PART 2

Static Types

5

Haskell

Haskell is a lazy, functional programming language that was initially designed by a committee in the eighties and nineties. In contrast to many programming languages, it is *lazy*, meaning that expressions are evaluated when their values are needed, rather than when they are first encountered. It is *functional*, meaning that the main organizational construct in the language is the function, rather than a procedure or a class hierarchy.

Studying Haskell is useful for a number of reasons. First, studying a functional language will make you think differently about programming. Mainstream languages are all about manipulating mutable state, whereas functional languages are all about manipulating *values*. Understanding the difference will make you a better programmer in whatever language you regularly use. Second, Haskell serves as a good example of many of the topics we will be covering later in the book, particularly issues related to type systems. Finally, the Haskell language is at the forefront of language design, with many new research papers published each year about extensions to the language. Haskell thus provides a glimpse as to where mainstream languages may be going in the future.

Because we will use Haskell in various chapters of the book to illustrate properties of programming languages, we will study Haskell in a little more detail than we will study some other languages. Compilers for Haskell are available on the Internet without charge. Several books and manuals covering the language are available. In addition to on-line sources easily located by web search, *Real World Haskell* (O'Reilly, 2009) is a good reference.

5.1 INTERACTIVE SESSIONS AND THE RUN-TIME SYSTEM

In addition to providing standard batch compilation, most Haskell compilers provide the same kind of `read-eval-print` loop as many Lisp implementations. In this mode of use, programmers enter expressions and declarations one at a time. As each is entered, the source code is type checked, compiled, and executed. Once an identifier has been given a value by a declaration, that identifier can be used in subsequent expressions.

The program that implements the `read-eval-print` loop is often called a shell or an interpreter, even though the code is actually compiled. Such shells are useful for interactively experimenting with code and can function as debuggers. We will use such a shell to explore various features of Haskell.

5.1.1 Expressions

For expressions, user interaction with the Haskell shell has the form

```
Prelude> <expression>
<print_value>
it :: <type>
```

where "`Prelude>`" is the prompt for user input (The word "`Prelude`" indicates that only the standard prelude module has been loaded). The second two lines are output from the Haskell compiler and run-time system. These lines illustrate that if an expression is entered, the shell will compile the expression and evaluate it. The first line of the output `<print_value>` is the value of the expression, serialized as a string. The second line of output is a bit cryptic: `it` is a special identifier bound to the value of the last expression entered and the notation "`::`" can be read "has type," so `it :: <type>` means that the variable `it` is bound to this most recent expression and furthermore, it has type `type`.

It is probably easier to understand the idea from a few examples. Here are four lines of input and the resulting compiler output:

```
Prelude> (5+3) -2
6
it :: Integer
Prelude> it + 3
9
it :: Integer
Prelude> if True then 1 else 5
1
it :: Integer
Prelude> 5 == 4
False
it :: Bool
```

In words, the value of the first expression is the integer `6`. The second expression adds `3` to the value `it` of the previous expression, giving integer value `9`. The third expression is an `if-then-else`, which evaluates to the integer `1`, and the fourth expression is a Boolean-valued expression (comparison for equality) with value `False`.

Each expression is parsed, type checked, compiled, and executed before the next input is read. If an expression does not parse correctly or does not pass the type-checking phase of the compiler, no code is generated and no code is executed. The ill-typed expression

```
if True then 3 else False
```

for example, parses correctly because this has the correct form for an `if-then-else`. However, the type checker rejects this expression because the Haskell type checker requires the `then` and `else` parts of an `if-then-else` expression to have the same types, as described in the next subsection. The compiler output for this expression includes the error message

```
<interactive>:1:14:
No instance for (Num Bool)
arising from the literal ‘3’ at <interactive>:1:14
```

indicating a type mismatch. In more detail, the message says that the literal `False`, which has type `Bool`, does not belong to the *type class* `Num`, which is the collection of all types that support arithmetic operations and literals such as `3`. Since `Bool` does not belong to the `Num` type class, the type checker reports a type error because the two branches of the `if-then-else` have different types. A detailed discussion of the Haskell type system appears in Chapters 6 and 7.

