# Programming Language Principles and Paradigms

*Release 0.4*

**Amir Kamil**

**Jan 27, 2024**

# CONTENTS

# Part I

# Foundations

This text covers the fundamental concepts in programming languages. While we will be using several languages, the purpose of the text is not to learn different languages. Instead, it is to learn the concepts that will both facilitate learning a new language quickly and make better use of the programming constructs that a programming language provides. To analogize with spoken languages, the subject of this text is more akin to linguistics rather than a specific language.

Topics that are covered in this text include programming-language features for naming, control flow, and memory management, basic theory of programming languages, such as grammars and type systems, and various programming paradigms including functional, object-oriented, and logic-programming techniques. We will also consider advanced programming techniques such as generic programming and code generation.

CHAPTER

# ONE

# INTRODUCTION

> There are no solutions; there are only trade-offs. — Thomas Sowell

A programming language is a language designed for expressing computer programs at a higher level than a machine language. While many programmers consider programming languages such as C to be more powerful than assembly, and higher-level languages such as C++ and Python to be more powerful than C, in reality, all languages can solve exactly the same problems. This perceived power differential is due to the set of abstractions each language provides, and to what degree a language facilitates programming in different paradigms and patterns.

There are countless programming languages in existence. A list of notable languages on Wikipedia enumerates over 700 languages. If all languages can solve the same problems, why are there so many languages?

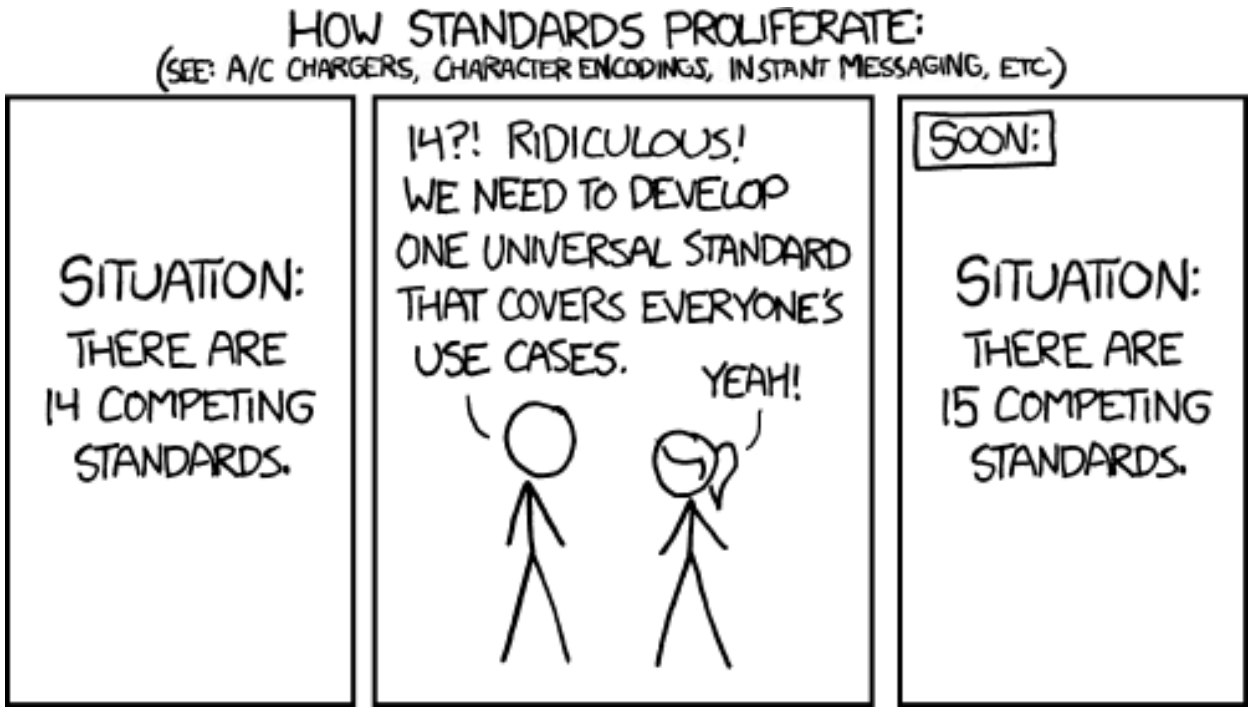



Figure 1.1: Credit: xkcd

A language occupies a point in the space of tradeoffs between different design goals. These include ease of writing code, readability, performance, maintainability, portability, modularity, safety, and many other considerations. It is impossible to optimize for all these goals simultaneously. Instead, they must be balanced according to the intended use of a language.

A language may also be intended for a specific problem domain and therefore support target design goals and abstractions that are important to that domain. A concrete example is Fortran, which is suited to numerical computations by

providing a multidimensional array abstraction with excellent performance.

Programming languages are often designed with a particular language paradigm in mind. One such paradigm is *imperative programming*, where a program is decomposed into explicit computational steps in the form of *statements*. Another general pattern is *declarative programming*, where computation is expressed in terms of *what* it should accomplish rather than *how*. More specific styles within this space include *functional programming*, which models computation after mathematical functions and avoids mutation, and *logic programming*, which expresses a program in the form of facts and rules. One last example is *object-oriented programming*, which organizes data into *objects* and computation into *methods* that are associated with those objects. These language paradigms are not mutually exclusive, and higher-level languages often support a combination of paradigms.

Languages also differ in the design of their *type systems*. Entities in a programming language are generally associated with a type, which determines what operations are valid on those entities and how to perform those operations. Two common methodologies are *static typing*, in which types are inferred directly from a program's source code and checked at compile time, and *dynamic typing*, where types are tracked and checked at runtime. Often languages use a combination of these systems, such as with dynamic casting in C++.

A final consideration in designing and implementing a language is whether it is intended to be *compiled* or *interpreted*. In compilation, a program is transformed from its original code into a form that is more suited to direct execution on a particular system. This usually occurs separately from running the program, and the translation need only be done once for a program on a specific system. In contrast, interpreting code entails simulating its execution at runtime, which generally results in lower performance than compilation. However, interpreters can enable greater flexibility than compilers, since the original code is available and program state is more easily accessible. Modern languages often use a combination of compilation and interpretation.

A common aspect of these design areas is that they do not consist of discrete choices. Rather, they present a continuum between different canonical choices, and programming languages often fall somewhere along that continuum. When we say that a language, for instance, is statically typed, in actuality we mean that the predominant form of type checking is static, even though the language may have some elements of dynamic typing.

CHAPTER

TWO

# BASIC PYTHON

A language isn't something you learn so much as something you join. — Arika Okrent

Python is a widely used programming language that supports many programming paradigms and has numerous libraries in a wide variety of application domains. We will use Python, along with other languages, to explore the design space of programming languages. While some systems come with a version of Python already installed, in this text, we will be using the most recent stable release of Python 3. Installation packages can be found on the downloads page of the Python website.

Python is an interpreted language, and a good way to gain familiarity with Python is to start the interpreter and interact with it directly. In order to start up the interpreter, you will need to go to your command prompt and type `python`, `python3` or `python3.5` depending on how many versions are installed on your machine. Depending on the operating system you are using, you might also have to modify your PATH.

Starting the interpreter will bring up the >>> prompt, allowing you to type code directly into the interpreter. When you press enter, the Python interpreter will interpret the code you typed, or if the code is syntactically incomplete, wait for more input. Upon evaluating an expression, the interactive interpreter will display the evaluation result, unless evaluation resulted in the special `None` value.

```
>>> 3 + 4
7
```

```
>>> abs(-2.1)
2.1
```

```
>>> None
```

Each session keeps a history of what you have typed. To access that history, press `<Control>-P` (previous) and `<Control>-N` (next). `<Control>-D` exits a session, which discards this history. Up and down arrows also cycle through history on some systems.

## 2.1 Variables

Variables in Python do not have a static type. They are introduced by assigning a value to a name:

```
>>> x = 4
>>> x
4
```

Binding a variable to a value of one type does not preclude binding it to a value of a different type later on:

```
>>> x = 4
>>> x = 'hello'
>>> x
'hello'
>>> x = 4.1
>>> x
4.1
```

Multiple variables can be assigned to in a single statement using a comma to separate names on the left-hand side and values on the right-hand side:

```
>>> y, z = x + 1, x + 2
>>> y
5.1
>>> z
6.1
```

## 2.2 Basic Data Structures

Multiple assignment is actually an example of using a *tuple*, which is an immutable compound data type. In the context of programming languages, something is immutable if its state cannot be changed after it was first created. A tuple is constructed by separating values by commas, and then optionally surrounding the values with parentheses.

```
>>> a = (3, 4)
>>> a
(3, 4)
```

Individual elements of a tuple can be accessed with square brackets.

```
>>> a[0]
3
>>> a[1]
4
```

Negative indices access a container in reverse, with -1 corresponding to the last element:

```
>>> a[-1]
4
>>> a[-2]
3
```

Lists are mutable containers, and they are constructed using square brackets around the values.

```
>>> b = [5, 6]
>>> b
[5, 6]
```

Unlike tuples, list elements can be modified, and new elements can be appended to the end of a list:

```
>>> b[1] = 7
>>> b.append(8)
```

(continues on next page)

(continued from previous page)

```
>>> b
[5, 7, 8]
```

The `dir` function can be used to inspect the full interface of the `list` type:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getslice__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert',
'pop', 'remove', 'reverse', 'sort']
```

Documentation of a particular method can be retrieved with the `help` function:

```
>>> help(list.append)
Help on method_descriptor:
append(...)
    L.append(object) -- append object to end
```

A `dict` (short for *dictionary*) is an associative container that maps a key to a value. It is created by enclosing key-value pairs within curly braces.

```
>>> d = { 1 : 2, 'hello' : 'world' }
>>> d
{1: 2, 'hello': 'world'}
>>> d[1]
2
>>> d['hello']
'world'
```

Strings are denoted by either single or double quotes. A common convention is to use single quotes unless the string contains a single quote as one of its characters.

```
>>> 'hello world'
'hello world'
>>> "hello world"
'hello world'
```

Furthermore, A string can span multiple lines if it is enclosed in triple quotes. For example:

```
>>> x = """
... Hello
... World!
... """
>>> x
'\nHello\nWorld!\n'
```

Where `\n` is the newline character.

## 2.3 Compound Statements

In Python, a sequence of statements, also called a *suite*, consists of one or more statements preceded by the same indentation. Unlike other languages, such as C++, indentation is meaningful, and inconsistent indentation is a syntax error. Common convention in Python is to use four spaces per indentation level. Avoid using tabs, as they are not visually distinguishable from spaces but are considered distinct by the interpreter.

A conditional statement is composed of an `if` clause, zero or more `elif` clauses, and an optional `else` clause:

```
if <expression>:
    <suite>
elif <expression>:
    <suite>
else:
    <suite>
```

A suite must be indented further than its header, and each statement in the suite must have the same indentation. Each header must end with a colon. The conditional expression need not be parenthesized.

```
>>> if pow(2, 3) > 5:
        print('greater than')
    elif pow(2, 3) == 5:
        print('equal')
    else:
        print('less than')
greater than
```

While loops have similar syntax:

```
while <expression>:
    <suite>
```

For loops iterate over a sequence, similar to the range-based for loop in C++:

```
for <variable> in <sequence>:
    <suite>
```

```
>>> for i in [3, 4, 5]:
        print(i)
3
4
5
```

## 2.4 Function Definitions

A function is defined with the `def` statement:

```
def <function>(<arguments>):
    <suite>
```

In keeping with Python's lack of static typing, the return and argument types are not specified.

```
>>> def square(x):
        return x * x
```

If a function does not explicitly return a value when it is called, then it returns the special `None` value.

```
>>> def print_twice(s):
        print(s)
        print(s)
>>> x = print_twice(3)
3
3
>>> x
>>> print(x)
None
```

A `def` statement binds a function object to the given name. Unlike in some other languages, this name can be rebound to something else.

```
>>> print_twice
<function print_twice at 0x103e0e488>
>>> print_twice = 2
>>> print_twice
2
>>> print_twice(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
```

In Python, definitions are actually executed when they are encountered. For a function definition, this creates a new function object and binds it to the name specified in the definition.

## 2.5 Class Definitions

A class is defined with a `class` statement:

```
class <name>(<base classes>):
    <suite>
```

The list of base classes can be elided, in which case the base class is `object`.

When defining an instance method in Python, the definition explicitly takes in `self` as the first parameter. When the method is called, the receiving object is implicitly passed in to this first parameter.

```
>>> class Cat:
        def speak(self):
            print('meow')
>>> Cat().speak()
meow
```

The constructor is defined using the special `__init__` method. Member variables, more properly called *attributes* in Python, are introduced using the `self` parameter and dot syntax.

```
>>> class Square:
        def __init__(self, side_length):
            self.side = side_length
        def perimeter(self):
            return 4 * self.side
        def area(self):
            return self.side * self.side
>>> s = Square(3)
>>> s.perimeter()
12
>>> s.area()
9
```

## 2.6 Modules

Python has a number of built-in libraries organized as modules, and an individual `.py` file also represents a module. Modules can be loaded using the `import` statement:

```
import <modules>
```

This binds module objects to their corresponding names in the current environment, which can then be used to access an attribute of a module.

```
>>> import operator, math
>>> math.pow(operator.mul(2, 3), 2)
36.0
```

Individual attributes of a module can also be introduced into the environment using another form of the import statement:

```
from <module> import <attributes>
```

```
>>> from math import pow
>>> pow(2, 3)
8
```

Another variant imports all names from a module:

```
from <module> import *
```

```
>>> from operator import *
>>> mul(2, 3)
6
```

## 2.7 Executing a Module

Python does not specify a special `main` function like the C family of languages. Instead, all code in a module is interpreted when it is loaded, starting from the top.

It is possible to specify a piece of code that does not run when a module is imported, but runs when a module is executed directly at the command-line, as in:

```
python3 program.py <arguments>
```

This is accomplished by checking if the `__name__` attribute is set to `'__main__'`:

```
if __name__ == '__main__':
    <suite>
```

The suite will only be executed if the module is executed on the command-line.

Command-line arguments can be obtained using the `argv` list in the `sys` module. As in C and C++, the first argument is the name of the program.

## 2.8 Python Reference Semantics

A variable in Python is actually an indirect *reference* to an object, rather than holding the object directly in the variable's memory location. Thus, a Python variable is analogous to a C++ pointer, and assigning from one variable to another merely copies the indirect reference rather than copying the object. The following example illustrates this:

```
>>> x = []
>>> y = x
>>> y.append(3)
>>> x
[3]
```

The assignment `y = x` results in both `x` and `y` referring to the same list object, so that the `x` reference reflects the modification to the list that was made through the `y` reference. Thus, an assignment does not make a copy of an object. To copy an object, we can use the `copy()` function in the `copy` module (or the `deepcopy()` function in the same module if we want a deep rather than a shallow copy). Alternatively, many types can be copied by invoking the constructor with an existing object, as in the following:

```
>>> x = [3]
>>> y = list(x)
>>> y.append(-7)
>>> y
[3, -7]
>>> x
[3]
```

We will discuss *reference semantics* in more detail later.

CHAPTER

THREE

# BASIC ELEMENTS

A programming language is a formal system for expressing computation. Any formal language, whether natural, mathematical, or programming, has rules that specify what sequences of symbols are meaningful in the language. We will see many of the rules that govern a programming language throughout this text, but we begin with the basic elements that comprise a program.

## 3.1 Levels of Description

A language, whether a spoken language or a programming language, can be described at multiple levels of abstraction, from how the most basic pieces of the language can be formed, to how they can be combined to construct meaningful phrases, to how those phrases can be used to accomplish a purpose. The following are the levels we consider when it comes to a programming language:

- *Grammar* determines what phrases are correct. It can be further divided into *lexical structure*, which defines how the words of the language are constructed, and *syntax*, which determines what sequences of words form correct phrases.
- *Semantics* specify the meaning of a correct phrase.
- *Pragmatics* are concerned with the practical use of correct phrases. In programming languages, this includes common design patterns and programming practices.
- An *implementation* determines how the actions specified by a meaningful phrase are accomplished. This level of description is unique to programming languages, which we use to write programs that perform actual tasks that need to be done.

We proceed to take a closer look at the first two levels of description. We will consider the latter two levels later.

### 3.1.1 Lexical Structure

The *lexical structure* of a language determines what constitutes the words that are in the language, more commonly called *tokens* in the context of programming languages. Valid characters are defined by the alphabet, generally ASCII or Unicode in a programming language, and tokens are composed of one or more consecutive characters. Tokens are often separated by whitespace, and a token also ends if it is followed by a character that is invalid for the token.

The classes of tokens depend on the particular language, but common classes are identifiers, keywords, literals, operators, and separators.

A *literal* represents a particular value directly in source code. Literals include integer and floating-point numbers, booleans, characters, and strings. Often a language provides different literal representations for each primitive type. For example, C++ includes `int`, `long`, and `long long` integer literals by using the `l` and `ll` suffixes for the latter two. A language may also support different representations for literals of a particular type, such as decimal, hexadecimal,

octal, and binary integer literals. Some languages, such as C++11, even allow user-defined literals that can represent arbitrary types.

*Operators* such as + and == are commonly defined as special tokens. However, some languages such as Scheme do not treat operators as special; instead, they are considered to be identifiers.

An *identifier* is a sequence of characters that can be used to name entities in a program. In languages such as Python and C++, an identifier begins with a letter or underscore and can subsequently contain letters, underscores, and digits. Java allows identifiers to contain the dollar sign ($) symbol, though general practice is to reserve it for machine-generated names. Scheme allows many more symbols to be part of an identifier. Most languages are case sensitive in that capitalization is significant. However, some other languages, such as Scheme, treat identifiers in a case-insensitive manner.

A *keyword* is a sequence of characters that has the form of an identifier but has special meaning in the language, such as the token `if` in many languages. Depending on the language, a keyword can be forbidden from being used as a name, or its meaning can be determined based on context.

*Separators*, also called *delimiters* or *punctuators*, are the punctuation of a language, denoting the boundary between programmatic constructs or their components. Common separators include parentheses, curly braces, commas, and semicolons. In some cases, a token may act as a separator or as an operator depending on the context, such as a comma in C and C++.

The lexical structure of a language is usually specified using regular expressions, and breaking source code into tokens is often the first step in compiling or interpreting a program. The particulars of regular expressions will be discussed later on in this text.

### 3.1.2 Syntax

The *syntax* of a language specifies what sequences of tokens constitute valid fragments of the language. Syntax concerns only the structure of a program; source code may be syntactically correct but semantically invalid, resulting in an invalid program.

An example of a syntactic rule is that parentheses must be balanced within a code fragment. For example, the following code consists of valid tokens in C++ but is not syntactically valid:

```
x = (1 + ;
```

Another example of a syntax rule is that consecutive identifiers are generally illegal in Python or C++ (declarations being an exception in the latter).

The syntax rules of a language are specified using a formal grammar, a topic we will return to later in the text.

### 3.1.3 Semantics

Whereas syntax is concerned with the structure of code fragments, *semantics* determines the meaning of a code fragment. In particular, it indicates what value is computed or what action is taken by a code fragment.

Defining a programming language requires assigning semantics to each syntactic construct in the language. As we will see later, there are formal methods for describing the semantics of a construct. However, given the complexity of most languages and the fact that most programmers are not trained in formal semantics, semantics are often described using natural language.

Semantics further restrict what constitutes valid code. For example, the following is syntactically correct in C++ but semantically invalid:

```
int x = 3;
x.foo(); // invalid
```

## 3.2 Entities, Objects, and Variables

An *entity*, also called a *citizen* or *object* (though we use the latter term more specifically, as defined below), denotes something that can be named in a program. Examples include types, functions, data objects, and values.

A *first-class entity* is an entity that supports all operations generally available to other entities, such as being associated with a variable, passed as an argument, returned from a function, and created at runtime. The set of first-class entities differs between programming languages. For example, functions and types are first-class entities in Python, but not in C++ or Java. (Functions in C++ have many of the characteristics of first-class entities, but they cannot be created dynamically, so they are not quite first class.) Control of execution may also be a first-class entity, as we will see in *Continuations*. Table 3.1 summarizes the first-class entities in C++, Java, Python, and Scheme.

Table 3.1: First-class entities in C++, Java, Python, and Scheme.

| | C++ | Java | Python | Scheme |
|---|---|---|---|---|
| Functions | no (almost) | no | yes | yes |
| Types | no | no | yes | no |
| Control | no | no | no | yes |

An *object* is a location in memory that holds a value. An object may be modifiable, in which case the value it holds may change, or it may be constant. A *variable* is a name paired with an object. In some languages, multiple names may be associated with the same object, in which case the names are said to *alias* the same object.

An object has a *lifetime* during which it is valid to use that object while a variable has a *scope*, which specifies the parts of a program that have access to that variable. An object also has a *type* that determines what its data represents and the operations that the object supports. We will examine these concepts in more detail later on.

## 3.3 L-Values and R-Values

An object actually has two values associated with it: its memory location and the contents of that memory location. The former is called an *l-value* while the latter is an *r-value*, after the fact that they are generally used on the left-hand side and right-hand side of an assignment, respectively. Most languages implicitly convert l-values to r-values when necessary.

As a concrete example, consider an integer variable `x`:

```
int x = 3;
```

The name `x` denotes a memory location that is initialized to hold the value `3`. When the name `x` is evaluated, the result is an l-value. However, it is automatically converted to an r-value in the following definition:

```
int y = x;
```

The initialization of the variable `y` requires an r-value, so `x` is converted to its r-value `3`. On the other hand, in the following assignment, an l-value is required on the left-hand side:

```
x = 4;
```

The left-hand side evaluates to the memory location denoted by `x` and changes its contents to the r-value `4`.

Temporary objects, such as the result of `x + 3`, have r-values but do not necessarily have l-values. Most languages do not allow access to a temporary's l-value even if it has one.

We will return to l-values and r-values when we discuss *value and reference semantics*.

## 3.4 Expressions

An *expression* is a syntactic construct that results in a value. An expression is *evaluated* to produce the resulting value.

The simplest expressions are literals, which evaluate to the value they represent, and identifiers, which evaluate to the l-value or r-value of the corresponding object, assuming that a variable is in scope that associates the identifier with an object.

Simple expressions can be combined to form *compound expressions* according to the rules defined by a language. Combinators include operators such as + or `.`. A function call is also generally a compound expression, as in:

```
print("Hello", "world")
```

Depending on the language, the functor itself (`print` in the example above) can be an expression. Each argument is also an expression.

Operators have *precedence* rules that determine how subexpressions are grouped when multiple operators are involved. For example, the following expression typically evaluates to 7 in languages that have infix operators, since * has higher precedence than +:

```
1 + 2 * 3
```

Infix languages generally allow subexpressions to be explicitly grouped using parentheses:

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

An operator also has an *associativity* that determines how its operands group when there are multiple operators of the same precedence. Binary operators typically associate from left to right, while unary operators generally have right associativity. A notable exception are assignment operators in languages such as C++, which associate right to left. This allows expressions such as:

```
a = b = c = 0
```

This is equivalent to:

```
(a = (b = (c = 0)))
```

So the end result is that all of `a`, `b`, and `c` are assigned the value 0.

In addition to defining how subexpressions are grouped together, the language must specify the order in which those subexpressions are evaluated. In many languages, such as Python and Java, subexpressions are generally evaluated in order from left to right. In Scheme, C, and C++, however, order of evaluation is left up to the implementation in many cases. Consider the following example in C++:

```
int x = 3;
cout << ++x << " " << x << endl;
```

With C++14 and earlier, this code can result in `4 3` or `4 4` being printed, depending on the implementation. C++17 modified the order-of-evaluation rules such that this code always results in `4 4`, though there are other cases were the order of evaluation is still left to the implementation (e.g. the order in which arguments to a function call are evaluated).

## 3.5 Statements

In addition to expressions, imperative programming languages also have *statements*, which specify some action to be carried out but do not produce a value. Thus, a statement is *executed* rather than evaluated. Statements usually modify the state of a program or the underlying system. These modifications are called *side effects*.

The syntax of a language determines what constitutes a statement. In the C family of languages, a simple statement is terminated by a semicolon, while in Python, a newline terminates a simple statement. The following are examples of simple statements in C++:

```
x + 1;
x = 3;
foo(1, 2, 3);
a[3] = 4;
return 2;
break;
goto some_label;
```

Languages also provide syntax for constructing *compound statements* out of simpler statements and expressions. In C-like languages, a *block* is a compound statement composed of a set of curly braces surrounding a suite of zero or more statements:

```
{
  int x = 10;
  int y = x + 3;
  cout << x << " " << y << endl;
}
```

Conditionals and loops are also compound statements, whether they have a block or just an individual statement as a body.

Some languages make a distinction between statements, declarations, and definitions, since the latter two may not be executed at runtime. A *declaration* introduces a name into a program, as well as properties about the entity it refers to, such as whether it refers to a function or data and what its type is. A *definition* additionally specifies the data or code that the name refers to. In Java, every declaration is also a definition, so the two terms are often used interchangeably. In C and C++, however, a declaration need not be a definition, as in the following:

```
extern int x;
void foo(int x, int y, int z);
class SomeClass;
```

Python does not have declarations, and definitions are statements that are executed.

CHAPTER

# FOUR

# NAMES AND ENVIRONMENTS

Names are the most fundamental form of abstraction, providing a mechanism to refer to anything from simple data values, to complex sets of data and behavior in object-oriented programming, to entire libraries in the form of modules.

An important principle is that the *scope* of a name, or region in which the name maps to a particular entity, should have a restricted context. For example, if a name defined within the implementation of one function or module were to cause a conflict with a name defined in another function or module, abstraction would be violated, since implementation details affect outside code:

```
void foo() {
  int x;
}

void bar() {
  double x;
}
```

Here, even though the name `x` is repeated, each introduction of the name `x` should have a context that is restricted to the individual functions.

Scope is a feature of source code, and it determines what entity a name refers to within the source code. If the name refers to an object whose value is not known until runtime, then the program must defer part of the lookup process until runtime. The mapping of names to objects in each scope region is tracked in a data structure called a *frame* or *activation record*. The collective set of contexts active in a program is called the *environment*. A name is *bound* to an object in a frame or environment if the frame maps that name to the object.

Names that do not map to objects are generally not tracked in activation records. Instead, the compiler or interpreter can determine the entity that the name refers to from the source code itself. However, due to the strong connection between scope regions and frames, we often discuss the name-resolution process in the context of frames, even if the actual lookup process happens at compile time.

Though a name is used as an abstraction for an entity, the name itself is distinct from the entity it names. In particular, the same name can refer to different entities in different code contexts, as in the example above. A single entity may also have multiple names that refer to it, as in the following C++ code:

```
int main() {
  int x = 3;
  int &y = x;
  y = 4;
  cout << x; // prints 4
}
```

In this example, both `x` and `y` refer to the same object at runtime, so they alias each other.

Similarly, the same name can refer to different objects in different *runtime* contexts:

```
int baz(int x) {
  int y = x + 1;
  return y;
}

int main() {
  cout << baz(3) << endl;
  cout << baz(4) << endl;
}
```

The names `x` and `y` defined in `baz()` refer to distinct pairs of objects, with their own lifetimes, within the context of the two separate calls to `baz()`.

Every language defines a set of built-in names that are available to the programmer at program start. These include names for primitive types, built-in functions or modules, and pre-defined constants. A user can also introduce a name through a declaration or definition, as discussed in the previous section.

It is also important to note that names are not actually necessary to do computation. In fact, all programs could be written without names (as with *Turing machines*). Names, however, provide an abstraction that is easily used and incredibly useful for the programmer.

## 4.1 Blocks

Blocks are a fundamental unit of program organization common to most languages. A *block* is a section of program text that contains name bindings that are local to the block. Thus, a block corresponds to a frame in the environment.

Languages generally have two types of blocks: a block that corresponds to the body of a function, and an inline block that is not the body of a function but is nested in another block. Some languages, such as Python and Pascal, do not have inline blocks that contain their own bindings.

The syntax that introduces a block depends on the language, though a common feature is separate syntax that indicates the beginning and end of a block. For example, in the ALGOL family, a block starts with `begin` and ends with `end`, while in the C family, left and right braces indicate the start and end of a block. An interesting case is the Lisp family, including Scheme, which has special `let` constructs to introduce a frame:

```
(let ((x 3) (y 4))
  (display (+ x y))
  (display (- x y))
)
```

This code first binds `x` to 3 and `y` to 4 and then prints their sum and difference. As we will see later, this is generally implemented by translating the `let` into a function that has parameters `x` and `y`:

```
((lambda (x y)
   (display (+ x y))
   (display (- x y))
 ) 3 4)
```

Here, `lambda` introduces an unnamed function, a concept we will return to later. Thus, Lisp does not actually have inline blocks, as any such blocks are really just function blocks.

Inline blocks are by definition nested inside other blocks, resulting in inner frames that are enclosed by outer frames. This means that the code inside the inner block exists in the context of multiple frames, and a well-defined lookup procedure is required to determine the meaning of a name.

Blocks associated with functions also result in nested frames, but there are complications that arise, so we will defer discussion of them until later.

## 4.2 Name Lookup

We first consider a general rule for how name lookup should proceed in an environment with nested frames. Consider an environment that consists of the frames `(A (B (C)))`, with `B` nested inside of `A` and `C` nested inside of `B`. This can result from code with nested regions of scope, as in the following in C++:

```
int main(int argc, char **argv) {   // frame A
  int x = 3;
  int y = -1;
  if (argc > x) {                   // frame B
    int y = stoi(argv[x]);
    if (y > x) {                    // frame C
      int x = argc;
      int z = y - x;                // which x and y?
      cout << z;
    }
  }
}
```

What should the process be for looking up the name `x` in the context of `C`? If the name `x` exists in only one of the active frames `A`, `B`, or `C`, there is no possibility of ambiguity as to which binding `x` refers to. On the other hand, if `x` is bound in more than one frame, as in the example above, then a decision needs to be made as to which binding it refers to. The standard rule is that lookup prefers the innermost binding. Thus, since `x` is bound in `C`, that binding is preferred even if though it is also bound in `A`. On the other hand, `y` is not bound in `C`, so looking up `y` in the context of `C` prefers the binding in `B` over the one in `A`. Finally, looking up `argc` in `C` finds it in neither `C` nor `B`, so the binding in `A` is used.

Thus, the standard lookup procedure is to search for a name in the innermost frame (or scope) and only proceed to the next one if the name is not found. This process is then recursively applied to that next frame (or scope). We often illustrate this process by drawing links between frames, as in Figure 4.1.

A name is said to be *overloaded* if it refers to multiple entities in the same scope. A language that allows overloading must specify further rules on how the lookup process chooses between the applicable entities. For example, in the case of overloaded function names, the arguments of a function call can be compared to the parameter types of each overload to determine which one is the most appropriate:

```
void foo(int x);
int foo(const string &s);

int main() {
  foo(3);         // calls foo(int x)
  foo("hello");   // calls foo(const string &s)
}
```

In some languages, name lookup takes into account how the name is used in order to disambiguate between entities defined in *different* scopes. For example, the following is valid Java code:

```
class SomeClass {
  public static void main(String[] args) {
    int main = 3;
    main(null);   // recursive call
```

(continues on next page)

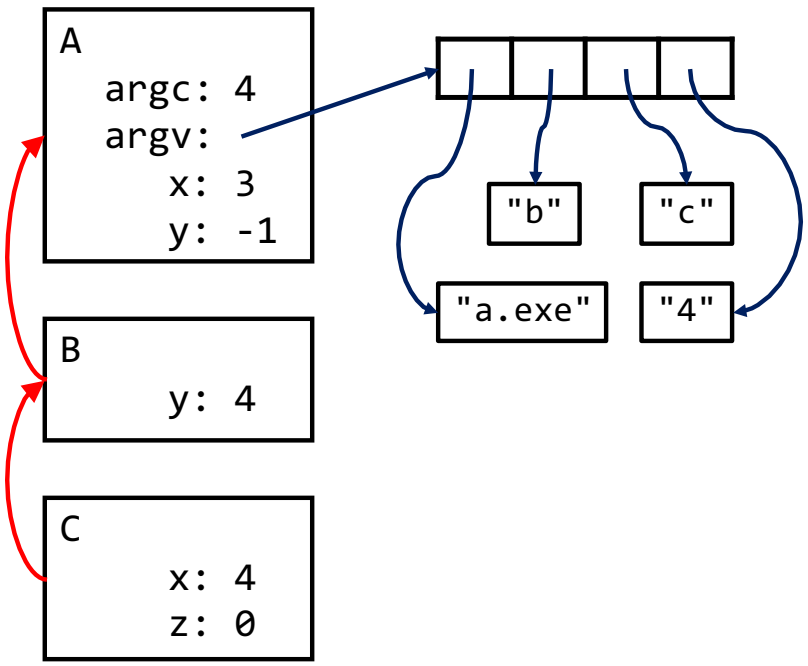



Figure 4.1: An environment corresponding to a set of nested scopes.

(continued from previous page)

```
  }
}
```

Even though the name `main` is defined in the innermost scope to be a variable of type `int`, its use in a function call causes the compiler to look for a function named `main`, disregarding the variable of the same name. A candidate function is found in the enclosing scope, so that is what the name-lookup procedure produces.

## 4.3 Nested Inline Blocks

Now that we have a general rule for looking up names in nested frames, let us consider the environments that correspond to nested inline blocks. Each block corresponds to a frame, resulting in an environment with nested frames. The *visibility rules* of names within nested blocks thus match the general rule discussed above. A name introduced by a block is *visible* within a block nested inside of it, unless the nested block redefines the name. In this case, the former binding is *hidden* or *shadowed* by the latter.

Consider the following example in a C-like language:

```
{
  int x = 0;
  int y = 1;
  {
    int x = 2;
    int z = 3;
  }
}
```

The binding of `x` introduced by the outer block is not visible in the inner block, since the inner block redefines the name

`x`. However, the binding of `y` is visible in the inner block, since `y` is not redefined. Finally, the name `z` is only visible in the inner block, since the outer block is not nested inside the inner.

## 4.4 Scope in Functions

Functions introduce an element of choice that is not present in inline blocks. An inline block is both textually nested inside an outer block, and its execution takes place during the execution of the outer block. On the other hand, the program text in which a function is defined is distinct from the context in which it is called. Consider the following code in a C-like language:

```
int x = 0;

void foo() {
  print(x);
}

void bar() {
  int x = 1;
  foo();
}
```

The function `foo()` is textually located at top-level, or *global*, scope. However, it is called from within the block associated with the function `bar()`. So which `x` is visible within `foo()`, and what value is printed?

Either binding of `x`, and therefore either the value of 0 or 1, is a valid choice depending on the sequence of frames that make up the environment in `foo()`. The two choices are known as static (lexical) scope and dynamic scope, and they are illustrated in Figure 4.2.

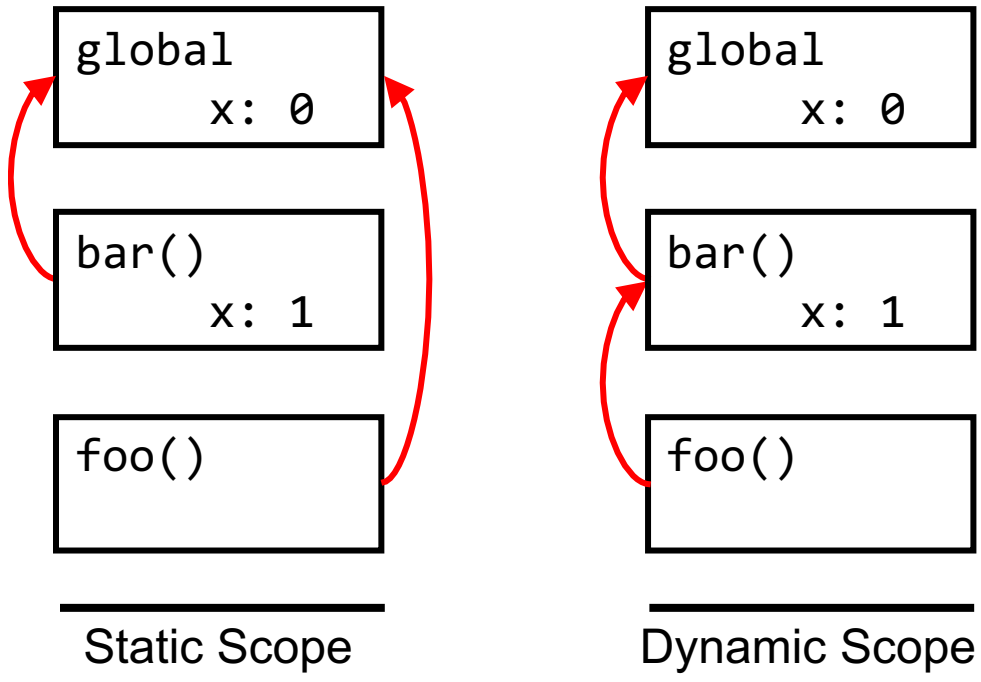



Figure 4.2: Environment structure in static and dynamic scope.

Before considering each of the choices in more detail, let us define some terminology common to both schemes. The *local environment* of a function consists of the subset of the environment that is local to the function. This includes parameter names and all names defined in the context of the function body. The *global environment* consists of names defined at the top-level of a program, either at global or module scope depending on the language. Finally, the *non-local environment* of a function consists of those names that are visible from a function but are neither local to the function nor at global or module scope. It is in what constitutes the non-local environment that static and dynamic scope differ.

For both types, looking up a name follows the general rule we introduced above; the local environment is checked first, followed by the non-local environment, followed by the global environment.

## 4.5 Static Scope

In *static* or *lexical* scope, the environment at any point in a program can be deduced from the syntactic structure of the code, without considering how the computation evolves at runtime. In this scheme, the non-local environment of a function consists of those non-global bindings that are visible in the program text in which the function definition appears.

Considering the example above, the definition `int x = 0` introduces a binding of `x` into the global environment. The definition of `foo()` is located in the context of the global frame, so it has no non-local bindings. Therefore, the binding of `x` that is visible in `foo()` is the one defined at global scope, so the value 0 is printed.

A more interesting case of static scope arises in languages that allow the definition of functions inside other functions. This set of languages includes the Lisp family, Python, Pascal, and to a limited extent, newer versions of C++ and Java. Let's consider a concrete example in Python:

```
x = 0

def foo():
    x = 2

    def baz():
        print(x)

    return baz

def bar():
    x = 1
    foo()()  # call baz()

bar()
```

This program calls the function `baz()` that is defined locally in the context of `foo()`, while the call itself is located in the context of `bar()`. The global environment consists of the binding of `x` to 0 at the top-level, as well as bindings of the names `foo` and `bar` to their respective functions. There are no bindings in the local environment of `baz()`. Static scoping requires that the non-local environment of `baz()` be the environment in which its definition textually appears, which is the environment frame introduced by the function `foo()`. This frame contains a binding of `x` to 2. Following our lookup procedure, the value 2 is printed out since the non-local binding of `x` is the one that is visible.

Figure 4.3 shows a visualization of the environment, as illustrated by Python Tutor.

Since function definitions are statements in Python that bind the given name to a function object, they introduce bindings in the frame in which the function is defined. Python Tutor visualizes the non-local parent of a locally defined function by naming the parent frame and annotating the function with the name of the parent frame, as in `[parent=f2]`. If this is elided, then the parent frame is the global frame. Thus, the non-local environment of the call to `baz()` is the frame for the call to `foo()`, while the parent frame of the latter is the global frame.

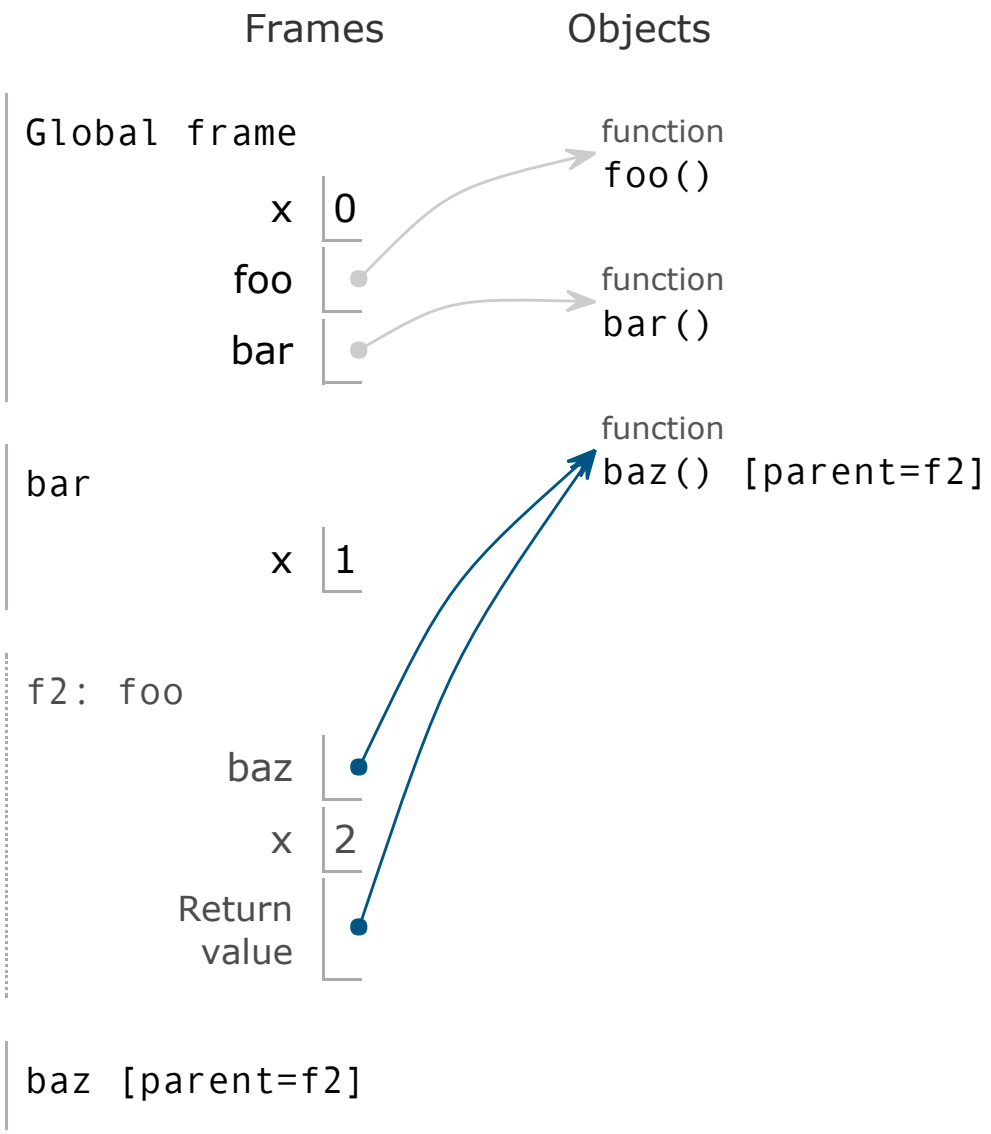



Figure 4.3: Illustration of environment using static scope.

Note that the binding of `x` to 1 introduced by `bar()` does not appear anywhere in the environment of `baz()`, since the definition of `baz()` is not textually located inside of `bar()`.

Most modern languages use static scope, since it tends to be more efficient than dynamic scope, as the lookup process can be facilitated by the compiler. Static scope also generally makes it easier for programmers to reason about the code, since they don't have to trace through the execution in order to figure out what a name refers to.

## 4.6 Dynamic Scope

In *dynamic* scope, the environment at any point in a program is dependent on how execution evolves at runtime. The non-local environment of a function consists of those bindings that are visible at the time the function is called. This rule is applied recursively, so that a sequence of function calls results in a sequence of frames that are part of the non-local environment of the innermost function call.

As a concrete example, consider the following C-like code:

```
int x = 0, y = 1;

void foo() {
  print(x);
  print(y);
}

void bar() {
  int x = 2;
  foo();
}

int main() {
  int y = 3;
  bar();
  return 0;
}
```

The global environment includes the bindings of `x` to 0 and `y` to 1. When execution starts at `main()`, its environment consists of the global frame and the local frame that it introduces that binds `y` to 3. In the call to `bar()`, the environment of `bar()` consists of the global frame, the non-local frame of `main()`, and the local frame of `bar()`. Finally, in the call to `foo()`, the environment of `foo()` consists of the global frame, the non-local frame of `main()`, the non-local frame of `bar()`, and the local frame of `foo()`. Name lookup starts in the innermost frame and proceeds outward until it finds a binding for the name. A binding for `x` is found in the frame of `bar()`, and for `y` in the frame of `main()`, so that the values 2 and 3 are printed.

Dynamic scope can be simpler to implement than static, since the frames in an environment correspond exactly to the set of frames that are active during program execution. However, it can result in behavior that is less obvious from reading the code, as it requires tracing out the runtime execution of the code to understand what it does. As a result, few modern languages use dynamic scope.

Languages that allow functions themselves to be passed as arguments introduce a further complexity when it comes to dynamic scope in the form of *binding policy*. We will defer discussion of binding policy until we examine *higher-order functions*.

## 4.7 Point of Declaration or Definition

The rules we've described thus far do not fully specify name lookup and visibility in languages that allow names to be introduced in the middle of a block. In particular, does the scope of a name start at the beginning of the block in which it is introduced or at the point of introduction? Consider the following C-like code:

```
int foo() {
  print(x);
  int x = 3;
}
```

Is this code valid? The initialization of `x` occurs after the print, so allowing code like this would result in undefined behavior.

The C family of languages avoids this problem by stating that the scope of a name begins at its *point of declaration* and ends at the end of the block in which the declaration appears. Thus, the code above would be a compile-time error. On the other hand, consider the following:

```
int x = 2;

int foo() {
  print(x);
  int x = 3;
}
```

Since the local binding of `x` is not in scope at the print call, the global binding of `x` is visible and the value 2 is printed.

Python, however, does not follow this rule. If a name is defined within a function body, then its scope starts at the beginning of the body. However, it is illegal to reference the name before its initialization. Thus, the following code is erroneous:

```
x = 2

def foo():
    print(x)
    x = 3

foo()
```

This results in an error like the following:

```
UnboundLocalError: local variable 'x' referenced before assignment
```

Suppose the intent of the programmer in the code above was to modify the binding of `x` in the global environment rather than to introduce a new binding in the local frame. Python enables the programmer to specify this intent with the `global` statement:

```
x = 2

def foo():
    global x  # specify that x refers to the global binding
    print(x)
    x = 3
```

(continues on next page)

(continued from previous page)

```
foo()
print(x)
```

The code now prints out the value 2, modifies the global `x` to be bound to 3, and prints out 3. A similar `nonlocal` statement is available to specify that a name refers to a binding in the non-local environment.

A final consideration is how to handle scope in the context of mutually recursive functions or classes. Consider the following code:

```
int foo(int x) {
  return bar(x + 1);
}

int bar(int x) {
  return foo(x - 1);
}
```

Ignoring the fact that the code does not terminate, the scope rules we described for the C family do not permit this code, since `bar()` is not in scope when `foo()` is defined. C and C++ get around this problem by allowing *incomplete* declarations:

```
int foo(int x) {
  int bar(int);  // incomplete declaration of bar
  return bar(x + 1);
}

int bar(int x) {
  return foo(x - 1);
}
```

Java, on the other hand allows methods and classes to be used before they are declared, avoiding the need for incomplete declarations. Similarly, older versions of C allowed functions to be used before declaration, though this was prone to error due to how such uses were handled in the compiler and linker.

## 4.8 Implementation Strategies

A binding is an association between a name and an object, making an associative container such as a dictionary a natural abstraction for keeping track of bindings. A dictionary-based implementation strategy can represent each frame with its own dictionary, as well as a pointer to the next outer frame, if there is one. Adding bindings and looking up names can be done dynamically by inserting new entries into frames at runtime or searching through the list of frames for an entry that matches a given name.

Static languages often take a more efficient approach of translating a name to an offset in a frame at compile time. This strategy requires static scope so that names can be resolved to frames by the compiler. As an example, consider the following code written in a C-like syntax, but with nested function definitions:

```
int foo(int x) {
  double y = x + 3.1;

  double bar(double x) {
    return x - y;
  }
```

(continues on next page)

(continued from previous page)

```
  return bar;
}

foo(3)(4);  // evaluates to -2.1
```

A compiler can examine the code in `foo()` to determine how much space its activation record requires, factoring in parameters, local variables, temporaries, and control data. It can then associate each variable with a specific offset in the activation record, as in Figure 4.4.

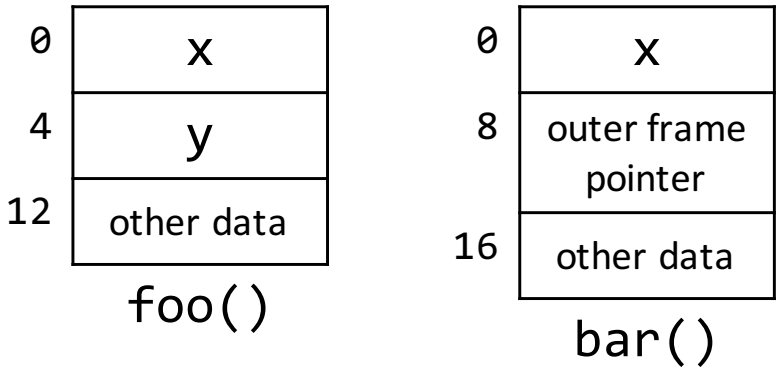



Figure 4.4: An offset-based layout scheme associates fixed offsets with individual pieces of data.

The value of `x` in the scope of `foo()` is stored at offset zero from the beginning of the activation record, while the value of `y` is stored at offset four. In the activation record for `bar()`, its parameter `x` is stored at offset zero, while a pointer to the invocation's next outer frame is stored at offset eight. (Alternatively, a direct pointer to the memory location for `y` can be stored, rather than a pointer to the activation record containing `y`.) Figure 4.5 shows the actual activation records created by the invocations `foo(3)(4)`.

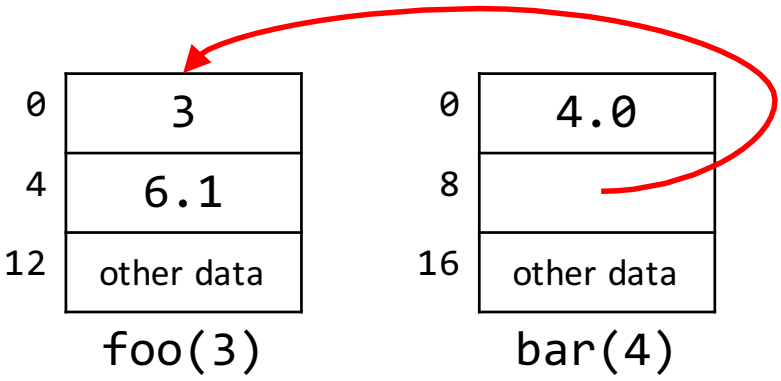



Figure 4.5: Data stored using an offset-based layout.

When the compiler generates code for the body of `bar()`, the reference to `x` is translated to an offset of zero into the activation record for `bar()`, while the reference to `y` is translated into first retrieving the outer frame pointer from offset eight in the activation record for `bar()`, followed by an offset of four in the outer frame. Thus, the values 4.0 and 6.1 are retrieved for `x` and `y`, respectively, resulting in a difference of -2.1.

The offset-based implementation requires only a single memory access for a local variable, as opposed to a dictionary

lookup in a dictionary-based implementation. For a local variable in the $n$th other frame, an offset-based strategy requires $n$ memory accesses, while a dictionary-based scheme does $n$ dictionary lookups. A memory access is likely to be much more efficient than a dictionary lookup, resulting in better performance for offset-based implementations.

CHAPTER

FIVE

# CONTROL FLOW

We now turn our attention to the problem of managing the sequence of actions that take place in a program. Sequencing is of particular importance in imperative programming; in this paradigm, each programming construct specifies some action to be taken, and the flow of control between constructs is instrumental to the meaning of a program.

## 5.1 Expression Sequencing

As we saw in *Expressions*, the order in which subexpressions are evaluated is a consideration in the evaluation of a compound expression, though a well-defined order is most important in languages that allow expressions to have side effects. Here, we consider some cases in which the evaluation semantics are of particular importance.

### 5.1.1 Short Circuiting

Consider a conditional of the following form in C++:

```
if (x != 0 && foo(x)) {
  ...
}
```

If the order of evaluation of the operands to the `&&` operator were left up to the implementation, it would be legal to evaluate the call to `foo()` on the right-hand side before the comparison with 0 on the left-hand side. This is problematic in two cases. First, if `foo()` requires that its argument is non-zero, such as in the case that it uses the argument as a divisor, then its evaluation can lead to a runtime error or, even worse, undefined behavior. Second, if `foo()` performs a very expensive computation, then it would be unnecessarily computed in the case that `x` is 0.

To address these problems, boolean operators in many languages evaluate their left-hand operand before the right-hand one and are also *short circuiting*. This means that the right-hand side is not computed if the overall value of the expression can be determined from the left-hand side alone. This is the case in conjunction (logical and) if the left-hand side evaluates to a false value, and in disjunction (logical or) if it evaluates to a true value.

A similar situation occurs with ternary conditional operators, such as `?:` in the C family:

```
int y = (x != 0 ? z / x : 0);
```

Here, if `x` is 0, the the second operand is not computed, and `y` is set to 0. On the other hand, if `x` is not 0, then the second operand is computed but not the third, so `y` is set to the value obtained by dividing `z` by `x`.

### 5.1.2 Explicit Sequences

Some languages provide an explicit mechanism for chaining expressions in an ordered sequence. Generally, the result of the expression sequence as a whole is the result of the last expression in the sequence. In C and C++, the comma operator sequences expressions in this manner:

```
int x = (3, 4);
cout << x;
```

This prints out the value 4, since the expression `3, 4` evaluates to 4. Similarly, in the Lisp family, the `begin` form chains expressions together:

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

### 5.1.3 Compound Assignment

In the evaluation of compound-assignment operators, the number of times the left-hand side is evaluated can affect the result in the presence of side effects. In most languages with compound assignment, the following two operations are **not** equivalent in general:

```
x += 1
x = x + 1
```

The difference is that in the first case, the expression `x` is only evaluated once, while in the second, it is evaluated twice. As a concrete example of where the results differ, consider the following Python code:

```
def foo(values):
    values.append(0)
    return values

mylist = []
foo(mylist)[0] += 1
```

This results in `mylist` being equal to `[1]`. On the other hand, consider the following:

```
mylist = []
foo(mylist)[0] = foo(mylist)[0] + 1
```

Here, `mylist` ends up equal to `[1, 0]`. Thus, the two operations are not equivalent.

## 5.2 Statement Sequences

Statements by their very nature generally have side effects, so their order of execution is of fundamental importance in imperative programming. Imperative languages generally specify that statements execute in the order in which they appear in the program text[1].

Sequences of statements are often grouped in the form of *blocks*, which can appear in contexts where a single statement is expected. Some languages, such as Python, restrict where a sequence of statements can appear, such as the body of a structured control statement. Python uses the term *suite* for such a sequence rather than *block*.

[1] The compiler or interpreter can reorder operations if it can prove that the reordered execution is semantically equivalent to the original sequence. In single-threaded programs, this reordering is generally not observable, but it can have tangible effects in parallel programs. However, we will not discuss the details here.

A language's syntax specifies how statements are separated in a block or a sequence. Two common strategies are to use a separator character between each statement, or to require that all statements be terminated by a particular character. For example, if a semicolon is used to separate statements, a sequence of statements could have the following structure:

```
S_1; S_2; ... ; S_N
```

On the other hand, if a semicolon is used to terminate the statements, the sequence would have the following form:

```
S_1; S_2; ... ; S_N;
```

The key difference is that in the second case, the last statement would require a terminating semicolon.

## 5.3 Unstructured Transfer of Control

Many languages provide a simple mechanism for transferring control in the form of a *goto*. This is generally used in conjunction with a label that specifies which statement is to be executed next. For example, the following C code prints integers in sequence starting at 0:

```
int x = 0;
LOOP: printf("%d\n", x);
x++;
goto LOOP;
```

The code initializes `x` to 0 and proceeds to print it out. It then increments `x` and transfers control back to the print statement.

Goto statements are a very low-level mechanism of control, usually mapping directly to a direct jump instruction in machine code. However, on their own, simple gotos are insufficient to implement most algorithms since they do not provide any branching. The example above is an infinite loop and also suffers from integer overflow, resulting in the values wrapping around. In some languages, variants of goto exist that do provide branching capability, such as computed goto in older versions of FORTRAN. Machine code often provides branching through the use of indirect jump instructions.

While the various forms of goto are very powerful, they are also open to abuse, resulting in incomprehensible *spaghetti code* that makes it difficult to follow the control flow in a program. Part of the problem is that this unstructured form of transferring control is not amenable to conventions for improving readability, such as indentation. In the example above, all statements occur at the same level, and it is not visually obvious where the loop is. This is even more of a problem when the goto is many lines away from the label that it references. And if a piece of code has many labels and many gotos, drawing out the set of possible paths through the code can result in a mess, resembling a plate of spaghetti.

Another problem with goto is how to handle the initialization or destruction of local variables when control passes into or out of a block. We will see more details about initialization and destruction shortly, but languages such as C++ with complicated initialization and destruction semantics often place restrictions on how goto can be used.

While goto is very powerful, it is not necessary for any algorithm. As a result, it is common practice to discourage the use of gotos, and some languages do not include it in their set of control constructs.

There are a few cases, however, where goto or a restricted version of it can result in simpler and more readable code. However, an example must wait until after we discuss structured control constructs.

## 5.4 Structured Control

Modern languages provide higher-level control constructs than goto, allowing code to be structured in a more readable and maintainable way. The most basic constructs are those for expressing conditional computation and repetition, two features required for a language to be *Turing complete*, meaning that the language is equivalent in computational power to Turing machines.

### 5.4.1 Conditionals

We have already seen the ternary conditional operator provided by some languages for conditional evaluation of expressions. Imperative languages provide an analogous construct for conditional execution of statements in the form of the *if* statement, which has the general form:

```
if <test> then <statement1> else <statement2>
```

Here, `<test>` is an expression that has a boolean value; depending on the language, this expression may be required to be of the boolean type, or the language may allow conversions of other types to a boolean value. If the resulting value is true, then `<statement1>` is executed. Otherwise, `<statement2>` is executed.

Often, languages allow the else branch to be elided:

```
if <test> then <statement>
```

However, this can lead to the *dangling else* problem. Consider the following example:

```
if <test1> then if <test2> then <statement1> else <statement2>
```

The grouping of the branches can be interpreted as either of the following:

```
if <test1> then (if <test2> then <statement1> else <statement2>)
if <test1> then (if <test2> then <statement1>) else <statement2>
```

Some languages resolve this ambiguity by specifying that an else belongs to the closest `if`. Others formulate their syntax to avoid this problem by explicitly indicating where a branch starts and ends.

Another common language feature is to provide a *cascading* form of `if`. The following is an example in C:

```
if (<test1>) <statement1>
else if (<test2>) <statement2>
...
else if (<testN>) <statementN>
else <statementN+1>
```

As another example, Python also provides an equivalent form, but with the keyword `elif` rather than `else if`.

A cascading `if` acts as a conditional with more than two branches. Though it can always be rewritten as a sequence of nested `if` statements, the cascaded form can improve readability by making it visually clear what the disjoint branches are.

A similar, though often more restricted, form of multiple branching is provided by the *case* or *switch* statement. It has the following general form:

```
switch <expression>:
  case <value1>: <statement1>
  case <value2>: <statement2>
```

(continues on next page)

(continued from previous page)

```
...
case <valueN>: <statementN>
default: <statementN+1>
```

The `switch` expression is evaluated, and then its value is compared to those specified in the `case` branches. If the value matches one of the branches, then that branch is executed. If the value does not match the value in any `case` branch, then the `default` branch is executed.

There are many variations in both the syntax and the exact semantics of a `switch` statement. Usually, the values in the `case` branches must be compile-time constants, restricting the set of types that the `switch` expression may have. Some languages allow multiple alternative values to be specified for a single `case`. Depending on the language, execution of the `case` branches may be disjoint, or execution from one branch "falls" into the next branch unless an explicit `break` or `goto` is present. Often, the `default` branch may be elided. In some languages, such as Swift, eliding the `default` clause requires the combination of the `case` branches to cover all possible values that can be provided to the `switch`.

Part of the motivation for providing separate `if` and `switch` statements is that the latter often can be implemented more efficiently. More importantly, however, is that the two constructs are more suitable for different situations. The `switch` statement is ideal for when execution can follow multiple discrete paths based on the value of an expression that isn't necessarily true or false, while the `if` statement is appropriate if the flow of execution is determined by a set of boolean conditions.

### 5.4.2 Loops

Loops are a common mechanism for repetition in imperative languages. They allow a programmer to specify that a computation should repeat either a certain number of times, or until some condition is met.

Some languages provide loop constructs that repeat for a bounded number of iterations determined at the beginning of the loop. Such a construct is actually insufficient to express all algorithms, so languages that only provide bounded iteration, without some other mechanism such as unbounded loops or gotos, are not Turing complete.

The most general form of unbounded iteration is the `while` loop:

```
while <expression> do <statement>
```

Such a loop tests the expression to see if it is true, and if so, executes the statement and repeats the process.

There are many variations on `while` loops. Some languages have a form similar to:

```
do <statement> until <expression>
```

This repeatedly executes a statement until a condition is met. Another variant is the `do while` loop:

```
do <statement> while <expression>
```

This is the same as `do until`, except that the control expression is negated. In both forms, the statement is executed at least once, while a standard `while` loop need not execute its body.

While the `while` loop and its variants are general enough to express any form of repetition, it is common enough to iterate through a sequence that languages often provide syntactic sugar to facilitate the expression of such loops. The `for` loop in the C family of languages is one example:

```
for (<initialization>; <test>; <update>) <statement>
```

This is, ignoring scope and lifetime details, mostly equivalent to:

```
<initialization>;
while (<test>) {
  <statement>
  <update>
}
```

Another, more abstract, type of loop is a *foreach* loop that iterates through the elements in a sequence, with the compiler inferring the initialization, test, and update. Such a loop may also be called a *range-based for loop*. The following is an example in C++11:

```
template <typename Container>
void print_all(const Container &values) {
  for (auto i : values) {
    cout << i << endl;
  }
}
```

The function `print_all()` iterates through all the values in any container that supports the iterator interface and prints out each value. The Python `for` loop provides a similar abstraction.

### 5.4.3 Loop Termination

Normally, a loop terminates when the specified condition no longer holds, or in the case of foreach loops, when the elements of the sequence are exhausted. However, certain algorithms can be better expressed if a loop can be explicitly terminated in the middle of its execution. An example is the following C++ function that determines if a particular value is in an array:

```
bool contains(int *array, size_t size, int value) {
  for (size_t i = 0; i < size; i++) {
    if (array[i] == value) {
      return true;
    }
  }
  return false;
}
```

Once a value is found in the array, it is no longer necessary to examine the remaining elements of the array, so the function returns immediately rather than waiting for the loop to terminate normally.

For the cases where an early termination is desired without immediately returning, a `goto` may be used in a language that provides such a construct. For example:

```
bool found = false;
for (size_t i = 0; i < size; i++) {
  if (array[i] == value) {
    found = true;
    goto end;
  }
}
end: cout << "found? " << found << endl;
```

However, as it is considered desirable to avoid `goto` wherever possible, many languages provide a restricted `break` statement that explicitly exits a loop and proceeds to the next statement:

```
bool found = false;
for (size_t i = 0; i < size; i++) {
  if (array[i] == value) {
    found = true;
    break;
  }
}
cout << "found? " << found << endl;
```

A related construct is `continue`, which merely ends the current loop iteration rather than exiting the loop entirely.

The simple `break` and `continue` statements suffice when a single loop is involved. What if, on the other hand, we have nested loops, such as the following:

```
for (...) {
  for (...) {
    if (...) break;
  }
}
```

Which loop does the `break` statement terminate? As with dangling else, generally the innermost loop is the one that is terminated. If we wish to terminate the outer loop, however, we are forced to use a `goto` in C and C++:

```
for (...) {
  for (...) {
    if (...) goto end;
  }
}
end: ...
```

Java address this problem by allowing loops to be labeled and providing forms of `break` and `continue` that take a label:

```
outer: for (...) {
  for (...) {
    if (...) break outer;
  }
}
```

Some languages, such as Python, do not provide a specific mechanism for terminating or continuing an outer loop and require code to be refactored in such a case.

## 5.5 Exceptions

Exceptions provide a mechanism for implementing error handling in a structured manner. They allow the detection of errors to be separated from the task of recovering from an error, as it is often the case that the program location where an error occurs doesn't have enough context to recover from it. Instead, an exception enables normal flow of execution to be stopped and control to be passed to a handler that can recover from the error.

In general, languages with exceptions provide:

1. A syntactic construct for specifying what region of code a set of error handlers covers.

2. Syntax for defining error handlers for a particular region of code and specifying the kinds of exceptions they handle.
3. A mechanism for *throwing* or *raising* an exception.

Some languages also provide a means for defining new kinds of exceptions. For example, in Java, an exception must be a subtype of `Throwable`, in Python, it must be a subtype of `BaseException`, and in C++, it can be of any type

An exception may be thrown by the runtime while executing a built-in operation, such as dividing by zero. It may also be raised directly by the user, with syntax similar to the following:

```
throw Exception();
```

This consists of a keyword such as `throw` or `raise` indicating that an exception is to be thrown, as well as the exception value to be thrown. Some languages, such as Python, allow an exception class to be specified instead of an instance.

The code that throws an exception may be in a different function than the code that handles it. Exception handlers are **dynamically scoped**, so that when an exception is raised, the closest set of active handlers on the dynamic call stack handles the exception. If that group of handlers does not handle exceptions of the type that was thrown, then the next set of handlers on the call stack is used. If the call stack is exhausted without finding an appropriate handler, execution terminates.

The following is an example in Python:

```
def average_input():
    while True:
        try:
            data = input('Enter some values: ')
            mean = average(list(map(float, data.split())))
        except EOFError:
            return
        except ValueError:
            print('Bad values, try again!')
        else:
            return mean

def average(values):
    count = len(values)
    if count == 0:
        raise ValueError('Cannot compute average of no numbers')
    return sum(values) / count

average_input()
```

The `try` statement indicates the block of code for which it defines error handlers. If an exception is raised during execution of the following suite, and that exception is not handled by a `try` statement further down in the execution stack, then this `try` statement attempts to handle the exception. The `except` headers and their associated suites define the actual exception handlers, indicating what kinds of exceptions they can handle. When an exception is raised in the `try` suite, the type of the exception is compared against the `except` clauses in sequence, and the first one that can handle an exception of that type is executed. Thus, only one handler is actually run. The `else` clause, if present, only executes if no exception is raised in the `try` clause.

In this particular example, an exception may be raised by the built-in `float()` constructor, if the user enters a value that does not correspond to a `float`. In this case, a `ValueError` is raised, and the second `except` clause is executed. If the user enters no values, then `average()` will directly throw a `ValueError`. Since the `try` statement in `average_input()` is the closest exception handler on the execution stack, it is checked for an `except` clause that handles `ValueErrors`, and the second clause runs. Another case is if the input stream ends, in which case an `EOFError`

is raised, resulting in execution of the first `except` clause. Finally, if the user enter one or more valid values, then no exception is raised, and the `else` clause executes, returning the mean.

Python also allows a `finally` clause to be specified, with code that should be executed whether or not an exception is raised. Languages differ in whether they provide `finally` or `else` clauses. For example, Java provides `finally` while C++ has neither.

Exceptions introduce new control paths in a program, and some algorithms make use of them for things other than error handling. For example, in Python, iterators raise a `StopIteration` exception when the sequence of values they contain is exhausted. Built-in mechanisms like `for` loops use such an exception to determine when the loop should terminate.

CHAPTER

SIX

# MEMORY MANAGEMENT

Programs operate on data, which are stored in memory. In general, the set of data in use in a program can differ over time, and the amount of storage required by a program cannot be predicted at compile time. As a result, a language and its implementation must provide mechanisms for managing the memory use of a program.

As mentioned in *Entities, Objects, and Variables*, a data object has a lifetime, also called a *storage duration*, during which it is valid to use that object. Once an object's lifetime has ended, its memory may be reclaimed for use by other objects. Languages differ from those in which the user is primarily responsible for managing memory to languages where the compiler (or interpreter) and runtime bear the sole responsibility of memory management.

In languages that allow a user to manually manage the memory of objects, many programming errors result from incorrectly managing memory. These errors include *memory leaks*, where a programmer neglects to release memory that is no longer needed, and *dangling references*, where an object is still accessible to a program even though the user has marked the object as dead. Errors relating to memory management can be particularly difficult to detect and debug, since the resulting behavior depends on the complex interplay between the program and the runtime storage manager and can be different in separate runs of the program.

There are several strategies that reduce the possibility of errors related to memory management. This usually involves moving the role of managing memory from the programmer to the language and its implementation. Specific examples include tying an object's lifetime to the scope of a variable that references it, and to provide automatic memory management of objects that are not directly linked with variables.

## 6.1 Storage Duration Classes

Many languages make distinctions between the storage duration of different objects. This can based on the type of the object, where its corresponding variable is declared, or manually specified by a programmer. Common storage duration classes (using C++ terminology) include static, automatic, thread-local, and dynamic.

### 6.1.1 Static Storage

Variables declared at global scope can generally be accessed at any point in a program, so their corresponding objects must have a lifetime that spans the entire program. These objects are said to have *static* storage duration. In addition to global variables, static class member variables usually also have static storage duration in object-oriented languages. Some languages, such as C and C++, also allow a local variable to be declared with static storage duration, in which case the corresponding object is shared among all calls to the associated function.

Since the compiler or linker can determine the set of objects with static storage duration, such objects are often placed in a special region of memory at program start, and the memory is not reclaimed during execution. While the **storage** is pre-allocated, some languages allow the **initialization** of such objects to be deferred until their first use.

### 6.1.2 Automatic Storage

Objects associated with local variables often have *automatic* storage duration, meaning they are created at the start of the variable's scope and destroyed upon final exit from the scope. As we saw in *Blocks*, in many languages, a block is associated with its own region of scope. Most languages create a new *activation record* or *frame* upon entry to a block to store the local objects declared in the block. This frame is usually destroyed when execution exits the block. It is not destroyed, however, when control enters a nested block or a function call, since control will return back to the block.

Many languages store activation records in a stack structure. When execution first enters a block, its activation record (or stack frame) is pushed onto the stack. If control passes to a nested block or called function, a stack frame corresponding to the new code is pushed on the stack, and execution passes to that code. When execution returns to the original block, the new stack frame is popped, and the activation record for the original block is again at the top of the stack. When this block completes, its activation record is popped off, and the local objects contained within are destroyed.

As we will see later, languages that implement full closures for nested function definitions cannot always discard a frame upon exit from a block, since a nested function may require access to the variables declared in that block. These languages do not place frames that may be needed later in a stack structure. Instead, they manage frames by detecting when they are no longer in use and reclaiming them.

### 6.1.3 Thread-Local Storage

Languages that include multithreading often allow variables to be declared with *thread-local* storage duration. The lifetime of their respective objects matches the duration of execution of a thread, so that a thread-local object is created at the start of a thread and destroyed at its end.

Since multiple threads execute concurrently, each thread needs its own stack for automatic objects and its own memory region for thread-local objects. These structures are created when a thread begins and are reclaimed when a thread ends.

### 6.1.4 Dynamic Storage

Objects whose lifetimes are not tied to execution of a specific piece of code have *dynamic* storage duration. Such objects are usually created explicitly by the programmer, such as by a call to a memory-allocation routine like `malloc()` or through an object-creation mechanism like `new`. While creation of dynamic objects is usually an explicit operation, languages differ in whether the programmer controls destruction of dynamic objects or whether the runtime is responsible for managing their memory.

Languages with low-level memory-management routines such as `malloc()` generally have a corresponding `free()` call that releases the memory allocated by a call to `malloc()`. The user is responsible for calling `free()` on an object when it is no longer needed.

Some languages with explicit object-creation mechanisms such as `new` provide an explicit means for object-destruction, such as `delete` in C++. As with `malloc()` and `free()`, the programmer is responsible for applying `delete` to an object when it is no longer in use.

Other languages manage the destruction of objects automatically rather than relying on the programmer to do so. These languages implement *garbage collection*, which detects when objects are no longer in use and reclaims their memory. We will discuss *garbage collection* in more detail later in this text.

Since the lifetimes of dynamic objects are not tied to a particular scope and their destruction need not occur in an order corresponding to their construction, a stack-based management scheme is insufficient for dynamic objects. Instead, dynamic objects are usually placed in a memory region called the *heap*; the language implementation manages the storage resources in the heap. We will not discuss techniques for heap management here.

## 6.2 Value and Reference Semantics

Languages differ as to whether the storage for a variable is the same as the object it refers to, or whether a variable holds an indirect reference to an object. The first strategy is often called *value semantics*, and the second *reference semantics*.

To illustrate the distinction between value and reference semantics, we first examine the semantics of variables in C++. In C++, declaring a local variable creates an object on the stack, and the object has automatic storage duration. Within the scope of the variable, it always refers to the same object. Consider the following code:

```
int x = 3;
cout << &x << endl;
x = 4;
cout << &x << endl;
//Note: x's memory address does not change, even when its value does
```

The declaration of `x` creates an association between the name `x` and a new object whose value is initialized to 3. Thereafter, as long as `x` remains in scope, it always refers to that same object. The assignment `x = 4` copies the value from the right-hand side into the object named by the left-hand side, but it does not change which object `x` refers to. This can be seen by noting that the address of `x` remains the same before and after the assignment. Thus, the storage for the variable `x` is always the same as the object it refers to. We therefore say that C++ has value semantics.

C++ also has a category of variables called *references*, which do not have the semantics of allocating memory when they are created. Instead, they share memory with an existing object. Consider the following:

```
int x = 3;
int &y = x;
//The following two lines will print the same memory address
cout << &x << endl;
cout << &y << endl;
y = 4;
cout << x << endl; //This prints 4
```

In this code, the declaration of `x` creates a new object and initializes it to 3. The declaration of `y` as a reference does not create a new object. Instead, `y` refers to the same memory as `x`, as can be seen by examining their respective addresses. Assigning to `y` changes the value stored in the memory that `y` refers to, and subsequently examining `x` shows that its value also changed, since it shares memory with `y`. Figure 6.1 is an illustration of what this looks like in memory.

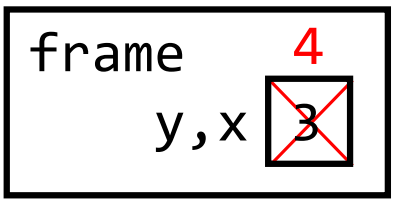



Figure 6.1: A reference in C++ refers to the same memory location as an existing object.

Finally, C++ has *pointers*, which are objects that store the address of another object. A pointer indirectly refers to another object, and dereferencing the pointer obtains the object it is referring to:

```
int x = 3;
int *y = &x;
*y = 4;
cout << x << endl;
```

This code creates a pointer that holds the address of `x` and then dereferences it to change the value of the corresponding object. Figure 6.2 illustrates this in memory.

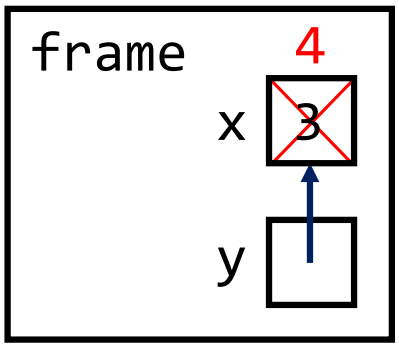



Figure 6.2: A pointer refers to an object indirectly by storing the address of that object.

Pointers refer to objects indirectly, so they provide a form of reference semantics. And since they refer to objects indirectly, it is possible to change which objects they refer to after creation:

```
int x = 3;
int y = 4;
int *z = &x;
z = &y;
*z = 5;
cout << x << ", " << y << endl;
```

In this code, the pointer `z` originally holds the address of `x`, so it indirectly refers to the object associated with `x`. The value of `z` is then modified to be the address of `y`, so `z` now indirectly refers to the object associated with `y`. Dereferencing `z` and modifying the resulting object now changes the value of `y` instead of that of `x`. This ability to change which object a pointer refers to is different than the direct association between names and objects provided by normal C++ variables and references, which cannot be broken while the name is in scope.

In a language with reference semantics, variables behave in the same manner as C++ pointers. In most cases, the variable is allocated on the stack but indirectly refers to a dynamic object located on the heap. Thus, the variable has storage that is distinct from the object it is referencing. This indirect reference can be represented by an address as in C++ pointers or through a similar mechanism, allowing the association between variables and the objects they reference to be changed.

As an example of reference semantics, consider the following Python code:

```
>>> x = []
>>> y = x
>>> id(x)
4546751752
>>> id(y)
4546751752
```

The variable `x` is bound to a new list, and then `x` is assigned to `y`. The `id()` function returns a unique identifier for an object, which is actually the address of the object in some implementations. Calling `id()` on `x` and `y` show that they refer to the same object. This differs from non-reference variables in C++, which never refer to the same object while they are in scope. Figure 6.3 is a representation of the Python program in memory.

Now consider the following additional lines of code:

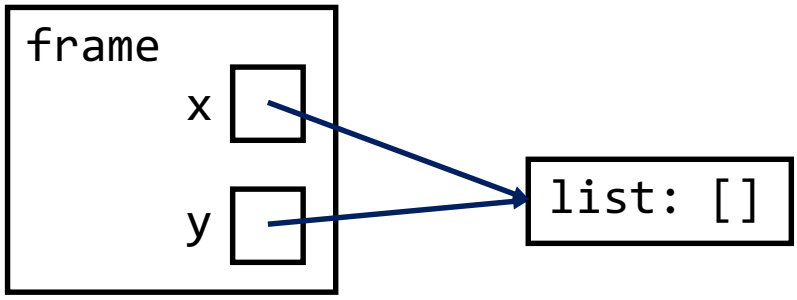



Figure 6.3: In reference semantics, variables indirectly refer to objects on the heap.

