place type annotations on this program, we get

```
{{fun {f : (num -> num)} : num
   {f 10}}
 {fun {i : num} : num
   {f i}}}
```

These last two steps don't matter, of course. This program doesn't result in an infinite loop, because the `f` in the body of the function isn't bound, so after the first iteration, the program halts with an error.

As an aside, this error is easier to see in the typed program: when the type checker tries to check the type of the annotated program, it finds no type for `f` on the last line. Therefore, it would halt with a type error, preventing this erroneous program from ever executing.[2]

Okay, that didn't work, but we knew about that problem: we saw it in Section 9 when introducing recursion. At the time, we asked you to consider whether it was possible to write a recursive function without an explicit recursion construct, and Section 22 shows that it is indeed possible. The essence of the solution presented there is to use *self-application*:

```
{with {omega {fun {x}
                {x x}}}
  {omega omega}}
```

How does this work? Simply substituting `omega` with the function, we get

```
{{fun {x} {x x}}
 {fun {x} {x x}}}
```

Substituting again, we get

```
{{fun {x} {x x}}
 {fun {x} {x x}}}
```

and so on. In other words, this program executes forever. It is conventional to call the function ω (lower-case Greek "omega"), and the entire expression Ω (upper-case Greek "omega").[3]

Okay, so Ω seems to be our ticket. This is clearly an infinite loop in FAE. All we need to do is convert it to TFAE, which is simply a matter of annotating all procedures. Since there's only one, ω, this should be especially easy.

To annotate ω, we must provide a type for the argument and one for the result. Let's call the argument type, namely the type of `x`, τ_a and that of the result τ_r, so that $\omega : \tau_a \rightarrow \tau_r$. The body of ω is `{x x}`. From this, we can conclude that τ_a must be a function (arrow) type, since we use `x` in the function position of an application. That is, τ_a has the form $\tau_1 \rightarrow \tau_2$, for some τ_1 and τ_2 yet to be determined.

[2] In this particular case, of course, a simpler check would prevent the erroneous program from starting to execute, namely checking to ensure there are no free variables.

[3] Strictly speaking, it seems anachronistic to refer to the lower and upper "case" for the Greek alphabet, since the language predates moveable type in the West by nearly two millennia.

What can we say about τ_1 and τ_2? τ_1 must be whatever type `x`'s argument has. Since `x`'s argument is itself `x`, τ_1 must be the same as the type of `x`. We just said that `x` has type τ_a. This immediately implies that

$$\tau_a = \tau_1 \rightarrow \tau_2 = \tau_a \rightarrow \tau_2$$

In other words,

$$\tau_a = \tau_a \rightarrow \tau_2$$

What type can we write that satisfies this equation? In fact, no types in our type language can satisify it, because this type is recursive without a base case. Any type we try to write will end up being infinitely long. Since we cannot write an infinitely long type (recall that we're trying to annotate ω, so if the type is infinitely long, we'd never get around to finishing the text of the program), it follows by contradiction[4] that ω and Ω cannot be typed in our type system, and therefore their corresponding programs are not programs in TFAE. (We are being rather lax here—what we've provided is informal reasoning, not a proof—but such a proof does exist.)

26.3 Termination

We concluded our exploration of the type of Ω by saying that the annotation on the argument of ω must be infinitely long. A curious reader ought to ask, is there any connection between the boundlessness of the type and the fact that we're trying to perform a non-terminating computation? Or is it mere coincidence?

TFAE, which is a first cousin of a language you'll sometimes see referred to as the *simply-typed* lambda calculus,[5] enjoys a rather interesting property: it is said to be *strongly normalizing*. This intimidating term says of a programming language that no matter what program you write in the language, it will *always terminate*!

To understand why this property holds, think about our type language. The only way to create compound types is through the function constructor. But every time we apply a function, we discharge one function constructor: that is, we "erase an arrow". Therefore, after a finite number of function invocations, the computation must "run out of arrows".[6] Because only function applications can keep a computation running, the computation is forced to terminate.

This is a *very* informal argument for why this property holds—it is cetainly far from a proof (though, again, a formal proof of this property does exist). However, it does help us see why we must inevitably have bumped into an infinitely long type while trying to annotate the infinite loop.

What good is a language without infinite loops? There are in fact lots of programs that we would like to ensure will *not* run forever. These include:

- inner-loops of real-time systems
- program linkers

[4] We implicitly assumed it would be possible to annotate ω and explored what that type annotation would be. The contradiction is that no such annotation is possible.

[5] Why "simply"? You'll see what other options there are next week.

[6] Oddly, this never happens to mythological heroes.

- packet filters in network stacks
- client-side Web scripts
- network routers
- device (such as photocopier) initialization
- configuration files (such as Makefiles)

and so on. That's what makes the simply-typed lambda calculus so wonderful: instead of pondering and testing endlessly (so to speak), we get mathematical certitude that, with a correct implementation of the type checker, no infinite loops can sneak past us. In fact, the module system of the SML programming language is effectively an implementation of the simply-typed lambda calculus, thereby guaranteeing that no matter how complex a linking specification we write, the linking phase of the compiler will always terminate.

Exercise 26.3.1 *We've been told that the Halting Problem is undecidable. Yet here we have a language accompanied by a theorem that proves that all programs will terminate. In particular, then, the Halting Problem is not only very decidable, it's actually quite simple: In response to the question "Does this program halt", the answer is* always *"Yes!" Reconcile.*

Exercise 26.3.2 *While the simply-typed lambda calculus is fun to discuss, it may not be the most pliant programming language, even as the target of a compiler (much less something programmers write explicitly). Partly this is because it doesn't quite focus on the right problem. To a Web browsing user, for instance, what matters is whether a downloaded program runs* immediately*; five minutes isn't really distinguishable from non-termination.*

Consequently, a better variant of the lambda calculus might be one whose types reflect resources*, such as time and space. The "type" checker would then ask the user running the program for resource bounds, then determine whether the program can actually execute within the provided resources. Can you design and implement such a language? Can you write useful programs in it?*

26.4 Typed Recursive Programming

Strong normalization says we must provide an explicit recursion construct. To do this, we'll simply reintroduce our `rec` construct to define the language *TRCFAE*. The BNF for the language is

```
<TRCFAE> ::= ...
           | {rec {<id>  : <type> <TRCFAE>} <TRCFAE>}
```

with the same type language. Note that the `rec` construct needs an explicit type annotation also.

What is the type judgment for `rec`? It must be of the form

$$\frac{???}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

since we want to conclude something about the entire term. What goes in the antecedent? We can determine this more easily by realizing that a `rec` is a bit like an immediate function application. So just as with functions, we're going to have *assumptions* and *guarantees*—just both in the same rule.

We want to assume that τ_i is a legal annotation, and use that to check the body; but we also want to guarantee that τ_i is a legal annotation. Let's do them in that order. The former is relatively easy:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad ???}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

Now let's hazard a guess about the form of the latter:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma \vdash v : \tau}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

But what is the structure of the term named by v? Surely it has references to the identifier named by i in it, but i is almost certainly not bound in Γ (and even if it is, it's not bound to the value we want for i). Therefore, we'll have to extend Γ with a binding for i—not surprising, if you think about the scope of i in a `rec` term—to check v also:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma[i \leftarrow \tau_i] \vdash v : \tau}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

Is that right? Do we want v to have type τ, the type of the entire expression? Not quite: we want it to have the type we promised it would have, namely τ_i:

$$\frac{\Gamma[i \leftarrow \tau_i] \vdash b : \tau \qquad \Gamma[i \leftarrow \tau_i] \vdash v : \tau_i}{\Gamma \vdash \{\texttt{rec}\ \{i\ :\ \tau_i\ v\}\ b\} : \tau}$$

Now we can understand how the typing of recursion works. We extend the environment not once, but twice. The extension to type b is the one that *initiates* the recursion; the extension to type v is the one that *sustains* it. Both extensions are therefore necessary. And because a type checker doesn't actually run the program, it doesn't need an infinite number of arrows. When type checking is done and execution begins, the run-time system does, in some sense, need "an infinite quiver of arrows", but we've already seen how to implement that in Section 10.

Exercise 26.4.1 *Define the* BNF *entry and generate a type judgment for* `with` *in the typed language.*

Exercise 26.4.2 *Typing recursion looks simple, but it's actually worth studying in detail. Take a simple example such as* Ω *and work through the rules:*

- *Write* Ω *with type annotations so it passes the type checker. Draw the type judgment tree to make sure you understand why this version of* Ω *types.*
- *Does the expression named by v in* `rec` *have to be a procedure? Do the typing rules for* `rec` *depend on this?*

Chapter 27

Typing Data

27.1 Recursive Types

27.1.1 Declaring Recursive Types

We saw in the previous lecture how `rec` was necessary to write recursive *programs*. But what about defining recursive *types*? Recursive types are fundamental to computer science: even basic data structures like lists and trees are recursive (since the rest of a list is also a list, and each sub-tree is itself a tree).

Suppose we try to type the program

```
{rec {length : ???
       {fun {l : ???} : number
         {if {empty? l}
             0
             {+ 1 {length {rest l}}}}}}}
  {length {numCons 1 {numCons 2 {numCons 3 numEmpty}}}}}
```

What should we write in place of the question marks?

Let's consider the type of `l`. What kind of value can be an argument to `l`? Clearly a numeric cons, because that's the argument supplied in the first invocation of `length`. But eventually, a numeric empty is passed to `l` also. This means `l` needs to have *two* types: (numeric) cons and empty.

In languages like ML (and Java), procedures do not consume arguments of more than one distinct type. Instead, they force programmers to define a new type that encompasses all the possible arguments. This is precisely what a datatype definition, of the kind we have been writing in Scheme, permits us to do. So let's try to write down such a datatype in a hypothetical extension to our (typed) implemented language:

```
{datatype numList
    {[numEmpty]
     [numCons {fst : number}
              {rst : ???}]}
  {rec {length : (numList -> number) ...}
    {length ...}}}
```

We assume that a datatype declaration introduces a collection of *variants*, followed by an actual body that uses the datatype. What type annotation should we place on `rst`? This should be precisely the new type we are introducing, namely `numList`.

A datatype declaration therefore enables us to do a few distinct things all in one notation:

1. Give names to new types.
2. Introduce conditionally-defined types (*variants*).
3. Permit recursive definitions.

If these are truly distinct, we should consider whether there are more primitive operators that we may provide so a programmer can mix-and-match them as necessary.[1]

But how distinct are these three operations, really? Giving a type a new name would be only so useful if the type were simple (for instance, creating the name `bool` as an alias for `boolean` may be convenient, but it's certainly not conceptually significant), so this capability is most useful when the name is assigned to a complex type. Recursion needs a name to use for declaring self-references, so it depends on the ability to introduce a new name. Finally, well-founded recursion depends on having both recursive and non-recursive cases, meaning the recursive type must be defined as a collection of variants (of which at least one is not self-referential). So the three capabilities coalesce very nicely.

As you may have noticed above, the datatypes we have introduced in our typed language are a bit different from those we're using in Scheme. Our Scheme datatypes are defined at the top-level, while those in the implemented language enclose the expressions that refer to them. This is primarily to make it easier to deal with the scope of the introduced types. Obviously, a full-fledged language (like ML and Haskell) permits apparently top-level datatype declarations, but we'll make this simplifying assumption here.

27.1.2 Judgments for Recursive Types

Let's consider another example of a recursive type: a family tree.

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  ...}
```

This data definition allows us to describe as much of the genealogy as we know, and terminate the construction when we reach an unknown person. What type declarations ensue from this definition?

$$\mathit{unknown} : \rightarrow \mathit{FamilyTree}$$

$$\mathit{person} : \texttt{string} \times \mathit{FamilyTree} \times \mathit{FamilyTree} \rightarrow \mathit{FamilyTree}$$

[1] "Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary".

This doesn't yet give us a way of distinguishing between the two variants, and of selecting the fields in each variant. In Scheme, we use **type-case** to perform both of these operations. A corresponding case dispatcher for the above datatype might look like

```
{FamilyTree-cases v
  [{unknown} ...]
  [{person n m f} ...]}
```

Its pieces would be typed as follows:

$$\frac{\Gamma \vdash v : \mathit{FamilyTree} \qquad \Gamma \vdash e_1 : \tau \qquad \Gamma[n \leftarrow \texttt{string}, m \leftarrow \mathit{FamilyTree}, f \leftarrow \mathit{FamilyTree}] \vdash e_2 : \tau}{\Gamma \vdash \{\mathit{FamilyTree\text{-}cases}\ v\ \{[\mathit{unknown}]\ e_1\}\ \{[\mathit{person}\ n\ m\ f]\ e_2\}\} : \tau}$$

In other words, to determine the type of the entire `FamilyTree-cases` expression, τ, we first ensure that the value being dispatched is of the right type. Then we must make sure each branch of the switch returns a τ.[2] We can ensure that by checking each of the bodies in the right type environment. Because `unknown` has no fields, its `cases` branch binds no variables, so we check e_1 in Γ. In the branch for `person`, however, we bind three variables, so we must check the type of e_2 in a suitably extended Γ.

Though the judgment above is for a very specific type declaration, the general principle should be clear from what we've written. Effectively, the type checker introduces a new type rule for each typed `cases` statement based on the type declaration at the time it sees the declaration. Writing the judgment above in terms of subscripted parameters is tedious but easy.

Given the type rules above, consider the following program:

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {person "Mitochondrial Eve" {unknown} {unknown}}}
```

What is the type of the expression in the body of the datatype declaration? It's *FamilyTree*. But when the value escapes from the body of the declaration, how can we access it any longer? (We assume that the type checker renames types consistently, so *FamilyTree* in one scope is different from *FamilyTree* in another scope—just because the names are the same, the types should not conflate.) It basically becomes an *opaque type* that is no longer usable. This does not appear to be very useful at all![3]

At any rate, the type checker permitted a program that is quite useless, and we might want to prevent this. Therefore, we could place the restriction that the type defined in the datatype (in this case, *FamilyTree*) should be different from the type of the expression body τ. This prevents programmers from inadvertently returning values that nobody else can use.

[2] Based on the preceding discussion, if the two cases needed to return different types of values, how would you address this need in a language that enforced the type judgment above?

[3] Actually, you could use this to define the essence of a module or object system. These are called *existential types*. But we won't study them further in this course.

Obviously, this restriction doesn't reach far enough. Returning a vector of *FamilyTree* values avoids the restriction above, but the effect is the same: no part of the program outside the scope of the datatype can use these values. So we may want a more stringent restriction: the type being different *should not appear free* in τ.

This restriction may be overreaching, however. For instance, a programmer might define a new type, and return a package (a vector, say) consisting of two values: an instance of the new type, and a procedure that accesses the instances. For instance,

```
{datatype FamilyTree
    {[unknown]
     [person {name : string}
             {mother : FamilyTree}
             {father : FamilyTree}]}
  {with {unknown-person : FamilyTree {unknown}}
    {vector
      {person "Mitochondrial Eve"
              unknown-person
              unknown-person}
      {fun {v : FamilyTree} : string
        {FamilyTree-cases v
          [{unknown}      {error ...}]
          [{person n m f} n]}}}}}
```

In this vector, the first value is an instance of *FamilyTree*, while the second value is a procedure of type

$$\textit{FamilyTree} \rightarrow \texttt{string}$$

Other values, such as `unknown-person`, are safely hidden from access. If we lift the restriction of the previous paragraph, this becomes a legal pair of values to return from an expression. Notice that the pair in effect forms an *object*: you can't look into it, so the only way to access it is with the "public" procedure. Indeed, this kind of type definition sees use in defining object systems.

That said, we still don't have a clear description of what restriction to affix on the type judgment for datatypes. Modern programming languages address this quandary by affixing no restriction at all. Instead, they effectively force all type declarations to be at the "top" level. Consequently, no type name is ever unbound, so the issues of this section do not arise. When we do need to restrict access, we employ module systems to delimit the scope of type bindings.

27.1.3 Space for Datatype Variant Tags

One of the benefits programmers incur from using datatypes—beyond the error checking—is slightly better space consumption. (Note: "better space consumption" = "using less space".) Whereas without static type checking we would need tags that indicate both the type *and* the variant, we now need to store *only* the variant. Why? Because the type checker statically ensures that we won't pass the wrong kind of value to procedures! Therefore, the run-time system needs to use only as many bits as are necessary to distinguish

between all the *variants* of a type, rather than all the datatypes as well (in addition). Since the number of variants is usually quite small, of the order of 3–4, the number of bits necessary for the tags is usually small also.

We are now taking a big risk, however. In the liberal tagging regime, where we use both type and variant tags, we can be sure a program will never execute on the wrong kind of data. But if we switch to a more liberal tagging regime—one that doesn't store type tags also—we run a huge risk. If we perform an operation on a value of the wrong type, we may completely destroy our data. For instance, suppose we can somehow pass a *NumList* to a procedure expecting a *FamilyTree*. If the `FamilyTree-cases` operation looks only at the variant bits, it could end up accessing a `numCons` as if it were a `person`. But a `numCons` has only two fields; when the program accesses the third field of this variant, it is essentially getting junk values. Therefore, we have to be very careful performing these kinds of optimizations. How can we be sure they are safe?

Chapter 28

Type Soundness

We would like a guarantee that a program that passes a type checker will never exhibit certain kinds of errors when it runs. In particular, we would like to know that the type system did indeed abstract over values: that running the type checker *correctly predicted* (up to the limits of the abstraction) what the program would do. We call this property of a type system *type soundness*:[1]

> For all programs p, if the type of p is τ, then p will evaluate to a value that has type τ.

Note that the statement of type soundness *connects types with execution*. This tells the user that the type system is not some airy abstraction: what it predicts has bearing on practice, namely on the program's behavior when it eventually executes.

We have to be more careful about how we define type soundness. For instance, we say above (emphasis added) "*p will* evaluate to a value such that ...". But what if the program doesn't terminate? So we must recast this statement to say

> For all programs p, if the type of p is τ and p evaluates to v, then $v : \tau$.[2]

Actually, this isn't quite true either. What if the program executes an expression like (*first empty*)? There are a few options open to the language designer:

- Return a value such as -1. We hope you cringe at this idea! It means a program that fails to properly check for return values at every single place will potentially produce nonsensical results. (Such errors are common in C programs, where operators like `malloc` and `fopen` return special values but programmers routinely forget to check them.[3] Indeed, many of these errors lead to expensive, frustrating and threatening security violations.)

[1] The term "soundness" comes from mathematical logic.

[2] We could write this more explicitly as: For all programs p, if the type checker assigns p the type τ, and if the semantics says that p evaluates to a value v, then the type checker will also assign v the type τ.

[3] Ian Barland: "In Ginsberg's *Howl*, the name of the ancient god Moloch is used to exemplify society's embracing of soulless machines, machines for the sake of machines rather than the elegance they can embody. I do not consider it coincidental that this name sounds like 'malloc'."

- Diverge, i.e., go into an infinite loop. This approach is used by theoreticians (study the statement of type soundness carefully and you can see why), but as software engineers we should soundly (ahem) reject this.
- Raise an exception. This is the preferred modern solution.

Raising exceptions means the program does not terminate with a value, nor does it not terminate. We must therefore refine this statement still further:

> For all programs p, if the type of p is τ, p will, if it terminates, either evaluate to a value v such that $v : \tau$, or raise one of a well-defined set of exceptions.

The exceptions are a bit of a cop-out, because we can move arbitrarily many errors into that space. In Scheme, for instance, the trivial type checker rejects no programs, and all errors fall under the exceptions. In contrast, researchers work on very sophisticated languages where some traditional actions that would raise an exception (such as violating array bounds) instead become type errors. This last phrase of the type soundness statement therefore leaves lots of room for type system design.

As software engineers, we should care deeply about type soundness. To paraphrase Robin Milner, who first proved a modern language's soundness (specifically, for ML),

Well-typed programs do not go wrong.

That is, a program that passes the type checker (and is thus "well-typed") absolutely cannot exhibit certain classes of mistakes. [4]

Why is type soundness not obvious? Consider the following simple program (the details of the numbers aren't relevant):

```
{if0 {+ 1 2}
     {{fun {x : number} : number {+ 1 x}} 7}
     {{fun {x : number} : number {+ 1 {+ 2 x}}} 1}}
```

During execution, the program will explore only one branch of the conditional:

$$\dfrac{\dfrac{\texttt{1},\emptyset \Rightarrow \texttt{1} \qquad \texttt{2},\emptyset \Rightarrow \texttt{2}}{\texttt{\{+ 1 2\}},\emptyset \Rightarrow \texttt{3}} \qquad \dfrac{\qquad\qquad \ldots \qquad\qquad}{\texttt{\{\{fun ...\} 1\}},\emptyset \Rightarrow \texttt{4}}}{\texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 7\} \{\{fun ...\} 1\}\}},\emptyset \Rightarrow \texttt{4}}$$

but the type checker must explore both:

$$\dfrac{\dfrac{\emptyset \vdash \texttt{1} : \texttt{number} \qquad \emptyset \vdash \texttt{2} : \texttt{number}}{\emptyset \vdash \texttt{\{+ 1 2\}} : \texttt{number}} \qquad \dfrac{\qquad\qquad \ldots \qquad\qquad}{\emptyset \vdash \texttt{\{\{fun ...\} 7\}} : \texttt{number}} \qquad \dfrac{\qquad\qquad \ldots \qquad\qquad}{\emptyset \vdash \texttt{\{\{fun ...\} 1\}} : \texttt{number}}}{\emptyset \vdash \texttt{\{if0 \{+ 1 2\} \{\{fun ...\} 7\} \{\{fun ...\} 1\}\}} : \texttt{number}}$$

[4] The term "wrong" here is misleading. It refers to a particular kind of value, representing an erroneous configuration, in the semantics Milner was using; in that context, this slogan is tongue-in-cheek. Taken out of context it is misleading, because a well-typed program can still go wrong in the sense of producing erroneous output.