5.1.2 Declarations

User input can be an expression or a declaration. The standard form for Haskell declarations, followed by compiler output, is

```
Prelude> let <identifier> = <expression>
<identifier> :: <type>
```

The keyword `let` indicates we are introducing a new identifier whose value comes from `<expression>`. The compiler type checks the expression and the shell output indicates that the newly introduced identifier has the type of the expression. If we ask the shell for the value of the identifier, the shell evaluates the expression bound to the identifer and prints the result, as before. Here are some examples:

```
Prelude> let x = 7 + 2
x :: Integer
Prelude> x
9
it :: Integer
```

```
Prelude> let y = x + 3
y :: Integer
Prelude> let z = x * y - (x + y)
z :: Integer
Prelude> z
87
it :: Integer
```

In words, the first declaration binds the expression `7+2` to the identifier `x`. When we ask for the value of `x` by entering it at the command-line, the compiler evaluates the expression associated with `x` and prints the resulting value `9`. After this evaluation, `x` is bound to the value `9`, and so the expression `7+2` will not be evaluated again. The second declaration refers to the value of `x` from the first declaration and binds the expression `x+3` to the identifier `y`. The third declaration binds the identifier `z` to an expression that refers to both of the previous declarations. When we ask for the value of `z`, the compiler evaluates the associated expression and prints `87` as the result. Because the value of `z` depends upon the value of `y`, the shell will evaluate the expression associated with `y` in the process of computing the value of `z`.

Functions can also be declared with the keyword `let`. The general form of user input and compiler output is

```
Prelude> let <identifier> <arguments> = <expression>
<identifier> :: <arg_type> -> <result_type>
```

This declares a function whose name is `<identifier>`. The argument type is determined by the form of `<arguments>` and the result type is determined by the form of `<expression>`.

Here is an example:

```
Prelude> let f x = x + 5
f :: (Num a) => a -> a
```

This declaration binds a function value to the identifier `f`. The value of `f` is a function with an interesting type. It says that for any type `a` that belongs to the `Num` type class (the "`(Num a) =>`" part of the syntax), function `f` can take `a` as an argument and will return `a` as a result (the "`a -> a`" part of the syntax). Since the type `Integer` belongs to the type class `Num`, a particular instance of the type for `f` is `Integer -> Integer`. We will see more about this kind of type in Chapter 7.

The same function can be declared using an anonymous function written as

```
Prelude> let f = \x -> x + 5
f :: Integer -> Integer
```

In this declaration, the identifier `f` is given the value of expression `\x => x + 5`, which is a function expression like `(lambda (x) (+ x 5))` in Lisp or $\lambda x.x+5$ in lambda calculus. Haskell requires that anonymous functions have *monomorphic types*, which means that their types cannot mention type variables such as the `a` in the earlier definition of `f`. As a result, the Haskell type checker assigns this later version of `f` the monomorphic type `Integer -> Integer`. We will discuss functions further in Section 1.3 and typechecking issues in Chapter 6.

Identifiers vs. Variables. An important aspect of Haskell is that the value of an identifier cannot be changed by assignment. More specifically, if an identifier `x` is declared by `let x = 3`, for example, then the value of `x` will always be `3`. It is not possible to change the value of `x` by assignment. In other words, Haskell declarations introduce constants, not variables. The way to declare an assignable variable in Haskell is to define a reference cell, which is similar to a cons cell in Lisp, except that reference cells do not come in pairs. References and assignment are explained in Chapter 8.

The Haskell treatment of identifiers and variables is more uniform than the treatment of identifiers and variables in languages such as C and Java. If an integer identifier is declared in C or Java, it is treated as an assignable variable. On the other hand, if a function is declared and given a name in either of these languages, the name of the function is a constant, not a variable. It is not possible to assign to the function name and change it to a different function. Thus, C and Java choose between variables and constants according to the type of the value given to the identifier. In Haskell, a `let` declaration works the same way for all types of values.