```
>>> x = []
>>> id(x)
4546749256
>>> id(y)
4546751752
```

Assigning a new list to `x` changes which object `x` is bound to, but it does not change which object `y` is bound to. This differs from C++-style references, which cannot change what object they refer to. Instead, the behavior is analogous to the following pseudocode with C++-style pointers:

```
list *x = new list();
list *y = x;
x = new list();
cout << x << ", " << y << endl;
```

The result in memory is shown in Figure 6.4.

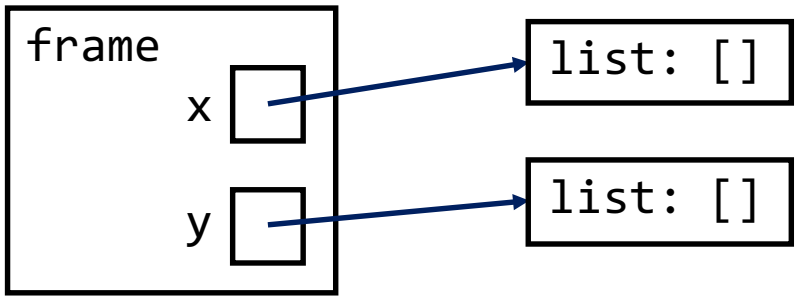



Figure 6.4: In reference semantics, assigning to a variable changes which object it refers to rather than the value of the object itself.

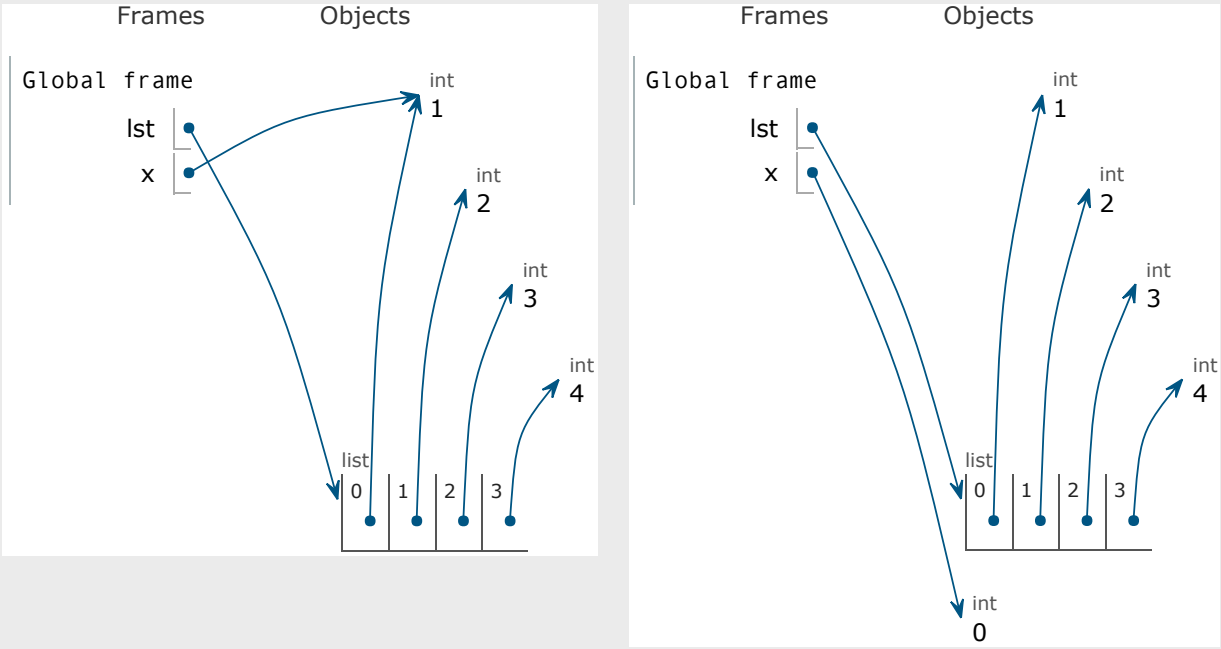
The examples above illustrate the key difference between value and reference semantics: In value semantics, assignment to a variable changes **the value** of the object that the variable refers to. In reference semantics, however, assignment to a variable changes **which** object the variable refers to. The latter can be seen in the following Python example:

```
>>> lst = [1, 2, 3, 4]
>>> for x in lst:
        x = 0
>>> lst
[1, 2, 3, 4]
```

The assignment to `x` in the loop changes which object `x` refers to rather than the value of the object, as illustrated by

Python Tutor in Table 6.1.

Table 6.1: Assignment to a loop variable in Python changes the object the variable is pointing to, rather than its value.




The left side of Table 6.1 shows the environment at the start of the first iteration, after `x` is bound to an element of the list but before it is assigned to 0. Executing the assignment results in the right-hand side, with `x` rebound but the list element unaffected. Thus, none of the values in the list are modified.

As can be seen from the previous examples, C++ has value semantics while Python has reference semantics. Java, on the other hand, has value semantics for primitive types but reference semantics for `Object` and its subclasses, which are often called *object types* or *reference types*.

## 6.3 RAII and Scope-Based Resource Management

Programs often make use of complex data abstractions whose implementations allocate memory for their own internal use. An example is a growable array, such as the `vector` template in C++ or the `list` type in Python. These data structures use a contiguous piece of memory to store elements. When a structure runs out of space, it must allocate a new region of memory, copy (or move) over the elements, and release the old memory region. This operation is hidden behind the abstraction barrier of the data structure, and the growable array's implementation handles its own memory management.

For languages with garbage collection, an object that internally allocates memory does not pose any problems in most cases. If the object is no longer in use, the garbage collector can usually detect that the memory it allocated is also no longer in use. In languages without garbage collection, however, other mechanisms must be used in order to manage internal resources.

A simple solution is for the interface of a data structure to include a function that must be explicitly called when the structure is no longer needed, with a name along the lines of `close()`, `release()`, or `destroy()`. This is called the *dispose pattern*, and it is well-suited to languages where it is idiomatic to deallocate objects by calling a function such as `free()`; since the user must explicitly call `free()`, calling another function to release the object's internal resources

does not break the pattern of explicit memory management. The following is an example of how this pattern could be provided for a data type in C:

```
typedef struct { ... } vector;

void vector_init(vector *);
void vector_destroy(vector *);
```

The user would be responsible for calling `vector_init()` after `malloc()` and `vector_destroy()` before `free()`:

```
vector *v = malloc(sizeof vector);
vector_init(v);
... // use the vector
vector_destroy(v);
free(v);
```

In some object-oriented languages, this style of resource management is directly integrated in the form of destructors. A *destructor* is a special method that is responsible for releasing the internal resources of an object, and the language ensures that an object's destructor is called just before the object is reclaimed. Destructors are the analogue of constructors: a constructor is called when an object is being initialized, while a destructor is called when an object is being destroyed.

The semantics of constructors and destructors give rise to a general pattern known as *resource acquisition is initialization*, or *RAII*. This ties the management of a resource to the lifetime of an object that acts as the resource manager, so perhaps a better name for this scheme is *lifetime-based resource management*. In the growable array example above, the constructor allocates the initial memory to be used by the array. If the array grows beyond its current capacity, a larger memory area is allocated and the previous one released. The destructor then ensures that the last piece of allocated memory is released. Since the constructor is always called when the growable array is created and the destructor when it is destroyed, the management of its internal memory is not visible to the user.

The RAII pattern can be used to manage resources other than memory. For example, an `fstream` object in C++ manages a file handle, which is a limited resource on most operating systems. The `fstream` constructor allocates a file handle and its destructor releases it, ensuring that the lifetime of the file handle is tied to that of the `fstream` object itself. A similar strategy can be used in a multithreaded program to tie the acquisition and release of a lock to the lifetime of an object.

When a resource manager is allocated with automatic storage duration, its lifetime matches the scope of its corresponding local variable. Thus, RAII is also known as *scope-based resource management*. However, RAII can also be used with dynamic objects in languages that are not garbage collected. We will see shortly why RAII does not work well with garbage collection.

Since the specific mechanism of RAII is unsuitable for general resource management in garbage-collected languages, some languages provide a specific construct for scope-based resource management. For example, Python has a `with` construct that works with *context managers*, which implement `__enter__()` and `__exit__()` methods:

```
with open('some_file') as f:
    <suite>
```

The `open()` function returns a file object, which defines the `__enter__()` and `__exit__()` methods that acquire and release a file handle. The `with` construct ensures that `__enter__()` is called before the suite is executed and `__exit__()` is called after the suite has executed. Python ensures that this is the case even if the suite exits early due to an exception or return.

Newer versions of Java provide a variant of `try` that enables scope-based resource management. Java also has a `synchronized` construct that specifically manages the acquisition and release of locks.

## 6.4 Garbage Collection

To avoid the prevalence of memory errors in languages that rely on programmers to manage memory, some languages provide automatic memory management in the form of *garbage collection*. This involves the use of runtime mechanisms to detect that objects are no longer in use and reclaim their associated memory. While a full treatment is beyond the scope of this text, we briefly discuss two major schemes for garbage collection: reference counting and tracing collection.

### 6.4.1 Reference Counting

*Reference counting* is a pattern of memory management where each object has a count of the number of references to the object. This count is incremented when a new reference to the object is created, and it is decremented when a reference is destroyed or modified to refer to a different object. As an example, consider the following Python code:

```
def foo():
    a = object()  # object A
    b = a
    b = object()  # object B
    a = None
    return
```

A reference-counting implementation of Python, such as CPython, keeps track of the number of references to each object. Upon a call to `foo()` and the initialization of `a`, the object `A` has a reference count of 1. The assignment of `a` to `b` causes the reference count of `A` to be incremented to 2. Assigning the new object `B` to `b` causes the count of `A` to be decremented and the count of `B` to be 1. Assigning `None` to `a` reduces the count of `A` to 0. At this point, the program no longer has any way to access the object `A`, so it can be reclaimed. Finally, returning from `foo()` destroys the variable `b`, so the count of `B` reduces to 0, and `B` can also be reclaimed.

Reference counting makes operations such as assignment and parameter passing more expensive, degrading overall performance. As a result, many language implementations use tracing schemes instead. However, reference counting has the advantage of providing predictable performance, making it well-suited to environments where the unpredictable nature of tracing collection can be problematic, such as real-time systems.

Some languages that are not garbage collected provide a mechanism for making use of reference counting in the form of *smart pointers*. In C++, the `shared_ptr` template is an abstraction of a reference-counting pointer. When a `shared_ptr` is created, the referenced object's count is incremented, and when the `shared_ptr` is destroyed, the count is decremented. The referenced object is destroyed when the count reaches 0. More details on `shared_ptr` and other C++ smart pointers such as `unique_ptr` and `weak_ptr` can be found in a handout from EECS 381.

A weakness of reference counting is that it cannot on its own detect when circular object chains are no longer in use. A simple example is a doubly linked list with multiple nodes, where each node holds a reference to its successor and predecessor, as shown in Figure 6.5.

Even if the first node is no longer accessible from program code after destruction of the list object on the left, the node still has a reference count of one since the second node holds a reference to the first. This prevents a reference-counting algorithm from reclaiming the nodes.

One solution is to provide *weak references*, which hold a reference to an object without incrementing the object's reference count. In the case of a doubly linked list, the reverse links can be represented using weak references so that they do not affect the reference counts of predecessor nodes.

The weak references in Figure 6.6 are shown as dashed lines. Now if the list object is reclaimed, the first node will no longer have any non-weak references to it, so its reference count will be zero. Thus, the first node will be reclaimed, which will then cause the second node's count to reach zero, allowing it to be reclaimed in turn, and so on.

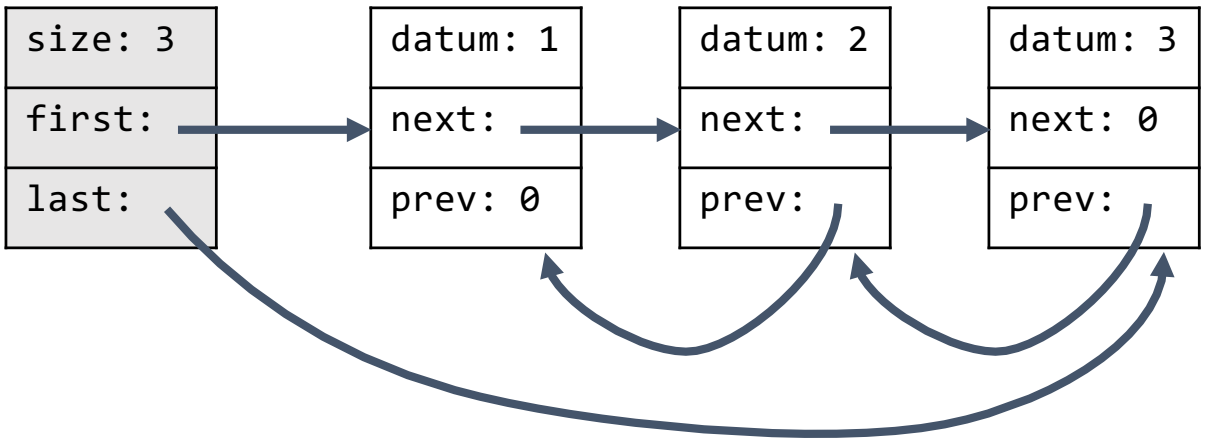



Figure 6.5: The nodes in a doubly linked list hold circular references to each other.

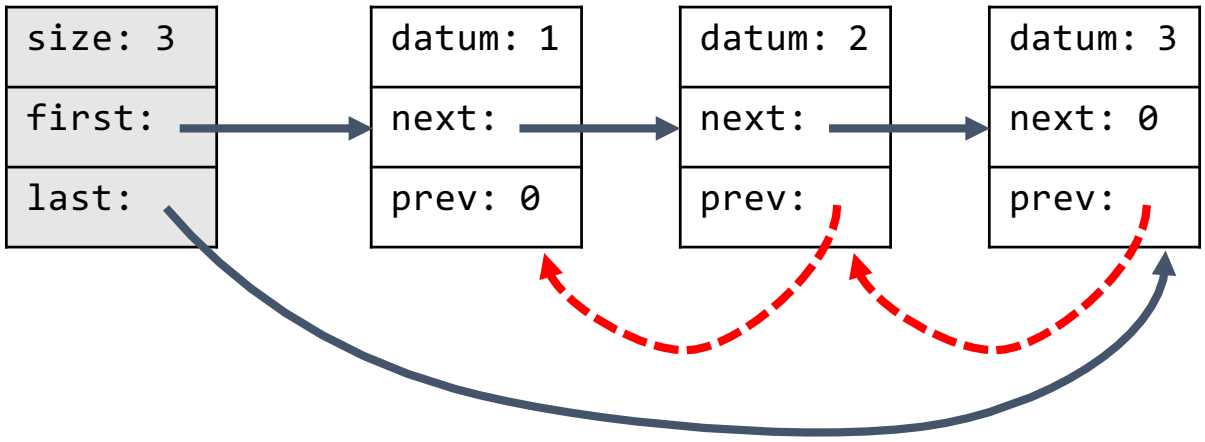



Figure 6.6: Weak references can be used to refer to the previous node, to avoid incrementing a node's reference count.

Weak references must be used carefully to ensure that cyclic data structures can be collected. This places a burden on the programmer, requiring more effort than the tracing schemes below.

### 6.4.2 Tracing Collectors

More common than reference counting is *tracing garbage collection*, which periodically traces out the set of objects in use and collects objects that are not reachable from program code. These collectors start out with a *root set* of objects, generally consisting of the objects on the stack and those in static or thread-local storage. They then recursively follow the references inside those objects, and the objects encountered are considered live. Objects that are not encountered in this process are reclaimed. For example, if the root set consists of objects $A$ and $H$ in the object graph in Figure 6.7, then objects $A$ through $K$ would be alive while objects $L$ through $O$ would be garbage.

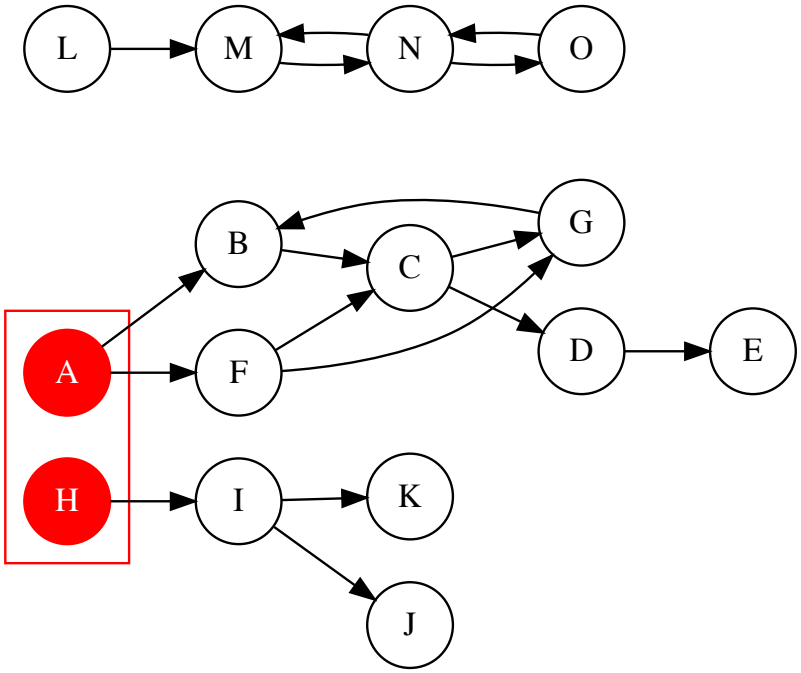



Figure 6.7: Tracing garbage collectors trace references starting at a root set, and objects that are not reachable from the root set are reclaimed.

There are many variants of tracing collectors. A common pattern is *mark and sweep*, which is split into separate mark and sweep phases. Objects are first recursively marked starting from the root set, and when this completes, unmarked objects are collected. Another pattern is *stop and copy*, which copies live objects to a separate, contiguous region of memory as they are encountered. The latter is slower and requires more free space but results in better locality of live objects. It also reduces the problem of memory *fragmentation*, where there is sufficient total free space to allocate an object, but each individual free region of space is too small for the object. However, since objects are moved, it also requires references and pointers to such objects to be updated, and the runtime must be able to distinguish references and pointers from other data values.

Tracing collectors often only run when free space is running low, so many programs do not even trigger garbage collection. Even in programs that do require collection, the amortized cost of tracing collection is often lower than that

of reference counting. On the other hand, the collection process itself can take a significant amount of time, and it can be problematic if a collection is triggered immediately before an event that the program needs to respond to, such as user input.

### 6.4.3 Finalizers

Garbage-collected languages often allow *finalizers* to be defined, which are analogous to destructors in a language such as C++. A finalizer is called when an object is being collected, allowing it to release internal resources in the same manner as destructors. However, finalizers give rise to a number of issues that do not occur in destructors. First, a finalizer may not be called in a timely manner, particularly in implementations that use a tracing collector, since such a collector often only collects objects when memory resources are running low. This makes finalizers unsuitable for managing resources that can be exhausted before memory is. Second, a finalizer may leak a reference to the object being collected, resurrecting it from the dead. A collector must be able to handle this case, and this also leads to the question of whether or not a finalizer should be rerun when the resurrected object is collected again. Another issue with finalizers is that they do not run in a well-defined order with respect to each other, preventing them from being used where the release of resources must be done in a specific order. Finally, many languages do not guarantee that finalizers will be called, particularly on program termination, so programmers cannot rely on them.

For the reasons above and several others, programmers are often discouraged from using finalizers for resource management. Instead, a scope-based mechanism such as the ones discussed previously should be used when available.

CHAPTER

SEVEN

# GRAMMARS

The *grammar* of a language specifies what sequences of character constitute valid fragments of the language. Grammar is only concerned with the structure of fragments, rather than the meaning. As in *Levels of Description*, the lexical structure of a language determines what the valid tokens are, and the syntax determines what sequences of tokens are valid. Here, we consider tools for specifying the lexical structure and syntax of a language.

## 7.1 Regular Expressions

The lexical structure of a language is often specified with regular expressions. A *regular expression* is a sequence of characters that defines a pattern against which strings can be matched.

The fundamental components of a regular expression are the following:

- the empty string, usually denoted by the Greek letter epsilon: $\varepsilon$
- individual characters from the alphabet of a language, such as *a* or *b* in the English alphabet
- concatenation, often denoted by listing a sequence of components, such as *ab*
- *alternation*, representing a choice between two options, often denoted by a vertical pipe, as in *a|b*
- repetition with the *Kleene star*, representing zero or more occurrences of a component, such as in *a**

Parentheses can be used to disambiguate application of concatenation, alternation, and the Kleene star. When parentheses are elided, the Kleene star has highest priority, followed by concatenation, followed by alternation.

The following are examples of regular expressions, as well as the strings they match:

- $a|b$ — matches exactly the strings *a* and *b*
- $a^*b$ — matches the strings containing any number of *a*'s followed by a single *b*, i.e. *b, ab, aab, aaab, …*
- $(a|b)^*$ — matches any string that contains no characters other than *a* or *b*, including the empty string, i.e. $\varepsilon$, *a, b, aa, ab, ba, bb, aaa, …*
- $ab^*(c|\varepsilon)$ — matches strings that contain a single *a*, followed by any number of *b*'s, followed by an optional *c*, i.e. *a, ac, ab, abc, abb, abbc, …*

Many regular expression systems provide shorthands for common cases. For example, the question mark *?* is often used to denote zero or one occurrence of an element, so that the last example above could be written as $ab^*c?$. Similarly, the plus sign + usually indicates one or more occurrences of an element, so that $a^+b$ matches the strings *ab, aab, aaab, …* Other common extensions include a mechanism for specifying a range of characters, shorthand for a set of characters as well the negation of a set of characters, and escape sequences, such as for whitespace.

As an example, the following regular expression matches an identifier or keyword in C++:

```
[a-zA-Z_][a-zA-Z_0-9]*
```

In this particular notation, square brackets denote a set of characters, acting as shorthand for alternation. A dash specifies a range of characters, so *a*-*z* denotes all the ASCII characters between *a* and *z*, inclusive. Thus, the regular expression matches any string that begins with a letter or underscore and follows that with any number of letters, underscores, or digits.


Figure 7.1: Credit: xkcd

Regular expressions are a very powerful mechanism in searching for and matching patterns. However, they are too limited to specify many common syntax rules. For example, there is no way to write a regular expression to match strings of the form $a^n b^n$, strings that contain any number of *a*'s followed by the same number of *b*'s, such as $\varepsilon$, *ab, aabb, aaabbb, aaaabbbb, …* This is an even simpler set of strings than that corresponding to matching sets of parentheses,

which include strings such as *()()* and *((()(())))*, that are common to many languages.

## 7.2 Context-Free Grammars

While the lexical structure of a language is often specified using regular expressions, the syntactic structure is generally specified with a *context-free grammar (CFG)*. A context-free grammar consists of a set of variables, a set of *terminals*, and a collection of *production rules* that specify how variables can be replaced with other variables or terminals. The *start variable* is the variable that should be used to begin the replacement process. Variables are replaced until no more variables remain, leaving just a sequence of terminals.

As a first example, consider the set of strings containing any number of $a$'s followed by the same number of $b$'s. We can specify a CFG that matches this set of strings. The terminals consist of the empty string $\varepsilon$, $a$, and $b$. We need a single variable, which we will call *S*, that will also be the start variable. Then the replacement rules are:

$$(1)\ S \to \varepsilon$$
$$(2)\ S \to a\ S\ b$$

To match a particular string, such as *aabb*, we begin with the start variable *S* and recursively apply production rules until we are left with just terminals that match the string. The following series of applications leads to the target string:

$$\begin{aligned} S &\to a\ S\ b && \text{by application of rule (2)} \\ &\to a\ a\ S\ b\ b && \text{by application of rule (2)} \\ &\to a\ a\ b\ b && \text{by application of rule (1)} \end{aligned}$$

The sequence of applications above is called a *derivation*, and it demonstrates that the string *aabb* is matched by the CFG above.

As another example, the following CFG defines the set of strings consisting of matching parentheses, where *P* is the start variable:

$$(1)\ P \to \varepsilon$$
$$(2)\ P \to (\ P\ )$$
$$(3)\ P \to P\ P$$

We can derive the string *(())* as follows:

$$\begin{aligned} P &\to (\ P\ ) && \text{by application of rule (2)} \\ &\to (\ (\ P\ )\ ) && \text{by application of rule (2)} \\ &\to (\ (\ )\ ) && \text{by application of rule (1)} \end{aligned}$$

We can derive the string *()()* as follows:

$$\begin{aligned} P &\to P\ P && \text{by application of rule (3)} \\ &\to (\ P\ )\ P && \text{by application of rule (2)} \\ &\to (\ )\ P && \text{by application of rule (1)} \\ &\to (\ )\ (\ P\ ) && \text{by application of rule (2)} \\ &\to (\ )\ (\ ) && \text{by application of rule (1)} \end{aligned}$$

An alternate derivation is as follows:

$$\begin{aligned} P &\to P\ P && \text{by application of rule (3)} \\ &\to P\ (\ P\ ) && \text{by application of rule (2)} \\ &\to P\ (\ ) && \text{by application of rule (1)} \\ &\to (\ P\ )\ (\ ) && \text{by application of rule (2)} \\ &\to (\ )\ (\ ) && \text{by application of rule (1)} \end{aligned}$$

Other derivations exist as well. However, the derivations have the same fundamental structure, which we can see by drawing a *derivation tree* that represents the recursive application of rules in a tree structure. The first derivation above has the structure in Figure 7.2.

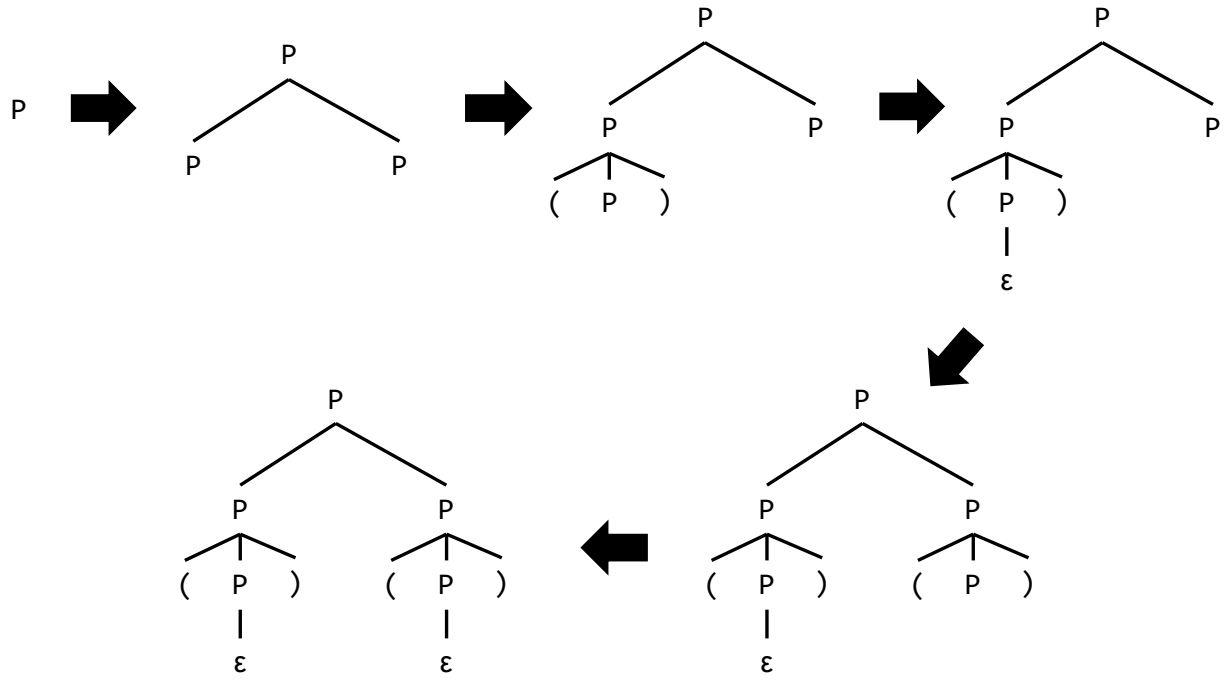



Figure 7.2: A derivation of *()()* that derives the left set of parentheses first.

The second derivation constructs the same structure in a different order, as shown in Figure 7.3.

The leaves of a derivation tree are terminals, and the in-order traversal is the string that is matched. In both derivations above, both the in-order traversal as well as the structure of the tree are the same.

Let us consider another grammar, representing arithmetic operations over symbols $a$ and $b$:

$$
\begin{aligned}
&(1)\ E \to E\ +\ E \\
&(2)\ E \to E\ *\ E \\
&(3)\ E \to a \\
&(4)\ E \to b
\end{aligned}
$$

This CFG has the terminals +, *, $a$, and $b$, and the variable $E$, which is also the start variable. Consider the string $a + b * a$. We can derive it as follows:

$$
\begin{aligned}
E &\to E\ +\ E && \text{by application of rule (1)} \\
&\to E\ +\ E\ *\ E && \text{by application of rule (2) on the second } E \\
&\to a\ +\ E\ *\ E && \text{by application of rule (3)} \\
&\to a\ +\ b\ *\ E && \text{by application of rule (4)} \\
&\to a\ +\ b\ *\ a && \text{by application of rule (3)}
\end{aligned}
$$

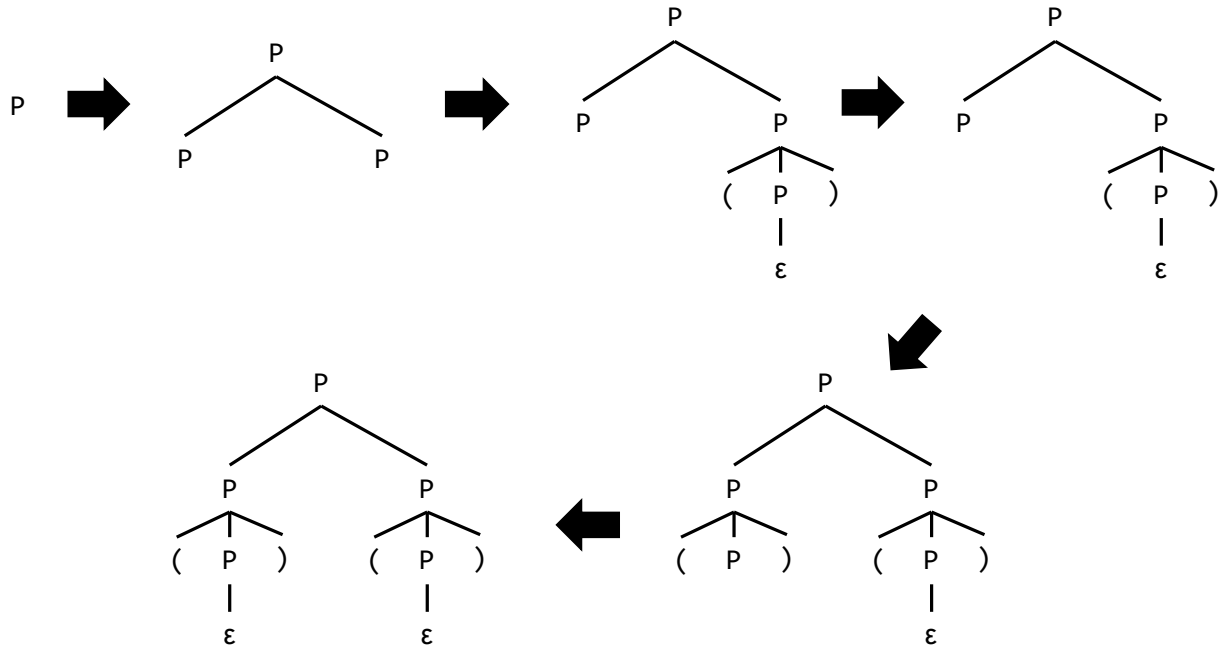



Figure 7.3: A derivation of *()()* that derives the right set of parentheses first.

Alternatively, we can derive the string as follows:

$$
\begin{aligned}
E &\rightarrow E * E && \text{by application of rule (2)} \\
&\rightarrow E + E * E && \text{by application of rule (1) on the first } E \\
&\rightarrow a + E * E && \text{by application of rule (3)} \\
&\rightarrow a + b * E && \text{by application of rule (4)} \\
&\rightarrow a + b * a && \text{by application of rule (3)}
\end{aligned}
$$

The derivation trees corresponding to the two derivations are in Figure 7.4, with the left tree as the result of the first derivation and the right tree the second.

While both derivation trees have the same in-order traversal, they have a fundamentally different structure. In fact, the first tree corresponds to the * operator having higher precedence than +, while the second tree is the reverse. Since the CFG admits both derivations, it is *ambiguous*.

We can rewrite the CFG to unambiguously give * the higher precedence, but doing so is cumbersome, particularly when a language has many operators. Instead, languages often resolve ambiguities by specifying precedence rules that determine which production rule to apply when there is a choice of rules that can lead to ambiguity.

## 7.3 Grammars in Programming Languages

The syntax of a programming language is usually specified using a context-free grammar. In some languages, the lexical structure is also specified with a CFG, as every regular expression can be written as a CFG. In languages that specify the lexical structure with regular expressions, the terminals of their grammars consist of program tokens. On the other hand, in those that specify the lexical structure with a CFG, the terminals are individual characters.

A language's context-free grammar is often written in *extended Backus-Naur form*, which adds convenient shorthands to the basic form discussed above. In particular, many grammars use notation from regular expressions, such as alternation

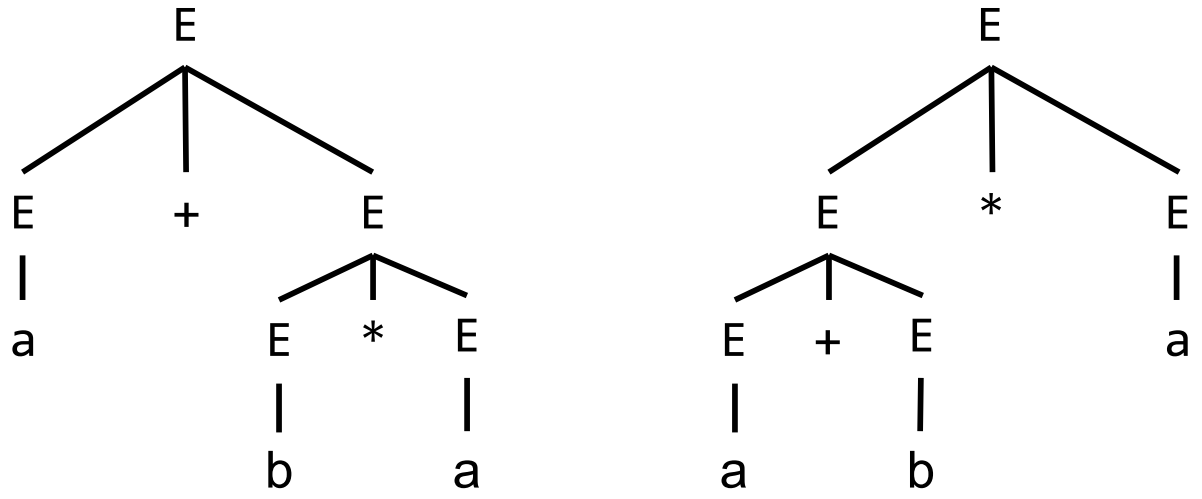



Figure 7.4: The string $a + b * a$ can be derived with two different resulting structures, so the grammar is ambiguous.

and Kleene stars, and the right arrow specifying a production rule is often replaced with an ASCII character such as = or ∴.

As an example, the following is a grammar specifying keywords, boolean literals, and identifiers in a C-like language, with identifiers taking the form of the regular expression `[a-zA-z_][a-zA-Z_0-9]*`:

```
Identifier: except Keyword and BooleanLiteral
    IdentifierStartCharacter
    IdentifierStartCharacter IdentifierCharacters

IdentifierStartCharacter:
    _
    LowerCaseLetter
    UpperCaseLetter

IdentifierCharacters:
    IdentifierCharacter
    IdentifierCharacters IdentifierCharacter

IdentifierCharacter:
    IdentifierStartCharacter
    Digit

LowerCaseLetter: one of
    a b c d e f g h i j k l m n o p q r s t u v w x y z

UpperCaseLetter: one of
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Digit: one of
    0 1 2 3 4 5 6 7 8 9

Keyword: one of
```

(continues on next page)

(continued from previous page)

```
    if else while struct break continue return

BooleanLiteral: one of
    true false
```

The grammar here follows the convention used in the Java language specification. In particular, it uses a colon rather than a right arrow to specify a production rule, and alternation is specified by placing the different choices on different lines. Finally, it includes shorthand such as "except" and "one of" to simplify the structure of the grammar.

Here is the Java specification for a C-style comment:

```
TraditionalComment:
    / * CommentTail

CommentTail:
    * CommentTailStar
    NotStar CommentTail

CommentTailStar:
    /
    * CommentTailStar
    NotStarNotSlash CommentTail

NotStar:
    InputCharacter but not *
    LineTerminator

NotStarNotSlash:
    InputCharacter but not * or /
    LineTerminator
```

This grammar takes care to ensure that a */ sequence terminates a comment, even if it immediately follows other stars, but that a single star or slash does not do so.

The following is a subset of the Scheme grammar specifying the form of a list:

$$\langle \text{list} \rangle \rightarrow (\langle \text{datum} \rangle^*) \mid (\langle \text{datum} \rangle^+ \,.\, \langle \text{datum} \rangle) \mid \langle \text{abbreviation} \rangle$$
$$\langle \text{abbreviation} \rangle \rightarrow \langle \text{abbrev prefix} \rangle \, \langle \text{datum} \rangle$$
$$\langle \text{abbrev prefix} \rangle \rightarrow \text{'} \mid \text{`} \mid \text{,} \mid \text{,@}$$

Here, the grammar uses the pipe | to denote alternation. Thus, a list can take the form of zero or more items enclosed by parentheses, as in the following examples:

```
()
(+)
(define x 3)
```

A list can also be a *dotted list*, with one or more items followed by a period followed by another item, all enclosed by parentheses. This produces a list that is terminated by the last item rather than by an empty list. Here are some examples:

```
(1 . 2)
(a b . c)
(a . (list))
```

Finally, a list can take the form of a quotation marker followed by an item:

```
'hello
`world
,foo
,@bar
```

These combinations are syntactic sugar for lists representing quotation forms:

```
(quote hello)
(quasiquote world)
(unquote foo)
(unquote-splicing bar)
```

### 7.3.1 Vexing Parse

In a particularly complex language such as C++, ambiguity cannot be avoided in the grammar itself. Instead, external rules have to be specified for resolving ambiguity. These rules can be based on context that is impossible to capture in a context-free grammar. For example, in C++, whether or not a name refers to a type is used in disambiguation, and C++ prefers to disambiguate in favor of a declaration wherever possible. Coupled with the fact that C++ allows names to be parenthesized even in declarations, this leads to a class of vexing parses.

Consider the following example:

```
struct foo {
  foo() {
    cout << "foo::foo()" << endl;
  }
  foo(int x) {
    cout << "foo::foo(" << x << ")" << endl;
  }
  void operator=(int x) {
    cout << "foo::operator=(" << x << ")" << endl;
  }
};

int a = 3;
int b = 4;

int main() {
  foo(a);
  foo(b) = 3;
}
```

The two lines in `main()` are interpreted as declarations, not as a call to the constructor in the first line or a call to the constructor followed by the assignment operator in the second. Instead, the code is equivalent to:

```
int main() {
  foo a;
  foo b = 3;
}
```

A perhaps more vexing case results from the fact that C++ allows parameter names to be elided in a function declaration, and the elided name can be parenthesized. The following is an example of a function declaration with an elided

parameter name:

```
void func(int);
```

Parenthesizing the elided name results in:

```
void func(int());
```

Now consider the following class:

```
struct bar {
  bar(foo f) {
    cout << "bar::bar(foo)" << endl;
  }
};
```

Then the following line is ambiguous:

```
bar c(foo());
```

This can be the declaration of an object `c` of type `bar`, with a newly created `foo` object passed to the constructor. On the other hand, it can be the declaration of a function `c`, with return type `bar`, that takes in an unnamed parameter of type `foo`. In this case, the elided name is parenthesized. The C++ standard requires such a situation to be disambiguated in favor of a function declaration, resulting in the latter.

The disambiguation above is often referred to as the *most vexing parse*, and many compilers produce a warning when they encounter it. For example, Clang reports the following message on the code above:

```
foo.cpp:29:8: warning: parentheses were disambiguated as a function
declaration
      [-Wvexing-parse]
  bar c(foo());
     ^~~~~~~
foo.cpp:29:9: note: add a pair of parentheses to declare a variable
  bar c(foo());
        ^
        (    )
```

The extra pair of parentheses force the compiler to treat `foo()` as an expression, resulting in an object declaration rather than a function declaration.

# Part II

# Functional Programming

We now turn our attention to *procedural abstraction*, a strategy for decomposing complex programs into smaller pieces of code in the form of *functions* (also called *procedures* or *subroutines*; there are subtle differences in how these terms are used in various contexts, but for our purposes, we will treat them as synonyms). A function encapsulates some computation behind an interface, and as with any abstraction, the user of a function need only know what the function does and not how it accomplishes it. A function also generalizes computation by taking in arguments that affect what it computes. The result of the computation is the function's *return value*.

In this unit, we start by introducing Scheme, a functional language in the Lisp family. We then discuss aspects of functions that are relevant to all procedural languages before proceeding to take a closer look at *functional programming*, a programming paradigm that models computation after mathematical functions.

CHAPTER

# EIGHT

# INTRODUCTION TO SCHEME

In this section, we introduce a high-level programming language that encourages a functional style. Our object of study, the R5RS Scheme language, employs a very similar model of computation to Python's, but uses only expressions (no statements) and specializes in symbolic computation.

Scheme is a dialect of Lisp, the second-oldest programming language that is still widely used today (after Fortran). The community of Lisp programmers has continued to thrive for decades, and new dialects of Lisp such as Clojure have some of the fastest growing communities of developers of any modern programming language. To follow along with the examples in this text, you can download a Scheme interpreter or use an online interpreter.

## 8.1 Expressions

Scheme programs consist of expressions, which are either simple expressions or *combinations* in the form of lists. A simple expression consists of a literal or a symbol. A combination is a compound expression that consists of an operator expression followed by zero or more operand sub-expressions. Both the operator and operands are contained within parentheses:

```
> (quotient 10 2)
5
```

Scheme exclusively uses prefix notation. Operators are often symbols, such as + and *. Compound expressions can be nested, and they may span more than one line:

```
> (+ (* 3 5) (- 10 6))
19
> (+ (* 3
        (+ (* 2 4)
           (+ 3 5)
        )
     )
     (+ (- 10 7)
        6
     )
  )
57
```

Evaluating a combination requires first examining the operator to see if it represents a *special form*[1], which has its own evaluation procedure. If the operator is not a special form, then the operator and operand expressions are evaluated in

[1] Scheme also allows the definition of *macros*, which perform code transformations to a combination before evaluating it. We will revisit *Scheme macros* later.

some arbitrary order. The function that is the value of the operator is then applied to the arguments that are the values of the operands.

The `if` expression in Scheme is an example of a special form. While it looks syntactically like a call expression, it has a different evaluation procedure. The general form of an `if` expression is:

```
(if <predicate> <consequent> <alternative>)
```

To evaluate an `if` expression, the interpreter starts by evaluating the `<predicate>` part of the expression. If the `<predicate>` evaluates to a true value, the interpreter then evaluates the `<consequent>` and returns its value. Otherwise it evaluates the `<alternative>` and returns its value. The `<alternative>` may be elided.

Numerical values can be compared using familiar comparison operators, but prefix notation is used in this case as well:

```
> (>= 2 1)
#t
```

Truth values in Scheme, including the boolean values `#t` (for true) and `#f` (for false), can be combined with boolean special forms, which have evaluation procedures as follows:

- `(and <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to a false value, the value of the `and` expression is that false value, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to true values, the value of the `and` expression is the value of the last one.
- `(or <e1> ... <en>)` The interpreter evaluates the expressions `<e>` one at a time, in left-to-right order. If any `<e>` evaluates to a true value, that value is returned as the value of the `or` expression, and the rest of the `<e>`'s are not evaluated. If all `<e>`'s evaluate to false values, the value of the `or` expression is the value of the last one.

Truth values can also be manipulated with the `not` procedure:

- `(not <e>)` The value of a not expression is `#t` when the expression `<e>` evaluates to a false value, and `#f` otherwise.

## 8.2 Definitions

Values can be named using the `define` special form:

```
> (define pi 3.14)
> (* pi 2)
6.28
```

New functions (usually called *procedures* in Scheme) can be defined using a second version of the `define` special form. For example, to define squaring, we write:

```
(define (square x) (* x x))
```

The general form of a procedure definition is:

```
(define (<name> <formal parameters>) <body>)
```

The `<name>` is a symbol to be associated with the procedure definition in the environment. The `<formal parameters>` are the names used within the body of the procedure to refer to the corresponding arguments of the procedure. The `<body>` is an expression that will yield the value of the procedure application when the formal parameters are replaced by the actual arguments to which the procedure is applied. The `<name>` and the `<formal parameters>` are grouped within parentheses, just as they would be in an actual call to the procedure being defined.

Having defined square, we can now use it in call expressions:

```
> (square 21)
441

> (square (+ 2 5))
49

> (square (square 3))
81
```

User-defined functions can take multiple arguments and include special forms in their bodies:

```
> (define (average x y)
    (/ (+ x y) 2))
> (average 1 3)
2
> (define (abs x)
    (if (< x 0)
        (- x)
        x
    )
  )
> (abs -3)
3
```

Scheme supports local function definitions with static scope. We will defer discussion of this until we cover *higher-order functions*.

Anonymous functions, also called *lambda functions*, are created using the `lambda` special form. A `lambda` is used to create a procedure in the same way as `define`, except that no name is specified for the procedure:

```
(lambda (<formal-parameters>) <body>)
```

The resulting procedure is just as much a procedure as one that is created using `define`. The only difference is that it has not been associated with any name in the environment. In fact, the following expressions are equivalent:

```
> (define (plus4 x) (+ x 4))
> (define plus4 (lambda (x) (+ x 4)))
```

Like any expression that has a procedure as its value, a lambda expression can be used as the operator in a call expression:

```
> ((lambda (x y z) (+ x y (square z))) 1 2 3)
12
```

We will examine *lambda functions* in more detail later.

## 8.3 Compound Values

Pairs are built into the Scheme language. For historical reasons, pairs are created with the `cons` built-in function (and thus, pairs are also known as *cons cells*), and the elements of a pair are accessed with `car` and `cdr`:

```
> (define x (cons 1 2))
> x
(1 . 2)
> (car x)
1
> (cdr x)
2
```

Figure 8.1 illustrates the pair structure created by `(cons 1 2)`.

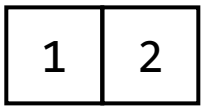



Figure 8.1: A pair consisting of the elements `1` and `2`.

Recursive lists are also built into the language, using pairs. A special value denoted `'()` represents the empty list. A recursive list value is rendered by placing its elements within parentheses, separated by spaces:

```
> (cons 1
        (cons 2
              (cons 3
                    (cons 4 '())
              )
         )
  )
(1 2 3 4)
> (list 1 2 3 4)
(1 2 3 4)
> (define one-through-four (list 1 2 3 4))
> (car one-through-four)
1
> (cdr one-through-four)
(2 3 4)
> (car (cdr one-through-four))
2
> (cons 10 one-through-four)
(10 1 2 3 4)
> (cons 5 one-through-four)
(5 1 2 3 4)
```

Figure 8.2 demonstrates that the structure corresponding to the list whose text representation is `(1 2 3 4)` consists of a sequence of pairs, terminated by the empty list (represented in the diagram as a box containing the symbol ∅).

A sequence of pairs that is terminated by something other than the empty list is called an *improper list*. Such a list is rendered by the interpreter with a dot preceding the last element; the result of `(cons 1 2)` is an example, as shown above, consisting of just a single pair in the sequence. The following demonstrates a more complex sequence:

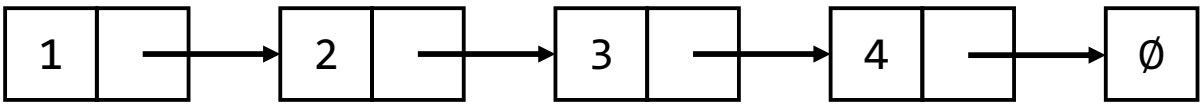



Figure 8.2: A list containing the elements 1, 2, 3, and 4.

```
> (cons 1
        (cons 2
              (cons 3 4)
        )
  )
(1 2 3 . 4)
```

Figure 8.3 demonstrates the pair structure corresponding to the improper list above.

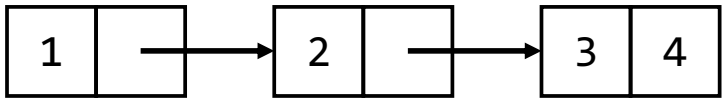



Figure 8.3: An improper list containing the elements 1, 2, and 3, and terminated by 4 rather than the empty list.

The illustrations in Figure 8.2 and Figure 8.3 demonstrate that pairs and other compound objects have *reference semantics* – the `cdr` part of a pair stores a reference to the next pair in the sequence. The following code further demonstrates these reference semantics with variables:

```
> (define x (cons 1 2))
> (define y x)
> (eqv? x y)
#t
> (set-car! y 7)
> x
(7 . 2)
```

Here, the definition `(define y x)` results in both `x` and `y` referring to the same pair object. The `eqv?` procedure when applied to pairs returns true only when the two arguments refer to the same pair object (as opposed to `equal?`, which compares pairs structurally). Furthermore, when we use the `set-car!` mutator to modify the first item of the pair referenced by `y`, we can see that `x` references the same pair since it too shows the modification.

Whether an object is the empty list can be determined using the primitive `null?` predicate. Using it, we can define the standard sequence operations for computing the length of a proper list and selecting elements:

```
> (define (list-length items)
    (if (null? items)
        0
        (+ 1 (list-length (cdr items)))
    )
  )
> (define (getitem items n)
    (if (= n 0)
        (car items)
        (getitem (cdr items) (- n 1))
    )
  )
```

(continues on next page)

(continued from previous page)

```
> (define squares (list 1 4 9 16 25))
> (length squares)
5
> (getitem squares 3)
16
```

The built-in `length` and `list-ref` procedures provide the same functionality as `list-length` and `getitem` here.

## 8.4 Symbolic Data

All the compound data objects we have used so far were constructed ultimately from numbers. One of Scheme's strengths is working with arbitrary symbols as data.

In order to manipulate symbols we need a new element in our language: the ability to *quote* a data object. Suppose we want to construct the list `(a b)`. We can't accomplish this with `(list a b)`, because this expression constructs a list of the values of `a` and `b` rather than the symbols themselves. In Scheme, we refer to the symbols `a` and `b` rather than their values by preceding them with a single quotation mark:

```
> (define a 1)
> (define b 2)
> (list a b)
(1 2)
> (list 'a 'b)
(a b)
> (list 'a b)
(a 2)
```

In Scheme, any expression that is not evaluated is said to be *quoted*. This notion of quotation is derived from a classic philosophical distinction between a thing, such as a dog, which runs around and barks, and the word "dog" that is a linguistic construct for designating such things. When we use "dog" in quotation marks, we do not refer to some dog in particular but instead to a word. In language, quotation allow us to talk about language itself, and so it is in Scheme:

```
> (list 'define 'list)
(define list)
```

Quotation also allows us to type in compound objects, using the conventional printed representation for lists. We have already seen that `'()` denotes an empty list. Here are other examples:

```
> (car '(a b c))
a

> (cdr '(a b c))
(b c)
```

Quotation in Scheme is distinct from strings: the latter represent raw, unstructured data in character format, while the former represents structured data:

```
> "(- 3)"  ; a string containing the characters #\( #\- #\space #\3 #\)
"(- 3)"
> '(- 3)   ; produces a list containing the symbol - and number 3
(- 3)
> (car '(- 3))
```

(continues on next page)

(continued from previous page)

```
-
> (cdr '(- 3))
(3)
> (- 3)    ; calls the - procedure on the number 3
-3
```

In the examples above, the string literal `"(- 3)"` evaluates to itself. The quoted expression `'(- 3)` evaluates to a list containing the symbol `-` as its first element and the number 3 as its second. The last example evaluates the symbol `-` to obtain the corresponding procedure, evaluates the number 3 to itself, and then calls the `-` procedure on the number 3, producing `-3`. Put another way, data in a string literal remains as character data, neither evaluated nor parsed. A quoted expression is parsed but not evaluated, producing a structured representation of the data. An unquoted expression is both parsed and evaluated by the interpreter.

The full Scheme language contains additional features, such as mutation operations, vectors, and maps. However, the subset we have introduced so far provides a rich functional programming language capable of implementing many of the ideas we have discussed so far.

CHAPTER

NINE

# FUNCTIONS

We first consider various schemes that are used for passing data to functions in the form of parameters and arguments. We make a distinction between the parameters that appear in a function definition, which are also called *formal parameters*, and the actual values that are passed to the function when it is called. The latter are often called *actual parameters*, but we will use the term *argument* to refer to these values and the shorthand *parameter* for formal parameters.

## 9.1 Keyword Arguments

Some languages allow, or even require, parameter names to be provided when calling a function. This strategy is called *named parameters* or *keyword arguments*.

Keyword arguments generally allow arguments to be provided in a different order than the parameter list of a function. In Python, for example, a keyword argument can be used for any parameter. Consider the following code:

```
def foo(x, y):
    print(x, y)
```

Calling `foo()` without keyword arguments passes the first argument as the first parameter, and the second argument as the second parameter:

```
>>> foo(1, 2)
1 2
```

However, the arguments can be reordered using the parameter names:

```
>>> foo(y = 1, x = 2)
2 1
```

Python also provides mechanisms for defining parameters to be positional-only or keyword-only, but we will not discuss these mechanisms here.

A handful of languages require names to be provided for all or most arguments by default, as well as requiring that they be given in the same order as the parameters. The following is an example in Swift 3:

```
func greet(name: String, withGreeting: String) {
  print(withGreeting + " " + name)
}

greet(name: "world", withGreeting: "hello")
```

Calling `greet()` with the arguments in reverse order is erroneous.

Swift is also rare in that it allows different argument and parameter names to be specified for a parameter. This means that the name provided for an argument when calling a function can differ from the internal name of the parameter used in the body of the function.

## 9.2 Default Arguments

In some languages, a function declaration or definition may be provided with a *default argument* value that allows the function to be called without that argument. This can be an alternative to overloading, where separate function definitions are written to handle the cases where an argument is present or missing.

The following is an example in Python:

```
def power(base, exponent=2):
    return base ** exponent
```

The `power()` function can be called with a single argument, in which case the default argument `2` is used to compute the square of the number. It can also be called with two arguments to compute an arbitrary power:

```
>>> power(3)
9
>>> power(3, 4)
81
```

Parameters that have default arguments generally must appear at the end of the parameter list. Languages differ on when and in which environment they evaluate the default argument. The most common strategy is to evaluate a default argument every time a function is called, but to do so in the definition environment (static scope). Python is rare in that it only evaluates default arguments once, when the function definition statement is executed. This means that if the value of the parameter is modified in the function, subsequent calls to the same function could have different default values for the same parameter. For example:

```
def test(x=[]):
    x.append(1)
    print(x)

test()
test()
```

This will print:

```
[1]
[1, 1]
```

C and C++ have numerous rules concerning default arguments, necessitated by the fact that an entity can be declared multiple times. Default arguments can be provided in both stand-alone declarations as well as definitions. However, it is illegal for multiple visible declarations of the same entity to provide a default argument for the same parameter, even if the provided value is the same. The set of default arguments is the union of all visible declarations within the same scope, and a declaration may only introduce a default argument for a parameter if all following parameters have been supplied with default arguments by the previous and current declarations. Names used in a default argument are resolved at the point of declaration, but the argument expressions are evaluated when the function is called.

The following is a legal example of multiple declarations in C++:

```
int foo(int x, int y = 4);
int foo(int x = 3, int y) {
  return x + y;
}
```

C++ allows default arguments for template parameters in addition to function parameters, with similar validity rules.

## 9.3 Variadic Functions

A language may provide a mechanism for a function to be called with a variable number of arguments. This feature is often referred to as *varargs*, and functions that make use of it are *variadic*. The mechanism may provide type safety, or it may permit unsafe uses that result in erroneous or undefined behavior. A variadic parameter generally must appear at the end of a parameter list, and it matches arguments that remain once the non-variadic parameters are matched. Usually, only a single variadic parameter is allowed.

In languages that provide safe variadic functions, a common mechanism for doing so is to automatically package variable arguments into a container, such as an array or tuple. For example, the following Python function computes the product of its arguments:

```
def product(*args):
    result = 1
    for i in args:
        result *= i
    return result
```

The * in front of a parameter name indicates a variadic parameter, and the variable arguments are passed as a tuple bound to that name. The function above iterates over the elements of the tuple, updating the total product. In order to call `product()`, 0 or more arguments must be provided:

```
>>> product()
1
>>> product(1, 2, 3)
6
```

Python also provides variadic keyword arguments, which are packaged into a dictionary. Placing ** in front of a parameter specifies that it is a variadic keyword parameter, and such a parameter must be the last one. As an example, the following function has both a non-keyword variadic parameter and a variadic keyword parameter, printing out the tuple corresponding to the former and the dictionary for the latter:

```
def print_args(*args, **kwargs):
    print(args)
    print(kwargs)
```

```
>>> print_args(3, 4, x = 5, y = 6)
(3, 4)
{'x': 5, 'y': 6}
```

Finally, Python allows a sequence or dictionary to be "unpacked" using the * or ** operator, allowing the unpacked values to be used where a list of values is required. For example, the following unpacks a list to make a call to `product()`:

```
>>> product(*[1, 2, 3])
6
```

Scheme also supports variadic arguments. A procedure can be defined to take variadic arguments by using an improper list as the parameter list, terminated by a symbol rather than an empty list. The variadic parameter binds to any number of arguments, packaged into a (proper) list:

```
> (define (func . args)
    args
  )
> (func)
()
> (func 1 2 3)
(1 2 3)
```

The procedure `func` takes in any number of arguments and returns the list containing those arguments. It thus behaves as the built-in `list` procedure. We can also define a procedure that takes in both required and variadic arguments, as in the following definition of `average`:

```
> (define (average x . nums)
    (/ (apply + x nums)
       (+ 1 (length nums))
    )
  )
> (average 1)
1
> (average 1 3)
2
> (average 1 3 5 7)
4
```

The procedure takes in one or more arguments, with the first bound to the parameter `x` and the rest encapsulated in a list bound to the variadic `nums` parameter. We can forward variadic arguments using `apply`, which takes a procedure, any number of regular arguments, and lastly, a list containing the remaining arguments. For example, `(apply + 1 2 '(3 4))` is equivalent to the call `(+ 1 2 3 4)`. In the first example using `average` above, `nums` is bound to an empty list in the call `(average 1)`, and `(apply + x nums)` is equivalent to `(apply + 1 '())`, which itself is equivalent to `(+ 1)`. In the third example, `nums` is bound to a list `(3 5 7)`, so `(apply + x nums)` is equivalent to `(apply + 1 '(3 5 7))`, which is in turn equivalent to `(+ 1 3 5 7)`.

In both Python and Scheme, a variadic parameter can match arguments with any type because the two languages are dynamically typed. In statically typed languages, however, variadic parameters are usually restricted to a single type, though that type may be polymorphic. For example, the following is a variadic method in Java:

```
public static void print_all(String... args) {
  for (String s : args) {
    System.out.println(s);
  }
}
```

The arguments to `print_all()` must be `String`s, and they are packaged into a `String` array. Java also allows a single `String` array to be passed in as an argument:

```
print_all("hello", "world");
print_all(new String[] { "good", "bye" });
```

C and C++ also have a mechanism for variadic arguments, but it poses significant safety issues. In particular, it provides no information about the number of arguments and their types to the function being called. The following is an example of a function that returns the sum of its arguments:

```
#include <stdarg.h>

int sum(int count, ...) {
  va_list args;
  int total = 0;
  int i;
  va_start(args, count);
  for (i = 0; i < count; i++) {
    total += va_arg(args, int);
  }
  va_end(args);
  return total;
}
```

In this function, the first argument is assumed to be the number of remaining arguments, and the latter are assumed to have type `int`. Undefined behavior results if either of these conditions is violated. Another strategy is to use a format string to determine the number and types of arguments, as used in `printf()` and similar functions. The lack of safety of variadic arguments enables vulnerabilities such as format string attacks.

C++11 provides *variadic templates* that are type safe. We will discuss them later in the text.

## 9.4 Parameter Passing

Another area in which languages differ is in the semantics and mechanism used in order to communicate arguments between a function and its caller. A function parameter may be unidirectional (used for only passing input to a function or only passing output from a function to its caller), or it may be bidirectional. These cases are referred to as *input*, *output*, and *input/output* parameters. A language need not support all three parameter categories.

Different parameter passing techniques, or *call modes*, are used by languages. These affect the semantics of arguments and parameters as well as what parameter categories are supported. The following are specific call modes used by different languages:

- *Call by value*. A parameter represents a new variable in the frame of a function invocation. The argument value is copied into the storage associated with the new variable. Call-by-value parameters only provide input to a function, as in the following example in C++:

  ```
  void foo(int x) {
    x++;
    cout << x << endl;
  }

  int main() {
    int y = 3;
    foo(y);               // prints 4
    cout << y << endl;    // prints 3
  }
  ```

  Even though `foo()` modifies the input value, the modified value is not propagated back to the caller.

- *Call by reference*. An l-value must be passed as the argument, as the parameter aliases the object that is passed in. Any modifications to the parameter are reflected in the argument object. Thus, call by reference parameters provide both input and output. In C++, reference parameters provide call by reference, and they may be restricted to just input by declaring them `const`[2]. The following C++ example uses call by reference to swap the values of

[2] The `const` qualification further allows r-values to be passed as an argument, since C++ allows `const` l-value references to bind to r-values.

two objects:

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main() {
  int x = 3, y = 4;
  swap(x, y);
  cout << x << " " << y << endl;   // prints 4 3
}
```

Call by reference is sometimes used to refer to passing objects indirectly using pointers. The following C/C++ function swaps object values using pointers:

```
void swap(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main() {
  int x = 3, y = 4;
  swap(&x, &y);
  printf("%d %d\n", x, y);   // prints 4 3
}
```

Technically speaking, however, the arguments and parameters are separate pointer objects that are passed by value. That being said, the effect emulates call by reference, enabling both input and output to be achieved through a parameter.

- *Call by result*. In this mode, a parameter represents a new variable that is not initialized with a value from the caller. Instead, the caller specifies an l-value for the argument, and when the function call terminates, the final value of the parameter is copied to the l-value. Thus, call by result only provides output parameters. The following is an example, using C-like syntax with call by result:

```
void foo(result int x) {
  x = 3;
  x++;      // x is now 4
}

int y = 5;
foo(y);     // y is now 4
print(y);   // prints 4
```

- *Call by value-result*. This is the combination of call by value and call by result. The argument value is copied into a new variable corresponding to the parameter, and then upon return from the function, the value of the parameter is copied back to the l-value provided by the caller. This differs from call by reference in that copies are made upon entry and exit to the function. This can be illustrated by passing the same l-value to multiple parameters, as in the following example using C-like syntax with call by value-result:

```
int foo(value-result int x, value-result int y) {
  x++;
```

(continues on next page)

(continued from previous page)

```
  return x - y;
}

int z = 3;
print(foo(z, z));   // prints 1
print(z);           // prints 3 or 4, depending on the semantics
```

In this code, `x` and `y` are new variables that are initialized to the value of `z`, i.e. 3. The increment of `x` does not affect `y`, since they are separate variables, so the call to `foo()` returns 1. Thus, 1 is printed. (The final value of `z` depends on the semantics of the language as to whether it is copied from `x` or `y`.) If call by reference were used instead, then `x` and `y` would alias the same object, and the call to `foo()` would return 0.

- *Call by name*. In this mode, a full expression can be provided as an argument, but it is not evaluated at the time a function is called. Instead, the parameter name is replaced by the expression where the name occurs in the function, and the expression is evaluated at the time that it is encountered in the body. This is a form of *lazy evaluation*, where a value is not computed until it is needed. The following is an example using C-like syntax with call by name:

```
void foo(name int a, name int b) {
  print(b);   // becomes print(++y)
  print(b);   // becomes print(++y)
}

int x = -1, y = 3;
foo(++x, ++y);   // prints 4, then 4 or 5 depending on the exact
                 // language semantics; y is now 4 or 5
print(x);        // prints -1 -- x is unchanged
```

In this example, the argument expression `++x` is never evaluated since the corresponding call-by-name parameter `a` is never used. On the other hand, the expression `++y` is computed since the corresponding parameter `b` does get used. Depending on the language semantics, the expression may only be evaluated once and the value cached for subsequent use, or it may be evaluated each time the parameter is used.

There is a subtle issue that arises in call by name. Consider the following code that uses C-like syntax with call by name:

```
void bar(name int x) {
  int y = 3;
  print(x + y);
}

int y = 1;
bar(y + 1);
```

If we replace the occurrence of the parameter `x` in `bar()` with the argument expression, we get `y + 1 + y` as the argument to `print()`. If this is evaluated in the environment of `bar()`, the result is 7. This is undesirable, since it means that the implementation detail of a local declaration of `y` changes the behavior of the function.

Instead, the argument expression should be evaluated in the environment of caller. This requires passing both the argument and its environment to the function invocation. Languages that use call by name often use a compiler-generated local function, called a *thunk*, to encapsulate the argument expression and its environment. This thunk is then passed to the invoked function, and it is the thunk that is called when the parameter is encountered.

In some languages, the expression corresponding to a call-by-name parameter is only evaluated the first time the parameter is referenced, caching the result. The cached result is then used in each subsequent occurrence of the

parameter.

Call by value is the call mode used by most modern languages, including C, C++ (for non-reference parameters), Java, Scheme, and Python. Programmers often mistakenly believe the latter three languages use call by reference, but in reality, they combine call by value with reference semantics. This combination is sometimes called *call by object reference*. The following example illustrates that Python is call by value:

```
def swap(x, y):
    tmp = x
    x = y
    y = tmp
```

```
>>> x, y = 1, 2
>>> swap(x, y)
>>> x, y
(1, 2)
```

The erroneous `swap()` function merely changes the values of the local variables, which changes the objects they refer to, without affecting the variables used as arguments. This demonstrates that the storage for the global `x` and `y` is distinct from that of the parameters, so Python does not use call by reference. In fact, Python cannot even emulate call by reference in the manner that C and C++ pointers do.

## 9.5 Evaluation of Function Calls

We proceed to summarize to evaluation process of a function call.

The first step is to determine the non-local environment of a call to a nested function. In languages with nested functions and static scope, a reference to the non-local environment is stored in the associated function object when the nested-function definition itself is executed. Under dynamic scope with deep binding, the non-local environment is determined when the function is referenced by name. Finally, in dynamic scope with shallow binding, the non-local environment is the environment that is active when the function is called.

The next step is to pass the arguments to the function, using a newly created activation record for the function call. The arguments are evaluated in the existing environment and passed to the callee as follows:

1. **Call by value and call by value-result**: the argument is evaluated to obtain its r-value. The r-value is copied into the storage for the corresponding parameter in the new activation record.
2. **Call by reference**: the argument is evaluated to obtain its l-value. The corresponding parameter is bound to the object associated with the l-value.
3. **Call by result**: the argument is evaluated to obtain its l-value. Storage is allocated but not initialized within the new activation record.
4. **Call by name**: the argument expression is packaged into a thunk with the current environment. The parameter is bound to a reference to the thunk.

Once the parameters have been passed, execution of the caller pauses, and the body of the callee is executed in an environment consisting of the newly created activation record along with the callee's non-local environment. For call by name, an occurrence of a call-by-name parameter invokes the corresponding thunk either the first time the parameter is named or every time, according to the semantics of the language.

When the called function returns, its return value, if there is one, is placed in a designated storage location, generally in the activation record of the caller. For a call-by-result or call-by-value-result parameter, the current r-value of the parameter is copied into the object associated with the l-value of the corresponding function-call argument. The activation record for the callee is then destroyed, and execution resumes in the caller at the point following the function call. The evaluation result of the function call itself is the return value of the function.

CHAPTER

# TEN

# RECURSION

Recursion is a mechanism for repetition that makes use of functions and function application. It involves a function calling itself directly or indirectly, usually with arguments that are in some sense "smaller" than the previous arguments. A recursive computation terminates when it reaches a *base case*, an input where the result can be computed directly without making any recursive calls.

It is sufficient for a language to provide recursion and conditionals in order for it to be Turing complete.