Furthermore, even for each expression, the proof trees in the semantics and the type world will be quite different (imagine if one of them contains recursion: the evaluator must iterate as many times as necessary to produce a value, while the type checker examines each expression only once). As a result, it is *far from obvious* that the two systems will have any relationship in their answers. This is why a theorem is not only necessary, but sometimes also difficult to prove.

Type soundness is, then, really a claim that the type system and run-time system (as represented by the semantics) are in sync. The type system erects certain abstractions, and the theorem states that the run-time system mirrors those abstractions. Most modern languages, like ML and Java, have this flavor.

In contrast, C and C++ *do not have sound type systems*. That is, the type system may define certain abstractions, but the run-time system does not honor and protect these. (In C++ it largely does for object types, but not for types inherited from C.) This is a particularly insidious kind of language, because the static type system lulls the programmer into thinking it will detect certain kinds of errors, but it fails to deliver on that promise during execution.

Actually, the reality of C is much more complex: C has *two different type systems*. There is one type system (with types such as `int`, `double` and even function types) at the level of the program, and a different type system, defined *solely* by lengths of bitstrings, at the level of execution. This is a kind of "bait-and-switch" operation on the part of the language. As a result, it isn't even meaningful to talk about soundness for C, because the static types and dynamic type representations simply don't agree. Instead, the C run-time system simply interprets bit sequences according to specified static types. (Procedures like `printf` are notorious for this: if you ask to print using the specifier `%s`, `printf` will simply print a sequence of characters until it hits a null-teriminator: never mind that the value you were pointing to was actually a double! This is of course why C is very powerful at low-level programming tasks, but how often do you actually need such power?)

To summarize all this, we introduce the notion of *type safety*:

> Type *safety* is the property that no primitive operation ever applies to values of the wrong type.

By primitive operation we mean not only addition and so forth, but also procedure application. *A safe language honors the abstraction boundaries it erects*. Since abstractions are crucial for designing and maintaining large systems, safety is a key software engineering attribute in a language. (Even most C++ libraries are safe, but the problem is you have to be sure no legacy C library isn't performing unsafe operations, too.) Using this concept, we can construct the following table:

	statically checked	not statically checked
type safe	ML, Java	Scheme
type unsafe	C, C++	assembly

The important thing to remember is, due to the Halting Problem, some checks simply can never be performed statically; something must always be deferred to execution time. The trade-off in type design is to maximize the number of these decisions statically without overly restricting the power of the programmer. The designers of different languages have divergent views on the powers a programmer should have.

So what is "strong typing"? This appears to be a meaningless phrase, and people often use it in a nonsensical fashion. To some it seems to mean "The language has a type checker". To others it means "The

language is sound" (that is, the type checker and run-time system are related). To most, it seems to just mean, "A language like Pascal, C or Java, related in a way I can't quite make precise". If someone uses this phrase, be sure to ask them to define it for you. (For amusement, watch them squirm.)

Chapter 29

Explicit Polymorphism

29.1 Motivation

Earlier, we looked at examples like the length procedure (from now on, we'll switch to Scheme with imaginary type annotations):

```
(define lengthNum
  (lambda (l : numlist) : number
    (cond
      [(numEmpty? l) 0]
      [(numCons? l) (add1 (lengthNum (numRest l)))])))
```

If we invoke *lengthNum* on (list 1 2 3), we would get 3 as the response.

Now suppose we apply *lengthNum* to (list 'a 'b 'c). What do we expect as a response? We might *expect* it to evaluate to 3, but that's not what we're going to get! Instead, we are going to get a type error (before invocation can even happen), because we are applying a procedure expecting a numlist to a value of type symlist (a list of symbols).

We can, of course, define another procedure for computing the length of lists of symbols:

```
(define lengthSym
  (lambda (l : symlist) : number
    (cond
      [(symEmpty? l) 0]
      [(symCons? l) (add1 (lengthSym (symRest l)))])))
```

Invoking *lengthSym* on (list 'a 'b 'c) will indeed return 3. But look closely at the difference between *lengthNum* and *lengthSym*: what changed in the code? Very little. The changes are almost all in the *type* annotations, not in the code that executes. This is not really surprising, because there is only one *length* procedure in Scheme, and it operates on all lists, no matter what values they might hold.

This is an unfortunate consequence of the type system we have studied. We introduced types to reduce the number of errors in our program (and for other reasons we've discussed, such as documentation), but in the process we've actually made it more difficult to write some programs. This is a constant tension in the

design of typed programming languages. Introducing new type mechanisms proscribes certain programs,[1] but in return it invalidates some reasonable programs, making them harder to write. The *length* example is a case in point.

Clearly computing the length of a list is very useful, so we might be tempted to somehow add *length* as a primitive in the language, and devise special type rules for it so that the type checker doesn't mind what kind of list is in use. This is a bad idea! There's a principle of language design that says it's generally unadvisable for language designers to retain special rights for themselves that they deny programmers who use their language. It's unadvisable because its condescending and paternalistic. It suggests the language designer somehow "knows better" than the programmer: trust us, we'll build you just the primitives you need. In fact, programmers tend to always exceed the creative bounds of the language designer. We can already see this in this simple example: Why *length* and not *reverse*? Why *length* and *reverse* but not *append*? Why all three and not *map*? Or *filter* or *foldl* and *foldr* or.... Nor is this restricted to lists: what about trees, graphs, and so forth? In short, special cases are a bad idea. Let's try to do this right.

29.2 Solution

To do this right, we fall back on an old idea: abstraction. The two length functions are nearly the same except for small differences; that means we should be able to parameterize over the differences, define a procedure once, and instantiate the abstraction as often as necessary. Let's do this one step at a time.

Before we can abstract, we should identify the differences clearly. Here they are, boxed:

```
(define length Num
  (lambda (l : num list) : number
    (cond
      [(num Empty? l) 0]
      [(num Cons? l) (add1 (length Num (num Rest l)))])))

(define length Sym
  (lambda (l : sym list) : number
    (cond
      [(sym Empty? l) 0]
      [(sym Cons? l) (add1 (length Sym (sym Rest l)))])))
```

Because we want only one *length* procedure, we'll drop the suffixes on the two names. We'll also abstract over the num and sym by using the parameter τ, which will stand (of course) for a type:

```
(define length
  (lambda (l : τ list) : number
    (cond
      [(τ Empty? l) 0]
      [(τ Cons? l) (add1 (length (τ Rest l)))])))
```

[1] It had better: if it didn't prevent some programs, it wouldn't catch any errors!

It's cleaner to think of list as a *type constructor*, analogous to how variants define value constructors: that is, list is a constructor in the type language whose argument is a type. We'll use an applicative notation for constructors in keeping with the convention in type theory. This avoids the odd "concatenation" style of writing types that our abstraction process has foisted upon us. This change yields

```
(define length
  (lambda (l : list(τ)) : number
    (cond
      [(τ Empty? l) 0]
      [(τ Cons? l) (add1 (length (τ Rest l)))])))
```

At this point, we're still using concatenation for the list operators; it seems to make more sense to make those also parameters to *Empty* and *Cons*. To keep the syntax less cluttered, we'll write the type argument as a subscript:

```
(define length
  (lambda (l : list(τ)) : number
    (cond
      [(Empty?_τ l) 0]
      [(Cons?_τ l) (add1 (length (Rest_τ l)))])))
```

The resulting procedure declaration says that *length* consumes a list of any type, and returns a single number. For a given type of list, *length* uses the type-specific empty and non-empty list predicates and rest-of-the-list selector.

All this syntactic manipulation is hiding a great flaw, which is that we haven't actually defined τ anywhere! As of now, τ is just a free (type) variable. Without binding it to specific types, we have no way of actually providing different (type) values for τ and thereby instantiating different typed versions of *length*.

Usually, we have a simple technique for eliminating unbound identifiers, which is to bind them using a procedure. This would suggest that we define *length* as follows:

```
(define length
  (lambda (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length (Rest_τ l)))]))))
```

but this is horribly flawed! To wit:

1. The procedure *length* now has the wrong form: instead of consuming a list as an argument, it consumes a value that it will bind to τ, *returning* a procedure that consumes a list as an argument.

2. The program isn't even syntactically valid: there is no designation of argument and return type for the procedure that binds τ![2]

[2] You might wonder why we don't create a new type, call it type, and use this as the type of the type arguments. This is trickier than it seems: is type also a type? What are the consequences of this?

3. The procedure bound to *length* expects one argument which is a *type*. This violates our separation of the static and dynamic: types are supposed to be static, whereas procedure arguments are values, which are dynamic.

So on the one hand, this seems like the right sort of idea—to introduce an abstraction—but on the other hand, we clearly can't do it the way we did above. We'll have to be smarter.

The last complaint above is actually the most significant, both because it is the most insurmountable and because it points the way to a resolution. There's a contradiction here: we *want* to have a type parameter, but we *can't* have the type be a value. So how about we create procedures that bind *types*, and execute these procedures during type checking, not execution time?

As always, name and conquer. We don't want to use **lambda** for these type procedures, because **lambda** already has a well-defined meaning: it creates procedures that evaluate during execution. Instead, we'll introduce a notion of a type-checking-time procedure, denoted by Λ (capital λ). A Λ procedure takes only types as arguments, and its arguments do not have further type annotations. We'll use angles rather than parentheses to denote their body. Thus, we might write the *length* function as follows:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length (Rest_τ l)))]))>)
```

This is a lot better than the previous code fragment, but it's still not quite there. The definition of *length* binds it to a type procedure of one argument, which evaluates to a run-time procedure that consumes a list. Yet *length* is applied in its own body to a list, not to a type.

To remedy this, we'll need to *apply* the type procedure to an argument (type). We'll again use the angle notation to denote application:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?_τ l) 0]
        [(Cons?_τ l) (add1 (length<τ> (Rest_τ l)))]))>)
```

If we're going to apply *length* to τ, we might as well assume *Empty?*, *Cons?* and *Rest* are also type-procedures, and supply τ explicitly through type application rather than through the clandestine subscript currently in use:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?<τ> l) 0]
        [(Cons?<τ> l) (add1 (length<τ> (Rest<τ> l)))]))>)
```

Thus, an expression like (*Rest*<τ> *l*) first applies *Rest* to τ, resulting in an actual *rest* procedure that applies to lists of values of type τ; this procedure consumes *l* as an argument and proceeds as it would in the type-system-free case. In other words, every type-parameterized procedure, such as *Rest* or *length*, is a generator of infinitely many procedures that each operate on specific types. The use of the procedure becomes

(*length*<num> (list 1 2 3))
(*length*<sym> (list 'a 'b 'c))

We call this language *parametrically polymorphic with explicit type parameters*. The term *polymorphism* means "having many forms"; in this case, the polymorphism is induced by the type parameters, where each of our type-parameterized procedures is really a representative of an infinite number of functions that differ only in the type parameter. The "explicitly" comes from the fact that our language forces the programmer to write the Λ's and type application.

29.3 The Type Language

As a result of these ideas, our type language has grown considerably richer. In particular, we now permit *type variables* as part of the type language. These type variables are introduced by type procedures (Λ), and discharged by type applications. How shall we write such types? We may be tempted to write

$$\mathit{length} : \mathit{type} \rightarrow (\mathsf{list}(\mathit{type}) \rightarrow \mathsf{number})$$

but this has two problems: first, it doesn't distinguish between the two kinds of arrows ("type arrows" and "value arrows", corresponding to Λ and **lambda**, respectively), and secondly, it doesn't really make clear which type is which, a problem if there are multiple type parameters:

$$\mathit{map} : \mathit{type}, \mathit{type} \rightarrow \mathsf{list}(\mathit{type}) \times (\mathit{type} \rightarrow \mathit{type}) \rightarrow \mathsf{list}(\mathit{type})$$

Instead, we adopt the following notation:

$$\mathit{length} : \forall\alpha.\ \mathsf{list}(\alpha) \rightarrow \mathsf{number}$$

where it's understood that every $\forall$ parameter is introduced by a type procedure (Λ).[3] Here are the types for a few other well-known polymorphic functions:

$$\mathit{filter} : \forall\alpha.\ \mathsf{list}(\alpha) \times (\alpha \rightarrow \mathsf{boolean}) \rightarrow \mathsf{list}(\alpha)$$
$$\mathit{map} : \forall\alpha,\beta.\ \mathsf{list}(\alpha) \times (\alpha \rightarrow \beta) \rightarrow \mathsf{list}(\beta)$$

The type of *map*, in particular, makes this type notation is superior to our previous proposal: when multiple types are involved, we must give each one a name to distinguish between them.

[3] It's conventional to use the beginning of the Greek alphabet—α, β and so on—as the canonical names of polymorphic types, rather than begin from τ. This has two reasons. First, τ is conventionally a *meta-variable*, whereas α and β are type *variables*. Second, not many people know what Greek letter comes after τ....

29.4 Evaluation Semantics and Efficiency

While we have introduced a convenient *notation*, we haven't entirely clarified its meaning. In particular, it appears that every type function application actually happens during program execution. This seems extremely undesirable for two reasons:

- it'll slow down the program, in comparison to both the typed but non-polymorphic programs (that we wrote at the beginning of the section) and the non-statically-typed version, which Scheme provides;
- it means the types must exist as values at run-time.

Attractive as it may seem to students who see this for the first time, we really *do not* want to permit types to be ordinary values. A type is an abstraction of a value; conceptually, therefore, it does not make any sense for the two to live in the same universe. If the types were not supplied until execution, the type checker not be able to detect errors until program execution time, thereby defeating the most important benefit that types confer.

It is therefore clear that the type procedures must accept arguments and evaluate their bodies before the type checker even begins execution. By that time, if all the type applications are over, it suffices to use the type checker built earlier, since what remains is a language with no type variables remaining. We call the phase that performs these type applications the *type elaborator*.

The problem with any static procedure applications is to ensure they will lead to terminating processes! If they don't, we can't even begin the next phase, which is traditional type checking. In the case of using *length*, the first application (from the procedure use) is on the type num. This in turn inspires a recursive invocation of *length* also on type num. Because this latter procedure application is no different from the initial invocation, *the type expander does not need to perform the application*. (Remember, if the language has no side-effects, computations will return the same result every time. Type application has no side-effects.)

This informal argument suggests that only one pass over the body is necessary. We can formalize this with the following type judgments:

$$\frac{\Gamma \vdash e : \forall \alpha.\tau}{\Gamma \vdash e < \tau' > : \tau[\alpha \leftarrow \tau']}$$

This judgment says that on encountering a type application, we substitute the quantified type with the type argument replacing the type variable. The program source contains only a fixed number of type applications (even if each of these can execute arbitrarily many times), so the type checker performs this application only once. The corresponding rule for a type abstraction is

$$\frac{\Gamma[\alpha] \vdash e : \tau}{\Gamma \vdash < \Lambda\ (\alpha)\ e > : \forall \alpha.\tau}$$

This says that we extend Γ with a binding for the type variable α, but leave the associated type unspecified so it is chosen nondeterministically. If the choice of type actually matters, then the program must not type-check.

Observe that the type expander conceptually creates many monomorphically typed procedures, but we don't really want most of them during execution. Having checked types, it's fine if the length function that

actually runs is essentially the same as Scheme's *length*. This is in fact what most evaluators do. The static type system ensures that the program does not violate types, so the program that runs doesn't need type checks.

29.5 Perspective

Explicit polymorphism seems extremely unwieldy: why would anyone want to program with it? There are two possible reasons. The first is that it's the only mechanism that the language designer gives for introducing parameterized types, which aid in code reuse. The second is that the language includes some additional machinery so you don't have to write all the types every time. In fact, C++ introduces a little of both (though much more of the former), so programmers are, in effect, manually programming with explicit polymorphism virtually every time they use the STL (Standard Template Library). Similarly, the Java 1.5 and C# languages support explicit polymorphism. But we can possibly also do better than foist this notational overhead on the programmer.

Chapter 30

Type Inference

30.1 Inferring Types

We've seen the value of having explicit polymorphism in our language—it lets us write programs that work on may different types of values. Even mainstream languages like C++ and, more recently, Java have recognized the value of this form of parameteric polymorphism, and they have noticed that it complements and is not subsumed by the polymorphism common to object-oriented languages (called *subtype polymorphism*).

Despite its benefits, it's very unwieldy to use explicit parametric polymorphism to write programs such as this:

```
(define length
  <Λ (τ)
    (lambda (l : list(τ)) : number
      (cond
        [(Empty?<τ> l) 0]
        [(Cons?<τ> l) (add1 (length<τ> (Rest<τ> l)))]))>)
```

when we could instead write

```
(define length
  (lambda (l)
    (cond
      [(empty? l) 0]
      [(cons? l) (add1 (length (rest l)))])))
```

As computer scientists, we should ask: Is it possible for a programming environment to convert the latter into the former? That is, can the environment *automatically fill in* the types necessary for the former? This would be the best of both worlds, because the programmer would avoid the trouble of the typing while still getting the benefit of the typing.

While this would be nice, it also seems nearly magical. It seems hard enough for humans to get this right; can a program (the environment) do better? Still, we should not despair too much. We've already seen several instances such as closure creation, garbage collection, and so on, where the language implementation

was able to do a more accurate job than a human could have done, thereby providing a valuable feature while reducing the programmer's burden. Maybe inserting type annotations could be another of those tasks.

Because this is obviously challenging, let's try to make the problem easier. Let's ignore the polymorphism, and just focus on generating types for *monomorphic* programs (i.e., those that don't employ polymorphism). In fact, just to make life really simple, let's just consider a program that operates over numbers, such as factorial.

30.1.1 Example: Factorial

Suppose we're given the following program:

```
(define fact
  (lambda (n)
    (cond
      [(zero? n) 1]
      [true      (* n (fact (sub1 n)))])))
```

We've purposely written `true` instead of **else** for reasons we'll soon see. It should be clear that using `true` doesn't affect the meaning of the program (in general, **else** is just a more readable way of writing `true`).

If we were asked to determine the type of this function and had never seen it before, our reasoning might proceed roughly along these lines. First, we would name each expression:

```
(define fact
  1(lambda (n)
     2(cond
        [3(zero? n) 4 1]
        [5true      6(* n 7(fact 8(sub1 n)))])))
```

We would now reason as follows. We'll use the notation $[\![\cdot]\!]$ to mean the type of the expression within the brackets.

- The type of the expression labeled $\boxed{1}$[1] is clearly a function type (since the expression is a **lambda**). The function's argument type is that of n, and it computes a value with the type of $\boxed{2}$. In other words,

$$[\![\boxed{1}]\!] = [\![n]\!] \rightarrow [\![\boxed{2}]\!]$$

- Because $\boxed{2}$ is a conditional, we want to ensure the following:
  - The first and second conditional expressions evaluate to boolean values. That is, we would like the following to hold:

$$[\![\boxed{3}]\!] = boolean$$

$$[\![\boxed{5}]\!] = boolean$$

[1] We'll need to use this phrase repeatedly, and it's quite a mouthful. Therefore, we will henceforth say "the type of $\boxed{n}$" when we mean "the type of the expression labeled by $\boxed{n}$".

  - We would like both branches of the conditional to evaluate to a value of the same type, so we can assign a meaningful type to the entire conditional expression:

$$[\![\boxed{2}]\!] = [\![\boxed{4}]\!] = [\![\boxed{6}]\!]$$

- What is the type of $\boxed{3}$? We have a constraint on what it can be:

$$[\![\mathit{zero?}]\!] = [\![n]\!] \rightarrow [\![\boxed{3}]\!]$$

  Because we know the type of *zero?*, we know that the right-hand-side of the above equality must be:

$$[\![n]\!] \rightarrow [\![\boxed{3}]\!] = \mathit{number} \rightarrow \mathit{boolean}$$

  which immediately tells us that $[\![n]\!] = \mathit{number}$.

The first response in the **cond** tells us that $[\![\boxed{4}]\!] = \mathit{number}$, which immediately resolves the type of $\boxed{2}$ and determines the type of $\boxed{1}$ in atomic terms. That is, the type of *fact* must be number → number. However, it's worthwhile to continue with this process as an illustration:

- We have a constraint on the type of $\boxed{6}$: it must be the same as the result type of multiplication. Concretely,

$$[\![n]\!] \times [\![\boxed{7}]\!] \rightarrow [\![\boxed{6}]\!] = \mathit{number} \times \mathit{number} \rightarrow \mathit{number}$$

- The type of $\boxed{7}$ must be whatever *fact* returns, while $\boxed{8}$ must be the type that *fact* consumes:

$$[\![\boxed{1}]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$

- Finally, the type of $\boxed{8}$ must be the return type of *sub1*:

$$[\![\mathit{sub1}]\!] = [\![n]\!] \rightarrow [\![\boxed{8}]\!] = \mathit{number} \rightarrow \mathit{number}$$

30.1.2 Example: Numeric-List Length

Now let's look at a second example:

```
(define nlength
  (lambda (l)
    (cond
      [(nempty? l) 0]
      [(ncons? l)  (add1 (nlength (nrest l)))])))
```

First, we annotate it:

```
(define nlength
  1(lambda (l)
    2(cond
      [3(nempty? l) 4 0]
      [5(ncons? l)  6(add1 7(nlength 8(nrest l)))])))
```

We can begin by deriving the following constraints:

$$[\![\boxed{1}]\!] = [\![l]\!] \rightarrow [\![\boxed{2}]\!]$$

$$[\![\boxed{2}]\!] = [\![\boxed{4}]\!] = [\![\boxed{6}]\!]$$

$$[\![\boxed{3}]\!] = [\![\boxed{5}]\!] = \mathit{boolean}$$

Because $\boxed{3}$ and $\boxed{5}$ are each applications, we derive some constraints from them:

$$[\![\mathit{nempty?}]\!] = [\![l]\!] \rightarrow [\![\boxed{3}]\!] = \mathit{numlist} \rightarrow \mathit{boolean}$$

$$[\![\mathit{ncons?}]\!] = [\![l]\!] \rightarrow [\![\boxed{5}]\!] = \mathit{numlist} \rightarrow \mathit{boolean}$$

The first conditional's response is not very interesting:[2]

$$[\![\boxed{4}]\!] = [\![0]\!] = \mathit{number}$$

Finally, we get to the second conditional's response, which yields several constraints:

$$[\![\mathit{add1}]\!] = [\![\boxed{7}]\!] \rightarrow [\![\boxed{6}]\!] = \mathit{number} \rightarrow \mathit{number}$$

$$[\![\boxed{1}]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$

$$[\![\mathit{nrest}]\!] = [\![l]\!] \rightarrow [\![\boxed{8}]\!] = \mathit{numlist} \rightarrow \mathit{numlist}$$

Notice that in the first and third set of constraints above, because the program applies a primitive, we can generate an extra constraint which is the type of the primitive itself. In the second set, because the function is user-defined, we cannot generate any other meaningful constraint just by looking at that one expression.

Solving all these constraints, it's easy to see both that the constraints are compatible with one another, and that each expression receives a monomorphic type. In particular, the type of $\boxed{1}$ is $\mathsf{numlist} \rightarrow \mathsf{number}$, which is therefore the type of *nlength* also (and proves to be compatible with the use of *nlength* in expression $\boxed{7}$).

30.2 Formalizing Constraint Generation

What we've done so far is extremely informal. Let's formalize it.

Constraints relate different portions of the program by determining how they should be compatible for the program to execute without error. Consequently, a single program point may result in multiple constraints. Each set of constraints represents a "wish list" about that particular point in the program. Consequently, a program may lead to contradictory constraints; hopefully we will be able to find these later. One slightly confusing aspect of constraints is that we write them to look like equations, but they reflect

[2] Note that the $\mathsf{0}$ inside the $[\![\cdot]\!]$ is an expression itself, not a number labeling an expression.

what we *hope* will be true, not what we *know* is true. Specifically, they represent what suffices for safe program execution.[3]

For each expression n in the program's abstract syntax tree, we introduce a variable of the form $[\![n]\!]$. That is, if the program has the form (*foo* 1 2), we would want to introduce variables for 1, 2 and (*foo* 1 2). Because abstract syntax tree nodes are unwieldy to write down explicitly, we will associate the node with the expression at that node. We use $[\![\cdot]\!]$ to represent the type of a node, so the types of the expressions in the example above would be $[\![1]\!]$, $[\![2]\!]$ and $[\![(foo\ 1\ 2)]\!]$.

Each expression type generates different constraints. We present below a table that relates the type of expression at a node to the (set of) constraints generated for that node. Remember to always read $[\![\cdot]\!]$ as "the type of the expression" (within the brackets):

Expression at Node	Generated Constraints
n, where n is a numeral	$[\![n]\!] = \texttt{number}$
true	$[\![\mathsf{true}]\!] = \texttt{boolean}$
false	$[\![\mathsf{false}]\!] = \texttt{boolean}$
(*add1 e*)	$[\![(add1\ e)]\!] = \texttt{number} \quad [\![e]\!] = \texttt{number}$
(+ *e1 e2*)	$[\![(+\ e1\ e2)]\!] = \texttt{number} \quad [\![e1]\!] = \texttt{number} \quad [\![e2]\!] = \texttt{number}$
(*zero? e*)	$[\![(zero?\ e)]\!] = \texttt{boolean} \quad [\![e]\!] = \texttt{number}$
(*ncons e1 e2*)	$[\![(ncons\ e1\ e2)]\!] = \texttt{list(num)} \quad [\![e1]\!] = \texttt{number} \quad [\![e2]\!] = \texttt{list(num)}$
(*nfirst e*)	$[\![(nfirst\ e)]\!] = \texttt{number} \quad [\![e]\!] = \texttt{list(num)}$
(*nrest e*)	$[\![(nrest\ e)]\!] = \texttt{list(num)} \quad [\![e]\!] = \texttt{list(num)}$
(*nempty? e*)	$[\![(nempty?\ e)]\!] = \texttt{boolean} \quad [\![e]\!] = \texttt{list(num)}$
nempty	$[\![nempty]\!] = \texttt{list(num)}$
(**if** *c t e*)	$[\![(\textbf{if}\ c\ t\ e)]\!] = [\![t]\!] \quad [\![(\textbf{if}\ c\ t\ e)]\!] = [\![e]\!] \quad [\![c]\!] = \texttt{boolean}$
(**lambda** (*x*) *b*)	$[\![(\textbf{lambda}\ (x)\ b)]\!] = [\![x]\!] \rightarrow [\![b]\!]$
(*f a*)	$[\![f]\!] = [\![a]\!] \rightarrow [\![(f\ a)]\!]$

Notice that in the two prior examples, we did not create new node numbers for those expressions that consisted of just a program identifier; correspondingly, we have not given a rule for identifiers. We *could* have done this, for consistency, but it would have just created more (unnecessary) variables.

Exercise 30.2.1 *What is the complexity of constraint generation?*

Exercise 30.2.2 *Using the expression at the node, rather than the node itself, introduces a subtle ambiguity. Do you see why?*

30.3 Errors

Here's an erroneous program:

[3] We use the term "suffices" advisedly: these constraints are sufficient but not necessary. They may reject some programs that might have run without error had the type system not intervened. This is inherent in the desire to statically approximate dynamic behavior: the Halting Problem is an insurmountable obstacle. An important constraint on good type system design is to maximize the set of legal problems while still not permitting errors: *balancing programmer liberty with execution safety*.

```
(define nlsum
  (lambda (l)
    (cond
      [(nempty? l) 0]
      [(ncons? l)  (+ (nrest l)
                      (nlsum (nrest l)))])))
```

Can you spot the problem?

First, we'll annotate the sub-expressions:

```
(define nlsum
  [1](lambda (l)
    [2](cond
      [[3](nempty? l) [4]0]
      [[5](ncons? l)  [6](+ [7](nrest l)
                            [8](nlsum [9](nrest l)))])))
```

Generating constraints as usual, we get the following (amongst others):

$$[\![\boxed{8}]\!] = number$$

because the function returns a number in both branches of the conditional, and

$$[\![\boxed{9}]\!] = numlist$$

from the type of *nrest*. Consequently, it appears we can infer that the value bound to *nlsum* has the type numlist → number. This is indeed the type we expect for this procedure.

We should not, however, annotate any types before we're generated, examined and resolved *all* the constraints: we must make sure there are *no inconsistencies*. Completing the generation and solution process does, in fact, result in an inconsistency for this program. In particular, we have

$$[\![\boxed{7}]\!] = numlist$$

from the type of *nrest*, while

$$[\![\boxed{7}]\!] = number$$

from the type of +. Indeed, the latter is the type we want: the numlist only materializes because of the faulty use of *nrest*. Had the programmer used *nfirst* instead of *nrest* in the left-hand-side argument to the addition, the entire program would have checked correctly. Instead, the type inference engine must recognize that there is an error resulting from a *type conflict*: the expression (*nrest l*) is expected to have both the type number and the type numlist. Because these are not compatible types, the type "checker" must present the user with this error.

We use quotes around "checker" because it has, in some sense, disappeared. Instead of checking types annotated by the programmer, the type system now tries to fill in the programmer's annotations. If it succeeds, it can do so only by respecting the types of operations, so there is no checking left to be done. Failure to annotate the program *completely and unambiguously* is now the indication of a type error.

30.4 Example: Using First-Class Functions

We will consider one final example of constraint generation, to show that the process scales in the presence of functions as arguments. Consider the following program:

(**define** *nmap*
  [1](**lambda** (*f l*)
    [2](**cond**
      [[3](*nempty? l*) [4]*nempty*]
      [[5](*ncons? l*) [6](*ncons* [7](*f* [8](*nfirst l*))
                    [9](*nmap f* [10](*nrest l*)))]))))

This program generates the following constraints:

$$[\![\boxed{1}]\!] = [\![f]\!] \times [\![l]\!] \rightarrow [\![\boxed{2}]\!]$$

We get the usual constraints about boolean conditional tests and the type equality of the branches (both must be of type numlist due to the first response). From the second response, we derive

$$[\![ncons]\!] = [\![\boxed{7}]\!] \times [\![\boxed{9}]\!] \rightarrow [\![\boxed{6}]\!] = number \times numlist \rightarrow numlist$$

The most interesting constraint is this one:

$$[\![f]\!] = [\![\boxed{8}]\!] \rightarrow [\![\boxed{7}]\!]$$

In other words, we don't need a sophisticated extension to handle first-class functions: the constraint generation phase we described before suffices.

Continuing, we obtain the following three constraints also:

$$[\![nfirst]\!] = [\![l]\!] \rightarrow [\![\boxed{8}]\!] = numlist \rightarrow number$$

$$[\![nmap]\!] = [\![f]\!] \times [\![\boxed{10}]\!] \rightarrow [\![\boxed{9}]\!]$$

$$[\![nrest]\!] = [\![l]\!] \rightarrow [\![\boxed{10}]\!] = numlist \rightarrow numlist$$

Since l is of type numlist, we can substitute and solve to learn that f has type number $\rightarrow$ number. Consequently, *nmap* has type

$$(\mathsf{number} \rightarrow \mathsf{number}) \times \mathsf{numlist} \rightarrow \mathsf{numlist}$$

which is the type we would desire and expect!

30.5 Solving Type Constraints

30.5.1 The Unification Algorithm

To solve type constraints, we turn to a classic algorithm: *unification*. Unification consumes a set of constraints and either

- identifies inconsistencies amongst the constraints, or
- generates a *substitution* that represents the solution of the constraints.

A substitution associates each identifier (for which we are trying to solve) with a constant or another identifier. Our identifiers represent the types of expressions (thus $\boxed{4}$ is a funny notation for an identifier that represents the type of the expression labeled 4). In our universe, inconsistencies indicate type errors, and the constants are terms in the type language (such as number and number $\rightarrow$ boolean).

The unification algorithm is extremely simple. Begin with an empty substitution. Push all the constraints onto a stack. If the stack is empty, return the substitution; otherwise, pop the constraint $X = Y$ off the stack:

1. If X and Y are identical identifiers, do nothing.
2. If X is an identifier, replace all occurrences of X by Y both on the stack and in the substitution, and add $X \mapsto Y$ to the substitution.
3. If Y is an identifier, replace all occurrences of Y by X both on the stack and in the substitution, and add $Y \mapsto X$ to the substitution.
4. If X is of the form $C(X_1, \ldots, X_n)$ for some constructor C,[4] and Y is of the form $C(Y_1, \ldots, Y_n)$ (i.e., it has the same constructor), then push $X_i = Y_i$ for all $1 \leq i \leq n$ onto the stack.
5. Otherwise, X and Y do not unify. Report an error.

Does this this algorithm terminate? On every iteration of the main loop, it pops a constraint off the stack. In some cases, however, we push new constraints on. The *size* of each of these constraints is, however, smaller than the constraint just popped. Therefore, the total number of iterations cannot be greater than the sum of the sizes of the initial constraint set. The stack must therefore eventually become empty.

Exercise 30.5.1 *What are the space and time complexity of this algorithm?*

30.5.2 Example of Unification at Work

Let's consider the following example:

$\boxed{1}$($\boxed{2}$**(lambda** (x) x)
$\boxed{3}$7)

[4] In our type language, the type constructors are $\rightarrow$ and the base types (which are constructors of arity zero). More on this in Section 30.5.3.

This generates the following constraints:

$$[\![\boxed{2}]\!] = [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$$

$$[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$$

$$[\![\boxed{3}]\!] = \mathit{number}$$

The unification algorithm works as follows:

Action	Stack	Substitution
Initialize	$[\![\boxed{2}]\!] = [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{2}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	empty
Step 2	$[\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!] = [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 4	$[\![\boxed{3}]\!] = [\![x]\!]$ $[\![\boxed{1}]\!] = [\![x]\!]$ $[\![\boxed{3}]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![\boxed{3}]\!] \rightarrow [\![\boxed{1}]\!]$
Step 2	$[\![\boxed{1}]\!] = [\![x]\!]$ $[\![x]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![\boxed{1}]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$
Step 2	$[\![x]\!] = \mathit{number}$	$[\![\boxed{2}]\!] \mapsto [\![x]\!] \rightarrow [\![x]\!]$ $[\![\boxed{3}]\!] \mapsto [\![x]\!]$ $[\![\boxed{1}]\!] \mapsto [\![x]\!]$
Step 2	empty	$[\![\boxed{2}]\!] \mapsto \mathit{number} \rightarrow \mathit{number}$ $[\![\boxed{3}]\!] \mapsto \mathit{number}$ $[\![\boxed{1}]\!] \mapsto \mathit{number}$ $[\![x]\!] \mapsto \mathit{number}$

At this point, we have solutions for all the sub-expressions *and* we know that the constraint set is consistent.

Writing these in detail is painstaking, but it's usually easy to simulate this algorithm on paper by just crossing out old values when performing a substitution. Be sure to work through our examples for more practice with unification!

30.5.3 Parameterized Types

In the presentation of unification above, we saw only one type constructor with positive arity: $\rightarrow$. A regular programming language will typically have many more constuctors. A common source of parameterized types is *containers*: lists, trees, queues, stacks, and so forth. For instance, it is common to think of lists as parameterized over their content, thus yielding list(number), list(symbol), list(list(number)), list(number $\rightarrow$ symbol) and so on. Identifying list as one of the type constructors for the unification algorithm suffices for typing *unityped* lists.

30.5.4 The "Occurs" Check

Suppose we generate the following type constraint:

$$list([\![x]\!]) = list(list([\![x]\!]))$$

By Step 4, we should push $[\![x]\!] = \mathsf{list}([\![x]\!])$ onto the stack. Eventually, obeying to Step 2, we will add the mapping $[\![x]\!] \mapsto \mathsf{list}([\![x]\!])$ to the substition but, in the process, attempt to replace all instances of $[\![x]\!]$ in the substitution with the right-hand side, which does not terminate.

This is a familiar problem: we saw it earlier when trying to define substitution in the presence of recursion. Because these are problematic to handle, a traditional unification algorithm checks in Steps 2 and 3 whether the identifier about to be (re)bound in the substitution *occurs* in the term that will take its place. If the identifier does occur, the unifier halts with an error.[5] Otherwise, the algorithm proceeds as before.

Exercise 30.5.2 *Write a program that will generate the above constraint!*

30.6 Underconstrained Systems

We have seen earlier that if the system has too many competing constraints—for instance, forcing an identifier to have both type number and boolean—there can be no satisfying type assignment, so the system should halt with an error. We saw this informally earlier; Step 5 of the unification algorithm confirms that the implementation matches this informal behavior.

But what if the system is *under*-constrained? This is interesting, because some of the program identifiers never get assigned a type! In a procedure such as *map*, for instance:

```
(define (map f l)
  (cond
    [(empty? l) empty]
    [(cons? l)  (cons (f (first l))
                      (map f (rest l)))]))
```

This is an instructive example. Solving the constraints reveals that there is no constraint on the type passed as a parameter to the list type constructor. Working through the steps, we get a type for *map* of this form:

$$(\alpha \rightarrow \beta) \times list(\alpha) \rightarrow list(\beta)$$

where α and β are unconstrained type identifiers. This is the same type we obtained through explicit parametric polymorphism... except that the unification algorithm has found it for us *automatically*!

[5] This is not the only reasonable behavior! It is possible to define *fixed-point types*, which are solutions to the circular constraint equations above. This topic is, however, beyond the scope of this text.

30.7 Principal Types

The type generated by this Hindley-Milner[6] system has a particularly pleasing property: it is a *principal* type. What does that mean? For a term t, consider a type τ. τ is a principal type of t if, for any other type τ' that types t, there exists a substitution (perhaps empty) that, when applied to τ, yields τ'.

There are a few ways of re-phrasing the above:

- The Hindley-Milner type system infers the "most general" type for a term.
- The type generated by the Hindler-Milner type system imposes fewest constraints on the program's behavior. In particular, it imposes constraints necessary for type soundness, but no more.

From a software engineering perspective, this is very attractive: it means a programmer could not possibly annotate a procedure with a more general type than the type inference algorithm would derive. Thus, using the algorithm instead of performing manual annotation will not restrict the reusability of code, and may even increase it (because the programmer's annotation may mistakenly overconstrain the type). Of course, there are other good reasons for manual annotation, such as documentation and readability, so a good programming style will mix annotation and inference judiciously.

[6]Named for Roger Hindley and Robin Milner, who independently discovered this in the late 1960s and early 1970s.

Chapter 31

Implicit Polymorphism

31.1 The Problem

Consider the function

(**lambda** (*x*) *x*)

The type inference engine would infer that this function has type

$$\alpha \rightarrow \alpha$$

(or some other type, modulo renaming the type variable).

Now consider a program of the form (the **let** construct is similar to the `with` in our interpreted language):

```
(let ([id (lambda (x) x)])
  (+ (id 5)
     (id 6)))
```

First we need a type judgment for **let**. Here is a reasonable one: it is exactly what one gets by using the existing rules for functions and applications, since we have consistently defined `with` as an application of an immediate function:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \tau'] \vdash b : \tau}{\Gamma \vdash (\textbf{let}([xv])b) : \tau}$$

Given this judgment, the type variable α in the type inferred for id would unify with the type of 5 (namely, number) at one application and with the type of 6 (also number) at the other application. Since these are consistent, the unification algorithm would conclude that id is being used as if it had the type

$$number \rightarrow number$$

in this program.

Now suppose we use id in a context where we apply it to values of different types. The following program is certainly legal in Scheme:

```
(let ([id (lambda (x) x)])
```

```
(if (id true)
    (id 5)    ;; then
    (id 6)))  ;; else
```

This should be legal even in typed Scheme, because we're returning the same type of value in both branches of the conditional. But what happens when we supply this program to our type system? It infers that id has type $\alpha \to \alpha$. But it then unifies α with the type of each of the arguments. Two of these are the same (number) but the first application is to a value of type boolean. This forces the type inference algorithm to try and unify number with boolean. Since these are distinct base types, type inference fails with a type error!

We definitely do not want this program to be declared erroneous. The problem is not with the program itself, but with the algorithm we employ to infer types. That suggests that we should try to improve the algorithm.

31.2 A Solution

What's the underlying problem in the type algorithm? We infer that id has type $\alpha \to \alpha$; we are then stuck with that type for every use of id. It must therefore be *either* number → number or boolean → boolean—but it can't be both. That is, we cannot use it in a truly polymorphic manner!

This analysis makes clear that the problem has something to do with type variables and their unification. We arrive at a contradiction because α must unify with both number and boolean. But what if α didn't need to do that? What if we didn't use the same type variable every time? Then perhaps we could avoid the problem entirely.

One way to get fresh type variables for each application of id is to *literally substitute* the uses of id with their value. That is, instead of type checking the program above, suppose we were to check the following program:

```
(let ([id (lambda (x) x)])
  (if ((lambda (x) x) true)
      ((lambda (x) x) 5)
      ((lambda (x) x) 6)))
```

We don't want to have to *write* this program, of course, but that's okay: a simple pre-processor can substitute every **let**-bound identifier in the body before type-checking. If we did that, we get a different result from type-checking:

```
(let ([id (lambda (x) x)])
  (if (1 (lambda (x) x) true)
      (2 (lambda (x) x) 5)
      (3 (lambda (x) x) 6)))
```

Each use of id results in a different type; for instance, the id procedure at 1 might have type $\alpha \to \alpha$, 2 might have type $\beta \to \beta$ and 3 might have $\gamma \to \gamma$. Then α could unify with type boolean, β with type number and γ with type number. Because these are distinct type variables, they need not unify with one another.

Each application would succeed, and the entire program would successfully pass the type checker. This in fact corresponds more accurately with what happens during execution, because on the first invocation the identifier x in id holds a value of boolean type, and on the subsequent invocation (in the first branch of the conditional, in this case) it holds a number. The separate type variables accurately reflect this behavior.

31.3 A Better Solution

The solution we've presented above has two problems:

1. It can lead to considerable code size explosion. For instance, consider this program:

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

   Expand it in full. In general, how big can a program grow upon expansion?

2. Since **let** does not permit recursion, consider **letrec** or **local**, the Scheme analog of `rec`. What happens if we substitute code in a recursive definition?

In short, the code substitution solution is not workable, but it does contain the germ of a good idea. We see that what it does is generate *fresh type variables* at every use: this is the essence of the solution. So perhaps we can preserve the essence while dispensing with that particular implementation.

Indeed, we can build further on the intuition we have developed. A closure has only one name for an identifier, but a closure can be used multiple times, even recursively, without confusion. This is because, in effect, each application consistently renames the bound identifier(s) so they cannot be confused across instances. Working backwards, since we want fresh identifiers that cannot be confused across instances, we want to create an analogous *type closure* that we instantiate at every use of a polymorphic function.

We will therefore use a modified rule for typing **let**:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \mathsf{CLOSE}(\tau')] \vdash b : \tau}{\Gamma \vdash (\mathbf{let}([xv])b) : \tau}$$

That is, we bind x to a "closed" type when we check the body. The idea is, whenever we encounter this special type in the body, we instantiate its type variables with fresh type variables:

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau')}{\Gamma \vdash e : \tau}$$

where τ is the same as τ', except all type variables have been renamed consistently to unused type variables.

Returning to the identity procedure, the type inference algorithm binds id to the type $\mathsf{CLOSE}(\alpha \rightarrow \alpha)$. At each use of id, the type checker renames the type variables, generating types such as $\alpha_1 \rightarrow \alpha_1$, $\alpha_2 \rightarrow \alpha_2$, and so on. As we have seen before, these types permit the body to successfully type check. Therefore, we have successfully captured the intuition behind code-copying without the difficulties associated with it.

31.4 Recursion

We have provided a rule for **let** above, but in fact a similar rule can apply to **letrec** also. There are some subtleties that we must defer to a more advanced setting, but safe uses of **letrec** (namely, those where the right-hand side is syntactically a procedure) can safely employ the type closure mechanism described above to infer polymorphic types.[1]

31.5 A Significant Subtlety

Alas something is still rotten in the state of inferring polymorphic types. When we rename all type variables in a CLOSE type, we may rename variables that were not bound in the **let** or **letrec** expression: for instance,

```
(lambda (y)
  (let ([f (lambda (x) y)])
    (if (f true)
        (+ (f true) 5)
        6)))
```

Our algorithm would infer the type $\mathsf{CLOSE}(\alpha \to \beta)$ (or the equivalent under renaming) for f. (Because x and y are not linked in the body, the inference process assigns them potentially different types; hence the presence of both α and β in the type.)

At the first application, in the test of the conditional, we generate fresh type names, α_1 and β_1. The type α_1 unifies with boolean, and β_1 unifies with boolean (since it's used in a conditional context). At the second application, the algorithm generates two fresh names, α_2 and β_2. α_2 will unify with boolean (since that is the type of the argument to f), while β_2 unifies with number, because the entire expression is the first argument to addition. Reasoning thus, we can see that the program successfully passes the type checker.

But this program should fail! Simply looking at it, it's obvious that f can return *either* a boolean or a numeric value, but not both. Indeed, if we apply the entire expression to true, there will be a type error at the addition; if we apply it to 42, the type error will occur at the conditional. Sure enough, in our earlier type systems, it would have failed with an error while unifying the constraints on the return types of f. So how did it slip through?

The program successfully passed the type checker because of our use of type closures. We did not, however, correctly apply our intuition about closures. When we apply a closure, we only get new identifiers for those bound by the closures—not those in its lexical scope. The variables in the closure's lexical scope are shared between all applications of the closure. So should it be in the case of type closures. We should only generate fresh type variables for the types introduced by the **let** or **letrec**.

Concretely, we must modify our rule for **let** (and correspondingly that for **letrec**) so the type closures track which identifiers must be renamed:

$$\frac{\Gamma \vdash v : \tau' \qquad \Gamma[x \leftarrow \mathsf{CLOSE}(\tau', \Gamma)] \vdash b : \tau}{\Gamma \vdash (\mathbf{let}([xv])b) : \tau}$$

[1] In particular, we are no longer using code-copying, which encounters an obvious difficulty in the presence of recursion.

That is, a type closure tracks the environment of closure creation. Correspondingly,

$$\frac{\Gamma \vdash e : \mathsf{CLOSE}(\tau', \Gamma')}{\Gamma \vdash e : \tau}$$

where τ is the same as τ', except the renaming applies only to type variables in τ' that are *not bound* by Γ'.

Applying these rules to the example above, we rename the α's but not β, so the first use of f gets type $\alpha_1 \rightarrow \beta$ and the second use $\alpha_2 \rightarrow \beta$. This forces $\beta = number = boolean$, which results in a type error during unification.

31.6 Why Let and not Lambda?

The kind of polymorphism we have seen above is called *let-based polymorphism*, in honor of the ML programming language, which introduced this concept. Note that **let** in ML is recursive (so it corresponds to Scheme's **letrec** or **local**, and the `rec` we have studied in this class). In particular, ML treats `let` as a primitive construct, rather than expanding it into an immediate function application as Scheme does (and as we did with `with` in our interpreters).

The natural question is to wonder why we would have a rule that makes **let**-bound identifiers polymorphic, but not admit the same polymorphic power for **lambda**-bound identifiers. The reason goes back to our initial approach to polymorphism, which was to substitute the body for the identifier. When we have access to the body, we can successfully perform this substitution, and check for the absence of errors. (Later we saw how type closures achieve the same effect while offering several advantages, but the principle remains the same.)

The last example above shows the danger in generalizing the type of **lambda**-bound identifiers: without knowing what they will actually receive as a value (which we cannot know until run-time), we cannot be sure that they are in fact polymorphic. Because we have to decide at type-checking time whether or not to treat an identifier polymorphically, we are forced to treat them monomorphically, and extend the privilege of polymorphism only to **let**-bound identifiers. Knowing exactly which value will be substituted turns out to be a gigantic advantage for the type system!

31.7 The Structure of ML Programs

While our type inference algorithm inferred types with type variables, we could not actually exploit this power directly. We could use such a value several times in the same type contexts, and the same expression elsewhere several times in a different type context, but not combine the two copies of the code through a binding. Let-based polymorphism earned us this power of abstraction.

Let-based polymorphism depends fundamentally on having access to the bound value when checking the scope of the binding. As a result, an ML program is typically written as a series of **let** expressions; the ML evaluator interprets this as a sequence of nested **let**s. It similarly treats the initial environment as one long sequence of **let**s so, for instance, if a programmer uses *map* in a top-level expression, the evaluator effectively treats the use of *map* as being in the body of the binding of *map*. Therefore, the uses of *map* benefit from the polymorphic nature of that function.

Exercise 31.7.1 *What is the time and space complexity of the polymorphic type inference algorithm that uses type closures?*

31.8 Interaction with Effects

Suppose we add polymorphically-typed boxes to the language:

;; box : alpha ⟶ ref(alpha)
;; unbox : ref(alpha) ⟶ alpha
;; set-box! : ref(alpha) alpha ⟶ ref(alpha)

(We're assuming here that *set-box!* returns the box as its result.) On their own, they look harmless.

Now consider the following program:

```
(let ([f (box (lambda (x) x))])
  (begin
    (set-box! f (lambda (y) (+ y 5)))
    ((unbox f) true)))
```

When run, this program will yield a run-time error because y is bound to the value `true`, then used in an addition. A sound type system should, therefore, flag this program as erroneous.

In fact, however, this program type checks without yielding an error. Notice that f has the closed type *ref* $(\alpha \to \alpha)$ in the empty type environment. This type is renamed at each use, which means the function applied to `true` has type (say) $\alpha_2 \to \alpha_2$, even though the value in the box has been re-bound to `number` $\to$ `number`. In fact, this bug resulting from the unexpected interaction between state and polymorphism lay dormant in ML for many years, and this brief program could crash the system.

What has happened here is that we've destoyed the semantics of boxes. The whole point of introducing the box is to introduce sharing; the implementation of the type system has, however, lost that very sharing.

One solution to this problem would be to prohibit the use of boxes on the right-hand side of **let** (and **letrec**) expressions, or at least not polymorphically generalize them. The problem is actually more general, however: any potential *effect* (such as mutation, continuation capture, and so on) runs into similar problems. Studies of large bodies of ML code have shown that programmers don't in fact need the power of polymorphic generalization for these effects. Therefore, rather than create a vastly more complicated type system, a simple, practical solution is to simply prohibit such effects in locations that the type system will automatically treat as polymorphic.

Part XI

Programming by Searching

Chapter 32

Introduction

While we have already seen a wide variety of languages, in some ways they are all—Scheme, Haskell, and so on—still quite conventional. But we have also seen what a language can do for us grow steadily: laziness gave us the ability to express infinite computations; continuations gave us the ability to better structure interactive applications; garbage collection gave us the ability to pretend we had an infinite amount of memory; and type inference gave us the ability to drop type annotations. Progressing, we now find that we don't even have to view computations as functions. A program defines its data, and then defines relations over those data; the act of *computing* an answer is replaced with the act of *searching* for an answer amongst those data and their interrelationships. The canonical language with this power is Prolog.

Some people colloquiually describe Prolog's implementation by saying it absolves programmers of having to specify algorithms. In an abstract sense, this is true: the programmer only specifies the data, and Prolog searches these exhaustively. In practice, however, this strategy fails for many reasons, mostly having to do with efficiency. Nevertheless, it is helpful to understand Prolog's behavior, because many sub-problems can often benefit from this kind of search.

Chapter 33

Programming in Prolog

33.1 Example: Academic Family Trees

A Prolog program consists of a set of *facts* and a collection of *rules*. Given a program, a user can ask the Prolog evaluator whether a particular fact is true or not. The query may be true atomically (because it's in the set of facts); if it's not, the evaluator needs to apply the rules to determine truth (and, if no collection of rules does the trick, then the query is judged false). With this little, Prolog accomplishes a lot!

Let's plunge into a Prolog program. It's inevitable, in the course of studying Prolog, to encounter a genealogy example. We'll look at the genealogy of a particular, remarkable mathematician, where parenthood is determined by PhD advisors.[1]

We'll first list a few facts:

```
advisor(barwise,feferman).
advisor(feferman,tarski).
advisor(tarski,lesniewski).
advisor(lesniewski,twardowski).
advisor(twardowski,brentano).
advisor(brentano,clemens).
```

All facts are described by name of a relation (here, `advisor`) followed by a tuple of values in the relation. In this case, we will assume that the person in the first position was advised by the person in the second position. Prolog does not ask us to declare relations formally before providing their contents; nor does it provide a means (beyond comments) of formally describing the relationship. Therefore, Prolog programmers must be careful to keep track of how to interpret each relation.

Facts relate *constants*. In the example above, `barwise` and `feferman` are both constants. A constant is, in fact, just a relation of arity zero, but understanding this isn't necessary for the rest of this material.

In this example, all relations (including constants) are named by lower-case initial letters. This is not a coincidence; Prolog requires this. Accidentally writing `Barwise` instead of `barwise` would change the meaning of code quite a bit, because an initial capital denotes a *variable*, which we will study soon. Just keep in mind that the case of the initial letter matters.

[1] Thanks to the Mathematics Genealogy Project at `http://www.genealogy.ams.org/`.

Given this input, we can ask Prolog to confirm a few basic facts:

```
:- advisor(barwise,feferman).
yes
```

This asks Prolog at its prompt whether the constants `barwise` and `feferman` are in the `advisor` relationship. Prolog responds affirmatively. In contrast,

```
:- advisor(feferman,barwise).
no
```

Prolog responds negatively because it has no such fact in the database specified by the program.

So far, Prolog has done nothing interesting at all. But with rules, we can begin to explore this universe.

A standard genealogical question is whether one person is another's *ancestor*. Let's build up this rule one step at a time:

```
ancestor(x,y):-advisor(x,y).
```

This says that `y` is `x`'s (academic) ancestor if `y` is `x`'s advisor. But this isn't very interesting either: it just sets up `ancestor` to be an alias for `advisor`. Just to be sure, however, let's make sure Prolog recognizes it as such:

```
:- ancestor(barwise,feferman).
no
```

What?!? Oh, that's right: `x` and `y` are constants, not variables. This means Prolog currently knows how to relate only the *constants* `x` and `y`, not the constants `barwise` and `feferman`, or indeed any other constants. This isn't what we mean at all! What we should have written is

```
ancestor(X,Y):-advisor(X,Y).
```

Now, sure enough,

```
:- ancestor(barwise,feferman).
yes
```

So far, so good. There is another way for one person to be another's academic ancestor: by transitivity. We can describe this verbally, but it's at least as concise, and just as readable, to do so in Prolog:[2]

```
ancestor(X,Y):-
  advisor(X,Z),
  ancestor(Z,Y).
```

Read the '`,`' as "and", while the multiple definitions for `ancestor` combine with "or" (i.e., each represents a valid way to be an ancestor, so to be in the `ancestor` relation it's enough to satisfy one rule or the other). All Prolog rules are written in this "or of and's" form (Disjunctive Normal Form). Notice the use of `Z` twice on the right-hand side. This is intentional: this is what captures the fact that the *same* person must be both the immediate advisor and herself a descendant.

Armed with this extended rule, we can ask more interesting queries of Prolog:

[2](Careful with those capital letters!)

```
:- advisor(barwise,tarski).
yes
```

so we can confirm that Barwise was advised by a legend. But it's always a good idea to write tests that ensure we didn't write a faulty rule that made the relation *too* large:

```
:- advisor(tarski,barwise).
no
```

By the way, here's an easy kind of mistake to make in Prolog: suppose you write

```
advisor(tawrdowski,brentano).
```

instead of

```
advisor(twardowski,brentano).
```

Then you get

```
:- advisor(barwise,clemens).
no
```

Prolog doesn't have any way of knowing about slight misspellings of Polish names. It accepts your facts as truth; garbage in, garbage out. This is another important pitfall (along with capitalization and making a relation too large) to keep in mind.

Now let's expand the relation with a few more facts. Franz Brentano actually had two advisors, of whom we've given credit to only one above. So we should add the fact

```
advisor(brentano,trendelenburg).
```

We can now ask Prolog

```
:- advisor(barwise,clemens).
yes
:- ancestor(barwise,trendelenburg).
yes
```

and it shows that the relationships Prolog tracks really are *relations*, not functions: the mapping truly can be one-to-many. (We could already have guessed this from the rule for `ancestor`, where we provided multiple ways of determining whether or not a pair of constants was in that relation.)

Now let's add some more rules. The simplest is to ask questions in the other direction:

```
descendant(X,Y):-ancestor(Y,X).
```

As of now, each person has only one immediate descendant. But most of these people produced many students. Tarski, one of the great logicians and mathematicians, not only generated a rich corpus of material, but also trained a string of remarkable students. We already know of one, Feferman. Let's add a few more:

```
advisee(tarski,montague).
advisee(tarski,mostowski).
advisee(tarski,robinson).
```

Now, clearly, there are more ways of being one's descendant. There are two ways of fixing the `descendant` relation. One, which involves a lot of work, is to add more facts and rules. A much easier fix is to note that every advisee relationship subscribes a corresponding advisor relationship:

```
advisor(X,Y):-advisee(Y,X).
```

And sure enough,

```
:- descendant(clemens,montague).
yes
:- descendant(trendelenburg,montague).
yes
:- descendant(feferman,montague).
no
:- descendant(barwise,montague).
no
```

We haven't at all explained how Prolog evaluates, and that's largely because it seems so very intuitive (though once we have multiple clauses, as in `descendant`, it may be a little less than obvious). But then, we also haven't seen Prolog do anything truly superlative. Let's explore some more.

Let's first assume we've removed Trendelenburg from the database, so Brentano has only one advisor. (We can do this in a source file by using C-style comments, delimiting text in `/*` and `*/`.) Then let's ask Prolog the following query:

```
:- ancestor(barwise,X).
```

What does this mean? We know all the parts: `advisor` is a relation we've defined (by both facts and rules); `barwise` is a constant; and `X` is a variable. We should interpret this as a *query*, asking Prolog whether there is a value for `X` that would *satisfy* (make true) this query. In fact, we know there is (`clemens`). But Prolog's response is worth studying. This particular Prolog system[3] prints

```
SOLUTION:
  X=feferman
```

So not only did Prolog establish that the query was valid, it also found a solution for `X`! Now this isn't the solution we expected above, but if you think about it for a moment, it's clear that the query has multiple solutions, and Prolog has picked one of them. In fact, at the bottom of the window (in this interface), Prolog says Press cancel to stop, or continue for more solutions. Clicking on the Continue button provides one more solution, then another, then another, and so on until there are no more, so the final output is

[3]Trinc-Prolog R3. In many textual Prolog systems, it's conventional to print a caret to indicate that another solution is available. The user types a semi-colon to ask Prolog to present it.

```
SOLUTION:
  X=feferman
SOLUTION:
  X=tarski
SOLUTION:
  X=lesniewski
SOLUTION:
  X=twardowski
SOLUTION:
  X=brentano
SOLUTION:
  X=clemens
no
```

Wow! Prolog actually "filled in the blank". In fact, if we put Trendelenburg back into the picture, Prolog prints one more solution:

```
SOLUTION:
  X=trendelenburg
```

We can ask a similar query with the variable in the first position instead:

```
:- ancestor(X,clemens).
SOLUTION:
  X=brentano
SOLUTION:
  X=barwise
SOLUTION:
  X=feferman
SOLUTION:
  X=tarski
SOLUTION:
  X=lesniewski
SOLUTION:
  X=twardowski
```

This shows that Prolog isn't just working as a functional program might, where the last position in the relation is like a "return" location. Prolog really doesn't discriminate between different positions where you might put variables.

Maybe this isn't so surprising. After all, Prolog is merely listing the same chain of relationship that we entered as facts at the top of the program. Actually, this isn't *quite* true: it had to apply the transitive rule of `ancestor` to find *all* the solutions (and these are indeed all of them). But perhaps a more impressive test would be to ask a query that runs counter to the facts we entered. For this, we should employ the `advisee` relation.

```
:- descendant(tarski,X).
SOLUTION:
  X=feferman
SOLUTION:
  X=montague
SOLUTION:
  X=mostowski
SOLUTION:
  X=robinson
SOLUTION:
  X=barwise
```

Sure enough, Prolog produces the entire part of Alfred Tarski's family tree that we've taught it. Notice that to get to Barwise it had to recur through Feferman using `advisor`, and to get to Robinson it had to employ `advisee`.

This is pretty impressive already. But we can take this a step further. Why stop with only one variable? We could in fact ask

```
:- ancestor(X,Y).
```

In response, Prolog will actually compute all pairs in the `ancestor` relation and present them sequentially:

```
SOLUTION:
  X=barwise
  Y=feferman
SOLUTION:
  X=feferman
  Y=tarski
SOLUTION:
  X=tarski
  Y=lesniewski
SOLUTION:
  X=lesniewski
  Y=twardowski
```

and so on and on.

Now let's explore another relation: academic siblings. We can define a sibling pretty easily: they must have a common advisor.