5.2 BASIC TYPES AND TYPE CONSTRUCTORS

The core expression and declaration parts of Haskell are best summarized by a list of the basic types along with the expression forms associated with each type.

5.2.1 Unit

Haskell's unit type has only one element. Both the type and the element are written as empty parentheses:

```
() :: ()
```

The unit type is used as the type of argument for functions that have no arguments. C programmers may be confused by the fact that the term unit suggests one element, whereas `void` seems to mean no elements. From a mathematical point of view, "one element" is correct. In particular, if a function is supposed to return an element of an empty type, then that function cannot return because the empty set (or empty type) has no elements. On the other hand, a function that returns an element of a one-element type can return. However, we do not need to keep track of what value such a function returns, as there is only one thing that it could possibly return. The Haskell type system is based on years of theoretical study of types; most of the typing concepts in Haskell have been considered with great care.

5.2.2 Bool

There are two values of type `Bool`, `True` and `False`:

```
True :: Bool
False :: Bool
```

The most common expression associated with Booleans is the conditional, with

```
if e1 then e2 else e3
```

having the same type as `e2` and `e3` if these have the same type and `e1` has type `Bool`. There is no `if-then` without `else`, as a *conditional expression* must have a value whether the test is true or false. For example, an expression

```
Prelude> let nonsense = if a then 3
```

is not legal Haskell because there is no value for `nonsense` if the expression `a` is `False`. More specifically, the input `if a then 3` does not even parse correctly; there is no parse tree for this string in the syntax of Haskell.

There are also Haskell Boolean operations for `and`, `or`, `not`, and so on. Conjunction (`and`) is written as `&&`, disjunction (`or`) is written as ||, and negation is written as `not`. For example, here is a function that determines whether its two arguments have the same Boolean value, followed by an expression that calls this function:

```
Prelude> let equiv x y = (x && y) || ((not x) && (not y))
equiv :: (Bool, Bool) -> Bool
Prelude> equiv True False
False
it :: Bool
```

In words, Boolean arguments `x` and `y` are the same Boolean value if they are either both true or both false. The first subexpression, `(x && y)`, is true if `x` and `y` are both true and the second subexpression, `((not x) && (not y))`, is true if they are both false.

The conjunction and disjunction operations come with a fixed evaluation order. In the expression `(e1 && e2)`, where `e1` and `e2` are both expressions, `e1` is evaluated first. If `e1` is true, then `e2` is evaluated. Otherwise the value of the expression `(e1 && e2)` is determined to be false without evaluating `e2`. Similarly, `e2` in `(e1 || e2)` is evaluated only if the value of `e1` is false.

5.2.3 Integers

Haskell provides a number of different integer types of varying precision. The type `Integer` corresponds to arbitrary precision signed integers; `Int8`, `Int16`, `Int32`, and `Int64` correspond to signed integers with the indicated size in bits; while `Word8`, `Word16`, `Word32`, and `Word64` correspond to unsigned integers with the indicated sizes.

Many Haskell integer expressions are written in the usual way, with numeric constants and standard arithmetic operations:

```
0,1,2,...,-1,-2,...:: (Num a) => a
+, -, * :: (Num a) => a -> a -> a
```

The type for the literals indicates that each integer literal can have any type `a` as long as `a` belongs to the `Num` type class. In addition, the arithmetic operators for addition (`+`), subtraction (`-`), and multiplication (`*`) work over any type `a` belonging to the `Num` type class. Each of these operators is an infix binary operator. In Haskell, any identifier comprised of symbols (such as `+`, `-`, etc.) is an infix operator.