## 10.1 Activation Records

On a machine, recursion works due to the fact that each invocation of a function has its own activation record that maps its local variables to values. Consider the following recursive definition of factorial:

```
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Calling `factorial(4)` results in five invocations of `factorial()`, with arguments from 4 down to 0. Each has its own activation record with its own binding for the parameter `n`:

```
factorial(4):   n --> 4
factorial(3):   n --> 3
factorial(2):   n --> 2
factorial(1):   n --> 1
factorial(0):   n --> 0
```

Figure 10.1 is an illustration of the set of activation records as produced by Python Tutor.

When `n` is looked up while executing the body of `factorial()`, each invocation obtains its own value of `n` without being affected by the other activation records.

An activation record requires more than just storage for parameters and local variables in order for function invocation to work. Temporary values also need to be stored somewhere, and since each invocation needs its own storage for temporaries, they are generally also placed in the activation record. An invocation also needs to know where to store its return value, usually in temporary storage in the frame of the caller. Finally, a function needs to know how to return execution to its caller. Details are beyond the scope of this text, but included in this information is the instruction address that follows the function call in the caller and the address of the caller's activation record.

The set of temporary objects can be conservatively determined statically, so the size of an activation record, as well as the placement of objects within it, can be determined at compile time. For `factorial()` above, temporary storage is required for `n - 1` as well as the result of the recursive call to `factorial()`. The location of the latter in the caller is

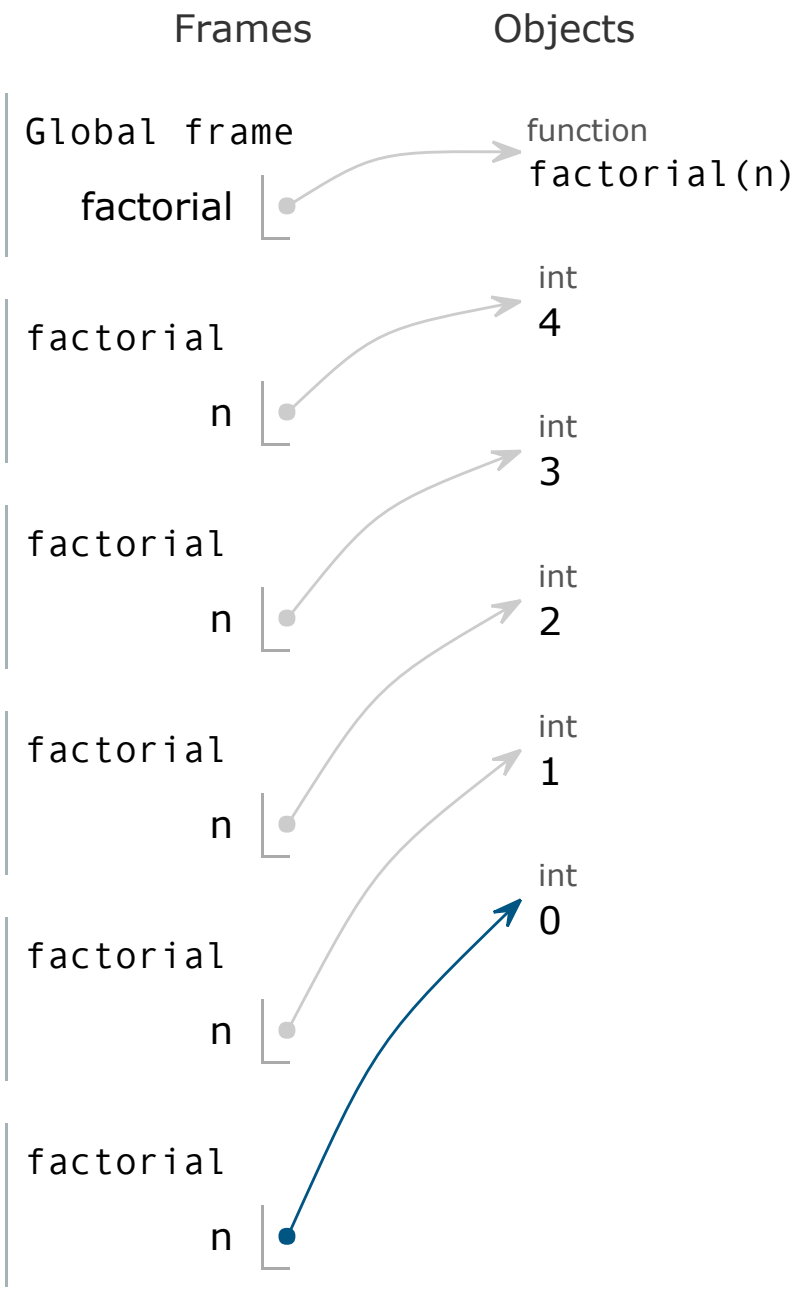



Figure 10.1: Activation records used to compute `factorial(4)`.

used by a recursive call to store its return value. Depending on the implementation, the invocation of `factorial(0)` may still have space for these temporary objects in its activation record even though they will not be used.

## 10.2 Tail Recursion

A recursive computation uses a separate activation record for each call to a function. The amount of space required to store these records is proportional to the number of active function calls. In `factorial(n)` above, when the computation reaches `factorial(0)`, all `n + 1` invocations are active at the same time, requiring space in `O(n)`. Contrast this with the following iterative implementation that uses constant space:

```
def factorial_iter(n):
    result = 1
    while n > 0:
        result *= n
        n -= 1
    return result
```

The space requirements of the recursive version of `factorial()`, however, is not intrinsic to the use of recursion but is a result of how the function is written. An invocation of `factorial(k)` cannot complete until the recursive call to `factorial(k - 1)` does, since it has to multiply the result by `k`. The fact that the invocation has work that needs to be done after the recursive call requires its activation record to be retained during the recursive call, leading to the linear space requirement.

Consider an alternative recursive computation of factorial:

```
def factorial_tail(n, partial_result = 1):
    if n == 0:
        return partial_result
    return factorial_tail(n - 1, n * partial_result)
```

Observe that the `factorial_tail()` function does not do any work after the completion of its recursive call. This means that it no longer needs the storage for parameters, local variables, or temporary objects when the recursive call is made. Furthermore, since `factorial(n, k)` directly returns the result of the recursive call `factorial(n - 1, n * k)`, the latter can store its return value in the location meant for the return value of `factorial(n, k)` in the caller of `factorial(n, k)`, and it can return execution directly to that caller. Thus, an optimizing implementation can reuse the space for the activation record of `factorial_tail(n, k)` for `factorial_tail(n - 1, n * k)` since the activation record of the former is no longer required.

This process can be generalized to any function call, not just recursive calls. A function call is a *tail call* if its caller directly returns the value of the call without performing any additional computation. A function is *tail recursive* if all of its recursive calls are tail calls. Thus, `factorial_tail()` is tail recursive.

A tail-recursive computation uses only a constant number of activation records, so its space usage matches that of an equivalent iterative computation. In fact, many functional languages do not provide constructs for iteration, since they can be expressed equivalently using tail recursion. These languages often require that implementations perform *tail-call optimization*, reusing the space for activation records where possible.

Since a tail call requires that no computation be performed after it returns, calls that syntactically appear to be tail calls may not be when implicit computation may occur at the end of a function. A specific example of this is scope-based resource management, as in the example below:

```
int sum(vector<int> values, int index, int partial_result = 0) {
  if (values.size() == index) {
    return 0;
```

(continues on next page)

(continued from previous page)

```
  }
  return sum(values, index + 1, partial_result + values[index])
}
```

While it appears that this code does not do computation after the recursive call, the local `vector<int>` object has a destructor that must run after the recursive call completes. Thus, the recursive call to `sum()` is not a tail call, and this computation is not tail recursive.

Another situation that prevents tail-call optimization is when a function contains a function definition within it, in languages that use static scope and support the full power of higher-order functions. The nested function requires access to its definition environment, so that environment must be retained if the nested function can be used after the invocation of its enclosing function completes or within a tail call.

CHAPTER

# ELEVEN

# HIGHER-ORDER FUNCTIONS

Recall that a first-class entity is one that supports the operations that can be done on other entities in a language, including being passed as a parameter, returned from a function, and created dynamically. In a language in which functions are first class, it is possible to write *higher-order functions* that take in another function as a parameter or return a function. Other languages may also support higher-order functions, even if functions are not first-class entities that can be created at runtime.

## 11.1 Function Objects

In some languages, it is possible to define objects that aren't functions themselves but provide the same interface as a function. These are known as *function objects* or *functors*. In general, languages enable functors to be written by allowing the function-call operator to be overloaded. Consider the following example in C++:

```
class Counter {
public:
  Counter : count(0) {}

  int operator()() {
    return count++;
  }

private:
  int count;
};
```

The `Counter` class implements a functor that returns how many times it has been called. Multiple `Counter` objects can exist simultaneously, each with their own count:

```
Counter counter1, counter2;
cout << counter1() << endl;  // prints 0
cout << counter1() << endl;  // prints 1
cout << counter1() << endl;  // prints 2
cout << counter2() << endl;  // prints 0
cout << counter2() << endl;  // prints 1
cout << counter1() << endl;  // prints 3
```

Functors allow multiple instances of a function-like object to exist, each with their own state that persists over the lifetime of the functor. This is in contrast to functions, where automatic objects do not persist past a single invocation, and static objects persist over the entire program execution.

Python also allows functors to be written by defining the special `__call__` method:

```
class Counter:
    def __init__(self):
        self.count = 0

    def __call__(self):
        self.count += 1
        return self.count - 1
```

In general, additional parameters can be specified when overloading the function-call operator, emulating functions that can take in those arguments.

Some languages do not allow the function-call operator itself to be overloaded but specify conventions that allow functor-like objects to be defined and used. For example, the following is an implementation of `Counter` in Java using the `Supplier<T>` interface, which specifies a zero-argument method that produces a `T`:

```
class Counter implements Supplier<Integer> {
  public Integer get() {
    return count++;
  }

  private int count = 0;
}
```

This functor-like object is then invoked by explicitly calling the `get()` method:

```
Supplier<Integer> counter1 = new Counter();
Supplier<Integer> counter2 = new Counter();
System.out.println(counter1.get()); // prints 0
System.out.println(counter1.get()); // prints 1
System.out.println(counter1.get()); // prints 2
System.out.println(counter2.get()); // prints 0
System.out.println(counter2.get()); // prints 1
System.out.println(counter1.get()); // prints 3
```

As another example, the `Predicate` interface in Java is implemented by functor-like objects that take in an argument and return a boolean value:

```
interface Predicate<T> {
  boolean test(T t);
  ...
}

class GreaterThan implements Predicate<Integer> {
  public GreaterThan(int threshold) {
    this.threshold = threshold;
  }

  public boolean test(Integer i) {
    return i > threshold;
  }

  private int threshold;
}
```

Code that uses these functor-like objects calls the `test()` method rather than calling the object directly:

```
GreaterThan gt3 = new GreaterThan(3);
System.out.println(gt3.test(2));    // prints out false
System.out.println(gt3.test(20));   // prints out true
```

Separate interfaces are provided for common patterns in the `java.util.function` library package.

## 11.2 Functions as Parameters

A higher-order function may take another function as a parameter. We first examine languages that only have top-level functions and allow a pointer or reference to a function to be passed as an argument. We then examine how passing a function as an argument can affect the environment in which the function's code is executed.

### 11.2.1 Function Pointers

In some languages, functions can be passed as parameters or return values but cannot be created within the context of another function. In these languages, all functions are defined at the top level, and only a pointer or reference to a function may be used as a value. Consider the following example in C, a language that provides function pointers:

```
void apply(int *array, size_t size, int (*func)(int)) {
  for (; size > 0; --size, ++array) {
    *array = func(*array);
  }
}

int add_one(int x) {
  return x + 1;
}

int main() {
  int A[5] = { 1, 2, 3, 4, 5 };
  apply(A, 5, add_one);
  printf("%d, %d, %d, %d, %d\n", A[0], A[1], A[2], A[3], A[4]);
  return 0;
}
```

The `apply()` function takes in an array, its size, and a pointer to a function that takes in an `int` and returns an `int`. It applies the function to each element in the array, replacing the original value with the result. The `add_one()` function is passed as an argument to `apply()` (C automatically converts a function to a function pointer), and the result is that each element in `A` has been incremented.

### 11.2.2 Binding Policy

In the code above, there are three environments associated with the `add_one()` function: its definition environment, the environment where it was referenced (in `main()`), and the environment where it was called (in `apply()`). Depending on the semantics of the language, any of these three environments may be components of the environment in which the body of `add_one()` is executed.

Recall that in static scope, the code in a function has access to the names in its definition environment, whereas in dynamic scope, it has access to the names in the environment of its use. Considering dynamic scope, is the non-local environment of a function the one where the function was referenced or the one where it was called? The following is an example where this distinction is relevant:

```
int foo(int (*bar)()) {
  int x = 3;
  return bar();
}

int baz() {
  return x;
}

int main() {
  int x = 4;
  print(foo(baz));
}
```

In dynamic scope, a function has access to the environment of its use. In the example above, however, the result is different depending on if the use environment of `baz()` is where the function was referenced or where it was called. In the former case, the non-local environment of `baz()` is the environment of `main()`, and the `x` in the body of `baz()` would refer to the one defined in `main()`. This is known as *deep binding*. In the latter case, the non-local environment of `baz()` is the environment of `foo()`, and `x` in `baz()` would refer to the one defined in `foo()`. This is called *shallow binding*. Both approaches are valid, and the binding policy of a language determines which one is used.

Binding policy can also make a difference when static scope is used in the case of functions defined locally inside of a recursive function. However, deep binding is universally used in languages with static scope, so that the environment established at the time of a function's definition is the one the function has access to.

## 11.3 Nested Functions

A key feature of functional programming is the ability to define a function from within another function, allowing the dynamic creation of a function. In languages with static scoping, such a nested function has access to its definition environment, and the combination of a function and its definition environment is called a *closure*. Variables used in the nested function but defined in the enclosing environment are said to be *captured* by the closure. If a nested function is returned or otherwise leaks from the enclosing function, the environment of the enclosing function generally must persist after the function returns, since bindings within it may be accessed by the nested function.

As an example, consider the following higher-order function in Python that returns a nested function:

```
def make_greater_than(threshold):
    def greater_than(x):
        return x > threshold

    return greater_than
```

The `make_greater_than()` function takes in a threshold value and constructs a nested function that determines if its input is greater than the threshold value. The `threshold` variable is located in the activation record of `make_greater_than()` but is captured by `greater_than()`. Since the latter is returned, the activation record must persist so that invocations of `greater_than()` can access the binding for `threshold`.

Observe that each time `make_greater_than()` is called, a different instance of `greater_than()` is created with its own enclosing environment. Thus, different invocations of `make_greater_than()` result in different functions:

```
>>> gt3 = make_greater_than(3)
>>> gt30 = make_greater_than(30)
>>> gt3(2)
```

(continues on next page)

(continued from previous page)

```
False
>>> gt3(20)
True
>>> gt30(20)
False
>>> gt30(200)
True
```

Figure 11.1 from Python Tutor shows the state when `gt3(2)` is called.

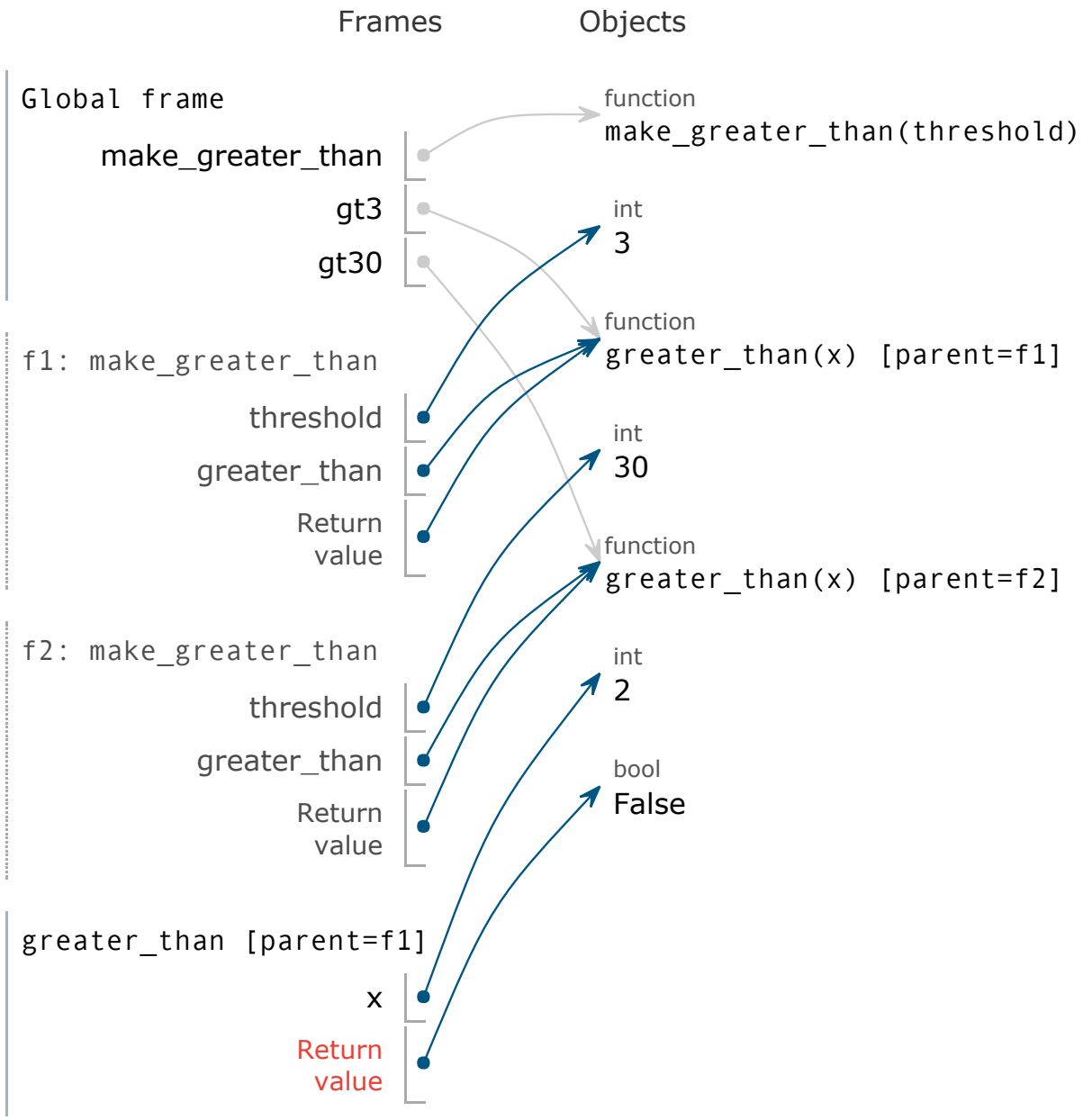



Figure 11.1: Environment for multiple instances of a nested function.

The parent frame of the invocation is that in which `threshold` is bound to 3, so `x > threshold` evaluates to false.

Languages that are not purely functional may allow modification of a captured variable. For example, the following defines a data abstraction for a bank account using nested functions:

```
def make_account(balance):
    def deposit(amount):
        nonlocal balance
        balance += amount
        return balance

    def withdraw(amount):
        nonlocal balance
        if 0 <= amount <= balance:
            balance -= amount
            return amount
        else:
            return 0

    return deposit, withdraw
```

The `nonlocal` statements are required in Python, since it assumes that assignments are to local variables by default. We can then use the created functions as follows:

```
>>> deposit, withdraw = make_account(100)
>>> withdraw(10)
10
>>> deposit(0)
90
>>> withdraw(20)
20
>>> deposit(0)
70
>>> deposit(10)
80
>>> withdraw(100)
0
>>> deposit(0)
80
```

We will return to *data abstraction using functions* later.

### 11.3.1 Decorators

A common pattern in Python is to transform a function (or class) by applying a higher-order function to it. Such a higher-order function is called a *decorator*, and Python has specific syntax for decorating functions:

```
@<decorator>
def <name>(<parameters>):
    <body>
```

This is largely equivalent to:

```
def <name>(<parameters>):
    <body>
```

(continues on next page)

(continued from previous page)

```
<name> = <decorator>(<name>)
```

The decorated function's definition is executed normally, and then the decorator is called on the function. The result of this invocation is then bound to the name of the function.

As an example, suppose we wanted to trace when a function is called by printing out the name of the function as well as its arguments. We could define a higher-order function that takes in a function and returns a new nested function that first prints out the name of the original function and its arguments and then calls it:

```
def trace(fn):
    def tracer(*args):
        args_string = ', '.join(repr(arg) for arg in args)
        print(f'{fn.__name__}({args_string})')
        return fn(*args)

    return tracer
```

Here, we make use of variadic arguments to pass any number of arguments to the original function. (For simplicity, we ignore keyword arguments.) We can then use decorator syntax to apply this to a function:

```
@trace
def factorial(n):
    return 1 if n == 0 else n * factorial(n - 1)
```

Now whenever a call to `factorial()` is made, we get a printout of the arguments:

```
>>> factorial(5)
factorial(5)
factorial(4)
factorial(3)
factorial(2)
factorial(1)
factorial(0)
120
```

Notice that the recursive calls also call the transformed function. This is because the name `factorial` is now bound to the nested tracer function in the enclosing environment of `factorial()`, so looking up the name results in the tracer function rather than the original one. A side effect of this is that we have *mutual recursion* where a set of functions indirectly make recursive calls through each other. In this case, the tracer calls the original `factorial()`, which calls the tracer, as shown in the diagram in Figure 11.2 for `factorial(2)` from Python Tutor.

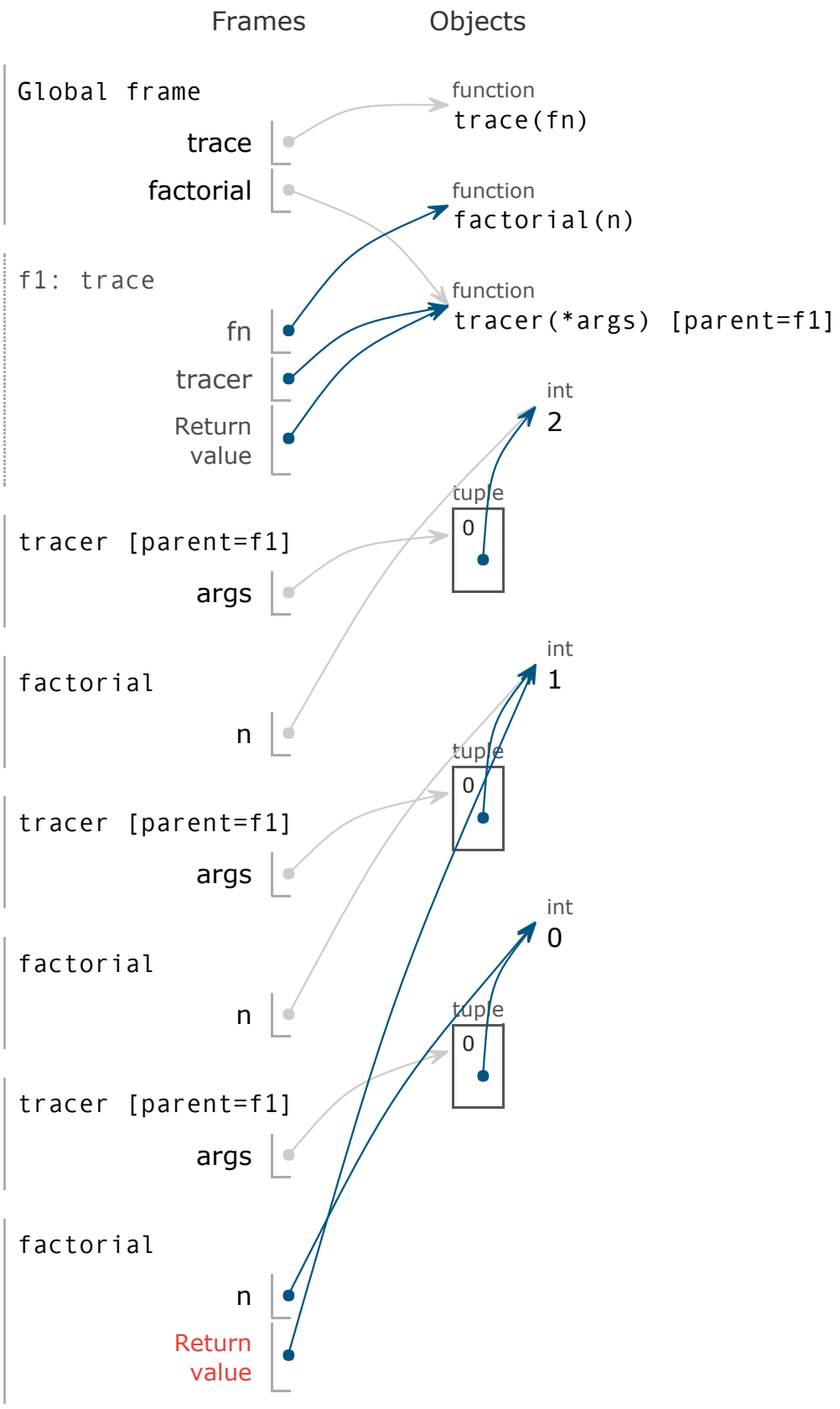



Figure 11.2: Mutual recursion resulting from decorating a recursive function.

CHAPTER

TWELVE

# LAMBDA FUNCTIONS

Nested function definitions allow the construction of functions at runtime, fulfilling one of the requirements for functions to be a first-class entity. So far, however, we've only seen nested function definitions that are named, introducing a binding into the definition environment. This is in contrast to other first-class entities, such as data values, that can be created without being bound to a name. Just like it can be useful to construct a value without a name, such as when passing it as an argument or returning it, it can be useful to construct unnamed functions. These are called *anonymous* or *lambda* functions.

Lambda functions are ubiquitous in functional languages, but many common imperative languages also provide some form of lambda functions. The syntax and capabilities differ between different languages, and we will examine a few representative examples.

## 12.1 Scheme

Lambdas are a common construct in the Lisp family of languages, those languages being primarily functional, and Scheme is no exception. The `lambda` special form constructs an anonymous function:

```
(lambda (<parameters>) <body>)
```

A function definition using the `define` form can then be considered a shorthand for a variable definition and a `lambda`:

```
(define (<name> <parameters>) <body>)
  -->
(define <name> (lambda (<parameters>) <body>))
```

As an example, consider the following function that creates and returns an anonymous function that adds a given number to its argument:

```
(define (make-adder n)
  (lambda (x)
    (+ x n)
  )
)
```

This is simpler and more appropriate than an equivalent definition that only uses `define`:

```
(define (make-adder n)
  (define (adder x)
    (+ x n)
  )
```

(continues on next page)

(continued from previous page)

```
  adder
)
```

We can then call the result of `make-adder` on individual arguments:

```
> (define add3 (make-adder 3))
> (add3 4)
7
> (add3 5)
8
> ((make-adder 4) 5)
9
```

Nested functions in Scheme use static scope, so the anonymous function has access to the variable `n` in its definition environment. It then adds its own argument `x` to `n`, returning the sum.

Scheme is not purely functional, allowing mutation of variables and compound data. Nested functions, whether anonymous or not, can modify variables in their non-local environment. The following function creates a counter function that returns how many times it has been called:

```
(define (make-counter)
  (let ((count 0))
    (lambda ()
      (set! count (+ count 1))
      (- count 1)
    )
  )
)
```

The `set!` form mutates a variable to the given value. We can then use the `make-counter` function as follows:

```
> (define counter (make-counter))
> (counter)
0
> (counter)
1
> (counter)
2
```

## 12.2 Python

Python supports anonymous functions with the `lambda` expression. This takes the following form:

```
lambda <parameters>: <body expression>
```

The syntax of lambda expressions in Python produce a constraint on anonymous functions that is not present in named nested functions: the body must be a single expression, and the value of that expression is automatically the return value of the function. In practice, this limitation is usually not a problem, since lambdas are often used in functional contexts where statements and side effects may not be appropriate.

The following is a definition of the `greater_than()` higher-order function that uses a lambda:

```
def make_greater_than(threshold):
    return lambda value: value > threshold
```

As can be seen in this example, simple nested functions that are used in only a single place can be written more succinctly with a lambda expression than with a definition statement.

While lambda functions in Python have access to their definition environment, they are syntactically prevented from directly modifying bindings in the non-local environment.

## 12.3 Java

Java does not allow nested function definitions, but it does have syntax for what it calls "lambda expressions." In actuality, this construct constructs an anonymous class with a method corresponding to the given parameters and body, and the compiler infers the base type of this class from the context of its use.

The following example uses a lambda expression to construct a functor-like object:

```
public static IntPredicate makeGreaterThan(int threshold) {
  return value -> value > threshold;
}
```

We can then use the result as follows:

```
IntPredicate gt3 = makeGreaterThan(3);
System.out.println(gt3.test(2));    // prints out false
System.out.println(gt3.test(20));   // prints out true
```

Java allows a lambda to take in any number of arguments, and providing types for the parameters is optional. The body can be a single expression or a block containing arbitrary statements.

On the other hand, Java places a significant restriction on lambda expressions. A lambda can only access variables in its definition environment that are never reassigned, and it cannot modify them itself. This is because lambdas are not implemented as closures, but rather as functor-like objects that store "captured" variables as members. The following is effectively equivalent to the code above, but using named classes and methods:

```
public static IntPredicate makeGreaterThan(int threshold) {
  return Anonymous(threshold);
}

class Anonymous implements IntPredicate {
  Anonymous(int threshold) {
    this.threshold = threshold;
  }

  public boolean test(int value) {
    return value > threshold;
  }

  private final int threshold;
}
```

## 12.4 C++

Like Java, C++ has lambda expressions, but they provide more functionality than those in Java. A programmer can specify which variables in the definition environment are captured, and whether they are captured by value or by reference. The former creates a copy of a variable, while the latter allows a captured variable to be modified by the lambda.

The simplest lambda expressions are those that do not capture anything from the enclosing environment. Such a lambda can be written as a top-level function instead[3], and C++ even allows a captureless lambda to be converted to a function pointer. For example, the following code passes a lambda function to a higher-order function that takes in a function pointer:

```
int max_element(int *array, size_t size, bool (*less)(int, int)) {
  assert(size > 0);
  int max_so_far = array[0];
  for (size_t i = 1; i < size; i++) {
    if (less(max_so_far, array[i])) {
      max_so_far = array[i];
    }
  }
  return max_so_far;
}

int main() {
  int array[5] = { 3, 1, 4, 2, 5 };
  cout << max_element(array, 5,
                      [](int a, int b) {
                        return a > b;
                      })
       << endl;
}
```

The code constructs a lambda function that returns true if the first element is bigger than the second, and passing that to `max_element()` finds the minimum rather than the maximum element.

Lambdas that capture variables, whether by value or by reference, have state that is associated with a specific evaluation of a lambda expression, and this state can differ between different calls to the enclosing function. As a result, such a lambda is not representable as a top-level function. Instead, C++ implicitly defines a functor type for a capturing lambda. Evaluating a capturing lambda expression constructs an instance of this functor type, with the captured values and references stored as non-static members. Since the functor type is implicitly defined, type deduction with the `auto` keyword is usually used where the type of the functor is required.

The following is an example that uses a lambda to define a greater-than functor:

```
auto make_greater_than(int threshold) {
  return [=](int value) {
    return value > threshold;
  };
}

int main() {
  auto gt3 = make_greater_than(3);
```

(continues on next page)

[3] A captureless lambda is actually implemented as a functor, avoiding an indirection when the lambda is invoked without first converting it to a function pointer.

(continued from previous page)

```
  cout << gt3(2) << endl;                      // prints 0
  cout << gt3(20) << endl;                     // prints 1
  cout << make_greater_than(30)(20) << endl;   // prints 0
}
```

The = in the capture list for the lambda specifies that all variables from the enclosing environment that are used by the lambda should be captured by value. The code above is equivalent to the following that explicitly uses a functor:

```
class GreaterThan {
public:
  GreaterThan(int threshold_in) : threshold(threshold_in) {}

  bool operator()(int value) const {
    return value > threshold;
  }

private:
  const int threshold;
};

auto make_greater_than(int threshold) {
  return GreaterThan(threshold);
}
```

As indicated in the code above, a variable captured by value is implicitly qualified as `const`.

An enclosing variable may also be captured by reference. However, a variable that is captured by reference does **not** have its lifetime extended. The reasoning for this is twofold. The first, practical reason is that C++ implementations generally use stack-based management of automatic variables, and when a function returns, its activation record on the stack is reclaimed. Requiring that a variable live past its function invocation prevents activation records from being managed using a stack. The second, more fundamental reason is that the RAII (i.e. scope-based resource management) paradigm in C++ requires that when an automatic variable goes out of scope, the destructor for its corresponding object is run and the object reclaimed. Relaxing this requirement would result in undesirable effects similar to those of finalizers in garbage-collected languages.

The end result is that a lambda functor that captures by reference should not be used past the existence of its enclosing function invocation. The following counter definition is therefore erroneous:

```
auto make_counter() {
  int count = 0;
  return [&]() {
    return count++;
  };
}
```

The lifetime of the `count` variable ends when `make_counter()` returns, so that calling the lambda functor afterwards erroneously uses a dead object.

An alternative is to capture `count` by value, which stores a copy as a member of the lambda, and then mark the lambda as `mutable`. This removes the implicit `const` qualification from variables captured by value, allowing them to be modified:

```
auto make_counter() {
  int count = 0;
```

(continues on next page)

(continued from previous page)

```
  return [=]() mutable {
    return count++;
  };
}
```

This definition is equivalent to the `Counter` functor we defined in *Function Objects*.

## 12.5 Common Patterns

We now take a look at some common computational patterns in functional programming. We will look at how to abstract these patterns as higher-order functions, as well as how to use them with lambda functions.

### 12.5.1 Sequence Patterns

A number of functional patterns operate over sequences. These patterns take in a sequence and a function and apply the function to elements of the sequence, producing a new sequence or value as a result. Since these are functional patterns, the original sequence is left unchanged.

#### Map

The *map* pattern takes a sequence and a function and produces a new sequence that results from applying the function to each element of the original sequence. For example, the following adds 1 to each element of a Scheme list:

```
> (map (lambda (x) (+ x 1)) '(1 2 3))
(2 3 4)
```

We can define the `map` higher-order function as follows:

```
(define (map func lst)
  (if (null? lst)
      lst
      (cons (func (car lst))
            (map func (cdr lst)))
  )
)
```

Applying `map` to an empty list results in an empty list. Otherwise, `map` applies the given function to the first item in the list and recursively calls `map` on the rest of the list.

Python has a built-in `map()` function that takes in a function and an iterator and returns an iterator that results from applying the function to each item in the original iterator.

### Reduce

In the *reduce* pattern, a two-argument function is applied to the first two items in a sequence, then it is applied to the result and the next item, then to the result of that and the next item, and so on. A reduction may be left or right associative, but the former is more common. Figure 12.1 illustrates the difference between left- and right-associative reductions.

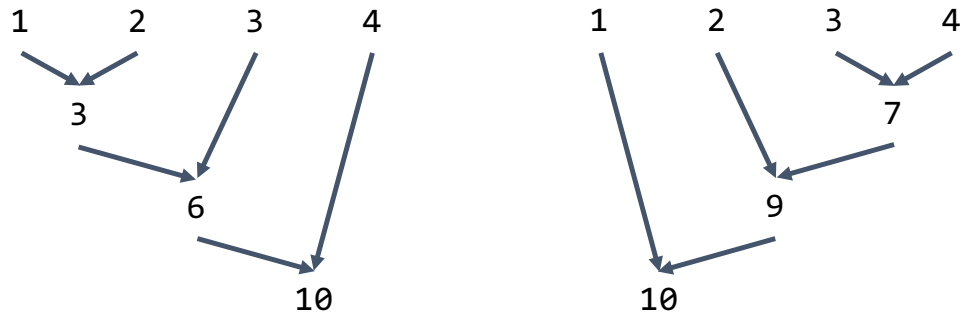



Figure 12.1: Left-associative and right-associative reductions.

Often, if only a single item is in the sequence, that item is returned without applying the function. Some definitions allow an initial value to be specified as well for the case in which the sequence is empty.

The following examples compute the sum and maximum element of a Scheme list:

```
> (reduce-right (lambda (x y) (+ x y)) '(1 2 3 4))
10
> (reduce-right (lambda (x y) (if (> x y) x y)) '(1 2 3 4))
4
```

We can define a right-associative reduction as follows, which assumes that the given list has at least one element:

```
(define (reduce-right func lst)
  (if (null? (cdr lst))
      (car lst)
      (func (car lst) (reduce-right func (cdr lst)))
  )
)
```

Python includes a left-associative `reduce()` function in the `functools` module.

### Filter

The *filter* pattern uses a predicate function to filter items out of a list. A *predicate* is a function that takes in a value and returns true or false. In filter, elements that test true are retained while those that test false are discarded.

The following example filters out the odd elements from a list:

```
> (filter (lambda (x) (= (remainder x 2) 0)) '(1 2 3 4))
(2 4)
```

The following is a definition of `filter`:

```
(define (filter pred lst)
  (if (null? lst)
      lst
      (if (pred (car lst))
          (cons (car lst) (filter pred (cdr lst)))
          (filter pred (cdr lst))
      )
  )
)
```

Python provides a built-in `filter()` function as well.

#### Any

The *any* pattern is a higher-order version of `or` (disjunction). It takes a predicate and applies the predicate to each successive item in a list, returning the first true result from the predicate. If no item tests true, then false is returned. Some languages use the name *find* for this pattern rather than *any*.

The following examples search a list for an even value:

```
> (any (lambda (x) (= (remainder x 2) 0)) '(1 2 3 4))
#t
> (any (lambda (x) (= (remainder x 2) 0)) '(1 3))
#f
```

A short-circuiting `any` function can be defined as follows:

```
(define (any pred lst)
  (if (null? lst)
      #f
      (let ((result (pred (car lst))))
        (or result
            (any pred (cdr lst))
        )
      )
  )
)
```

The *every* pattern can be similarly defined as the higher-order analogue of conjunction.

### 12.5.2 Composition

Programs often compose functions, applying a function to the result of applying another function to a value. Wrapping these two function applications together in a single function enables both operations to be done with a single call. For example, the following multiplies each item in a list by three and then adds one:

```
> (map (compose (lambda (x) (+ x 1))
                (lambda (x) (* 3 x)))
       '(3 5 7)
  )
(10 16 22)
```

We can define `compose` as follows:

```
(define (compose f g)
  (lambda (x)
    (f (g x))
  )
)
```

### 12.5.3 Partial Application and Currying

Partial application allows us to specify some arguments to a function at a different time than the remaining arguments. Supplying $k$ arguments to a function that takes in $n$ arguments results in a function that takes in $n - k$ arguments.

As an example, suppose we want to define a function that computes powers of two. In Python, we can supply 2 as the first argument to the built-in `pow()` function to produce such a function. We need a partial-application higher-order function such as the following:

```
def partial(func, *args):
    def newfunc(*nargs):
        return func(*args, *nargs)

    return newfunc
```

We can then construct a powers-of-two function as follows:

```
>>> power_of_two = partial(pow, 2)
>>> power_of_two(3)
8
>>> power_of_two(7)
128
```

Python actually provides a more general implementation of `partial()` that works for keyword arguments as well in the `functools` module. C++ provides partial application using the `bind()` template in the <`functional`> header.

A related but distinct concept is *currying*, which transforms a function that takes in $n$ arguments to a sequence of $n$ functions that each take in a single argument. For example, the `pow()` function would be transformed as follows:

```
>>> curried_pow(2)(3)
8
```

The curried version of the function takes in a single argument, returning another function. The latter takes in another argument and produces the final value. Since the original `pow()` takes in two arguments, the curried function chain has length two.

We can define currying for two-parameter functions as follows in Python:

```
def curry2(func):
    def curriedA(a):
        def curriedB(b):
            return func(a, b)

        return curriedB

    return curriedA
```

Then we can call `curry2(pow)` to produce a curried version of `pow()`.

We can also define an "uncurry" operation that takes in a function that must be applied to a sequence of $n$ arguments and produce a single function with $n$ parameters. The following does so for a sequence of two arguments:

```
def uncurry2(func):
    def uncurried(a, b):
        return func(a)(b)

    return uncurried
```

```
>>> uncurried_pow = uncurry2(curried_pow)
>>> uncurried_pow(2, 3)
8
```

Some functional languages, such as Haskell, only permit functions with a single parameter. Functions that are written to take in more than one parameter are automatically curried.

CHAPTER

# THIRTEEN

# CONTINUATIONS

An running program encompasses two types of state: the data that the program is using and the control state of the program, such as the stack of active functions and the code locations in each of those functions. This control state can be represented in the form of a *continuation*.

A continuation can be *invoked* in order to return control to a previous state. Since a continuation only represents control state, invoking a continuation does **not** return data to their previous state. Instead, data retain the values they had at the time the continuation was invoked. The following is an analogy of invoking a continuation by Luke Palmer:

> Say you're in the kitchen in front of the refrigerator, thinking about a sandwitch [sic]. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwitch, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwitch. But fortunately, there's a sandwitch on the counter, and all the materials used to make it are gone. So you eat it. :-)

In most non-functional languages, a continuation only exists in implicit form, and there is a restricted set of operations that can be done to invoke a continuation. In many functional languages, however, continuations are first-class entities that can be passed as parameters and returned from functions. We first examine restricted forms of continuations before considering the more general, first-class version.

## 13.1 Restricted Continuations

Simple forms of control flow, such as conditionals and loops, do not involve continuations, since they do not return to a previous state of control. Subroutines and exceptions, on the other hand, do revert control to a previous state and thus make implicit use of continuations.

### 13.1.1 Subroutines

Subroutines involve transfer of control between a caller and callee. When a subroutine is called, the control state of the caller must be saved, so that when the subroutine completes, control can be transferred back to the caller. Implementations make use of activation records and call stacks that record the sequence of active calls as well as information about how to return execution to a previous call. These data structures represent the control state of a program and thus constitute a continuation.

Languages restrict how the implicit continuation representing a caller's state can be invoked. In some languages, including many functional languages such as Scheme, the caller's continuation is only invoked when the subroutine completes normally. Other languages have a mechanism to terminate a subroutine early, sometimes called *abrupt termination*, and invoke the continuation of the caller. In imperative languages, this usually takes the form of a *return* statement. For example, the following Python function uses a return to immediately invoke the caller's continuation:

```
def foo(x):
    return x   # invoke caller's continuation
    # more code, but not executed
    if x < 0:
        bar(x)
    baz(x)
    ...
```

As with any continuation, invoking a caller's continuation does not restore the previous state of data. For example, consider the following Python code:

```
def outer():
    x = 0

    def inner():
        nonlocal x
        x += 1

    inner()
    print(x)
```

When the call to `inner()` completes, the continuation of `outer()` is resumed, but the value of `x` is not restored to its state before the call to `inner()`. Instead, it retains its modified value, and the code prints 1.

A more general concept provided by some languages is a *coroutine*, which involves two routines passing control to each other by invoking each other's continuation. Coroutines differ from mutual recursion in that each routine's control state is resumed when it is invoked rather than creating a fresh function invocation with its own state.

The following is pseudocode for coroutines that pass control to each other, with one producing items and the other consuming them:

```
var q := new queue

coroutine produce
    loop
        while q is not full
            create some new items
            add the items to q
        yield to consume

coroutine consume
    loop
        while q is not empty
            remove some items from q
            use the items
        yield to produce
```

Both coroutines yield control to the other. Unlike with subroutines, when a coroutine is passed control, execution resumes from where it previously paused and in the context of the same environment.

Python provides an implementation of coroutines over a tasking layer, with several abstractions for passing data between running coroutines and waiting for completion of an action. The following implements the producer/consumer model using an `asyncio.Queue` for passing values between the producer and consumer:

```
import asyncio

q = asyncio.Queue(2)      # queue capacity of 2

async def produce():
    for i in range(5):
        print('[producer] putting', i)
        await q.put(i)
        print('[producer] done putting', i)

async def consume():
    for i in range(5):
        print('[consumer] got:', await q.get())

loop = asyncio.get_event_loop()
loop.run_until_complete(asyncio.gather(produce(), consume()))
```

The latter two statements start the producer and consumer coroutines running and wait for their completion. The producer passes control to the coroutine returned by `q.put(i)`, which places an item into the queue. Execution will not return to the producer until this completes, so the producer will be forced to wait if the queue is full. The consumer extracts items from the queue using the `q.get()` coroutine, waiting if no items are available. The following is the output when the code is run:

```
[producer] putting 0
[producer] done putting 0
[producer] putting 1
[producer] done putting 1
[producer] putting 2
[consumer] got: 0
[consumer] got: 1
[producer] done putting 2
[producer] putting 3
[producer] done putting 3
[producer] putting 4
[consumer] got: 2
[consumer] got: 3
[producer] done putting 4
[consumer] got: 4
```

This demonstrates how execution passes back and forth between the consumer and producer coroutines.

### 13.1.2 Exceptions

Exceptions also cause control to be passed from one execution state to an earlier one, but unlike returning from a subroutine, the receiver of control need not be the direct caller of a function. Upon entering a `try` block, the control state is saved and the associated exception handlers are added to a stack of active handlers. When an exception is raised, the handler stack is searched for a handler that can accommodate the exception type, the continuation of the associated function is invoked, and the handler code is executed.

As a concrete example, consider the following Python code:

```
def foo(x):
    try:
```

(continues on next page)

(continued from previous page)

```
        bar(x)
    except:
        print('Exception')

def bar(x):
    baz(x)

def baz(x):
    raise Exception

foo(3)
```

When the `try` statement in the invocation of `foo(3)` is reached. the associated exception handler is added to the handler stack. Execution proceeds to the call to `bar(3)` and then to `baz(3)`, which raises an exception. This passes control to the first exception handler that can handle an exception of type `Exception`, which was located in the call to `foo(3)`. Thus, the latter's continuation is invoked and the exception handler is run.

The specific mechanisms used to provide exceptions vary between languages and implementations. Some languages don't incorporate exceptions directly but provide a control mechanism that enables an exception mechanism to be built on top of it. For example, the C standard library header `setjmp.h` defines a `setjmp()` function that saves the execution state of a function, and a corresponding `longjmp()` function that restores the state at the time of the call to `setjmp()`. Exceptions can also be implemented with first-class continuations, as we will see below.

### 13.1.3 Generators

A *generator* is a generalization of a subroutine, allowing its execution to be paused and later resumed. A subroutine is always executed from its entry point, and every entry into a subroutine creates a new activation record. On the other hand, a generator can suspend its execution, and the programmer can resume execution of the generator at the point where its execution state was suspended and using the same activation record. Thus, the paused state of a generator is a form of continuation.

Generators are usually used to write iterators that compute their values lazily. When a generator computes an item, it *yields* the item to its caller by invoking the continuation of the caller, much like a subroutine. Upon resumption of the generator, the next value is computed and yielded to its caller, which need not be the same function as the previous caller.

The following is a generator in Python that produces an infinite sequence of natural numbers:

```
def naturals():
    num = 0
    while True:
        yield num
        num += 1
```

Generators in Python implement the same interface as an iterator, so the next item can be obtained by calling the `next()` function on a generator:

```
>>> numbers = naturals()
>>> next(numbers)
0
>>> next(numbers)
1
```

(continues on next page)

(continued from previous page)

```
>>> next(numbers)
2
```

We can use a generator to represent a range, computing each value as the generator is resumed:

```
def range2(start, stop, step = 1):
    while start < stop:
        yield start
        start += step
```

The sequence of values produced by this generator is finite, and after the last value is produced and the body of `range2()` exits, a `StopIteration` exception is automatically raised:

```
>>> values = range2(0, 10, 3)
>>> next(values)
0
>>> next(values)
3
>>> next(values)
6
>>> next(values)
9
>>> next(values)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A `StopIteration` is used by the Python `for` loop to determine the end of an iterator:

```
>>> for i in range2(0, 10, 3):
...     print(i)
...
0
3
6
9
```

As another example, we can define a unary version of the built-in `map` using a generator:

```
>>> def map_unary(func, iterable):
...     for item in iterable:
...         yield func(item)
...
>>> map_unary(lambda x: x + 1, [1, 4, -3, 7])
<generator object map_unary at 0x1032f3f40>
>>> list(map_unary(lambda x: x + 1, [1, 4, -3, 7]))
[2, 5, -2, 8]
```

The built-in `map` is actually variadic, applying an $n$-ary function to items taken from $n$ iterables:

```
>>> for item in map(lambda x, y: x - y, [1, 2, 3], (-4, -5, -6, -7)):
...     print(item)
...
```

(continues on next page)

(continued from previous page)

```
5
7
9
```

As can be seen from this example, `map` stops when the shortest input iterable is exhausted. We can attempt to write a variadic generator similar to `map`:

```
>>> def map_variadic(func, *iterables):
...     iterators = [iter(it) for it in iterables]
...     items = [0] * len(iterables)
...     while True:
...         for i in range(len(iterators)):
...             items[i] = next(iterators[i])
...         yield func(*items)
...
```

We start by obtaining an iterator from each iterable, and then constructing a list that will hold an element from each iterator, initialized with dummy zero values. We follow this with an infinite loop that obtains the next item from each iterator, storing it in the list, invoking `func` on these items, and yielding the result. When the shortest iterator is exhausted, invoking `next()` on it will raise a `StopIteration`, and our intent was for this to end the `map_variadic()` generator as well. Unfortunately, Python does not allow a `StopIteration` to be propagated out of a generator:

```
>>> list(map_variadic(lambda x, y: x - y, [1, 2, 3], (-4, -5, -6, -7)))
Traceback (most recent call last):
  File "<stdin>", line 6, in map_variadic
StopIteration

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
RuntimeError: generator raised StopIteration
```

Instead, a generator is required to terminate normally or with a `return` when it is complete. Thus, we need to catch the `StopIteration` and then exit:

```
>>> def map_variadic(func, *iterables):
...     iterators = [iter(it) for it in iterables]
...     items = [0] * len(iterables)
...     try:
...         while True:
...             for i in range(len(iterators)):
...                 items[i] = next(iterators[i])
...             yield func(*items)
...     except StopIteration:
...         pass
...
```

Here, we've used a dummy `pass` statement in the `except` clause, since execution will proceed to the end of the generator body and exit. It would be equally valid to use a `return` statement instead. The generator now works as intended:

```
>>> list(map_variadic(lambda x, y: x - y, [1, 2, 3], (-4, -5, -6, -7)))
[5, 7, 9]
```

The `yield from` statement can be used to delegate to another generator (or iterator). The following is a definition of a `positives()` generator that delegates to an instance of `naturals()`:

```
>>> def positives():
...     numbers = naturals()
...     next(numbers)       # discard 0
...     yield from numbers  # yield the remaining items in numbers
...
>>> numbers = positives()
>>> next(numbers)
1
>>> next(numbers)
2
>>> next(numbers)
3
```

We construct a `naturals()` generator, discard the initial 0, and then use `yield from` to produce the remaining items from the `naturals()` generator.

Python also has generator expressions, similar to list comprehensions, that succinctly produce a generator. The following produces a generator of negative integers from `naturals()`:

```
>>> negatives = (-i for i in naturals() if i != 0)
>>> next(negatives)
-1
>>> next(negatives)
-2
>>> next(negatives)
-3
```

As with list comprehensions, the filtering conditional is optional in a generator expression.

Generators are also called *semicoroutines*, since they involve a standard routine that passes control to a resumable routine. Unlike a coroutine, however, a generator can only return control to its caller, while a full coroutine can pass control to any other coroutine.

## 13.2 First-Class Continuations

In some languages, continuations are first-class entities, allowing the current control state to be saved in an explicit data structure, passed as a parameter, and invoked from arbitrary locations. First-class continuations can be used to emulate any of the restricted forms of continuations above. Depending on the language, it may only be permitted to invoke a continuation once, or a continuation may be resumed any number of times.

In Scheme, the `call-with-current-continuation` procedure, often abbreviated as `call/cc`, creates a continuation object representing the current control state. The `call/cc` procedure must be passed an argument:

```
(call-with-current-continuation <procedure>)
```

Here, `<procedure>` must be a Scheme procedure that takes an argument, and `call/cc` invokes this procedure with the newly created continuation object as the argument. The called procedure may use the continuation like any other data item, including discarding it, saving it in a data structure, and returning it, as well as invoking it. For example, in the following code, the procedure discards the continuation and returns a value normally:

```
> (+ 1
     (call/cc (lambda (cc)
                3
              )
     )
  )
4
```

The continuation object constructed by the invocation of `call/cc` above represents the following execution state:

```
(+ 1 <value>)
```

Here, `<value>` replaces the call to `call/cc`, and it will be replaced by the value with which the continuation is invoked.

If the procedure invoked by `call/cc` returns a value normally, the invocation of `call/cc` evaluates to that same value, the same behavior as a standard function call. In the example above, the procedure returns the value 3, which replaces the call to `call/cc`, resulting in a final value of 4.

On the other hand, if the continuation created by `call/cc` is invoked, then control resumes at the location of the `call/cc`. A continuation must be passed a value when it is invoked, and the `call/cc` evaluates to that value:

```
> (+ 1
     (call/cc (lambda (cc)
                (cc 5)
                3
              )
     )
  )
6
```

In the code above, the continuation represents the same execution state of `(+ 1 <value>)`. The function argument of `call/cc` invokes the continuation with value 5, causing execution to immediately resume at the point where `call/cc` is called, with the value 5 replacing the call to `call/cc`, as if it were a standard function call that produced the given value. This results in the execution `(+ 1 5)`, resulting in a final value of 6.

More interesting behavior can occur when a continuation is saved in a variable or data structure. Consider the following:

```
> (define var
    (call/cc (lambda (cc)
               cc
             )
    )
  )
```

The procedure called by `call/cc` returns the continuation, so the `call/cc` invocation evaluates to the continuation, which is then bound to `var`. The continuation itself represents the execution:

```
(define var <value>)
```

We can bind another variable to the same object:

```
> (define cont var)
```

Now we can use this new variable to invoke the continuation:

```
> (cont 3)   ; executes (define var 3)
> var
3
> (cont 4)   ; executes (define var 4)
> var
4
```

Invoking the continuation with a value causes evaluation to resume at the `call/cc`, with the given value replacing the `call/cc`. Thus, invoking `cont` with the value 3 results in the following:

```
(define var
  (call/cc (lambda (cc)
             cc
           )
  )
)
  -->
(define var <value>)
  -->
(define var 3)
```

Thus, `var` is bound to 3. If we invoke `cont` with 4, we get:

```
(define var
  (call/cc (lambda (cc)
             cc
           )
  )
)
  -->
(define var <value>)
  -->
(define var 4)
```

The result is that `var` is now bound to 4.

As a more complex example, consider the following definition of a `factorial` procedure:

```
(define cont '())

(define (factorial n)
  (if (= n 0)
      (call/cc (lambda (cc)
                 (set! cont cc)
                 1
               )
      )
      (* n (factorial (- n 1)))
  )
)
```

The base case is a call to `call/cc`. Then when `(factorial 3)` is called, the execution state when the base case is reached is:

```
(* 3 (* 2 (* 1 <value>)))
```

As before, <value> represents the call to `call/cc`. The argument to `call/cc` sets the global variable `cont` to refer to the newly created continuation and then evaluates normally to 1. The value 1 thus replaces the `call/cc`, resulting in a final value of 6:

```
> (factorial 3)
6
```

If we then invoke the continuation with the value 3, the 3 replaces the `call/cc` in the execution state represented by the continuation:

```
> (cont 3)   ; executes (* 3 (* 2 (* 1 3)))
18
```

If we call `(factorial 5)`, `cont` is modified to refer to a continuation representing the execution:

```
(* 5 (* 4 (* 3 (* 2 (* 1 <value>)))))
```

Invoking the continuation on 4 then results in 480:

```
> (factorial 5)
120
> (cont 4)   ; executes (* 5 (* 4 (* 3 (* 2 (* 1 4)))))
480
```

### 13.2.1 Signaling Errors

We can use first-class continuations to implement a basic mechanism for aborting a computation and signaling an error. We begin with a simple procedure to print an error message:

```
(define (report-error message)
  (begin (display "Error: ")
         (display message)
         (newline)
  )
)
```

This procedure expects to be called with a message string, and it prints out `Error:` followed by the message to standard out. However, invoking the procedure does not abort the computation in the caller. Thus, if we encounter an error in a larger computation, invoking `report-error` causes a message to print but continues where the computation left off. The following is an example:

```
(define (inverse x)
  (if (= x 0)
      (report-error "0 has no inverse")
      (/ 1 x)
  )
)
```

The `inverse` procedure reports an error if the argument `x` is zero. However, it still returns the (undefined) result of calling `report-error` to the caller of `inverse`. This can result in an error at the interpreter level:

```
> (+ (inverse 0) 1)
Error: 0 has no inverse
+: contract violation
  expected: number?
  given: #<void>
  argument position: 1st
  other arguments...:
   1
  context...:
    [context elided]
```

In this Scheme implementation, the `newline` procedure returns a special `#void` value, which gets returned by `report-error` and then by `inverse`. The caller of `inverse` then attempts to add 1 to this result, resulting in an interpreter error.

In order to abort the computation entirely once an error has been signaled, we can make use of a continuation. We arrange for the continuation to save the control state at the top level of a program. but with a following invocation to `report-error` if an error message is provided:

```
(define error-continuation
  (let ((message (call/cc
                   (lambda (c) c)
                 )
        )
       )
    (if (string? message)
        (report-error message)
    )
    message
  )
)
```

Here, the call to `call/cc` saves the control state with the program about to bind the name `message` within a `let` to the result of invoking the continuation. In the initial computation, the continuation object is passed to the `lambda`, which immediately returns it. The call to `call/cc` evaluates to this value, so `message` is bound to the continuation object itself, and the body of the `let` is evaluated. This checks if `message` is a string, calling `report-error` if this is the case. The `let` as a whole evaluates to the value of `message`, which is then bound to `error-continuation` in the global frame.

If we invoke `error-continuation` again, execution will resume at the point of binding `message`, and it will eventually result in `error-continuation` being rebound to something other than the continuation object. To avoid losing the continuation, we can bind another name to it:

```
(define error error-continuation)
```

Now even if `error-continuation` is rebound, the name `error` still refers to the continuation object.

If we invoke `error` with a string, the continuation is invoked with that value, and the value is plugged into where the continuation was created. Thus, `message` is bound to the string, and the body of the `let` is evaluated. Since `message` is a string, `report-error` is called, printing an error message. The let evaluates to the message string, which is then bound to the name `error-continuation` in the global frame. At this point, execution has reached the top level, so computation is completed without causing an error in the interpreter.

If we repeat our previous example, but invoking `error` rather than `report-error`, we get the following:

```
(define (inverse x)
  (if (= x 0)
      (error "0 has no inverse")
      (/ 1 x)
  )
)

> (+ (inverse 0) 1)
Error: 0 has no inverse
```

We no longer have an error reported by the interpreter itself.

### 13.2.2 Call and Return

First-class continuations can be used to emulate the more restricted control constructs provided by imperative languages. For instance, Scheme does not provide a specific mechanism that allows a procedure to terminate abruptly, returning a value to the caller. However, we can emulate call and return, including abrupt returns, with continuations. We do so by explicitly representing the call stack in a data structure that provides push and pop operations:

```
(define call-stack '())
(define (push-call call)
  (set! call-stack (cons call call-stack))
)
(define (pop-call)
  (let ((caller (car call-stack)))
     (set! call-stack (cdr call-stack))
     caller
  )
)
```

We will use this call stack to store a procedure's continuation when it calls another procedure. A return just pops a continuation off the stack and invokes it with the given return value:

```
(define (return value)
  ((pop-call) value)
)
```

We then provide a mechanism for saving a caller's continuation, by pushing it onto the call stack, and invoking the callee. For simplicity, we restrict ourselves to single-argument functions here, but this can be generalized using Scheme's variadic arguments.

```
(define (call func x)
  (call-with-current-continuation (lambda (cc)
                                    (push-call cc)
                                    (func x)
                                  )
  )
)
```

We can then write procedures that use the call stack to terminate abruptly:

```
(define (foo x)
  (if (<= x 10)
```

(continues on next page)

(continued from previous page)

```
      (return x)              ; return x if <= 10
  )
  (let ((y (- x 10)))
    (return (+ x (/ x y)))    ; otherwise return x + x / (x - 10)
  )
  (some more stuff here)      ; control never reaches here
)

(define (bar x)
  (return (- (call foo x)))   ; call foo and return the negation
  (dead code)                 ; control never reaches here
)
```

We can then call `foo` and `bar`:

```
> (+ 1 (call foo 3))
4
> (+ 1 (call foo 20))
23
> (+ 2 (call bar 3))
-1
> (+ 2 (call bar 20))
-20
```

### 13.2.3 Exceptions

We can simulate exception handling with a handler stack, using the same approach as call and return above. The following is a complete implementation:

```
(define handler-stack '())
(define (push-handler handler)
  (set! handler-stack (cons handler handler-stack))
)
(define (pop-handler)
  (let ((handler (car handler-stack)))
    (set! handler-stack (cdr handler-stack))
    handler
  )
)

(define exception-state #f)
(define (set-exception)
  (set! exception-state #t)
)
(define (clear-exception x)
  (set! exception-state #f)
  x
)

(define (throw exception)
  (set-exception)
```

(continues on next page)

(continued from previous page)

```
  ((pop-handler) exception)
)

(define (try func x handler_func)
  (let ((result (call-with-current-continuation (lambda (cc)
                                                  (push-handler cc)
                                                  (func x)
                                                )
                )
        )
        )
    (if exception-state
        (clear-exception (handler_func result))
        result
    )
  )
)
```

We can then define functions that use exceptions:

```
(define (foo x)
  (if (= x 0)
      (throw "invalid argument: 0\n")
      (/ 10 x)
  )
)

(define (bar x)
  (+ (foo x) 1)
)

(define (baz x)
  (try bar x (lambda (exception)
               (display exception)
               '()
             )
  )
)
```

Now we can invoke `baz` with a valid and an erroneous argument:

```
> (baz 2)
5
> (baz 0)
illegal argument: 0
()
```

# Part III

# Theory

We now turn our attention to theoretical foundations of programming languages and the meaning of code. These foundations are crucial to understanding how languages, programs, and their implementations work.

CHAPTER

# FOURTEEN

# LAMBDA CALCULUS

We start by examining lambda calculus, the mathematical foundation of functional programming, and use it to reason about how to construct abstractions and model computations. Its simplicity allows us to understand every detail about how it works, yet it is general enough to enable the expression of arbitrary computations.

Lambda calculus (also $\lambda$-calculus), introduced by Alonzo Church in the 1930s, is a model of computation based on functions. All functions in lambda calculus are anonymous, providing the inspiration for lambda expressions in modern programming languages.

Lambda calculus is composed of only three elements: variables, function abstraction, and function application. *Function abstraction* is the process of defining a new function through a lambda ($\lambda$) expression. The following is a context-free grammar for $\lambda$-calculus:

$$
\begin{aligned}
Expression \;\rightarrow\; & Variable \\
\mid\; & \lambda\, Variable\, .\, Expression \quad \text{(function abstraction)} \\
\mid\; & Expression\; Expression \quad \text{(function application)} \\
\mid\; & (\; Expression\; )
\end{aligned}
$$

We will use individual letters, such as $x$ to denote a variable. Function application is left associative and has higher precedence than abstraction, and we will use parentheses where necessary as a result of associativity and precedence. All functions have exactly one parameter, and functions that would otherwise have multiple parameters must be *curried*.

Since function application is left associative, a sequence of applications such as $f\ g\ h$ is equivalent to $((f\ g)\ h)$. And since function application has higher precedence than abstraction, abstraction extends as far to the right as possible. Consider the following example:

$$\lambda x.\ x\ \lambda y.\ x\ y\ z$$

The $\lambda x$ introduces a function abstraction, which extends as far right as possible:

$$\lambda x.\ \underline{x\ \lambda y.\ x\ y\ z}$$

Thus, this is equivalent to

$$\lambda x.\ (x\ \lambda y.\ x\ y\ z)$$

Then within the parentheses, the $\lambda y$ introduces a new abstraction, which now extends as far right as possible, to the point of the existing closing parenthesis:

$$
\begin{aligned}
&\lambda x.\ (x\ \lambda y.\ \underline{x\ y\ z}) \\
=\ &\lambda x.\ (x\ \lambda y.\ (x\ y\ z))
\end{aligned}
$$

Finally, within the body of the inner abstraction, we have a sequence of function applications, which are left associative:

$$\lambda x.\ (x\ \lambda y.\ ((x\ y)\ z))$$

Using the syntax of Scheme, the following is a representation of the function above:

```
(lambda (x)
  (x (lambda (y)
       ((x y) z)
     )
  )
)
```

(This is merely for illustration. Function semantics are different between Scheme and $\lambda$-calculus, so using this syntax is not meant to imply an equivalence.)

The following is the identity function:

$$\lambda x.\ x$$

The function takes in an argument, binds it to the parameter $x$, and immediately returns it.

Functions themselves are first-class values, so they can be bound to parameters and returned. The following is a function that discards its input and returns the identity function:

$$\lambda y.\ \lambda x.\ x$$

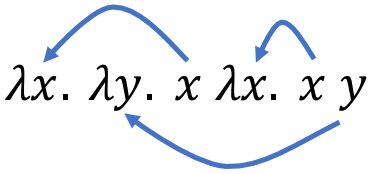
Since abstraction extends as far to the right as possible, this is equivalent to the following parenthesization:

$$\lambda y.\ (\lambda x.\ x)$$

As another example, the following function takes in another function as its argument and applies it to the identity function:

$$\lambda f.\ f\ \lambda x.\ x$$

In $\lambda$-calculus, functions are statically scoped. The result is that in $\lambda x.\ E$, $x$ is bound in $E$, and $E$ is the scope of $x$. If the same name is introduced multiple times within nested scopes, then use of the name resolves to the closest abstraction that introduced it. The following illustrates these rules:




The first abstraction introduces the name $x$, so the scope of $x$ is the body of the first abstraction. Thus, when $x$ appears within the second abstraction, it resolves to the parameter of the first abstraction. The second abstraction itself introduces the name $y$, so use of the name within its body resolves the the associated parameter. Finally, the third abstraction reintroduces the name $x$, so $x$ within its body resolves to the closest introduction, i.e. the parameter of the third abstraction.

An unbound variable is allowed to appear in an expression, and such a variable is called *free*. For example, in $\lambda y.\ x\ y$, $x$ is a free variable in the expression $x\ y$ but $y$ is bound. In $\lambda x.\ \lambda y.\ x\ y$, both $x$ and $y$ are bound in the expression $\lambda y.\ x\ y$. Free variables are useful for reasoning about subexpressions such as $\lambda y.\ x\ y$ in isolation without needing to consider the full context in which the subexpression appears.

In the expression $\lambda x.\ E$, replacing all occurrences of $x$ with another variable $y$ does not affect the meaning as long as $y$ does not occur in $E$. For example, $\lambda y.\ y$ is an equivalent expression of the identity function. This process of variable

replacement is called $\alpha$-*reduction*, and we denote this replacement process as follows:

$$\begin{aligned} & \lambda x.\ x \\ \rightarrow_\alpha\ & \lambda y.\ y \end{aligned}$$

The expressions $\lambda x.\ x$ and $\lambda y.\ y$ are $\alpha$-*equivalent*, and we denote this equivalence is follows:

$$\lambda x.\ x\ =_\alpha\ \lambda y.\ y$$

In function application, $\alpha$-reduction is used to ensure that names are restricted to the appropriate scope. This translation has the same effect as environments in an interpreter. As an example, consider applying the identity function to itself:

$$(\lambda x.\ x)\ (\lambda x.\ x)$$

First, we apply $\alpha$-reduction on the argument to ensure that variables in the argument are distinct from those in the function being applied:

$$\begin{aligned} & (\lambda x.\ x)\ (\lambda x.\ x) \\ \rightarrow_\alpha\ & (\lambda x.\ x)\ (\lambda y.\ y) \end{aligned}$$

We then replace each occurrence of the parameter with the argument expression in the body of the function being applied. The result is the body itself after this substitution process:

$$\begin{aligned} & (\lambda x.\ x)\ (\lambda y.\ y) \\ \Longrightarrow\ & (\cancel{\lambda x.\ x}(\lambda y.\ y)) \end{aligned}$$

This argument-substituting procedure is called $\beta$-*reduction*, and it is similar to the call-by-name argument-passing convention in programming languages. We denote $\beta$-reduction as follows:

$$\begin{aligned} & (\lambda x.\ x)\ (\lambda y.\ y) \\ \rightarrow_\beta\ & \lambda y.\ y \end{aligned}$$

This expression is itself $\alpha$-equivalent to the identity function, and the original expression $(\lambda x.\ x)(\lambda x.\ x)$ is $\beta$-*equivalent* to the identity function since it $\beta$-reduces to the same expression as the identity function:

$$(\lambda x.\ x)\ (\lambda x.\ x)\ =_\beta\ \lambda x.\ x$$

As a more complex example, consider the following:

$$(\lambda x.\ x\ x\ \lambda w.\ \lambda y.\ y\ w)\ \lambda z.\ z$$

In the first function application, the variable names are already distinct, so no $\alpha$-reduction is necessary. We can then apply $\beta$-reduction to obtain:

$$(\lambda z.\ z)\ (\lambda z.\ z)\ \lambda w.\ \lambda y.\ y\ w$$

This results in another function application, where the function and argument do share variable names. Applying $\alpha$-reduction, we get:

$$(\lambda z.\ z)\ (\lambda x.\ x)\ \lambda w.\ \lambda y.\ y\ w$$

This $\beta$-reduces to

$$(\lambda x.\ x)\ \lambda w.\ \lambda y.\ y\ w$$

Another $\beta$-reduction results in

$$\lambda w.\ \lambda y.\ y\ w$$

This cannot $\beta$-reduce any further, so it is said to be in *normal form*. The following denotes the full computation:

$$\begin{aligned}
&(\lambda x.\ x\ x\ \lambda w.\ \lambda y.\ y\ w)\ \lambda z.\ z \\
\rightarrow_\beta\ &(\lambda z.\ z)\ (\lambda z.\ z)\ \lambda w.\ \lambda y.\ y\ w \\
\rightarrow_\alpha\ &(\lambda z.\ z)\ (\lambda x.\ x)\ \lambda w.\ \lambda y.\ y\ w \\
\rightarrow_\beta\ &(\lambda x.\ x)\ \lambda w.\ \lambda y.\ y\ w \\
\rightarrow_\beta\ &\lambda w.\ \lambda y.\ y\ w
\end{aligned}$$

## 14.1 Non-Terminating Computation

Evaluating an expression in $\lambda$-calculus applies $\beta$-reduction as long as possible, until the expression is in normal form. Not all evaluations terminate. Consider a function abstraction that applies an argument to itself:

$$\lambda x.\ x\ x$$

If we apply this to the identity function, we get:

$$\begin{aligned}
&(\lambda x.\ x\ x)\ (\lambda x.\ x) \\
\rightarrow_\alpha\ &(\lambda x.\ x\ x)\ (\lambda y.\ y) \\
\rightarrow_\beta\ &(\lambda y.\ y)\ (\lambda y.\ y) \\
\rightarrow_\alpha\ &(\lambda y.\ y)\ (\lambda z.\ z) \\
\rightarrow_\beta\ &\lambda z.\ z
\end{aligned}$$

This evaluation terminates, and as expected, we obtain the identity function. Now consider what happens when we apply the original function to itself:

$$\begin{aligned}
&(\lambda x.\ x\ x)\ (\lambda x.\ x\ x) \\
\rightarrow_\alpha\ &(\lambda x.\ x\ x)\ (\lambda y.\ y\ y) \\
\rightarrow_\beta\ &(\lambda y.\ y\ y)\ (\lambda y.\ y\ y) \\
\rightarrow_\alpha\ &(\lambda y.\ y\ y)\ (\lambda z.\ z\ z) \\
\rightarrow_\beta\ &(\lambda z.\ z\ z)\ (\lambda z.\ z\ z) \\
&\ldots
\end{aligned}$$

This evaluation never terminates, as reduction continues to produce an expression that is $\alpha$-equivalent to the original one.

## 14.2 Normal-Order Evaluation

Function application in $\lambda$-calculus is similar to *call by name* in that the argument is not evaluated before the function is applied. Instead, the argument expression is substituted for the parameter directly in the body. This results in lazy evaluation, where the argument expression is not evaluated unless it is needed. As an example, consider the following:

$$(\lambda y.\ \lambda z.\ z)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x))$$

The argument expression is a *non-terminating computation*, so if we were to evaluate it prior to substitution, the computation as a whole would not terminate. Instead, $\lambda$-calculus specifies that the substitution happens first:

$$\begin{aligned}
&(\lambda y.\ \lambda z.\ z)\ ((\lambda x.\ x\ x)\ (\lambda x.\ x\ x)) \\
\rightarrow_\beta\ &\lambda z.\ z
\end{aligned}$$

Since the parameter $y$ does not appear in the body, the argument expression is eliminated once the argument substitution is made. Thus, the computation terminates, and its end result is the identity function.

There is an important distinction between the evaluation process in $\lambda$-calculus and call by name. In the former, function bodies are reduced to normal form before the function is applied. This is referred to as *normal-order evaluation*. By contrast, call by name performs argument substitution before manipulating the body of the function. The following illustrates normal-order evaluation:

$$\begin{aligned}
&(\lambda x.\ (\lambda y.\ y\ y)\ x)\ (\lambda z.\ z) \\
\rightarrow_\beta\ &(\lambda x.\ x\ x)\ (\lambda z.\ z) \\
\rightarrow_\beta\ &(\lambda z.\ z)\ (\lambda z.\ z) \\
\rightarrow_\alpha\ &(\lambda z.\ z)\ (\lambda w.\ w) \\
\rightarrow_\beta\ &\lambda w.\ w
\end{aligned}$$

Before the function on the left is applied, its body is reduced, which involves applying the function $\lambda y.\ y\ y$ to its argument $x$. This results in the expression $x$, so the function on the left becomes $\lambda x.\ x\ x$. This is in normal form, so the function can now be applied to its argument. Further $\alpha$- and $\beta$-reductions result in the final value of the identity function.