```
sibling(X,Y):-
  advisor(X,Z),
  advisor(Y,Z).
```

We can either ask Prolog to confirm relationships (and just to be sure, try them both ways):

```
:- sibling(robinson,montague).
yes
:- sibling(montague,robinson).
yes
```

or to generate their instances:

```
:- sibling(robinson,X).
SOLUTION:
  X=feferman
SOLUTION:
  X=montague
SOLUTION:
  X=mostowski
SOLUTION:
  X=robinson
```

How's that? When Robinson comes along to find out who her academic siblings are, she finds... herself!

It's not very surprising that we got this output. What we meant to say was that *different* people, `X` and `Y`, are academic siblings if... and so on. While we may have mentally made a note that we expect `X` and `Y` to be different people, we didn't tell Prolog that. And indeed, because Prolog programs can be "run backwards", it's dangerous to not encode such assumptions. Making this assumption explicit is quite easy:[4]

```
sibling(X,Y):-
  advisor(X,Z),
  advisor(Y,Z),
  X \== Y.
```

Now, sure enough, we get the right number of siblings.

33.2 Intermission

At this point, we have seen most of the elements of (the core of) Prolog. We've seen fact declarations and the expression of rules over them to create extended relations. We've also seen that Prolog evaluates programs as simple rule-lookups or as queries (where the former are a special case of the latter). We've seen Prolog's variables, known as *logic variables*, which can take on multiple values over time as Prolog boldly and tirelessly seeks out new solutions until it has exhausted the space of possibilities. And finally, related to this last step, Prolog *backtracks* as necessary to find solutions, in accordance with the *non-determinism* of the rules.

[4]For a very subtle reason, we cannot move the last line earlier. We will understand why better once we've implemented Prolog.

33.3 Example: Encoding Type Judgments

Let's look at another use for Prolog: to encode type judgments. Recall that we had rules of the form

$$\Gamma \vdash e : \tau$$

where some were *axioms* and the others were conditionally-defined judgments. The former we will turn into facts, the latter into rules.

First, we must determine a representation for abstract syntax in Prolog. We don't want to deal with parsing, and we don't actually distinguish between individual values of a type, so we'll assume constants have been turned into an abstract node that hides the actual value. Thus, we use the constant `numConst` to represent all syntactically numeric expressions (i.e., those abstract syntax terms of type *numE*), `boolConst` to represent true and false, and so on.

Given this, we will define a three-place relation, `type`. The first place will be the type environment, represented as a list; the second will be the expression; and the third the type of the expression. (When writing this as a *function* in a traditional language, we might define it as a two-argument function that computes the expression's type. But because Prolog can "run backward", it doesn't have a distinguished "return". Instead, what we normally think of as the "result" is just another tuple in the relation.) Our axioms therefore become:

```
type(_,numConst,num).
type(_,boolConst,bool).
```

The _ represents that we don't care what goes in that position. (We could as well have used a fresh logic variable, but the underscore makes our intent clearer.) That is, no matter what the type environment, numeric constants will always have type `num`.

The easiest judgment to tackle is probably that for conditional. It translates very naturally into:

```
type(TEnv,if(Test,Then,Else),Tau) :-
  type(TEnv,Test,bool),
  type(TEnv,Then,Tau),
  type(TEnv,Else,Tau).
```

Pay close attention to lower- and upper-case initials! Both `type` and `if` are in lower-case: the former represents the type relation, while the latter is the abstract syntax term's constructor (the choice of name is arbitrary). Everything else is a type variable. (Notice, by the way, that Prolog performs pattern-matching on its input, just as we saw for Haskell.)

Given these two facts and one rule for `type`, we can ask Prolog to type-check some programs (where `[]` denotes the empty list):

```
:- type([],boolConst,bool).
yes
:- type([],if(boolConst,numConst,numConst),num).
yes
:- type([],if(boolConst,numConst,boolConst),num).
no
```

The implementation of this rule in your type checkers reflected exactly the semantics we gave: *if* the three conditions in the antecedent were met, *then* the consequent holds. In contrast, because Prolog lets us query relations in any way we please, we can instead *use the same implementation* to ask what the type of an expression is (i.e., make Prolog perform type inference):

```
:- type([],boolConst,T).
T=bool
no

:- type([],if(boolConst,numConst,numConst),T).
T=num
no

:- type([],if(boolConst,numConst,boolConst),T).
no
```

It should be no surprise that Prolog "inferred" a type in the first case, since the use precisely matches the axiom/fact. In the second case, however, Prolog used the rule for conditionals to determine solutions to the type of the first expression and matched these against those for the second, finding the only result. In the third case, since the program does not have a type, Prolog fails to find any solutions.

We can now turn evaluation around by asking Prolog strage questions, such as "What expression have type `num`?"

```
:- type([],T,num).
```

Amazingly enough, Prolog responds with:[5]

```
SOLUTION:
  T=numConst
SOLUTION:
  T=if(boolConst, numConst, numConst)
SOLUTION:
  T=if(boolConst, numConst, if(boolConst, numConst, numConst))
SOLUTION:
  T=if(boolConst, numConst,
                  if(boolConst, numConst,
                                if(boolConst, numConst, numConst)))
```

The output here actually gives us a glimpse into the search order being employed by this implementation (notice that it depth-first expands the else-clause of the conditional).

Next let's deal with identifiers. We've said that the type environment is a list; we'll use a two-place `bind` relation to track what type each identifier is bound to.[6] To look up the type of an identifier in the type environment, therefore:

[5] The output has been indented for readability.

[6] We'll make the simplifying assumption that all bound identifiers in the program are consistently renamed to be distinct.

```
type([bind(V,T)|_],var(V),T).
type([bind(_,_)|TEnvRest],var(V),T):-
  type(TEnvRest,var(V),T).
```

A quick test:

```
:- type([bind(w,bool),bind(v,num)],var(v),T).
T=num
```

Next we'll specify the rule for functions:

```
type(TEnv,fun(Var,Body),arrow(T1,T2)) :-
  type([bind(Var,T1)|TEnv],Body,T2).
```

Testing this:

```
:- type([],fun(x,if(var(x),numConst,boolConst)),T).
no

:- type([],fun(x,if(var(x),numConst,numConst)),T).
T=arrow(bool,num)
```

Notice that in the second example, Prolog has determined that the bound identifier must be a boolean, since it's used in the test expression of a conditional.

Finally, the rule for applications holds no surprises:

```
type(TEnv,app(Fun,Arg),T2) :-
  type(TEnv,Fun,arrow(T1,T2)),
  type(TEnv,Arg,T1).
```

Running it:

```
:- type([],
        app(fun(x,if(var(x),
                  numConst,
                  numConst)),
            boolConst),
        T).
T=num
```

Now let's try some more interesting functions:

```
:- type([],fun(x,var(x)),T).
T=arrow(__2823020, __2823020)
```

This is Prolog's way of saying that parts of the answer are indeterminate, i.e., there are no constraints on it. In short, Prolog is inferring parameterized types!

```
:- type([],
        app(fun(id,
                if(app(var(id),boolConst),
                   app(var(id),boolConst),
                   app(var(id),boolConst))),
            fun(x,var(x))),
        T).
T=bool

:- type([],
        app(fun(id,
                if(app(var(id),boolConst),
                   app(var(id),numConst),
                   app(var(id),numConst))),
            fun(x,var(x))),
        T).
no
```

Finally, we have to try:

```
:- type([], fun(x, app(var(x), var(x))), num).
no

:- type([], fun(x, app(var(x), var(x))), T).
T=arrow(arrow(arrow...
```

Exercise 33.3.1 *Are Prolog's types truly polymorphic? Do they automatically exhibit let-based polymorphism (Section 31)? Write appropriate test expressions and present Prolog's output to justify your case.*

33.4 Final Credits

We've now seen even more of Prolog. We've encountered the "don't care" notation. Prolog computes the most general response it can, so if there are no constraints on some part of the answer, it leaves them undefined (using the same symbol to show sharing constraints, as in the inferred type of the identity function). Prolog will match patterns as deep as they are nested, and programmers can use the same variable twice in a rule to indicate that they intend for the values of both to be the same. (Having already seen this with genealogical trees, we made much more extensive use of it to encode type judgments, mimicking the use of meta-variables when we wrote the judgments on paper.)

Putting together the pieces, we found that Prolog was a very convenient encoding of the rules of a type checker. Indeed, for free, we were able to turn our type *checker* into a type *inference* engine. Thinking about how we implemented type inference manually may give us some clues as to how to implement Prolog!

Chapter 34

Implementing Prolog

A Prolog program is a collection of facts and rules. Against these, we ask Prolog a query, also known as providing a *goal* that it must *satisfy*. If a goal contains logic variables, Prolog must find assignments (if necessary) for those variables consistent with the facts and rules.

It is easy to see that in the typical case, a collection of Prolog facts subscribes a flat relational database. Searching such a database is not especially hard (there are obviously challenges to doing so efficiently, but these concerns are handled well by database management systems; they're outside the scope of our study). Prolog evaluation becomes interesting with the introduction of rules.

34.1 Implementation

As we have seen, Prolog rules are in disjunctive normal form, i.e., an "or of and's". Such formulas naturally subscribe a tree, where the individual nodes in the tree are labeled by a Boolean connective. Given a goal, Prolog can construct a disjunctive tree of the facts and rules that match the head term in the goal. For instance, if the query is

```
:- ancestor(barwise,feferman).
```

Prolog can construct an or-labeled tree of all the ways of being an ancestor (there are two ways, corresponding to two rules and no facts). Prolog then explores each node in turn: each of these is, reasonably enough, called a *sub-goal*. If the sub-goal is a fact, there is nothing further to explore; otherwise it can expand into further sub-goals that must, in turn, each be satisfied. In short, Prolog naturally constructs and then explores a classical *search tree*. Thus we say that programming in Prolog is "programming by searching". More precisely, Prolog's search is non-deterministic: it permits rules to have multiple definitions and searches them all, looking for any—and, indeed, every—way to satisfy it.

Armed with this background, we can now delve into the pieces. To clarify the execution of Prolog, we must provide an account of (at least) the following:

- How does Prolog search the space of relations?
- What exactly constitutes satisfaction?

- What are logic variables? How do logic variables take on a value, and how does this value change over the course of a search?
- What causes computation to terminate?

34.1.1 Searching

In what order should Prolog search the tree? Both of the canonical answers make sense:

breadth-first search This has the benefit of never getting "stuck" in an infinite expansion exploring one sub-goal while a different one is satisfiable and might have led to an answer. Unfortunately, it is also rather expensive to maintain the queue necessary for breadth-first search.

depth-first search This has the disadvantage that it may get "stuck" exploring one non-satisfying disjunct while another can satisfy the query. It has the advantage of executing relatively efficiency because it corresponds well to a stack-based execution, which modern systems are tuned well to implement.

Though breadth-first search will therefore produce satisfying answers in situations when depth-first search will not, Prolog chooses the latter, sacrificing some purity at the altar of efficiency.

Exercise 34.1.1 *Write a Prolog program that would yield responses under a breadth-first order but that fails to terminate in a traditional implementation.*
Hint: *Even when choosing depth-first search, a Prolog system has the choice of choosing sub-goals left-to-right, right-to-left, etc. You can glean insight into the order chosen by the Prolog implementation you're using by examining the order in which it prints output for queries that have many satisfying answers.*

Exercise 34.1.2 *Rewrite your example from the previous exercise to retain as much of its structure as possible, but to provide as many answers as possible in the Prolog system you're using. Discuss what you had to change and its implications for programming by searching.*

34.1.2 Satisfaction

The example we saw in Section 33.1 helps illustrate numerous concepts:

```
sibling(X,Y):-
  advisor(X,Z),
  advisor(Y,Z),
  X \== Y.
```

Given the query

```
sibling(robinson,feferman}.
```

we can pretend Prolog "binds" `X` to `robinson` and `Y` to `feferman`; this therefore appears to make two "recursive calls" to `advisor`. However, these cannot be function invocations in the traditional sense, because `Z` is not bound. In fact, `Z` is—as we have said—a logic variable.

Prolog searches every pair in the `advisor` relation to find values that it can associate with `Z`. None of the facts about `advisor` yield a match, but we also have a rule relating `advisor` to `advisee`. This spawns a fresh goal, trying to satisfy `advisee(Z,robinson)`. This eventually finds a satisfying match, with `Z` bound to `tarski`. Now that Prolog has a value of `Z`, it can proceed with the second clause (recall that `,` should be read as "and", so there is no point proceeding until the first clause has been satisfied).

Now we understand why logic variables are called *variables*: they change their value. Initially `Z` had no value at all, whereas now it holds the value `tarski`. To satisfy the next goal, Prolog must therefore attempt to satisfy `advisor(feferman,tarski)`. This is, of course, a fact about `advisor`, so Prolog immediately satisfies it. Now it proceeds to successfully discharge the inequality test, and return an affirmative answer—that is, Robinson and Feferman are indeed academic siblings.

This account does not yet explain two things:

- How many times can a logic variable vary? Does it change its value just once (going from unbound to bound), or does it change more than once?
- Why did we have to place the inequality test at the end of the rule?

To study these questions, let us consider the more general query,

```
sibling(mostowski,B).
```

Initially, `X` is bound to `mostowski` and `Y` to the logic variable `B`. This yields the sub-goal

```
advisor(mostowski,Z).
```

which eventually binds `Z` to `tarski` (by the reasoning we saw earlier). The first clause having been satisfied, we now effectively evaluate the sub-goal `advisor(B,tarski)` (replacing `Y` with `B`).

What is the value of `B` in this sub-goal? Actually, it is not meaningful to ask for *the* value, because there are several. First, Prolog assigns `B` to `feferman`, because this is the first matching rule in the program. Evaluation now proceeds as before, with the inequality test succeeding. Now Prolog can not only report success, it can also report a value for `B`, namely `feferman`. This is indeed what Prolog prints.

In fact, Prolog has done more than just assign a value to `B`: it has also recorded the last successful satisfying clause. When the user now asks for an additional output, the program *resumes execution* from that clause onward. That is, Prolog has the ability to store the state of a computation (specifically, of a search), and restore it on demand. This sounds remarkably similar to the behavior of a *continuation*.

When the user asks for resumption, Prolog tries the same sub-goal again. It now finds a different student who was advised by Tarski, namely `montague`. Again, we find that the computation can successfully complete, so the user sees `montague` presented as a binding for `B`. Observe that in this process, the same logic variable `B` has gone from having no value at all to being bound to `tarski` to being bound to `montague`. As the user explores more of the computation tree, the (same) logic variable keeps changing its binding.

Eventually, after more prompting from the user, Prolog finds that `mostowski` is also a satisfying assignment for the sub-goal. Program control now returns to the conjunction of clauses in the rule for

`sibling` to evaluate the third clause. When Prolog now performs the comparison, we find that the values bound to the two logic variables are the same so the comparison fails, thereby causing the clause to fail.

This explains why we must perform the check at the end. Until the end, we cannot be sure both the logic variables, `X` and `Y`, will actually be bound, because we cannot know for which parameter—or both—the user will supply a logic variable. Only at the end do they have values. Thus, even though the language appears "logical", there is a clear imperative effect in the binding of values to logic variables, so the programmer must take care to sequence correctly (just as with variables in less peculiar languages).

Having failed the comparison, what does Prolog do next? It does not return to the top-level reporting failure. Instead, it *continues searching* by looking for more assignments to the logic variable. It finds such an assignment: `robinson`. Therefore `B` is bound to `robinson`, an assignment that satisfies the comparison, resulting in the next displayed bound value. Further continuing the computation yields no more successful bindings, thereby causing the program to halt with no more successful satisfying assignments.

For Prolog to resume the computation when a clause (specifically, the inequality test) fails, it must have access to the continuation of the previous satisfying clause. Each clause must also provide the continuation of its own evaluation, which a failing computation can use to resume computation. That means every Prolog expression must *consume a continuation and return one*. It consumes a continuation that dictates where to resume in case of failure, and it returns its continuation to indicate success (to be used as some other computation's resumption point in case of failure).

Computation terminates when no more terms satisfy.

34.1.3 Matching with Logic Variables

Our explanation has focused on matching logic variables against atomic terms, which looks like a simple assignment process. As we saw in Section 33.3, however, we may need to match against terms with a complex structure (such as the environment). Given a goal such as