The function `div` provides integer division for any type that belongs to the `Integral` type class:

```
div :: (Integral a) => a -> a -> a
```

Types in the `Num` type class support operations that work for either floating point numbers or integers, while the types that belong to the `Integral` type class support operations on integers (like `div`). Integer types like `Int32` and `Word8` belong to both the `Num` and `Integral` type classes. Because `div` is comprised of characters and not symbols, it is not an infix operator:

```
Prelude> let quotient x y = div x y
quotient :: (Integral a) => a -> a -> a
```

However, we can convert any function into an infix operator by enclosing it in backticks:

```
Prelude> let quotient x y = x `div` y
quotient :: (Integral a) => a -> a -> a
```

Similarly, we can convert any infix operator into a normal function by enclosing it in parentheses:

```
Prelude> let plus2 x y = (+) x y
```

```
plus2 :: (Num a) => a -> a -> a
```

5.2.4 Strings

Strings are written as a sequence of symbols between double quotes:

```
"Simon Peyton Jones" :: String
"Brendan Eich" :: [Char]
```

In Haskell, the type `String` is a synonym for the type `[Char]`, which denotes a list of characters. This design means that strings can be treated as lists of characters, supporting all the usual list operations. String concatenation is written as `++`, so we have

```
Prelude> "James" ++ " " ++ "Gosling"
"James Gosling"
it :: [Char]
```

5.2.5 Real

Haskell has two types for floating-point numbers of varying precision: `Float` and `Double`, as well as a type `Rational` for numbers that can be expressed as the ratio of two integers. As we saw with integral types, Haskell uses a type class to group the operations common to these types, in this case, the type class `Fractional`.

Literals with decimal points thus have type:

```
1.0, 2.0, 3.14159, 4.44444, ... :: (Fractional t) => t
```

meaning they can be given any type that belongs to the `Fractional` type class: `Float`, `Double`, and `Rational`. These three types also belong to the `Num` class, so we can apply the operations `+`, `-`, and * to `Float`s, `Double`s, and `Rational`s as well. The `div` operation, however, only works for elements of the `Integral` type class, which does not include `Float`s, `Double`s, or `Rational`. The operation `(/)` is used to divide fractional numbers instead.

5.2.6 Tuples

A tuple may be a pair, triple, quadruple, and so on. In Haskell, tuples may be formed of any types of values. Tuple values and types are written with parentheses. For example, here is the compiler output for a pair, a triple, and a quadruple:

```
Prelude> (True, "John McCarthy")
(True,"John McCarthy")
it :: (Bool, [Char])
Prelude> ("Brian", "Dennis", "Bjarne")
("Brian","Dennis","Bjarne")
it :: ([Char], [Char], [Char])
Prelude>(1, 2.3, 4.6, 10)
(1,2.3,4.6,10)
it :: (Integer, Double, Double, Integer)
```

For all types τ_1 and τ_2, the type (τ_1, τ_2) is the type of pairs whose first component has type τ_1 and whose second component has type τ_2. The type (τ_1, τ_2, τ_3) is a type of triples, the type $(\tau_1, \tau_2, \tau_3, \tau_4)$ a type of quadruples, and so on.

Components of a pair can be accessed using the functions `fst` and `snd`, respectively. For examples:

```
Prelude> fst(3,4)
3
it :: Integer
Prelude> snd("Niklaus", "Wirth")
"Wirth"
it :: [Char]
```

Components of tuples with more elements can be accessed using pattern matching constructs, which we will see in more detail in Section 1.3.

5.2.7 Lists

Haskell lists can have any length, but all elements of a list must have the same type. We can write lists by listing their elements, separated by commas, between brackets. Here are some example lists of different types:

```
Prelude> [1,2,3,4]
[1,2,3,4]
it :: [Integer]
Prelude> [True, False]
[True,False]
it :: [Bool]
Prelude> ["red", "yellow", "blue"]
["red","yellow","blue"]
it :: [[Char]]
```

The type `[`τ`]` is the type of all lists whose elements have type τ.

In Haskell, the empty list is written `[]`. The "cons" operation for adding an element to the front of a list is an infix operator written as a single colon:

```
Prelude> 3 : []
[3]
it :: [Integer]
Prelude> 4 : 5 : it
[4,5,3]
it :: [Integer]
```

In the first list expression, `3` is "consed" onto the front of the empty list. The result is a list containing the single element `3`. In the second expression, `4` and `5` are consed onto this list. In both cases, the result is an integer list, `[Integer]`.