Summarizing the evaluation rules for a function application $f\ x$, we have the following:

1. Reduce the body of the function $f$ until it is in normal form $f_{normal}$.
2. If a bound-variable name appears in both $f_{normal}$ and $x$, then perform $\alpha$-reduction on $x$ so that this is no longer the case[1], obtaining $x_\alpha$.
3. Perform $\beta$-reduction by substituting $x_\alpha$ for the parameter of $f_{normal}$ in the body of the latter. The result of this reduction is the substituted body itself.
4. Proceed to reduce the substituted body until it is in normal form.

If a variable is free in $f$ but bound in $x$ or vice versa, then $\alpha$-reduction must be applied in step 2 to rename the bound variable. Thus:

$$\begin{aligned}
(\lambda x.\ a\ x)\ \lambda a.\ a\ &\rightarrow_\alpha\ (\lambda x.\ a\ x)\ \lambda y.\ y \\
(\lambda a.\ a\ x)\ a\ &\rightarrow_\alpha\ (\lambda y.\ y\ x)\ a \\
(\lambda x.\ a\ x)\ \lambda a.\ a\ x\ &\rightarrow_\alpha\ (\lambda x.\ a\ x)\ \lambda z.\ z\ x \\
&\rightarrow_\alpha\ (\lambda y.\ a\ y)\ \lambda z.\ z\ x
\end{aligned}$$

## 14.3 Encoding Data

Lambda calculus consists solely of variables and functions, and we can apply $\beta$-reduction to substitute functions for variables. However, none of the familiar values exist directly in $\lambda$-calculus, such as integers or booleans. It is thus surprising that $\lambda$-calculus can model any computational process. We demonstrate this by encoding values as functions.

[1] Our convention is to $\alpha$-reduce the argument rather than the function, though the result of evaluation would be equivalent in either case.

### 14.3.1 Booleans

To start with, let us define an abstraction for the booleans $true$ and $false$. The only building block we have to work with is functions, and we need to ensure that the functions that represent the two values are not $\beta$-equivalent so that we can distinguish between them. There are many ways we can do so, but the one we use is to define $true$ and $false$ as functions that take two values and produce either the first or the second value:

$$\begin{aligned} true &= \lambda t.\ \lambda f.\ t \\ false &= \lambda t.\ \lambda f.\ f \end{aligned}$$

The $=$ sign here means that we take this as a mathematical definition; it does not denote assignment. Since all functions in $\lambda$-calculus must take a single argument, the actual definitions of $true$ and $false$ are *curried*. Applying $true$ to two values results in the first:

$$\begin{aligned} true\ a\ b &= (\lambda t.\ \lambda f.\ t)\ a\ b \\ &\rightarrow_\beta (\lambda f.\ a)\ b \\ &\rightarrow_\beta a \end{aligned}$$

Similarly, applying $false$ to two values yields the second:

$$\begin{aligned} false\ a\ b &= (\lambda t.\ \lambda f.\ f)\ a\ b \\ &\rightarrow_\beta (\lambda f.\ f)\ b \\ &\rightarrow_\beta b \end{aligned}$$

We can proceed to define logical operators as follows:

$$\begin{aligned} and &= \lambda a.\ \lambda b.\ a\ b\ a \\ or &= \lambda a.\ \lambda b.\ a\ a\ b \\ not &= \lambda b.\ b\ false\ true \end{aligned}$$

To see how these work, let us apply them to some examples:

$$\begin{aligned} and\ true\ bool &= ((\lambda a.\ \lambda b.\ a\ b\ a)\ true)\ bool \\ &\rightarrow (\lambda b.\ true\ b\ true)\ bool \\ &\rightarrow (\lambda b.\ b)\ bool \\ &\rightarrow bool \\ or\ true\ bool &= ((\lambda a.\ \lambda b.\ a\ a\ b)\ true)\ bool \\ &\rightarrow (\lambda b.\ true\ true\ b)\ bool \\ &\rightarrow (\lambda b.\ true)\ bool \\ &\rightarrow true \end{aligned}$$

Here, we use $\rightarrow$ on its own to denote some sequence of $\alpha$- and $\beta$-reductions. Applying $and$ to $true$ and any other boolean results in the second boolean, while applying $or$ to $true$ and another boolean always results in $true$. Similarly:

$$\begin{aligned} and\ false\ bool &= ((\lambda a.\ \lambda b.\ a\ b\ a)\ false)\ bool \\ &\rightarrow (\lambda b.\ false\ b\ false)\ bool \\ &\rightarrow (\lambda b.\ false)\ bool \\ &\rightarrow false \\ or\ false\ bool &= ((\lambda a.\ \lambda b.\ a\ a\ b)\ false)\ bool \\ &\rightarrow (\lambda b.\ false\ false\ b)\ bool \\ &\rightarrow (\lambda b.\ b)\ bool \\ &\rightarrow bool \end{aligned}$$

Applying $and$ to $false$ and some other boolean always results in $false$, while applying $or$ to $false$ and another boolean results in the second boolean. Finally, $not$ works as follows:

$$\begin{aligned} not\ true &= (\lambda b.\ b\ false\ true)\ true \\ &\rightarrow true\ false\ true \\ &\rightarrow false \\ not\ false &= (\lambda b.\ b\ false\ true)\ false \\ &\rightarrow false\ false\ true \\ &\rightarrow true \end{aligned}$$

Applying $not$ to $true$ results in $false$, and vice versa.

We can define a conditional as follows:

$$if = \lambda p.\ \lambda a.\ \lambda b.\ p\ a\ b$$

If the condition $p$ is $true$, then applying $p$ to $a$ and $b$ results in $a$, since $true$ selects the first of two values. On the other hand, if $p$ is $false$, then applying $p$ to $a$ and $b$ results in $b$, since $false$ selects the second of two values.

### 14.3.2 Pairs

In order to represent structured data, we need an abstraction for a pair of two values. As with booleans, the only mechanism at our disposal is functions, so we need to produce a "container" function that holds the two values within its body.:

$$pair = \lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y$$

The $pair$ constructor takes two items $x$ and $y$ and produces as a result a function that contains $x$ and $y$ in its body. Applying $pair$ to two concrete items $a$ and $b$ results in:

$$\begin{aligned} pair\ a\ b &= (\lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y)\ a\ b) \\ &\rightarrow_\beta (\lambda y.\ \lambda f.\ f\ a\ y)\ b \\ &\rightarrow_\beta \lambda f.\ f\ a\ b \end{aligned}$$

In order to obtain the first item $a$ above, we can substitute $true$ for $f$, so that $f\ a\ b$ evaluates to $a$. Similarly, to obtain the second item, we can substitute $false$ for $f$. This leads to the following definitions of the $first$ and $second$ selectors:

$$\begin{aligned} first &= \lambda p.\ p\ true \\ second &= \lambda p.\ p\ false \end{aligned}$$

The following demonstrates how selectors work:

$$\begin{aligned} first\ (pair\ a\ b) &= (\lambda p.\ p\ true)\ (pair\ a\ b) \\ &\rightarrow (pair\ a\ b)\ true \\ &= (\lambda f.\ f\ a\ b)\ true \\ &\rightarrow true\ a\ b \\ &\rightarrow a \\ second\ (pair\ a\ b) &= (\lambda p.\ p\ false)\ (pair\ a\ b) \\ &\rightarrow (pair\ a\ b)\ false \\ &= (\lambda f.\ f\ a\ b)\ false \\ &\rightarrow false\ a\ b \\ &\rightarrow b \end{aligned}$$

We can also define a representation for $nil$, as well as a predicate to test for $nil$:

$$\begin{aligned} nil &= \lambda x.\, true \\ null &= \lambda p.\, p\ (\lambda x.\, \lambda y.\, false) \end{aligned}$$

Let us see how the $null$ predicate works:

$$\begin{aligned} null\ nil &= (\lambda p.\, p\ (\lambda x.\, \lambda y.\, false))\ \lambda x.\, true \\ &\rightarrow (\lambda x.\, true)\ (\lambda x.\, \lambda y.\, false) \\ &\rightarrow true \\ null\ (pair\ a\ b) &= (\lambda p.\, p\ (\lambda x.\, \lambda y.\, false))\ (pair\ a\ b) \\ &\rightarrow (pair\ a\ b\ )\ (\lambda x.\, \lambda y.\, false) \\ &= (\lambda f.\, f\ a\ b)\ (\lambda x.\, \lambda y.\, false) \\ &\rightarrow (\lambda x.\, \lambda y.\, false)\ a\ b \\ &\rightarrow (\lambda y.\, false)\ b \\ &\rightarrow false \end{aligned}$$

With a definition for pairs, we can represent arbitrary data structures. For example, we can represent trees using nested pairs:

$$\begin{aligned} tree &= \lambda d.\, \lambda l.\, \lambda r.\, pair\ d\ (pair\ l\ r) \\ datum &= \lambda t.\, first\ t \\ left &= \lambda t.\, first\ (second\ t) \\ right &= \lambda t.\, second\ (second\ t) \end{aligned}$$

### 14.3.3 Church Numerals

Many representations of numbers are possible in $\lambda$-calculus. For example, we can represent natural numbers in unary format, using pairs:

$$\begin{aligned} zero &= \lambda x.\, nil \\ one &= \lambda x.\, pair\ x\ nil \\ two &= \lambda x.\, pair\ x\ (pair\ x\ nil) \\ &\dots \end{aligned}$$

However, the most common representation is the *Church numerals*, which represents a natural number by how many times it applies a function to an input:

$$\begin{aligned} zero &= \lambda f.\, \lambda x.\, x \\ one &= \lambda f.\, \lambda x.\, f\ x \\ two &= \lambda f.\, \lambda x.\, f\ (f\ x) \\ three &= \lambda f.\, \lambda x.\, f\ (f\ (f\ x)) \\ &\dots \end{aligned}$$

A number $n$ is a higher-order function that, given another function $f$, produces a new function that applies $f$ to its argument $n$ times in succession. Using the mathematical notation $f^k$ to denote the composition of $f$ with itself $k$ times, e.g $f^3 = f \circ f \circ f$, the Church numeral $n$ is a function that takes $f$ and produces $f^n$.

As a concrete example, the $right$ function above applies the $second$ function twice to its argument, so we can define it instead as:

$$right = two\ second$$

The following demonstrates how this works[2]:

$$\begin{aligned}
right\ (tree\ a\ b\ c) &= right\ (pair\ a\ (pair\ b\ c)) \\
&= (two\ second)\ (pair\ a\ (pair\ b\ c)) \\
&= ((\lambda f.\ \lambda x.\ f\ (f\ x))\ second)\ (pair\ a\ (pair\ b\ c)) \\
&\rightarrow (\lambda x.\ second\ (second\ x))\ (pair\ a\ (pair\ b\ c)) \\
&\rightarrow second\ (second\ (pair\ a\ (pair\ b\ c))) \\
&\rightarrow second\ (pair\ b\ c) \\
&\rightarrow c
\end{aligned}$$

By applying $right$ to a tree with $c$ as its right subtree, we obtain $c$.

We can define an increment function as follows:

$$incr = \lambda n.\ \lambda f.\ \lambda y.\ f\ (n\ f\ y)$$

Given a number, $incr$ produces a new one that applies a function to an argument one more time than the original number. Thus, where $n$ turns its input $f$ into $f^n$, the result of $incr\ n$ turns its input $f$ into $f^{n+1}$. This is accomplished by first applying $n\ f$, which is equivalent to $f^n$, and then applying $f$ one more time. For example:

$$\begin{aligned}
incr\ zero &= (\lambda n.\ \lambda f.\ \lambda y.\ f\ (n\ f\ y))\ zero \\
&\rightarrow \lambda f.\ \lambda y.\ f\ (zero\ f\ y) \\
&= \lambda f.\ \lambda y.\ f\ ((\lambda x.\ x)\ y) \\
&\rightarrow \lambda f.\ \lambda y.\ f\ y \\
&=_\alpha one \\
incr\ one &= (\lambda n.\ \lambda f.\ \lambda y.\ f\ (n\ f\ y))\ one \\
&\rightarrow \lambda f.\ \lambda y.\ f\ (one\ f\ y) \\
&= \lambda f.\ \lambda y.\ f\ ((\lambda x.\ f\ x)\ y) \\
&\rightarrow \lambda f.\ \lambda y.\ f\ (f\ y) \\
&=_\alpha two
\end{aligned}$$

We can then define $plus$ as follows:

$$plus = \lambda m.\ \lambda n.\ m\ incr\ n$$

This applies the $incr$ function $m$ times to $n$. For example:

$$\begin{aligned}
plus\ two\ three &= (\lambda m.\ \lambda n.\ m\ incr\ n)\ two\ three \\
&\rightarrow (\lambda n.\ two\ incr\ n)\ three \\
&= (\lambda n.\ (\lambda f.\ \lambda x.\ f\ (f\ x))\ incr\ n)\ three \\
&\rightarrow (\lambda n.\ (\lambda x.\ incr\ (incr\ x))\ n)\ three \\
&\rightarrow (\lambda n.\ incr\ (incr\ n))\ three \\
&\rightarrow incr\ (incr\ three) \\
&\rightarrow incr\ four \\
&\rightarrow five
\end{aligned}$$

We can then use the same strategy to define multiplication:

$$times = \lambda m.\ \lambda n.\ m\ (plus\ n)\ zero$$

[2] To simplify reasoning about the results, we depart from normal-order evaluation for the remainder of our discussion on λ-calculus when reducing expressions. In particular, we do not reduce a function body before applying it. However, applying the resulting expressions would have the same effect as those generated by normal-order evaluation.

Here, we perform $m$ additions of $n$, starting at zero, resulting in the product of $m$ and $n$.

We can define exponentiation using the same pattern. Decrement and subtraction are a little more difficult to define, but are possible. Finally, we need a predicate to determine when a number is zero:

$$iszero \;=\; \lambda n.\; n\;(\lambda y.\; false)\;true$$

We apply a number to a function that returns $false$ and a starting value of $true$. Only if the function is never applied is the result $true$, otherwise it is $false$:

$$\begin{aligned}
iszero\;zero &= (\lambda n.\; n\;(\lambda y.\; false)\;true)\;zero \\
&\rightarrow\; zero\;(\lambda y.\; false)\;true \\
&= (\lambda f.\; \lambda x.\; x)\;(\lambda y.\; false)\;true \\
&\rightarrow\; (\lambda x.\; x)\;true \\
&\rightarrow\; true \\
iszero\;two &= (\lambda n.\; n\;(\lambda y.\; false)\;true)\;two \\
&\rightarrow\; two\;(\lambda y.\; false)\;true \\
&= (\lambda f.\; \lambda x.\; f\;(f\;x))\;(\lambda y.\; false)\;true \\
&\rightarrow\; (\lambda x.\; (\lambda y.\; false)\;((\lambda y.\; false\;)\;x))\;true \\
&\rightarrow\; (\lambda x.\; (\lambda y.\; false)\;false)\;true \\
&\rightarrow\; (\lambda x.\; false)\;true \\
&\rightarrow\; false
\end{aligned}$$

## 14.4 Recursion

Church numerals allow us to perform bounded repetition, but in order to express arbitrary computation, we need a mechanism for unbounded repetition. Since $\lambda$-calculus only has functions, recursion is a natural mechanism for repetition.

In recursion, a function needs to be able to refer to itself by name. However, in $\lambda$-calculus, the only way to introduce a name is as a function parameter. Thus, a recursive function must take itself as input. For example, the following defines a factorial function:

$$factorial \;=\; \lambda f.\; \lambda n.\; if\;(iszero\;n)\;one\;(times\;n\;(f\;f\;(decr\;n)))$$

As an analogy, the equivalent form in Python is as follows:

```
>>> factorial = lambda f: (lambda n: 1 if n == 0 else n * f(f)(n-1))
```

In order to actually apply the $factorial$ function, we need another function that applies its argument to itself:

$$apply \;=\; \lambda g.\; g\;g$$

We can then compute a factorial as follows:

$$\begin{aligned}
apply\;factorial\;m &= (\lambda g.\; g\;g)\;factorial\;m \\
&\rightarrow\; factorial\;factorial\;m \\
&= (\lambda f.\; \lambda n.\; if\;(iszero\;n)\;one\;(times\;n\;(f\;f\;(decr\;n))))\;factorial\;m \\
&\rightarrow\; (\lambda n.\; if\;(iszero\;n)\;one\;(times\;n\;(factorial\;factorial\;(decr\;n))))\;m \\
&\rightarrow\; if\;(iszero\;m)\;one\;(times\;m\;(factorial\;factorial\;(decr\;m))) \\
&=_{\beta}\; if\;(iszero\;m)\;one\;(times\;m\;(apply\;factorial\;(decr\;m)))
\end{aligned}$$

$\ldots$

Further evaluation results in the factorial of $m$. Performing the analogous operation in Python:

```
>>> apply = lambda g: g(g)
>>> apply(factorial)(4)
24
```

The *apply* function can be generalized as the following function in $\lambda$-calculus, known as a *fixed-point combinator* and, by convention, the *Y combinator*:

$$Y = \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

Applying the Y combinator to a function $F$ results in:

$$\begin{aligned} Y\ F &= (\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ F \\ &\rightarrow (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \\ &\rightarrow (\lambda x.\ F\ (x\ x))\ (\lambda y.\ F\ (y\ y)) \\ &\rightarrow F\ ((\lambda y.\ F\ (y\ y))\ (\lambda y.\ F\ (y\ y))) \\ &= F\ (Y\ F) \end{aligned}$$

This allows us to define $factorial$ more simply. Let us first define a concrete function $F$:

$$F = \lambda f.\ \lambda n.\ if\ (iszero\ n)\ one\ (times\ n\ (f\ (decr\ n)))$$

Notice that this is the same as $factorial$, except that we have not passed the input function to itself in the recursive application. If we apply the Y combinator to $F$ and apply the result to a number, we get:

$$\begin{aligned} Y\ F\ m &\rightarrow F\ (Y\ F)\ m \\ &= (\lambda f.\ \lambda n.\ if\ (iszero\ n)\ one\ (times\ n\ (f\ (decr\ n))))\ (Y\ F)\ m \\ &\rightarrow (\lambda n.\ if\ (iszero\ n)\ one\ (times\ n\ ((Y\ F)\ (decr\ n))))\ m \\ &\rightarrow if\ (iszero\ m)\ one\ (times\ m\ ((Y\ F)\ (decr\ m))) \end{aligned}$$

Letting $factorial = Y\ F$, we get

$$factorial\ m \rightarrow if\ (iszero\ m)\ one\ (times\ m\ (factorial\ (decr\ m)))$$

Thus, we see that applying the Y combinator to $F$ results in a recursive $factorial$ function, and the Y combinator enables us to write recursive functions in a simpler manner.

## 14.5 Equivalent Models

Lambda calculus models functional programming in its simplest and purest form, and its ability to encode data and perform recursion demonstrates the power of functional programming. It is not the only model for computation, however. Perhaps the most famous model is the *Turing machine*, described by Alan Turing around the same time as Church's work on $\lambda$-calculus. The Turing model is imperative at its core, and it is more closely related to the workings of modern machines than $\lambda$-calculus.

Many variants of Turing machines have been defined, but the following is a description of one variant:

- A *tape* device is used for storage, divided into individual cells in a linear layout. Each cell contains a symbol from a finite alphabet. The tape extends infinitely in both left and right directions.
- A *head* reads and writes symbols from the tape. It can be moved one step at a time to the right or left.
- A *state register* keeps track of the state of the machine. There are a finite number of states the machine can be in, including special start and halt states.

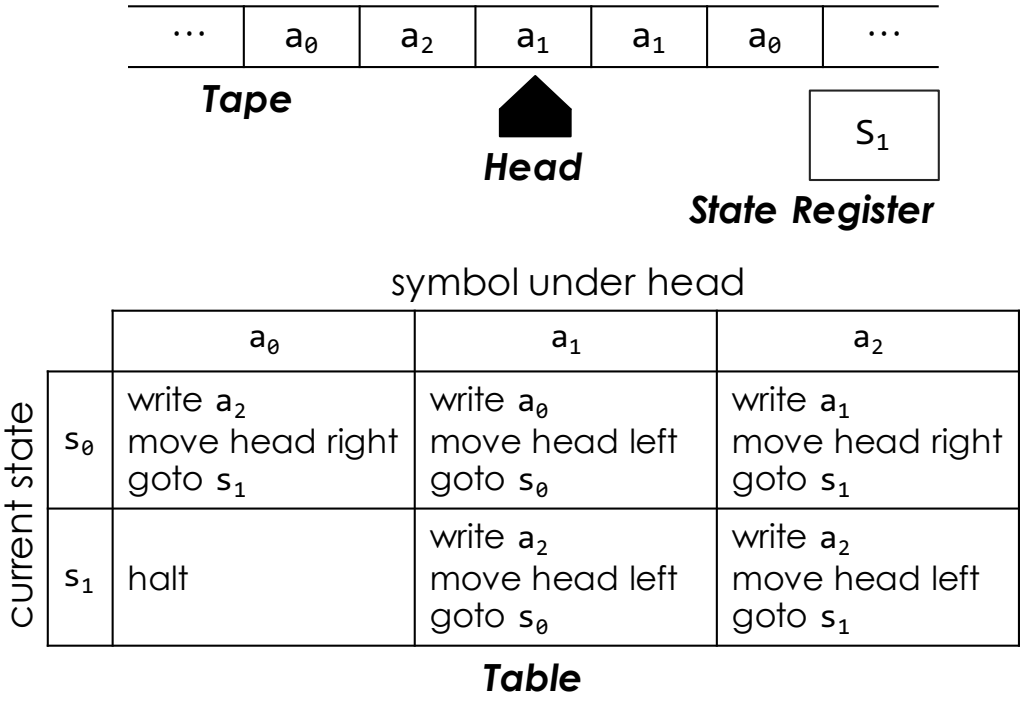



Figure 14.1: An example of a Turing machine.

- A *table* of instructions specifies what the machine is to do for each combination of state and symbol. Since the sets of states and symbols are finite, the instruction table is also finite. At each step in the computation, the machine looks up the current state and the symbol currently under the head in the table and follows the specified instruction.
- An *instruction* can either halt the machine, ending computation, or do the following:
  - Write a specific symbol at the current position of the head.
  - Move the head either one step to the left or the right.
  - Go to a specified new state.

Analogizing with imperative programming, each instruction in a Turing machine can be considered a statement, and each statement transfers control to a new one in a manner similar to a `goto`.

Despite the vastly different model of computation, Alan Turing proved that a Turing machine can solve exactly the same problems as $\lambda$-calculus. This suggests that both models encompass all of computation, a conjecture formalized in the *Church-Turing thesis*. The thesis states that a function is computable by a human following an algorithm if and only if it is computable by a Turing machine, or equivalently, an expression in $\lambda$-calculus.

All known models of computation have been shown to be either computationally equivalent to or weaker than Turing machines. Equivalent models are said to be *Turing complete*. A programming language also defines a model of computation, and all general-purpose programming languages are Turing complete, whether they follow a functional paradigm, an imperative one, or an alternative approach.

CHAPTER

FIFTEEN

# OPERATIONAL SEMANTICS

As mentioned previously, *semantics* is concerned with the meaning of code fragments, as opposed to syntax, which is concerned with their structure. We have seen that syntax can be formally described with regular expressions and context-free grammars. Semantics can also be described formally, and there are a number of approaches. *Denotational semantics* specifies program behavior using set and domain theory, with program fragments described as partial functions over program state. *Axiomatic semantics* is concerned with proving logical assertions over program state, so it specifies the meaning of each construct with respect to its effect on these logical assertions. *Operational semantics* specifies what each computational step does to the state of a program, and what value is computed in each step. Operational semantics more closely describes what an interpreter for a language must perform for each step than denotational or axiomatic semantics.

In this section, we will examine a form of operational semantics known as *structured operational semantics*, and more specifically, *natural* or *big-step* semantics. This form of semantics is particularly well-suited to implementation in a recursive interpreter. We specify rules for how the computation evolves for each syntactic construct in a programming language. We will begin our exploration with a simple imperative language.

## 15.1 Language

Consider a simple imperative language with variables, integers, booleans, statements, conditionals, and loops. The following is a context-free grammar that describes this language:

$$
\begin{aligned}
P &\rightarrow S \\
S &\rightarrow \textbf{skip} \\
&\quad \mid S;\ S \\
&\quad \mid V\ =\ A \\
&\quad \mid \textbf{if}\ B\ \textbf{then}\ S\ \textbf{else}\ S\ \textbf{end} \\
&\quad \mid \textbf{while}\ B\ \textbf{do}\ S\ \textbf{end} \\
A &\rightarrow N \\
&\quad \mid V \\
&\quad \mid (\ A\ +\ A\ ) \\
&\quad \mid (\ A\ -\ A\ ) \\
&\quad \mid (\ A\ *\ A\ ) \\
B &\rightarrow \textbf{true} \\
&\quad \mid \textbf{false} \\
&\quad \mid (\ A\ <=\ A\ ) \\
&\quad \mid (\ B\ \textbf{and}\ B\ ) \\
&\quad \mid \textbf{not}\ B \\
V &\rightarrow Identifier \\
N &\rightarrow IntegerLiteral
\end{aligned}
$$

In order to avoid ambiguities, arithmetic and boolean expressions are parenthesized where necessary, and conditionals and loops end with the `end` keyword. The `skip` statement simply does nothing, and it is equivalent to the empty statement in many languages. It allows us to write conditionals that do nothing in a branch. Variables consist of any identifier, and we will use combinations of letters and numbers to denote them. Any integer can be used as a number literal.

## 15.2 States and Transitions

The *state* of a program consists of a mapping from variables to values, and we will use the lowercase Greek sigma ($\sigma$) to denote a state. In our simple language, variables only hold integer values, and the value of a variable $v$ is specified as $\sigma(v)$. In the initial state, the value of each variable is undefined. We use the notation

$$\sigma[v := n]$$

to denote a state where the value of the variable $v$ has value $n$, but the remaining variables have the same value as in $\sigma$. Formally, we have

$$\sigma[v := n](w) = \begin{cases} n, & \text{if } v = w \\ \sigma(w), & \text{if } v \neq w \end{cases}$$

A *transition* denotes the result of a computation:

$$\langle s, \sigma \rangle \Downarrow \langle u, \sigma' \rangle$$

The left-hand side is the combination of a program fragment $s$ and an initial state $\sigma$. The right-hand side consists of a value $u$ and a new state $\sigma$. The transition as a whole denotes that $s$, when computed in the context of state $\sigma$, results in

the value $u$ and a new state $\sigma'$. If the computation does not produce a value, then no value appears on the right-hand side:

$$\langle s, \sigma \rangle \Downarrow \sigma'$$

Similarly, if the computation does not result in a new state, then the state may be elided from the right-hand side:

$$\langle s, \sigma \rangle \Downarrow u$$

In big-step operational semantics, a transition may only result in a value and/or state. Program fragments may not appear on the right-hand side of a transition. Thus, a transition specifies the complete result of computing a program fragment.

We specify computation in the form of *transition rules*, also called *derivation rules*. They have the following general form:

$$\frac{\langle s_1, \sigma_1 \rangle \Downarrow \langle u_1, \sigma_1' \rangle \quad \ldots \quad \langle s_k, \sigma_k \rangle \Downarrow \langle u_k, \sigma_k' \rangle}{\langle s, \sigma \rangle \Downarrow \langle u, \sigma' \rangle}$$

Only transitions may appear at the top or bottom of a rule. The top of a rule is called the *premise*, and the bottom the *conclusion*. It should thus be read as a conditional rule: if program fragment $s_1$, when computed in state $\sigma_1$, evaluates to value $u_1$ in state $\sigma_1'$, ..., and $s_k$, when computed in state $\sigma_k$, evaluates to $u_k$ in state $\sigma_k'$, then fragment $s$ in state $\sigma$ can evaluate to $u$ in state $\sigma'$. If a computation does not affect the state of the program, then the state may be elided from the right-hand side of a transition. Similarly, if a computation does not result in a value, as in the execution of a statement, then the right-hand side of a transition will not include a value.

A transition rule prescribes how to perform a computation in an interpreter. A particular program fragment $p$ can be interpreted by finding a transition rule where $p$ appears in the conclusion and performing the computations listed in the premise of the rule. The results of these smaller computations are then combined as specified in the rule to produce the result of program fragment $p$. If more than one rule has $p$ in its conclusion, then the interpreter is free to chose which of the rules to apply. Each computational step in a program applies a transition rule, and a program terminates when no more transition rules can be applied.

## 15.3 Expressions

Expressions are generally used for the values to which they evaluate, and in our language, expressions do not have side effects. As a result, the right-hand side of transitions will not include a new state in most of the rules we define below.

### 15.3.1 Arithmetic Expressions

An integer literal evaluates to the respective integer in all cases. The transition rule is as follows, where $n$ denotes an arbitrary integer:

$$\frac{}{\langle n, \sigma \rangle \Downarrow n}$$

A rule like this, with an empty premise, is called an *axiom*. Axioms are the starting point of computation, as we will see below.

A variable, denoted by $v$ below, evaluates to its value as tracked by the state:

$$\frac{}{\langle v, \sigma \rangle \Downarrow \sigma(v)}$$

The rules for addition, subtraction, and multiplication are as follows:

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle (a_1 + a_2), \sigma\rangle \Downarrow n} \quad \text{where } n = n_1 + n_2$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle (a_1 - a_2), \sigma\rangle \Downarrow n} \quad \text{where } n = n_1 - n_2$$

$$\frac{\langle a_1, \sigma\rangle \Downarrow n_1 \quad \langle a_2, \sigma\rangle \Downarrow n_2}{\langle (a_1 * a_2), \sigma\rangle \Downarrow n} \quad \text{where } n = n_1 \times n_2$$

In evaluating $(a_1 + a_2)$ in state $\sigma$, if $a_1$ evaluates to $n_1$ in $\sigma$ and $a_2$ to $n_2$, then $(a_1 + a_2)$ evaluates to the sum of $n_1$ and $n_2$. Similarly for subtraction and multiplication.

The process of evaluating a compound expression results in a derivation tree starting with axioms. For example, consider the evaluation of $((x + 3) * (y - 5))$, where $x$ and $y$ are variables with values 1 and 2, respectively, in state $\sigma$. The full derivation tree is as follows:

$$\frac{\dfrac{\dfrac{}{\langle x, \sigma\rangle \Downarrow 1} \quad \dfrac{}{\langle 3, \sigma\rangle \Downarrow 3}}{\langle (x + 3), \sigma\rangle \Downarrow 4} \quad \dfrac{\dfrac{}{\langle y, \sigma\rangle \Downarrow 2} \quad \dfrac{}{\langle 5, \sigma\rangle \Downarrow 5}}{\langle (y - 5), \sigma\rangle \Downarrow -3}}{\langle ((x + 3) * (y - 5)), \sigma\rangle \Downarrow -12}$$

In this tree, we've applied transition rules to each subexpression to get from axioms to the conclusion that $((x + 3) * (y - 5))$ evaluates to -12 in $\sigma$.

The tree above demonstrates how computation could proceed in an interpreter. The program fragment $((x+3)*(y-5))$ has the form $(a_1*a_2)$, where $a_1 = (x+3)$ and $a_2 = (y-5)$. The interpreter would thus apply the rule for multiplication, which in turn requires computing $(x + 3)$ and $(y - 5)$. The former has the form $(a_1 + a_2)$, so the interpreter would apply the rule for addition, which itself requires the computation of $x$ and 3. The former is a variable, so applying the rule for a variable results in the value 1, while the latter is an integer literal, which evaluates to the value 3 that it represents. Thus, the addition $(x + 3)$ evaluates to the value 4. Repeating the same process for the expression $(y - 5)$ results in the value $-3$, so the full program fragment evaluates to $-12$.

### 15.3.2 Order of Evaluation

If expressions may have side effects, then transitions must include a new state, and we need to consider the order of evaluation of operands. The following rule specifies that the left-hand operand of an addition must be evaluated before the right-hand operand:

$$\frac{\langle a_1, \sigma\rangle \Downarrow \langle n_1, \sigma_1\rangle \quad \langle a_2, \sigma_1\rangle \Downarrow \langle n_2, \sigma_2\rangle}{\langle (a_1 + a_2), \sigma\rangle \Downarrow \langle n, \sigma_2\rangle} \quad \text{where } n = n_1 + n_2$$

In this rule, we've specified that the first operand is to be evaluated in the original state, while the second operand is to be evaluated in the new state produced by evaluating the first operand. The final state is the new state produced by evaluating the second operand.

If, on the other hand, we choose to allow operands to be evaluated in either order, but require that they be evaluated in *some* order, we can introduce a second rule for addition that enables the evaluation to done in reverse order:

$$\frac{\langle a_2, \sigma\rangle \Downarrow \langle n_2, \sigma_2\rangle \quad \langle a_1, \sigma_2\rangle \Downarrow \langle n_1, \sigma_1\rangle}{\langle (a_1 + a_2), \sigma\rangle \Downarrow \langle n, \sigma_1\rangle} \quad \text{where } n = n_1 + n_2$$

Now, in evaluating $(a_1 + a_2)$, we can apply either rule to get either order of evaluation. Thus, implementations are now free to evaluate operands in either order.

### 15.3.3 Boolean Expressions

There are two axioms corresponding to boolean expressions:

$$\frac{}{\langle \textbf{true}, \sigma \rangle \Downarrow \textbf{true}}$$

$$\frac{}{\langle \textbf{false}, \sigma \rangle \Downarrow \textbf{false}}$$

The following are the rules for comparisons, assuming that expressions have no side effects:

$$\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \quad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle (a_1 <= a_2), \sigma \rangle \Downarrow \textbf{true}} \quad \text{if } n_1 \leq n_2$$

$$\frac{\langle a_1, \sigma \rangle \Downarrow n_1 \quad \langle a_2, \sigma \rangle \Downarrow n_2}{\langle (a_1 <= a_2), \sigma \rangle \Downarrow \textbf{false}} \quad \text{if } n_1 > n_2$$

The rules for negation are as follows:

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{true}}{\langle \textbf{not } b, \sigma \rangle \Downarrow \textbf{false}}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{false}}{\langle \textbf{not } b, \sigma \rangle \Downarrow \textbf{true}}$$

Conjunction can be specified as follows:

$$\frac{\langle b_1, \sigma \rangle \Downarrow t_1 \quad \langle b_2, \sigma \rangle \Downarrow t_2}{\langle (b_1 \textbf{ and } b_2), \sigma \rangle \Downarrow t} \quad \text{where } t = t_1 \wedge t_2$$

Notice that this rule does not short circuit: it requires both operands of **and** to be evaluated. If we want short circuiting, we can use the following rules for conjunction instead:

$$\frac{\langle b_1, \sigma \rangle \Downarrow \textbf{false}}{\langle (b_1 \textbf{ and } b_2), \sigma \rangle \Downarrow \textbf{false}}$$

$$\frac{\langle b_1, \sigma \rangle \Downarrow \textbf{true} \quad \langle b_2, \sigma \rangle \Downarrow t_2}{\langle (b_1 \textbf{ and } b_2), \sigma \rangle \Downarrow t_2}$$

Here, the right-hand side need only be evaluated when the left-hand side is true. An interpreter, upon encountering a conjunction, would evaluate the left-hand operand. If the result is false, the first rule must be applied, but if it is true, then the second rule must apply.

## 15.4 Statements

Statements in imperative programs are generally used for their side effects, so they change the state of the program. In our language, statements do not have a value. In our transition rules below, the right-hand side of a transition will be a new state, representing the state that results from completely executing the statement:

$$\langle s, \sigma \rangle \Downarrow \sigma'$$

The intended meaning of such a transition is that executing statement $s$ in state $\sigma$ terminates in a new state $\sigma'$. Not all statements terminate; a statement that does not terminate will not yield a final state through any sequence of transition rules.

The **skip** statement terminates with no effect on the state:

$$\frac{}{\langle \textbf{skip}, \sigma \rangle \Downarrow \sigma}$$

Assignment produces a new state such that the given variable now has the value of the given expression:

$$\frac{\langle a, \sigma \rangle \Downarrow n}{\langle v = a, \sigma \rangle \Downarrow \sigma[v := n]}$$

As described in *States and Transitions*, the notation $\sigma[v := n]$ denotes a state where variable $v$ has the value $n$, but all other variables have the same value is in $\sigma$. Thus, the assignment $v = a$ produces a new state where $v$ has the value that is the result of evaluating $a$, but the remaining variables are unchanged.

Sequencing ensures that the second statement executes in the new state produced from executing the first:

$$\frac{\langle s_1, \sigma \rangle \Downarrow \sigma_1 \quad \langle s_2, \sigma_1 \rangle \Downarrow \sigma_2}{\langle s_1; s_2, \sigma \rangle \Downarrow \sigma_2}$$

Conditionals require separate rules for when the predicate is true or false:

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{true} \quad \langle s_1, \sigma \rangle \Downarrow \sigma_1}{\langle \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end}, \sigma \rangle \Downarrow \sigma_1}$$

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{false} \quad \langle s_2, \sigma \rangle \Downarrow \sigma_2}{\langle \textbf{if } b \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ end}, \sigma \rangle \Downarrow \sigma_2}$$

If the test evaluates to true, then the first rule applies, executing the *then* statement. If the test is false, on the other hand, the second rule applies, executing the *else* statement.

A loop whose predicate is false has no effect:

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{false}}{\langle \textbf{while } b \textbf{ do } s \textbf{ end}, \sigma \rangle \Downarrow \sigma}$$

On the other hand, a loop whose predicate is true has the same effect as executing the body and then recursively executing the loop in the resulting state:

$$\frac{\langle b, \sigma \rangle \Downarrow \textbf{true} \quad \langle s, \sigma \rangle \Downarrow \sigma_1 \quad \langle \textbf{while } b \textbf{ do } s \textbf{ end}, \sigma_1 \rangle \Downarrow \sigma_2}{\langle \textbf{while } b \textbf{ do } s \textbf{ end}, \sigma \rangle \Downarrow \sigma_2}$$

The following demonstrates the execution of the terminating loop **while** $(x <= 2)$ **do** $x = (x + 1)$ **end**, with $x$ having an initial value of 1. Applying a single transition rule for **while**, along with fully evaluating the predicate and executing one iteration of the body, yields:

$$\frac{\dfrac{\dfrac{\overline{\langle x, \sigma \rangle \Downarrow 1} \quad \overline{\langle 2, \sigma \rangle \Downarrow 2}}{\langle (x <= 2), \sigma \rangle \Downarrow \textbf{true}} \quad \dfrac{\dfrac{\overline{\langle x, \sigma \rangle \Downarrow 1} \quad \overline{\langle 1, \sigma \rangle \Downarrow 1}}{\langle (x + 1), \sigma \rangle \Downarrow 2}}{\langle x = (x + 1), \sigma \rangle \Downarrow \sigma[x := 2]} \quad \langle \textbf{while } (x <= 2) \textbf{ do } x = (x + 1) \textbf{ end}, \sigma[x := 2] \rangle \Downarrow \sigma'}{\langle \textbf{while } (x <= 2) \textbf{ do } x = (x + 1) \textbf{ end}, \sigma \rangle \Downarrow \sigma'}$$

Recursively executing the **while** produces the following, where we've truncated the derivation tree for the predicate and body:

$$\frac{\langle (x <= 2), \sigma[x := 2] \rangle \Downarrow \textbf{true} \; \langle x = (x + 1), \sigma[x := 2] \rangle \Downarrow \sigma[x := 3] \; \langle \textbf{while } (x <= 2) \textbf{ do } x = (x + 1) \textbf{ end}, \sigma[x := 3] \rangle \Downarrow \sigma'}{\langle \textbf{while } (x <= 2) \textbf{ do } x = (x + 1) \textbf{ end}, \sigma[x := 2] \rangle \Downarrow \sigma'}$$

One more recursive execution results in:

$$\frac{\langle (x <= 2), \sigma[x := 3] \rangle \Downarrow \textbf{false}}{\langle \textbf{while } (x <= 2) \textbf{ do } x = (x + 1) \textbf{ end}, \sigma[x := 3] \rangle \Downarrow \sigma[x := 3]}$$

This implies that the final state is $\sigma' = \sigma[x := 3]$, so the result of the **while** loop is that $x$ now has value 3.

As an example of a non-terminating or *divergent* computation, consider the loop **while true do skip end**. Applying the transition rule for **while** results in:

$$\frac{\overline{\langle \textbf{true}, \sigma\rangle \Downarrow \textbf{true}} \quad \overline{\langle \textbf{skip}, \sigma\rangle \Downarrow \sigma} \quad \langle \textbf{while true do skip end}, \sigma\rangle \Downarrow \sigma'}{\langle \textbf{while true do skip end}, \sigma\rangle \Downarrow \sigma'}$$

In order to execute the **while** in the premise, we need to recursively apply the same transition rule, producing the same result. This repeats forever, resulting in a divergent computation.

## 15.5 Examples

Operational semantics allows us to reason about the execution of programs, specify equivalences between program fragments, and prove statements about programs. As an example, the following rule specifies an equivalence between two forms of `define` in Scheme:

$$\frac{\langle (\texttt{define}\ f\ (\texttt{lambda}\ (params)\ body)), \sigma\rangle \Downarrow \langle u, \sigma_1\rangle}{\langle (\texttt{define}\ (f\ params)\ body), \sigma\rangle \Downarrow \langle u, \sigma_1\rangle}$$

In Scheme, an expression produces a value but may also have side effects, so the right-hand side of a transition includes a new state. The rule above states that if the expression $(\texttt{define}\ f\ (\texttt{lambda}\ (params)\ body))$ results in a particular value and new state, then the expression $(\texttt{define}\ (f\ params)\ body)$ evaluates to the same value and new state. Thus, an interpreter could handle the latter `define` form by translating it to the former and proceeding to evaluate the translated form.

As another example, in our simple language above, we can demonstrate that swapping operands in an addition is a legal transformation, since $x + y$ and $y + x$ always evaluate to the same value:

$$\frac{\langle x, \sigma\rangle \Downarrow n_x \quad \langle y, \sigma\rangle \Downarrow n_y}{\langle (x + y), \sigma\rangle \Downarrow n} \quad \text{where } n = n_x + n_y$$

$$\frac{\langle y, \sigma\rangle \Downarrow n_y \quad \langle x, \sigma\rangle \Downarrow n_x}{\langle (y + x), \sigma\rangle \Downarrow n} \quad \text{where } n = n_x + n_y$$

## 15.6 Operational Semantics for Lambda Calculus

As another example, we proceed to develop operational semantics for *lambda calculus*. Computation in $\lambda$-calculus does not involve a state that maps variables to values. Thus, transitions have the following form, with neither a state on the left-hand nor on the right-hand side:

$$e_1 \Downarrow e_2$$

An expression $e$ that is in normal form evaluates to itself:

$$\frac{}{e \Downarrow e} \quad \text{where } normal(e)$$

The following defines whether or not an expression is in normal form:

$$\begin{aligned} normal(v) &= true \\ normal(\lambda v.\ e) &= normal(e) \\ normal(v\ e) &= true \\ normal((e_1\ e_2)\ e_3) &= normal(e_1\ e_2) \\ normal((\lambda v.\ e_1)\ e_2) &= false \end{aligned}$$

Here, $v$ denotes a variable, while $e$, and $e_i$ denote arbitrary expressions. A variable is always in normal form, while a function abstraction is in normal form exactly when its body is in normal form. For a function application, if the left-hand side is a variable or application in normal form, then the overall expression is in normal form. On the other hand, if the left-hand side is an abstraction, then a $\beta$-reduction can be applied, so the application is not in normal form.

A function abstraction that is not in normal form is evaluated by evaluating its body:

$$\frac{e_1 \Downarrow e_2}{\lambda v.\ e_1 \Downarrow \lambda v.\ e_2}$$

In a function application, a $\beta$-reduction involves substituting the parameter of a function for its argument in the body of the function, then evaluating the substituted body. Assuming that no variable names are shared between the function and its argument, the following rule specifies this computation:

$$\frac{e_1 \Downarrow e_3 \quad subst(e_3, v, e_2) \Downarrow e_4}{(\lambda v.\ e_1)\ e_2 \Downarrow e_4}$$

The $subst(body, var, arg)$ transformer performs substitution of an expression $arg$ for a variable $var$ in some larger expression $body$. It can be defined as follows:

$$subst(v_1, v, e) = \begin{cases} e \text{ if } v = v_1 \\ v_1 \text{ otherwise} \end{cases}$$

$$subst(\lambda v_1.\ e_1, v, e) = \begin{cases} \lambda v_1.e_1 \text{ if } v = v_1 \\ \lambda v_1.\ subst(e_1, v, e) \text{ otherwise} \end{cases}$$

$$subst(e_1\ e_2, v, e) = subst(e_1, v, e)\ subst(e_2, v, e)$$

A variable is substituted with the argument expression if it is the same as the variable being replaced. Otherwise, substitution has no effect on the variable.

For a function abstraction, if the function's parameter has the same name as the substitution variable, then all references to that name within the body of the function refer to the parameter rather than the substitution variable. Thus, the body should remain unchanged. On the other hand, if the parameter name is different, then the body itself should recursively undergo substitution.

Finally, applying substitution to a function application recursively applies it to both the function and its argument.

The transition rule above for $\beta$-reduction assumes that no $\alpha$-reduction is necessary between a function and its argument. However, $\alpha$-reduction becomes necessary in the following cases:

- The argument contains a bound variable with the same name as a bound or free variable in the function. The following are examples:

$$(\lambda x.\ \lambda y.\ x\ y)\ (\lambda y.\ y)$$
$$(\lambda x.\ x\ y)\ (\lambda y.\ y)$$

- The function contains a bound variable with the same name as a free variable in the argument. The following is an example:

$$(\lambda x.\ \lambda y.\ x\ y)\ y$$

In the first case, our convention is to apply $\alpha$-reduction to the argument, while in the second, we are forced to $\alpha$-reduce the function.

Thus, we need to determine the bound and free variables of an expression. We first define $boundvars(expr)$ to collect the bound variables of an expression $expr$:

$$boundvars(v) = \emptyset$$
$$boundvars(e_1\ e_2) = boundvars(e_1) \cup boundvars(e_2)$$
$$boundvars(\lambda v.\ e) = \{v\} \cup boundvars(e)$$

A variable on its own contributes no bound variables. The bound variables of a function application are the union of the bound variables in the function and its argument. The bound variables of a function abstraction are the bound variables of the body plus the parameter of the function itself.

In order to determine the free variables of an expression, we require as input the set of variables that are bound when the expression is encountered. We define $freevars(bound, expr)$ as follows:

$$freevars(bound, v) = \begin{cases} \{v\} \text{ if } v \notin bound \\ \emptyset \text{ otherwise} \end{cases}$$
$$freevars(bound, e_1\ e_2) = freevars(bound, e_1) \cup freevars(bound, e_2)$$
$$freevars(bound, \lambda v.\ e) = freevars(bound \cup \{v\}, e)$$

A variable is free if it is not included in the bound set. The free variables of a function application are the union of the free variables in the function and its argument. The free variables of a function abstraction are the free variables in the body, using a bound set that includes the parameter of the abstraction.

We can also define a transformer $alpha(vars, expr)$ to rename the bound variables in $expr$ that occur in the set $vars$:

$$alpha(vars, v) = v$$
$$alpha(vars, e_1\ e_2) = alpha(vars, e_1)\ alpha(vars, e_2)$$
$$alpha(vars, \lambda v.\ e) = \begin{cases} \lambda w.\ alpha(vars, subst(e, v, w)) \text{ if } v \in vars, \text{ where } w \text{ is fresh} \\ \lambda v.\ alpha(vars, e) \text{ otherwise} \end{cases}$$

A variable on its own is not bound, so it should not be renamed. A function application is renamed by renaming both the function and its argument. For a function abstraction, if the parameter appears in $vars$, we replace it with a new name that is *fresh*, meaning that it is not used anywhere in the program. This requires applying substitution to the body, replacing the old variable name with the new one. We also have to recursively apply renaming to the body, whether the parameter is replaced or not.

To put this all together, we define a transformer $beta(func, arg)$ for performing $\beta$-reduction when $func$ is applied to $arg$:

$$alpha_{arg}(func, arg) = alpha(boundvars(func) \cup freevars(\emptyset, func), arg)$$
$$alpha_{func}(func, arg) = alpha(freevars(\emptyset, arg), func)$$
$$beta(func, arg) = subst(e', v', alpha_{arg}(func, arg)), \text{ where } \lambda v'.\ e' = alpha_{func}(func, arg)$$

Here, $alpha_{arg}(func, arg)$ applies renaming to $arg$ given the bound and free variables in $func$, and $alpha_{func}(func, arg)$ applies renaming to $func$ given the free variables in $arg$. The result must be an abstraction of the form $\lambda v'.\ e'$, so $beta(func, arg)$ proceeds to substitute $v'$ for the renamed argument in the body $e'$.

We can now proceed to write a general transition rule for $\beta$-reduction:

$$\frac{e_1 \Downarrow e_3 \quad beta(\lambda v.\ e_3, e_2) \Downarrow e_4}{(\lambda v.\ e_1)\ e_2 \Downarrow e_4}$$

Finally, we need a transition rule for a sequence of function applications:

$$\frac{e_1\ e_2 \Downarrow e_4 \quad e_4\ e_3 \Downarrow e_5}{(e_1\ e_2)\ e_3 \Downarrow e_5}$$

We can apply the rules to derive the following computation for $(\lambda x.\ \lambda y.\ y\ x)\ x\ a$:

$$\frac{\dfrac{\dfrac{}{\lambda y.\ y\ x \Downarrow \lambda y.\ y\ x} \quad \dfrac{}{beta(\lambda x.\ \lambda y.\ y\ x, x) = \lambda y.\ y\ x \Downarrow \lambda y.\ y\ x}}{(\lambda x.\ \lambda y.\ y\ x)\ x \Downarrow \lambda y.\ y\ x} \quad \dfrac{\dfrac{}{y\ x \Downarrow y\ x} \quad \dfrac{}{beta(\lambda y.\ y\ x, a) = a\ x \Downarrow a\ x}}{(\lambda y.\ y\ x)\ a \Downarrow a\ x}}{(\lambda x.\ \lambda y.\ y\ x)\ x\ a \Downarrow a\ x}$$

Here, we've applied the rule for a sequence of function applications, then applied the $\beta$-reduction rule to each of the premises. The end result is $a\ x$.

CHAPTER

# SIXTEEN

# FORMAL TYPE SYSTEMS

We now turn our attention to type systems and *type checking*, which determines whether or not the use of types in a program is correct. Given a language, we will define rules to determine the type of each expression in the language. Where the rules do not assign a type for a particular expression, that expression should be considered erroneous.

We start with a simple language of boolean and integer expressions, parenthesized where necessary to avoid ambiguity:

$$
\begin{aligned}
P &\rightarrow E \\
E &\rightarrow N \\
&\quad \mid B \\
&\quad \mid (\ E\ +\ E\ ) \\
&\quad \mid (\ E\ -\ E\ ) \\
&\quad \mid (\ E\ *\ E\ ) \\
&\quad \mid (\ E\ <=\ E\ ) \\
&\quad \mid (\ E\ \textbf{and}\ E\ ) \\
&\quad \mid \textbf{not}\ E \\
&\quad \mid (\ \textbf{if}\ E\ \textbf{then}\ E\ \textbf{else}\ E\ ) \\
N &\rightarrow IntegerLiteral \\
B &\rightarrow \textbf{true} \\
&\quad \mid \textbf{false}
\end{aligned}
$$

The two types in this language are $Int$ for integer expressions and $Bool$ for boolean expressions. We use the notation $t\ :\ T$ to denote that a term $t$ has the type $T$. A statement of the form $t\ :\ T$ is often called a *typing relation* or *type judgment*.

The base typing rules assign types to integer and boolean literals:

$$\frac{}{IntegerLiteral\ :\ Int}$$

$$\frac{}{\textbf{true}\ :\ Bool}$$

$$\frac{}{\textbf{false}\ :\ Bool}$$

For more complex expressions, we have derivation rules that are similar to those in *operational semantics*, where the top of the rule is the *premise* and the bottom the *conclusion*. The following is the rule for addition:

$$\frac{t_1\ :\ Int \qquad t_2\ :\ Int}{(t_1 + t_2)\ :\ Int}$$

This rule states that if $t_1$ has type $Int$, and $t_2$ has type $Int$, then the term $(t_1 + t_2)$ also has type $Int$. Thus, the rule allows us to compute the type of a larger expression from the types of the subexpressions, as in the following derivation:

$$\dfrac{\dfrac{}{1 \ : \ Int} \qquad \dfrac{\dfrac{}{3 \ : \ Int} \qquad \dfrac{}{5 \ : \ Int}}{(3 + 5) \ : \ Int}}{(1 + (3 + 5)) \ : \ Int}$$

On the other hand, an expression such as $(\textbf{true} + 1)$ is not well typed: since $\textbf{true} \ : \ Bool$, the premise in the rule for addition does not hold, so it cannot be applied to derive a type for $(\textbf{true} + 1)$. Since no type can be derived for the expression, the expression does not type check, and it is erroneous.

The following rules for subtraction and multiplication are similar to that of addition:

$$\dfrac{t_1 \ : \ Int \qquad t_2 \ : \ Int}{(t_1 - t_2) \ : \ Int}$$

$$\dfrac{t_1 \ : \ Int \qquad t_2 \ : \ Int}{(t_1 * t_2) \ : \ Int}$$

The rule for comparison requires that the two operands have type $Int$, in which case the type of the overall expression is $Bool$:

$$\dfrac{t_1 \ : \ Int \qquad t_2 \ : \ Int}{(t_1 <= t_2) \ : \ Bool}$$

The rule for conjunction requires that the operands have type $Bool$, and the resulting expression also has type $Bool$. Negation similarly requires its operand to have type $Bool$:

$$\dfrac{t_1 \ : \ Bool \qquad t_2 \ : \ Bool}{(t_1 \ \textbf{and} \ t_2) \ : \ Bool}$$

$$\dfrac{t \ : \ Bool}{\textbf{not} \ t \ : \ Bool}$$

The conditional expression requires the test to have type $Bool$. However, the only restrictions on the remaining two operands is that they are well typed, and that they both have the same type. For example, an expression such as $(\textbf{if} \ test \ \textbf{then} \ 3 \ \textbf{else} \ 5)$ will always produce an integer, regardless of the value of $test$, while $(\textbf{if} \ test \ \textbf{then} \ false \ \textbf{else} \ true)$ will always produce a boolean. Thus, our typing rule has a type variable $T$ to represent the type of the last two operands, ensuring that they match:

$$\dfrac{t_1 \ : \ Bool \qquad t_2 \ : \ T \qquad t_3 \ : \ T}{(\textbf{if} \ t_1 \ \textbf{then} \ t_2 \ \textbf{else} \ t_3) : T}$$

## 16.1 Variables

Now that we have typing rules for a simple language of booleans and integers, we proceed to introduce variables into the language. For the purposes of typing, we will assume that each variable in a program has a distinct name. As we saw in *lambda calculus*, we can rename variables if necessary so that this is the case.

We introduce the following syntax for a binding construct to the language:

$$\begin{aligned} E \ &\rightarrow (\ \textbf{let} \ V \ = \ E \ \textbf{in} \ E \ ) \\ &\quad \mid V \\ V \ &\rightarrow Identifier \end{aligned}$$

The semantics of this construct are to replace all occurrences of the given variable in the body of the **let** with the variable's bound value. Thus, an expression such as the following should produce an integer:

$$(\textbf{let } x = 3 \textbf{ in } (x + 2))$$

On the other hand, the following expression should not type check, since replacing $x$ with its bound value results in an ill-typed body:

$$(\textbf{let } x = 3 \textbf{ in not } x)$$

In order to determine whether or not the body of a **let** is well typed, we need a *type context* or *type environment* that keeps track of the type of the variables that are in scope. The following is the notation we use for a type environment:

- The symbol $\Gamma$ represents a type environment.
- The notation $x : T \in \Gamma$ denotes that $\Gamma$ maps the name $x$ to the type $T$.
- We extend a type environment as $\Gamma, x : T$, which denotes the type environment that assigns the type $T$ to $x$ but assigns all other variables the same type as in $\Gamma$.
- We express a type judgment as $\Gamma \vdash t : T$, which states that the term $t$ has type $T$ within the context of the type environment $\Gamma$.

As indicated by the last point above, type judgments are now made in the context of a type environment that maps variables to their types. If a particular term has the same type regardless of typing environment, then we can elide the environment in a type judgment. For example, the judgment $\vdash$ **true** : $Bool$ indicates that **true** always has type $Bool$ within the context of any type environment.

The following are our existing typing rules using the notation of type environments:

$$\frac{}{\vdash IntegerLiteral : Int}$$

$$\frac{}{\vdash \textbf{true} : Bool}$$

$$\frac{}{\vdash \textbf{false} : Bool}$$

$$\frac{\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int}{\Gamma \vdash (t_1 + t_2) : Int}$$

$$\frac{\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int}{\Gamma \vdash (t_1 - t_2) : Int}$$

$$\frac{\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int}{\Gamma \vdash (t_1 * t_2) : Int}$$

$$\frac{\Gamma \vdash t_1 : Int \qquad \Gamma \vdash t_2 : Int}{\Gamma \vdash (t_1 <= t_2) : Bool}$$

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : Bool}{\Gamma \vdash (t_1 \textbf{ and } t_2) : Bool}$$

$$\frac{\Gamma \vdash t : Bool}{\Gamma \vdash \textbf{not } t : Bool}$$

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : T \qquad \Gamma \vdash t_3 : T}{\Gamma \vdash (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) : T}$$

We need a rule for typing a variable:

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T}$$

Here, we use $v$ to denote a variable. The rule states that if the type environment $\Gamma$ maps the variable $v$ to type $T$, then the term consisting of $v$ itself has type $T$ within the context of $\Gamma$.

We can now add a rule for the **let** binding construct:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, v : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\textbf{let}\ v = t_1\ \textbf{in}\ t_2) : T_2}$$

Here, we use $v$ to denote the name of the variable introduced by the **let**. The rule states that if the initializer expression is assigned the type $T_1$ within the context of the original type environment $\Gamma$, and the body has type $T_2$ within the context of the original environment extended with the mapping $v : T_1$, then the overall **let** expression also has type $T_2$. We can use this to derive the type of our first **let** example in the context of any type environment:

$$\frac{\dfrac{}{\vdash 3 : Int} \qquad \dfrac{\dfrac{\dfrac{}{x : Int \in x : Int}}{x : Int \vdash x : Int} \qquad \dfrac{}{x : Int \vdash 2 : Int}}{x : Int \vdash (x + 2) : Int}}{\vdash (\textbf{let}\ x = 3\ \textbf{in}\ (x + 2)) : Int}$$

## 16.2 Functions

Now that we have typing rules for expressions of booleans and integers, we proceed to add functions to our language and introduce rules for computing the types of function abstractions and applications. As in *lambda calculus*, we will consider functions that take in exactly one argument. A function then has two types that are relevant: the type of the argument to the function, and the type of its return value. We will use the notation $T_1 \to T_2$ to denote the type of a function that takes in an argument of type $T_1$ and returns a value of type $T_2$.

For simplicity, we will require that the parameter type of a function be explicitly specified. It would also be reasonable to infer the type of the parameter from how it is used in the body, or to deduce the type of the parameter independently each time the function is applied to an argument. The latter would provide a form of *parametric polymorphism*. However, we will not consider such schemes here.

To allow functions to be defined, with explicit typing of parameters, we extend our language as follows:

$$\begin{aligned} E &\to (\ \textbf{lambda}\ V\ :\ T\ .\ E\ ) \\ &\quad \mid (\ E\ E\ ) \\ T &\to Int \\ &\quad \mid Bool \\ &\quad \mid T\ \to\ T \\ &\quad \mid (\ T\ ) \end{aligned}$$

We introduce two new expressions, one for function abstraction and one for function application, borrowing syntax from $\lambda$-calculus. We also introduce types into our grammar, with $Int$ and $Bool$ as the non-function types. A function type is specified by separating its input and output types by the *type constructor* $\to$. When chained, the type constructor is right associative, so that $Int \to Int \to Bool$ is equivalent to $Int \to (Int \to Bool)$, denoting a function that takes in an $Int$ and returns a function with type $Int \to Bool$.

As with **let**, we will assume that parameter names introduced by **lambda** expressions are distinct from any other names in the program, knowing that we can always rename variables to ensure that this is the case.

We can now define the typing rule for abstraction as follows:

$$\frac{\Gamma, v : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\textbf{lambda}\ v : T_1.\ t_2) : T_1 \to T_2}$$

The rule states that if the body $t_2$ has type $T_2$ in the type environment that consists of $\Gamma$ extended with the mapping $v : T_1$ for the parameter, then the function as a whole has type $T_1 \to T_2$. Thus, the function takes in a value of type $T_1$ as an argument and returns a value of type $T_2$.

The following is the rule for application:

$$\frac{\Gamma \vdash t_1 : T_2 \to T_3 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1\ t_2) : T_3}$$

This states that if the function has type $T_2 \to T_3$, taking in a $T_2$ and returning a $T_3$, and the argument has the requisite type $T_2$, then the application results in the type $T_3$.

As an example, consider the following program fragment:

$$(\textbf{let}\ f = (\textbf{lambda}\ x : Int.\ (x <= 10))\ \textbf{in}\ (f\ 3))$$

We can derive the type of this expression in any type environment as follows:

$$\dfrac{\dfrac{\dfrac{x : Int \in x : Int}{x : Int \vdash x : Int} \qquad \dfrac{}{x : Int \vdash 10 : Int}}{x : Int \vdash (x <= 10) : Bool}}{\vdash (\textbf{lambda}\ x : Int.\ (x <= 10)) : Int \to Bool} \qquad \dfrac{\dfrac{f : Int \to Bool \in f : Int \to Bool}{f : Int \to Bool \vdash f : Int \to Bool} \qquad \dfrac{}{f : Int \to Bool \vdash 3 : Int}}{f : Int \to Bool \vdash (f\ 3) : Bool}$$

$$\vdash (\textbf{let}\ f = (\textbf{lambda}\ x : Int.\ (x <= 10))\ \textbf{in}\ (f\ 3)) : Bool$$

At the bottom of the derivation, we apply the rule for **let**, requiring us to compute the type of the variable initializer as well as the type of the body in a type environment where the new variable has its computed type.

To compute the type of the initializer, we apply the rule for abstraction, requiring us to compute the type of the body in a type environment with the function parameter having its designated type of $Int$. This applies the rule for $<=$, further requiring computation of types for the variable $x$ and integer literal 10. The body then has the type $Bool$, so the abstraction has type $Int \to Bool$.

We can then compute the type of the body of the $let$, in a type context where $f$ has type $Int \to Bool$. This requires us to apply the rule for function application, computing the type of both the function and its argument. The function is the variable $f$, which has type $Int \to Bool$ in the type environment. The argument is the integer literal 3, which has type $Int$. Thus, the application is applying an $Int \to Bool$ to an $Int$, resulting in $Bool$. This is also the type of the **let** expression as a whole.

## 16.3 Subtyping

Our working language now has the base types $Bool$ and $Int$, as well as function types. Let us extend the language by adding floating-point numbers:

$$\begin{aligned} E &\to F \\ F &\to FloatingLiteral \\ T &\to Float \end{aligned}$$

We add a typing rule for floating-point literals:

$$\frac{}{\vdash FloatingLiteral : Float}$$

We would also like to allow operations such as addition on expressions of type $Float$. We could define a separate rule for adding two $Float$s:

$$\frac{\Gamma \vdash t_1 : Float \qquad \Gamma \vdash t_2 : Float}{\Gamma \vdash (t_1 + t_2) : Float}$$

However, the combination of this rule and the rule for adding $Int$s does not permit us to add a $Float$ and an $Int$. Adding more rules for such a combination is not a scalable solution: introducing more numerical types would result in a combinatorial explosion in the number of rules required.

Functions pose a similar problem. If we define a function such as (**lambda** $x : Float.\ (x + 1.0)$), we would like to be able to apply it to an $Int$ as well as a $Float$. Conceptually, every integer is also a floating-point number[3], so we would expect such an operation to be valid.

Rather than adding more rules to permit this specific case, we introduce a notion of *subtyping* that allows a type to be used in contexts that expect a different type. We say that type $S$ is a *subtype* of type $T$ if a term of type $S$ can be substituted anywhere a term of type $T$ is expected. We use the notation $S <: T$ to denote that $S$ is a subtype of $T$.

The subtype relation $<:$ must satisfy the following requirements:

- It is *reflexive*, meaning that for any type $S$, it must be that $S <: S$, so that $S$ is a subtype of itself.
- It is *transitive*, so that $S <: T$ and $T <: U$ implies that $S <: U$.

Thus, the subtype relation must be a *preorder*. In many languages, the subtype relation is also a *partial order*, additionally satisfying the following:

- It is *antisymmetric*, so that $S <: T$ and $T <: S$ implies that $S = T$.

In our working language, we specify that $Int$ is a subtype of $Float$:

$$Int <: Float$$

To allow our type system to accommodate subtyping, we introduce a new typing rule, called the *subsumption* rule, to enable a subtype to be used where a supertype is expected:

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

This rule states that if the type of term $t$ has been computed as $S$, and $S$ is a subtype of $T$, then we can also conclude that $t$ has type $T$. This allows a function that expects a $Float$ to be applied to an $Int$ as well:

$$\frac{\Gamma \vdash f : Float \to Float \qquad \dfrac{\Gamma \vdash x : Int \qquad Int <: Float}{\Gamma \vdash x : Float}}{\Gamma \vdash (f\ x) : Float}$$

### 16.3.1 Subtyping and Arithmetic Operators

It may be tempting to rewrite the rules for arithmetic operators on numbers to require both operands to be of the $Float$ type:

$$\frac{\Gamma \vdash t_1 : Float \qquad \Gamma \vdash t_2 : Float}{\Gamma \vdash (t_1 + t_2) : Float}$$

However, such a rule always produces a $Float$ as a result. This precludes us from using the result as an argument to a function that expects an $Int$ as its argument: it is not the case that $Float <: Int$, so we cannot use a $Float$ in a context that requires $Int$.

Instead, we need to rewrite the rule such that it produces a $Float$ when at least one of the operands is a $Float$, but it results in an $Int$ if both operands are $Int$s. More generally, we desire the following, where $T_1$ and $T_2$ are the types of the two operands:

- Both operands are of numerical type. In our language, this means that they are each of a type that is some subtype of $Float$. Thus, we require that $T_1 <: Float$ and $T_2 <: Float$.

[3] This may not actually be the case in the implementation, depending on the representation used for the two types. However, it still makes sense semantically that an integer should be allowed where a floating-point number is expected.

- The result is the *least upper bound*, or *join*, of the two operand types. This means that the result type is the minimal type $T$ such that $T_1 <: T$ and $T_2 <: T$. We use the notation[4] $T = T_1 \sqcup T_2$ to denote that $T$ is the join of $T_1$ and $T_2$.

  Since $S <: S$, it is always the case that $S = S \sqcup S$. Thus, in our language, we have $Int = Int \sqcup Int$, $Float = Float \sqcup Float$, and $Float = Int \sqcup Float$.

Putting these requirements together, we can define the typing rule for addition as follows:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 <: Float \qquad T_2 <: Float \qquad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 + t_2) : T}$$

Subtraction and multiplication can be similarly defined.

### 16.3.2 The Top Type

Many languages include a $Top$ type, also written as $\top$, that is a supertype of every other type in the language. Thus, for any type $S$, we have:

$$S <: Top$$

The $Top$ type corresponds to the $Object$ type in many object-oriented languages. For example, the `object` type in Python is a supertype of every other type.

Introducing $Top$ into our language ensures that a join exists for every pair of types in the language. However, it is not necessarily the case in general that a particular language has a join for every pair of types, even if it has a $Top$ type.

The existence of a join for each pair of types allows us to loosen the typing rule for conditionals:

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T = T_2 \sqcup T_3}{\Gamma \vdash (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) : T}$$

Rather than requiring that both branches have exactly the same type, we allow each branch to have an arbitrary type. Since we can always compute the join of the two types in our language, the resulting type of the conditional is the join of the types of the branches.

### 16.3.3 Subtyping and Functions

In a language with higher-order functions, subtyping is also applicable to function types. There are contexts where it would be semantically valid to accept a function type that is different from the one that is expected. For instance, consider the following higher-order function:

$$(\textbf{lambda } f : Int \to Bool.\ (f\ 3))$$

This function takes in another function $f$ as an argument and then applies $f$ to an $Int$. If the actual function provided as an argument had type $Float \to Bool$ instead, it would still be semantically valid to invoke it on an $Int$. Thus, it should be the case that $Float \to Bool <: Int \to Bool$, since the former can be used in contexts that expect the latter.

Now consider another higher-order function:

$$(\textbf{lambda } f : Int \to Float.\ (f\ 3))$$

This new function takes in a function $f$ and applies it to an $Int$ to produce a $Float$. However, if the function we provide as the argument has type $Int \to Int$, it would produce an $Int$; the latter is a valid substitution for a $Float$, making such an argument semantically valid. Thus, it should also be the case that $Int \to Int <: Int \to Float$.

[4] The symbols $\vee$ and $\cup$ are also commonly used to denote the least upper bound. However, we will stick to $\sqcup$ to avoid confusion with disjunction and set union.

Putting both cases together, we end up with the following subtyping rule for functions:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

A function that accepts parameters of type $S_1$ accepts more general argument values than one that accepts type $T_1$; the former has a more general *domain* than the latter. Any contexts that expect to pass a $T_1$ as an argument would be just as well served if the function accepts an $S_1$. Thus, the function type that accepts an $S_1$ should be substitutable for the function type that accepts a $T_1$.

A function that produces a return value of type $S_2$ has a more restricted set of outputs, or *codomain*, than a function that produces a $T_2$. Any context that expects a $T_2$ as output would be just as well served by an $S_2$ as output. Thus, the function type that produces an $S_2$ should be substitutable for the function type that produces a $T_2$.

The subtyping rule permits a *contravariant* parameter type in the function subtype: it is contravariant since the direction of the relation $<:$ is reversed for the parameter types compared to the relation for the function types. The rule also permits a *covariant* return type, since the direction of $<:$ is the same for the return types and the function types.

Covariant return types often appear in object-oriented languages in a different context, that of overriding a base class's method, for the same semantic reasons they are valid here. We will discuss *covariance and contravariance* in object-oriented programming in more detail later.

## 16.4 Full Typing Rules

Putting together all the features we have discussed, the following are the rules for subtyping:

- $Top$:

$$S <: Top$$

- Numbers:

$$Int <: Float$$

- Functions:

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{S_1 \to S_2 <: T_1 \to T_2}$$

- Subsumption:

$$\frac{\Gamma \vdash t : S \qquad S <: T}{\Gamma \vdash t : T}$$

The typing rules for each kind of term are as follows:

- Literals:

$$\frac{}{\vdash IntegerLiteral : Int}$$

$$\frac{}{\vdash \textbf{true} : Bool}$$

$$\frac{}{\vdash \textbf{false} : Bool}$$

$$\frac{}{\vdash FloatingLiteral : Float}$$

- Arithmetic:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 <: Float \qquad T_2 <: Float \qquad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 + t_2) : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 <: Float \qquad T_2 <: Float \qquad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 - t_2) : T}$$

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 <: Float \qquad T_2 <: Float \qquad T = T_1 \sqcup T_2}{\Gamma \vdash (t_1 * t_2) : T}$$

- Comparisons[5]:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma \vdash t_2 : T_2 \qquad T_1 <: Float \qquad T_2 <: Float}{\Gamma \vdash (t_1 <= t_2) : Bool}$$

- Logic:

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : Bool}{\Gamma \vdash (t_1 \textbf{ and } t_2) : Bool}$$

$$\frac{\Gamma \vdash t : Bool}{\Gamma \vdash \textbf{not } t : Bool}$$

- Conditionals:

$$\frac{\Gamma \vdash t_1 : Bool \qquad \Gamma \vdash t_2 : T_2 \qquad \Gamma \vdash t_3 : T_3 \qquad T = T_2 \sqcup T_3}{\Gamma \vdash (\textbf{if } t_1 \textbf{ then } t_2 \textbf{ else } t_3) : T}$$

- Variables:

$$\frac{v : T \in \Gamma}{\Gamma \vdash v : T}$$

- **let**:

$$\frac{\Gamma \vdash t_1 : T_1 \qquad \Gamma, v : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\textbf{let } v = t_1 \textbf{ in } t_2) : T_2}$$

- Function abstraction and application:

$$\frac{\Gamma, v : T_1 \vdash t_2 : T_2}{\Gamma \vdash (\textbf{lambda } v : T_1 . \, t_2) : T_1 \to T_2}$$

$$\frac{\Gamma \vdash t_1 : T_2 \to T_3 \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1 \, t_2) : T_3}$$

[5] We avoid unnecessary conversions in this rule, in light of the fact that many implementations use different representations for integer and floating-point values. Of course, such an implementation would still require a conversion when the operand types are different.

# Part IV

# Data Abstraction

We now examine mechanisms for constructing *abstract data types (ADTs)*, which allow us to abstract the interface for a piece of data from its implementation. We also look at mechanisms for *encapsulation*, which bind together the data of an ADT along with the functions that operate on that data.

CHAPTER

SEVENTEEN

# FUNCTIONAL DATA ABSTRACTION

We start by modeling data using the tools of procedural abstraction, beginning with a simple pair abstraction and progressing to more complex abstract data types that encode behavior with messages.