```
type([bind(V,T)|_],var(V),T).
```

and a rule

```
type([bind(x,num)],var(x),num).
```

how do we perform the matching? Easy—we use *unification*![1] Recall that unification was a process for matching terms with constructors and variables, and it computed a substitution as a result (Section 30.5.1). This substitution is precisely the assignment of values to logic variables. The Prolog notation _ is simply shorthand for introducing a logic variable with a completely fresh name. Assigning logic variables to atomic values such as `feferman` is simply a trivial instance of unification, and terms such as `bind` and `var` are simply more term constructors.

The only difference in Prolog's unification is its interaction with continuations. When unification fails, Prolog must *undo* the assignment of values to logic variables. When it succeeds Prolog leaves the logic variables assigned, but these assignments must be undone before commencing to compute the *next* satisfying assignment for the same variables. Therefore, Prolog undoes the bindings before invoking the failure continuation, and makes the returned success continuation also undo the bindings upon invocation (i.e., when the computation is ready to seek another assignment).

[1] In short, it isn't accidental that Prolog automatically computed precisely the same types as we studied in Section 33.3.

34.2 Subtleties and Compromises

This account of Prolog's implementation hides several subtleties and extensions.

First, as we recall from the discussion of unification, it was possible to have underconstrained variables. What happens to these in Prolog? In fact, we saw precisely these in action, when we asked Prolog to infer (for instance) the type of the identity function. Before displaying them, Prolog consistently renames each unbound logic variable (e.g., `__2823020`) so we can distinguish them from one another.

Exercise 34.2.1 *Write a non-trivial Prolog program whose output contains two distinct unbound logic variables.*

Second, when discussing unification, we also mentioned the occurs check. While it would seem reasonable to use the occurs check whenever possible, it proves to be very computationally expensive. As a result, most Prolog systems forego the occurs check by default, usually providing it as a configurable option. This is another common compromise in Prolog systems.

Third, Prolog users sometimes want to better control either the performance or even the meaning of a computation by terminating a search at some point. Prolog provides an operator written `!`, pronounced "cut", for indicating this. Cuts are another controversial feature of Prolog because they sacrifice linguistic purity for pragmatic reasons.

34.3 Future Directions

Programming by searching is arguably still in its infancy. While the influence of Prolog has waxed and waned, its ideas live on in many interesting control operators, and the notion of specifying connections between data, and leaving it to the system to search for the answer, remains very attractive.

Because the system performs the search, people sometimes colloquially say that in Prolog, the user does not specify the "algorithm". This is a very misleading remark. Prolog programmers exploit the built-in exhaustive search algorithm. A user might therefore want a more sophisticated search, for instance one that exploited knowledge about numbers. This is an active area of research that has grown into the field of *constraint programming*.

There are also ways to ameliorate the ill-effects of the depth-first search strategy. In many cases, the sub-goal in the depth-first search repeats. In such cases, in the absence of other side-effects, repeating the sub-goal is not going to generate a different answer. In such cases, an implementation can use *memoization* to check whether it has (in the appropriate context) issued a sub-goal query before and, if it has, terminate the search as unsatisfiable (i.e., a failure). In such cases, the programmer gets the performance benefit of depth-first search without the danger of non-termination. Prolog systems call this memoization procedure *tabling*. Because it incurs some cost, the programmer is asked to stipulate which rules must be tabled. Tabled Prolog extends the expressive elegance of Prolog to many interesting cases such as certain kinds of fixed-point computations.

Part XII

Domain-Specific Languages and Metaprogramming

Chapter 35

Domain-Specific Languages

35.1 Language Design Variables

Programming languages differ in numerous ways:

1. Each uses rather different notations for writing down programs. As we've observed, however, syntax is only partially interesting. (This is, however, less true of languages that are trying to mirror the notation of a particular domain.)

2. Control constructs: for instance, early languages didn't even support recursion, while most modern languages still don't have continuations.

3. The kinds of data they support. Indeed, sophisticated languages like Scheme blur the distinction between control and data by making fragments of control into data values (such as first-class functions and continuations).

4. The means of organizing programs: do they have functions, modules, classes, . . . ?

5. Automation such as memory management, run-time safety checks, and so on.

Each of these items suggests natural questions to ask when you design your own languages in particular domains.

35.2 Languages as Abstractions

Languages are *abstractions*: ways of seeing or organizing the world according to certain patterns, so that a task becomes easier to carry out. More concretely, think about a loop in Java. When you write a loop, you expect the machine to carry out certain tasks for you automatically: testing the termination condition, running the loop code if the test passes, exiting the loop if it doesn't, etc. The loop is an abstraction: a reusable pattern where the language executes part of the pattern automatically, and you supply the parts that are different. You *could* write down all of those steps manually, but then your program would be longer, harder to read, and more painful to write, debug and maintain. Scheme's *map* and *filter* are also abstractions,

which differ from Java's loops in one significant way: you can define Scheme's loops as abstractions in user programs.

35.3 Domain-Specific Languages

Based on the above description, it becomes clear that some domains may be served better by programming in a language specialized to that domain. While we are familiar with such languages (often bundled with software packages that blur the boundary between the package and the language) such as Mathematica and Matlab, this principle is not new. Indeed, study the names of four of the oldest popular programming languages, and you spot a pattern:

Fortran Stands for "formula translator".

Algol An "algorithmic language".

COBOL An abbreviation for "COmmon Business-Oriented Language".

LISP Short for a "list-processing" language.

Notice the heavy emphasis on very concrete domains (or, in the case of LISP, of a language construct)? Indeed, it was not until the late 1960s and 1970s that programming languages really became liberated from their domains, and the era of general-purpose languages (GPL) began. Now that we know so much about the principles of such languages (as we've been seeing all semester long), it is not surprising that language designers are shifting their sights back to particular domains.

Indeed, I maintain that designing GPLs has become such a specialized task—well, at least designing *good* GPLs, without making too many mistakes along the way—that most lay efforts are fraught with peril. In contrast, most people entering the programming workforce are going to find a need to build languages specific to the domains they find themselves working in, be they biology, finance or the visual arts. Indeed, I expect many of you will build one or more "little languages" in your careers.

Before you rush out to design a domain-specific language (DSL), however, you need to understand some principles that govern their design. Here is my attempt to describe them. These are somewhat abstract; they will become clearer as we study the example that follows in more detail.

First and foremost—define the domain! If your audience doesn't understand what the domain is, or (this is subtly different) why programming for this domain is difficult, they're not going to pay attention to your language.

Justify why your language should exist in terms of the current linguistic terrain. In particular, be sure to explain why your language is better than simply using the most expressive GPLs around. (Small improvements are insufficient, compared with the odds that the considerably greater resources that are probably going into language implementation, library support, documentation, tutorials and so on for that GPL compared with your language.) In short, be very clear on what your DSL will do that is very difficult in GPLs. These reasons usually take on one or more of the following forms:

- Notational convenience, usually by providing a syntax that is close to established norms in the domain but far removed from the syntax of GPLs. (But before you get too wrapped up in fancy visual notations,

keep in mind that programs are not only written but also edited; how good is your editor compared with `vi` or Emacs?)

- Much better performance because the DSL implementation knows something about the domain. For instance, some toolkits take limited kinds of programs but will, in return, automatically compute the derivative or integral of a function—a very useful activity in many kinds of high-performance scientific computing.

- A non-standard semantics: for instance, when neither eager nor lazy evaluation is appropriate.

There are generally two kinds of DSLs, which I refer to as "enveloping" and "embedded". Enveloping languages are those that try to control other programs, treating them as components. Good examples are shell languages, and early uses of languages like Perl.

Enveloping languages work very well when used for simple tasks: imagine the complexity of spawning processes and chaining ports compared with writing a simple shell directive like `ls -l | sort | uniq`. However, they must provide enough abstraction capabilities to express a wide variety of controls, which in turn brings data structures through the back door (since a language with just functions but without, say, lists and queues, requires unreasonable encodings through the lambda calculus). Indeed, invariably programmers will want mapping and filtering constructs. The net result is that such languages often begin simple, but grow in an unwieldy way (responding to localized demands rather than proactively conducting global analysis).

One way to improve the power of an enveloping language without trying to grow it in an ad hoc way is to embed another language inside it. That is, the enveloping language provides basic functionality, but when you want something more powerful, you can escape to a more complete (or another domain-specific) language. For instance, the language of Makefiles has this property: the Makefile language has very limited power (mainly, the ability to determine whether files are up-to-date and, if not, run some set of commands), and purposely does not try to grow much richer (though some variants of `make` do try). Instead, the actual commands can be written in any language, typically Unix shell, so the `make` command only needs to know how to invoke the command language; it does not itself need to implement that language.

The other kinds of languages are embedded in an application, and expose part of the application's functionality to a programmer who wants to customize it. A canonical example is Emacs Lisp: Emacs functions as a stand-alone application without it, but it exposes some (most) of its state through Emacs Lisp, so a programmer can customize the editor in impressive ways. Another example may be the command language of the `sendmail` utility, which lets a programmer describe rewriting rules and custom mail handlers.

Any time one language is embedded inside another *language* (as opposed to an application), there are some problems with this seemingly happy symbiosis:

1. The plainest, but often most vexing, is syntactic. Languages that have different syntaxes often don't nest within one another very nicely (imagine embedding an infix language inside Scheme, or XML within Java). While the enveloping language may have been defined to have a simple syntax, the act of escaping into another language can significantly complicate parsing.

2. Can the embedded language access values from the language that encloses it? For example, if you embed an XML path language inside Java, can the embedded language access Java variables? And

even if it could, what would that mean if the languages treat the same kinds of values very differently? (For instance, if you embed an eager language inside a lazy one, what are the strictness points?)

3. Often, the DSL is able to make guarantees of performance only because it restricts its language in some significant way. (One interesting example we have seen is the simply-typed lambda calculus which, by imposing the restriction of annotations in its type language, is able to deliver unto us the promise of termination.) If the DSL embeds some other language, then the analysis may become impossible, because the analyzer doesn't understand the embedded language. In particular, the guarantees may not longer hold!

In general, as a DSL developer, be sure to map out a growth route. Anticipate growth and have a concrete plan for how you will handle it. No DSL designer ever went wrong predicting that her programmers might someday want (say) closures, and many a designer did go wrong by being sure his programmers wouldn't. Don't fall for this same trap. At the very least, think about all the features you have seen in this course and have good reasons for rejecting them.

You should, of course, have thought at the very outset about the relationship between your DSL and GPLs. It doesn't hurt for you to think about it again. Will your language grow into a GPL? And if so, would you be better off leveraging the GPL by just turning your language into a library? Some languages even come with convenient ways of creating little extension languages (as we will see shortly), which has the benefit that you can re-use all the effort already being poured into the GPL.

In short, the single most important concept to understand about your DSL is its *negative space*. Language designers, not surprisingly, invariably have a tendency to think mostly about what *is* there. But when you're defining a DSL remember that perhaps the most important part of it is what *isn't* there. Having a clear definition of your language's negative space will help you with the design; indeed, it is virtually a prerequisite for the design process. It's usually a lot easier to argue about what shouldn't (and should) be in the negative space than to contemplate what goes in. And to someone studying your language for the first time, a clear definition of the negative space will greatly help understand your rationale for building it, and perhaps even how you built it—all of which is very helpful for deciding whether or not one finds this the right language for the task, both now and in the future.

Chapter 36

Macros as Compilers

For these notes, please use the PRETTY BIG language level.

36.1 Language Reuse

We have so far implemented languages as interpreters. In the real world, however, programming languages are defined not only by their implementation but also by their toolkit: think of the times you've disliked programming in a language because you didn't like the default editor or the debugger or the lack of a debugger or .... Therefore, when we set out to implement a fresh language implementation, we run the risk that we'll upset our users if we don't provide all the programming tools they're already accustomed to.

One way around this is to not create an entire implementation from scratch. Instead, we could just *compile* the new language into an existing language. If we do that, we can be fairly sure of reusing most of the tools built for the existing language. There is one problem, which is that feedback such as error messages may not make too much sense to the programmer (since she is expecting messages in terms of the constructs of the DSL, while the messages are in terms of the constructs of the target language). This is a real concern, but it is ameliorated some by the tools we will use.

Many languages provide a syntactic preprocessor that translates terms before handing them off to the evaluator. In languages like C and Scheme they're called *macros*, while in C++ they're called *templates*. We will now study the Scheme macro system in some depth. By default, the Scheme macro system permits programmers to add constructs to Scheme, thereby effectively providing a compiler from Scheme+ (the extended Scheme language) to Scheme itself.

36.1.1 Example: Measuring Time

Suppose we want to add a construct to the language that measures the time elapsed while evaluating an expression. That is, we want (**my-time** *e*) which returns the time it took (in milliseconds, say) to evaluate *e*. (The actual *time* command in Scheme also returns the value, but this version suffices for now. We'll use **my-time** for our attempts to avoid clashing with the vesion built-in.)

This is easy; here's the code:

(**define** (**my-time** *e*)

```
(let ([begin-time (current-milliseconds)])
  (begin
    e
    (- (current-milliseconds) begin-time))))
```

Let's test it:

```
> (my-time (+ 1 2))
0
```

Good; that's about what we'd expect. Even for slightly more computationally expensive expressions, we get

```
> (my-time (expt 2 1000))
0
```

Well, that's because DrScheme is really fast, see. How about:

```
> (my-time (expt 2 10000))
0
```

Hmm. Zero *milliseconds*? Maybe not. So let's try

```
> (my-time (expt 2 1000000))
0
```

This time DrScheme noticeably gives pause—we can tell from a wristwatch—so something is afoot.

The problem is that we defined **my-time** to be a procedure, and Scheme is an eager language. Therefore, the entire expression reduced to a value before the body of **my-time** began to evaluate. As a result, the difference in time was always going to be a constant. On different machines we might get different values, but the value isn't going to change, no matter what the expression!

How do we define **my-time**? There are three options.

First would be to introduce lazy evaluation into the language. This may be tempting, but it's going to make a mess overall, because it'd be impossible to determine when an expression is going to reduce, and an expression that has already been reduced to a value may need to have not been reduced later. This is not a viable solution.

The second is to make **my-time** take a thunk (recall: a procedure of no arguments). That is, we would have

```
(define (my-time e-thunk)
  (let ([begin-time (current-milliseconds)])
    (begin
      (e-thunk)
      (- (current-milliseconds) begin-time))))
```

so that

```
> (my-time (lambda () (expt 2 10000)))
0
> (my-time (lambda () (expt 2 100000)))
```

```
60
> (my-time (lambda () (expt 2 1000000)))
2023
```

This may be sufficient, but it's certainly not satisfactory: we've introduced an unnecessary syntactic pattern into the code for which we have no explanation other than that's just what the language demands. This is not an acceptable abstraction.

Finally, another is to accomplish the effect of textual substitution by using…textual substitution. In Scheme, we can instead write

```
(define-syntax my-time
  (syntax-rules ()
    [(my-time e)
     (let ([begin-time (current-milliseconds)])
       (begin
         e
         (- (current-milliseconds) begin-time)))]))
```

When we test this, we find

```
> (my-time (expt 2 1000))
0
```

Hmm! But ever hopeful:

```
> (my-time (expt 2 10000))
10
> (my-time (expt 2 100000))
70
> (my-time (expt 2 1000000))
2053
```

which is what we expect.

How does this version of **my-time** work? The Scheme macro system trawls the program source and gathers all the syntax definitions. It then substitutes all the uses of these syntax definitions with the bodies, where each syntax definition is defined by pattern-matching (we'll see several more examples). Only after finishing all the substitution does it hand the program to the Scheme evaluator, which therefore doesn't need to know anything about the syntax definitions. That is, given the above syntax definition and the program (**my-time** (*expt* 2 10000)), the program that the Scheme evaluator actually sees is

```
(let ([begin-time (current-milliseconds)])
  (begin
    (expt 2 10000)
    (- (current-milliseconds) begin-time)))
```

This is the right-hand-side of the first (and only) clause in the list of rules, except *e* has been substituted with the exponentiation expression. This is now an ordinary Scheme expression that the evaluator can reduce to a

value. Notice that the current time is now measured before and after the expression evaluates, thus ensuring that we do in fact clock its evaluation.[1]

Exercise 36.1.1 *Do macros merely implement laziness?*

36.1.2 Example: Local Definitions

We saw earlier this semester that

```
{with {var val} body}
```

could be rewritten as

```
{{fun {var} body} val}
```

by a preprocessor, so our core evaluator did not need to implement `with` directly. The same is true of the **let** construct in Scheme. Here's a simple macro for **let** (again, we'll use the **my-** convention to avoid any clashes):

```
(define-syntax my-let-1
  (syntax-rules ()
    [(my-let-1 (var val) body)
     ((lambda (var) body) val)]))
```

Sure enough,

```
> (my-let-1 (x 3) (+ x 4))
7
> (my-let-1 (x 3) (my-let-1 (y 4) (+ x y)))
7
```

In full Scheme, however, the **let** construct is a bit more complex: it permits binding several identifiers at the same time (as we saw in a homework assignment regarding `with`). Therefore, the true translation should be regarded as something along these lines:

(**let** ([*var val*] ...) *body*) $\Longrightarrow$ ((**lambda** (*var* ...) *body*) *val* ...)

That is, we want each of the variables to remain in the same order, and likewise each of the value expressions—except we don't know how many we will encounter, so we use ... to indicate "zero or more".

How are we going to define this macro? In fact, it couldn't be easier. A researcher, Eugene Kohlbecker, observed that numerous extensions to Scheme had this same "zero or more" form, and noticed that people always wrote them informally using the stylized notation above. He therefore simply defined a macro system that processed that notation:

```
(define-syntax my-let
  (syntax-rules ()
    [(my-let ([var val] ⋯) body)
     ((lambda (var ⋯) body) val ⋯)]))
```

[1] Technically, this isn't exactly the expression that evaluates. We'll return to this in a bit.

Therefore (**my-let** ([*x* 3] [*y* 4]) (+ *x y*)) translates into ((**lambda** (*x y*) *body*) 3 4) which, sure enough, reduces to 7. Notice how the macro system is smart enough to treat the ([*var val*] ⋯) pattern as being the composite of the *var* ⋯ and *val* ⋯ patterns.[2] In particular, if no identifiers are bound, then this turns into an immediate application of a thunk to no arguments, which just evaluates the body.

36.1.3 Example: Nested Local Definitions

In a **let**, all the named expressions are bound in the same scope, which doesn't include any of the bound names. Sometimes, it's useful to bind names sequentially so later bindings can refer to earlier ones. Scheme provides the construct **let**∗ for this task:

```
(let* ([a 5]
       [b 12]
       [a^2 (* a a)]
       [b^2 (* b b)]
       [a^2+b^2 (+ a^2 b^2)])
  (sqrt a^2+b^2))
```

(Think of what this would evaluate to with **let** instead of **let**∗.)

We can implement **let**∗ very easily by unraveling it into a sequence of **let**s:

$$(\textbf{let}* ([var\ val] \cdots)\ body) \Longrightarrow (\textbf{let}\ ([var_0\ val_0])\ (\textbf{let}\ ([var_1\ val_1])\ \cdots\ (\textbf{let}\ ([var_n\ val_n])\ body)))$$

There is a stylized way of writing such macros in Scheme, which is to split them into two cases: when the sequence is empty and when the sequence has one or more elements. When there are no identifiers being bound, then **let**∗ does the same thing as **let** (which is to reduce to the expression itself):

$$(\textbf{let}* ()\ body) \Longrightarrow body$$

Since each ⋯ means "zero or more", we need to use a more refined pattern to indicate "one or more":

(**let**∗ ([*var0 val0*] [*var-rest val-rest*] ⋯) *body*)

The rewrite rule then becomes

$$(\textbf{let}* ([var0\ val0]\ [var\text{-}rest\ val\text{-}rest] \cdots)\ body) \Longrightarrow (\textbf{let}\ ([var0\ val0])\ (\textbf{let}* ([var\text{-}rest\ val\text{-}rest]\ \vdots)\ body))$$

That is, we apply the macro for **let**∗ recursively. Written in Scheme syntax, this is expressed as (notice the two cases):

[2] The use of brackets versus parentheses is purely stylistic.

```
(define-syntax my-let*
  (syntax-rules ()
    [(my-let* () body)
     body]
    [(my-let* ([var0 val0]
               [var-rest val-rest] ...)
       body)
     (let ([var0 val0])
       (my-let* ([var-rest val-rest] ...)
         body))]))
```

There is nothing in Scheme that prevents a runaway expansion. Therefore, it's possible to write a misbehaving macro that expands forever, so that evaluation never even begins. However, most macros follow the simple stylistic pattern above, which guarantees termination (the recursion is over the bound identifiers, and each time through, one more identifier-value pair is taken off).

36.1.4 Example: Simple Conditional

Let's say we want a simplified form of conditional that has only two branches and one conditional. This is effectively the same as **if**:

(**cond2** [*t e1*] [*else e2*]) $\Longrightarrow$ (**if** *t e1 e2*)

We might try the following macro:

```
(define-syntax cond2
  (syntax-rules ()
    [(cond2 (t e1) (else e2))
     (if t e1 e2)]))
```

This correctly evaluates expressions such as

```
(cond2 [(even? (current-seconds)) 'even]
       [else                      'odd])
```

Unfortunately, this also permits expressions such as

```
(cond2 [(even? (current-seconds)) 'even]
       [(odd? (current-seconds))  'odd])
```

This shouldn't be syntactically legal, because **cond2** permits only one conditional; in place of the second, we require programmers to write **else**. We can see that this second expression doesn't get evaluated at all by writing something atrocious:

```
(cond2 [false  'even]
       [(/ 1 0) 'odd])
```

which evaluates to 'odd.

What we want is for the **cond2** macro to simply reject any uses that don't have **else** in the second question position. This is where the mystical () after **syntax-rules** comes in: it lists the *keywords* in the macro. That is, we should instead define the macro as

```
(define-syntax cond2
  (syntax-rules (else)
    [(cond2 (t e1) (else e2))
     (if t e1 e2)]))
```

Then, we get the following interaction:

```
> (cond2 [false  'even]
         [(/ 1 0) 'odd])
cond2: bad syntax in: (cond2 (false (quote even)) ((/ 1 0) (quote odd)))}
```

Without the keyword designation, Scheme has no way of knowing that **else** has a special status; naturally, it makes no sense to build that knowledge into the macro system. Absent such knowledge, it simply treats *else* as a macro variable, and matches it against whatever term is in that position. When we put **else** in the keyword list, however, the expander no longer binds it but rather expects to find it in the right position—or else rejects the program.

36.1.5 Example: Disjunction

Let's consider one more example from Scheme lore. In Scheme, conditionals like **or** and **and** *short-circuit*: that is, when they reach a term whose value determines the result of the expression, they do not evaluate the subsequent terms. Let's try to implement **or**.

To begin with, let's define the two-arm version of **or**:

```
(define (my-or2-fun e1 e2)
  (if e1
      e1
      e2))
```

Sure enough, a very simple example appears to work

```
> (my-or2-fun false true)
#t
```

but it fails on a more complex example:

```
> (let ([x 0])
    (my-or2-fun (zero? x)
                (zero? (/ 1 x))))
/: division by zero
```

whereas a short-circuiting evaluator would not have permitted the error to occur. The problem is, once again, Scheme's eager evaluation regime, which performs the divison before it ever gets to the body of *my-or2-fun*. In contrast, a macro does not have this problem:

```
(define-syntax my-or2
  (syntax-rules ()
    [(my-or2 e1 e2)
     (if e1 e1 e2)]))
```

which yields

```
> (my-or2 false true)
#t
> (let ([x 0])
    (my-or2 (zero? x)
            (zero? (/ 1 x))))
#t
```

In particular, the second expression translates into

```
(let ([x 0])
  (if (zero? x)
      (zero? x)
      (zero? (/ 1 x))))
```

(just replace *e1* and *e2* consistently).

As this expansion begins to demonstrate, however, this is an unsatisfying macro. We evaluate the first expression twice, which has the potential to be inefficient but also downright wrong. (Suppose the first expression were to output something; then we'd see the output twice. If the expression wrote a value into a database and returned a code, executing it a second time may produce a different result than the first time.) Therefore, we'd really like to hold on to the value of the first evaluation and return it directly if it's not false. That is, we want

```
(define-syntax my-or2
  (syntax-rules ()
    [(my-or2 e1 e2)
     (let ([result e1])
       (if result
           result
           e2))]))
```

This expands the second expression into

```
(let ([x 0])
  (let ([result (zero? x)])
    (if result
        result
        (zero? (/ 1 x)))))
```

Since Scheme is eager, the expression in the *e1* position gets evaluated only once. You should construct test cases that demonstrate this.

36.2 Hygiene

Now what if the use of **my-or2** really looked like this?