5.3 PATTERNS, DECLARATIONS, AND FUNCTION EXPRESSIONS

The declarations we have seen so far bind a value to a single identifier. One very convenient syntactic feature of Haskell is that declarations can also bind values to a set of identifiers by using patterns.

5.3.1 Value Declarations

The general form of value declaration associates a value with a pattern. A pattern is an expression containing variables (such as `x, y, z` ...) and constants (such as `true`, `false`, `1, 2, 3` ...), combined by certain forms such as tupling, record expressions, and a form of operation called a constructor. The general form of value declaration is

```
let <pattern> = <exp>
```

where the common forms of patterns are summarized by the following BNF grammar:

```
<pattern> ::= <id> | <tuple> | <cons> | <constr> | <record>
<tuple>   ::= (<pattern>, ..., <pattern>)
<cons>    ::= <pattern> : <pattern>
<constr>  ::= <id> <patterns>
<record>  ::= <id> {<id>=<pattern>, ..., <id>=<pattern>}

<patterns> :: <pattern> ... <pattern>
```

In words, a pattern can be an identifier, a tuple pattern, a list cons pattern, a declared data-type constructor pattern, or a record pattern. A tuple pattern is a sequence of patterns between parentheses, a list cons pattern is two patterns separated by a colon, a constructor pattern is an identifier (a declared constructor) applied to the number of pattern arguments required by the constructor, and a record pattern is a record-like expression with each field in the form of a pattern. This BNF does not define the set of patterns exactly, as some conditions on patterns are not context free and therefore cannot be expressed by BNF. For example, the conditions that in a constructor pattern the identifier must be a declared constructor and that the constructor must be applied to the right number of pattern arguments are not context-free conditions. An additional condition on patterns, subsequently discussed in connection with function declarations, is that no variable can occur twice in any pattern.

Because a variable is a pattern, a value declaration can simply associate a value with a variable. For example, here is a declaration that binds a tuple to the identifier `t`, followed by a declaration that uses a tuple pattern to bind identifiers `x`, `y`, and `z`.

```
Prelude> let t = (1,2,3)
t :: (Integer, Integer, Integer)
Prelude> let (x,y,z) = t
x :: Integer
y :: Integer
z :: Integer
```

Note that there are two lines of input in this example and four lines of compiler output. In the first declaration, the identifier `t` is bound to a tuple. In the second declaration, the tuple pattern `(x,y,z)` is given the value of `t`. When the pattern `(x,y,z)` is matched against the triple `t`, identifier `x` gets value `1`, identifier `y` gets value `2`, and identifier `z` gets value `3`.

5.3.2 Function Declarations

The general form of a function declaration uses patterns. A single-clause definition has the form

```
let f <patterns> = <exp>
```

and a multiple-clause definition has the form

```
let { f <patterns> = <exp> ; ...; f <patterns> = <exp> }
```

For example, a function adding its arguments can be written as

```
let f (x,y) = x + y
```

Technically, the formal parameter of this function is a pattern `(x, y)` that must match the actual parameter on a call to `f`. The formal parameter to `f` is a tuple, which is broken down by pattern matching into its first and second components. You may think you are calling a function of two arguments. In reality, you are calling a function of one argument. That argument happens to be a pair of values. Pattern matching takes the tuple apart, binding `x` to what you might think is the first parameter and `y` to the second.

Here are some more examples illustrating other forms of patterns, each shown with an associated compiler output:

```
Prelude> let f(x, (y,z)) = y
f :: (t, (t1, t2)) -> t1
Prelude> let g(x:y:z) = x:z
g :: [t] -> [t]
```

The first is a function on nested tuples and the second a function on lists that have at least two elements.