## 17.1 Pairs and Lists

Recall that in $\lambda$-calculus, a pair is implemented as a function that takes in two items and returns another function:

$$pair = \lambda x.\ \lambda y.\ \lambda f.\ f\ x\ y$$

We could then obtain the first item by applying the resulting function to $true$, and the second item by applying it to $false$:

$$first = \lambda p.\ p\ true$$
$$second = \lambda p.\ p\ false$$

Following a similar strategy, we can define a pair constructor in Python:

```
def pair(x, y):
    def get(i):
        return x if i == 0 else y

    return get
```

As in $\lambda$-calculus, the `pair()` function returns a function with the two items located in the latter's non-local environment. Now instead of applying the resulting function to a boolean, we call it on an index to retrieve the first or the second item:

```
def first(p):
    return p(0)

def second(p):
    return p(1)
```

```
>>> p = pair(3, 4)
>>> first(p)
3
>>> second(p)
4
```

Using pairs, we can build a full sequence abstraction, as in Scheme's pairs and lists. Before we proceed to do so, however, observe that our current pair implementation does not support mutation, which is a key feature of the sequence abstractions provided in imperative languages. We can implement mutation by defining separate get and set functions, using an immutable pair to return both when we construct a mutable pair:

```
def mutable_pair(x, y):
    def get(i):
        return x if i == 0 else y

    def set(i, value):
        nonlocal x, y
        if i == 0:
            x = value
        else:
            y = value

    return pair(get, set)

def mutable_first(p):
    return first(p)(0)

def mutable_second(p):
    return first(p)(1)

def set_first(p, value):
    second(p)(0, value)

def set_second(p, value):
    second(p)(1, value)
```

```
>>> p = mutable_pair(3, 4)
>>> mutable_first(p)
3
>>> mutable_second(p)
4
>>> set_first(p, 5)
>>> set_second(p, 6)
>>> mutable_first(p)
5
>>> mutable_second(p)
6
```

We use an immutable pair rather than a mutable one to return the get and set functions so as to avoid infinite recursion in `mutable_pair()`. In the definition of `set()`, the `nonlocal` statement is required so that the `x` and `y` in the non-local environment are modified.

While this representation works, it does not provide any encapsulation. We now have four functions that manipulate mutable pairs, and we had to name them carefully to avoid conflict with those that work with immutable pairs.

## 17.2 Message Passing

An alternative strategy, assuming that we have access to a string data type, is *message passing*, in which we send specific messages to an ADT that determine what operations are performed on the data. This can be implemented with a *dispatch function* that checks the input message against a known set of behaviors and then takes the appropriate action. Using message passing, we can define a mutable pair as follows:

```
def mutable_pair(x, y):
    def dispatch(message, value=None):
        nonlocal x, y
        if message == 'first':
            return x
        elif message == 'second':
            return y
        elif message == 'set_first':
            x = value
        elif message == 'set_second':
            y = value

    return dispatch
```

```
>>> p = mutable_pair(3, 4)
>>> p('first')
3
>>> p('second')
4
>>> p('set_first', 5)
>>> p('set_second', 6)
>>> p('first')
5
>>> p('second')
6
```

We still represent a pair as a function, but now instead of calling external functions on a pair, we pass it a message and, if appropriate, a value to obtain the action we want. The pair ADT is entirely encapsulated within the `mutable_pair()` function.

## 17.3 Lists

Now that we have mutable pairs, we can implement a mutable list as a sequence of pairs, as in Scheme. We will use the `None` object to represent an empty list:

```
def mutable_list():
    empty_list = None
    head = empty_list
    tail = empty_list

    def size(mlist):
        if mlist is empty_list:
            return 0
        return 1 + size(mlist('second'))
```

(continues on next page)

(continued from previous page)

```
    def getitem(mlist, i):
        if i == 0:
            return mlist('first')
        return getitem(mlist('second'), i - 1)

    def setitem(mlist, i, value):
        if i == 0:
            mlist('set_first', value)
        else:
            setitem(mlist('second'), i - 1, value)

    def to_string():
        if head is empty_list:
            return '[]'
        return ('[' + str(head('first')) +
                to_string_helper(head('second')) + ']')

    def to_string_helper(mlist):
        if mlist is empty_list:
            return ''
        return (', ' + str(mlist('first')) +
                to_string_helper(mlist('second')))

    def append(value):
        nonlocal head, tail
        if head is empty_list:
            head = mutable_pair(value, empty_list)
            tail = head
        else:
            tail('set_second', mutable_pair(value, empty_list))
            tail = tail('second')

    def dispatch(message, arg1=None, arg2=None):
        if message == 'len':
            return size(head)
        elif message == 'getitem':
            return getitem(head, arg1)
        elif message == 'setitem':
            return setitem(head, arg1, arg2)
        elif message == 'str':
            return to_string()
        elif message == 'append':
            return append(arg1)

    return dispatch
```

To avoid implementing all our functionality within the `dispatch()` function, we've defined separate functions to perform each action. Then the task of the `dispatch()` function is to call the appropriate function based on the input message. The following demonstrates how to use the mutable list ADT:

```
>>> l = mutable_list()
```

(continues on next page)

(continued from previous page)

```
>>> l('str')
'[]'
>>> l('len')
0
>>> l('append', 3)
>>> l('append', 4)
>>> l('append', 5)
>>> l('str')
'[3, 4, 5]'
>>> l('len')
3
>>> l('getitem', 1)
4
>>> l('setitem', 1, 6)
>>> l('str')
'[3, 6, 5]'
```

## 17.4 Dictionaries

We can implement a dictionary ADT using a list of records, each of which consists of a key-value pair.

```
def dictionary():
    records = mutable_list()

    def get_record(key):
        size = records('len')
        i = 0
        while i < size:
            record = records('getitem', i)
            if key == record('first'):
                return record
            i += 1
        return None

    def getitem(key):
        record = get_record(key)
        return record('second') if record is not None else None

    def setitem(key, value):
        record = get_record(key)
        if record is None:
            records('append', mutable_pair(key, value))
        else:
            record('set_second', value)

    def dispatch(message, key=None, value=None):
        if message == 'getitem':
            return getitem(key)
        elif message == 'setitem':
            setitem(key, value)
```

(continues on next page)

(continued from previous page)

```
    return dispatch
```

For simplicity, we only implement two messages, one for inserting a key-value pair into a dictionary and one for retrieving the value of a key. A key is looked up by searching through the records for a matching key, and if it is found, the associated value is returned. A key-value pair is inserted by looking up the key and modifying the associated value if it is found. If it is not found, then a new record is inserted.

```
>>> d = dictionary()
>>> d('setitem', 'a', 3)
>>> d('setitem', 'b', 4)
>>> d('getitem', 'a')
3
>>> d('getitem', 'b')
4
>>> d('setitem', 'a', 5)
>>> d('getitem', 'a')
5
```

Compare this to code that works with Python's built-in dictionaries, with special methods invoked directly rather than using operators:

```
>>> d = dict()
>>> d.__setitem__('a', 3)
>>> d.__setitem__('b', 4)
>>> d.__getitem__('a')
3
>>> d.__getitem__('b')
4
>>> d.__setitem__('a', 5)
>>> d.__getitem__('a')
5
```

The abstraction we provide is almost the same, with only minor differences in syntax. On the other hand, our dictionary implementation is particularly inefficient, requiring $\mathcal{O}(n^2)$ time to perform an operation on a dictionary with $n$ keys. We can reduce this to linear time by implementing an iterator abstraction on lists, but we will not do so here.

## 17.5 Dispatch Dictionaries

Now that we have dictionaries, we can make use of them to simplify our handling of messages. Previously, our dispatch function consisted of a lengthy conditional that called the appropriate internal function based on the message. In order to accommodate internal functions that take in different numbers of arguments, we had to arrange for the dispatch function to be able to take in the maximum number of arguments over the internal functions, and we had to use default arguments to enable fewer arguments to be passed. This can get unwieldy and error-prone the more complex our ADTs become.

Instead, we can store the mapping of messages to functions inside of a *dispatch dictionary*. When we pass a message to an ADT, it returns back the function corresponding to that message, which we can then call with the appropriate arguments. The following uses this pattern to define an ADT for a bank account:

```
def account(initial_balance):
    def deposit(amount):
        new_balance = dispatch('getitem', 'balance') + amount
        dispatch('setitem', 'balance', new_balance)
        return new_balance

    def withdraw(amount):
        balance = dispatch('getitem', 'balance')
        if amount > balance:
            return 'Insufficient funds'
        balance -= amount
        dispatch('setitem', 'balance', balance)
        return balance

    def get_balance():
        return dispatch('getitem', 'balance')

    dispatch = dictionary()
    dispatch('setitem', 'balance', initial_balance)
    dispatch('setitem', 'deposit', deposit)
    dispatch('setitem', 'withdraw', withdraw)
    dispatch('setitem', 'get_balance', get_balance)

    def dispatch_message(message):
        return dispatch('getitem', message)

    return dispatch_message
```

The dispatch dictionary contains an entry for the account balance, as well as functions to deposit, withdraw, and obtain the balance. The dispatch function just retrieves the appropriate function from the dispatch dictionary. We can then use an account as follows:

```
>>> a = account(33)
>>> a('get_balance')()
33
>>> a('deposit')(4)
37
>>> a('withdraw')(7)
30
>>> a('withdraw')(77)
'Insufficient funds'
```

Compare this to the interface provided by a bank account implemented as a Python class:

```
>>> a = Account(33)
>>> a.get_balance()
33
>>> a.deposit(4)
37
>>> a.withdraw(7)
30
>>> a.withdraw(77)
'Insufficient funds'
```

Once again, our implementation provides a very similar interface with only minor differences in syntax.

We have now constructed a hierarchy of ADTs using functions, progressing from immutable pairs to mutable pairs, lists, and dictionaries, finally arriving at a message-passing abstraction that bears striking resemblance to object-oriented programming. Next, we will examine language-level mechanisms for defining ADTs in the object-oriented paradigm.

CHAPTER

# EIGHTEEN

# OBJECT-ORIENTED PROGRAMMING

Object-oriented languages provide mechanisms for defining abstract data types in a systematic manner. Such languages provide means for the following features of ADTs:

- *Encapsulation*: The ability to bundle together the data of an ADT along with the functions that operate on that data[1].
- *Information hiding*: The ability to restrict access to implementation details of an ADT.
- *Inheritance*: The ability to reuse code of an existing ADT when defining another ADT. This includes *implementation inheritance*, where the actual implementation of an ADT is reused, and *interface inheritance*, where the new ADT merely supports the same interface as the existing ADT.
- *Subtype polymorphism*: The ability to use an instance of a derived ADT where a base ADT is expected. This requires some form of *dynamic binding* or *dynamic dispatch*, where the derived ADT's functionality is used at runtime when the base ADT's version is expected at compile time.

In object-oriented languages, an ADT is specified by a *class*, which defines the pattern to be used in instantiating *objects* of the class.

## 18.1 Members

An object is composed of individual pieces of data, variously called *fields*, *attributes*, or *data members*. Functions that are defined within a class and operate on the contents of an object are often called *methods*, or in C++ terminology, *member functions*.

```
class Foo {
public:
  int x;
  Foo(int x_);
  int baz(int y);
};
```

In the example above, `x` is a field, `Foo()` is a *constructor* that is called to initialize a new object of type `Foo`, and `baz()` is a member function.

A class may also have fields associated with it that are shared among all instances of the class. These are often called *static fields* or *class attributes*, and they are often specified with the `static` keyword, as in the following Java code:

```
class Foo {
  static int bar = 3;
}
```

[1] The term "encapsulation" is often used to encompass information hiding as well.

A static field usually can be accessed through the class or through an instance:

```
System.out.println(Foo.bar); // access through class
System.out.println(new Foo().bar); // access through instance
```

The following is the same example in C++:

```
class Foo {
public:
  static int bar;
};

int Foo::bar = 3;

int main() {
  cout << Foo::bar << endl;
  cout << Foo().bar << endl;
}
```

C++ requires an out-of-line definition for static data members that are not compile-time constants to designate a storage location. Class members are accessed using the scope-resolution operator (`::`).

Finally, the following demonstrates this example in Python:

```
class Foo:
   bar = 3

print(Foo.bar)
print(Foo().bar)
```

Attributes that are defined directly within a class definition are automatically class attributes.

## 18.2 Access Control

Information hiding requires the ability to restrict access to the members of a class or object. Many object-oriented languages provide a mechanism for restricting accessibility (also called *visibility*) of members. Common categories of access include:

- allowing only an object itself to access its own data
- allowing all code in a class to access any data of the class or its instances
- allowing the data inherited from a base class to be accessed by code in a derived class
- allowing the data of a class and its instances to be accessed by all code in the same package or module
- allowing all code in a program to access the data of a class and its instances

In C++, Java, and C#, the `public` keyword grants all code access to a member, while the `private` keyword restricts access to the class itself. In C++ and C#, the `protected` keyword grants access to inherited data to derived classes, while in Java, it additionally grants access to all code in the same package. In C#, the `internal` keyword grants access to a package. In Java, a member that does not have an access qualifier is accessible to other code in the same package but not to derived classes in other packages.

In many dynamic languages, such as Smalltalk and Python, all members have public accessibility. In Ruby, fields of an object are only accessible to the object itself and not to other objects of the same class.

Table 18.1 summarizes the access control provided by several languages.

Table 18.1: Access control in different languages

| Access | public | private | C++ protected | Java protected | C# internal/Java default | Python |
|---|---|---|---|---|---|---|
| Same instance | X | X | X | X | X | X |
| Same class | X | X | X | X | X | X |
| Derived classes | X | | X | X | | X |
| Same package | X | | | X | X | X |
| Global access | X | | | | | X |

A subtlety arises when it comes to the `protected` access level. Suppose a class `Derived` derives from `Base`, and `Base` defines a `protected` member `x`. Is `Derived` allowed to access the `x` member of instances of `Base` that are not also instances of `Derived`? The following C++ code demonstrates this case:

```
class Base {
protected:
  int x = 4;
};

class Derived : public Base {
public:
  void foo(Base *b, Derived *d) {
    cout << b->x << endl; // ERROR
    cout << d->x << endl; // OK
  }
};
```

C++, C#, and Java all prohibit `Derived` from accessing the protected member `x` of `Base`, unless the access is through an instance that is also of type `Derived`. Thus, the expression `b->x` above is erroneous, while `d->x` is permitted.

## 18.3 Kinds of Methods

Methods that operate on instances of a class generally take in the instance itself as a parameter. Often, this parameter is named `self` or `this`, either by convention or as a language keyword. In most languages, the instance is an implicit parameter, as in the following C++ code:

```
class Foo {
public:
  int x;

  int get_x() {
    return this->x;
  }
};
```

In many languages, the `this` qualification on a member can be elided, though it is necessary if another variable hides the declaration of the member:

```
class Bar {
public:
  int x, y;
```

(continues on next page)

(continued from previous page)

```
  void baz(int x) {
    cout << this->x << endl;  // x hidden by parameter
    cout << y << endl;        // y not hidden, so this-> not needed
  }
};
```

In Python, the instance must be an explicit parameter, conventionally named `self`. The `self` qualification cannot be elided:

```
class Foo:
    def __init__(self, x):
        self.x = x

    def get_x(self):
        return self.x
```

In most languages, method-call syntax implicitly passes the instance as the implicit or explicit instance parameter, as the instance is syntactically provided as part of the method call:

```
f = Foo(3)
f.get_x()  # passes f as self parameter to get_x()
```

Most languages also provide a means for defining static methods, which do not operate on an instance but can generally be called on a class or instance. In languages in the C++ family, the `static` keyword specifies a static method. In Python, the `@staticmethod` decorator accomplishes this:

```
class Baz:
    @staticmethod
    def name():
        return 'Baz'

print(Baz.name())
print(Baz().name())
```

Without the `@staticmethod` decorator, the function `name()` cannot be called on an instance of `Baz`. Python also has a `@classmethod` decorator that allows definition of a static-like method that takes in the class itself as the first argument:

```
class Baz:
    @classmethod
    def name(cls):
        return cls.__name__

class Fie(Baz):
    pass

print(Baz.name())     # prints Baz
print(Baz().name())   # prints Baz
print(Fie.name())     # prints Fie
print(Fie().name())   # prints Fie
```

Some languages, such as C# and Python, provide a mechanism for defining *property* methods that act as accessors to fields. Such a method is called using field-access syntax and is useful for controlling access to a field. A property

method can also be used to provide a field interface for data that must be computed on the fly, such as in the following complex-number representation:

```
import math

class Complex(object):
    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5

    @magnitude.setter
    def magnitude(self, mag):
        old_angle = self.angle
        self.real = mag * math.cos(old_angle)
        self.imag = mag * math.sin(old_angle)

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)

    @angle.setter
    def angle(self, ang):
        old_magnitude = self.magnitude
        self.real = old_magnitude * math.cos(ang)
        self.imag = old_magnitude * math.sin(ang)
```

The `@property` decorator defines a getter, followed by which the `@<method>.setter` decorator can be used to define a setter, where `<method>` is the name of the function used with `@property`. With `magnitude` and `angle` defined as properties with both getters and setters, we can use them as follows:

```
>>> c = Complex(1, math.sqrt(3))
>>> c.magnitude
2.0
>>> c.angle / math.pi
0.3333333333333333
>>> c.magnitude = math.sqrt(2)
>>> c.angle = math.pi / 4
>>> c.real
1.0000000000000002
>>> c.imag
1.0
```

Thus, property methods allow the interface of a field to be abstracted from its implementation. In the example of `Complex`, we could change the implementation such that `magnitude` and `angle` are stored as standard fields and `real` and `imag` are implemented as property methods. This would not change the interface of `Complex` at all, abstracting the implementation change from outside code.

## 18.4 Nested and Local Classes

Some object-oriented languages allow a *nested class* to be defined within the scope of another class. This enables a helper class to be encapsulated within the scope of an outer class, enabling it to be hidden from users of the outer class. A language may also allow a class to be defined at local scope as well.

Languages in which classes are first-class entities allow the creation of new classes at runtime. Generally, such a creation may happen at any scope, and the class has access to its definition environment (i.e. it has *static scope*). Python is an example of such a language.

In C++, nested and local classes act as any other classes, except that they have access to the private members of the enclosing class. On the other hand, the enclosing class must be declared as a `friend` of a nested class in order to have access to the private members of the nested class. A local class does not have access to the local variables in the enclosing stack frame.

Java provides more flexibility in its nested and local classes. Local classes have access to local variables that are *effectively final*, meaning that they are only assigned once and never modified. When defined in a non-static scope, both nested and local classes are associated with an actual instance of the enclosing class and have direct access to its fields:

```
class Outer {
  private int x;

  Outer(int x) {
    this.x = x;
  }

  class Inner {
    private int y;

    Inner(int y) {
      this.y = y;
    }

    int get() {
      return x + y;
    }
  }
}

class Main {
  public static void main(String[] args) {
    Outer out = new Outer(3);
    Outer.Inner inn = out.new Inner(4);
    System.out.println(inn.get());
  }
}
```

In Java, nested and local classes have access to private members of the enclosing class, and the enclosing class has access to the private members of a nested class. The definition of a nested class can be prefaced with the `static` keyword to dissociate it from any instance of the enclosing class.

## 18.5 Implementation Strategies

In concept, object-oriented programming is built around the idea of passing messages to objects, which then respond in a manner appropriate for the object. Access to a member can be thought of as sending a message to the object. Languages differ in whether or not the set of messages an object responds to is fixed at compile time, as well as whether the actual message that is passed to an object is fixed at compile time.

In efficiency-oriented languages such as C++ and Java, the set of messages that an object supports is fixed at compile time and is the same for all instances of a class. Such a language enables objects to be implemented in a manner similar to *records* or structs: the fields of an object can be stored contiguously within the memory for the object, with one slot for each field. Access to a field can then be translated at compile time to a fixed offset into the object, similar to an *offset-based implementation of activation records* . As an example, consider the following class in C++:

```
class Foo {
public:
  int x, y;
  Foo(int x_, int y_);
};
```

The fields `x` and `y` are stored contiguously within the `Foo` object, with `x` at an offset of zero bytes from the beginning of the `Foo` object and `y` at an offset of four bytes, since `x` takes up four bytes (assuming that `sizeof(int) == 4`). Figure 18.1 illustrates this layout:

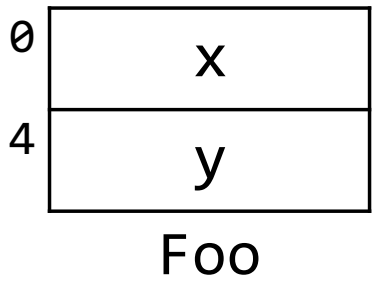



Figure 18.1: Record-based implementation of an object.

Then given a `Foo` object `f`, the field access `f.x` is translated at compile time to an offset of zero from the address of `f`, while `f.y` is translated to an offset of four. No lookup is required at runtime, making such an implementation very efficient.

In languages that enable a member to be added to a class or even an individual object at runtime, members are usually stored within a dictionary, analogous to a *dictionary-based implementation of activation records*. This is similar to the message-passing scheme demonstrated in the last section. Such a language defines a process for looking up a member. For example, in Python, accessing an attribute of an object first checks the dictionary for the object before proceeding to the dictionary for its class:

```
class Foo:
    y = 2

    def __init__(self, x):
        self.x = x

f = Foo(3)
```

(continues on next page)

(continued from previous page)

```
print(f.x, f.y, Foo.y)   # prints 3 2 2
f.y = 4                  # adds binding to instance dictionary
print(f.x, f.y, Foo.y)   # prints 3 4 2
```

The class `Foo` has a class attribute `y`, and the constructor creates an instance attribute `x`. Looking up `f.x` first looks in the instance dictionary, finding a binding there. On the other hand, looking up `f.y` within the first call to `print()` does not find `y` in the instance dictionary, so lookup proceeds to the class dictionary, finding it there. The assignment `f.y = 4` introduces a binding for `y` in the instance dictionary, so subsequent lookups find `y` there.

Python actually takes a hybrid approach, using a dictionary by default but allowing a class to specify a record-like implementation using the special `__slots__` attribute. The following is a definition of the `Complex` class to use this mechanism:

```
import math

class Complex(object):
    __slots__ = ('real', 'imag')

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    @property
    def magnitude(self):
        return (self.real ** 2 + self.imag ** 2) ** 0.5

    @magnitude.setter
    def magnitude(self, mag):
        old_angle = self.angle
        self.real = mag * math.cos(old_angle)
        self.imag = mag * math.sin(old_angle)

    @property
    def angle(self):
        return math.atan2(self.imag, self.real)

    @angle.setter
    def angle(self, ang):
        old_magnitude = self.magnitude
        self.real = old_magnitude * math.cos(ang)
        self.imag = old_magnitude * math.sin(ang)
```

Instances of a class that uses `__slots__` no longer store attributes in a dictionary, saving space and providing better performance. However, they lose the ability of adding attributes to a specific instance at runtime.

Dictionary-based languages usually provide a mechanism for dynamically constructing a message and passing it to an object, such as the special `__getattribute__` method of Python objects:

```
>>> x = [1, 2, 3]
>>> x.__getattribute__('append')(4)
>>> x
[1, 2, 3, 4]
```

Java also supports dynamic invocation of messages through a powerful *reflection* API, which provides a form of *runtime*

*type information*:

```
import java.lang.reflect.Method;

class Main {
  public static void main(String[] args) throws Exception {
    String s = "Hello World";
    Method m = String.class.getMethod("length", null);
    System.out.println(m.invoke(s));
  }
}
```

CHAPTER

NINETEEN

# INHERITANCE AND POLYMORPHISM

Inheritance and polymorphism are two key features of object-oriented programming, enabling code reuse as well as allowing the specialization of behavior based on the dynamic type of an object. Languages differ greatly in the design choices they make in the specifics of how they support inheritance and polymorphism. In this section, we discuss some of these design choices as well as how they are typically implemented.

## 19.1 Types of Inheritance

In *Object-Oriented Programming*, we alluded to the fact that interface inheritance only reuses the interface of an ADT, while implementation inheritance reuses the implementation. These two types of inheritance are strongly coupled in most languages; specifically, implementation inheritance almost always includes interface inheritance as well. C++ is an exception, allowing fields and methods to be inherited without exposing them as part of the interface of the derived class.

In particular, C++ supports *private*, *protected*, and *public* inheritance, which designate the accessibility of inherited members. In private inheritance, all inherited members are made private in the derived class. In protected inheritance, inherited members that were originally public are made protected, while more restricted members retain their original accessibility. In public inheritance, all inherited members retain their original accessibility. The general rule is that the accessibility of an inherited members is the more restrictive of its original accessibility and the type of inheritance. In keeping with the meaning of `private` discussed previously, inherited members that were originally private are not accessible to the derived class itself.

The default inheritance variant is public for classes defined using the `struct` keyword, while it is private if the `class` keyword is used. The programmer can override the default by placing an access modifier in front of the base class, as in the following:

```
class A {
public:
  void foo();

protected:
  void bar();

private:
  void baz();
};

class B : public A {
};

class C : protected A {
```

(continues on next page)

(continued from previous page)

```
};

class D : A {
};
```

In this example, the method `foo()` is public in `B`, protected in `C`, and private in `D`. Thus, `D` inherits the implementation of `foo()` without exposing it as part of its interface. The method `bar()` is protected in `B` and `C` and private in `D`. Finally, the member function `baz()` is private in all three derived classes, while also being inaccessible to the classes themselves.

C++ also allows derived classes to delete *non-virtual* inherited methods.

Some languages allow an interface to be inherited without the implementation, requiring concrete derived classes to provide their own implementation. A method is *abstract* (*pure virtual* in C++ terminology) if no implementation is provided, and a class is abstract if it has at least one abstract method, whether the abstract method is declared directly in the class or inherited. In Java, abstract classes must be labeled as such:

```
abstract class A {
  abstract void foo();
}
```

A class that only has abstract methods is often called an *interface*, and Java has specific mechanisms for defining and implementing an interface:

```
interface I {
  void bar();
}

class C extends A implements I {
  void foo() {
    System.out.println("foo() in C");
  }

  public void bar() {
    System.out.println("bar() in C");
  }
}
```

Abstract methods in Java may have any access level except private, but interface methods are implicitly public. Java allows a class to implement multiple interfaces, though it only allows it to derive from a single class.

Some languages further decouple inheritance from polymorphism by allowing methods to be inherited without establishing a parent-child relationship between two classes. The class that defines these methods is called a *mixin*, and a mixin can be included from another class to obtain those methods. The use of mixins is particularly common in Ruby. The following is an example:

```
class Counter
  include Comparable
  attr_accessor :count

  def initialize()
    @count = 0
  end
```

(continues on next page)

(continued from previous page)

```
  def increment()
    @count += 1
  end

  def <=>(other)
    @count <=> other.count
  end
end

c1 = Counter.new()
c2 = Counter.new()
c1.increment()
print c1 == c2
print c1 < c2
print c1 > c2
```

By including the `Comparable` mixin, the `Counter` class obtains comparison methods such as < and <= that use the general <=> comparison method defined in `Counter`.

We will see later how to implement mixins in C++ using the *curiously recurring template pattern*.

## 19.2 Class Hierarchies

In some languages, such as Java and Python, every class eventually derives from a root class, called `Object` in Java and `object` in Python. This results in a single class hierarchy rooted at the root class. In Java, this hierarchy is a tree, since Java does not allow multiple inheritance outside of interfaces. Python does allow multiple inheritance, so the hierarchy is a directed acyclic graph. Other languages, including C++, do not have a root class.

A root class enables code to be written that works on all class-type objects. For example, a `Vector<Object>` in Java can hold objects of any type. Because the `Object` class defines an `equals()` method, such a data structure can be searched to find an object that is semantically equal to an item:

```
Vector<Object> unique(Vector<Object> items) {
  Vector<Object> result = new Vector<Object>();
  for (Object item : items) {
    if (!result.contains(item)) {
      result.add(item);
    }
  }
  return result;
}
```

In this example, the `contains()` method of `Vector<Object>` calls the `equals()` method on an element. Since the root `Object` class defines `equals()`, it is valid to call on an instance of any class.

In contrast, C++ allows `void *` to hold a pointer to any object, so that a `vector<void *>` can store pointers to arbitrary objects. However, a `void *` does not implement any behavior, so we can only compare such pointers by pointer value and not whether the actual referenced objects are equal.

## 19.3 Method Overriding

The ability to *override* a method in a derived class is the key to polymorphism in object-oriented programming. Overriding requires *dynamic binding*, where the actual method to be invoked is determined by an object's dynamic type rather than the type apparent in the program source.

As we will see shortly, dynamic binding comes at a runtime cost. To avoid this cost wherever possible, instance methods do not use dynamic binding by default in C++. Instead, an instance method must be designated as *virtual* in order for dynamic binding to be used. Java, on the other hand, uses dynamic binding for all instance methods, except those designated as *final* in some cases, since they cannot be overridden. Both languages use static binding for static methods, whether or not they are dispatched through an object.

*Dynamically typed* languages universally support dynamic binding, since objects do not have a static type. Such languages include Python and Ruby.

In languages that support method overloading, including C++ and Java, a method generally must have the same parameter list as the method it is overriding. Otherwise, the new definition is treated as overloading or hiding the base-class method instead. This can lead to unexpected behavior, such as the following code in Java:

```
class Foo {
  int x;

  Foo(int x) {
    this.x = x;
  }

  public boolean equals(Foo other) {
    return x == other.x;
  }
}

Vector<Foo> vec = new Vector<Foo>();
vec.add(new Foo(3));
System.out.println(vec.contains(new Foo(3)));
```

This code, when run, prints out `false`. The problem is that the `equals()` method defined in `Object` has the signature:

```
public boolean equals(Object other)
```

The difference in the parameter type causes the `equals()` that is defined in `Foo` to be an overload rather than overriding the inherited method. Combined with the fact that *generics* in Java do not generate code that is specialized to the type parameter, this results in the original `equals()` method being called from the `contains()` method in `Vector`.

Java allows a method to be annotated to assert that it is an override, as follows:

```
@Override
public boolean equals(Foo other) {
  return x == other.x;
}
```

The compiler will then detect that the method does not in fact override a base-class method and will report an error. C++11 has a similar `override` keyword that can be placed at the end of a method signature:

```
virtual void foo(Bar b) override;
```

### 19.3.1 Covariance and Contravariance

Some statically typed languages, including C++ and Java, permit *covariant return types*, where the return type of an overriding method is a derived type of the return type in the overridden method. Such a narrowing is semantically valid, since a derived object can be used (at least as far as the type system is concerned) where a base type is expected. The `clone()` method in Java is an example, where the version in `Object` returns `Object`:

```
class Foo {
  int x;

  @Override
  public Foo clone() {
    Foo f = new Foo();
    f.x = x;
    return f;
  }
}
```

Equally valid semantically for parameter types is *contravariance*, where an overriding method takes in a base type of the parameter type in the overridden method. However, in languages that allow overloading, parameter contravariance results in an ambiguity: is the newly defined method an override of the original method, an overload of the method, or does it hide the base-class method? Consider the following example in Java:

```
class Foo {
  int foo(Foo other) {
    return 0;
  }
}

class Bar extends Foo {
  int foo(Object other) {
    return 1;
  }
}
```

The call to `b.foo(arg)`, where `b` is of type `Bar`, results in different behavior depending on the type of `arg`:

```
Bar b = new Bar();
System.out.println(b.foo(new Bar()));     // prints 0
System.out.println(b.foo(new Object()));  // prints 1
```

Thus, in Java, defining a method with a parameter that is contravariant to the base-class method results in an overload. On the other hand, in C++, this pattern hides the base-class method:

```
class Base {
};

class Foo : public Base {
public:
  int foo(const Foo &other) const {
    return 0;
  }
};
```

(continues on next page)

(continued from previous page)

```
class Bar : public Foo {
public:
  int foo(const Base &other) const {
    return 1;
  }
};

int main() {
  Bar b;
  cout << b.foo(Bar()) << endl;   // prints 1
  cout << b.foo(Base()) << endl;  // prints 1
}
```

In both languages, the derived-class method with contravariant parameters does not override the base-class method.

### 19.3.2 Accessing Hidden or Overridden Members

In many languages, base-class members that are not overridden but redefined in a derived class are *hidden* by the definition in the derived class. This is the case for non-virtual methods in C++, as well as virtual methods that differ in signature from the method defined in the base class. In Java, on the other hand, a derived-class method with the same name as a base-class method but a different signature overloads the base-class method rather than override or hide it, as we saw in *Method Overriding*.

In record-based languages, redefining a field in a derived class usually results in the derived object containing both the hidden and the redefined field. In dictionary-based languages, however, objects usually only have a single field for a given name. Using `__slots__` in Python, space is reserved for both the hidden and the redefined field, but field access always accesses the slot defined in the derived class.

A common pattern in a derived-class method is to add functionality to that of the base-class method that it is overriding or hiding. In order to avoid repeating code, most languages provide a means of calling the base-class method. In C++, the scope-resolution operator enables this:

```
struct A {
  void foo() {
    cout << "A::foo()" << endl;
  }
};

struct B : A {
  void foo() {
    A::foo();
    cout << "B::foo()" << endl;
  }
};
```

More common is some variation of `super`, as in the following in Java:

```
class A {
  void foo() {
    System.out.println("A.foo()");
  }
}
```

(continues on next page)

(continued from previous page)

```
class B extends A {
  void foo() {
    super.foo();
    System.out.println("B.foo()");
  }
}
```

Python uses similar syntax:

```
class A:
    def foo(self):
        print('A.foo()')

class B(A):
    def foo(self):
        super().foo()
        print('B.foo()')
```

The same mechanisms can be used to access a hidden field, i.e. the scope-resolution operator in C++ and `super` in Java. In Python, `super()` can be used to access hidden static fields; instance fields are not replicated within an object.

Perhaps the most common case where a base class member needs to be accessed is the constructor for the derived class, where the base-class constructor needs to be invoked. In C++, a base-class constructor can be explicitly invoked from a constructor's initializer list:

```
struct A {
  A(int x);
};

struct B : A {
  B(int x) : A(x) {}
};
```

If no explicit call is made to a base-class constructor, a call to the default constructor of the base class is inserted by the compiler, and it is an error if such a constructor does not exist. The base-class constructor runs before any other initializers or the body of the derived-class constructor, regardless of where the former appears in the latter's initializer list.

In Java, a call to a base-class constructor must be the first statement in a constructor, and the compiler implicitly inserts a call to the zero-argument base-class constructor if an explicit call is not provided.

```
class A {
  A(int x) {
  }
}

class B extends A {
  B(int x) {
    super(x);
  }
}
```

In Python, a call to a base-class constructor must be made explicitly, and the interpreter does not insert one if it is

missing.

```
class A:
    def __init__(self, x):
        pass

class B(A):
    def __init__(self, x):
        super().__init__(x)
```

## 19.4 Implementing Dynamic Binding

In dictionary-based languages such as Python, dynamic binding is straightforward to implement with a sequence of dictionary lookups at runtime. In particular, when accessing an attribute of an object in Python, Python first searches the dictionary for the object itself. If it is not found, then it searches the dictionary for the object's class. If the attribute is still not found, it proceeds to the base-class dictionaries.

In record-based languages, however, efficiency is a primary concern, and dynamic name lookup can be prohibitively expensive. Instead, such languages commonly store pointers to methods that need to be looked up dynamically in a structure called a *virtual table*, or *vtable* for short. This name is a reflection of the term "virtual" in C++, which denotes methods that are dynamically bound.

As an example, consider the following C++ code:

```
struct A {
  int x;
  double y;
  virtual void a();
  virtual int b(int i);
  virtual void c(double d);
  void f();
};

struct B : A {
  int z;
  char w;
  virtual void d();
  virtual double e();
  virtual int b(int i);
  void f();
};
```

The storage for an object of type `A` contains as its first item a pointer to the vtable for class `A`, which is then followed by entries for fields `x` and `y`. The vtable for `A` contains pointers to each of its virtual methods in order, as shown in Figure 19.1.

Neither the storage for an object of type `A` nor the vtable for `A` contains a pointer to `A::f`: the latter is not a virtual method and so is not dynamically bound. Instead, the compiler can generate a direct dispatch to `A::f` when the method is called on an object whose static type is `A`.

The storage for an object of type `B` also contains a vtable pointer as its first item. This is then followed by inherited fields, after which are slots for fields introduced by `B`. The vtable for `B` contains pointers for each of its methods. First come methods inherited from `A` or overridden, in the same order as in the vtable for `A`. Then the new methods introduced by `B` follow, as illustrated in Figure 19.2.

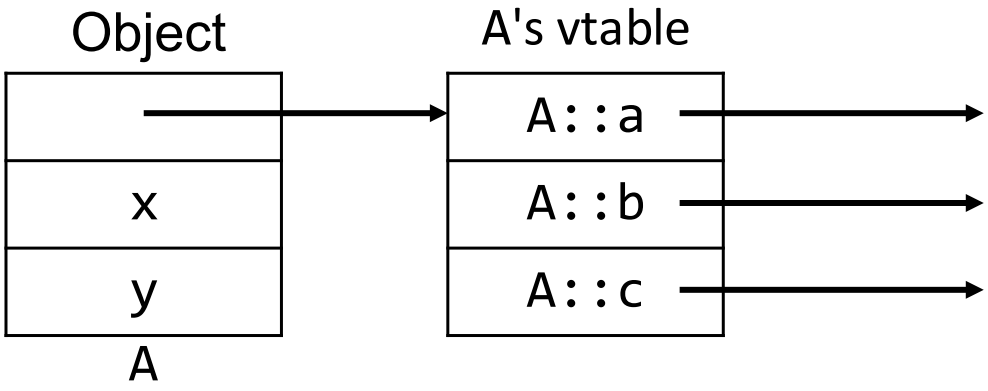



Figure 19.1: A record-based implementation of an object with dynamically bound methods stores a vtable pointer at the beginning of the object. The vtable stores pointers to each dynamically bound method.

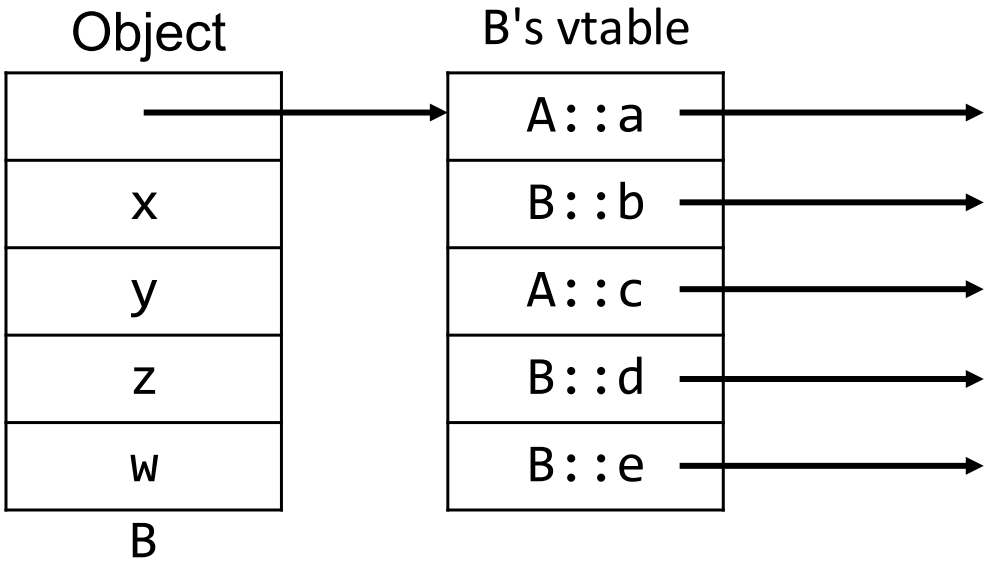



Figure 19.2: The layout of a derived-class object consists of a vtable pointer, then inherited fields, followed by fields introduced by the derived class. The vtable for the derived class begins with the same layout as that of the base class, followed by new methods introduced by the derived class.

As mentioned previously, fields are statically bound, but fields that are inherited from `A` are at the same offsets in both `A` and `B`. Thus, the compiler can translate a field access to an offset into an object, and the same offset will work for a base class and its derived classes. We can observe this by computing the offset of the member `x` in an `A` and a `B` from the beginning of the object:

```
A a;
B b;
cout << (((uintptr_t) &a.x) - (uintptr_t) &a) << endl;
cout << (((uintptr_t) &b.x) - (uintptr_t) &b) << endl;
```

Converting a pointer to a `uintptr_t` results in its address value. Running the above code results in the same offset of 8 using Clang on a 64-bit Intel machine, reflecting the size of the vtable pointer that comes before `x`.

Dynamically bound methods, on the other hand, require an indirection. A method override has the same offset in a derived class's vtable as the overridden method in the base class's vtable. In the example above, `B::b` is located in the second entry in the vtable for `B`, which is the same offset as where `A::b` is stored in the vtable for `A`. Thus, the compiler can translate a dynamic method call to a dereference into the object to get to its vtable, a fixed offset into the vtable, followed by another dereference to get to the actual code pointer. As an example, consider the following:

```
A *aptr = new B;
aptr->b();
```

The following pseudocode demonstrates the process of calling `b()`:

```
// extract vtable pointer from start of object
vtable_ptr = aptr-><vtable>;
// index into vtable at statically computed offset for b
func_ptr = vtable_ptr[1];
// call function, passing the implicit this parameter
func_ptr(aptr);
```

This process requires two dereferences to obtain the code location of the dynamically bound method, one to extract the vtable pointer from the object and another to index into the vtable. In contrast, the code location for a statically bound method call can be determined at compile time, which is more efficient than the two runtime dereferences required in dynamic binding.

### 19.4.1 Full Lookup and Dispatch Process

In general, the *receiver* of a method call in a statically typed language can have a dynamic type that differs from its static type. For example, in the code below, the receivers of the first two method calls have static type `A` while their dynamic type is `B`:

```
A *aptr = new B;
A &aref = *aptr;
B *bptr = new B;
aptr->b();    // receiver has static type A, dynamic type B
aref.f();     // receiver has static type A, dynamic type B
bptr->b();    // receiver has static type B, dynamic type B
```

The following is the general pattern that statically typed languages use to look up the target method and generate a dispatch to the appropriate code:

1. Look up the member (e.g. `b` in the case of `aptr->b()`) in the *static* type of the receiver, performing function-overload resolution if necessary to determine which method is being called.

2. If the resolved method is non-virtual, then generate a direct dispatch to the code for that method. For example, in the call `aref.f()` above, a direct dispatch to `A::f` would be generated since `A::f`, the result of the lookup, is non-virtual.

3. If the resolved method is virtual, then determine its offset in the vtable of the static type. In the case of `aptr->b()`, the resolved method is `A::b`, which is the second entry in the vtable for `A`. Then an indirect dispatch is generated, as described previously:

```
vtable_ptr = aptr-><vtable>;
func_ptr = vtable_ptr[1];
func_ptr(aptr);
```

## 19.5 Multiple Inheritance

Some languages allow a class to directly inherit from multiple base classes. This includes the limited form enabled by Java's interfaces, as well as the fully general multiple inheritance provided by Python and C++. Multiple inheritance raises several semantic and implementation issues that do not occur in single inheritance.

### 19.5.1 Dictionary-Based Implementation

In Python, where instance fields are stored in an object's dictionary by default, there is no concept of inheriting instance fields from a base class. Thus, in the absence of `__slots__`, multiple inheritance poses no problems for looking up an instance field. On the other hand, methods are generally stored in the dictionary for a class, along with static fields. Thus, a key question raised by multiple inheritance is in which order to search base-class dictionaries if an attribute is not found in the dictionary for an object or its class. The solution is non-trivial, as can be seen in the example below:

```
class Animal:
    def defend(self):
        print('run away!')

class Insect(Animal):
    pass

class WingedAnimal(Animal):
    def defend(self):
        print('fly away!')

class Butterfly(Insect, WingedAnimal):
    pass
```

If `defend()` is called on a `Butterfly`, there are several orders in which the method can be looked up among its base classes. A naive depth-first search would result in `Animal.defend`, but `WingedAnimal.defend` is in a sense "more derived" than `Animal.defend` and should be preferred in most cases. The actual algorithm used by Python is C3 linearization, which results in an order that preserves certain important aspects of the inheritance hierarchy. The details are beyond the scope of this text, but the result is that `WingedAnimal.defend` is used:

```
>>> Butterfly().defend()
fly away!
```

### 19.5.2 Record-Based Implementation

In a record-based implementation, multiple inheritance makes it impossible to ensure that a field is stored at a consistent offset from the beginning of an object. Consider the following C++ code:

```
struct A {
  int x;
  virtual void a();
  virtual void b();
};

struct B {
  int y;
  virtual void c();
  virtual void d();
};

struct C : A, B {
  int z;
  virtual void a();
  virtual void c();
  virtual void e();
};
```

In objects of type `A`, the field `x` is stored in the first entry after the vtable pointer. Similarly, `y` in `B` is stored in the first entry. With `C` deriving from both `A` and `B`, only one of those fields can be stored in the first entry for `C`. A similar problem occurs for method entries in a vtable.

Python classes that define `__slots__` suffer the same problem, as in the following:

```
class A:
    __slots__ = 'x'

class B:
    __slots__ = 'y'

class C(A, B):
    pass
```

Python's solution to this conflict is to make it illegal for a class to derive from multiple base classes that define `__slots__`.

C++, on the other hand, does permit code like the above. The solution that C++ uses is to combine different views of an object that has multiple base classes within the storage for the object. In the example above, we would have one view of the object from the perspective of `C` and `A`, and a separate view from the perspective of `B`, each with its own vtable. Figure 19.3 illustrates the two views.

Now the view used depends on the type of pointer or reference that refers to a `C` object:

```
C *c_ptr = new C();    // uses view A, C
A *a_ptr = c_ptr;      // uses view A, C
B *b_ptr = c_ptr;      // uses view B
```

When a pointer of type `C *` is converted to one of type `B *`, the compiler automatically adjusts the pointer to use the view for `B`. Then the offset for `y` from that view is the same as that of an object of type `B`. Similarly, the methods that

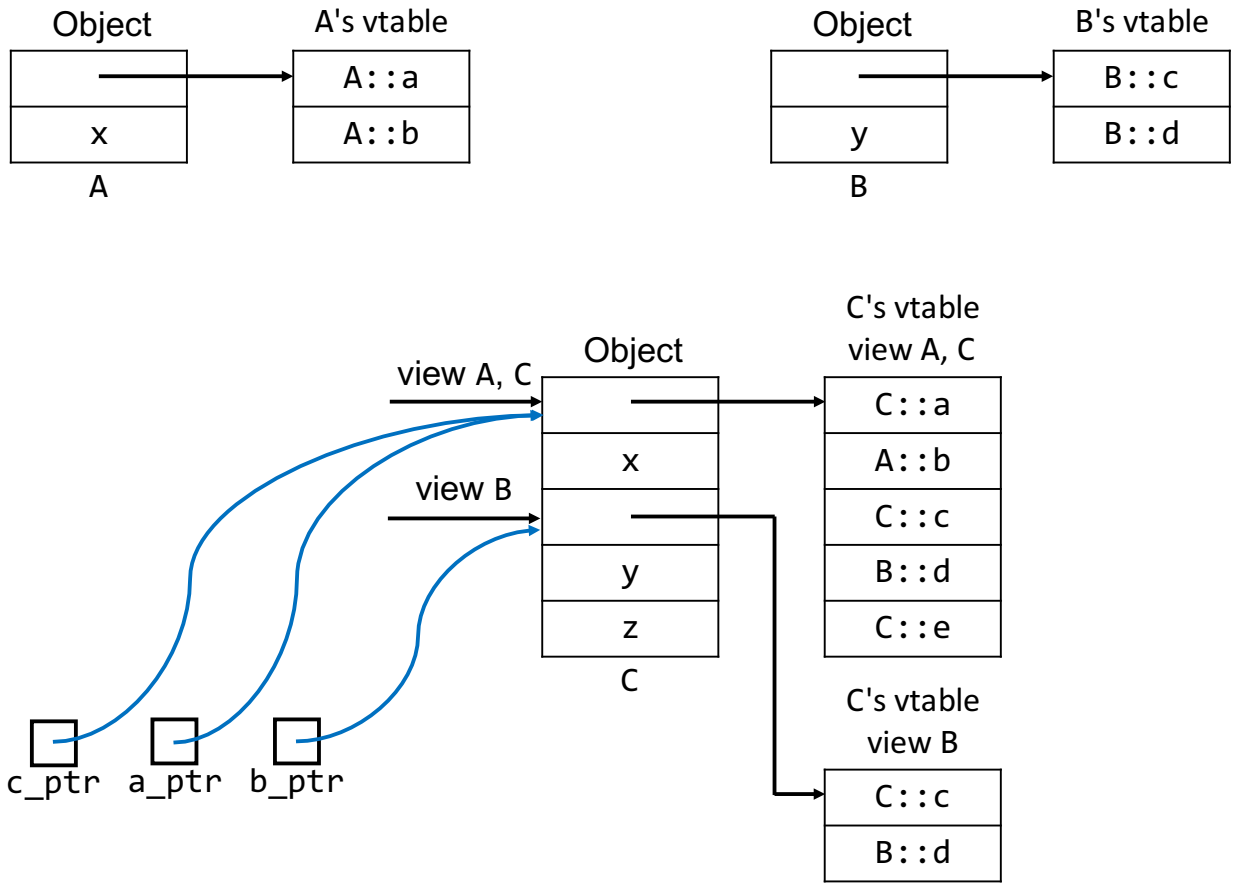



Figure 19.3: Multiple inheritance in a record-based implementation results in multiple views of an object, each with its own vtable.

are inherited from B or overridden are located at the same vtable offsets in the vtable for view B as in the vtable for an object of type B itself. The same properties hold for the A view and objects of actual type A.

The problem is not yet completely solved, however. What happens when we invoke an overridden method through the B view? Specifically, consider the following:

```
void C::c() {
  cout << z;
}

C *c_ptr = new C();
B *b_ptr = c_ptr;
c_ptr->c();
b_ptr->c();
```

If the code generated for C::c() assumes an offset for z based on the C view, then that same offset is not valid for the B view. In particular, z is two vtable pointers and two ints away from the beginning of the C view, but it is one vtable pointer and one int away in the B view. We need to arrange for the view of the object to be the C view in the body of C::c(), even when the method is invoked through a B pointer. One way to do this is to store offsets in vtable entries that designate how to change the pointer when the given method is invoked, as in Figure 19.4.

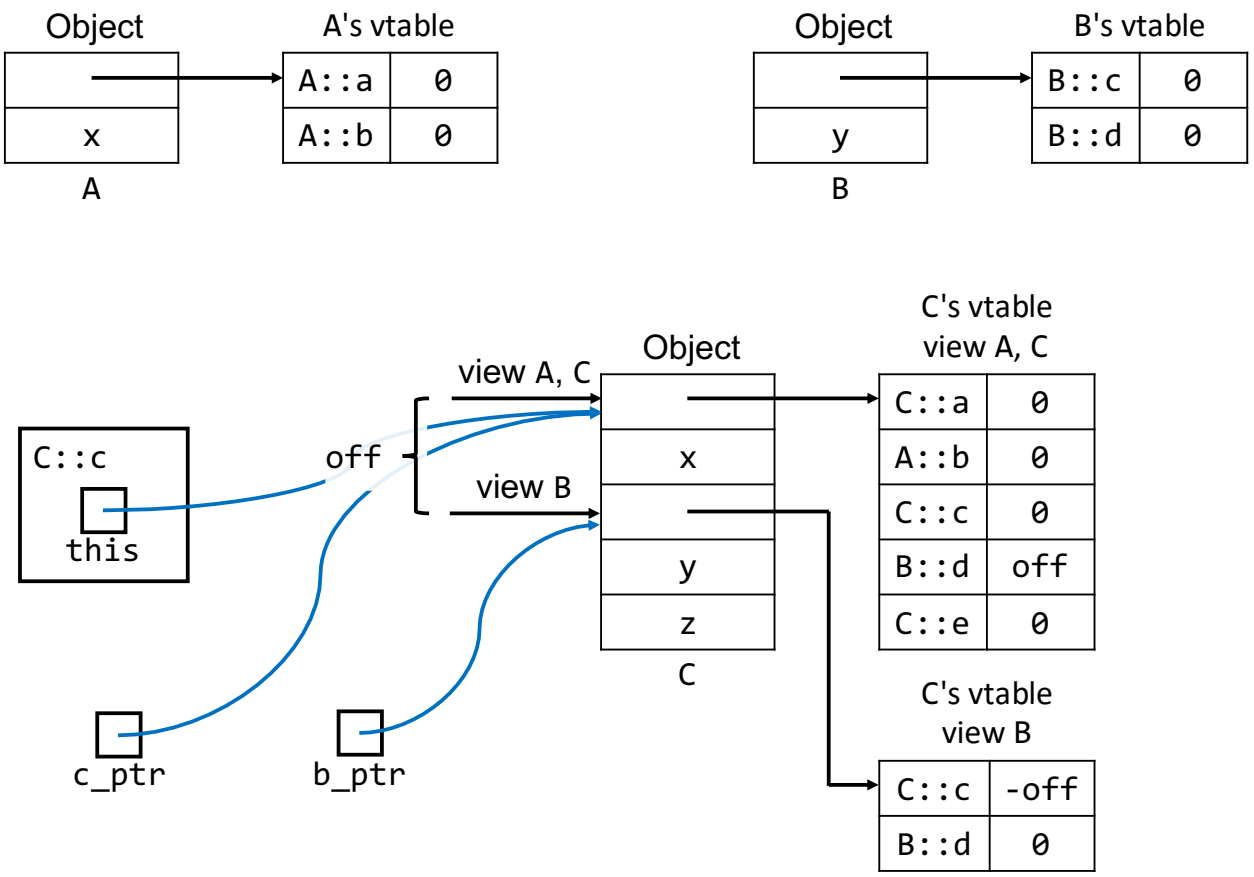



Figure 19.4: Calling a base-class method on an object that uses multiple inheritance may require a this-pointer correction to switch from one view of the object to another.

Now, when the entry for C::c is looked up in C's vtable for view B, the this pointer in C::c should be corrected by -off before it is invoked, where off is the distance between the C and B views of an object of type C. This will ensure

that `C::c` receives the `C` view of the object.

In practice, a *thunk* (a compiler-generated function) is often used to both perform this correction and call the target method. The vtable entry for the method can then store a pointer to the thunk, and no offset need be stored in the vtable. This avoids replicating the correction code everywhere a method is called.

Another complication arises when multiple base classes define the same function, as in the Python example above. The following is the same example in C++:

```
class Animal {
public:
  void defend() const {
    cout << "run away!" << endl;
  }
};

class Insect : public Animal {
};

class WingedAnimal : public Animal {
public:
  void defend() const {
    cout << "fly away!" << endl;
  }
};

class Butterfly : public Insect, public WingedAnimal {
};
```

A call to `defend()` on a `Butterfly` object can resolve to either the version in `Animal` or `WingedAnimal`. Vtables alone cannot solve this problem, and a more involved dynamic lookup process such as C3 linearization would be required instead. However, C++ considers such a method call to be ambiguous and will produce a compile-time error if the call is attempted. Instead, C++ requires the programmer to select a specific version using the scope-resolution operator:

```
Butterfly bf;
bf.WingedAnimal::defend();
```

A final consideration in record-based implementations is how to handle the *diamond problem*, where a single class occurs multiple times as the base class for another class:

```
struct A {
  int x;
};

struct B : A {
};

struct C : A {
};

struct D : B, C {
};
```

In the example above, `D` derives from `A` twice, once through `B` and once through `C`. Should an object of type `D` contain a single copy of the fields inherited from `A`, or should there be two copies? Different situations may call for different

approaches, and C++ allows both. The default is replication, but a shared copy of `A` can be specified using *virtual inheritance*:

```
struct A {
  int x;
};

struct B : virtual A {
};

struct C : virtual A {
};

struct D : B, C {
};
```

Virtual inheritance is commonly implemented by introducing indirection to access data members of the virtual base class, in a manner similar to vtables and vtable pointers.

As this example demonstrates, the intermediate classes `B` and `C` are the ones that must declare `A` as a virtual base class, even though it is the class `D` that actually gives rise to the diamond problem. This implies that the writer of the intermediate classes must know a priori that derived classes may run into the diamond problem. Thus, to some degree, this breaks the abstraction barrier between base and derived classes.

CHAPTER

TWENTY

# STATIC ANALYSIS

In processing a program, a compiler performs *static analysis* on the source code without actually running the program. Analysis is done to detect bugs in programs as well as to determine information that can be used to generate optimized code. However, as demonstrated by Rice's theorem, the general problem of static analysis of the behavior of a program written in a Turing-complete language is *undecidable*, meaning that it is not solvable by a computer. Thus, program analysis on Turing-complete languages can only approximate the answer, and there will always be cases where the result determined by the analysis is incorrect.

In designing an analysis, we usually make the choice between the analysis being *sound* or *complete*, which characterizes where the analysis may produce an incorrect result:

- A *sound* analysis only accepts programs that are correct with respect to the behavior being analyzed. If a program is incorrect, the analysis is guaranteed to reject it. On the other hand, if a program is correct, it is possible for the analysis to reject it, resulting in *false negatives*. Since a sound analysis only accepts correct programs, if a program passes the analysis, we know that it must be correct. However, if a program fails the analysis, it may or may not be correct.
- A *complete* analysis accepts all programs that are correct with respect to the behavior being analyzed. If a program is incorrect, it is possible for the analysis to accept it, resulting in *false positives*. If a program fails the analysis, it must be erroneous, but if the program passes the analysis, it may or may not be correct.

It is often the case that static analyses are designed to be sound, while dynamic (runtime) analyses are typically designed to be complete.

An analysis cannot be both sound and complete, but it is possible for an analysis to be neither. In such a case, it produces both false positives and false negatives, which is undesirable. However, in practice, real-world analyses often end up being neither sound nor complete due to the complexity of the problems they are trying to solve.

We proceed to discuss two common forms of static analysis, on types and on control flow.

## 20.1 Types

In *Formal Type Systems*, we explored the theoretical underpinnings of types and type checking. Here, we take a less formal look at how languages handle types, reviewing some concepts from type checking along the way.

In most programming languages, expressions and objects have a type associated with them. An object's type determines how its data are interpreted; all data are represented as bits, and it is a datum's type that determines the meaning of those bits. Types also prevent common errors, such as attempting to perform a semantically invalid operation like adding a floating-point number and an array. For languages in which types of variables and functions are specified in the source code, they also serve as useful documentation concerning for what a variable or function is used. In languages that support ad-hoc polymorphism in the form of operator or function overloading, types determine the specific operation to be applied to the input data. Finally, types enable compilers to generate code that is specialized to the type of an object or expression.

Compilers perform *type checking* to ensure that types are used in semantically valid ways in a program. Languages that enable static analysis to perform type checking at compile time are *statically typed*, while those that can only be checked at runtime are *dynamically typed*. Many languages use a mixture of static and dynamic type checking.

Languages often provide a predefined set of *primitive* types, such as integers, floating-point numbers, and characters, as well as a mechanism for constructing *composite* types whose components, or *fields*, are simpler types. Common examples are arrays, lists, and *records*, the latter of which are known as *structs* in C and C++.

### 20.1.1 Type Equivalence

In some languages, composite types are distinguished by their structure, so that all types with the same structure are considered to be equivalent. This strategy is called *structural equivalence*, and under this scheme, the following two types (using C-like syntax) would be equivalent:

```
record A {
  int a;
  int b;
};

record B {
  int a;
  int b;
};
```

In a handful of languages, such as ML, reordering the fields does not affect type equivalence. Thus, a type such as the following would also be equivalent:

```
record C {
  int b;
  int a;
};
```

Most modern languages, on the other hand, use *name equivalence*, which distinguishes between different occurrences of definitions within a program. Under name equivalence, the types `A` and `B` above would be considered distinct.

Some languages allow aliases to be defined for an existing type, such as the following declarations in C++:

```
typedef double weight;
using height = double;
```

Under *strict name equivalence*, aliases are considered distinct types, so that `weight` and `height` are not equivalent. This can prevent errors involving inadvertently interchanging types that alias the same underlying type but are semantically distinct, as in the following involving `weight` and `height`:

```
height h = weight(200.);
```

The languages in the C family, however, have *loose name equivalence*, so that aliases are considered equivalent to each other and to the original type. The code above is permitted under loose name equivalence.

### 20.1.2 Type Compatibility

In most languages, strict equivalence of types is not required in order for the types to be used in the same context. Rather, most languages specify *type compatibility* rules that determine when one type can be used where another one is expected.

*Subtype polymorphism* is one example of type compatibility. Languages that support subtypes, such as those that support the object-oriented paradigm, allow an object of a derived type to be used where an object of a base type is expected.

In other contexts, a language may allow a type to be used where another is expected by converting a value of the former type to the latter. Such a conversion, when done implicitly, is called a *type coercion*. A common example is when performing an arithmetic operation on different numeric types. In an expression such as `a + b`, if one of the operands has integral type and the other has floating-point type, most languages coerce the integral value to floating-point before performing the addition. Languages usually specify rules for which numeric types are coerced, or *promoted*, to others. A few languages, such as C++, include a mechanism for defining type coercions on user-defined types.

Some languages allow coercion when initializing or assigning an object with a value from a different type. For numeric types, some languages only allow initialization or assignment that performs a coercion that follows the type promotion rules. For example, in Java, coercing an `int` value to a `double` is allowed, while the latter is prohibited:

```
int x = 3.4;   // error
double y = 3;  // OK
```

The promotion rules are often designed to avoid loss of information. In particular, converting a `double` value to an `int` loses information about the fractional part of the value. In other languages, however, such as C and C++, lossy coercions are permitted, and both definitions above would be accepted.

Another common example of coercion that we've already seen is that of l-values to r-values, where an r-value is expected.

Languages with type qualifiers specify rules for when a type with one qualification can be coerced to the same type with a different qualification. For example, C++ specifies when `const` and non-`const` types can be coerced to each other. In particular, a non-`const` l-value can be coerced to a `const` l-value, but the reverse is not allowed without an explicit `const_cast`. On the other hand, a `const` l-value can be coerced to a non-`const` r-value. The following illustrates some examples:

```
int a = 3;
const int b = a;    // OK: l-value to r-value
a = b;              // OK: const l-value to r-value
int &c = a;         // OK: no coercion
int &d = b;         // ERROR: const l-value to non-const l-value
const int &e = a;   // OK: non-const l-value to const l-value
```

In order to check the types in a program, a strongly typed language determines the type of every expression in the program. For example, in the compound expression `a + b + c`, the type of the subexpression `a + b` must be known in order to determine what operation to apply to its result and `c`, whether or not a coercion is necessary or permitted. In the case of a function-call expression, the type of the expression is the return type of the function. In the case of an operator, the language defines what the type of the expression is based on the types of the operands.

The following is an example in C++:

```
cout << ("Weight is " + to_string(10) + " grams") << endl;
```

The `to_string()` function returns a `string`, so that is the type of the expression `to_string(10)`. Applying the + operator to a `string` and a string (character-array) literal in turn results in `string`. Applying the << operator to an `ostream&` and a `string` results in an `ostream&`. Lastly, `endl` is a function that is an *I/O manipulator*, and applying << to an `ostream&` and such a function also produces an `ostream&`.

A particular non-trivial case is that of the conditional expression in languages that use static typing. Consider the following example in C++:

```
int x = 3;
double y = 3.4;
rand() < RAND_MAX / 2 ? x : x + 1;
rand() < RAND_MAX / 2 ? x : y;
```

What are the types of the conditional expression? In the first case, both options are of type `int`, so the result should be of type `int`. In the second case, however, one option is of type `int` while the other is of type `double`. C++ uses a complex set of rules to determine which of the two types can be coerced to the other, and the coercion rules here differ from those in other contexts. The expression is only valid if exactly one of the types can be coerced to the other. In this case, the resulting expression has type `double`.

### 20.1.3 Type Inference

Since the type of each expression is not specified in source code, compilers perform *type inference* to compute their types. Some languages allow programmers to make use of the type-inference facilities of the compiler by allowing types to be elided from declarations if they can be inferred. Many modern statically typed languages allow types to be elided completely in certain contexts.

As an example, we have already seen that Java and C++ allow the return type to be elided from a *lambda expression*, and that Java also allows the parameter types to be elided:

```
public static IntPredicate makeGreaterThan(int threshold) {
  return value -> value > threshold;
}
```

We have also seen that C++ allows the type of a variable to be deduced with the `auto` keyword:

```
int x = 3;
auto y = x;    // deduced to have type int
auto &z = x;   // deduced to have type int &
```

The rules for type deduction in C++ have complex interactions with reference types, as illustrated above. We will not consider them here.

The `auto` keyword requires that a variable be initialized at declaration, so that the type can be deduced from the initializer. There are cases where this is not possible. Consider the following class template:

```
template<typename T, typename U>
class Foo {
  T a;
  U b;
  ??? c;   // type of a + b
};
```

Here, we want the type of `Foo::c` to be the same as the type of `a + b`, but without actually initializing it to that value. In fact, C++ prohibits `auto` from being used with a non-static class member. Instead, C++ provides the `decltype` keyword that computes the type of an expression:

```
template<typename T, typename U>
class Foo {
  T a;
```

(continues on next page)

(continued from previous page)

```
  U b;
  decltype(a + b) c;   // type of a + b
};
```

## 20.2 Control-Flow Analysis

Compilers often perform analysis on the flow of control or data through a program in order to provide early detection of bugs as well as to optimize generated code. Such an analysis is referred to as *control-flow* or *data-flow analysis*. Here, we consider a few common examples of control-flow analysis.

Many imperative languages allow variables to be declared without being explicitly initialized. Some languages specify semantics for default initialization. In C and C++, however, variables of primitive type have undefined values upon default initialization, so the behavior of a program that uses such a variable is undefined. Other languages, such as Java, reject programs in which it cannot be proven that a variable has been initialized before being used. The compiler analyzes the source code to determine whether or not a control-flow path exists that may result in the use of a variable without initialization. This analysis is conservative, so that the standard Java compiler rejects the following code:

```
class Foo {
  public static void main(String[] args) {
    int i;
    if (args.length > 0) {
      i = args.length;
    }
    if (args.length <= 0) {
      i = 0;
    }
    System.out.println(i);
  }
}
```

Even though it may seem obvious that the body of one of the conditionals must be executed, the compiler is unable to determine that this is the case. Instead, it conservatively assumes that it is possible for neither conditional test to succeed, so that `i` may be used uninitialized. Thus, the compiler reports an error like the following:

```
foo.java:10: error: variable i might not have been initialized
    System.out.println(i);
                       ^
1 error
```

On the other hand, modifying the code as follows enables the compiler to determine that `i` must be initialized:

```
class Foo {
  public static void main(String[] args) {
    int i;
    if (args.length > 0) {
      i = args.length;
    } else {
      i = 0;
    }
    System.out.println(i);
```

(continues on next page)

(continued from previous page)

```
  }
}
```

Here, the compiler can determine that one of the two branches of the conditional must execute, so that `i` is always initialized before use.

Some C and C++ compilers perform the same analysis and report a warning if a default-initialized variable of primitive type may be used. Java also performs a similar analysis to ensure that `final` variables are initialized no more than once.

In languages that require a function to explicitly return an object, a program may have control paths that do not ensure that a function encounters a return statement before exiting. Compilers often perform an analysis that is analogous to that of variable initialization in order to ensure that a function reaches a return statement. Consider the following method in Java:

```
static int bar(int x) {
  if (x > 0) {
    return 1;
  }
  if (x <= 0) {
    return 0;
  }
}
```

Once again, the compiler cannot guarantee that one of the conditionals will have its body executed, and it reports an error such as the following:

```
bar.java:9: error: missing return statement
  }
  ^
1 error
```

An equivalent example in C++ produces a warning in some compilers, such as the following in Clang:

```
bar.cpp:12:1: warning: control may reach end of non-void function
      [-Wreturn-type]
}
^
1 warning generated.
```

In some non-obvious cases, the compiler can guarantee that a return must be reached before a function exits. The following example succeeds in both the standard Java compiler and in Clang for equivalent C++ code:

```
static int baz(int x) {
  while (true) {
    if (x < 0) {
      return 0;
    }
  }
}
```

Here, the compiler can determine that the only way to exit the loop is through a return, so that the only way to exit the function is by reaching the return statement.

The same analysis can be used to detect code that will never be reached, and depending on the language and compiler,

this may be considered an error. For example, the following modification to `baz()` is rejected by the standard Java compiler:

```
static int baz(int x) {
  while (true) {
    if (x < 0) {
      return 0;
    }
  }
  return 1;
}
```

The compiler reports the following error:

```
baz.java:8: error: unreachable statement
    return 1;
    ^
1 error
```

In Java, the language explicitly disallows statements that can be proven to be unreachable.

CHAPTER

# TWENTYONE

# DYNAMIC TYPING

In addition to dynamic binding, languages and implementations often make other uses of dynamic type information, also called *runtime type information (RTTI)*, as well as making it available in some form to programmers.

Many languages provide a mechanism for checking whether or not an object has a specific type at runtime. Depending on the language, the query type may need to be specified at compile time, particularly if the language does not support first-class types, or it may be computed at runtime. For example, the following C++ code checks whether the dynamic type of an object referred to by a base class pointer is of the derived class:

```
struct A {
  virtual void bar() {
  }
};

struct B : A {
};

void foo(A *a) {
  if (dynamic_cast<B *>(a)) {
    cout << "got a B" << endl;
  } else {
    cout << "not a B" << endl;
  }
}

int main() {
  A a;
  B b;
  foo(&a);
  foo(&b);
}
```

The `dynamic_cast` operation attempts to cast an `A *` to a `B *`, which will only succeed if the pointed-to object is actually an instance of `B`. If the cast fails, then it produces a null pointer, which has truth value false. C++ also allows `dynamic_cast` to be used on references, in which case an exception is thrown upon failure.

In order for `dynamic_cast` to work, the types involved must define at least one virtual method. This allows an implementation to use vtable pointers, or entries in the vtable itself, to determine the dynamic type of an object. Types that do not have virtual methods do not have vtables, and their instances do not include vtable pointers.

C++ also has the `typeid` operator, which produces an object that contains information about the type of the given operand. In order to make use of `typeid`, the `<typeinfo>` header must be included. The operator works on objects of any type, as well as types themselves, and the result is an instance of `std::type_info`, which contains basic information about the type. The following is an example:

```
int main() {
  const type_info &i1 = typeid(int);
  const type_info &i2 = typeid(new A());
  const type_info &i3 = typeid(main);
  cout << i1.name() << " " << i2.name() << " " << i3.name() << endl;
}
```

The resulting names are implementation dependent. For example, GCC 5.5 produces `i P1A FivE` when the code above is run.

Java supports the `instanceof` operator, which determines whether or not an object is an instance of the given type at runtime. Python has the similar `isinstance()` function, which takes in an object and a type as arguments.

Java also supports an operation similar to `typeid` in the form of the `getClass()` method defined on all objects. The result is an instance of `Class`, which contains extensive information about the class of the object. Similarly, Python has a `type()` function. This returns the actual type of an object, since types are first-class entities in Python.

In Java, all casts on objects are dynamically checked. Rather than producing a null pointer on failure, Java throws a `ClassCastException`.

A specific case where Java needs to check the type of an object in its internal implementation is when an item is stored in an array. Originally, Java did not support parametric polymorphism, so the decision was made to support polymorphic functions on arrays by making all arrays whose elements are of object type derive from `Object[]`. This allowed methods like the following to be defined and called on any array of object type:

```
static void printAll(Object[] items) {
  for (int i = 0; i < items.length; i++) {
    System.out.println(items[i]);
  }
}
```

More specifically, Java specifies that `A[]` is a subtype of `B[]` if `A` is a subtype of `B`.

As an example of where this subtype relation can permit erroneous code, consider the following:

```
String[] sarray = new String[] { "foo", "bar" };
Object[] oarray = sarray;
oarray[0] = "Hello";
oarray[1] = new Integer(3);
sarray[1].length();
```

The second line is valid, since a `String[]` object can be assigned to a variable of type `Object[]`. The third line is also valid, since a `String` object can be stored in an `Object[]`. The fourth line is valid according to the type system, since `Integer` derives from `Object`, which can be stored in an element of an `Object[]` variable. However, `Integer` does not derive from `String`, so at runtime, we have an attempt to store an `Integer` object into an array of dynamic type `String[]`. This should be prevented, since we could then call a `String` method on the element as in the fifth line. Thus, Java checks the dynamic types of the array and the item being stored at runtime and throws an `ArrayStoreException` if they are incompatible.

A better solution to the problem would be to use parametric polymorphism for operations on arrays, rather than making arrays support subtype polymorphism. Unfortunately, parametric polymorphism was introduced much later in Java's existence, leading to a significant body of code that depends on the subtype polymorphism of arrays.

CHAPTER

# TWENTYTWO

# GENERICS

Subtype polymorphism relies on subtype relationships and dynamic binding in order to provide the ability of a single piece of code to behave according to the dynamic type of an object. In contrast, *parameteric polymorphism* allows the same code to operate on different types without relying on either subtype relationships or dynamic binding. Languages that support parametric polymorphism do so in different ways, and we will examine the different strategies here.

## 22.1 Implicit Parametric Polymorphism

Many functional languages in the ML family, including OCaml and Haskell, are statically typed but allow the programmer to elide types from a function. In such a case, the function is implicitly polymorphic, and the compiler will infer the types for each use of the function. For example, the following defines a polymorphic `max` function in OCaml:

```
let max x y =
    if x > y then
        x
    else
        y;;
```

We can then call the function on two values of the same type:

```
# max 3 4;;
- : int = 4
# max 4.1 3.1;;
- : float = 4.1
# max "Hello" "World";;
- : string = "World"
```

## 22.2 Explicit Parametric Polymorphism

In other languages, a function or type must be explicitly specified as polymorphic. In C++, the `template` keyword introduces a polymorphic entity, and the parameters are specified in angle brackets:

```
template <typename T>
T max(const T &x, const T &y) {
  return x > y ? x : y;
}
```

While the definition of a parametric function must be explicitly denoted as such, in many languages the use of a parametric function does not normally require an explicit instantiation. Instead, as in implicit parametric polymorphism, the compiler uses type inference to determine the appropriate instantiation. Thus, we can use `max()` as follows:

```
max(3, 4); // returns 4
max(4.1, 3.1); // returns 4.1
max("Hello"s, "World"s) // returns "World"s
```

In the last call, we made use of C++14 string literals to compare `std::string` s rather than character arrays.