```
(let ([result true])
     (my-or2 false
             result))
```

which should evaluate to true. The expansion, however, is

```
(let ([result true])
  (let ([result false])
    (if result
        result
        result)))
```

which evaluates to false!

What happened here? When we look at just the input expression, we do not see only one binding of *result*. Reasoning locally to that expression, we assume that **my-or2** will evaluate the first expression and, finding it false, will evaluate the second; since this is *result*, which is bound to true, the overall response should also be true. Instead, however, the use of *result* within the macro definition interferes with *result* in the context of its use, resulting in the incorrect result.

The problem we see here should seem awfully familiar: this is exactly the same problem we saw under a different guise when trying to understand scope. Here, *result* in the second arm of the disjunction is bound in the **let** just outside the disjunction. In contrast, *result* inside the macro is bound inside the macro. We as programmers should not need to know about all the names used within macros—*just as we don't need to know the names of identifiers used within functions!* Therefore, macros should be forced to obey the scoping rules of the language.

Just to be sure, let's try this expression in our evaluator:

```
> (let ([result true])
    (my-or2 false result))
#t
```

We get true! This is because Scheme's macro system is *hygienic*. That is, it automatically renames identifiers to avoid accidental name clashes. The expression that actually evaluates is something like

```
(let ([result true])
  (let ([g1729 false])
    (if g1729
        g1729
        result)))
```

where *g1729* is a uniquely-generated identifier name. Notice that only the *result*s within the macro definition get renamed. In fact, because **let** is itself a macro, its identifiers also get renamed (as do those introduced by **lambda** and other binding forms), so the real program sent to the evaluator might well be

```
(let ([g4104 true])
```

```
(let ([g1729 false])
  (if g1729
      g1729
      g4104)))
```

Many macro systems, such as that of C, are not hygienic. Programmers sometimes try to circumvent this by using hideous identifier names, such as `__macro_result_`. *This is not a solution!*

1. Not only is a it painful to have to program this way, small typos would greatly increase development time, and the macro would be much harder to decipher when a programmer tries to modify or correct it later.

2. This solution is only as good as the programmer's imagination; the problem still persists, lying in wait for just the right (wrong!) identifier to be bound in the context of use. Indeed, while a programmer may choose a sufficiently obscure name from the perspective of other programmers, not all source is written by humans. A tool generating C code (such as a Scheme-to-C compiler) may happen to use exactly this naming convention.

3. This name is only obscure "upto one level". If the macro definition is recursive, then recursive instances of the macro may interfere with one another.

4. If you use this macro to debug the source that contains the macro (e.g., compiling the C compiler using itself), then your carefully-chosen "obscure" name is now *guaranteed* to clash!

In short, to return to a theme of this course: we should view these kinds of contortions by programmers as a symptom of a problem that must be addressed by better language design. Don't settle for mediocrity! In this case, hygiene is that solution.[3]

Notice, by the way, that we needed hygiene for the proper execution of our very first macro, because **my-time** introduced the identifier *begin-time*. At the time, we never even gave a thought to this identifier, which means in the absence of hygiene, we had a disaster waiting to happen. With hygiene, we can program using normal names (like *begin-time* and *result*) and not have to worry about the consequencs down the line, just as with static scope we can use reasonable names for local identifiers.

36.3 More Macrology by Example

Many languages provide a looping construct for iterating through integers sequentially. Scheme doesn't for three reasons:

1. Because most such loops are anyway inappropriate: the indices only exist to traverse sequential data structures. Uses of *map* or *filter* over a list accomplish the same thing but at a higher level of abstraction.

2. Because recursion in the presence of tail calls has the same computational effect.

[3]The algorithm, in effect, "paints" each expression on expansion, then consistently renames identifiers that have the same paints.

3. Because, if we really crave a more traditional syntax, we can define it using a macro!

We'll build up a loop macro in three stages.

36.3.1 Loops with Named Iteration Identifiers

Here's our first attempt at a **for** loop macro.[4] We've generously embellished it with keywords to increase readability:

```
(define-syntax for0
  (syntax-rules (from to in)
    [(for0 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (local ([define loop (lambda (⟨var⟩)
                            (if (> ⟨var⟩ ⟨high⟩)
                                'done
                                (begin
                                  ⟨bodies⟩ ···
                                  (loop (+ ⟨var⟩ 1)))))])
       (loop ⟨low⟩))]))
```

This lets us write programs such as

```
(for0 x
      from 2
      to 5
      in (display x))
```

which prints 2, 3, 4 and 5. However, when we try this on a program like this

```
(for0 x
      from 2
      to (read)
      in (display x))
```

we notice an unpleasant phenomenon: the program reads the upper-bound of the loop *every time through the loop*. To correct it, we should make sure it evaluates the upper-bound expression only once, which we can do with a small change to the macro:

```
(define-syntax for1
  (syntax-rules (from to in)
    [(for1 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (local ([define high-value ⟨high⟩]
             [define loop (lambda (⟨var⟩)
                            (if (> ⟨var⟩ high-value)
                                'done
                                (begin
```

[4] We're using the convention of wrapping macro pattern-variables in $\langle\cdot\rangle$ to emphasize their relationship to BNF.

```
                    ⟨bodies⟩ ···
                    (loop (+ ⟨var⟩ 1)))))])
    (loop ⟨low⟩))]))
```

In general, we must be very careful with macros to ensure expressions are evaluated the right number of times. In this instance, ⟨*low*⟩ is going to be evaluated only once and ⟨*var*⟩ is only an identifier name, but we have to make sure ⟨*high*⟩ is evaluated only once.

In fact, however, this version is *also* buggy! If there is a (*read*) in the ⟨*low*⟩ position, that's going to get evaluated second instead of first, which is presumably not what we wanted (though notice that we didn't formally specify the behavior of **for**, either). So to get it right, we really need to evaluate ⟨*low*⟩ and bind its value to an identifier first.

In general, it's safer to bind all expression positions to names. Scheme's eager evaluation semantics ensures the expressions will only be evaluted once. We don't *always* want this, but we want it so often that we may as well do it by default. (The times we accidentally bind an expression too early—for instance, the conditional expression of a **while** loop—we will usually discover the problem pretty quickly by testing.) In addition we must be sure to do this binding in the right order, mirroring what the user expects (and what our documentation for the new language construct specifies). (Observe that the problematic expression in this example is (*read*), which has the side-effect of prompting the user. Of course, we may want to limit evaluation for efficiency reasons also.)

36.3.2 Overriding Hygiene: Loops with Implicit Iteration Identifiers

When we define a loop such as the one above, we often have no real use for the loop variable. It might be convenient to simply introduce an identifier, say **it**, that is automatically bound to the current value of the loop index. Thus, the first loop example above might instead be written as

```
(for2 from 2
      to 5
      in (display it))
```

Here's a proposed macro that implements this construct:

```
(define-syntax for2
  (syntax-rules (from to in)
    [(for2 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (local ([define high-value ⟨high⟩]
             [define loop (lambda (it)
                            (if (> it high-value)
                                'done
                                (begin
                                  ⟨bodies⟩ ···
                                  (loop (+ it 1)))))])
       (loop ⟨low⟩))]))
```

Notice that in place of ⟨*var*⟩, we are now using *it*. When we run this in DrScheme, we get:

> (**for2 from** 2 **to** 5 **in** (*display it*))
reference to undefined identifier: *it*

Oops! What happened?

Actually, the macro system did exactly what it should. Remember hygiene? This was supposed to prevent *inadvertent capture* of identifiers across the macro definition/macro use boundary. It just so happens that in this case, we really do want **it** written in the macro *use* to be bound by **it** in the macro *definition*. Clearly, here's a good example of where we want to "break" hygiene, intentionally.

Unfortunately, the simple **syntax-rules** mechanism we've been using so far isn't quite up to this task; we must instead switch to a slightly more complex macro definition mechanism called **syntax-case**. For the most part, this looks an awful lot like **syntax-rules**, with a little more notation. For instance, we can define **for3** to be the same macro as **for1**, except written using the new macro definition mechanism instead:

```
(define-syntax (for3 x)
  (syntax-case x (from to in)
    [(for3 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (syntax
       (local ([define high-value ⟨high⟩]
               [define loop (lambda (⟨var⟩)
                              (if (> ⟨var⟩ high-value)
                                  'done
                                  (begin
                                    ⟨bodies⟩ ···
                                    (loop (+ ⟨var⟩ 1)))))])
         (loop ⟨low⟩)))]))
```

To convert any **syntax-rules** macro definition into a corresponding one that uses **syntax-case**, we must make the three changes boxed above (adding a parameter to the macro name, providing the parameter as an explicit argument to **syntax-case**, and wrapping the entire output expression in (**syntax** ···).

We can similarly define *for4*:

```
(define-syntax (for4 x)
  (syntax-case x (from to in)
    [(for4 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (syntax
       (local ([define high-value ⟨high⟩]
               [define loop (lambda (it)
                              (if (> it high-value)
                                  'done
                                  (begin
                                    ⟨bodies⟩ ···
                                    (loop (+ it 1)))))])
         (loop ⟨low⟩)))]))
```

This does not solve the hygiene problem; it simply enables it by converting the macro to use **syntax-case**. The reason is that **syntax-case** provides additional capabilities. In particular, it provides a procedure called *datum→syntax-object*, which takes an arbitrary Scheme datum and a term in the macro body, and "paints" the datum with the same colors as those on the macro body. This has the effect of persuading the hygiene mechanism to treat the introduced term as if it were written by the programmer. As a result, it gets renamed consistently. Thus, we must write

```
(define-syntax (for4 x)
  (syntax-case x (from to in)
    [(for4 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (with-syntax ([it (datum→syntax-object (syntax for4) 'it)])
       (syntax
        (local ([define high-value ⟨high⟩]
                [define loop (lambda (it)
                               (if (> it high-value)
                                   'done
                                   (begin
                                     ⟨bodies⟩ ···
                                     (loop (+ it 1)))))])
          (loop ⟨low⟩)))))]))
```

The **with-syntax** construct introduces new pattern variables for use in the output. The first argument to *datum→syntax-object* identifies which expression the identifier the expander must pretend "introduced" the identifier. The second, in this case, is the symbol that will be painted appropriately. Therefore, the result of expansion on our running example will look something like

```
(local ([define high-value 5]
        [define loop (lambda (g1729)
                       (if (> g1729 high-value)
                           'done
                           (begin
                             (display g1729)
                             (loop (+ g1729 1)))))])
  (loop 2))
```

Observe how the uses of **it** are all renamed consistently. (In practice, other bound identifiers such as *high-value* and even *loop* will also acquire fresh names, but we don't show that here to keep the code more readable.) Indeed, this mechanism is sufficiently robust that it will even do the right thing with nested loops:

```
(for4 from 2 to 5 in
      (for4 from 1 to it in
            (display it))
      (newline))
```

generates

```
12
```

```
123
1234
12345
```

In the inner loop, notice that the **it** in the loop bound (**from** 1 **to it**) is the iteration index for the *outer* loop, while the **it** in (*display* **it**) is the index for the inner loop. The macro system associates each **it** appropriately because each use of **for4** gets a different coat of colors. Unfortunately, we have lost the ability to refer to the outer iteration in the inner loop.

36.3.3 Combining the Pieces: A Loop for All Seasons

A better design for an iteration construct would be to combine these ways of specifying the iteration identifier (explicitly and implicitly). This is easy to do: we simply have two rules.[5] If an identifier is present, use it as before, otherwise bind **it** and recur in the macro.

```
(define-syntax (for5 x)
  (syntax-case x (from to in)
    [(for5 from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (with-syntax ([it (datum→syntax-object (syntax for5) 'it)])
       (syntax
        (for5 it from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)))]
    [(for5 ⟨var⟩ from ⟨low⟩ to ⟨high⟩ in ⟨bodies⟩ ···)
     (syntax
      (local ([define high-value ⟨high⟩]
              [define loop (lambda (⟨var⟩)
                             (if (> ⟨var⟩ high-value)
                                 'done
                                 (begin
                                   ⟨bodies⟩ ···
                                   (loop (+ ⟨var⟩ 1)))))])
        (loop ⟨low⟩))))]))
```

This passes all the expected tests: both the following expressions print the numbers 2 through 5:

```
(for5 x from 2 to 5 in (display x))
(for5 from 2 to 5 in (display it))
```

while this

```
(for5 x from 2 to 5 in
      (for5 from 1 to x in
            (printf "[~a, ~a] " x it))
      (newline))
```

[5] When defining such macros, be very sure to test carefully: if an earlier rule subsumes a later rule, the macro system will not complain, but the code will never get to a later rule! In this case we need not worry since the two rules have truly different structure.

prints

```
[2, 1] [2, 2]
[3, 1] [3, 2] [3, 3]
[4, 1] [4, 2] [4, 3] [4, 4]
[5, 1] [5, 2] [5, 3] [5, 4] [5, 5]
```

There are still ways to many ways of improving this macro. First, we might want to make sure ⟨*var*⟩ is really a variable. We can use *identifier?* for this purpose. The **syntax-case** mechanism also permits *guards*, which are predicates that refine the patterns and don't allow a rule to fire unless the predicates are met. Finally, the following program does not work:

```
(for5 x from 2 to 5 in
     (for5 from 1 to it in
          (printf "[~a, ~a] " x it))
     (newline))
```

It reports that the boxed **it** is not bound (why?). Try to improve the macro to bind **it** in this case.

36.4 Comparison to Macros in C

Macro systems have a bad rap in the minds of many programmers. This is invariably because the only macro system they have been exposed to is that of C. C's macros are pretty awful, and indeed used to be worse: macros could contain *parts* of lexical tokens, and macro application would glue them together (e.g., the identifier *list-length* could be assembled by a macro that generated *lis*, another generating *t-le* and yet another generating *ngth*). C macros are not hygienic. Because C has no notion of local scope, C macros could not easily introduce local identifiers. Finally, C macros are defined by the C pre-processor (`cpp`), which operates on files a *line at a time*. Therefore, to apply a macro over a multi-line argument, a C programmer would have to use a \ at the end of each line to fool the pre-processor into concatenating the adjacent line with the present one. Failing to remember to use the line-continuation character could lead to interesting errors.

In contrast, Scheme's macros operate over parenthesized expressions instead of pestering programmers with lines. They respect lexical boundaries (to create a new identifier, you must do so explicitly—it cannot happen by accident). Scheme macros are hygienic. They have many more features that we haven't discussed here. In short, they correct just about every mistake that C's macro system made.

36.5 Abuses of Macros

When shouldn't a programmer use macros?

As you can see, macros provide a programmer-controlled form of *inlining*, that is, directly substituting the body of an abstraction in place of its use. Compilers often inline small procedures to avoid paying the cost of procedure invocation and return. This permits programmers to define abstractions for simple operations—such as finding the corresponding matrix element in the next row when the matrix is stored

linearly, or performing some bit-twiddling—without worrying about the overhead of function invocation. Normally, inlining is done automatically by a compiler, after it has examined the size of the procedure body and determined whether or not it is cost-effective to inline.

Unfortunately, early compilers were not savvy enough to inline automatically. C programmers, looking to squeeze the last bit of performance out of their programs, therefore began to replace function definitions with macro definitions, thereby circumventing the compiler. Invariably, compilers got smarter, architectures changed, the cost of operations altered, and the hard-coded decisions of programmers came to be invalidated. Nowadays, we should *regard the use of macros to manually implement inlining as a programming error*. Unfortunately, many C programmers still think this is the primary use of macros (and in C, it's not useful for a whole lot else), thereby further despoiling their reputation. (The previous sentence is intentionally ambiguous.)

Another bad use of macros is to implement laziness. *Macros do not correspond to lazy evaluation.* Laziness is a property of when the implementation evaluates arguments to functions. Macros are not functions. For instance, in Scheme, we cannot pass a macro as an argument: try passing **or** as an argument to *map* and see what happens. Indeed, macro expansion (like type-checking) happens in a completely different phase than evaluation, while laziness is very much a part of evaluation. So please don't confuse the two.

36.6 Uses of Macros

When should a programmer use macros?

providing cosmetics Obviously, macros can be used to reduce the syntactic burden on programmers. These are perhaps the least interesting use; at least, a macro that does this should also fulfill one of the other uses.

introducing binding constructs Macros can be used to implement non-standard binding constructs. We have seen two examples, **let** and **let**$*$, above. If these were not already in the language, we could easily build them using macros. They would be impossible to define *as language contructs* in most other languages.

altering the order of evaluation Macros can be used to impose new orders-of-evaluation. For instance, we saw *time* suspend evaluation until the clock's value had been captured. The **or** construct introduced short-circuit evaluation. Often, programmers can obtain the same effect by thunking all the sub-expressions and thawing (the opposite of thunking) them in the desired order, but then the programmer would be forced to write numerous **lambda** () $\cdots$)'s—replacing one intrusive, manual pattern with another. (In particular, if a programmer fails to obey the pattern faithfully, the behavior may become quite difficult to predict.)

defining data languages Sometimes the sub-terms of a macro application may not be Scheme expressions at all. We have seen simple instances of this: for example, in (**my-let** ([**x** 3] [*y* 4]) (+ **x** *y*)), neither the parentheses wrapping the two bindings, nor those surrounding each name-value pair, signify applications. In general, the terms may have arbitrary structure, even including phrases that would be meaningless in Scheme, such as (*my-macro* (**lambda** (**x**))) that would be syntactic errors otherwise.

We can get some of the same benefit from using quotations, but those are run-time values, whereas here the macro can traverse the sub-terms and directly generate code.

In particular, suppose you wish to describe a datum without choosing whether it will be represented as a structure or as a procedure. In ordinary Scheme, you have to make this decision up front, because you cannot "introduce a **lambda**" after the fact. Designating the datum using a macro lets you hide this decision, deferring the actual representation to the macro.

Chapter 37

Macros and their Impact on Language Design

37.1 Language Design Philosophy

The *Revised*^{5} *Report on the Algorithmic Language Scheme* famously begins with the following design manifesto:

> Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

Scheme augments a minimal set of features with a powerful macro system, which enable the creation of higher-level language primitives. This approach can only work, however, with a carefully designed target language for expansion. Its success in Scheme depends on a potent combination of two forces:

- A set of very powerful core features.
- Very few restrictions on what can appear where (i.e., values in the language are truly *first-class*, which in turn means the expressions that generate them can appear nearly anywhere).

The first means many macros can accomplish their tasks with relatively little effort, and the second means the macros can be written in a fairly natural fashion.

This manner of structuring a language means that even simple programs may, unbenownst to the programmer, invoke macros, and tools for Scheme must be sensitive to this fact. For instance, DrScheme is designed to be friendly to beginners. Even simple beginner programs expand into rather complicated and relatively mangled code, many of whose constructs the beginner will not understand. Therefore, when reporting errors, DrScheme uses various techniques to make sure this complexity is hidden from the programmer.

Building a language through macros does more than just complicate error reporting. It also has significant impact on the forms of generated code that the target implementation must support. Programmers who build these implementations make certain assumptions about the kinds of programs they must handle well; these are invariably based on what "a normal human would write". Macros, however, breaks these unwritten

rules. They produce unusual and unexpected code, resulting in correctness and, particularly, performance errors. Sometimes these are easy to fix; in many other cases they are not. We will study examples to illustrate instances where macros crucially depend on the target language's handling of certain key code patterns.

37.2 Example: Pattern Matching

We will now examine a rather unusual construct that a programmer would never write, and explain why an implementation should nevertheless search for instances of it and handle it efficiently. To set the stage, consider the Scheme construct **let**, which binds names to values in a local lexical context. Though this (or an equivalent way of introducing local scope) would be a language primitive in most languages, in Scheme this is expressible as a rather simple macro in terms of first-class functions. That is,

(**let** ((*v e*) ⋯) *b*)

can be implemented by expanding into[1]

((**lambda** (*v* ⋯) *b*) *e* ⋯)

where (**lambda** (*v* ⋯) *b*) introduces an (anonymous) procedure with argument list *v* ⋯ and body *b*, and the outer parentheses apply this procedure to the argument expressions *e* ⋯. The application binds the variables *v* ⋯ to the values of the expressions *e* ⋯, and in that extended environment evaluates the body *b*—exactly what we would intend as the semantics for **let**. For instance, the program

```
(let ([x 3]
      [y 2])
  (+ x y))
```

which evaluates to 2 + 3, i.e., 5, is transformed into

```
((lambda (x y)
   (+ x y))
 3 2)
```

This macro is, in fact, quite easy to implement: thanks to hygiene and pattern matching, the implementer of **let** merely needs to write

```
(define-syntax let
  (syntax-rules ()
    [(let ([v e] ⋯) b)
     ((lambda (v ⋯) b) e ⋯)]))
```

The Scheme pre-processor finds all bodies of the form (**let** ⋯), matches them against the input pattern (here, (**let** ([*v e*] ⋯) *b*)), binds the pattern variables (*v*, *e* and *b*) to the corresponding sub-expressions, and replaces the body with the output pattern in which the pattern variables have been replaced by the sub-expressions bound to them.

[1] For simplicity, we assume the body has only one expression. In reality, Scheme permits multiple expressions in the body, which is useful in imperative programs.

There is, however, a significant performance difference between the two forms. A compiler can implement **let** by extending the current activation record with one more binding (for which space can be pre-allocated by the creator of the record). In contrast, the expanded code forces the compiler to both create a new closure and then apply it—both relatively more expensive operations.

Given this expense, you might think it silly for a Scheme system to implement the **let**-to-**lambda** macro: why take an efficient source-language instruction, whose intent is apparent, and make it less efficient behind the programmer's back? Yet at least one Scheme compiler (Chez Scheme) does precisely this. Furthermore, in the back end, it finds instances of ((**lambda** ···) ···) and effectively handles them as it would have **let**.

Why would a compiler behave so perversely? Surely no human would intentionally write ((**lambda** ···) ···), so how else could these arise? The operative phrase is, of course, "no human". Scheme programs are full of program-generating programs, and by treating this odd syntactic pattern as a primitive, *all* macros that resolve into it benefit from the compiler's optimizations.

Consider a simple symbol-based conditional matcher: the user writes a series of symbol and action pairs, such as

```
(switch [off 0]
        [on 1])
```

The matcher performs the symbol comparison and, when a symbol matches, executes the corresponding action (in this case, the actions are already numerical values). The entire (**switch** ···) expression becomes a function of one argument, which is the datum to compare. Thus, a full program might be

```
(define m
  (switch [off 0]
          [on 1]))
```

with the following interactions with the Scheme evaluator:

```
> (m 'off)
0
> (m 'on)
1
```

To implement **switch**, we need a macro rule when there are one or more cases:

```
(switch                    ⇒    (lambda (v)
  [sym0 act0]                   (if (symbol=? v (quote sym0))
  [pat-rest act-rest]               act0
  ···)                              ((switch
                                      [pat-rest act-rest]
                                      ···)
                                     v)))
```

This yields a function that consumes the actual value (v) to match against. The matcher compares v against the first symbol. If the comparison is successful, it invokes the first action. Otherwise it needs to invoke the

```
(define-syntax switch
  (syntax-rules ()
    [(switch) (lambda (v) false)]
    [(switch [sym0 act0]
             [pat-rest act-rest]
             ...)
     (lambda (v)
       (if (symbol=? v (quote sym0))
           act0
           ((switch
             [pat-rest act-rest]
             ...)
            v)))]))
```

Figure 37.1: Simple Pattern Matcher

pattern-matcher on the remaining clauses. Since a matcher is a function, invoking it is a matter of function application. So applying this function to *v* will continue the matching process.[2]

For completeness, we also need a rule when no patterns remain. For simplicity, we define our matcher to return **false**[3] (a better response might be to raise an exception):

(**switch**) ⇒ (**lambda** (*v*) **false**)

Combining these two rules gives us the complete macro, shown in Figure 37.1.

Given this macro, the simple use of **switch** given above generates

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
      0
      ((lambda (v1)
         (if (symbol=? v1 (quote on))
             1
             ((lambda (v2)
                false)
              v1)))
       v0)))
```

(I've used different names for each *v*, as the hygienic expander might, to make it easy to keep them all apart. Each *v* is introduced by another application of the **switch** macro.)