An example of a multi-clause function is the following function, which computes the length of a list:

```
Prelude> let {length [] = 0; length (x:xs) = 1 + length xs}
length :: (Num t1) => [t] -> t1
```

The first line in this code fragment is input (the declaration of the function `length`) and the second line is the compiler output giving the type of this function. Here is an example application of `length` and the resulting value:

```
Prelude> length ['a', 'b', 'c', 'd']
4
it :: Integer
```

When the function `length` is applied to an argument, the clauses are matched in the order they are written. If the argument matches the constant `[]` (i.e., the argument is the empty list), then the function returns the value `0`, as specified by the first clause. Otherwise the argument is matched against the pattern given in the second clause `(x:xs)`, and then the code for the second branch is executed. Because type checking guarantees that `length` will be applied only to a list, these two clauses cover all values that could possibly be passed to this function. The type of `length`, `(Num t1) => [t] -> t1`, indicates that the function takes a list of any type `t` and returns any type `t1` that belongs to the `Num` type class. Such types will be explained in the next two chapters.

The syntax given so far specifies how to define a function within a `let` block, which is the context provided by the Haskell interpreter. Inside a code file, however, functions call also be declared at the top level, in which case the syntax for a single-line function is

```
f <patterns> = <exp>
```

and for a multi-line function is

```
f <patterns> = <exp>
...
f <patterns> = <exp>
```

For example, we can define the `length` function in a code file as

```
length [] = 0
length (x:xs) = 1 + (length xs)
```

Note that pattern matching is applied in order. For example, when the function

```
f (x,0) = x
f (0,y) = y
f (x,y) = x+y
```

is applied to an argument `(a,b)`, the first clause is used if `b=0`, the second clause if `b≠0` and `a=0`, and the third clause if `b≠0` and `a≠0`. The Haskell type system will keep `f` from being applied to any argument that is not a pair `(a,b)`.

An important condition on patterns is that no variable can occur twice in any pattern. For example, the following function declaration is not syntactically correct because the identifier `x` occurs twice in the pattern:

```
Prelude> let { eq (x,x) = True; eq(x,y) = False }
<interactive>:1:11:
Conflicting definitions for ‘x’
Bound at: <interactive>:1:11
<interactive>:1:13
In the definition of ‘eq’
```

This function is not allowed because multiple occurrences of variables express equality, and equality must be written explicitly into the body of a function.

In addition to declared functions, Haskell has syntax for anonymous functions. The general form allows the argument to be given by a pattern:

```
\<pattern> -> <exp>
```

As an example, the anonymous function `\x -> x + 1` adds one to its argument. Anonymous functions are often used with *higher-order functions*, which are functions that take other functions as arguments. For example, the `map` function is a higher-order function that takes as arguments a function `f` and a list and applies `f` to every element in the list. Here is an example, with compiler output:

```
Prelude> map (\x -> x + 1) [0,1,2,3]
[1,2,3,4]
it :: [Integer]
```

5.4 HASKELL DATA-TYPE DECLARATION

The Haskell data-type declaration is a special form of type declaration that declares a type name and operations for building and making use of elements of the type. The Haskell data-type declaration has the syntactic form

```
data <type_name> = <constructor_clause> | ... |
<constructor_clause>
```

where a constructor clause has the form

```
<constructor_clause> ::= <constructor> <arg_types> |
<constructor>
```

where both `<type_name>` and `<constructor>` are capitalized identifiers. The idea is that each constructor clause specifies one way to construct elements of the type. Elements of the type may be "deconstructed" into their constituent parts by pattern matching. The following three examples illustrate some common ways of using data-type declarations in Haskell programs.

Example. An Enumerated Data Type: Types consisting of a finite set of tokens can be declared as Haskell data types. Here is a type consisting of three tokens, named to indicate three specific colors:

```
data Color = Red | Blue | Green
```

This declaration indicates that the three elements of type color are `Blue`, `Green`, and `Red`. Technically, values `Blue`, `Green`, and `Red` are called *constructors.* They are called constructors because they are the ways of constructing values with type color.

Example. A Tagged Union Data Type: Haskell constructors can be declared to take arguments when constructing elements of the data type. Such constructors simply "tag" their arguments so that values constructed in different ways can be distinguished.