With a single template parameter, the compiler cannot infer the type parameter on a call that uses arguments of different types:

```
max(3, 4.1); // error
```

Instead, we can explicitly instantiate `max()`:

```
max<double>(3, 4.1); // OK
```

Alternatively, we can modify `max()` to have separate type parameters for each function parameter. However, with C++11, we also need to make use of *type deduction* for the return type:

```
template <typename T, typename U>
auto max(const T &x, const U &y) -> decltype(x > y ? x : y) {
  return x > y ? x : y;
}
```

As of C++14, the trailing return type can be elided, in which case the return type is deduced from the return statement:

```
template <typename T, typename U>
auto max(const T &x, const U &y) {
  return x > y ? x : y;
}
```

### 22.2.1 Non-Type Parameters

In some languages, a generic parameter need not be a type. In particular, Ada allows generics to be parameterized by values of any type. C++ is more restrictive, allowing a template parameter to be a value of an integral type, enumeration type, lvalue-reference type, pointer type, or pointer-to-member type. The template parameter must be a compile-time constant. A specific example of this is `std::array`, which is declared similar to the following:

```
template <typename T, int N>
class array;
```

We can then use it as follows:

```
array<double, 5> arr;
arr[3] = 4.1;
```

### 22.2.2 Constraints

An entity that supports parametric polymorphism can work with different types, but it is often the case that not every type is suitable for use in that entity. In the case of the `max` functions above, it does not make sense to call `max` on values of a type that does not support the > operator.

Depending on the language, the constraints on a polymorphic entity can be implicit or explicit. In the case of implicit constraints, the entity is instantiated for the given type argument, and then the result is checked for correctness. As an example, if we attempt to call `max()` on streams in C++, we get an error like the following:

```
foo.cpp:7:12: error: invalid operands to binary expression
      ('const std::__1::basic_istream<char>' and
      'const std::__1::basic_istream<char>')
  return x > y ? x : y;
         ~ ^ ~
foo.cpp:11:5: note: in instantiation of function template
      specialization 'max<std::__1::basic_istream<char> >'
      requested here
  ::max(cin, cin);
    ^
```

We then get a lengthy list of all the generic overloads of the operator < that could not be instantiated with a `basic_istream<char>`. The inscrutability of error messages produced by C++ compilers upon instantiation failure is an unavoidable byproduct of deferring type checking until instantiation.

Other languages allow a generic entity to specify explicit constraints on the arguments with which the entity can be instantiated. Java and C#, in particular, support powerful systems of constraints that can restrict a generic for use with derived classes of specific types. The code for a generic entity can then be checked once, assuming that the constraints are satisfied. Then upon instantiating a generic, the type arguments need only be checked against the constraints, resulting in much cleaner error messages than C++.

We will look at the Java system for generics in more detail shortly.

### 22.2.3 Implementation

Languages and compilers also differ in the implementation of generics at runtime. In languages with strong support for dynamic binding, a common implementation strategy is to only produce a single copy of the code for a generic entity, relying on operations that depend on the type parameter to be dynamically bound to the appropriate implementation. This is the strategy used by Java and ML.

An alternative implementation is to generate separate code for each instantiation of a generic entity, as is done in C++. This approach is more flexible, since it does not require there to be a single piece of generated code that works for any set of type arguments. It is also often more efficient, since it does not rely on dynamic binding. The downside is that it results in larger executables, a problem that is exacerbated by the fact that the compiler needs access to the full source of a generic entity when it is being instantiated. This can lead to multiple copies of the same instantiation being included in the resulting executable.

### 22.2.4 Java Generics

We now examine Java's support for generics in more detail, as there are key differences between how Java and C++ implement generics.

In Java, the basic syntax for using a generic is similar to C++. For example, the following uses the generic `ArrayList<T>` type:

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("Hello");
strings.add("World");
System.out.println(strings.get(1));
```

Defining a generic type, in its most basic form, also has syntax that is related to C++, except for the distinct lack of the `template` keyword:

```
class Foo<T> {
  private T x;

  public Foo(T x_in) {
    x = x_in;
  }

  public T get() {
    return x;
  }
}
```

A generic function requires its type parameters to be specified prior to the return type, as the return type may use the type parameter:

```
static <T> T max(T x, T y) {
  return x.compareTo(y) > 0 ? x : y;
}
```

Unfortunately, this code will fail to compile, since not all objects support the `compareTo()` method. By default, Java only allows methods defined on `Object` to be called from within a generic. The `compareTo()` method is not defined in `Object` but is defined in the following interface in the standard library:

```
interface Comparable<T> {
  int compareTo(T other);
}
```

Thus, we need to a mechanism for constraining the type parameter of `max()` be a derived type of `Comparable<T>`, so that an object of the type parameter can be compared to another object of the same type. We can do this by adding `extends Comparable<T>` to the type parameter when we introduce it:

```
static <T extends Comparable<T>> T max(T x, T y) {
  return x.compareTo(y) > 0 ? x : y;
}
```

We can modify the `Foo` class as follows to implement the `Comparable<Foo<T>>` interface:

```
class Foo<T> implements Comparable<Foo<T>> {
  private T x;
```

(continues on next page)

(continued from previous page)

```
  public Foo(T x_in) {
    x = x_in;
  }

  public T get() {
    return x;
  }

  public int compareTo(Foo<T> other) {
    return x.compareTo(other.x);
  }
}
```

To compare a `Foo<T>` to another `Foo<T>`, we in turn compare their respective `x` fields with another call to `compareTo()`. Again, we run into the problem that the type parameter `T`, which is the type of `x`, may not implement the `compareTo()` method. So we have to specify the constraint here as well that `T` be derived from `Comparable<T>`:

```
class Foo<T extends Comparable<T>> implements Comparable<Foo<T>> {
  private T x;

  public Foo(T x_in) {
    x = x_in;
  }

  public T get() {
    return x;
  }

  public int compareTo(Foo<T> other) {
    return x.compareTo(other.x);
  }
}
```

We can now use `max()` with instantiations of `Foo`:

```
Foo<String> f1 = new Foo<String>("Hello");
Foo<String> f2 = new Foo<String>("World");
System.out.println(max(f1, f2).get());     // prints World
```

A final problem is that an instance of a class may be comparable to an instance of a base class. Consider the following classes:

```
class Rectangle implements Comparable<Rectangle> {
  private int side1, side2;

  public Rectangle(int s1_in, int s2_in) {
    side1 = s1_in;
    side2 = s2_in;
  }

  public int area() {
    return side1 * side2;
```

(continues on next page)

(continued from previous page)

```
  }

  public int compareTo(Rectangle other) {
    return area() - other.area();
  }
}

class Square extends Rectangle {
  public Square(int side) {
    super(side, side);
  }
}
```

We can now try to use the `Foo` generic type with `Square`, as in:

```
public static void main(String[] args) {
  Foo<Square> f1 = new Foo<Square>(new Square(3));
  Foo<Square> f2 = new Foo<Square>(new Square(4));
  System.out.println(f1.compareTo(f2));
}
```

Unfortunately, we get errors like the following:

```
foo.java:36: error: type argument Square is not within bounds
  of type-variable T
    Foo<Square> f1 = new Foo<Square>(new Square(3));
        ^
  where T is a type-variable:
    T extends Comparable<T> declared in class Foo
```

The problem is that `Square` derives from `Comparable<Rectangle>`, not `Comparable<Square>` as required by the type parameter. However, semantically it should not be a problem, since if a `Square` can be compared to another `Rectangle`, it can also be compared to another `Square`. The solution is to modify the type constraint to allow a type argument as long as it is comparable to some superclass of the type:

```
class Foo<T extends Comparable<? super T>>
  implements Comparable<Foo<T>> {
  ...
}
```

The syntax `Comparable<? super T>` specifies that the type argument of `Comparable` can be any type, as long as it is a supertype of `T`. Thus, `Square` satisfies the constraint, since it derives from `Comparable<Rectangle>`, and `Rectangle` is a superclass of `Square`.

Java implements generics using *type erasure*. Once a generic has been checked, using any constraints it specifies, and once all uses have been checked, the generic is replaced with a version that is no longer parameterized, usually with the type parameters replaced by `Object`. This prevents a generic from being used directly with primitive types, since they do not derive from `Object`. However, Java does allow primitives to be implicitly converted to representations that derive from `Object`, at significant efficiency costs.

### 22.2.5 Curiously Recurring Template Pattern

In Java, the pattern of a type `T` deriving from a generic instantiated with `T` is quite common, as in the `Rectangle` class above. This pattern also exists in C++ templates, and it is known as the *curiously recurring template pattern (CRTP)*.

```
template<class T>
class GenericBase {
  ...
};

class Derived : public GenericBase<Derived> {
  ...
};
```

We can use such a pattern to construct a mixin, as in Ruby's `Comparable` mentioned in *Types of Inheritance*:

```
template<class T>
class Comparable {
public:
  bool operator<(const T &rhs) const {
    return compare(rhs) < 0;
  }

  bool operator<=(const T &rhs) const {
    return compare(rhs) <= 0;
  }

  ...

  virtual int compare(const T &other) const = 0;
};
```

The `Comparable` class template defines the comparison operators in terms of a `compare()` method, which the derived class must implement. We can thus implement a counter class that inherits the comparison operators:

```
class Counter : public Comparable<Counter> {
  int count = 0;

public:
  void increment() {
    ++count;
  }

  void decrement() {
    --count;
  }

  int get_count() const {
    return count;
  }

  virtual int compare(const Counter &other) const override {
    return count - other.count;
  }
```

(continues on next page)

(continued from previous page)

```
};
```

While the code above works, a major drawback of the implementation is that it requires dynamic binding, incurring the cost of a vtable pointer in every `Counter` object as well as a vtable lookup for each application of a comparison operator.

Surprisingly, we can actually eliminate dynamic binding by adding an implicit constraint to the `Comparable` class template: an instance of `Comparable<T>` must also be an instance of `T`. For example, a `Counter` object is an instance of `Comparable<Counter>`, but of course it is also an instance of `Counter`. With this constraint, we can perform an unchecked type cast of a `Comparable<T> *` down to `T *`:

```
template<class T>
class Comparable {
public:
  bool operator<(const T &rhs) const {
    return static_cast<const T *>(this)->compare(rhs) < 0;
  }

  ...
};
```

With the type cast, we no longer need to define `compare()` as a pure virtual method in `Comparable`. It need only exist in `T`, and it may be defined as a non-virtual function:

```
class Counter : public Comparable<Counter> {
  int count = 0;

public:
  ...

  int compare(const Counter &other) const {
    return count - other.count;
  }
};
```

The end result is polymorphism without dynamic binding, and it is known as *static polymorphism*[2] or *simulated dynamic binding*. The pattern is widely used in Microsoft's Active Template Library (ATL) and Windows Template Library (WTL) for development on Windows.

## 22.3 Duck Typing

Languages that do not have static typing are often implicitly polymorphic. Type information is not available at compile time, so a function is usable with values of any type that supports the required operations. This is called *duck typing*: it doesn't matter what the type of the value actually is; as long as it looks like a duck and quacks like a duck, it is considered for all intents and purposes a duck.

As an example, the following is a definition of `max()` in Python:

```
def max(x, y):
    return x if x > y else y
```

[2] The term "static polymorphism" is also used to mean parametric polymorphism, so the term "simulated dynamic binding" is preferable.

The function will work at runtime on any types that support the special `__gt__` method, which is called by the > comparison.

A downside of duck typing is that whether or not a type is considered to support an operation is based solely on the name of the operation, which may not have the same semantic meaning in different contexts. For example, a `run()` method on an `Athlete` object may tell the athlete to start running in a marathon, while a `run()` method on a `Thread` object may tell it to start executing some code. This can lead to unexpected behavior and confusing errors in duck-typed code that calls `run()`.

CHAPTER

# TWENTYTHREE

# MODULES AND NAMESPACES

An abstract data type (ADT) defines an abstraction for a single type. Some abstractions, however, consist of not just a single type, but a collection of interdependent types, variables, and other entities. Such a collection of items is a *module*, and the modularization of a system is a means of making its maintenance more manageable.

Many languages provide mechanisms for organizing items into modules. In some languages, the mechanism is closely tied to that used for separate compilation, such that each module is compiled independently and later linked together with other modules. In other languages, the mechanisms for modules and separate compilation are independent.

## 23.1 Translation Units

A *translation unit* or *compilation unit* is the unit of compilation in languages that support separate compilation. Often, it consists of a single source file. In languages such as C and C++ that enable other files to be included with a preprocessor directive, a translation unit consists of a source file and all the files that it recursively includes.

In order to support separate compilation, a translation unit need only know basic information about entities in other translation units. For example, in C++, only declarations of external entities that are used need be known[3]. For a variable, a declaration provides the name and type, and for functions, the name, return type, and parameter type. For classes, in order to be able to access members, the class declaration with its member declarations needs to be available, though actual definitions of member functions do not. Normally, this is accomplished by writing declarations in a header file and then including the header file in any translation unit that needs access to those declarations. The definitions of variables, functions, and member functions are written in a separate source file, which will usually be compiled as its own translation unit.

As an example, the following may be placed in the header file `Triangle.hpp` to provide the declarations for a `Triangle` ADT:

```
class Triangle {
  double a, b, c;
public:
  Triangle();
  Triangle(double, double, double);
  double area() const;
  double perimeter() const;
  void scale(double s);
};
```

Then the definitions would be placed in a `Triangle.cpp` file:

[3] Templates are an exception, since their definitions need to be instantiated upon use. Thus, the compiler must have the definitions available for templates.

```
#include "Triangle.hpp"

Triangle::Triangle()
  : Triangle(1, 1, 1) { }

Triangle::Triangle(double a_in, double b_in, double c_in)
  : a(a_in), b(b_in), c(c_in) { }

double Triangle::area() const {
  return a * b * c;
}

double Triangle::perimeter() const {
  return a + b + c;
}

void Triangle::scale(double s) {
  a *= s;
  b *= s;
  c *= s;
}
```

The `#include` directive pulls the code from `Triangle.hpp` into `Triangle.cpp`, making the `Triangle` declarations available to the latter.

In other languages, including Java and C#, there is no notion of a separate header file, and all declarations must also be definitions. Instead, the compiler automatically extracts the declaration information from a source file when needed by other translation units.

## 23.2 Modules, Packages, and Namespaces

Languages also specify units of organization for names in a program. This allows the same name to be used in different units without resulting in a conflict. In many cases, the unit of organization is at the granularity of a source file, while in other languages, an organizational unit can span multiple source files.

In Python, the first unit of organization is a source file, which is called a *module* in Python terminology. A module is associated with a scope in which the names defined in the module reside. In order to use names from another module, the external module, or names from within it, must be explicitly imported into the current scope. The `import` statement does so, and it can be located at any scope. Consider the following example:

```
from math import sqrt

def quadratic_formula(a, b, c):
    disc = sqrt(b * b - 4 * a * c)
    return (-b + disc) / (2 * a), (-b - disc) / (2 * a)

def main():
    import sys
    if len(sys.argv) < 4:
        print('Usage: {0} a b c'.format(sys.argv[0]))
    else:
        print(quadratic_formula(int(sys.argv[1]),
```

(continues on next page)

(continued from previous page)

```
                                int(sys.argv[2]),
                                int(sys.argv[3])))

if __name__ == '__main__':
    main()
```

In the code above, the import statement in the first line directly imports the `sqrt` name from the `math` module into the scope of the current module. It does not, however, import the `math` name itself. In the first line of `main()`, the name of the `sys` module is imported into the local scope of `main()`. The standard dot syntax can be used to refer to a name nested inside of `sys`.

Python also allows a second level of organization in the form of a *package*, which is a collection of modules. For example, if the code above were in a module named `quadratic`, we might want to organize it with other mathematical formulas in a package named `formulas`. Defining a file `__init__.py` within a directory enables the modules in that directory to constitute a package, with the directory name as the name of the package. Packages can then further have subpackages in the form of subdirectories with their own `__init__.py` files.

The following is an example of how a sound module can be organized in Python:

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Subpackage for sound effects
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                  Subpackage for filters
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Java follows a similar organizational scheme as Python. The first unit of organization is a class, since all code in Java must be contained within a class. Multiple classes may be contained within the same translation unit, but a translation unit does not constitute a scope on its own. (If a class is to be used outside a translation unit, however, it should be located in its own file in order to make it possible for the compiler to find its source when it is used.) A source file may include a `package` directive in order to place its code within the context of the specified package:

```
package formulas

public class Quadratic {
  ...
}
```

Packages can be nested, as in Python.

Also like Python, Java has import statements in order to import names into the local scope. Unlike Python, however, an unimported name can be used by giving it full *qualification*, including the sequence of packages that it is a part of:

```
java.util.Vector vec = new java.util.Vector();
```

Import statements in Java must appear at the top of a file, after any `package` declaration but before any class definition. A single member can be imported from a package, or all of the package's contents can be imported:

```
import java.util.Vector; // import just one member
import java.util.*;      // import all members
```

Java also allows static methods and constants to be imported from a class with the static import statement:

```
import static java.lang.System.out;

public class Foo {
  public static void main(String[] args) {
    out.println("Hello world!");
  }
}
```

C++ has the concept of *namespaces* rather than modules or packages. A namespace defines a scope in which names reside, and an entity can be defined within a namespace as follows:

```
namespace foo {

  struct A {
  };

  int x;
}

namespace foo {

  struct B : A {
  };

}
```

As demonstrated above, multiple entities can be defined within a single `namespace` block, and multiple `namespace` blocks can define entities for the same namespace. Namespaces can also be nested.

In order to access a namespace member from an external context, the scope-resolution operator is required:

```
foo::A *a = new foo::A;
```

C++ allows individual names to be imported from a namespace into the current scope with a `using` declaration:

```
using foo::A;
```

Alternatively, all of the names declared within a namespace may be imported as follows:

```
using namespace foo;
```

This latter form should be used with caution, as it significantly increases the likelihood of inadvertent name clashes.

An entity defined outside of a namespace is actually within the global namespace, and it can be referred to with the scope-resolution operator by including nothing on the left-hand side:

```
int bar();

void baz() {
  std::cout << ::bar() << std::endl;
}
```

Java similarly places code that is lacking a `package` specifier into the anonymous package.

C# combines the concept of Python's modules, which it calls *assemblies*, with namespaces as in C++.

## 23.3 Linkage

C does not have namespaces, so it uses an alternate mechanism to avoid name conflicts between translation units. (C++ also includes this, since it is mostly backwards compatible with C.) A programmer can specify a *linkage* for a function or variable, which determines whether or not the item is visible outside of the translation unit. The keyword `static`, when used on a function or variable at global scope, specifies that the given item has *internal linkage*, meaning that it is not available outside of the translation unit. This is crucial when the same name may be defined within different translation units, as it avoids a conflict at the link stage. In particular, global variables and functions that are not just declared but also defined in a header file should almost always be given internal linkage, since a header file is likely to be included from multiple translation units.

A global function or non-const variable has *external linkage* if it is missing the `static` specifier. This means that the name will be accessible from other translation units. A variable or function with external linkage must have exactly one definition between the translation units in a program. Otherwise, a conflict arises between the multiple definitions, and a linker error will result. For a function, the distinction between a simple declaration and a definition is clear, since the latter provides a function body. For a variable, however, a declaration is generally also a definition, since a missing initializer implies default initialization. The programmer must explicitly state that a declaration of a global variable is not a definition using the `extern` specifier:

```
extern int count; // just a declaration
int count;        // also a definition
```

A const global variable has internal linkage by default, and the `extern` keyword must be present to give it external linkage instead. An initialization can be provided, making a declaration of such a variable also a definition:

```
extern const int SIZE;      // just a declaration
extern const int SIZE = 10; // also a definition
```

## 23.4 Information Hiding

Many languages provide a mechanism for information hiding at the granularity of modules or packages. In Java, for example, a class that is declared without the `public` keyword is only available to other classes within the same package. In C and C++, the standard method of information hiding is to avoid declaring internal entities in a header file, but to declare them within a `.c` or `.cpp` file, and in the case of variables and functions, declare them with internal linkage.

As mentioned above, in order to use the members of a class in C++, the class definition itself must be available in the current translation unit. However, access to internal members of a class can be restricted using the `private` or `protected` specifiers.

C, on the other hand, does not provide a means of declaring struct members private. However, there is a common pattern of preventing direct access to struct members by providing only the declaration of a struct, without its definition, in the header file. As an example, the following defines the interface for a stack ADT:

```
typedef struct list *stack;
stack stack_make();
void stack_push(stack s, int i);
int stack_top(stack s);
void stack_pop(stack s);
void stack_free(stack s);
```

Here, no definition of the `list` struct is provided, making it an *opaque type*. This prevents another translation unit from creating a `list` object, since it can't event tell what the size of the object will be, or accessing its members directly. We can then write the definitions for the stack ADT in its own `.c` file:

```
typedef struct node {
  int datum;
  struct node *next;
} node;

struct list {
  node *first;
};

stack stack_make() {
  stack s = (stack) malloc(sizeof(struct list));
  s->first = NULL;
  return s;
}

void stack_push(stack s, int i) {
  node *new_node = (node *) malloc(sizeof(node));
  new_node->datum = i;
  new_node->next = s->first;
  s->first = new_node;
}

...
```

Another `.c` file can then make use of the stack, without being able to directly access internal details, as follows:

```
#include "stack.h"

int main(int argc, char **argv) {
  stack s = stack_make();
  stack_push(s, 3);
  stack_push(s, 4);
  printf("%d\n", stack_top(s));
  stack_pop(s);
  printf("%d\n", stack_top(s));
  stack_free(s);
}
```

## 23.5 Initialization

In a program with code organized among different modules or translation units, an important consideration is when the code that initializes a module is executed. Interdependencies between modules can lead to bugs due to the semantics of initialization, and there are cases where the only solution is to reorganize the structure of a program.

In Python, the code in a module is executed when it is first imported. Once a module has been imported, any subsequent imports of the module will not cause its code to be re-executed. However, it is possible to construct circular dependencies between modules that result in errors or unexpected behavior. Consider the following, located in module `foo`:

```
import bar

def func1():
    return bar.func3()

def func2():
    return 2

print(func1())
```

Assume that the following is located in module `bar`:

```
import foo

def func3():
    return foo.func2()
```

If we then run module `foo` from the command line, the `import` statement will cause the code in `bar` to be executed. The code in `bar` has as its first statement an import of `foo`. This is the first import of `foo` from `bar`, so the code for `foo` will execute. It starts with `import bar`; however, this is now the second import of `bar` into `foo`, so it will not have any effect. Then when `func1()` is called, the definition for `func3()` in `bar` has not yet been executed, so we will get an error:

```
Traceback (most recent call last):
  File "foo.py", line 1, in <module>
    import bar
  File "bar.py", line 1, in <module>
    import foo
  File "foo.py", line 9, in <module>
    print(func1())
  File "foo.py", line 4, in func1
    return bar.func3()
AttributeError: module 'bar' has no attribute 'func3'
```

One way to fix this is to delay the import of `foo` into `bar` until `func3()` is called:

```
def func3():
    import foo
    return foo.func2()
```

However, this still causes the code in `foo` to execute twice:

```
$ python3 foo.py
2
2
```

A better solution is to move `func2()` from `foo` into its own module, and then to import that module from both `foo` and `bar`.

In Java, the static initialization of a class occurs when it is first used, which includes creating an instance of the class or accessing a static member. Thus, the order of initialization depends on the dynamic execution of a program, and a programmer generally should not rely on a specific order of initialization between different classes.

In C++, initialization follows a multi-step process. First is what C++ calls *static initialization*, which initializes compile-time constants to their respective values and other variables with static storage duration to zero. Then is *dynamic initialization*, which runs the specified initialization for static-duration variables. In general, variables are initialized in program order within a translation unit, with some exceptions. However, the order of initialization between translation units is unspecified, and in fact may be delayed until the first time a translation unit is used. The end result is that a programmer should avoid any assumption that any other translation unit has been initialized when writing initialization code for a given translation unit.

# Part V

# Declarative Programming

Most of the languages we've considered so far in this text have followed the imperative programming model, where a computation is decomposed into individual statements that modify the state of the program. These languages have also been procedural, grouping statements into subroutines that are then called explicitly.

We have also seen the functional programming model, which decomposes a computation into functions that are closely related to those in mathematics. In such a model, programming is done with expressions that avoid side effects. We have also considered specific languages that provide a mix of the functional and imperative paradigms.

Functional programs are *declarative*, since they declare a relationship between the inputs and outputs of a function. We turn our attention to other models that are declarative, including those that express computation using logical relations, constraints, and dependencies.

CHAPTER

# TWENTYFOUR

# LOGIC PROGRAMMING

Whereas functional programming is based on the theoretical foundations of $\lambda$-calculus, logic programming is based on the foundation of formal logic. More specifically, it is based on *first-order predicate calculus*, which expresses quantified statements such as:

$$\forall X.\, \exists Y.\, P(X) \,\vee\, \neg Q(Y).$$

This states that for every value `X`, over some implicit universe of values, there is some value `Y` such that either `P(X)` is true or `Q(Y)` is false or both. This specific statement can also be written in the form of an implication:

$$\forall X.\, \exists Y.\, Q(Y) \implies P(X).$$

The implication $a \implies b$ is equivalent to $\neg a \,\vee\, b$.

In most logic languages, a program is specified in terms of *axioms* that are assumed to be true, and a programmer specifies a *goal* that the system should attempt to prove from the set of axioms. An axiom is usually written in the form of a *Horn clause*, which has the following structure:

```
H :- B1, B2, ..., BN
```

The `:-` symbol specifies a reverse implication, and the comma is used for conjunction. The equivalent form in predicate calculus is:

$$(B_1 \,\wedge\, B_2 \,\wedge\, \ldots \,\wedge\, B_N) \implies H$$

In the Horn clause above, `H` is the *head* of the clause, while `B1, B2, ..., BN` is the *body*. In natural language, the Horn clause is stating that if `B1` is true, and `B2` is true, ..., and `BN` is true, then it must be that `H` is also true. (Quantifiers are implicit in a Horn clause, though we will not discuss the details here.)

The individual elements of a Horn clause, such as `H` or `B2` above, are called *terms*. A term may be a variable, an *atom* in the form of a symbol, or a compound term, such as a *predicate* applied to some arguments which are themselves terms.

A set of Horn clauses establishes *relations* among data, which we can then use to query whether a relation holds or what pieces of data satisfy a particular relation.

As a concrete example, consider the following clauses that represent familial relationships:

```
parent(P, C) :- mother(P, C).                % rule 1
parent(P, C) :- father(P, C).                % rule 2
sibling(A, B) :- parent(P, A), parent(P, B).  % rule 3
```

Here, we have stated three *rules*. The first establishes that if `P` is the mother of `C`, then `P` is also a parent of `C`. The second states that if `P` is the father of `C`, then `P` is also a parent of `C`. The last rule states that if `P` is a parent of `A`, and `P` is also a parent of `B`, then `A` and `B` are siblings.

We can state some specific relationships as *facts*, which are Horn clauses without a body and thus are unconditionally true:

```
mother(molly, bill).      % fact 1
mother(molly, charlie).   % fact 2
```

We can give the logic interpreter a query of the form `sibling(bill, S)`. The interpreter will then attempt to solve this query using a process known as *resolution*, which applies rules to existing information. Part of this process is *unification*, which connects terms that match. One possible resolution sequence for the query above is:

```
   sibling(bill, S)
-> parent(P, bill), parent(P, S)                (rule 3)
-> mother(P, bill), parent(P, S)                (rule 1)
-> mother(molly, bill), parent(molly, S)        (fact 1)
-> mother(molly, bill), mother(molly, S)        (rule 1)
-> mother(molly, bill), mother(molly, charlie)  (fact 2)
```

The end result in this sequence would be that `S = charlie`.

In the process above, the third step unifies the term `mother(P, bill)` with `mother(molly, bill)`, which in turn unifies `P` with `molly`. This unification is reflected in all occurrences of `P`, resulting in the second term becoming `parent(molly, S)`. Unification is a generalized form of variable binding, except that full terms can be unified with each other rather than just binding variables to values.

In our formulation of familial relationships, however, there is nothing preventing the resolution engine from applying `mother(molly, bill)` in resolving `mother(molly, S)`, so another perfectly valid solution is that `S = bill`. We will see later how to fix this specific problem in Prolog.

## 24.1 Prolog

Before we proceed further in our exploration of logic programming, let us introduce a concrete programming language to work with. Among logic languages, Prolog is by far the most popular, and many implementations are available. For the purposes of this text, we will use a specific interpreter called SWI-Prolog, of which there is also a web-based version.

The syntax we used above is actually that of Prolog. A Prolog program consists of a set of Horn clauses that are assumed to be true. A clause is composed of a head term and zero or more body terms, and a term may be atomic, compound, or a variable. An atomic term may be an *atom*, which is either a Scheme-like symbol or a quoted string. The following are all atoms:

```
hello    =<    +    'logic programming'
```

If an atom starts with a letter, then that letter must be lowercase. Thus, `hEllo` is an atom, but `Hello` is not. Numbers, which can be integer or floating-point, are also atomic terms.

Variables are identifiers that begin with a capital letter. Thus, `Hello` is a variable, as are `A` and `X`.

Compound terms consist of a *functor*, which is itself an atom, followed by a list of one or more argument terms. The following are compound terms:

```
pair(1, 2)    wizard(harry)    writeln(hello(world))
```

A compound term is interpreted as a *predicate*, meaning that it has a truth value, when it occurs as the head term or one of the body terms of a clause, as well as when it is the goal query. Otherwise, it is generally interpreted as data, as in `hello(world)` in `writeln(hello(world))`.

While the syntax of a compound term resembles that of a function call in many imperative or functional languages, Prolog does not have functions, so a compound term is never interpreted as such.

A Horn clause with no body is a fact, since it is always true. Thus, the following are facts:

```
mother(molly, bill).
mother(molly, charlie).
```

Notice the period that signifies the end of a clause.

A Horn clause with a body is called a rule, and it consists of a head term, the reverse implication symbol (`:-`), and one or more body terms, separated by commas. The comma signifies conjunction so that the head is true when all the body terms are true. The following are rules:

```
parent(P, C) :- mother(P, C).
sibling(A, B) :- parent(P, A), parent(P, B).
```

The first rule states that if `mother(P, C)` is true, then `parent(P, C)` is also true. The second rule states that if both `parent(P, A)` and `parent(P, B)` are true, then `sibling(A, B)` is true.

A program is composed of a set of facts and rules. Once these have been established, we can query the Prolog interpreter with a a goal predicate. The interpreter will attempt to establish that the goal is true, and if it contains variables, instantiate them with terms that result in the satisfaction of the goal. If the query succeeds, the interpreter reports success, along with the terms that the variables unified with in order to establish the result. If more than one solution may exist, we can ask for the next one using a semicolon in most interpreters. If we ask for a solution and no more exist, the interpreter reports failure.

As an example, consider the query `sibling(bill, S)`. Loading a file containing the two facts and rules above will result in the interactive prompt `?-` (For now, we have elided the `parent` rule that depended on `father`, since we haven't established any `father` facts and our Prolog interpreter will report an error as a result.):

```
?- sibling(bill, S).
S = bill ;
S = charlie.
```

At the prompt, we've entered the query followed by a period to signify the end of the query. The interpreter reports `S = bill` as the first result, and we have the option of entering a semicolon to search for another or a period to end the query. We enter a semicolon, and the interpreter finds and reports `S = charlie`, as well as a period to indicate its certainty that no more solutions exist.

The actual order in which Prolog searches for a result is deterministic, as we will see shortly. Thus, the query will always find `S = bill` as its first result and `S = charlie` as its second.

### 24.1.1 Lists

Compound terms, which can relate multiple individual terms, allow us to represent data structures. For example, we can use the compound term `pair(First, Second)` to represent a pair composed of `First` and `Second`. The term will not appear on its own as a head or body term, so it will be treated as data. We can then define relations for lists as follows:

```
cons(First, Second, pair(First, Second)).
car(pair(First, _), First).
cdr(pair(_, Second), Second).
is_null(nil).
```

In the clauses above, an underscore represents an anonymous variable. Many Prolog implementations will raise a warning if a variable is used only once in a clause – such a variable is called a *singleton*, and it can be introduced inadvertently due to a typo, as in the following:

```
cons(First, Second, pair(Frist, Second)).
```

Here, we misspelled `First` as `Frist`, so that both are singleton variables. The warning from the Prolog interpreter directs our attention to this, so that we can fix it. If, on the other hand, we do intend a variable to be used only once, we can start it with an underscore to inform the implementation of that intent. We can also use a lone underscore, and each occurrence of a solitary underscore is considered a separate, anonymous variable.

We've set `nil` as our representation for an empty list. We can then make queries on lists as follows:

```
?- cons(1, nil, X).
X = pair(1, nil).

?- car(pair(1, pair(2, nil)), X).
X = 1.

?- cdr(pair(1, pair(2, nil)), X).
X = pair(2, nil).

?- cdr(pair(1, pair(2, nil)), X), car(X, Y), cdr(X, Z).
X = pair(2, nil),
Y = 2,
Z = nil.

?- is_null(nil).
true.

?- is_null(pair(1, pair(2, nil))).
false.
```

In the fourth example, we've used conjunction to obtain the cdr of the original list, as well as the car and the cdr of the result.

As in Scheme, lists are a fundamental data structure in Prolog, so Prolog provides its own syntax for lists. A list can be specified by placing elements in square brackets, separated by commas:

```
[]
[1, a]
[b, 3, foo(bar)]
```

A list can be decomposed into a number of items followed by a rest, much like the period in Scheme, using a pipe:

```
?- writeln([1, 2 | [3, 4]]).   % similar to (1 2 . (3 4)) in Scheme
[1,2,3,4]
true.
```

We can use this syntax to write rules on lists. For example, a contains predicate is as follows:

```
contains([Item|_Rest], Item).
contains([_First|Rest], Item) :-
    contains(Rest, Item).
```

The first clause asserts that a list whose first element is `Item` contains `Item`. The second clause states that a list contains `Item` if the remaining list, excluding the first item, contains `Item`. Thus:

```
?- contains([], a).
false.

?- contains([a], a).
true .

?- contains([b, c, a, d], a).
true .
```

The built-in `member` predicate works similarly to our definition of `contains`, except that it takes the arguments in reverse order:

```
?- member(a, []).
false.

?- member(a, [a]).
true.
```

### 24.1.2 Arithmetic

Prolog provides numbers, as well as comparison predicates on numbers. For convenience, these predicates may be written in infix order:

```
?- 3 =< 4.    % less than or equal
true.

?- 4 =< 3.
false.

?- 3 =:= 3.   % equal (for arithmetic)
true.

?- 3 =\= 3.   % not equal (for arithmetic)
false.
```

Prolog also provides arithmetic operators, but they merely represent compound terms. Thus, `3 + 4` is another means of writing the compound term `+(3, 4)`. If we attempt to unify this with 7 using the explicit unification operator =, it will fail:

```
?- 7 = 3 + 4.
false.
```

Similarly, if we attempt to unify a variable with an arithmetic expression, it will be unified with the compound term itself:

```
?- X = 3 + 4.
X = 3+4.
```

Comparison operators, however, do evaluate the arithmetic expressions in their operands::

```
?- 7 =:= 3 + 4.
true.

?- 2 + 5 =:= 3 + 4.
true.

?- 4 < 3 + 2.
true.
```

In order for the operands to be evaluated, variables in the operands must be *instantiated* with numeric values. A comparison cannot be applied to an uninstantiated variable:

```
?- X =:= 3 + 4.
ERROR: Arguments are not sufficiently instantiated
```

Instead, the `is` operator is defined to unify its first argument with the arithmetic result of its second argument, allowing the first argument to be an uninstantiated variable:

```
?- 7 is 3 + 4.
true.

?- X is 3 + 4.
X = 7.

?- X is 3 + 4, X =:= 7.
X = 7.

?- X is 3 + 4, X = 7.
X = 7.
```

In the third example, `X` is unified with 7, the result of adding 3 and 4. Since `X` is now instantiated with 7, it can be compared to 7. In the fourth example, `X` is 7 so it unifies with the number 7.

We can use this to define a length predicate on our list representation above:

```
len(nil, 0).
len(pair(_First, Second), Length) :-
    len(Second, SecondLength), Length is SecondLength + 1.
```

Here, `Length` is unified with the arithmetic result of adding 1 to `SecondLength`. This must occur after the recursive application of `len`, so that `SecondLength` is sufficiently instantiated to be able to perform arithmetic on it. Then:

```
?- len(nil, X).
X = 0.

?- len(pair(1, pair(b, nil)), X).
X = 2.
```

### 24.1.3 Side Effects

Prolog provides several predicates that perform input and output. We've already used the `writeln` predicate, which writes a term to standard out and then writes a newline. The `write` predicate also writes a term to standard out, but without a trailing newline:

```
?- X = 3, write('The value of X is: '), writeln(X).
The value of X is: 3
X = 3.
```

We will not discuss the remaining I/O routines here.

## 24.2 Unification and Search

The core computational engine in Prolog revolves around unification and search. The search procedure takes a set of goal terms and looks for a clause that has a head that can unify with one of the terms. The unification process can recursively unify subterms, which may instantiate or unify variables. If the current term unifies with the head of a clause, then the body terms, with variables suitably instantiated, are added to the set of goal terms. The search process succeeds when no more goal terms remain.

This process of starting from goal terms and working backwards, replacing heads with bodies, is called *backward chaining*. A logic interpreter may use *forward chaining* instead, which starts from facts and works forward to derive the goal. However, Prolog is defined to use backward chaining.

The unification rules for two terms in Prolog are as follows:

1. An atomic term only unifies with itself.
2. An uninstantiated variable unifies with any term. If the other term is not a variable, then the variable is instantiated with the value of the other term. If the other term is another variable, then the two variables are bound together such that if one of them is later instantiated with a value, then so is the other.
3. A compound term unifies with another compound term that has the same functor and number of arguments, and only if the arguments of the two compound terms also unify.

As stated by the first rule, the atomic term 1 only unifies with 1, and the term `abc` only unifies with `abc`.

The second rule states that a variable `X` unifies with a non-variable by instantiating it to the given value. This essentially means that all occurrences of the variable are replaced with the given value. Thus `X` unifies with 3 by instantiating `X` with 3, `Y` unifies with `foo(1, 3)` by instantiating it with `foo(1, 3)`, and `Z` unifies with `foo(A, B)` by instantiating it with `foo(A, B)`.

A variable unifies with another variable by binding them together. Thus, if `X` unifies with `Y`, and if `Y` is later instantiated with 3, then `X` is also instantiated with 3.

The last rule states that a compound term such as `foo(1, X)` unifies with `foo(Y, 3)` by recursively unifying the arguments, such that `Y` is instantiated with 1 and `X` with 3.

Care must be taken in the search process to treat variables that appear in independent contexts as independent, even if they have the same name. Thus, given the clause:

```
foo(X, Y) :- bar(Y, X).
```

and the goal `foo(3, X)`, the variable `X` should be treated as distinct in the contexts of the goal and the clause. One way to accomplish this is renaming before applying a rule, analogous to $\alpha$-reduction in $\lambda$-calculus:

```
foo(X1, Y1) :- bar(Y1, X1).
```

Thus, unifying the goal `foo(3, X)` with the head `foo(X1, Y1)` produces `X1 = 3` and `Y1 = X`, resulting in the subsequent goal `bar(X, 3)`.

### 24.2.1 Search Order and Backtracking

In pure logic programming, the order in which clauses are applied and body terms are resolved doesn't matter as long as the search process terminates. However, since Prolog has side effects and non-pure operations, it specifies a well-defined order for both. In particular, clauses for a predicate are attempted to be applied in program order, and terms in a conjunction are resolved from left to right. This provides the programmer with some control over how computation proceeds, which can be used to improve efficiency as well as sequence side effects.

A search process that goes down one path may end up in a dead end, where no clauses can be applied to a goal term. This should not immediately result in failure, since changing a previous decision made by the search may lead to a solution. Thus, the search process performs *backtracking* on failure, or even on success if a user requests more solutions. This reverts the search process to the last choice point with remaining options, at which a different choice is made about which clause to apply.

As an example, consider the following clauses:

```
sibling(A, B) :- mother(P, A), mother(P, B).

mother(lily, harry).
mother(molly, bill).
mother(molly, charlie).
```

Suppose the goal is `sibling(S, bill)`. Then the search tree is as in Figure 24.1.

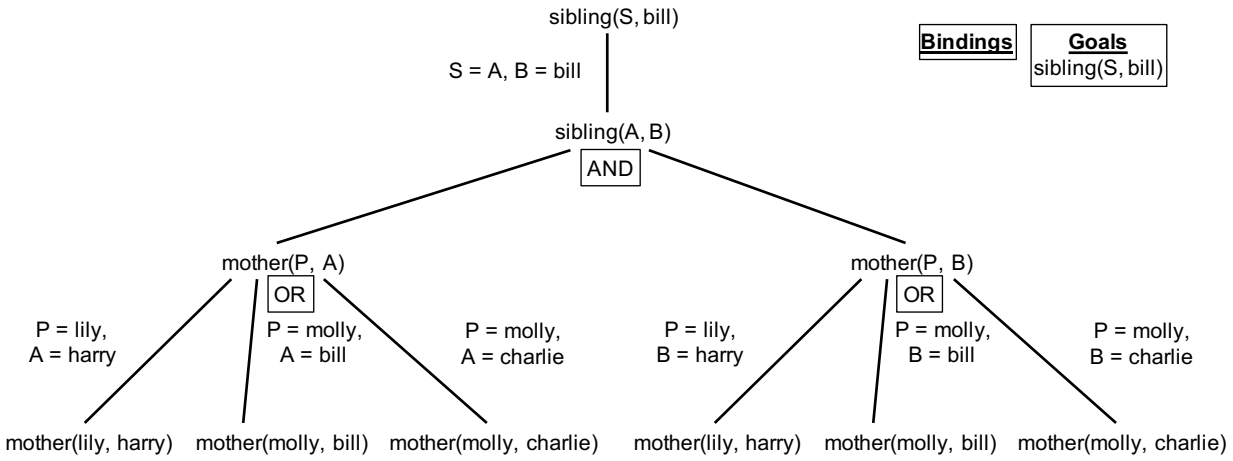



Figure 24.1: The search tree for the query `sibling(S, bill)`.

The search will first unify `sibling(S, bill)` with the goal term `sibling(A, B)`, binding `S` and `A` together and instantiating `B` with `bill`. We use the notation `S = A` to denote that `S` and `A` are bound together, as in Figure 24.2.

Prolog will then add the body terms to its set of goals, so that `mother(P, A)` and `mother(P, B)` need to be satisfied. It then searches for a solution to `mother(P, A)`, under an environment in which `S` and `A` are bound together and `B` is instantiated with `bill`. There are several clauses that can be applied to satisfy `mother(P, A)`, introducing a choice point. Prolog attempts to apply clauses in program order, so the first choice the search engine will make is to unify `mother(P, A)` with `mother(lily, harry)`, as shown in Figure 24.3.

This instantiates `A`, and therefore `S` since `A` and `S` are bound together, with `harry` and `P` with `lily`. Then only the goal term `mother(P, B)` remains, and since multiple clauses can be applied, another choice point is introduced. The first

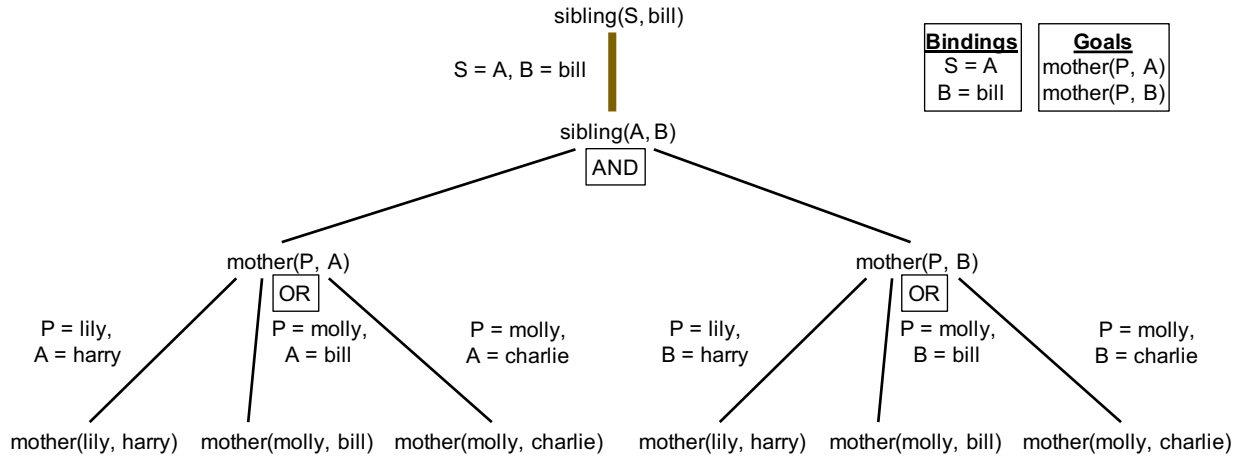



Figure 24.2: Unifying `sibling(S, bill)` with `sibling(A, B)` binds `S` and `A` together and instantiates `B` with `bill`. The body terms `mother(P, A)` and `mother(P, B)` are added to the goal set.

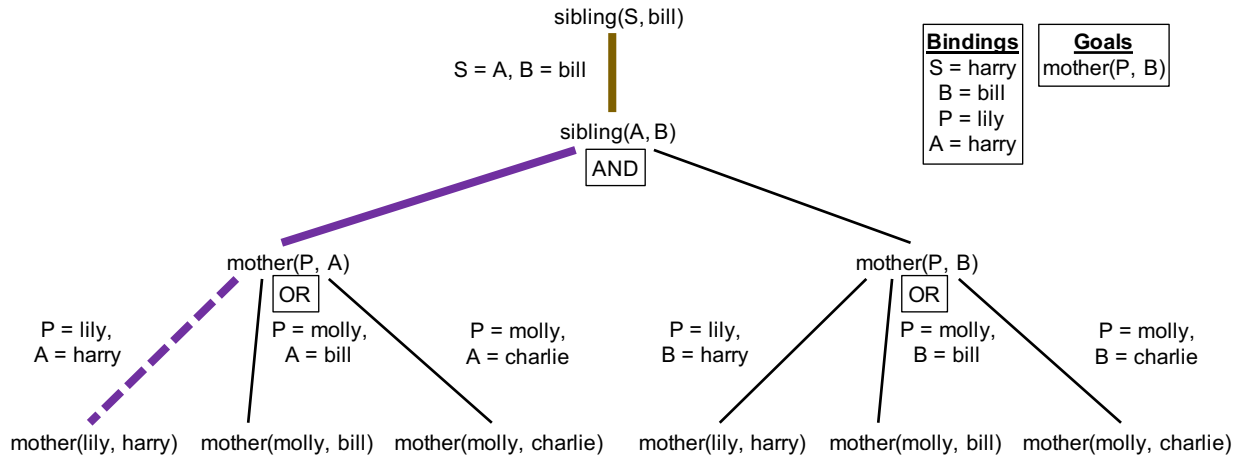



Figure 24.3: Unifying `mother(P, A)` with `mother(lily, harry)` instantiates `P` with `lily` and `A` and `S` with `harry`. The goal term `mother(P, A)` is satisfied, so it is removed from the goal set.

choice is to unify `mother(P, B)` with `mother(lily, harry)`, as demonstrated in Figure 24.4.

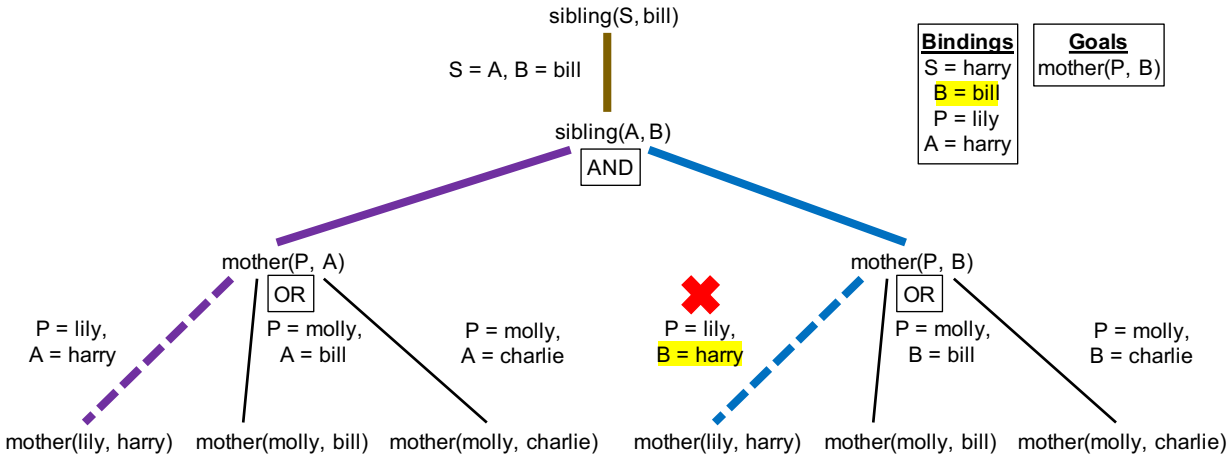



Figure 24.4: Unification of `mother(P, B)` with `mother(lily, harry)` fails, since `B` is instantiated with `bill`, which does not unify with `harry`.

This unification fails, since it requires `B` to be unified with `harry`. However, `B` is currently instantiated with the atom `bill`, and two atoms only unify if they are the same, so that `bill` and `harry` do not unify. The unification failure causes the search engine to backtrack to the previous choice point, so that it instead attempts to unify `mother(P, B)` with `mother(molly, bill)`. Figure 24.5 illustrates this.

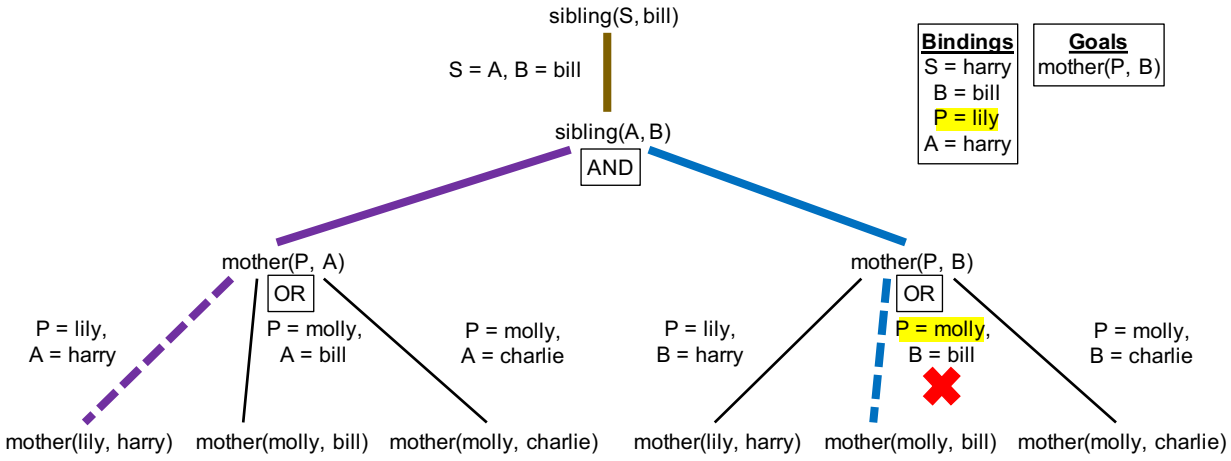



Figure 24.5: Unification of `mother(P, B)` with `mother(molly, bill)` fails, since `P` is instantiated with `lily`, which does not unify with `molly`.

This unification also fails, since it requires `P`, currently instantiated with `lily`, to be unified with `molly`. The search backtracks once again, trying to unify `mother(P, B)` with `mother(molly, charlie)`, as shown in Figure 24.6.

Again, the unification fails, so the search backtracks. At this point, it has exhausted all the choices for `mother(P, B)`, so it backtracks further to the preceding choice point. Now, it makes the choice of unifying `mother(P, A)` with `mother(molly, bill)`, as illustrated in Figure 24.7.

This instantiates `P` with `molly` and `A` and `S` with `bill`. Then, as shown in Figure 24.8, the search attempts to find a solution for `mother(P, B)`, first attempting to unify it with `mother(lily, harry)`.

This fails, since `P = molly` cannot unify with `lily`. Thus, the search backtracks to the previous choice point, attempt-

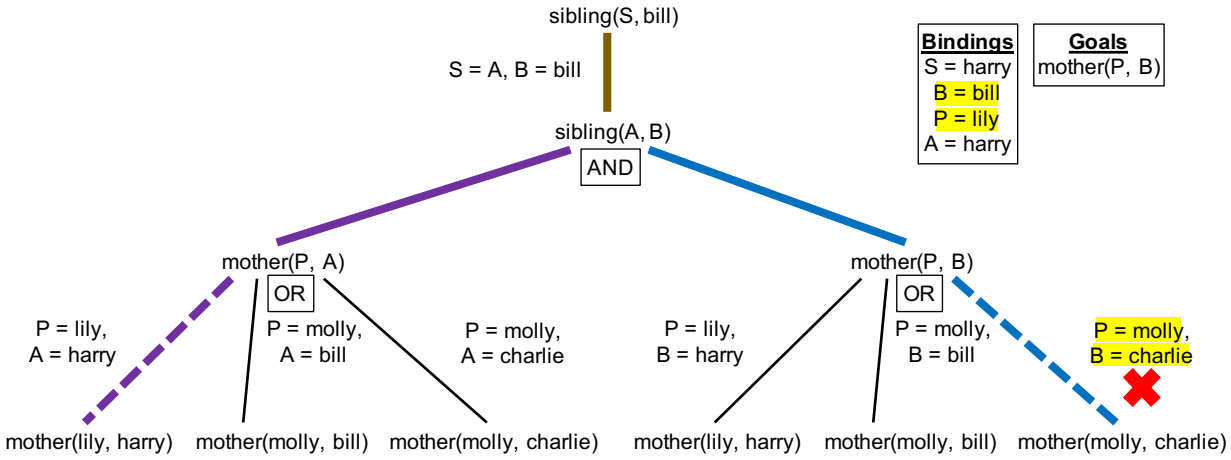



Figure 24.6: Unification of `mother(P, B)` with `mother(molly, charlie)` fails: `P` is instantiated with `lily`, which does not unify with `molly`, and `B` is instantiated with `bill`, which does not unify with `charlie`.

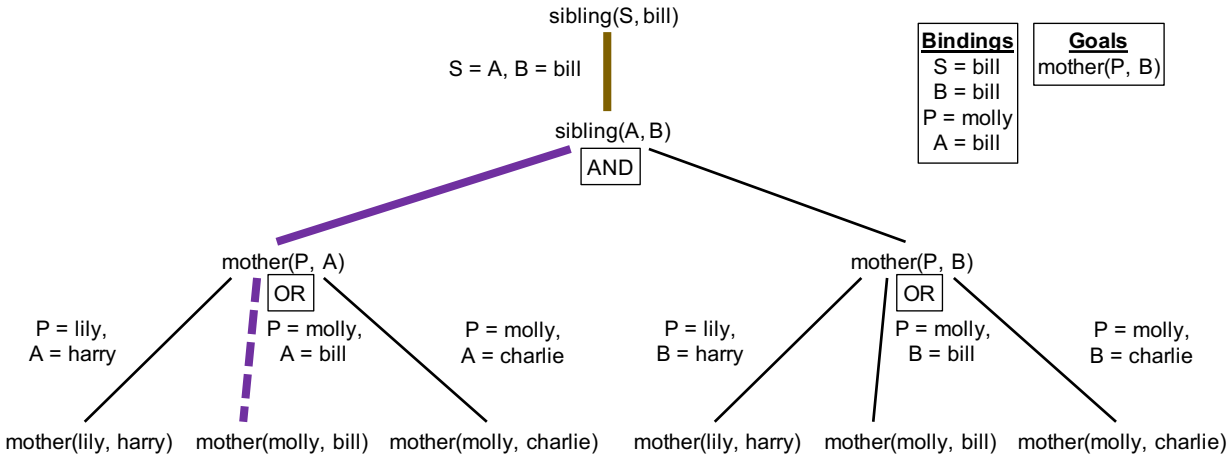



Figure 24.7: The search backtracks and unifies `mother(P, A)` with `mother(molly, bill)` instead. This instantiates `P` with `molly` and `A` and `S` with `bill`.

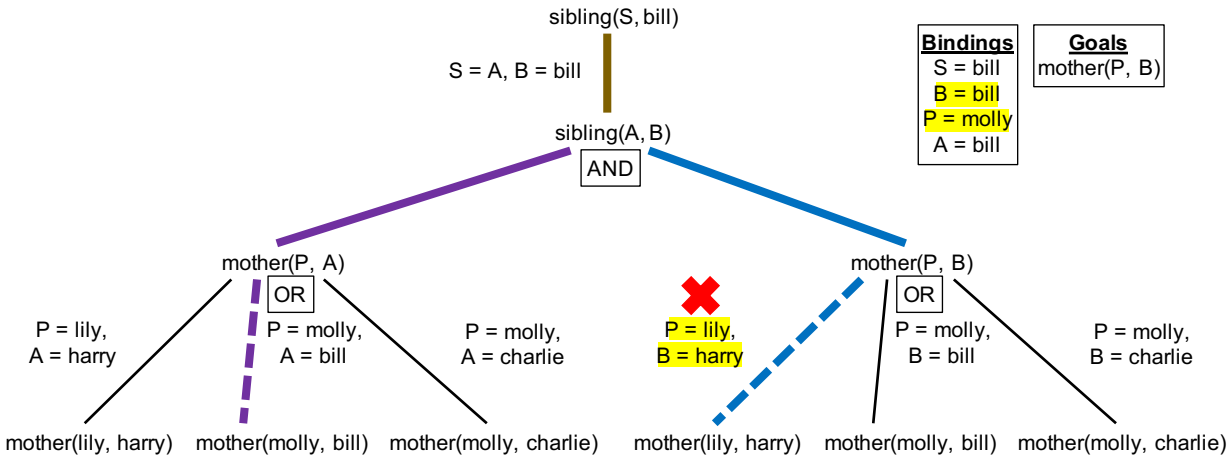



Figure 24.8: Unification of `mother(P, B)` with `mother(lily, harry)` fails: `P` is instantiated with `molly`, which does not unify with `lily`, and `B` is instantiated with `bill`, which does not unify with `harry`.

ing to unify `mother(P, B)` with `mother(molly, bill)`. Figure 24.9 demonstrates this.

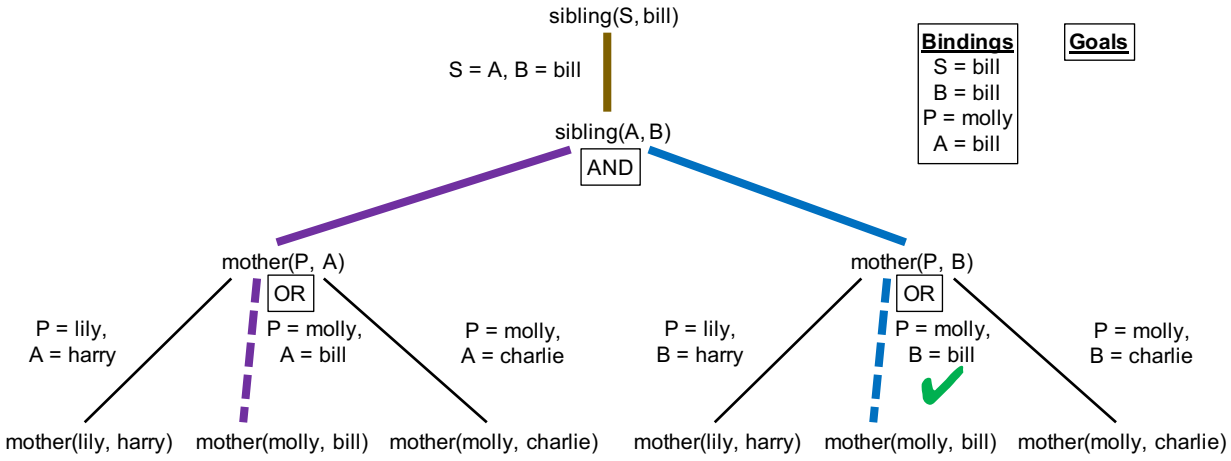



Figure 24.9: Unification of `mother(P, B)` with `mother(molly, bill)` succeeds. No goal terms remain, so the query is satisfied with `S = bill`.

This succeeds. No more goal terms remain, so the query succeeds with a solution of `S = bill`.

We can proceed to ask the interpreter for more solutions, which continues the search at the last choice point. One more choice remains, to unify `mother(P, B)` with `mother(molly, charlie)`, as shown in Figure 24.10.

However, this fails, so the search backtracks to the preceding choice point, unifying `mother(P, A)` with `mother(molly, charlie)`, as illustrated in Figure 24.11.

Continuing the search with `P` instantiated with `molly` and `A` and `S` with `charlie` reaches another choice point for `mother(P, B)`. As Figure 24.12 demonstrates, the first choice fails.

However, the second choice of unifying `mother(P, B)` with `mother(molly, bill)` succeeds, as shown in Figure 24.13.

Thus, we have another solution of `S = charlie`. We can then ask for another solution, resulting in the search engine trying the last choice for `mother(P, B)`, as demonstrated in Figure 24.14.

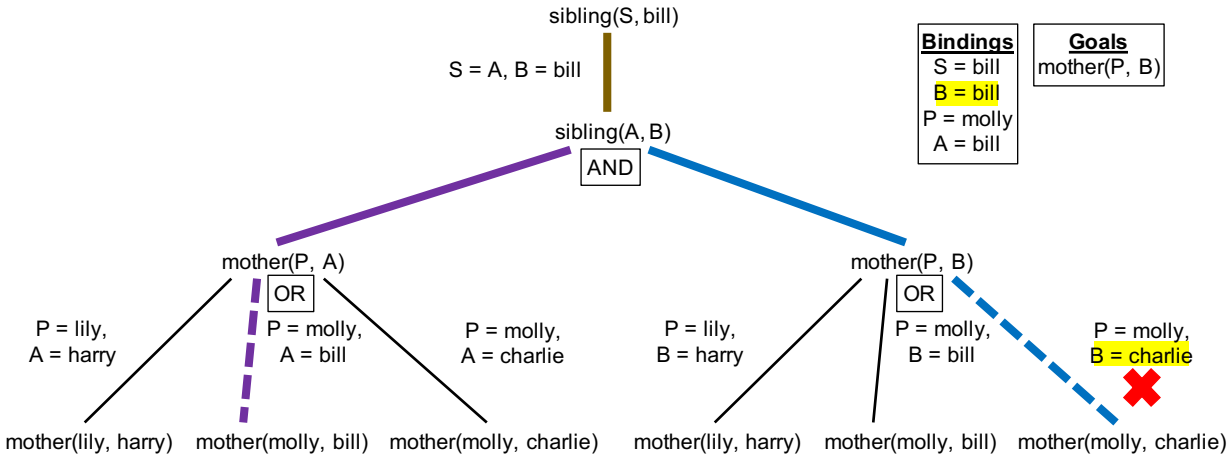



Figure 24.10: Continuing to search for more solutions, unification of `mother(P, B)` with `mother(molly, charlie)` fails, since `B` is instantiated with `bill`, which does not unify with `charlie`.

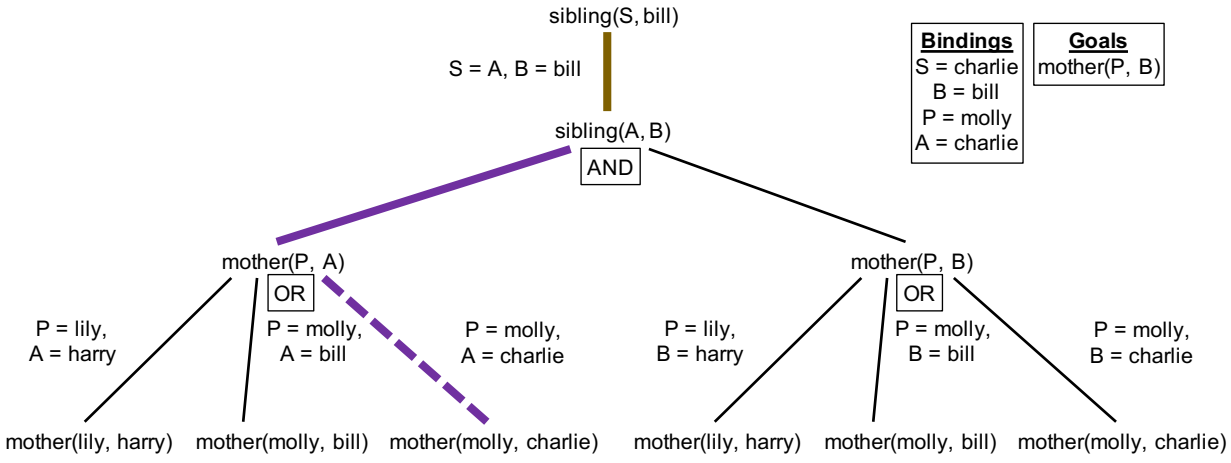



Figure 24.11: The search backtracks and unifies `mother(P, A)` with `mother(molly, charlie)` instead. This instantiates `P` with `molly` and `A` and `S` with `charlie`.

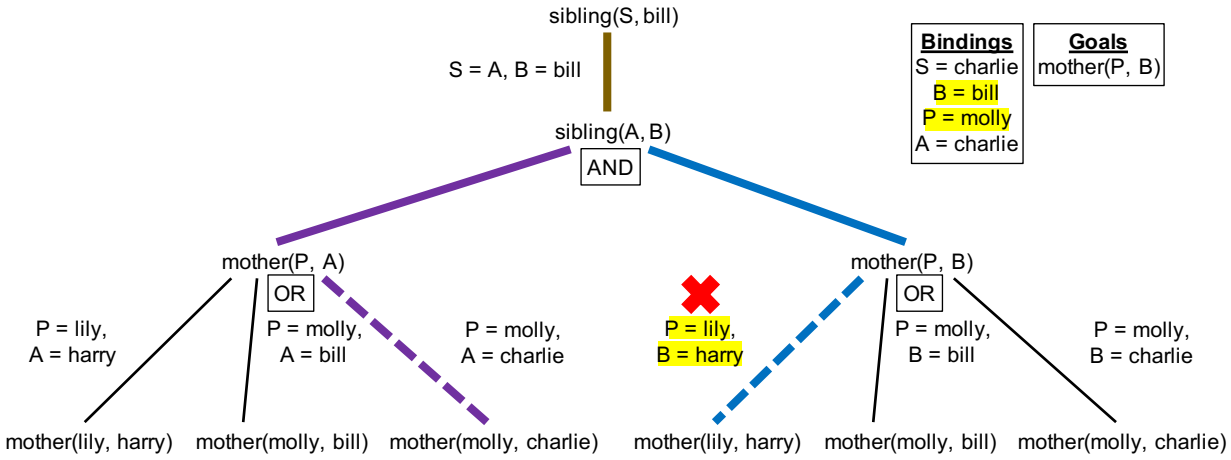



Figure 24.12: Unification of `mother(P, B)` with `mother(lily, harry)` fails: `P` is instantiated with `molly`, which does not unify with `lily`, and `B` is instantiated with `bill`, which does not unify with `harry`.

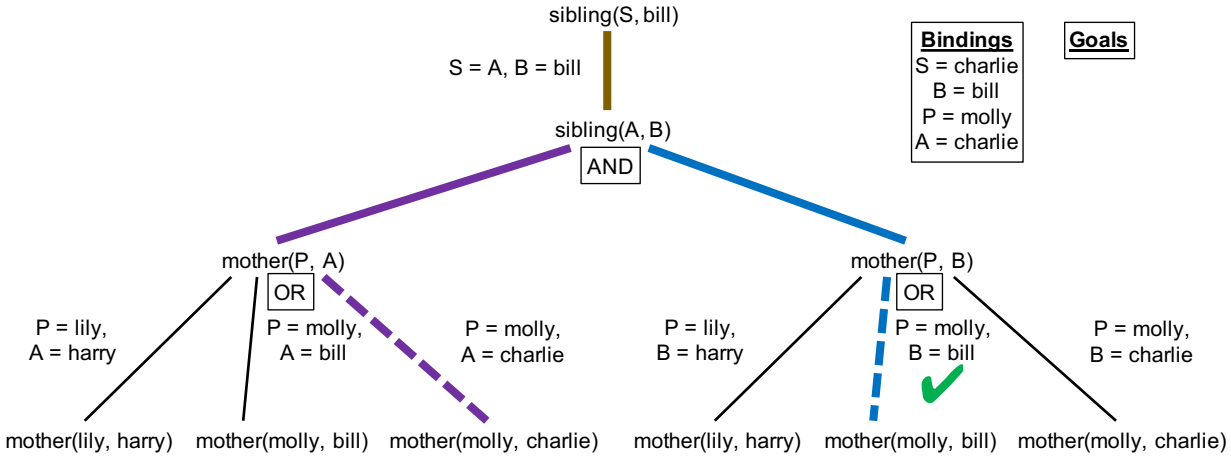



Figure 24.13: Unification of `mother(P, B)` with `mother(molly, bill)` succeeds. No goal terms remain, so the query is satisfied with `S = charlie`.

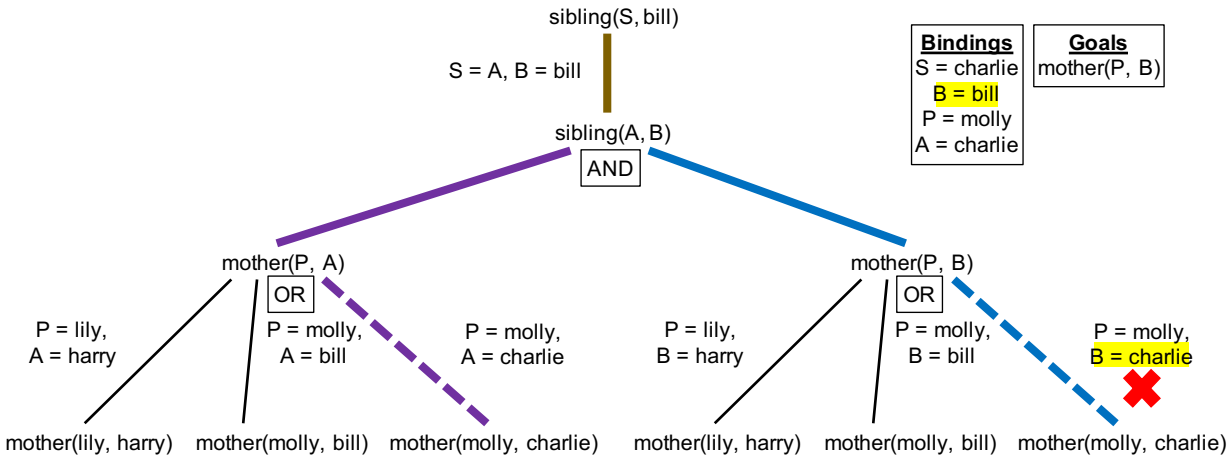



Figure 24.14: Continuing to search for more solutions, unification of `mother(P, B)` with `mother(molly, charlie)` fails, since `B` is instantiated with `bill`, which does not unify with `charlie`. No choice points remain, so the search terminates.

This choice fails. At this point, all choice points have been exhausted, so the interpreter reports that no more solutions can be found. The full interpreter interaction is as follows, reflecting the search process above:

```
?- sibling(S, bill).
S = bill ;
S = charlie ;
false.
```

## 24.3 The Cut Operator

By default, Prolog considers each possible alternative in turn when it reaches a choice point. However, Prolog provides the *cut operator*, written as `!`, to eliminate choice points associated with the current predicate. For example, recall the `contains` predicate:

```
contains([Item|_Rest], Item).
contains([_First|Rest], Item) :-
    contains(Rest, Item).
```

A query such as `contains([1, 2, 3, 4], 2)` introduces a choice point as to which clause to unify with the goal. The first choice fails, since `contains([1, 2, 3, 4], 2)` cannot unify with `contains([Item|_Rest], Item)`. However, the second choice succeeds, so that we have a new goal term of `contains([2, 3, 4], 2)`. Here another choice point occurs, and the first choice succeeds, with `Item` instantiated with 2 and `_Rest` instantiated with `[3, 4]`. Since no goal terms remain, the query as a whole succeeds. However, the interpreter still has an unexplored choice available, so it will report that more solutions may exist, requiring us to manually either continue the query with a semicolon or end it with a dot:

```
?- contains([1, 2, 3, 4], 2).
true ;
false.
```

Instead, we can add the cut operator to tell the interpreter to stop searching for alternatives once it has found a solution for `contains([1, 2, 3, 4], 2)`:

```
?- contains([1, 2, 3, 4], 2), !.
true.
```

We can similarly rewrite `contains` to eliminate choice points upon success:

```
contains([Item|_Rest], Item) :- !.
contains([_First|Rest], Item) :-
    contains(Rest, Item).
```

Here, as soon as a goal term unifies with `contains([Item|_Rest], Item)`, the choice point of which `contains` clause to unify with that goal term is eliminated. Thus, only one solution is found:

```
?- contains([1, 2, 3, 4], 2).
true.
```

On the other hand, the fact that the cut operator prevents other choices from being considered can result in queries that previously succeeded to now fail:

```
?- contains([1, 2, 3, 4], X), X = 3.
false.
```

Here, the first goal term succeeds by instantiating `X` with 1, and the cut operator prevents other choices for `X` from being considered. Then `X`, now instantiated as 1, fails to unify with 3 in the second goal term.

Given the potential negative consequences of eliminating choice points, using the cut operator is often considered bad practice, so that it should be avoided in most cases. In this text, we only use the cut operator as part of a query, not as part of a rule.

## 24.4 Negation

The search above for `sibling(S, bill)` produced the undesirable result `S = bill`. In order to eliminate results from consideration, Prolog provides a limited form of negation. For instance, we can rewrite the `sibling` rule as:

```
sibling(A, B) :- A \= B, mother(P, A), mother(P, B).
```

This states that `A` must not be unifiable with `B`. Unfortunately, our query will now fail completely:

```
?- sibling(S, bill).
false.
```

This is because when the body term `A \= B` is reached, `A` is uninstantiated while `B` is instantiated with `bill`. The unification rules above allow an uninstantiated variable to unify with anything: `A` can unify with `B` by instantiating `A` with `bill`. Since `A` is unifiable with `B`, the goal term `A \= B` fails.

On the other hand, if we write the rule as follows:

```
sibling(A, B) :- mother(P, A), mother(P, B), A \= B.
```

then our query succeeds:

```
?- sibling(S, bill).
S = charlie .
```

This is because `A` and `B` are instantiated as atoms by the time they get to the last term, and we can assert that two atoms not unify.

Prolog also provides an explicit negation predicate, `\+`. We can therefore query whether `harry` and `bill` are not siblings:

```
?- \+(sibling(harry, bill)).
true.
```

Unfortunately, we cannot obtain a more general result from the search engine, such as asking it to find someone who is not a sibling of `bill`:

```
?- \+(sibling(S, bill)).
false.
```

This is because negation in Prolog is handled by attempting to prove the term being negated, and only if the proof fails is the negation true. However, the query `sibling(S, bill)` does indeed succeed with `S = charlie`, so negation results in false.

Thus, while Prolog does provide negation, it is of limited use. This is not a deficiency in Prolog itself, but rather follows from the limits of the logic-programming paradigm as a whole, which cannot provide the full expressiveness of first-order predicate calculus.

## 24.5 Examples

We conclude with some more interesting examples expressed in the logic paradigm.

- Suppose we wish to find a seven digit number such that the first digit is the count of zeroes in the digits of the number, the second digit is the count of ones, and so on. Using Prolog, we can express this computation as follows. We will represent our results as a list of digits. First, we define a predicate to count the occurrences of a particular numerical value in a list:

  ```
  count(_Item, [], 0).
  count(Item, [Item|Rest], Count) :-
      count(Item, Rest, RestCount),
      Count is RestCount + 1.
  count(Item, [Other|Rest], Count) :-
      Item =\= Other,
      count(Item, Rest, Count).
  ```

  The first rule states that an arbitrary item occurs zero times in an empty list. The second states that if a value is the first item in a list, then the number times it occurs in the list is one more than the number of times it appears in the rest of the list. The last rule states that if a value is not equal to the first item, then its number of occurrences is that same as the number of times it appears in the rest of the lest.

  Next, we define facts to restrict the values of a digit:

  ```
  is_digit(0).
  is_digit(1).
  is_digit(2).
  is_digit(3).
  is_digit(4).
  is_digit(5).
  is_digit(6).
  ```

Alternatively, we can define the `is_digit` predicate using `member`:

```
is_digit(Digit) :-
    member(Digit, [0, 1, 2, 3, 4, 5, 6]).
```

Finally, we define a predicate to compute our result:

```
digits(List) :-
    List = [Digit0, Digit1, Digit2, Digit3, Digit4, Digit5, Digit6],
    is_digit(Digit0),
    is_digit(Digit1),
    is_digit(Digit2),
    is_digit(Digit3),
    is_digit(Digit4),
    is_digit(Digit5),
    is_digit(Digit6),
    count(0, List, Digit0),
    count(1, List, Digit1),
    count(2, List, Digit2),
    count(3, List, Digit3),
    count(4, List, Digit4),
    count(5, List, Digit5),
    count(6, List, Digit6).
```

We start by unifying the argument `List` with a list of seven items. We then specify that each item must be a digit. Finally, we require that the the first item be the count of zeroes in the list, the second the count of ones, and so on.

Entering our query, we get the sole result:

```
?- digits(List).
List = [3, 2, 1, 1, 0, 0, 0] ;
false.
```

We can proceed to write a predicate that relates a list of digits to an actual number:

```
digits_number([], 0).
digits_number([First|Rest], Number) :-
    digits_number(Rest, RestNumber),
    length(Rest, RestLength),
    Number is First * 10 ^ RestLength + RestNumber.
```

An empty list is related to zero. Otherwise, we compute the number represented by the list excluding its first item, as well of the length of that list. Then the number representing the total list is the number of the smaller list plus the multiple of the power of 10 represented by the first digit. Then:

```
?- digits(List), digits_number(List, Number), !.
List = [3, 2, 1, 1, 0, 0, 0],
Number = 3211000.
```

- The *Tower of Hanoi* is a classic puzzle that consists of three rods and a set of $N$ discs of different sizes that slide onto a rod. The puzzle starts with discs in ascending order from top to bottom on a single rod, and the goal is to move the entire stack to another rod by moving one disc at a time. It is prohibited to place a larger disc on top of a smaller one. The solution is to recursively move the $N - 1$ smaller discs to the third rod, move the remaining, largest disc to the second rod, and then to recursively move the other $N - 1$ discs to the second rod.

We can express this computation in Prolog as follows, using the `write` and `writeln` predicates to print a move to standard output:

```
move(Disc, Source, Target) :-
    write('Move disc '), write(Disc), write(' from '),
    write(Source), write(' to '), writeln(Target).

hanoi(1, Source, Target, _Temporary) :-
    move(1, Source, Target).
hanoi(NumDiscs, Source, Target, Temporary) :-
    Previous is NumDiscs - 1,
    hanoi(Previous, Source, Temporary, Target),
    move(NumDiscs, Source, Target),
    hanoi(Previous, Temporary, Target, Source).
```

The `move` predicate, given a disc and source and target rods, merely writes out the move to standard output. The `hanoi` predicate relates a number of discs and three rods, a source rod, a target rod, and a temporary rod. The base case is when there is one disc, and that disc can be moved directly from source to target. The second `hanoi` rule is the recursive case, which requires is recursively move all but the largest disc to the temporary rod, move the largest disc to the target rod, and then move the remaining discs from the temporary to the target rod. Since Prolog solves the body terms in order, the moves will occur in the right order.

The follows is the result of a query with $N = 4$:

```
?- hanoi(4, a, b, c).
Move disc 1 from a to c
Move disc 2 from a to b
Move disc 1 from c to b
Move disc 3 from a to c
Move disc 1 from b to a
Move disc 2 from b to c
Move disc 1 from a to c
Move disc 4 from a to b
Move disc 1 from c to b
Move disc 2 from c to a
Move disc 1 from b to a
Move disc 3 from c to b
Move disc 1 from a to c
Move disc 2 from a to b
Move disc 1 from c to b
true .
```

- The *quicksort* algorithm sorts a list by choosing a pivot, often the first item, partitioning the remaining list into elements that are less than and greater than or equal to the pivot, recursively sorting the partitions, and then appending them. The following Prolog code expresses this:

```
quicksort([], []).
quicksort([Pivot|Rest], Sorted) :-
    partition(Pivot, Rest, Smaller, GreaterOrEqual),
    quicksort(Smaller, SortedSmaller),
    quicksort(GreaterOrEqual, SortedGreaterOrEqual),
    append(SortedSmaller, [Pivot|SortedGreaterOrEqual], Sorted).
```

The first item is chosen as the pivot, and the remaining items are then partitioned into the smaller items and those that are greater than or equal to the pivot. The two smaller lists are recursively sorted, and then the results are

appended, with the pivot placed in front of the items that are greater than or equal to it, to produce the sorted result.

The `partition` predicate is as follows:

```
partition(_Pivot, [], [], []).
partition(Pivot, [Item|Rest], [Item|Smaller], GreaterOrEqual) :-
    Item < Pivot,
    partition(Pivot, Rest, Smaller, GreaterOrEqual).
partition(Pivot, [Item|Rest], Smaller, [Item|GreaterOrEqual]) :-
    Item >= Pivot,
    partition(Pivot, Rest, Smaller, GreaterOrEqual).
```

The first item in the list is either less than the pivot or greater than or equal to it. In the first case, the item should be the first one in the smaller partition, and the rest of the list is partitioned to produce the rest of the smaller and greater-than-or-equal partitions. In the second case, the item should be the first one in the greater-than-or-equal partition, and recursion handles the rest of the list.

Entering a query for a specific list produces:

```
?- quicksort([4, 8, 5, 3, 1, 2, 6, 9, 7], X).
X = [1, 2, 3, 4, 5, 6, 7, 8, 9] .
```

- The *sieve of Eratosthenes* is an algorithm for computing prime numbers up to some limit $N$. We start by constructing a list of integers in order from 2 to $N$. Then we repeat the following process, until no numbers remain:

  1. The first item in the list is prime.
  2. Filter out all multiples of the first item from the remaining list.
  3. Go to step 1.

  We can write this algorithm in Prolog as follows. First, we construct a list with the integers from 2 to the limit $N$:

```
numbers(2, [2]).
numbers(Limit, Numbers) :-
    LimitMinusOne is Limit - 1,
    numbers(LimitMinusOne, NumbersToM),
    append(NumbersToM, [Limit], Numbers).
```

  We do so by recursively computing a list of integers from 2 to $N - 1$ and then appending $N$ to the result.

  We can then use write a predicate to filter out the multiples of a factor from a list:

```
filter_not_multiple(_Factor, [], []).
filter_not_multiple(Factor, [Number|Rest],
                    [Number|FilteredRest]) :-
    Number mod Factor =\= 0,
    filter_not_multiple(Factor, Rest, FilteredRest).
filter_not_multiple(Factor, [Number|Rest], FilteredRest) :-
    Number mod Factor =:= 0,
    filter_not_multiple(Factor, Rest, FilteredRest).
```

  The `filter_not_multiple` predicate relates a factor and a list of numbers to a list with the multiples of the factor filtered out. The second rule retains `Number` in the resulting list if it is not a multiple of `Factor`. The third rule discards `Number` from the filtered list if it is a multiple of `Factor`.

We can proceed to define a `sieve` predicate that relates a list of numbers to the result of applying the prime-sieve algorithm to the list:

```
sieve([], []).
sieve([Number|Rest], [Number|SievedRest]) :-
    filter_not_multiple(Number, Rest, FilteredRest),
    sieve(FilteredRest, SievedRest).
```

The first number is retained in the result. All multiples of the first number are filtered out of the rest of the list. The sieve algorithm is then recursively applied to the filtered list to obtain the rest of the result list.

Finally, we write a `primes` predicate that relates an integer limit to a list of primes up to and including that limit:

```
primes(Limit, Primes) :-
    numbers(Limit, Numbers),
    sieve(Numbers, Primes).
```

This rule constructs a list of numbers from 2 up to the limit and then applies the sieve algorithm to the list. We can then use the sieve to compute prime numbers up to 100:

```
?- primes(100, P).
P = [2, 3, 5, 7, 11, 13, 17, 19, 23|...] [write]
P = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,
59, 61, 67, 71, 73, 79, 83, 89, 97] .
```

Pressing the `w` key when the solution is displayed in truncated form causes the interpreter to print out the non-truncated form in the second line above.

We can also use the built-in `numlist` predicate rather than the `numbers` predicate we wrote:

```
primes(Limit, Primes) :-
    numlist(2, Limit, Numbers),
    sieve(Numbers, Primes).
```

CHAPTER

TWENTYFIVE

# CONSTRAINTS AND DEPENDENCIES

In addition to functional and logic programming, the declarative paradigm includes programs that express constraints among variables and constants as well those that describe dependency graphs. We will look at the former in constraint logic programming and an instance of the latter in the `make` build automation tool.

## 25.1 Constraint Logic Programming

*Constraint logic programming* is an extension of logic programming to include constraints on variables. While logic programming allows a limited form of constraints, languages such as Prolog only allow arithmetic constraints to be applied to variables that have been instantiated. For example, suppose we wanted to find a number less than 1000 that is both a square and the sum of two squares. The following is an attempt to specify this in Prolog:

```
square_sum([N, X, Y, Z]) :-
    N =:= Z * Z, N =:= X * X + Y * Y,
    X > 0, Y > 0, Z > 0, X < Y, N < 1000.
```

We can attempt a query:

```
?- square_sum(S).
ERROR: =:=/2: Arguments are not sufficiently instantiated
```

Unfortunately, since `N` and `Z` are not instantiated in the comparison `N =:= Z`, we get an error.

On the other hand, using the CLP(FD) library for Prolog, which allows constraint logic programming over finite domains, we can specify the solution as follows:

```
:- use_module(library(clpfd)).  % load the clpfd library

square_sum_c([N, X, Y, Z]) :-
    N #= Z * Z, N #= X * X + Y * Y,
    X #> 0, Y #> 0, Z #> 0, X #< Y, N #< 1000,
    label([N, X, Y, Z]).
```

The first clause loads the library for use. We can then specify arithmetic constraints using operators that begin with a pound symbol. For instance, the `#=` operator constrains the two arguments to be equal, while the `#<` operator constraints the first argument to be smaller than the second. Finally, the `label` predicate forces the solver to *ground* the given variables, computing actual values for them rather than specifying their results as constraints. Entering a query, we can now obtain all solutions:

```
?- square_sum_c(S).
S = [25, 3, 4, 5] ;
```

(continues on next page)

(continued from previous page)

```
S = [100, 6, 8, 10] ;
S = [169, 5, 12, 13] ;
S = [225, 9, 12, 15] ;
S = [289, 8, 15, 17] ;
S = [400, 12, 16, 20] ;
S = [625, 7, 24, 25] ;
S = [625, 15, 20, 25] ;
S = [676, 10, 24, 26] ;
S = [841, 20, 21, 29] ;
S = [900, 18, 24, 30] ;
false.
```

### 25.1.1 Search

In constraint logic programming, resolution follows the same basic process as in plain logic programming. For a solver that uses backward chaining, a set of goal terms is maintained, and the solver searches for a clause whose head can be unified with the first goal term. If unification succeeds, then the body terms that are not constraints are added to the set of goals. Terms that are constraints are added to a separate set called the *constraint store*. When a new constraint is added to the store, in principle, the store is checked to make sure that the constraints are satisfiable, and if not, backtracking is done as in standard search. In practice, however, more limited checking is performed in order to obtain better efficiency from the solver. A solution is obtained when no more goal terms remain, and the set of constraints in the store is satisfiable.

### 25.1.2 Examples

As another example of using constraints, consider the canonical verbal arithmetic puzzle of finding a solution to the following equation:

```
    S E N D
+   M O R E
-----------
= M O N E Y
```

Requirements are that each letter be a distinct digit, and that the leading digit of a number not be zero. We can express this problem in plain Prolog as the following:

```
is_digit(Digit) :-
    numlist(0, 9, AllDigits),
    member(Digit, AllDigits).

money([S, E, N, D, M, O, R, Y]) :-
    is_digit(S), is_digit(E), is_digit(N), is_digit(D),
    is_digit(M), is_digit(O), is_digit(R), is_digit(Y),
    S \= 0, M \= 0,
    S \= E, S \= N, S \= D, S \= M, S \= O, S \= R, S \= Y,
    E \= N, E \= D, E \= M, E \= O, E \= R, E \= Y,
    N \= D, N \= M, N \= O, N \= R, N \= Y,
    D \= M, D \= O, D \= R, D \= Y,
    M \= O, M \= R, M \= Y,
    O \= R, O \= Y,
```

(continues on next page)

(continued from previous page)

```
    R \= Y,
                      1000 * S + 100 * E + 10 * N + D
                    + 1000 * M + 100 * O + 10 * R + E
    =:= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

First, we require that each variable be a digit in the range $[0, 9]$, and we further require that `S` and `M` not be zero. We then specify the pairwise uniqueness requirements. Finally, we specify that the variables must satisfy the target equation.

We can enter a query as follows:

```
?- money(S), !.
S = [9, 5, 6, 7, 1, 0, 8, 2].
```

Computing this solution takes close to a minute on the author's iMac computer, since the solver has to search a large portion of the solution space, with much backtracking.

We can simplify the implementation of `money` by using the higher-order `maplist` predicate, as well as the built-in `is_set`:

```
money(List) :-
   List = [S, E, N, D, M, O, R, Y],
   maplist(is_digit, List), S \= 0, M \= 0, is_set(List),
                    1000 * S + 100 * E + 10 * N + D
                  + 1000 * M + 100 * O + 10 * R + E
   =:= 10000 * M + 1000 * O + 100 * N + 10 * E + Y.
```

The term `maplist(is_digit, List)` applies the higher-order `maplist` predicate to map the `is_digit` predicate across the elements of `List`, and `is_set(List)` ensures that `List` has no duplicates. While this solution is shorter, it runs even slower than our original solution, taking about a minute and half on the same machine.

On the other hand, we can use CLP(FD) to specify the problem as follows:

```
:- use_module(library(clpfd)).

money_c([S, E, N, D, M, O, R, Y]) :-
    List = [S, E, N, D, M, O, R, Y],
    List ins 0 .. 9, S #\= 0, M #\= 0, all_distinct(List),
                    1000 * S + 100 * E + 10 * N + D
                  + 1000 * M + 100 * O + 10 * R + E
    #= 10000 * M + 1000 * O + 100 * N + 10 * E + Y,
    label(List).
```

The `ins` predicate is defined by CLP(FD) to constrain that the variables in the first argument each be contained in the set that is the second argument. The `..` operator specifies a range, so that `0 .. 9` is the range $[0, 9]$. The `all_distinct` predicate constraints the variables in the argument to take on distinct values. Finally, we use `label` at the end to ground the given variables with actual values. We obtain the same result:

```
?- money_c(S), !.
S = [9, 5, 6, 7, 1, 0, 8, 2].
```

The solver can use the set of constraints to eliminate most of the search space, and the remaining candidates are checked when the `label` predicate is reached. The result is that computing this solution takes about 0.003 seconds on the author's iMac, a speedup of about 18000x.

As another example, consider the problem of solving a Sudoku puzzle. The following predicate takes in a nested list of lists, in row-major order, with some entries provided but others filled with anonymous variables:

```
sudoku(Rows) :-
    length(Rows, 9), maplist(same_length(Rows), Rows),
    append(Rows, Values), Values ins 1..9,
    maplist(all_distinct, Rows),
    transpose(Rows, Columns),
    maplist(all_distinct, Columns),
    Rows = [Row1, Row2, Row3, Row4, Row5, Row6, Row7, Row8, Row9],
    blocks(Row1, Row2, Row3),
    blocks(Row4, Row5, Row6),
    blocks(Row7, Row8, Row9),
    maplist(label, Rows).
```

The first body term requires that the number of rows be 9. The second uses `maplist`, which maps a predicate over the items in a list, as we saw in the previous example. The `same_length(Rows)` argument is a partially applied predicate that, when applied to another argument, requires that the two argument lists have the same length. The term as a whole is checking that each row also has the same length as the number of rows. The `append` term takes a list of lists and concatenates them into the single list `Values`. We then constrain that each variable be in the range $[1, 9]$. The next term constrains each row to consist of distinct numbers, and the following two terms constrain each of the columns to consist of distinct numbers. The next four terms constrain each of the 9x9 squares to be composed of distinct numbers, with the `blocks` predicate defined below. Finally, the last term ensures that each variable is grounded to a value.

The `blocks` predicate is as follows:

```
blocks([], [], []).
blocks([N1, N2, N3 | RestRow1],
       [N4, N5, N6 | RestRow2],
       [N7, N8, N9 | RestRow3]) :-
    all_distinct([N1, N2, N3, N4, N5, N6, N7, N8, N9]),
    blocks(RestRow1, RestRow2, RestRow3).
```

The predicate takes in three rows, ensures that the set consisting of the first three items from each row contains distinct values, and then recursively checks this for the remaining items in each row.

We can now provide a query to solve a specific puzzle. The following has been called the "world's hardest Sudoku":

```
?- S = [[8,_,_,_,_,_,_,_,_],
        [_,_,3,6,_,_,_,_,_],
        [_,7,_,_,9,_,2,_,_],
        [_,5,_,_,_,7,_,_,_],
        [_,_,_,_,4,5,7,_,_],
        [_,_,_,1,_,_,_,3,_],
        [_,_,1,_,_,_,_,6,8],
        [_,_,8,5,_,_,_,1,_],
        [_,9,_,_,_,_,4,_,_]],
   sudoku(S).
S = [[8, 1, 2, 7, 5, 3, 6, 4, 9],
     [9, 4, 3, 6, 8, 2, 1, 7, 5],
     [6, 7, 5, 4, 9, 1, 2, 8, 3],
     [1, 5, 4, 2, 3, 7, 8, 9, 6],
     [3, 6, 9, 8, 4, 5, 7, 2, 1],
     [2, 8, 7, 1, 6, 9, 5, 3, 4],
     [5, 2, 1, 9, 7, 4, 3, 6, 8],
     [4, 3, 8, 5, 2, 6, 9, 1, 7],
     [7, 9, 6, 3, 1, 8, 4, 5, 2]] .
```

Solving this takes less than a tenth of a second on the author's iMac.

## 25.2 Make

Make is a family of tools used for automating the building of software packages, as well as tracking dependencies between the various components of a package. Make operates on programs called *makefiles*, which contain *rules* for how to build individual *targets*. A target may have *dependencies*, which are required to be satisfied before the target can be built, as well as *commands* that specify how the target should actually be built. Thus, a makefile is a combination of declarative components relating targets to dependencies and imperative actions specifying the actions required to build a target.

The structure of a rule in a makefile is as follows:

```
target: dependencies
        commands
```

Here, `dependencies` is a list of zero or more targets or files that the given target depends on, and `commands` is a list of zero or more actions to be taken, generally each on its own line and indented with a tab character.

As an example, consider the following simple makefile, located by convention in a file named `Makefile` (note the capitalization):

```
hello:
        echo "Hello world!"
```

We can run this from the terminal, if we are in the same directory, as:

```
$ make hello
echo "Hello world!"
Hello world!
```

This invokes the `hello` target, which has no dependencies and as its sole action invokes the shell command to print `Hello world!` to the screen. We can leave out the explicit target when invoking `make`, in which case it will build the first target in the makefile:

```
$ make
echo "Hello world!"
Hello world!
```

The target of a rule is commonly an executable file, and the dependencies are the files needed to build the target. For example, suppose we have a C++ project with the source files `a.cpp`, `b.cpp`, and `c.cpp`. We can structure our makefile as follows:

```
main: a.o b.o c.o
        g++ -o main a.o b.o c.o

a.o: a.cpp
        g++ --std=c++14 -Wall -Werror -pedantic -c a.cpp

b.o: b.cpp
        g++ --std=c++14 -Wall -Werror -pedantic -c b.cpp

c.o: c.cpp
        g++ --std=c++14 -Wall -Werror -pedantic -c c.cpp
```

Here, our default rule is `main`, which depends on the targets `a.o`, `b.o`, and `c.o`. In order to build `main`, those targets have to be built first, so Make will attempt to build each of those targets using their respective rules. The rule for `a.o` depends on the file `a.cpp`, and if it exists, the command invokes `g++` to build the object file `a.o`. The rules for `b.o` and `c.o` have the same structure. Once those targets have been built, Make can then build `main`, which links together the object files into the final `main` executable. Running `make` indicates the sequence of operations:

```
$ make
g++ --std=c++14 -Wall -Werror -pedantic -c a.cpp
g++ --std=c++14 -Wall -Werror -pedantic -c b.cpp
g++ --std=c++14 -Wall -Werror -pedantic -c c.cpp
g++ -o main a.o b.o c.o
```

Thus, we can specify complex dependency trees with rules in a makefile, and the Make tool will automatically resolve the dependencies and build the required targets. The relationship between a target and its dependencies is specified declaratively in a rule.

A key feature of Make is that it only builds a target if it has a dependency, direct or indirect through other rules, that is newer than the target itself. For instance, if we follow up the preceding build by modifying the timestamp on `b.cpp`, we can see that it is newer than the targets `b.o` and `main`:

```
$ touch b.cpp
$ ls -l
-rw-r--r--  1 kamil  staff    229 Nov 17 01:01 Makefile
-rw-r--r--  1 kamil  staff     90 Nov 17 00:57 a.cpp
-rw-r--r--  1 kamil  staff   6624 Nov 17 01:01 a.o
-rw-r--r--  1 kamil  staff     31 Nov 17 01:12 b.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 b.o
-rw-r--r--  1 kamil  staff     33 Nov 17 00:58 c.cpp
-rw-r--r--  1 kamil  staff    640 Nov 17 01:01 c.o
-rwxr-xr-x  1 kamil  staff  15268 Nov 17 01:01 main
```

If we then run `make`, it will only rebuild those targets that depend on `b.cpp`:

```
$ make
g++ --std=c++14 -Wall -Werror -pedantic -c b.cpp
g++ -o main a.o b.o c.o
```

This is a crucial feature for working with large projects, as only the components that depend on a modification are rebuilt rather than every target in the project.

As a more complex example, consider the following makefile that was used to build a previous version of this text:

```
all: foundations functional theory data declarative

foundations: foundations.html foundations.tex

functional: functional.html functional.tex

theory: theory.html theory.tex

data: data.html data.tex

declarative: declarative.html declarative.tex

asynchronous: asynchronous.html asynchronous.tex
```

(continues on next page)

(continued from previous page)

```
metaprogramming: metaprogramming.html metaprogramming.tex

%.html: %.rst
        rst2html.py --stylesheet=style/style.css $< > $@

%.tex: %.rst
        rst2latex.py --stylesheet=style/style.sty $< > $@
        pdflatex $@
        pdflatex $@
        pdflatex $@

clean:
        rm -vf *.html *.tex *.pdf *.aux *.log *.out
```

The default target is `all`, which depends on the `foundations`, `functional`, `theory`, `data`, and `declarative` targets. While there are also `asynchronous` and `metaprogramming` targets, they were not being built since we had not reached the corresponding units in the text.

Each of the following standard targets has two dependencies, an `.html` file and a `.tex` file. In order to build an `.html` file, Make looks for an appropriate target. We have a *pattern rule* for `.html` files, which depends on a corresponding `.rst` file. Thus, in order to build, for example, `declarative.html`, Make applies the pattern rule and invokes `rst2html.py`. The special symbol `$<` stands for the dependencies, while `$@` stands for the target. Thus, the result of `rst2html.py` is written to `declarative.html`, and the build for that target is complete.

We also have a pattern rule for `.tex` files, which invokes `rst2latex.py`, followed by several invocations of `pdflatex`. The end result is that building `declarative.tex` ends up creating `declarative.pdf` as well.

The last rule is to clean up target and temporary files. Thus, we can force the `all` target to be built from scratch with `make clean all`. Without requesting the `clean` target, only those targets that depend on an `.rst` file that has been modified will be rebuilt.

CHAPTER

# TWENTYSIX

# PATTERN MATCHING

Many languages that are primarily functional or imperative provide a declarative construct that does *pattern matching*, specifying separate cases that each define a pattern against which a value can match, and the computation to be done as a result. These separate cases are analogous to different axioms in Prolog or different pattern rules in Make, and the matching process is similar to *unification* in Prolog.

As an example, we take a look at the `match` statement in Python, which has the following syntax:

```
match <expression>:
    case <pattern>:
        <suite>
    case <pattern>:
        <suite>
    ...
```

A controlling expression provides the value to be matched, and one or more `case` clauses specify a matching pattern and a suite of statements to be executed upon a match. Only the first clause that matches the value is executed – the remaining clauses are skipped, even if their patterns also match the value. This is similar to the behavior of a sequence of `if` and `elif` branches, or a sequence of `except` clauses on a `try` statement.

There are several kinds of patterns that can be specified, a subset of which are the following:

- A *literal pattern* specifies a number, string, or boolean literal, or the `None` literal, and it matches a value that compares equal to the literal. The following is an example:

  ```
  def https_error_description(code):
      match code:
          case 400:
              return 'Bad Request'
          case 401:
              return 'Unauthorized'
          case 403:
              return 'Forbidden'
          case 404:
              return 'Not Found'
      return f'Unknown code {code}'
  ```

  This is similar to a *switch statement*, but the `match` construct is much more powerful in that it supports other patterns as well.

- A *capture pattern* specifies an identifier, and it matches against any value, binding the identifier to that value. In the example above, we can use such a pattern to incorporate the default result in the `match` statement:

```
def https_error_description(code):
    match code:
        ...
        case 404:
            return 'Not Found'
        case unknown:
            return f'Unknown code {code}'
```

Since we don't actually use the variable introduced in the last case, we can use a lone underscore instead as an anonymous variable:

```
def https_error_description(code):
    match code:
        ...
        case _:
            return f'Unknown code {code}'
```

A case with a pattern that consists solely of an identifier matches any value, so such a case is only permitted as the last one in a `match` statement.

- A *class pattern* only matches a value that is an instance of the class. The simplest class pattern consists of a type name followed by empty parentheses:

```
def https_error_description(code):
    match code:
        ...
        case int():  # only matches an int
            return f'Unknown code {code}'
        case _:
            raise ValueError(f'expected an int, got {code}')
```

A class pattern may also specify attributes that the object must have using syntax similar to keyword arguments:

```
case Point(x=0, y=0):
    ...
```

Alternatively, the class itself may define custom matching using syntax similar to positional arguments. For instance, several built-in types allow patterns of the form `<type>(<subpattern>)`, which matches against a value that is an instance of `<type>` and that also matches `<subpattern>`. Thus, the pattern `int(3)` matches an `int` whose value is 3, and the pattern `int(value)` matches any `int` and binds the name `value` to that object.

- A *sequence pattern* consists of a comma-separated list of subpatterns, which can be enclosed by either square brackets or parentheses[1]. Such a pattern can match a variety of sequence types (including user-defined ones that meet a certain set of conditions), though strings are excluded from matching a sequence pattern.

  The following is an example of simple sequence patterns:

```
def is_short(sequence):
    match sequence:
        case []:
            return True
        case [_]:
            return True
```

(continues on next page)

[1] If there is only one subpattern, then a trailing comma is required if parentheses are used, to distinguish from normal parenthesization. The parentheses or brackets can be elided for patterns consisting of two or more subpatterns.

(continued from previous page)

```
        case [_, _]:
            return True
        case _:
            return False
```

The first case matches an empty sequence, the second a one-element sequence, and the third a two-element sequence. We use anonymous variables since we are not concerned with the actual element values. Note that the pattern `[_, _]` has different behavior than `[var, var]`. The latter requires both elements to be equal, since the first occurrence of `var` binds the variable to a value and the second checks whether the corresponding element is equal to that value. On the other hand, the anonymous variable `_` does not do any binding, so each occurrence is independent of any others.

A sequence pattern may also contain a variadic subpattern, matching zero or more occurrences. The syntax is similar to variadic positional arguments:

```
def is_short(sequence):
    match sequence:
        case [_, _, *_]:
            return False
        case _:
            return True
```

Here, the pattern `[_, _, *_]` matches a sequence with at least two elements – the first occurrence of `_` matches the first element, the second occurrence matches second element, and the `*_` matches all remaining elements.

The following is an example of recursively computing the length of a sequence using sequence patterns:

```
def length(sequence):
    match sequence:
        case []:
            return 0
        case [_, *rest]:
            return 1 + length(rest)
```

The first case matches an empty sequence, the base case of the computation. The second matches a sequence with at least one element, and all but the first element match the variadic `*rest` subpattern. These elements are encapsulated in a list bound to the `rest` variable, and we can recurse on this list.

Compare this definition of `length` with an equivalent predicate in Prolog:

```
len([], 0).
len([_|Rest], Length) :-
  len(Rest, RestLength),
  Length is 1 + RestLength.
```

In both definitions, we specify two separate cases with patterns that are matched against an input list.

There are several other patterns, including mapping patterns, `as` patterns, value patterns, and "or" patterns, and a case can include a *guard* expressed as an `if` clause to further restrict what the case matches. More details are in the original specification as well as the tutorial.

We consider one more, complex example that illustrates the declarative nature of pattern matching. Suppose we want to compute the sum of all the elements in a nested list of numbers. The following expresses this computation in Prolog:

```
% deep_sum(NestedList, Sum).
% True if NestedList is a nested list of numbers, and Sum is the sum
% of all the numbers contained in NestedList.
deep_sum([], 0).
deep_sum([First|Rest], Sum) :-
  deep_sum(First, FirstSum),
  deep_sum(Rest, RestSum),
  Sum is FirstSum + RestSum.
deep_sum(Item, Item). % can restrict this to numbers with :- number(Item).
```

We have three cases: an empty list whose sum is zero, a non-empty list whose sum is the recursive sum of the first item (which itself may be a list) plus the recursive sum of the rest of the list, and a non-list item whose sum is itself. We can express this same computation in Python:

```
def deep_sum(nested_list):
    """Return the sum of all the numbers in nested_list.

    nested_list must be a nested list of numbers.
    """
    match nested_list:
        case []:
            return 0
        case [first, *rest]:
            return deep_sum(first) + deep_sum(rest)
        case item:  # can do int(item) | float(item) to restrict to numbers
            return item
```

There is a direct correspondence between the two implementations, reflecting the primarily declarative manner in which the computation is expressed. The fundamental difference between Prolog and Python here is that Prolog provides search and backtracking, so that other axioms can be tried if one fails (or more solutions are requested). In Python and other functional or imperative languages that have pattern matching, a value matches at most one case, and the remaining cases are never considered. Thus, while pattern matching is declarative, it does not provide the full expressiveness of logic programming.

# Part VI

# Metaprogramming

*Metaprogramming* is the technique of writing a computer program that operates on other programs. Systems such as compilers and program analyzers can be considered metaprograms, since they take other programs as input. The forms of metaprogramming we will discuss here are specifically concerned with generating code to be included as part of a program. In a sense, they can be considered rudimentary compilers.

CHAPTER

# TWENTYSEVEN

# MACROS AND CODE GENERATION

A *macro* is a rule that translates an input sequence into some replacement output sequence. This translation process is called *macro expansion*, and some languages provide macros as part of their specification. The macro facility may be implemented as a *preprocessing* step, where macro expansion occurs before lexical and syntactic analysis, or it may be incorporated as part of syntax analysis or a later translation step.

One of the most widely used macro systems is the C preprocessor (CPP), which is included in both C and C++ as the first step in processing a program. Preprocessor *directives* begin with a hash symbol and include `#include`, `#define`, `#if`, among others. For instance, the following defines a *function-like* macro to swap two items:

```
#define SWAP(a, b) { auto tmp = b; b = a; a = tmp; }
```

We can then use the macro as follows:

```
int main() {
  int x = 3;
  int y = 4;
  SWAP(x, y);
  cout << x << " " << y << endl;
}
```

Running the resulting executable will print a 4, followed by a 3.

The results of macro expansion can be obtained by passing the `-E` flag to `g++`:

```
$ g++ -E <source>
```

However, the results can be quite messy if there are `#include`s, since that directive pulls in the code from the given file.

CPP macros perform text replacement, so that the code above is equivalent to:

```
int main() {
  int x = 3;
  int y = 4;
  { auto tmp = y; y = x; x = tmp; };
  cout << x << " " << y << endl;
}
```

The semicolon following the use of the `SWAP` macro remains, denoting an empty statement. This is a problem, however, in contexts that require a single statement, such as a conditional branch that is not enclosed by a block:

```
if (x < y)
  SWAP(x, y);
```

(continues on next page)

(continued from previous page)

```
else
  cout << "no swap" << endl;
```

A common idiom to avoid this problem is to place the expansion code for the macro inside of a do/while:

```
#define SWAP(a, b) do {                         \
    auto tmp = b;                               \
    b = a;                                      \
    a = tmp;                                    \
  } while (false)
```

Here, we've placed a backslash at the end of a line to denote that the next line should be considered a continuation of the previous one. A do/while loop syntactically ends with a semicolon, so that the semicolon in `SWAP(x, y);` is syntactically part of the do/while loop. Thus, the expanded code has the correct syntax:

```
if (x < y)
  do { auto tmp = b; b = a; a = tmp; } while (false);
else
  cout << "no swap" << endl;
```

While textual replacement is useful, it does have drawbacks, stemming from the fact that though the macros are syntactically function like, they do not behave as functions. Specifically, they do not treat arguments as their own entities, and they do not introduce a separate scope. Consider the following example:

```
int main() {
  int x = 3;
  int y = 4;
  int z = 5;
  SWAP(x < y ? x : y, z);
  cout << x << " " << y << " " << z << endl;
}
```

Running the resulting program produces the unexpected result:

```
3 4 3
```

Using `g++ -E`, we can see what the preprocessed code looks like. Looking only at the output for `main()`, we find:

```
int main() {
  int x = 3;
  int y = 4;
  int z = 5;
  do {
    auto tmp = z;
    z = x < y ? x : y;
    x < y ? x : y = tmp;
  } while (false);
  cout << x << " " << y << " " << z << endl;
}
```

Here, we've manually added line breaks and whitespace to make the output more readable; the preprocessor itself places the macro output on a single line. The culprit is the last generated statement:

```
x < y ? x : y = tmp;
```

In C++, the conditional operator `? :` and the assignment operator = have the same precedence and associate right to left, so this is equivalent to:

```
x < y ? x : (y = tmp);
```

Since `x < y`, no assignment happens here. Thus, the value of `x` is unchanged.

We can fix this problem by placing parentheses around each use of a macro argument:

```
#define SWAP(a, b) do {                                 \
    auto tmp = (b);                                     \
    (b) = (a);                                          \
    (a) = tmp;                                          \
  } while (false)
```

This now produces the expected result, as the operations are explicitly associated by the parentheses:

```
int main() {
  int x = 3;
  int y = 4;
  int z = 5;
  do {
    auto tmp = (z);
    (z) = (x < y ? x : y);
    (x < y ? x : y) = tmp;
  } while (false);
  cout << x << " " << y << " " << z << endl;
}
```

A second problem, however, is not as immediately fixable. Consider what happens when we apply the `SWAP` macro to a variable named `tmp`:

```
int main() {
  int x = 3;
  int tmp = 4;
  SWAP(tmp, x);
  cout << x << " " << tmp << endl;
}
```

Running this code results in:

```
3 4
```

No swap occurs! Again, using `g++ -E` to examine the output, we see (modulo spacing):

```
int main() {
  int x = 3;
  int tmp = 4;
  do {
    auto tmp = (x);
    (x) = (tmp);
    (tmp) = tmp;
```

(continues on next page)

(continued from previous page)

```
  } while (false);
  cout << x << " " << tmp << endl;
}
```

Since the temporary variable used by `SWAP` has the same name as an argument, the temporary *captures* the occurrences of the argument in the generated code. This is because the macro merely performs text substitution, which does not ensure that names get resolved to the appropriate scope. (Thus, macros do not actually use call by name, which does ensure that a name in an argument resolves to the appropriate scope.) The reliance on text replacement makes CPP a *non-hygienic* macro system. Other systems, such as Scheme's, are hygienic, creating a separate scope for names introduced by a macro and ensuring that arguments are not captured.

## 27.1 Scheme Macros

The macro system defined as part of the R5RS Scheme specification is hygienic. A macro is introduced by one of the `define-syntax`, `let-syntax`, or `letrec-syntax` forms, and it binds the given name to the macro. As an example, the following is a definition of `let` as a macro:

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...)
       body1 body2 ...
     )
     ((lambda (name ...)
        body1 body2 ...
      )
      val ...
     )
    )
  )
)
```

The `syntax-rules` from specifies the rules for the macro transformation. The first argument is a list of literals that must match between the pattern of the rule and the input. An example is the `else` identifier inside of a `cond` form. In this case, however, there are no literals. The remaining arguments to `syntax-rules` specify transformations. The first item in a transformation is the input pattern, and the second is the output pattern. The `...` acts like a Kleene star, matching the previous item to zero or more occurrences in the input. The names that appear in an input pattern but are not in the list of literals, excepting the first item that is the macro name, are hygienic variables that match input elements. The variables can then be referenced in the output pattern to specify how to construct the output.

Evaluating the expression above in the global environment binds the name `let` to a macro that translates to a `lambda`.

Identifiers introduced by the body of a macro are guaranteed to avoid conflict with other identifiers, and the interpreter often renames identifiers to avoid such a conflict. Consider the following definition of a `swap` macro:

```
(define-syntax swap
  (syntax-rules ()
    ((swap a b)
     (let ((tmp b))
       (set! b a)
       (set! a tmp)
     )
    )
```

(continues on next page)

(continued from previous page)

```
  )
)
```

This translates a use of `swap` to an expression that swaps the two arguments through a temporary variable `tmp`. Thus:

```
> (define x 3)
> (define y 4)
> (swap x y)
> x
4
> y
3
```

However, unlike CPP macros, the `tmp` introduced by the `swap` macro is distinct from any other `tmp`:

```
> (define tmp 5)
> (swap x tmp)
> x
5
> tmp
4
```

Because macros are hygienic in Scheme, we get the expected behavior.

In order to support macros, the evaluation procedure of the Scheme interpreter evaluates the first item in a list, as usual. If it evaluates to a macro, then the interpreter performs macro expansion on the rest of the list without first evaluating the arguments. Any names introduced by the expansion are placed in a separate scope from other names. After expansion, the interpreter repeats the evaluation process on the result of expansion, so that if the end result is a `let` expression as in `swap` above, the expression is evaluated.

A macro definition can specify multiple pattern rules. Combined with the fact that the result of expansion is evaluated, this allows a macro to be recursive, as in the following definition of `let*`:

```
(define-syntax let*
  (syntax-rules ()
    ((let* ()
       body1 body2 ...
     )
     (let ()
       body1 body2 ...
     )
    )
    ((let* ((name1 val1) (name2 val2) ...)
       body1 body2 ...
     )
     (let ((name1 val1))
       (let* ((name2 val2) ...)
         body1 body2 ...
       )
     )
    )
  )
)
```

There is a base-case pattern for when the `let*` has no bindings, in which case it translates directly into a `let`. There is also a recursive pattern for when there is at least one binding, in which case the `let*` translates into a simpler `let*` nested within a `let`. The ellipsis (`...`) in a macro definition is similar to a Kleene star (*) in a regular expression, denoting that the preceding item can be matched zero or more times. Thus, a `let*` with a single binding matches the second pattern rule above, where `(name2 val2)` is matched zero times.

## 27.2 CPP Macros

We return our attention to CPP macros. Despite their non-hygienic nature, they can be very useful in tasks that involve metaprogramming.

CPP allows us to use `#define` to define two types of macros, *object-like* and *function-like* macros. An object-lke macro is a simple text replacement, substituting one sequence of text for another. Historically, a common use was to define constants:

```
#define PI 3.1415926535

int main() {
  cout << "pi = " << PI << endl;
  cout << "tau = " << PI * 2 << endl;
}
```

Better practice in C++ is to define a constant using `const` or `constexpr`.

A function-like macro takes arguments, as in `SWAP` above, and can substitute the argument text into specific locations within the replacement text.

A more complex example of using function-like macros is to abstract the definition of multiple pieces of code that follow the same pattern. Consider the definition of a type to represent a complex number:

```
struct Complex {
  double real;
  double imag;
};

ostream &operator<<(ostream &os, Complex c) {
  return os << "(" << c.real << "+" << c.imag << "i)";
}
```

Suppose that in addition to the overloaded stream insertion operator above, we wish to support the arithmetic operations +, -, and *. These operations all have the same basic form:

```
Complex operator <op>(Complex a, Complex b) {
  return Complex{ <expression for real>, <expression for imag> };
}
```

Here, we've used *uniform initialization syntax* to initialize a `Complex` with values for its members. We can then write a function-like macro to abstract this structure:

```
#define COMPLEX_OP(op, real_part, imag_part)    \
  Complex operator op(Complex a, Complex b) {   \
    return Complex{ real_part, imag_part };     \
  }
```

The macro has arguments for each piece that differs between operations, namely the operator, the expression to compute the real part, and the expression to compute the imaginary part. We can use the macro as follows to define the operations:

```
COMPLEX_OP(+, a.real+b.real, a.imag+b.imag);
COMPLEX_OP(-, a.real-b.real, a.imag-b.imag);
COMPLEX_OP(*, a.real*b.real - a.imag*b.imag,
           a.imag*b.real + a.real*b.imag);
```

As with our initial `SWAP` implementation, the trailing semicolon is extraneous but improves readability and interaction with syntax highlighters. Running the code through the preprocessor with `g++ -E`, we get (modulo spacing):

```
Complex operator +(Complex a, Complex b) {
  return Complex{ a.real+b.real, a.imag+b.imag };
};
Complex operator -(Complex a, Complex b) {
  return Complex{ a.real-b.real, a.imag-b.imag };
};
Complex operator *(Complex a, Complex b) {
  return Complex{ a.real*b.real - a.imag*b.imag,
                  a.imag*b.real + a.real*b.imag };
};
```

We can then proceed to define operations between `Complex` and `double` values. Again, we observe that such an operation has a specific pattern:

```
Complex operator <op>(<type1> a, <type2> b) {
  return <expr1> <op> <expr2>;
}
```

Here, `<exprN>` is the corresponding argument converted to its `Complex` representation. We can abstract this using a macro:

```
#define REAL_OP(op, typeA, typeB, argA, argB)   \
  Complex operator op(typeA a, typeB b) {       \
    return argA op argB;                        \
  }
```

We can also define a macro to convert from a `double` to a `Complex`:

```
#define CONVERT(a)                              \
  (Complex{ a, 0 })
```

We can then define our operations as follows:

```
REAL_OP(+, Complex, double, a, CONVERT(b));
REAL_OP(+, double, Complex, CONVERT(a), b);
REAL_OP(-, Complex, double, a, CONVERT(b));
REAL_OP(-, double, Complex, CONVERT(a), b);
REAL_OP(*, Complex, double, a, CONVERT(b));
REAL_OP(*, double, Complex, CONVERT(a), b);
```

Running this through the preprocessor, we get:

```
Complex operator +(Complex a, double b) { return a + (Complex{ b, 0 }); };
Complex operator +(double a, Complex b) { return (Complex{ a, 0 }) + b; };
```

(continues on next page)

(continued from previous page)

```
Complex operator -(Complex a, double b) { return a - (Complex{ b, 0 }); };
Complex operator -(double a, Complex b) { return (Complex{ a, 0 }) - b; };
Complex operator *(Complex a, double b) { return a * (Complex{ b, 0 }); };
Complex operator *(double a, Complex b) { return (Complex{ a, 0 }) * b; };
```

We can now use complex numbers as follows:

```
int main() {
  Complex c1{ 3, 4 };
  Complex c2{ -1, 2 };
  double d = 0.5;
  cout << c1 + c2 << endl;
  cout << c1 - c2 << endl;
  cout << c1 * c2 << endl;
  cout << c1 + d << endl;
  cout << c1 - d << endl;
  cout << c1 * d << endl;
  cout << d + c1 << endl;
  cout << d - c1 << endl;
  cout << d * c1 << endl;
}
```

This results in:

```
(2+6i)
(4+2i)
(-11+2i)
(3.5+4i)
(2.5+4i)
(1.5+2i)
(3.5+4i)
(-2.5+-4i)
(1.5+2i)
```

### 27.2.1 Stringification and Concatenation

When working with macros, it can be useful to convert a macro argument to a string or to concatenate it with another token. For instance, suppose we wanted to write an interactive application that would read input from a user and perform the corresponding action. On complex numbers, the target functions may be as follows:

```
Complex Complex_conjugate(Complex c) {
  return Complex{ c.real, -c.imag };
}

string Complex_polar(Complex c) {
  return "(" + to_string(sqrt(pow(c.real, 2) + pow(c.imag, 2))) +
    "," + to_string(atan(c.imag / c.real)) + ")";
}
```

The application would compare the user input to a string representing an action, call the appropriate function, and print out the result. This has the common pattern:

```
if (<input> == "<action>")
  cout << Complex_<action>(<value>) << endl;
```

Here, we both need a string representation of the action, as well as the ability to concatenate the `Complex_` token with the action token itself. We can define a macro for this pattern as follows:

```
#define ACTION(str, name, arg)                  \
  if (str == #name)                             \
    cout << Complex_ ## name(arg) << endl
```

The `#` preceding a token is the *stringification* operator, converting the token to a string. The `##` between `Complex_` and `name` is the *token pasting* operator, concatenating the tokens on either side.

We can then write our application code as follows:

```
Complex c1 { 3, 4 };
string s;
while (cin >> s) {
  ACTION(s, conjugate, c1);
  ACTION(s, polar, c1);
}
```

Running this through the preprocessor, we obtain the desired result:

```
Complex c1 { 3, 4 };
string s;
while (cin >> s) {
  if (s == "conjugate") cout << Complex_conjugate(c1) << endl;
  if (s == "polar") cout << Complex_polar(c1) << endl;
}
```

### 27.2.2 The Macro Namespace

One pitfall of using CPP macros is that they are not contained within any particular namespace. In fact, a macro, as long as it is defined, will replace *any* eligible token, regardless of where the token is located. Thus, defining a macro is akin to making a particular identifier act as a reserved keyword, unable to be used by the programmer. (This is one reason why constants are usually better defined as variables qualified `const` or `constexpr` than as object-like macros.)

Several conventions are used to avoid polluting the global namespace. The first is to prefix all macros with characters that are specific to the library defining them in such a way as to avoid conflict with other libraries. For instance, our complex-number macros may be prefixed with `COMPLEX_` to avoid conflicting with other macros or identifiers. The second strategy is to undefine macros when they are no longer needed, using the `#undef` preprocessor directive. For example, at the end of our library code, we may have the following:

```
#undef COMPLEX_OP
#undef REAL_OP
#undef CONVERT
#undef ACTION
```

This frees the identifiers to be used for other purposes in later code.

## 27.3 Code Generation

While macros allow us to generate code using the macro facilities provided by a language, there are some cases where such a facility is unavailable or otherwise insufficient for our purposes. In such a situation, it may be convenient to write a *code generator* in an external program, in the same language or in a different language. This technique is also called *automatic programming*.

As an example, the R5RS Scheme specification requires implementations to provide combinations of `car` and `cdr` up to four levels deep. For instance, `(caar x)` should be equivalent to `(car (car x))`, and `(caddar x)` should be equivalent to `(car (cdr (cdr (car x))))`. Aside from `car` and `cdr` themselves, there are 28 combinations that need to be provided, which would be tedious and error-prone to write by hand. Instead, we can define the following Python script to generate a Scheme library file:

```
import itertools

def cadrify(seq):
    if len(seq):
        return '(c{0}r {1})'.format(seq[0], cadrify(seq[1:]))
    return 'x'

def defun(seq):
    return '(define (c{0}r x) {1})'.format(''.join(seq), cadrify(seq))

for i in range(2, 5):
    for seq in itertools.product(('a', 'd'), repeat=i):
        print(defun(seq))
```

The `cadrify()` function is a recursive function that takes in a sequence such as `('a', 'd', 'a')` and constructs a call using the first item and the recursive result of the rest of the sequence. In this example, the latter is `(cdr (car x))`, so the result would be `(car (cdr (car x)))`. The base case is in which the sequence is empty, producing just `x`.

The `defun()` function takes in a sequence and uses it construct the definition for the appropriate combination. It calls `cadrify()` to construct the body. For the sequence `('a', 'd', 'a')`, the result is:

```
(define (cadar x) (car (cdr (car x))))
```

Finally, the loop at the end produces all combinations of `'a'` and `'d'` for each length. It uses the library function `itertools.product()` to obtain a sequence that is the `i`th power of the tuple `('a', 'd')`. For each combination, it calls `defun()` to generate the function for that combination.

Running the script results in:

```
(define (caar x) (car (car x)))
(define (cadr x) (car (cdr x)))
(define (cdar x) (cdr (car x)))
(define (cddr x) (cdr (cdr x)))
(define (caaar x) (car (car (car x))))
(define (caadr x) (car (car (cdr x))))
...
(define (cdddar x) (cdr (cdr (cdr (car x)))))
(define (cddddr x) (cdr (cdr (cdr (cdr x)))))
```

We can place the resulting code in a standard library to be loaded by the Scheme interpreter.

CHAPTER

TWENTYEIGHT

# TEMPLATE METAPROGRAMMING

*Template metaprogramming* is a technique that uses templates to produce source code at compile time, which is then compiled with the rest of the program's code. It generally refers to a form of compile-time execution that takes advantage of the language's rules for template instantiation. Template metaprogramming is most common in C++, though a handful of other languages also enable it.

The key to template metaprogramming in C++ is *template specialization*, which allows a specialized definition to be written for instantiating a template with specific arguments. For example, consider a class template that contains a static `value` field that is true if the template argument is `int` but false otherwise. We can write the generic template as follows:

```
template <class T>
struct is_int {
  static const bool value = false;
};
```

We can now define a *specialization* for this template when the argument is `int`:

```
template <>
struct is_int<int> {
  static const bool value = true;
};
```

The template parameter list in a specialization contains the non-specialized parameters, if any. In the case above, there are none, so it is empty. Then after the name of the template, we provide the full set of arguments for the instantiation, in this case just `int`. We then provide the rest of the definition for the instantiation.

Now when we use the template, the compiler uses the specialization if the template argument is compatible with the specialization, otherwise it uses the generic template:

```
cout << is_int<double>::value << endl;
cout << is_int<int>::value << endl;
```

This prints a 0 followed by a 1.

Template specialization enables us to write code that is conditional on a template argument. Combined with recursive instantiation, this results in template instantiation being Turing complete. Templates do not encode variables that are mutable, so template metaprogramming is actually a form of functional programming.

## 28.1 Pairs

As a more complex example, let us define pairs and lists that can be manipulated at compile time. The elements stored in these structures will be arbitrary types.

Before we proceed to define pairs, we construct a reporting mechanism that allows us to examine results at compile time. We arrange to include the relevant information in an error message generated by the compiler:

```
template <class A, int I>
struct report {
  static_assert(I < 0, "report");
};
```

For simplicity, we make use of an integer template parameter, though we could encode numbers using types instead. When instantiating the `report` template, the `static_assert` raises an error if the template argument `I` is nonnegative. Consider the following:

```
report<int, 5> foo;
```

The compiler will report an error, indicating what instantiation caused the `static_assert` to fail. In Clang, we get an error like the following:

```
pair.cpp:64:3: error: static_assert failed "report"
  static_assert(I < 0, "report");
  ^             ~~~~~
pair.cpp:67:16: note: in instantiation of template class 'report<int, 5>'
      requested here
report<int, 5> foo;
               ^
```

Using GCC, the error is as follows:

```
pair.cpp: In instantiation of 'struct report<int, 5>':
pair.cpp:67:16:   required from here
main.cpp:64:3: error: static assertion failed: report
   static_assert(I < 0, "report");
   ^
```

In both compilers, the relevant information is reported, which is that the arguments to the `report` template are `int` and 5.

We can then define a pair template as follows:

```
template <class First, class Second>
struct pair {
  using car = First;
  using cdr = Second;
};
```

Within the template, we define type aliases `car` and `cdr` to refer to the first and second items of the pair. Thus, `pair<int, double>::car` is an alias for `int`, while `pair<int, double>::cdr` is an alias for `double`.

We can also define type aliases to extract the first and second items from a pair:

```
template <class Pair>
using car_t = typename Pair::car;
template <class Pair>
using cdr_t = typename Pair::cdr;
```

The `typename` keyword is required before `Pair::car` and `Pair::cdr`, since we are using a nested type whose enclosing type is dependent on a template parameter. In such a case, C++ cannot determine that we are naming a type rather than a value, so the `typename` keyword explicitly indicates that it is a type. Using the aliases above, `car_t<pair<int, double>>` is an alias for `int`, while `cdr_t<pair<int, double>>` is an alias for `double`.

In order to represent recursive lists, we need a representation for the empty list:

```
struct nil {
};
```

We can now define a template to determine whether or not a list, represented either by the empty list `nil` or by a `nil`-terminated sequence of `pair`s, is empty. We define a generic template and then a specialization for the case of `nil` as the argument:

```
template <class List>
struct is_empty {
  static const bool value = false;
};

template <>
struct is_empty<nil> {
  static const bool value = true;
};
```

In order to use the field `value` at compile time, it must be a compile-time constant, which we can arrange by making it both static and `const` and initializing it with a compile-time constant. With C++14, we can also define global *variable templates* to encode the length of a list:

```
template <class List>
const bool is_empty_v = is_empty<List>::value;
```

The value of `is_empty_v<nil>` is true, while `is_empty<pair<int, nil>>` is false. Then we can determine at compilation whether or not a list is empty:

```
using x = pair<char, pair<int, pair<double, nil>>>;
using y = pair<float, pair<bool, nil>>;
using z = nil;
report<x, is_empty_v<x>> a;
report<y, is_empty_v<y>> b;
report<z, is_empty_v<z>> c;
```

Here, we introduce type aliases for lists, which act as immutable compile-time variables. We then instantiate `report` with a type and whether or not it is empty. This results in the following error messages from GCC:

```
pair.cpp: In instantiation of 'struct report<pair<char, pair<int,
  pair<double, nil> > >, 0>':
pair.cpp:82:28:   required from here
pair.cpp:73:3: error: static assertion failed: report
   static_assert(I < 0, "report");
```

(continues on next page)

(continued from previous page)

```
   ^~~~~~~~~~~~
pair.cpp: In instantiation of 'struct report<pair<float, pair<bool,
  nil> >, 0>':
pair.cpp:83:28:   required from here
pair.cpp:73:3: error: static assertion failed: report
pair.cpp: In instantiation of 'struct report<nil, 1>':
pair.cpp:84:28:   required from here
pair.cpp:73:3: error: static assertion failed: report
```

Examining the integer argument of `report`, we see that the lists `pair<char, pair<int, pair<double, nil>>>` and `pair<float, pair<bool, nil>>` are not empty, but the list `nil` is.

We can compute the length of a list using recursion:

```
template <class List>
struct length {
  static const int value = length<cdr_t<List>>::value + 1;
};

template <>
struct length<nil> {
  static const int value = 0;
};

template <class List>
const int length_v = length<List>::value;
```

Here, we are using a value from a recursive instantiation of the `length` struct. Since `value` is initialized with an expression consisting of an operation between compile-time constants, it is also a compile-time constant. The recursion terminates at the specialization for `length<nil>`, where the `value` member is directly initialized to 0. As with `is_empty_v`, we define a variable template `length_v` to encode the result. We can compute and report the length of the `x` type alias:

```
report<x, length_v<x>> d;
```

The first argument to `report` is arbitrary, since we only care about the second argument, so we just pass `x` itself. We get:

```
pair.cpp: In instantiation of 'struct report<pair<char, pair<int,
  pair<double, nil> > >, 3>':
pair.cpp:85:26:   required from here
pair.cpp:73:3: error: static assertion failed: report
```

The relevant information is that the length is 3.

We can define even more complex manipulation on lists. For instance, we can reverse a list as follows:

```
template <class List, class SoFar>
struct reverse_helper {
  using type =
    typename reverse_helper<cdr_t<List>,
                            pair<car_t<List>, SoFar>>::type;
};
```

(continues on next page)

(continued from previous page)

```
template <class SoFar>
struct reverse_helper<nil, SoFar> {
  using type = SoFar;
};

template <class List>
using reverse_t = typename reverse_helper<List, nil>::type;
```

Here, we use a helper template to perform the reversal, where the first template argument is the remaining list and the second is the reversed list so far. In each step, we compute a new partial result as `pair<car_t<List>, SoFar>`, adding the first item in the remaining list to the front of the previous partial result. Then `cdr_t<List>` is the remaining list excluding the first item.

The base case of the recursion is when the remaining list is `nil`, in which case the final result is the same as the partial result. We accomplish this with a *partial class template specialization*, which allows us to specialize only some of the arguments to a class template[1]. In `reverse_helper`, we specialize the first argument, so that any instantiation of `reverse_helper` where the first argument is `nil` will use the specialization. The specialization retains a template parameter, which is included in its parameter list. The full argument list appears after the template name, including both the specialized and unspecialized arguments.

We seed the whole computation in the `reverse_t` alias template with the original list and empty partial result. We apply `reverse_t` to `x`:

```
report<reverse_t<x>, 0> e;
```

Here, the second argument is an arbitrary nonnegative value. We get:

```
pair.cpp: In instantiation of 'struct report<pair<double, pair<int,
  pair<char, nil> > >, 0>':
pair.cpp:86:27:   required from here
pair.cpp:73:3: error: static assertion failed: report
```

As a last example, we can now write a template to append two lists:

```
template <class List1, class List2>
struct append {
  using type =
    pair<car_t<List1>,
         typename append<cdr_t<List1>, List2>::type>;
};

template <class List2>
struct append<nil, List2> {
  using type = List2;
};

template <class List1, class List2>
using append_t = typename append<List1, List2>::type;
```

Here, the template appends the second argument to the first argument. This is accomplished by prepending the first item of the first list to the result of appending the second list to the rest of the first list. The recursion terminates when the first list is empty. Applying `append_t` to `x` and `y`:

[1] C++ only allows partial specialization on class templates. Function templates may be specialized, but they cannot be partially specialized.

```
report<append_t<x, y>, 0> f;
```

We get:

```
pair.cpp: In instantiation of 'struct report<pair<char, pair<int,
  pair<double, pair<float, pair<bool, nil> > > > >, 0>':
pair.cpp:87:29:   required from here
pair.cpp:73:3: error: static assertion failed: report
```

## 28.2 Numerical Computations

Using just recursion and template specialization, we could encode numbers using a system like Church numerals. However, C++ also supports integral template parameters, so we can perform compile-time numerical computations using an integer parameter rather than just types.

As an example, consider the following definition of a template to compute the factorial of the template parameter:

```
template <int N>
struct factorial {
  static const long long value = N * factorial<N - 1>::value;
};

template <>
struct factorial<0> {
  static const long long value = 1;
};
```

The generic template multiplies its template argument `N` by the result of computing factorial on `N - 1`. The base case is provided by the specialization for when the argument is 0, where the factorial is 1.

Here, we've used a `long long` to hold the computed value, so that larger results can be computed than can be represented by `int`. We define a template to report a result as follows:

```
template <long long N>
struct report {
  static_assert(N > 0 && N < 0, "report");
};
```

The condition of the `static_assert` is written to depend on the template parameter so that the assertion fails during instantiation, rather than before. Then if we compute the factorial of 5:

```
report<factorial<5>::value> a;
```

We get:

```
factorial.cpp: In instantiation of 'struct report<120ll>':
factorial.cpp:37:34:   required from here
factorial.cpp:33:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

This shows that the result is 120.

We can use a macro to make our program more generic, encoding the argument to `factorial` as a macro that can be defined at compile time:

```
report<factorial<NUM>::value> a;
```

We can even provide a default value:

```
#ifndef NUM
#define NUM 5
#endif
```

Then at the command line, we can specify the argument as follows:

```
$ g++ --std=c++14 factorial.cpp -DNUM=20
factorial.cpp: In instantiation of 'struct report<2432902008176640000ll>':
factorial.cpp:27:33:   required from here
factorial.cpp:23:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

The command-line argument `-D` in GCC and Clang allows us to define a macro from the command line.

Suppose we now attempt to compute the factorial of a negative number:

```
$ g++ --std=c++14 factorial.cpp -DNUM=-1
factorial.cpp: In instantiation of 'const long long int
  factorial<-900>::value':
factorial.cpp:23:36:   recursively required from 'const long long int
  factorial<-2>::value'
factorial.cpp:23:36:   required from 'const long long int
  factorial<-1>::value'
factorial.cpp:37:27:   required from here
factorial.cpp:23:36: fatal error: template instantiation depth exceeds
  maximum of 900 (use -ftemplate-depth= to increase the maximum)
   static const long long value = N * factorial<N - 1>::value;
                                      ^
compilation terminated.
```

We see that the recursion never reaches the base case of 0. Instead, the compiler terminates compilation when the recursion depth reaches its limit. We can attempt to add an assertion that the template argument is non-negative as follows:

```
template <int N>
struct factorial {
  static_assert(N >= 0, "argument to factorial must be non-negative");
  static const long long value = N * factorial<N - 1>::value;
};
```

However, this does not prevent the recursive instantiation, so that what we get is an even longer set of error messages:

```
factorial.cpp: In instantiation of 'struct factorial<-1>':
factorial.cpp:38:25:   required from here
factorial.cpp:23:3: error: static assertion failed: argument to factorial
  must be non-negative
   static_assert(N >= 0, "argument to factorial must be non-negative");
```

(continues on next page)

(continued from previous page)

```
   ^
...
factorial.cpp: In instantiation of 'struct factorial<-900>':
factorial.cpp:24:36:   recursively required from 'const long long int
  factorial<-2>::value'
factorial.cpp:24:36:   required from 'const long long int
  factorial<-1>::value'
factorial.cpp:38:27:   required from here
factorial.cpp:23:3: error: static assertion failed: argument to factorial
  must be non-negative
factorial.cpp: In instantiation of 'const long long int
  factorial<-900>::value':
factorial.cpp:24:36:   recursively required from 'const long long int
  factorial<-2>::value'
factorial.cpp:24:36:   required from 'const long long int
  factorial<-1>::value'
factorial.cpp:38:27:   required from here
factorial.cpp:24:36: fatal error: template instantiation depth exceeds
  maximum of 900 (use -ftemplate-depth= to increase the maximum)
   static const long long value = N * factorial<N - 1>::value;
                                      ^
compilation terminated.
```

Here, we have removed the intermediate error messages between -1 and -900.

In order to actually prevent recursive instantiation when the argument is negative, we can offload the actual recursive work to a helper template. We can then check that the argument is non-negative in `factorial`, converting the argument to 0 if it is negative:

```
template <int N>
struct factorial_helper {
  static const long long value = N * factorial_helper<N - 1>::value;
};

template <>
struct factorial_helper<0> {
  static const long long value = 1;
};

template <int N>
struct factorial {
  static_assert(N >= 0, "argument to factorial must be non-negative");
  static const long long value = factorial_helper<N >= 0 ? N : 0>::value;
};
```

The key here is that `factorial` only instantiates `factorial_helper<0>` if the argument of `factorial` is nonnegative. Thus, we get:

```
$ g++ --std=c++14 factorial.cpp -DNUM=-1
factorial.cpp: In instantiation of 'struct factorial<-1>':
factorial.cpp:38:24:   required from here
factorial.cpp:17:3: error: static assertion failed: argument to factorial
  must be non-negative
```

(continues on next page)

(continued from previous page)

```
   static_assert(N >= 0, "argument to factorial must be non-negative");
   ^
factorial.cpp: In instantiation of 'struct report<1ll>':
factorial.cpp:38:33:   required from here
factorial.cpp:34:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

We no longer have an unbounded recursion. This demonstrates how we can achieve conditional compilation, even without a built-in conditional construct.

An alternative strategy is to use a second, defaulted template argument that tracks whether or not the first argument is positive:

```
template <int N, bool /*Positive*/ = (N > 0)>
struct factorial {
  static const long long value = N * factorial<N - 1>::value;
};

template <int N>
struct factorial<N, false> {
  static const long long value = 1;
};
```

When we instantiate `factorial` with a positive argument, as in `factorial<5>`, the second argument is defaulted to true. Since that does not match the partial specialization, the instantiation uses the generic version of the template. On the other hand, if we instantiate the template with a non-positive argument, such as in `factorial<0>`, the second argument defaults to false, resulting in the partial specialization being used. Thus, the defaulted argument serves to control whether the generic or specialized version is used. Since it's not used for anything else, we need not name the argument, but we have included the `/*Positive*/` comment to document the argument's purpose.