[2]The ... denotes "zero or more", so the pattern of using one rule followed by a ... is common in Scheme macros to capture the potential for an unlimited number of body expressions.

[3]In many Scheme systems, **true** and **false** are written as #t and #f, respectively.

While this expanded code is easy to generate, its performance is likely to be terrible: every time one clause fails to match, the matcher creates and applies another closure. As a result, *even if the programmer wrote a pattern matching sequence that contained no memory-allocating code, the code might yet allocate memory*! That would be most unwelcome behavior.

Fortunately, the compiler comes to the rescue. It immediately notices the ((**lambda** ⋯) ⋯) pattern and collapses these, producing effectively the code:

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
      0
      (let ([v1 v0])
        (if (symbol=? v1 (quote on))
            1
            (let ([v2 v1])
              false)))))
```

In fact, since the compiler can now see that these **let**s are now redundant (all they do is rename a variable), it can remove them, resulting in this code:

```
(lambda (v0)
  (if (symbol=? v0 (quote off))
      0
      (if (symbol=? v0 (quote on))
          1
          false)))
```

This is pretty much exactly what you would have been tempted to write by hand. In fact, read it and it's obvious that it implements a simple conditional matcher over symbols. Furthermore, it has a very convenient interface: a matcher is a first-class function value suitable for application in several contexts, being passed to other procedures, etc. The macro produced this by recursively generating lots of functions, but a smart choice of compiler "primitive"—((**lambda** ⋯) ⋯), in this case—that was sensitive to the needs of macros reduced the result to taut code. Indeed, it now leaves the code in a state where the compiler can potentially apply further optimizations (e.g., for large numbers of comparisons, it can convert the cascade of comparisons into direct branches driven by hashing on the symbol being compared).

37.3 Example: Automata

Next, we examine another optimization that is crucial for capturing the intended behavior of many programs. As an example, suppose we want to define automata manually. Ideally, we should be able to specify the automata once and have different interpretations for the same specification; we also want the automata to be as easy as possible to write (here, we stick to textual notations). In addition, we want the automata to execute fairly quickly, and to integrate well with the rest of the code (so they can, for instance, be written in-line in programs).

Concretely, suppose we want to write a simple automaton that accepts only patterns of the form $(01)^*$. We might want to write this textually as

```
automaton see0
  see0 : 0 -> see1
  see1 : 1 -> see0
```

where the state named after the keyword `automaton` identifies the initial state.

Consider a slightly more complex automaton, one that recognizes the Lisp identifier family *car*, *cdr*, *cadr*, *cddr*, *cddar* and so on. To simplify the example, let's say it should recognize the regular language $c(ad)^*r$. The corresponding automaton might look like

```
automaton init
  init : c -> more
  more : a -> more
         d -> more
         r -> end
  end  :
```

We leave defining a more formal semantics for the automaton language as an exercise for the reader.

It is easy to see that some representation of the textual description suffices for treating the automata statically. How do we implement them as programs with dynamic behavior? *We request you, dear reader, to pause now and sketch the details of an implementation before proceeding further.*

A natural implementation of this language is to create a vector or other random-access data structure to represent the states. Each state has an association indicating the actions—implemented as an association list, associative hash table, or other appropriate data structure. The association binds inputs to next states, which are references or indices into the data structure representing states. Given an actual input stream, a program would walk this structure based on the input. If the stream ends, it would accept the input; if no next state is found, it would reject the input; otherwise, it would proceed as per the contents of the data structure. (Of course, other implementations of acceptance and rejection are possible.)

One Scheme implementation of this program would look like this. First we represent the automaton as a data structure:

```
(define machine
  '((init (c more))
    (more (a more)
          (d more)
          (r end))
    (end)))
```

The following program is parameterized over machines and inputs:

```
(define (run machine init-state stream)
  (define (walker state stream)
    (or (empty? stream)       ;; if empty, return true, otherwise ...
        (let ([transitions (cdr (assv state machine))]
              [in (first stream)])
          (let ([new-state (assv in transitions)])
```

```
            (if new-state
                (walker (cadr new-state) (rest stream))
                false)))))
    (walker init-state stream))
```

Here are two instances of running this:

```
> (run machine 'init '(c a d a d d r))
true
> (run machine 'init '(c a d a d d r r))
false
```

This is not the most efficient implementation we could construct in Scheme, but it is representative of the general idea.

While this is a correct implementation of the semantics, it takes quite a lot of effort to get right. It's easy to make mistakes while querying the data structure, and we have to make several data structure decisions in the implementation (which we have done only poorly above). Can we do better?

To answer this question affirmatively, let's ignore the details of data structures and understand the *essence* of these implementations.

1. Per state, we need fast conditional dispatch to determine the next state.
2. Each state should be quickly accessible.
3. State transition should have low overhead.

Let's examine these criteria more closely to see whether we can recast them slightly:

fast conditional dispatch This could just be a conditional statement in a programming language. Compiler writers have developed numerous techniques for optimizing properly exposed conditionals.

rapid state access Pointers of any sort, including pointers to *functions*, would offer this.

quick state transition If only function calls were implemented as `gotos` ...

In other words, the `init` state could be represented by

```
(lambda (stream)
  (or (empty? stream)
      (case (first stream)
        [(c) (more (rest stream))]
        [else false])))
```

That is, if the stream is empty, the procedure halts returning a true value; otherwise it dispatches on the first stream element. Note that the boxed expression is invoking the code corresponding to the `more` state. The code for the `more` state would similarly be

```
(lambda (stream)
  (or (empty? stream)
```

```
(case (first stream)
  [(a) (more (rest stream))]
  [(d) (more (rest stream))]
  [(r) (end (rest stream))]
  [else false]))))
```

Each underlined name is a reference to a state: there are two self-references and one to the code for the `end` state. Finally, the code for the `end` state fails to accept the input if there are any characters in it at all. While there are many ways of writing this, to remain consistent with the code for the other states, we write it as

```
(lambda (stream)
  (or (empty? stream)
      (case (first stream)      ;; no matching clauses, so always false
        [else false])))
```

The full program is shown in Figure 37.2. This entire definition corresponds to the machine; the definition of *machine* is bound to *init*, which is the function corresponding to the `init` state, so the resulting value needs only be applied to the input stream. For instance:

```
> (machine '(c a d a d d r))
true
> (machine '(c a d a d d r r))
false
```

What we have done is actually somewhat subtle. We can view the first implementation as an *interpreter* for the language of automata. This moniker is justified because that implementation has these properties:

1. Its output is an answer (whether or not the automaton recognizes the input), not another program.
2. It has to traverse the program's source as a data structure (in this case, the description of the automaton) repeatedly across inputs.
3. It consumes both the program and a specific input.

It is, in fact, a very classical interpreter. Modifying it to convert the automaton data structure into some intermediate representation would eliminate the second overhead in the second clause, but would still leave open the other criteria.

In contrast, the second implementation given above is the *result of compilation*, i.e., it is what a compiler from the automaton language to Scheme might produce. Not only is the result a program, rather than an answer for a certain input, it also completes the process of transforming the original representation into one that does not need repeated processing.

While this compiled representation certainly satisfies the automaton language's semantics, it leaves two major issues unresolved: efficiency and conciseness. The first owes to the overhead of the function applications. The second is evident because our description has become much longer; the interpreted solution required the user to provide only a concise description of the automaton, and reused a generic interpreter to manipulate that description. What is missing here is the actual compiler that can generate the compiled version.

```
(define machine
  (letrec ([init
            (lambda (stream)
              (or (empty? stream)
                  (case (first stream)
                    [(c) (more (rest stream))]
                    [else false])))]
           [more
            (lambda (stream)
              (or (empty? stream)
                  (case (first stream)
                    [(a) (more (rest stream))]
                    [(d) (more (rest stream))]
                    [(r) (end (rest stream))]
                    [else false])))]
           [end
            (lambda (stream)
              (or (empty? stream)
                  (case (first stream)
                    [else false])))])
    init))
```

Figure 37.2: Alternate Implementation of an Automaton

37.3.1 Concision

First, let us slightly alter the form of the input. We assume that automata are written using the following syntax (presented informally):

```
(automaton init
  (init : (c → more))
  (more : (a → more)
          (d → more)
          (r → end))
  (end  : ))
```

The general transformation we want to implement is quite clear from the result of compilation, above:

```
(state : (label → target) ···)   ⇒   (lambda (stream)
                                       (or (empty? stream)
                                           (case (first stream)
                                             [(label) (target (rest stream))]
                                             ···
                                             [else false])))
```

```
(define-syntax automaton
  (syntax-rules (: →)      ;; match ':' and '→' literally, not as pattern variables
    [(automaton init-state
       (state : (label → target) ...)
       ...)
     (letrec ([state
               (lambda (stream)
                 (or (empty? stream)
                     (case (first stream)
                       [(label) (target (rest stream))]
                       ...
                       [else false])))]
              ...)
       init-state)]))
```

Figure 37.3: A Macro for Executable Automata

Having handled individual rules, we must make the automaton macro wrap all these procedures into a collection of mutually-recursive procedures. The result is the macro shown in Figure 37.3. To use the automata that result from instances of this macro, we simply apply them to the input:

```
> (define m (automaton init
                       [init : (c → more)]
                       [more : (a → more)
                               (d → more)
                               (r → end)]
                       [end : ]))
> (m '(c a d a d d r))
true
> (m '(c a d a d d r r))
false
```

By defining this as a macro, we have made it possible to truly embed automata into Scheme programs. This is true purely at a syntactic level—since the Scheme macro system respects the lexical structure of Scheme, it does not face problems that an external syntactic preprocessor might face. In addition, an automaton is just another applicable Scheme value. By virtue of being first-class, it becomes just another linguistic element in Scheme, and can participate in all sorts of programming patterns.

In other words, the macro system provides a convenient way of writing compilers from "Scheme+" to Scheme. More powerful Scheme macro systems allow the programmer to embed languages that are truly different from Scheme, not merely extensions of it, into Scheme. A useful slogan (due to Matthew Flatt and quite possibly others) for Scheme's macro system is that it's a *lightweight compiler API*.

37.3.2 Efficiency

The remaining complaint against this implementation is that the cost of a function call adds so much overhead to the implementation that it negates any benefits the **automaton** macro might conceivably manifest. In fact, that's not what happens here at all, and this section examines why not.

Tony Hoare once famously said, "Pointers are like jumps"[4]. What we are seeking here is the reverse of this phenomenon: what is the `goto`-like construct that corresponds to a dereference in a data structure? The answer was given by Guy Steele: the *tail call*.

Armed with this insight, we can now reexamine the code. Studying the output of compilation, or the macro, we notice that the conditional dispatcher invokes the function corresponding to the next state on the rest of the stream—but does not touch the return value. This is no accident: the macro has been carefully written to only make tail calls.[5]

In other words, the state transition is hardly more complicated than finding the next state (which is statically determinate, since the compiler knows the location of all the local functions) and executing the code that resides there. Indeed, the code generated from this Scheme source looks an awful lot like the automaton representation we discussed at the beginning of section 37.3: random access for the procedures, references for state transformation, and some appropriately efficient implementation of the conditional.

The moral of this story is that we get the same representation we would have had to carefully craft by hand virtually for free from the compiler. In other words, *languages represent the ultimate form of reuse*, because we get to reuse everything from the mathematical (semantics) to the practical (libraries), as well as decades of research and toil in compiler construction.

Tail Calls versus Tail Recursion

This example should help demonstrate the often-confused difference between tail *calls* and tail *recursion*. Many books discuss tail recursion, which is a special case where a function makes tail calls to *itself*. They point out that, because implementations must optimize these calls, using recursion to encode a loop results in an implementation that is really no less efficient than using a looping construct. They use this to justify, in terms of efficiency, the use of recursion for looping.

These descriptions unfortunately tell only half the story. While their comments on using recursion for looping are true, they obscure the subtlety and importance of optimizing all tail *calls*, which permit a family of functions to invoke one another without experiencing penalty. This leaves programmers free to write readable programs without paying a performance penalty—a rare "sweet spot" in the readability-performance trade-off. Traditional languages that offer only looping constructs and no tail calls force programmers to artificially combine procedures, or pay via performance.

The functions generated by the **automaton** macro are a good illustration of this. If the implementation did not perform tail-call optimization but the programmer needed that level of performance, the macro would be forced to somehow combine all the three functions into a single one that could then employ a looping

[4]The context for the quote is pejorative: "Pointers are like jumps, leading wildly from one part of the data structure to another. Their introduction into high-level languages has been a step backwards from which we may never recover."

[5]Even if the code did need to perform some operation with the result, it is often easy in practice to convert the calls to tail-calls using accumulators. In general, as we have seen, the conversion to continuation-passing style converts all calls to tail calls.

construct. This leads to an unnatural mangling of code, making the macro much harder to develop and maintain.

37.4 Other Uses

Scheme macros can do many more things. *datum→syntax-object* can be used to manufacture identifiers from syntax supplied to the macro. Macros can also define other macros! You will find such examples as you begin to employ macros in your work. Books such as Kent Dybvig's *The Scheme Programming Language* and Paul Graham's *On Lisp* elaborate on these programming styles.

37.5 Perspective

We have now seen several examples of Scheme's macro system at work. In the process, we have seen how features that would otherwise seem orthogonal, such as macros, first-class procedures and tail-calls, are in fact intimately wedded together; in particular, the absence of the latter two would greatly complicate use of the former. In this sense, the language's design represents a particularly subtle, maximal point in the design space of languages: removing any feature would greatly compromise what's left, while what is present is an especially good notation for describing algorithms.

Part XIII

What's Next?

Chapter 38

Programming Interactive Systems

Consider writing a program that solves a graph algorithm such as finding a minimum spanning tree. Your implementation of the algorithm may suffer from several problems: even if the program runs to completion, the output it produces may not be minimal, it may not span, and it may not even be a tree! While debugging such a program by manual inspection of intermediate data structures can be onerous, often (especially if the bug is in one of the latter two cases) it is easy to spot the problem visually. Suppose, however, you were writing a library that was meant to be used in textual applications. It would not make sense to add graph-drawing code to your library program. But how can you instrument your program from the "outside" to add this functionality?

In principle, adding this graph-drawing instrumentation is something a debugger should support well. It is, however, a very domain-specific instrumentation; it doesn't make sense for a general-purpose debugger to offer the ability to add graph-drawing code, because this is a property many applications don't require. In general, a debugger offers only a general, domain-independent view of a program's data, while we would often like to specialize this view to a domain-specific one. Ideally, that is, we would like to make the debugger *scriptable*, so each application can install its own programs that run atop a library of debugging primitives.

In most programming languages, programs expect to be in control of their execution. That is, the beginning of the program initiates a process of execution, and the end of the program terminates it. The program may invoke external agents (such as functions in libraries, procedures written in other languages through a foreign-function interface, or even a method residing on a remote host through a remote procedure call mechanism), but expect at some point (either synchronously or asynchronously) to get a response that resumes the computation.

In a debugger scripting language, in contrast, the script is most certainly not in control of the computation. Two other programs—the target program being debugged, and the debugger itself—control the flow of computation. The script should have the ability to install an event generator that triggers whenever some event of interest—such as a method invocation, a variable mutation, a thread spawn, and so forth—happens in the target program. The script only runs in response to these events. Furthermore, the script does not return any answers to the target program (which is essentially unaware of the script's existence) or to the debugger (which, being general-purpose, may not know what to do with it), but rather preserves information for its own (possible) subsequent execution.

In fact, most modern applications share this characteristic: the application must lay dormant waiting for some behavior from an external source of events. We have already seen this property in the case of Web applications; programmers needed to invert the structure of their source to expose the points of resumption, and manually represent the rest of the computation at each point. To remedy this problem, we devised the **send/suspend** primitive, and showed that better language support makes Web applications far easier to construct and maintain.

The Web is a simple interactive medium in the sense that there is only one kind of behavior available to a user, namely clicking on a link or on a button. Other user actions are masked by the browser and not conveyed to the Web application. In contrast, a modern GUI application—such as that Web browser itself—must respond to multiple kinds of behavior, including keystrokes, mouse button pushes and mouse clicks. Applications may also need to be sensitive to the receipt of network packets or the ticking of the system clock.

Confronted with the challenge of building complex interactive systems, most programming languages have taken a rather weak route. Traditional programming languages offer the *callback* as the primary method of registering code that will respond to events. A callback is essentially a procedure that consumes arguments corresponding to the event information—such as which key was pressed—and returns... nothing.

Why does a callback not return anything? A callback is a piece of application code that is "installed" in the underlying system layer, to be invoked when that layer receives an event that the application must process. That is, the system code invokes the callback. But it is the application that must process the event, so there is no meaningful information that the callback can return to the system. This explains the return type.

What do we know about procedures that do not return a value? To be of any use at all, they must accomplish their action by using side-effects (such as mutation). This explains why side-effects, especially mutation, are so prevalent in GUI applications. But side-effects, and procedures that don't return any value, are difficult to compose, and make it harder to reason about a program's behavior. Is it surprising that GUIs are difficult to develop correctly and to test for the absence of errors?

Many programmers see the current design of GUI libraries, with their plethora of callbacks, as a fixture in the design space. We should know better. The callback-based solution should be especially familiar to us from our Web programming experience: the continuation generated by CPS a procedure that does not return any meaningful value (in fact, it doesn't return at all), and represents the rest of the computation. As such, it is a special kind of callback. If through better language design we were able to improve the state of Web programming, perhaps better language design can improve GUI programming (and debugger scripting, and ...) also?

This is precisely the kind of challenge that leads to research in programming languages. For instance, the FrTime[1] language in DrScheme is designed to make developing interactive applications, from debugging scripts to GUIs, more direct. Though FrTime programs superficially look like Scheme programs, they evaluate under a very different semantics: certain expressions are known to yield sequences of events, and any computation that depends on it in turn gets *recomputed* every time a new event occurs. Thus, for instance, the expression

(*make-circle mouse-pos* 10 "blue")

[1] Pronounced "father time".

automatically draws a blue circle of radius ten pixels wherever the mouse is currently centered, because *mouse-pos* is a value in the language that updates whenever the mouse moves. Because it updates, every expression that depends on it must also update to remain fresh. There is, therefore, no need for a loop to ensure the circle is re-drawn: the "loop" is built into the language semantics! Note, of course, that there is no need for a callback either.

This is just a miniscule example of how a deeper analysis of a problem can lead to better linguistic tools, which can in turn make a problem much simpler than it was initially. Look up FrTime in DrScheme for more on this fascinating language.

Chapter 39

What Else is Next

At this point, it may be humbling to consider all the topics in programming languages that we *haven't* covered: object-oriented programming, class-oriented programming and type systems for the same; threads; module systems; middleware; and so on. And that's just looking at what programmers routinely use in practice. A bit further afield we find techniques such as mixins (parameterized classes), contracts and aspect-oriented programming, which are rapidly gaining a foothold in industry as well. And then we get to topics that are still very much in the realm of research, such as dependent types, type classes, delimited control, resource accounting or logic metaprogramming. We've applied programming language theory to one important practical problem (Web programming), but numerous ones abound (such as the static validation of spreadsheets).

The particular set of topics we've chosen to study in this book meet one of these critiera: they're essential knowledge before we can examine these other topics above, or they're *just plain fun*. Fortunately, most topics are both.

Go forth and populate the world with beautiful designs.

Index