Suppose we are keeping student records with names of B.S. students, names and undergraduate institutions of M.S. students, and names and faculty supervisors of Ph.D. students. Then we could define a type `student` that allows these three forms of tuples as follows:

```
data Student = BS Name | MS (Name, School) | PhD (Name,
Faculty)
```

In this data-type declaration, `BS`, `MS`, and `PhD` are each constructors. However, unlike in the color example, each `Student` constructor must be applied to arguments to construct a value of type `Student`. We must apply `BS` to a name, `MS` to a pair consisting of a name and a school, and `PhD` to a pair consisting of a name and a faculty name in order to produce a value of type `Student`.

In effect, the type `Student` is the union of three types,

```
Student ≈ union {Name, Name*School Name*Faculty }
```

except that in Haskell "unions" (which are defined by `data` declarations), each value of the union is tagged by a constructor that tells which of the constituent types the value comes from. This is illustrated in the following function, which returns the name of a student:

```
name :: Student → Name
name (BS n) = n
name (MS(n,s)) = n
name (PhD(n,f)) = n
```

The first line documents that the function `name` is a function from students to names. The next three lines declare the function `name`. The function has three clauses, one for each form of student.

Example. A Recursive Type: Data-type declaration may be recursive in that the type name may appear in one or more of the constructor argument types. Because of the way type recursion is implemented, Haskell data types are a convenient, high-level language construct that hides a common form of routine pointer manipulation.

The set of trees with integer labels at the leaves may be defined mathematically as follows:

> A *tree* is either
>
> a leaf, with an associated integer label, or
>
> a compound tree, consisting of a left subtree and a right subtree.

This definition can be expressed as a Haskell data-type declaration, with each part of the definition corresponding to a clause of the data-type declaration:

```
data Tree = Leaf Int | Node (Tree, Tree)
```

The identifiers `Leaf` and `Node` are constructors, and the elements of the data type are all values that can be produced by the application of constructors to legal (type-correct) arguments. In words, a `Tree` is either the result of applying the constructor `Leaf` to an integer (signifying a `leaf` with that integer label) or the result of applying the constructor `Node` to two trees. These two trees, of course, must be produced similarly with constructors `Leaf` and `Node`.

The following function shows how the constructors may be used to define a function on trees:

```
inTree :: Int → Tree → Bool
inTree x (Leaf y) = x == y
inTree x (Node(y,z)) = inTree x y || inTree x z
```

This function looks for a specific integer value `x` in a tree. If the tree has the form `Leaf y`, then `x` is in the tree only if `x==y`. If the tree has the form `Node (y,z)`, with subtrees `y` and `z`, then `x` is in the tree only if `x` is in the subtree `y` or the subtree `z`. The type shows that `inTree` is a function that, given an integer and a tree, returns a Boolean value.

An example of a polymorphic data-type declaration appears in Section 6.5.3, after the discussion of polymorphism in Section 6.4.

5.5 RECORDS

Like Pascal records and C structs, Haskell records are similar to tuples, but with named components. Like data types, Haskell records must be declared before they can be used. For example, the declaration

```
data Person = Person { firstName :: String,
                        lastName  :: String }
```

introduces the `Person` record type, which has two fields each with type string. Given a record declaration, we can create values of the type `Person` by giving a value for each field:

```
let dk = Person { firstName = "Donald", lastName = "Knuth" }
```

This `Person` record has two components, one called `firstName` and the other called `lastName`.

Each field in a record declaration introduces an accessor function of the same name. For example, we can use the `firstName` function to extract the corresponding name from the `dk` person.

```
Prelude> firstName dk
"Donald"
it :: String
```

Another way of selecting components of tuples and records is by pattern matching. For example, the following statement binds the variable `f` to the first name of `dk` and `l` to the last name.

```
Prelude> let (Person {firstName = f, lastName = l } ) = dk
f :: String
l :: String
```

5.6 A NOTE ON REFERENCE CELLS AND ASSIGNMENT

None of the Haskell constructs discussed in earlier sections of this chapter have side effects. Each expression has a value, but evaluating an expression does not have the side effect of changing the value of any other expression. Although most large Haskell programs are written in a style that avoids side effects when possible, most large Haskell programs do use assignment occasionally to change the value of a variable.