As another example of a numerical computation, the following computes Fibonacci numbers at compile time. For simplicity, we do not implement error checking for negative arguments:

```
template <int N>
struct fib {
  static const long long value = fib<N - 1>::value + fib<N - 2>::value;
};

template <>
struct fib<1> {
  static const long long value = 1;
};

template <>
struct fib<0> {
  static const long long value = 0;
};
```

We have two base cases, provided by separate specializations for when the argument is 0 or 1. As with factorial, we use a macro to represent the input:

```
report<fib<NUM>::value> a;
```

We can then specify the input at the command line:

```
$ g++ --std=c++14 fib.cpp -DNUM=7
fib.cpp: In instantiation of 'struct report<13ll>':
fib.cpp:26:27:   required from here
fib.cpp:22:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

We can even provide the largest input for which the Fibonacci number is representable as a `long long`:

```
$ g++ --std=c++14 fib.cpp -DNUM=92
fib.cpp: In instantiation of 'struct report<7540113804746346429ll>':
fib.cpp:26:27:   required from here
fib.cpp:22:3: error: static assertion failed: report
   static_assert(N > 0 && N < 0, "report");
   ^
```

This computation only takes a fraction of a second, since the C++ compiler only instantiates a template once for a given set of arguments within a single translation unit. Thus, the compiler automatically performs *memoization*, saving the result of a single computation rather than repeating it.

## 28.3 Templates and Function Overloading

While function templates can also be specialized, a function template can also be overloaded with a non-template function. In performing overload resolution, C++ prefers a non-template function over a template instantiation, as long as the parameter and return types of the template instantiation are not superior to the non-template in the given context.

As an example, consider the following function template to convert a value to a string representation:

```
template <class T>
string to_string(const T &item) {
  std::ostringstream oss;
  oss << item;
  return oss.str();
}
```

We can make use of this template, with the compiler performing template-argument deduction, as follows:

```
int main() {
  cout << to_string(Complex{ 3, 3.14 }) << endl;
  cout << to_string(3.14) << endl;
  cout << to_string(true) << endl;
}
```

This results in:

```
(3+3.14i)
3.14
1
```

If we then decide that the representation of a `bool` is undesirable, we can write a function overload as follows:

```
string to_string(bool item) {
  return item ? "true" : "false";
}
```

Since this is a non-template function, C++ will prefer it to the template instantiation `to_string<bool>` when the argument type is `bool`. Thus, the same code in `main()` now results in:

```
(3+3.14i)
3.14
true
```

## 28.4 SFINAE

In considering function overloads, the C++ compiler does not consider it an error if the types and expressions used in the header of a function template are unsuitable for a particular set of template arguments. This is known as *substitution failure is not an error (SFINAE)*, and it is a powerful feature of templates in C++. Rather than producing an error in such a case, the compiler simply removes the template from the set of candidate functions to be considered in overload resolution.

As an example, suppose we wanted to modify our `to_string()` to use `std::to_string()` for the types for which the latter is defined. We can place a dependence on the existence of a suitable `std::to_string()` overload in the header of a new function template:

```
template <class T>
auto to_string(const T &item) -> decltype(std::to_string(item)) {
  return std::to_string(item);
}
```

Here, the trailing return type is necessary so that `std::to_string(item)` appears in the header of the function. Then the function template will fail on substitution if there is no overload of `std::to_string()` such that it can be applied to a value of the template argument. For example, consider calling our `to_string()` on a `Complex` object:

```
cout << to_string(Complex{ 3, 3.14 }) << endl;
```

Our previous `to_string()` template is still viable, so it is considered in overload resolution. The new template we defined above, however, fails to substitute, since there is no definition of `std::to_string()` that can be applied to a `Complex`. Thus, rather than being an error, the second template is merely removed from consideration, and the call resolves to the original template.

With the second template definition, we can still call `to_string()` on a `bool`, since C++ will still prefer the non-template function. However, we run into trouble when attempting to call it on a `double`:

```
to_string.cpp:82:11: error: call to 'to_string' is ambiguous
  cout << to_string(3.14) << endl;
          ^~~~~~~~~~
to_string.cpp:65:8: note: candidate function [with T = double]
string to_string(const T &item) {
       ^
to_string.cpp:72:6: note: candidate function [with T = double]
auto to_string(const T &item) -> decltype(std::to_string(item)) {
     ^
to_string.cpp:76:8: note: candidate function
```

(continues on next page)

(continued from previous page)

```
string to_string(bool item) {
       ^
1 error generated.
```

Both templates are equally viable when the argument is of type `double`, so the compiler cannot disambiguate between them. The non-template overload that takes in a `bool` is also viable, since a `double` can be converted to a `bool`, so it is reported in the error message even though it is inferior to either template.

In order to fix this problem, we need to arrange for the first function template to be nonviable when there is a compatible overload for `std::to_string()`. This requires ensuring that there is a substitution failure for the template when that is the case.

## 28.5 Ensuring a Substitution Failure

There are many tools that are used to ensure a substitution failure. Perhaps the most fundamental is the `enable_if` template, defined in the standard library in the `<type_traits>` header as of C++11. We can also define it ourselves as follows:

```
template <bool B, class T>
struct enable_if {
  typedef T type;
};

template <class T>
struct enable_if<false, T> {
};
```

The generic template takes in a `bool` and a type and defines a member alias for the type argument. The specialization elides this alias when the `bool` argument is false. C++14 additionally defines `enable_if_t` as an alias template, as in the following:

```
template <bool B, class T>
using enable_if_t = typename enable_if<B, T>::type;
```

We can use `enable_if` or `enable_if_t` to induce a failure, as in the following definition for `factorial`:

```
template <int N>
struct factorial {
  static const std::enable_if_t<N >= 0, long long> value =
    N * factorial<N - 1>::value;
};
```

When the template argument `N` is negative, the `enable_if` instantiation has no `type` member, so we get an error:

```
In file included from factorial.cpp:1:0:
/opt/local/include/gcc5/c++/type_traits: In substitution of
    'template<bool _Cond, class _Tp> using enable_if_t = typename
    std::enable_if::type [with bool _Cond = false; _Tp = long long
    int]':
factorial.cpp:36:52:   required from 'struct factorial<-1>'
factorial.cpp:51:25:   required from here
/opt/local/include/gcc5/c++/type_traits:2388:61: error: no type
```

(continues on next page)

(continued from previous page)

```
    named 'type' in 'struct std::enable_if<false, long long int>'
     using enable_if_t = typename enable_if<_Cond, _Tp>::type;
                                                          ^
factorial.cpp: In function 'int main()':
factorial.cpp:51:10: error: 'value' is not a member of 'factorial<-1>'
   report<factorial<NUM>::value> a;
          ^
factorial.cpp:51:10: error: 'value' is not a member of 'factorial<-1>'
factorial.cpp:51:32: error: template argument 1 is invalid
   report<factorial<NUM>::value> a;
                                ^
```

This provides us another mechanism to prevent instantiation of a template with a semantically invalid argument. In this case, substitution failure *is* an error, since the failure did not occur in the header of a function template.

Another option we have is to rely on the fact that variadic arguments are the least preferred alternative in function-overload resolution. Thus, we can write our overloads as helper functions or function templates, with an additional argument to be considered in overload resolution:

```
string to_string_helper(bool item, int ignored) {
  return item ? "true" : "false";
}

template <class T>
auto to_string_helper(const T &item, int ignored)
  -> decltype(std::to_string(item)) {
  return std::to_string(item);
}

template <class T>
string to_string_helper(const T &item, ...) {
  std::ostringstream oss;
  oss << item;
  return oss.str();
}

template <class T>
string to_string(const T &item) {
  return to_string_helper(item, 0);
}
```

Here, `to_string()` calls `to_string_helper()` with the item and a dummy integer argument. We define three overloads of `to_string_helper()` as before, except that the overloads for `bool` and types for which `std::to_string()` is defined take in an extra `int` argument. The generic overload that is viable for all types, however, uses variadic arguments. Since variadic arguments have the lowest priority in function-overload resolution, if both the generic overload and another overload are viable, the latter is chosen. Thus, the overload that uses `std::to_string()` is preferred when `to_string_helper()` is called on a `double`. We no longer have an ambiguity, and we get the desired result when the program is compiled and run:

```
(3+3.14i)
3.140000
true
```

## 28.6 Variadic Templates

As of the C++11 standard, C++ supports *variadic templates*, which are templates that take a variable number of arguments. Both class and function templates can be variadic, and variadic templates enable us to write variadic function overloads that are type safe, unlike C-style varargs.

As an example, consider the definition of a tuple template that encapsulates multiple items of arbitrary type. For simplicity, we implement the template to require at least one item. We can declare such a template as follows:

```
template <class First, class... Rest>
struct tuple;
```

There is a non-variadic parameter, requiring at least one argument to be provided. This is followed by a *parameter pack*, which accepts zero or more arguments. In this case, the ellipsis follows the `class` keyword, so the arguments accepted by the parameter pack are types. We can then declare a tuple as follows:

```
tuple<int> t1;
tuple<double, char, int> t2;
```

In the first instantiation, the template parameter `First` is associated with the argument `int`, while the parameter pack is empty. In the second case, the parameter `First` is associated with `double`, while the parameter pack is associated with `char` and `int`.

Within the template definition, we can use the `sizeof...` operator to determine the size of the parameter pack. Thus, we can compute the size of the tuple as:

```
static const int size = 1 + sizeof...(Rest);
```

Parameter packs are often processed recursively. It is natural to define a tuple itself recursively as a combination of the first data item and a smaller tuple containing all but the first:

```
using first_type = First;
using rest_type = tuple<Rest...>;

first_type first;
rest_type rest;
```

The ellipsis, when it appears to the right of a pattern containing a parameter pack, expands the pattern into comma-separated instantiations of the pattern, one per item in the parameter pack. Thus, if `Rest` is associated with `char` and `int`, `tuple<Rest...>` expands to `tuple<char, int>`.

In the code above, we have introduced type aliases for the type of the first data item and the type of the rest of the tuple. We then declared data members for each of these components. We can write a constructor to initialize them as follows:

```
tuple(First f, Rest... r) : first(f), rest(r...) {}
```

With `First` as `double` and `Rest` as above, this expands to the equivalent of:

```
tuple(double f, char r0, int r1) :
  first(f), rest(r0, r1) {}
```

Both the parameter `Rest... r` as well as the use of the parameter `r...` expand, with `r` replaced by a unique identifier in each instantiation of the pattern.

The full template definition is as follows:

```
template <class First, class... Rest>
struct tuple {
  static const int size = 1 + sizeof...(Rest);

  using first_type = First;
  using rest_type = tuple<Rest...>;

  first_type first;
  rest_type rest;

  tuple(First f, Rest... r) : first(f), rest(r...) {}
};
```

Since this is a recursive definition, we need a base case to terminate the recursion. As stated above, we've chosen to make the base case a tuple containing one item. We can specify this base case with a specialization of the variadic template:

```
template <class First>
struct tuple<First> {
  static const int size = 1;

  using first_type = First;

  first_type first;

  tuple(First f) : first(f) {}
};
```

In order to facilitate using a tuple, we can write a function template to construct a tuple. This can then take advantage of argument deduction for function templates, which is not available for class templates prior to C++17. We write a `make_tuple` variadic function template as follows:

```
template <class... Types>
tuple<Types...> make_tuple(Types... items) {
  return tuple<Types...>(items...);
}
```

We can now make use of this function template to construct a tuple:

```
tuple<int> t1 = make_tuple(3);
tuple<double, char, int> t2 = make_tuple(4.9, 'c', 3);
```

While we now have the ability to construct a tuple, we have not yet provided a convenient mechanism for accessing individual elements from a tuple. In order to do so, we first write a class template to contain a reference to a single element from a tuple. We declare it as follows:

```
template <int Index, class Tuple>
struct tuple_element;
```

The parameter `Index` is the index corresponding to the item referenced by a `tuple_element`, and `Tuple` is the type of the tuple itself. We can then write the base case as follows:

```
template <class Tuple>
struct tuple_element<0, Tuple> {
```

(continues on next page)

(continued from previous page)

```
  using type = typename Tuple::first_type;

  type &item;

  tuple_element(Tuple &t) : item(t.first) {}
};
```

The type of the element at index 0 is aliased by the `first_type` member of a tuple. The element itself is represented by the `first` data member of a tuple object. Thus, we initialize our reference to the item with the `first` member of the tuple argument to the constructor. We also introduce a type alias `type` to refer to the type of the item.

The recursive case decrements the index and passes off the computation to a `tuple_element` instantiated with all but the first item in a tuple:

```
template <int Index, class Tuple>
struct tuple_element {
  using rest_type = tuple_element<Index - 1,
                                  typename Tuple::rest_type>;
  using type = typename rest_type::type;

  type &item;

  tuple_element(Tuple &t) : item(rest_type(t.rest).item) {}
};
```

The `rest_type` member alias of a tuple is the type representing all but the first item in the tuple. We alias `rest_type` in `tuple_element` to recursively refer to a `tuple_element` with a decremented index and the `rest_type` of the tuple. We then arrange to retrieve the item from this recursive instantiation. The constructor creates a smaller `tuple_element` and initializes `item` to refer to the item contained in the smaller `tuple_element`. We similarly alias `type` to refer to the `type` contained in the smaller `tuple_element`.

The following is an alias template for the type of a tuple element:

```
template <int Index, class Tuple>
using tuple_element_t = typename tuple_element<Index, Tuple>::type;
```

We can now write a function template to retrieve an item out of a tuple:

```
template <int Index, class... Types>
tuple_element_t<Index, tuple<Types...>> &get(tuple<Types...> &t) {
  return tuple_element<Index, tuple<Types...>>(t).item;
}
```

The work is offloaded to the `tuple_element` class template, out of which we retrieve both the type of the element and the element itself. But since `get` is implemented as a function template, we can rely on argument deduction for its second template parameter:

```
tuple<double, char, int> t2 = make_tuple(4.9, 'c', 3);
cout << get<0>(t2) << endl;
cout << get<1>(t2) << endl;
cout << get<2>(t2) << endl;
get<0>(t2)++;
get<1>(t2)++;
get<2>(t2)++;
```

(continues on next page)

(continued from previous page)

```
cout << get<0>(t2) << endl;
cout << get<1>(t2) << endl;
cout << get<2>(t2) << endl;
```

This results in:

```
4.9
c
3
5.9
d
4
```

The standard library provides a definition of `tuple`, allowing it to contain zero items, along with `make_tuple()` and `get()` in the `<tuple>` header.

CHAPTER

# TWENTYNINE

# EXAMPLE: MULTIDIMENSIONAL ARRAYS

As an extended example of using metaprogramming to build a complex system, let's consider the implementation of a multidimensional array library in C++. Built-in C++ arrays are very limited: they represent only a linear sequence of elements, and they do not carry any size information. Multidimensional arrays can be represented by arrays of arrays, but this representation can be cumbersome to use and can suffer from poor spatial locality. Instead, most applications linearize a multidimensional array and map a multidimensional index to a linear index. We will use this strategy, but we will abstract the translation logic behind an ADT interface.

## 29.1 Points

We start with an abstraction for a multidimensional index, which we call a *point*. A point consists of a sequence of integer indices, such as $(3, 4, 5)$ for a three-dimensional index. We define a point template as follows:

```
template <int N>
struct point {
  int coords[N];

  int &operator[](int i) {
    return coords[i];
  }

  const int &operator[](int i) const {
    return coords[i];
  }
};
```

The template is parameterized by the dimensionality of the point, and its data representation is an array of coordinates. We overload the index operator for both `const` and non-`const` points.

We provide a stream-insertion operator overload as follows:

```
template <int N>
std::ostream &operator<<(std::ostream &os, const point<N> &p) {
  os << "(" << p[0];
  for (int i = 1; i < N; i++) {
    os << "," << p[i];
  }
  return os << ")";
}
```

In order to work with points, it is useful to have point-wise arithmetic operations on points, as well as comparison operators. For instance, the following are possible definitions of addition and equality:

```
template <int N>
point<N> operator+(const point<N> &a, const point<N> &b) {
  point<N> result;
  for (int i = 0; i < N; i++)
    result[i] = a[i] + b[i];
  return result;
}

template <int N>
bool operator==(const point<N> &a, const point<N> &b) {
  bool result = true;
  for (int i = 0; i < N; i++)
    result = result && (a[i] == b[i]);
  return result;
}
```

There is a lot of similarity between these two functions: they share the same template header, arguments, and overall body structure, with an initial value, a loop to update the value, and a return of that value. Rather than writing several arithmetic and comparison operations with this structure, we can use a function-like macro to abstract the common structure:

```
#define POINT_OP(op, rettype, header, action, retval)          \
  template <int N>                                             \
  rettype operator op(const point<N> &a, const point<N> &b) {  \
    header;                                                    \
    for (int i = 0; i < N; i++)                                \
      action;                                                  \
    return retval;                                             \
  }
```

Then an arithmetic operators such as + or - can be defined as follows:

```
POINT_OP(+, point<N>, point<N> result,
         result[i] = a[i] + b[i], result);
POINT_OP(-, point<N>, point<N> result,
         result[i] = a[i] - b[i], result);
```

These in turn are very similar, with the only difference the two occurrences of + or -. We can then abstract this structure further for arithmetic operations:

```
#define POINT_ARITH_OP(op)                    \
  POINT_OP(op, point<N>, point<N> result,     \
           result[i] = a[i] op b[i], result)
```

Similarly, we can abstract the structure for comparison operations:

```
#define POINT_COMP_OP(op, start, combiner)                    \
  POINT_OP(op, bool, bool result = start,                     \
           result = result combiner (a[i] op b[i]), result)
```

We can now use these macros to define the point operations:

```
POINT_ARITH_OP(+);
POINT_ARITH_OP(-);
POINT_ARITH_OP(*);
POINT_ARITH_OP(/);

POINT_COMP_OP(==, true, &&);
POINT_COMP_OP(!=, false, ||);
POINT_COMP_OP(<, true, &&);
POINT_COMP_OP(<=, true, &&);
POINT_COMP_OP(>, true, &&);
POINT_COMP_OP(>=, true, &&);
```

Compared to writing ten separate functions, this strategy has much less repetition.

One last operation that would be useful is to construct a point of the desired dimensionality from a sequence of coordinates, analogous to `make_tuple()` from the previous section. We can define a variadic function to do so as follows, giving it the name `pt()` for succinctness:

```
template <class... Is>
point<sizeof...(Is)> pt(Is... is) {
  return point<sizeof...(Is)>{{ is... }};
}
```

We use the `sizeof...` operator to compute the dimensionality. The nested initializer lists are required, the outer one for the `point` struct itself and the inner one for initializing its `coords` member, since the latter is an array.

We can now perform operations on points:

```
cout << (pt(3, 4) + pt(1, -2)) << endl;
cout << (pt(1, 2, 3) < pt(3, 4, 5)) << endl;
```

This results in:

```
(4,2)
1
```

## 29.2 Domains

The *domain* of an array is the set of points that it maps to elements. A domain is *rectangular* if the start and end index for each dimension is independent of the indices for the other dimensions. Thus, an array over a rectangular domain maps a rectangular region of space to elements.

We can represent a rectangular domain by an inclusive lower-bound point and an exclusive upper-bound point:

```
template <int N>
struct rectdomain {
  point<N> lwb; // inclusive lower bound
  point<N> upb; // exclusive upper bound

  // Returns the number of points in this domain.
  int size() const {
    if (!(lwb < upb))
      return 0;
```

(continues on next page)

(continued from previous page)

```
    int result = 1;
    for (int i = 0; i < N; i++) {
      // multiple by the span of each dimension
      result *= upb[i] - lwb[i];
    }
    return result;
  }
};
```

We can define an iterator over a rectangular domain as follows, writing it as a nested class within the `rectdomain` template:

```
template <int N>
struct rectdomain {
  ...

  struct iterator {
    point<N> lwb; // inclusive lower bound
    point<N> upb; // inclusive upper bound
    point<N> current; // current item

    // Returns the current point.
    point<N> operator*() const {
      return current;
    }

    // Moves this iterator to the next point in the domain.
    iterator &operator++() {
      // Increment starting at the last dimension.
      for (int i = N - 1; i >= 0; i--) {
        current[i]++;
        // If this dimension is within bounds, then we are done.
        if (current[i] < upb[i])
          return *this;
        // Otherwise, reset this dimension to its minimum and move
        // on to the previous one.
        current[i] = lwb[i];
      }
      // We ran out of dimensions to increment, set this to an end
      // iterator.
      current = upb;
      return *this;
    }

    bool operator==(const iterator &rhs) const {
      return current == rhs.current;
    }

    bool operator!=(const iterator &rhs) const {
      return !operator==(rhs);
    }
  };
```

(continues on next page)

(continued from previous page)

```
  // Return an iterator that is set to the inclusive lower-bound
  // point.
  iterator begin() const {
    return iterator{ lwb, upb, lwb };
  }

  // Return an iterator that is set to the exclusive upper-bound
  // point.
  iterator end() const {
    return iterator{ lwb, upb, upb };
  }
};
```

The iterator keeps track of the lower and upper bounds, as well as the current point. Incrementing an iterator increments the last coordinate of the current point, and if that reaches the upper bound for that coordinate, it is set to the lower bound and the previous coordinate is incremented instead. This process is repeated as necessary, and if the first coordinate reaches its upper bound, the iterator reaches the end.

We can now use rectangular domains as follows:

```
for (auto p : rectdomain<3>{ pt(1, 2, 3), pt(3, 4, 5) })
  cout << p << endl;
```

This results in:

```
(1,2,3)
(1,2,4)
(1,3,3)
(1,3,4)
(2,2,3)
(2,2,4)
(2,3,3)
(2,3,4)
```

## 29.3 Arrays

We can now proceed to define an ADT for a multidimensional array. We can represent it with a rectangular domain and a C++ array to store the elements. We also keep track of the size of each dimension for the purposes of index computations. The following is an implementation:

```
template <class T, int N>
struct ndarray {
  rectdomain<N> domain; // domain of this array
  int sizes[N];         // cached size of each dimension
  T *data;              // storage for the elements

  // Constructs an array with the given domain, default initializing
  // the elements.
  ndarray(const rectdomain<N> &dom)
    : domain(dom), data(new T[dom.size()]) {
```

(continues on next page)

(continued from previous page)

```
    // Compute and store sizes of each dimension.
    for (int i = 0; i < N; i++) {
      sizes[i] = domain.upb[i] - domain.lwb[i];
    }
  }

  // Copy constructor does a deep copy.
  ndarray(const ndarray &rhs)
    : domain(rhs.domain), data(new T[domain.size()]) {
    std::copy(rhs.data, rhs.data + domain.size(), data);
    std::copy(rhs.sizes, rhs.sizes + N, sizes);
  }

  // Assignment operator does a deep copy.
  ndarray &operator=(const ndarray &rhs) {
    if (&rhs == this)
      return *this;
    delete[] data;
    domain = rhs.domain;
    data = new T[domain.size()];
    std::copy(rhs.data, rhs.data + domain.size(), data);
    std::copy(rhs.sizes, rhs.sizes + N, sizes);
    return *this;
  }

  // Destructor deletes the underlying storage and the elements
  // within.
  ~ndarray() {
    delete[] data;
  }

  // Translates a multidimensional point index into a
  // single-dimensional index into the storage array.
  int indexof(const point<N> &index) const;

  // Returns the element at the given multidimensional index.
  T &operator[](const point<N> &index) {
    return data[indexof(index)];
  }

  // Returns the element at the given multidimensional index.
  const T &operator[](const point<N> &index) const {
    return data[indexof(index)];
  }
};
```

The class template is parameterized by the element type and dimensionality. A constructor takes in a rectangular domain, allocates an underlying array of the appropriate size to hold the elements, and stores the size of each dimension. The Big Three are implemented as needed. (We elide the move constructor and move assignment operator for simplicity.) We then have a function to translate a multidimensional index into a linear one, which the overloaded index operators use to obtain an element.

The `indexof()` function uses the combination of the input point and the size of each dimension to linearize the index.

In our representation, the array is stored in *row-major format*, so that the last dimension is the contiguous one:

```
template <class T, int N>
int ndarray<T, N>::indexof(const point<N> &index) const {
  int result = index[0] - domain.lwb[0];
  for (int i = 1; i < N; i++) {
    result = result * sizes[i-1] + (index[i] - domain.lwb[i]);
  }
  return result;
}
```

Since the value of `N` is a compile-time constant, this loop can be trivially unrolled by the compiler, eliminating any branching and resulting in a faster computation.

## 29.4 Stencil

We can now use arrays to perform a *stencil computation*, which iteratively computes the value of a grid point based on its previous value and the previous values of its neighbors. Figure 29.1 is an example of a stencil update associated with Conway's Game of Life, on a $3 \times 3$ grid.

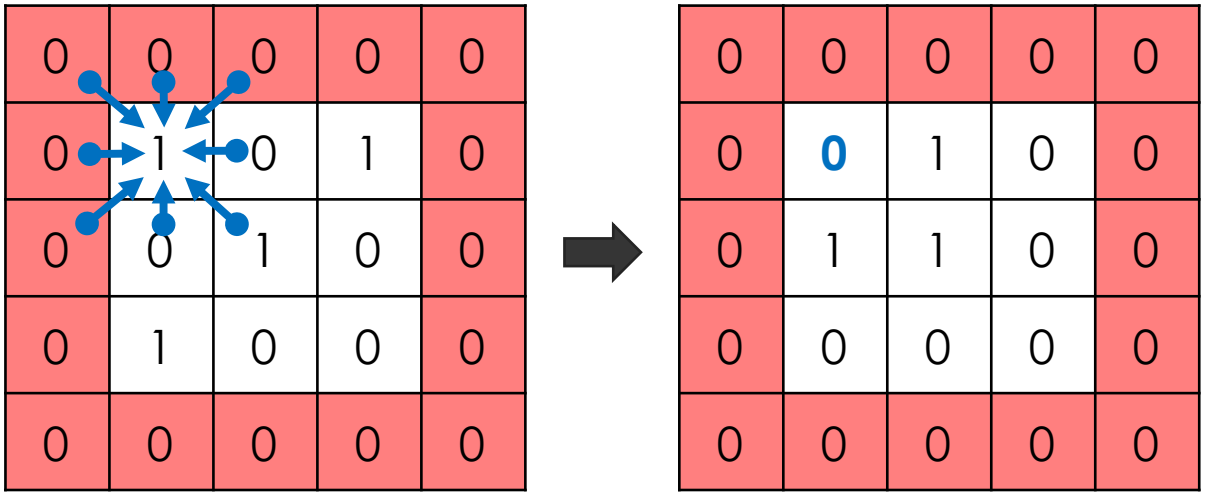



Figure 29.1: Stencil update associated with Conway's Game of Life.

We use two grids, one for the previous timestep and one for the current one. We use *ghost regions* at the edges of the grids, extending each edge by an extra point, to avoid having to do separate computations at the boundaries.

The following constructs three-dimensional grids of size $xdim \times ydim \times zdim$, with ghost regions:

```
rectdomain<3> domain{ pt(-1, -1, -1), pt(xdim+1, ydim+1, zdim+1) };
rectdomain<3> interior{ pt(0, 0, 0), pt(xdim, ydim, zdim) };
ndarray<double, 3> gridA(domain);
ndarray<double, 3> gridB(domain);
```

We initialize the grids as needed and then perform an iterative stencil computation as follows:

```
void probe(ndarray<double, 3> *gridA_ptr,
           ndarray<double, 3> *gridB_ptr,
```

(continues on next page)

(continued from previous page)

```
          const rectdomain<3> &interior, int steps) {
  for (int i = 0; i < steps; i++) {
    ndarray<double, 3> &gridA = *gridA_ptr;
    ndarray<double, 3> &gridB = *gridB_ptr;

    for (auto p : interior) {
      gridB[p] =
        gridA[p + pt( 0,  0,  1)] +
        gridA[p + pt( 0,  0, -1)] +
        gridA[p + pt( 0,  1,  0)] +
        gridA[p + pt( 0, -1,  0)] +
        gridA[p + pt( 1,  0,  0)] +
        gridA[p + pt(-1,  0,  0)] +
        WEIGHT * gridA[p];
    }

    // Swap pointers
    std::swap(gridA_ptr, gridB_ptr);
  }
}
```

We make use of iteration over a rectangular domain, arithmetic over points, and using points to index into the multidimensional array. At the end of each timestep, we swap which grid is the current and which is the previous.

While this code is simple to write, it does not perform well on many compilers. The linearized iteration over the rectangular domain can prevent a compiler from optimizing the iteration order to make the best use of the memory hierarchy, such as with a polyhedral analysis. In GCC, for example, we find that a nested loop structure such as the following can be five times more efficient:

```
for (p[0] = interior.lwb[0]; p[0] < interior.upb[0]; p[0]++) {
  for (p[1] = interior.lwb[1]; p[1] < interior.upb[1]; p[1]++) {
    for (p[2] = interior.lwb[2]; p[2] < interior.upb[2]; p[2]++) {
      gridB[p] =
        gridA[p + pt( 0,  0,  1)] +
        gridA[p + pt( 0,  0, -1)] +
        gridA[p + pt( 0,  1,  0)] +
        gridA[p + pt( 0, -1,  0)] +
        gridA[p + pt( 1,  0,  0)] +
        gridA[p + pt(-1,  0,  0)] +
        WEIGHT * gridA[p];
    }
  }
}
```

This code is less simple, and it introduces a further dependency on the dimensionality of the grid, preventing us from generalizing it to an arbitrary number of dimensions.

## 29.5 Nested Iteration

In order to solve the problem of linearized iteration, we can use metaprogramming to turn what appears to be a single loop into a nested one, making it more amenable to analysis and optimization. We start by writing a recursive template that introduces a loop nest at each level of the recursion:

```
template <int N>
struct rdloop {
  // Performs a nested loop over the set of loop indices in [lwb,
  // upb). The size of lwb and upb must be at least N. For each
  // index i1, ..., iN in [lwb, upb), calls func on the point
  // pt(is..., i1, ..., iN).
  template <class Func, class... Indices>
  static void loop(const Func &func, const int *lwb,
                   const int *upb, Indices... is) {
    for (int i = *lwb; i < *upb; i++) {
      rdloop<N-1>::loop(func, lwb+1, upb+1, is..., i);
    }
  }
};
```

We write our template as a class, since we will require a base case and would need partial function-template specialization, which is not supported by C++, to implement it purely with function templates. The class is parameterized by the dimensionality. Within the class is a single static member function template that is parameterized by a functor type and a variadic set of indices. The arguments to the function itself are a functor object, which will be applied in the innermost loop, lower and upper bounds for the remaining dimensions, and the set of indices computed so far.

The body introduces a new loop nest, using the lower and upper bounds, and recursively applies the template with one less dimension. The bound pointers are adjusted for the new dimension, and we pass the input indices along with the one for this dimension in the recursive call. Our base case, where there is only a single dimension, is then as follows:

```
template <>
struct rdloop<1> {
  template <class Func, class... Indices>
  static void loop(const Func &func, const int *lwb,
                   const int *upb, Indices... is) {
    for (int i = *lwb; i < *upb; i++) {
      func(pt(is..., i));
    }
  }
};
```

We construct a point from the collected set of indices from each dimension and then call the functor object on that point.

Now that we have a mechanism for constructing a set of nested loops, we start the recursion from a function object and domain as follows:

```
rdloop<N>::loop(func, domain.lwb.coords,
                domain.upb.coords);
```

In order to actually make use of this, we provide a loop abstraction as follows:

```
foreach (p, interior) {
  gridB[p] =
    gridA[p + pt( 0,  0,  1)] +
    gridA[p + pt( 0,  0, -1)] +
    gridA[p + pt( 0,  1,  0)] +
    gridA[p + pt( 0, -1,  0)] +
    gridA[p + pt( 1,  0,  0)] +
    gridA[p + pt(-1,  0,  0)] +
    WEIGHT * gridA[p];
};
```

We have the `foreach` keyword, which we will define shortly, that takes in a variable name to represent a point and the domain over which to iterate. We then have a loop body that uses the point variable. A semicolon appears after the body, and it is necessary due to how `foreach` is defined.

The loop body looks very much like the body of a lambda function, and since we require a function object in order to build the nested structure, it is natural to consider how we can arrange for the loop body to turn into a lambda function. We need a statement in which a lambda function can appear at the end, right before the terminating semicolon, and assignment fits this structure:

```
<var> = [<capture>](<parameters>) {
  <body>
};
```

Thus, we need to arrange for the `foreach` header to turn into the beginning of this statement:

```
<var> = [<capture>](<parameters>)
```

We would like the programmer to be able to use all local variables, so we should capture all variables by reference. The `foreach` also introduces a new variable for the point, so that should be in the parameter list:

```
<var> = [&](const point<N> &<name>)
```

There are several remaining things we need. First, we need to figure out the dimensionality of the point to use as the parameter. We can use `decltype` to do so from the domain:

```
<var> = [&](const decltype(<domain>.lwb) &<name>)
```

Second, we need a way to ensure that when this assignment happens, the nested loop structure is executed. We can do so by overloading the assignment operator of the object `<var>`. Finally, we also need to introduce the left-hand variable, preferably in its own scope. We can do both by introducing a dummy loop header:

```
#define foreach(p, dom)                                              \
  for (auto _iter = (dom).iter(); !_iter.done; _iter.done = true) \
    _iter = [&](const decltype((dom).lwb) &p)
```

In order for this to work, we need the `iter()` method on a domain to give us an object whose assignment operator takes in a functor. This operator would then call the functor within a nested set of loops. The object also needs a `done` field in order to ensure the dummy loop executes exactly one iteration. We can add the following members to the `rectdomain` template:

```
template <int N>
struct rectdomain {
  ...
```

(continues on next page)

(continued from previous page)

```
  struct fast_iter {
    const rectdomain &domain; // domain over which to iterate
    bool done;                // whether or not this loop has run

    // Constructs a fast_iter with the given domain.
    fast_iter(const rectdomain &dom)
      : domain(dom), done(false) {}

    // Loops over the associate domain, calling func on each point
    // in the domain.
    template <class Func>
    fast_iter &operator=(const Func &func) {
      rdloop<N>::loop(func, domain.lwb.coords,
                      domain.upb.coords);
      return *this;
    }
  };

  // Returns a fast_iter over this domain.
  fast_iter iter() const {
    return fast_iter(*this);
  }
};
```

The assignment operator of `fast_iter` is a template, taking in a functor object. It then uses our nested loop generation mechanism to generate a set of nested loops and call the functor from the innermost loop, with the appropriate point as the argument.

The result is a loop that has the simplicity of a range-based for loop but, depending on the compiler, the performance of a nested set of loops. As an example, with GCC 6.2 on the author's iMac computer, the range-based for loop takes 1.45 seconds to perform ten timesteps of the stencil above on a $256^3$ grid, while the nested loops and the `foreach` loop each take 0.28 seconds. This demonstrates the power of metaprogramming in order to extend the features of a language.

# Part VII

# Concurrent Programming

We now take a brief look at *concurrent programming*, where a program is structured so that several computations can execute concurrently during overlapping time periods. We focus on aspects of concurrency that are explicitly specified by a programmer, rather than the implicit concurrency provided by compiler optimizations or the underlying system hardware.

CHAPTER

# THIRTY

# PARALLEL COMPUTING

From the 1970s through the mid-2000s, the speed of individual processor cores grew at an exponential rate. Much of this increase in speed was accomplished by increasing the *clock frequency*, the rate at which a processor performs basic operations. In the mid-2000s, however, this exponential increase came to an abrupt end, due to power and thermal constraints, and the speed of individual processor cores has increased much more slowly since then. Figure 30.1 is graph from Stanford's CPU database that illustrates this trend:

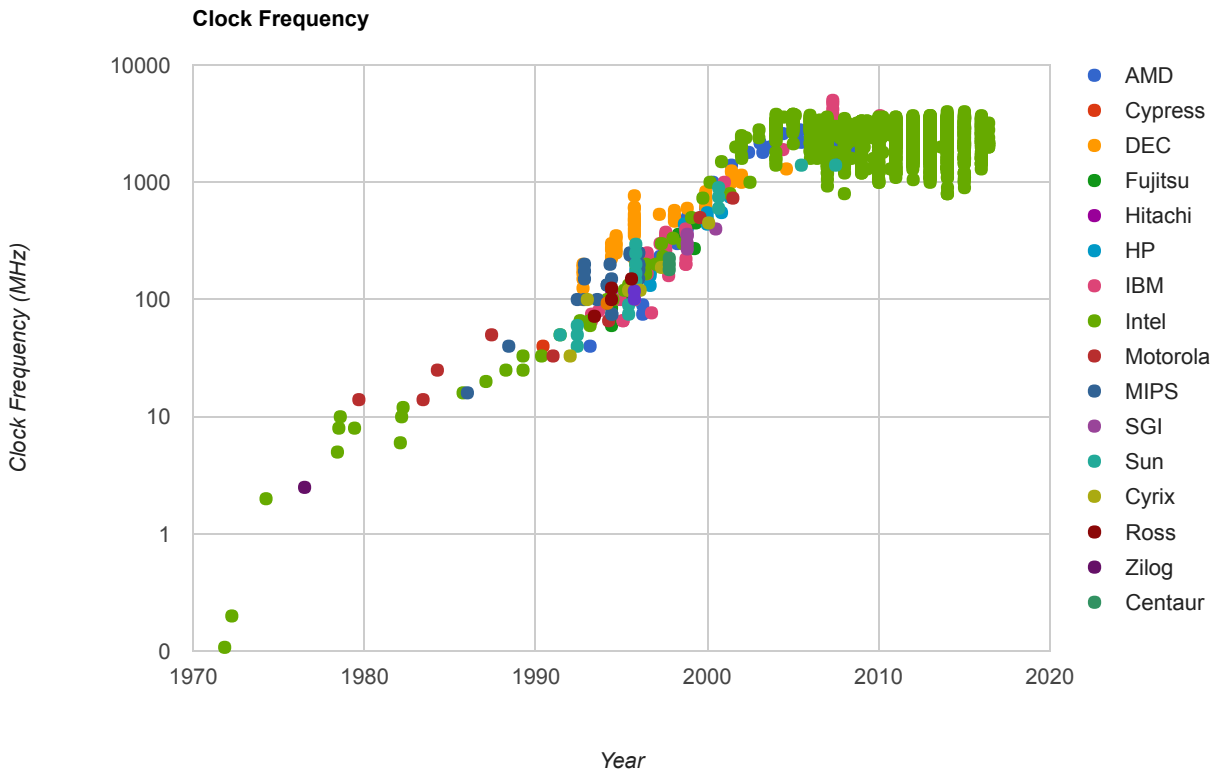



Figure 30.1: Historical data of CPU clock frequencies.

Instead of increasing clock frequency, CPU manufacturers began to place multiple cores in a single processor, enabling more operations to be performed concurrently.

Parallelism is not a new concept. Large-scale parallel machines have been used for decades, primarily for scientific computing and data analysis. Even in personal computers with a single processor core, operating systems and interpreters have provided the abstraction of concurrency. This is done through *context switching*, or rapidly switching between different tasks without waiting for them to complete. Thus, multiple programs can run on the same machine concurrently, even if it only has a single processing core.

Given the current trend of increasing the number of processor cores, individual applications must now take advantage of parallelism in order to run faster. Within a single program, computation must be arranged so that as much work can be done in parallel as possible. However, parallelism introduces new challenges in writing correct code, particularly in the presence of shared, mutable state.

For problems that can be solved efficiently in the functional model, with no shared mutable state, parallelism poses few problems. Pure functions provide *referential transparency*, meaning that expressions can be replaced with their values, and vice versa, without affecting the behavior of a program. This enables expressions that do not depend on each other to be evaluated in parallel. The MapReduce framework is one system that allows functional programs to be specified and run in parallel with minimal programmer effort. Several functional languages, including NESL and Clojure, have been designed with parallelism at their core.

Unfortunately, not all problems can be solved efficiently using functional programming. The Berkeley View project has identified thirteen common computational patterns in science and engineering, only one of which is MapReduce. The remaining patterns require shared state.

In the remainder of this section, we will see how mutable shared state can introduce bugs into parallel programs and a number of approaches to prevent such bugs. We will examine these techniques in the context of two applications, a web crawler and a particle simulator.

## 30.1 Parallelism in Python

Before we dive deeper into the details of parallelism, let us first explore Python's support for parallel computation. Python provides two means of parallel execution: threading and multiprocessing.

### 30.1.1 Threading

In *threading*, multiple "threads" of execution exist within a single interpreter. Each thread executes code independently from the others, though they share the same data. However, the CPython interpreter, the main implementation of Python, only interprets code in one thread at a time, switching between them in order to provide the illusion of parallelism. On the other hand, operations external to the interpreter, such as writing to a file or accessing the network, may run in parallel.

The `threading` module contains classes that enable threads to be created and synchronized. The following is a simple example of a multithreaded program:

```
import threading

def thread_hello():
    other = threading.Thread(target=thread_say_hello, args=())
    other.start()
    thread_say_hello()

def thread_say_hello():
    print('hello from', threading.current_thread().name)
```

```
>>> thread_hello()
hello from Thread-1
hello from MainThread
```

The `Thread` constructor creates a new thread. It requires a target function that the new thread should run, as well as the arguments to that function. Calling `start` on a `Thread` object marks it ready to run. The `current_thread` function returns the `Thread` object associated with the current thread of execution.

In this example, the prints can happen in any order, since we haven't synchronized them in any way. The output can even be interleaved on some systems.

### 30.1.2 Multiprocessing

Python also supports *multiprocessing*, which allows a program to spawn multiple interpreters, or *processes*, each of which can run code independently. These processes do not generally share data, so any shared state must be communicated between processes. On the other hand, processes execute in parallel according to the level of parallelism provided by the underlying operating system and hardware. Thus, if the CPU has multiple processor cores, Python processes can truly run concurrently.

The `multiprocessing` module contains classes for creating and synchronizing processes. The following is the hello example using processes:

```
import multiprocessing

def process_hello():
    other = multiprocessing.Process(target=process_say_hello,
                                    args=())
    other.start()
    process_say_hello()

def process_say_hello():
    print('hello from', multiprocessing.current_process().name)
```

```
>>> process_hello()
hello from MainProcess
>>> hello from Process-1
```

As this example demonstrates, many of the classes and functions in `multiprocessing` are analogous to those in `threading`. This example also demonstrates how lack of synchronization affects shared state, as the display can be considered shared state. Here, the interpreter prompt from the interactive process appears before the print output from the other process.

## 30.2 The Problem with Shared State

To further illustrate the problem with shared state, let's look at a simple example of a counter that is shared between two threads:

```
import threading
from time import sleep

counter = [0]     # store in a list to avoid global statements

def increment():
    count = counter[0]
    counter[0] = count + 1

other = threading.Thread(target=increment, args=())
other.start()
increment()
print('count is now: ', counter[0])
```

In this program, two threads attempt to increment the same counter. The CPython interpreter can switch between threads at almost any time. Only the most basic operations are *atomic*, meaning that they appear to occur instantly, with no switch possible during their evaluation or execution. Incrementing a counter requires multiple basic operations: read the old value, add one to it, and write the new value. The interpreter can switch threads between any of these operations.

In order to show what happens when the interpreter switches threads at the wrong time, we can attempt to force a switch by sleeping for 0 seconds:

```
from time import sleep

counter = [0]

def increment():
    count = counter[0]
    sleep(0) # try to force a switch to the other thread
    counter[0] = count + 1
```

When this code is run, the interpreter often does switch threads at the `sleep` call. This can result in the following sequence of operations:

```
Thread 0                    Thread 1
read counter[0]: 0
                            read counter[0]: 0
calculate 0 + 1: 1
write 1 -> counter[0]
                            calculate 0 + 1: 1
                            write 1 -> counter[0]
```

The end result is that the counter has a value of 1, even though it was incremented twice! Worse, the interpreter may only switch at the wrong time very rarely, making this difficult to debug. Even with the `sleep` call, this program sometimes produces a correct count of 2 and sometimes an incorrect count of 1.

This problem arises only in the presence of shared data that may be mutated by one thread while another thread accesses it. Such a conflict is called a *race condition*, and it is an example of a bug that only exists in the parallel world.

In order to avoid race conditions, shared data that may be mutated and accessed by multiple threads must be protected against concurrent access. For example, if we can ensure that thread 1 only accesses the counter after thread 0 finishes accessing it, or vice versa, we can guarantee that the right result is computed. We say that shared data is *synchronized* if it is protected from concurrent access. In the next few subsections, we will see multiple mechanisms providing synchronization.

## 30.3 When No Synchronization is Necessary

In some cases, access to shared data need not be synchronized, if concurrent access cannot result in incorrect behavior. The simplest example is read-only data. Since such data is never mutated, all threads will always read the same values regardless when they access the data.

In rare cases, shared data that is mutated may not require synchronization. However, understanding when this is the case requires a deep knowledge of how the interpreter and underlying software and hardware work. Consider the following example:

```
items = []
flag = []
```

(continues on next page)

(continued from previous page)

```
def consume():
    while not flag:
        pass
    print('items is', items)

def produce():
    for i in range(10):
        items.append(i)
    flag.append('go')

consumer = threading.Thread(target=consume, args=())
consumer.start()
produce()
```

Here, the producer thread adds items to `items`, while the consumer waits until `flag` is non-empty. When the producer finishes adding items, it adds an element to `flag`, allowing the consumer to proceed.

In most Python implementations, this example will work correctly. However, a common optimization in other compilers and interpreters, and even the hardware itself, is to reorder operations within a single thread that do not depend on each other for data. In such a system, the statement `flag.append('go')` may be moved before the loop, since neither depends on the other for data. In general, you should avoid code like this unless you are certain that the underlying system won't reorder the relevant operations.

## 30.4 Synchronized Data Structures

The simplest means of synchronizing shared data is to use a data structure that provides synchronized operations. The `queue` module contains a `Queue` class that provides synchronized first-in, first-out access to data. The `put` method adds an item to the `Queue` and the `get` method retrieves an item. The class itself ensures that these methods are synchronized, so items are not lost no matter how thread operations are interleaved. Here is a producer/consumer example that uses a `Queue`:

```
from queue import Queue

queue = Queue()

def synchronized_consume():
    while True:
        print('got an item:', queue.get())
        queue.task_done()

def synchronized_produce():
    for i in range(10):
        queue.put(i)
    queue.join()

consumer = threading.Thread(target=synchronized_consume, args=())
consumer.daemon = True
consumer.start()
synchronized_produce()
```

There are a few changes to this code, in addition to the `Queue` and `get` and `put` calls. We have marked the consumer thread as a *daemon*, which means that the program will not wait for that thread to complete before exiting. This allows

us to use an infinite loop in the consumer. However, we do need to ensure that the main thread exits, but only after all items have been consumed from the `Queue`. The consumer calls the `task_done` method to inform the `Queue` that it is done processing an item, and the main thread calls the `join` method, which waits until all items have been processed, ensuring that the program exits only after that is the case.

A more complex example that makes use of a `Queue` is a parallel *web crawler* that searches for dead links on a website.

## 30.5 Locks

When a synchronized version of a particular data structure is not available, we have to provide our own synchronization. A *lock* is a basic mechanism to do so. It can be *acquired* by at most one thread, after which no other thread may acquire it until it is *released* by the thread that previously acquired it.

In Python, the `threading` module contains a `Lock` class to provide locking. A `Lock` has `acquire` and `release` methods to acquire and release the lock, and the class guarantees that only one thread at a time can acquire it. All other threads that attempt to acquire a lock while it is already being held are forced to wait until it is released.

For a lock to protect a particular set of data, all the threads need to be programmed to follow a rule: no thread will access any of the shared data unless it owns that particular lock. In effect, all the threads need to "wrap" their manipulation of the shared data in `acquire` and `release` calls for that lock.

The following is an example of two threads incrementing a counter that is protected by a lock, avoiding a race condition:

```
from threading import Thread, Lock

counter = [0]
counter_lock = Lock()

def increment():
    counter_lock.acquire()
    count = counter[0]
    counter[0] = count + 1
    counter_lock.release()

other = Thread(target=increment, args=())
other.start()
increment()
other.join()
print('count is now', counter[0])
```

Acquiring the lock prevents another thread from acquiring it and proceeding to increment the counter. When the lock has been acquired, the thread can be assured that no other thread can enter the *critical section* that is protected by the lock. Once the thread has incremented the counter, it releases the lock so that another thread can access the counter.

In this code, we had to be careful not to return until after we released the lock. In general, we have to ensure that we release a lock when we no longer need it. This can be very error-prone, particularly in the presence of exceptions, so Python locks are context managers that can be used with *scope-based resource management*:

```
def increment():
    with counter_lock:
      count = counter[0]
      counter[0] = count + 1
```

The `with` statement ensures that `counter_lock` is acquired before its suite is executed and that it is released when the suite is exited for any reason.

Operations that must be synchronized with each other must use the same lock. However, two disjoint sets of operations that must be synchronized only with operations in the same set should use two different lock objects to avoid *over-synchronization*.

## 30.6 Barriers

Another way to avoid conflicting access to shared data is to divide a program into phases, ensuring that shared data is mutated in a phase in which no other thread accesses it. A *barrier* divides a program into phases by requiring all threads to reach it before any of them can proceed. Code that is executed after a barrier cannot be concurrent with code executed before the barrier.

In Python, the `threading` module provides a barrier in the form of the the `wait` method of a `Barrier` instance:

```
counters = [0, 0]
barrier = threading.Barrier(2)

def count(thread_num, steps):
    for i in range(steps):
        other = counters[1 - thread_num]
        barrier.wait() # wait for reads to complete
        counters[thread_num] = other + 1
        barrier.wait() # wait for writes to complete

def threaded_count(steps):
    other = threading.Thread(target=count, args=(1, steps))
    other.start()
    count(0, steps)
    print('counters:', counters)

threaded_count(10)
```

In this example, reading and writing to shared data take place in different phases, separated by barriers. The writes occur in the same phase, but they are disjoint; this disjointness is necessary to avoid concurrent writes to the same data in the same phase. Since this code is properly synchronized, both counters will always be 10 at the end.

## 30.7 Message Passing

A final mechanism to avoid improper mutation of shared data is to entirely avoid concurrent access to the same data. In Python, using multiprocessing rather than threading naturally results in this, since processes run in separate interpreters with their own data. Any state required by multiple processes can be communicated by passing messages between processes.

The `Pipe` function in the `multiprocessing` module constructs a communication channel between processes, returning a pair of connection endpoints. By default, the connection is duplex, meaning a two-way channel, though passing in the argument `False` results in a one-way channel. The `send` method on a connection sends an object over the channel, while the `recv` method receives an object. The latter is *blocking*, meaning that a process that calls `recv` will wait until an object is received.

The following is a producer/consumer example using processes and pipes:

```
def process_consume(in_pipe):
    while True:
```

(continues on next page)

(continued from previous page)

```
        item = in_pipe.recv()
        if item is None:
            return
        print('got an item:', item)

def process_produce(out_pipe):
    for i in range(10):
        out_pipe.send(i)
    out_pipe.send(None) # done signal

pipe = multiprocessing.Pipe(False)
consumer = multiprocessing.Process(target=process_consume,
                                   args=(pipe[0],))
consumer.start()
process_produce(pipe[1])
```

The two ends of the pipe are obtained by indexing into the result of `Pipe()`. Since the pipe is created as a one-way channel, the sender must use the end at index 1 and the receiver the end at index 2.

In this example, we use a `None` message to signal the end of communication. We also passed in one end of the pipe as an argument to the target function when creating the consumer process. This is necessary, since state must be explicitly shared between processes.

The `multiprocessing` module provides other synchronization mechanisms for processes, including synchronized queues, locks, and as of Python 3.3, barriers. For example, a lock or a barrier can be used to synchronize printing to the screen, avoiding the improper display output we saw previously.

## 30.8 Application Examples

We now examine two application examples in more detail, exploring how the techniques above can be used to properly synchronize access to shared resources.

### 30.8.1 Web Crawler

A *web crawler* is a program that systematically browses the Internet. Such a program may have several uses; one example is a crawler that validates links on a website, recursively checking that all links hosted by the site are to valid webpages. This crawler could be implemented with a work queue of URLs that need to be recursively checked and a set of URLs that have already been encountered by the program. Then for each URL in the work queue, the program would:

1. Load the webpage, parsing it for outgoing links.
2. For each link on the page:
    a) Check if the link has already been seen.
    b) If the link has not been seen, then add it to both the seen set and the work queue.

Since Python threading enables network requests to be serviced concurrently, this program can be parallelized by using several threads to process different URLs. However, the shared queue and set data structures must be protected from concurrent access.

The work queue can be represented using the synchronized `Queue` class, since it ensures that no more than one thread can perform an operation on the `Queue` at a time. However, Python does not provide a synchronized set, so we must use a lock to protect access to a normal set:

```
seen = set()
seen_lock = threading.Lock()

def already_seen(item):
    with seen_lock:
        if item not in seen:
            seen.add(item)
            return False
        return True
```

A lock is necessary here, in order to prevent another thread from adding the URL to the set between this thread checking if it is in the set and adding it to the set. Furthermore, adding to a set is not atomic, so concurrent attempts to add to a set may corrupt its internal data. The `already_seen()` function adds the given item to the set if it is not already in there, returning whether or not the item was added.

The following then checks if a URL has been seen and adds it to the work queue if not:

```
work_queue = Queue()

def queue_url(url):
    if not already_seen(url):
        work_queue.put(url)
```

The call to `already_seen()` ensures that a given URL has not been seen when it is added to the work queue, so that the URL is only processed once.

### 30.8.2 Particle Simulator

A particle simulator simulates the interactions between independent particles within a confined space. Each particle interacts with every other particle; for example, molecules may apply a repulsive force to other molecules based on the distance between them, resulting from the electric field of the electrons in each molecule. This interaction can be can be computed over the course of many discrete timesteps. A particle has a position, velocity, and acceleration, and a new acceleration is computed in each timestep based on the positions of the other particles. The velocity of the particle must be updated accordingly, and its position according to its velocity.

A natural way to parallelize a particle simulator is to divide the particles among several threads or processes, as illustrated in Figure 30.2.

Each thread or process is then responsible for computing the forces on its own particles, updating their positions and velocities accordingly. The algorithm for a single timestep on each thread can then be divided into the following phases:

1. Read the current position of every particle.
2. For each of its own particles, compute the force resulting from interactions with every other particle, using their current positions.
3. Update the velocities of its particles based on the forces computed.
4. Update the positions of its particles based on the new velocities.

In this algorithm, the positions of the particles constitute shared data and must be protected from concurrent access. The multithreaded implementation of the simulator uses barriers to separate phases 1 and 4, which access the shared data. Two barriers are required, one to ensure that all threads move together between phase 1 and 4 within a timestep,

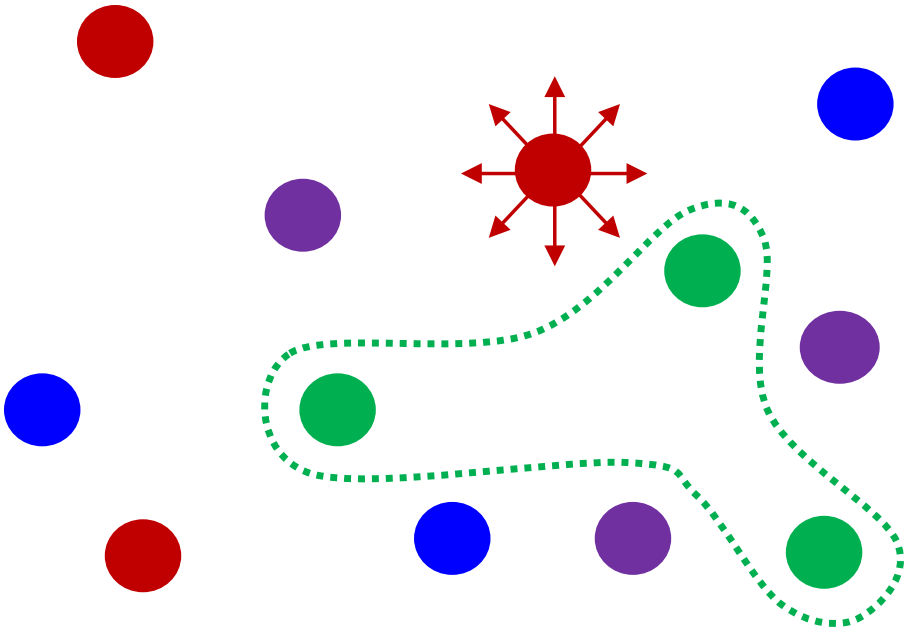


Figure 30.2: A particle interaction can be parallelized by splitting the particles among the computational units.

and another to ensure that they synchronously move between phase 4 in a timestep to phase 1 in the next timestep. The writes in phases 2 and 3 are to separate data on each thread, so they need not be synchronized.

An alternative algorithm is to use message passing to send copies of particle positions to other threads or processes. This is the strategy implemented by the multiprocess version of the particle simulator, with pipes used to communicate particle positions between processes in each timestep. A circular pipeline is set up between processes in order to minimize communication. Each process injects its own particles' positions into its pipeline stage, which eventually go through a full rotation of the pipeline, as shown in Figure 30.3.

At each step of the rotation, a process applies forces from the positions that are currently in its own pipeline stage on to its own particles, so that after a full rotation, all forces have been applied to its particles.

## 30.9 Synchronization Pitfalls

While synchronization methods are effective for protecting shared state, they can also be used incorrectly, failing to accomplish the proper synchronization, over-synchronizing, or causing the program to hang as a result of deadlock.

### 30.9.1 Under-synchronization

A common pitfall in parallel computing is to neglect to properly synchronize shared accesses. In the set example for the *web crawler*, we need to synchronize the membership check and insertion together, so that another thread cannot perform an insertion in between these two operations. Failing to synchronize the two operations together is erroneous, even if they are separately synchronized:

```
def already_seen(item):
    with seen_lock:
        present = item in seen
    if not present
        with seen_lock:
```

(continues on next page)

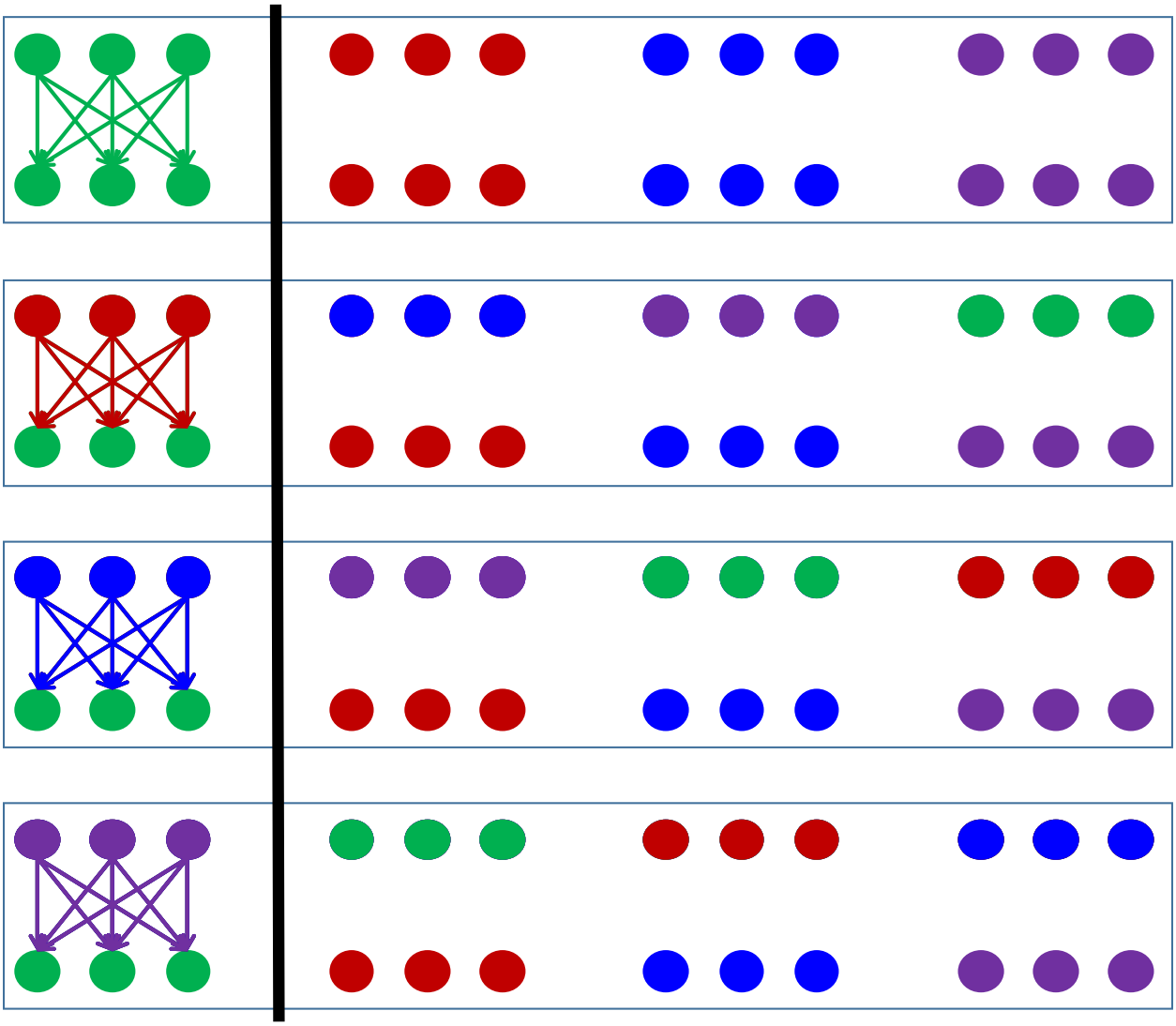


Figure 30.3: Copies of each particle can be rotated among the processes. A process computes the interaction between its own particles and the copies it sees in each step of the rotation.

(continued from previous page)

```
            seen.add(item)
    return not present
```

Here, it is possible for one thread to acquire `seen_lock` and see that the item is not in the set. But between releasing the lock and requiring it for insertion, another thread can obtain the lock and also see that the item is not in the set. This results in both threads thinking that they inserted the item, potentially resulting in duplicate work.

### 30.9.2 Over-synchronization

Another common error is to over-synchronize a program, so that non-conflicting operations cannot occur concurrently. As a trivial example, we can avoid all conflicting access to shared data by acquiring a master lock when a thread starts and only releasing it when a thread completes. This serializes our entire code, so that nothing runs in parallel. In some cases, this can even cause our program to hang indefinitely. For example, consider a consumer/producer program in which the consumer obtains the lock and never releases it:

```
items = []
lock = Lock()

def consume():
    with lock:
        while not items:
            sleep(1)    # wait for a bit
        print('got an item:', items.pop())

def synchronized_produce():
    with lock:
        for i in range(10):
            items.append(i)
```

This prevents the producer from producing any items, which in turn prevents the consumer from doing anything since it has nothing to consume.

While this example is trivial, in practice, programmers often over-synchronize their code to some degree, preventing their code from taking complete advantage of the available parallelism.

### 30.9.3 Deadlock

Because they cause threads or processes to wait on each other, synchronization mechanisms are vulnerable to *deadlock*, a situation in which two or more threads or processes are stuck, waiting for each other to finish. We have just seen how neglecting to release a lock can cause a thread to get stuck indefinitely. But even if threads or processes do properly release locks, programs can still reach deadlock.

The source of deadlock is a *circular wait*, illustrated in Figure 30.4 with processes. A process cannot continue because it is waiting for other processes, which are in turn waiting for the first process to complete.

As an example, we will set up a deadlock with two processes. Suppose they share a duplex pipe and attempt to communicate with each other as follows:

```
def deadlock(in_pipe, out_pipe):
    item = in_pipe.recv()
    print('got an item:', item)
    out_pipe.send(item + 1)
```

(continues on next page)

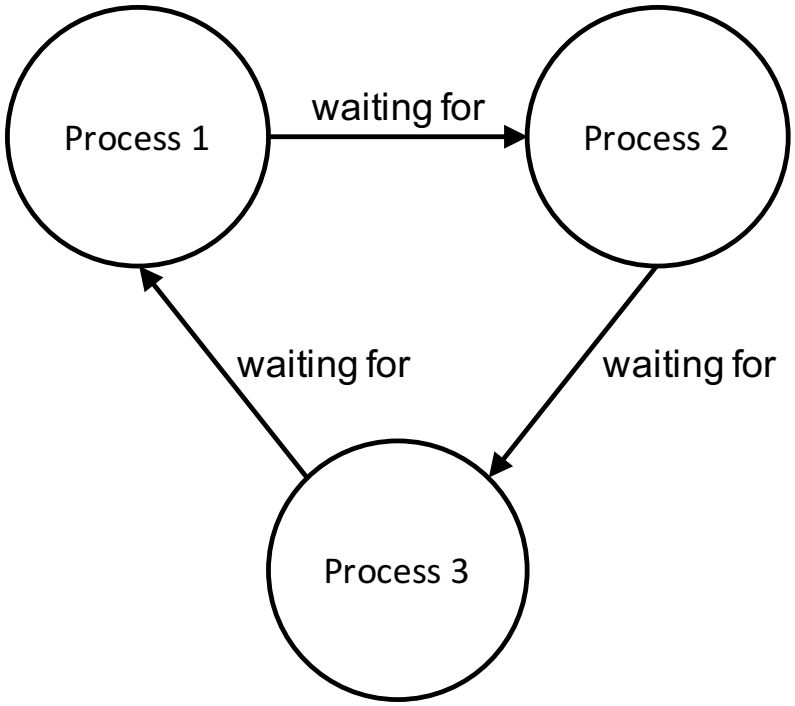



Figure 30.4: Deadlock arises when a set of threads or processes is each waiting on another thread or process.

(continued from previous page)

```
def create_deadlock():
    pipe = multiprocessing.Pipe()
    other = multiprocessing.Process(target=deadlock,
                                    args=(pipe[0], pipe[1]))
    other.start()
    deadlock(pipe[1], pipe[0])

create_deadlock()
```

Both processes attempt to receive data first. Recall that the `recv` method blocks until an item is available. Since neither process has sent anything, both will wait indefinitely for the other to send it data, resulting in deadlock.

Synchronization operations must be properly aligned to avoid deadlock. This may require sending over a pipe before receiving, acquiring multiple locks in the same order, and ensuring that all threads reach the right barrier at the right time.

## 30.10 Conclusion

As we have seen, parallelism presents new challenges in writing correct and efficient code. As the trend of increasing parallelism at the hardware level will continue for the foreseeable future, parallel computation will become more and more important in application programming. There is a very active body of research on making parallelism easier and less error-prone for programmers. Our discussion here serves only as a basic introduction to this crucial area of computer science.

CHAPTER

# THIRTYONE

# ASYNCHRONOUS TASKS

In parallelizing a computation, one strategy is to explicitly decompose a program over the set of workers, as we did in the previous section. Another option is to divide the work according to the natural granularity of an operation and to rely on the runtime system to schedule the work appropriately. This latter strategy can be accomplished with *asynchronous tasks*, where an operation is launched to be computed asynchronously, and its result used at some further point.

In C++11, an asynchronous task can be launched with the `async()` function template, contained in the `<future>` header. The first argument to `async()` is the function or function object representing a task, and the remaining arguments are the arguments with which to invoke that function. The following is a basic example:

```
void foo(int x, int y) {
  cout << (x + y) << endl;
}

int main() {
  async(foo, 3, 4);
  async(foo, 5, 6);
}
```

The code above launches separate tasks to compute `foo(3, 4)` and `foo(5, 6)` asynchronously. The print outputs 7 and 11 can appear in any order, since the two tasks aren't synchronized with respect to each other, and the outputs can even be interleaved with each other.

The return value of `async()` is a *future* object, which is a proxy for the result of the asynchronous task. In particular, the `async()` calls above return objects of type `future<void>`, since the return type of `foo()` is `void`. We can wait on the result of an asynchronous task by calling the `wait()` method of the corresponding `future` object, as in the following:

```
int main() {
  future<void> f1 = async(foo, 3, 4);
  f1.wait();
  future<void> f2 = async(foo, 5, 6);
  f2.wait();
}
```

Here, we wait for the first task to complete before launching the second. This ensures that the 7 will appear as output before the 11.

In the case of a function that returns a non-void value, we can also obtain the result by calling the `get()` method of the `future` object, which waits until the result is available and then returns the result. This is particularly useful if we have some computation that depends on the result of the task, as in the following:

```
int main() {
  future<int> f1 = async([](int x, int y) {
      return x + y;
    }, 3, 4);
  cout << (f1.get() + 5) << endl;
}
```

This launches a task to asynchronously call a lambda function, waits for the result and adds 5 to it, and prints the sum.

As a more complex example, let's consider the tree-recursive computation of the Fibonacci sequence. The following is a sequential function to compute a Fibonacci number:

```
long long fib(int n) {
  if (n <= 1)
    return n;
  return fib(n - 1) + fib(n - 2);
}
```

We can observe that the two recursive calls do not depend on each other, so we can compute them asynchronously by launching a separate task for one of the calls. The following code does so:

```
long long async_fib(int n) {
  if (n <= 1)
    return n;
  future<long long> res1 = async(async_fib, n - 1);
  long long res2 = async_fib(n - 2);
  return res2 + res1.get();
}
```

This code uses `async()` to compute one recursive call, while the other call is computed in the existing task. We require the result of the asynchronous task before we can compute the sum and return, so we use `get()` on its `future` object in order to obtain its result.

As an aside, we write the two recursive calls in separate statements to ensure that the asynchronous task is launched before the recursive call that takes place in the existing task. Consider the following version that makes both calls in the same statement:

```
return async(async_fib, n - 1).get() + async_fib(n - 2);
```

In C++, the order of evaluation of the two operands to + is unspecified, so it would be valid for the compiler to produce code that sequentially computes the right-hand side before launching the asynchronous task to compute the left-hand side. This would turn the whole computation into a sequential one. Thus, we need to use statement sequencing to ensure that the asynchronous task is launched before the sequential recursive call is made.

## 31.1 Limiting the Number of Tasks

Most implementations of C++ that execute tasks in parallel do so with the use of an internal *thread pool*, scheduling the tasks among the available threads in the pool. There is significant overhead to computing a function with a task, as it needs go through the scheduling system and then be dispatched to a thread. As such, we often need to limit the granularity of our tasks to be large enough to amortize this overhead, as well as to reduce the number of tasks to limit the total overhead.

As an example, computing `async_fib(15)` on the author's quad-core iMac computer takes about 4000 times longer than `fib(15)`, using Clang 8, due to the large number of small tasks that are launched. Instead, we need to rewrite

`async_fib()` to do the remaining computation sequentially when a threshold is reached. The following does so, using the number of tasks launched so far to determine if the threshold has been met:

```
long long async_fib(int n, int tasks, int max_tasks) {
  if (n <= 1)
    return n;
  if (tasks < max_tasks) {
    future<long long> res1 = async(async_fib, n - 1, 2 * tasks,
                                   max_tasks);
    long long res2 = async_fib(n - 2, 2 * tasks, max_tasks);
    return res2 + res1.get();
  } else {
    return fib(n - 1) + fib(n - 2);
  }
}
```

The function takes in two extra arguments, representing the current number of tasks and the threshold value for the maximum number of tasks. If the threshold has not been reached, then the recursion proceeds as before, launching a new asynchronous task for one of the calls. In making the recursive calls, we double the number of current tasks to account for the fact that each step of the recursion doubles the number of concurrent computations. On the other hand, if the threshold has been reached, then we do the rest of the computation sequentially by calling `fib()`. Figure 31.1 is the task graph for computing `async_fib(5, 1, 4)`, limiting the number of tasks to four.

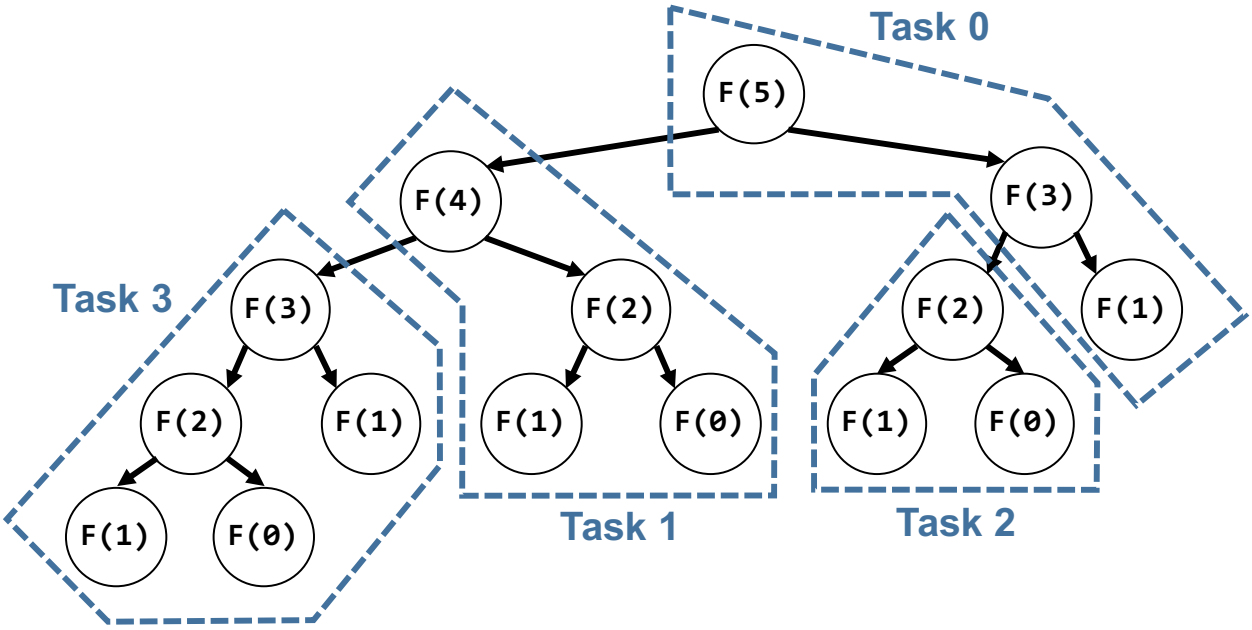



Figure 31.1: Task graph for computing `async_fib(5, 1, 4)`.

With the ability to limit the number of tasks, we find that `fib(42)` takes 1.63 seconds on the author's quad-core iMac, whereas `async_fib(42, 1, 512)` takes 0.47 seconds, about a 3.5x speedup. The 512-task limit was determined experimentally to be close to the optimal value.

As another example, let's write quicksort using asynchronous tasks. First, we write the sequential version as follows:

```
size_t partition(int *A, size_t size) {
  int pivot = A[0];
```

(continues on next page)

(continued from previous page)

```
  size_t start = 1;
  size_t end = size - 1;
  while (start <= end) {
    if (A[start] >= pivot)
      std::swap(A[start], A[end--]);
    else {
      std::swap(A[start - 1], A[start]);
      start++;
    }
  }
  return start - 1;
}

void quicksort(int *A, size_t size) {
  if (size <= CUTOFF) {
    std::sort(A, A + size);
    return;
  }
  int pivot = A[size/2];
  std::swap(A[0], A[size/2]);
  size_t pivot_index = partition(A, size);
  quicksort(A, pivot_index);
  quicksort(A + pivot_index + 1, size - pivot_index - 1);
}
```

This implements an in-place quicksort, partitioning the input array by swapping elements to the appropriate side of the pivot. We cut off the quicksort itself once we reach a small number of elements, since at that point other sorts such as insertion sort are more efficient. For simplicity, we use `std::sort()` when we reach the cutoff point, which will be 10 elements in our examples.

As with the Fibonacci sequence, we can launch a separate task to compute one of the recursive calls, limiting ourselves to a maximum number of tasks:

```
void async_quicksort(int *A, size_t size, int thread_count,
                     int max_tasks) {
  if (size <= CUTOFF) {
    std::sort(A, A + size);
    return;
  }
  int pivot = A[size/2];
  std::swap(A[0], A[size/2]);
  size_t pivot_index = partition(A, size);
  if (thread_count < max_tasks) {
    future<void> rec1 = async(async_quicksort, A, pivot_index,
                              2 * thread_count, max_tasks);
    async_quicksort(A + pivot_index + 1, size - pivot_index - 1,
                    2 * thread_count, max_tasks);
    rec1.wait();
  } else {
    quicksort(A, pivot_index);
    quicksort(A + pivot_index + 1, size - pivot_index - 1);
  }
}
```

In order to ensure that the asynchronous recursive call completes before returning, we call `wait()` on its associated `future` object. Sorting ten million elements with sequential `quicksort()` takes 0.93 seconds on the author's iMac, while sorting with `async_quicksort()` takes 0.35 seconds with the task limit at 128.

## 31.2 Launch Policy

By default, launching an asynchronous task does not require it to be immediately run in another thread. Rather, it merely allows the task to be run concurrently. Equally valid semantically is to defer execution of the task until the `wait()` or `get()` method is called on the associated `future` object, obtaining lazy evaluation of the task.

We can explicitly specify whether the task should be run in a different thread or deferred until its completion is required. We do so by specifying `std::launch::async` or `std::launch::deferred` as the first argument to `async()`, before the function to be run:

```
async(std::launch::async, async_fib, n - 1, 2 * tasks, max_tasks)
```

Without the policy specifier, the implementation is free to follow either launch policy.

We can use the `std::launch::async` policy to partition work over a fixed set of computational resources, as in multithreading. As an example, the we can estimate the value of $pi$ by choosing random points in the range $[(0,0),(1,1)]$ and determining whether the point lies in the upper-right quadrant of the unit circle, as illustrated by the shaded area in Figure 31.2.

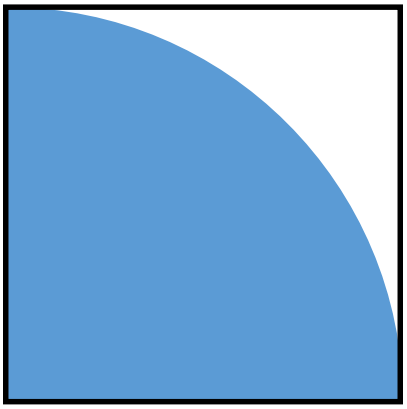


Figure 31.2: The value of $pi$ can be estimated by generating random points in $[(0,0),(1,1)]$ and counting how many lie within the upper-right quadrant of the unit circle.

The ratio of samples within the circle to total samples approximates $\frac{\pi}{4}$, the ratio of the area of a quadrant of the unit circle to the area of a unit square. The following sequential function implements this algorithm:

```
double compute_pi(size_t samples) {
  default_random_engine generator;
  uniform_real_distribution<> dist(0.0, 1.0);
  size_t count = 0;
  for (size_t i = 0; i < samples; i++) {
    double x = dist(generator), y = dist(generator);
    if (x * x + y * y <= 1.0)
      count++;
  }
  return 4.0 * count / samples;
}
```

We use the default random-generation engine from the <random> header, along with a uniform distribution of real numbers between 0.0 and 1.0. Run sequentially for 100 million samples, the computation takes 1.86 seconds on the author's iMac computer.

We can parallelize the computation over a fixed set of threads with the following:

```
double async_compute_pi(size_t samples, size_t num_workers) {
  future<double> *results = new future<double>[num_workers];
  for (size_t i = 0; i < num_workers; i++) {
    results[i] = async(std::launch::async,
                       compute_pi, samples / num_workers);
  }
  double total = 0;
  for (size_t i = 0; i < num_workers; i++) {
    total += results[i].get();
  }
  delete[] results;
  return total / num_workers;
}
```

Here, we construct a new task for each worker, launching it on a new thread using the `async::launch::async` policy. The main thread then waits on each worker thread in turn, accumulating the results from each worker. On the author's quad-core iMac, the computation takes 0.95 seconds for 100 million total samples with two worker threads, and 0.52 seconds with four worker threads. The latter is a speedup of about 3.6x over the sequential computation.

# Part VIII

# About

CHAPTER

# THIRTYTWO

# ABOUT

This text is based on many resources, including the classic textbook *Structure and Interpretation of Computer Programs* (SICP) by Abelson and Sussman, its Python adaptation *Composing Programs* by DeNero et al (available here), and Wikipedia. These resources are all licensed for adaptation and reuse under Creative Commons.

This text was originally written for EECS 490, the Programming Languages course at the University of Michigan, by Amir Kamil in Fall 2016. This is version 0.4 of the text.