The way that assignable variables are presented in Haskell is different from the way that assignable variables appear in other programming languages. The main reasons for this are to preserve the uniformity of Haskell as a programming language and to separate side effects from pure expressions.

Haskell assignment is restricted to reference cells. In Haskell, a reference cell has a different type than immutable values such as integers, strings, lists, and so on. Because reference cells have specific reference types, restrictions on Haskell assignment are enforced as part of the type system. This is part of the elegance of Haskell: Almost all restrictions on the structure of programs are part of the type system, and the type system has a systematic, uniform definition.

We will discuss reference cells and other imperative features of Haskell in Chapter 8.

5.7 CHAPTER SUMMARY

Haskell is a programming language that encourages programming with functions. It is easy to define functions with function arguments and function return results. In addition, most data structures in Haskell programs are not assignable. Although it is possible to construct reference cells for any type of value and modify reference cells by assignment, side effects occur only when reference cells are used. Although most large Haskell programs do use reference cells and have side effects, the pure parts of Haskell are expressive enough that reference cells are used sparingly.

Haskell has an expressive type system. There are basic types for many common kinds of computable values, such as Booleans, integers, strings, and reals. There are also *type constructors*, which are type operators that can be applied to any type. The type constructors include tuples, records, and lists. In Haskell, it is possible to define tuples of lists of functions, for example. There is no restriction on the types of values that can be placed in data structures.

The Haskell type system is often called a *strong type system*, as every expression has a type and there are no mechanisms for subverting the type system. When the Haskell type checker determines that an expression has type `Int`, for example, then any successful evaluation of that expression is guaranteed to produce an integer. There are no dangling pointers that refer to unallocated locations in memory and no casts that allow values of one type to be treated as values of another type without conversion.

Haskell has several forms that allow programmers to define their own types and type constructors. In this chapter, we looked at data-type declarations, which can be used to define Haskell versions of enumerated types (types consisting of a finite list of values), disjoint unions (types whose elements are drawn from the union of two or more types), and recursively defined types. Another important aspect of the Haskell type system is polymorphism, which we will study in the next chapter, along with other aspects of the Haskell type system. We will discuss additional type definition and module forms in Chapter ??.

EXERCISES

5.1 Algol 60 Procedure Types

In Algol 60, the type of each formal parameter of a procedure must be given. However, *proc* is considered a type (the type of procedures). This is much simpler than the ML types of function arguments. However, this is really a type loophole; because calls to procedure parameters are not fully type checked, Algol 60 programs may produce run-time type errors.

Write a procedure declaration for `Q` that causes the following program fragment to produce a run-time type error:

```
proc P (proc Q)
   begin Q(true) end;
P(Q);
```

where `true` is a Boolean value. Explain why the procedure is statically type correct, but produces a run-time type error. (You may assume that adding a Boolean to an integer is a run-time type error.)

5.2 Algol 60 Pass-By-Name

The following Algol 60 code declares a procedure `P` with one pass-by-name integer parameter. Explain how the procedure call `P(A[i])` changes the values of `i` and `A` by substituting the actual parameters for the formal parameters, according to the Algol 60 copy rule. What integer values are printed by `tprogram`? By using pass-by-name parameter passing?

The line `integer x` does not declare local variables – this is just Algol 60 syntax declaring the type of the procedure parameter:

```
begin
   integer i;
   integer array A[1:2];

   procedure P(x);
      integer x;
      begin
          i := x;
         x := i
      end

   i := 1;
   A[1] := 2; A[2] := 3;
   P (A[i]);
   print (i, A[1], A[2])
end
```

5.3 Nonlinear Pattern Matching

Haskell patterns cannot contain repeated variables. This exercise explores this language design decision. A declaration with a single pattern is equivalent to a sequence of declarations using destructors. For example,

```
p = (5,2)
(x,y) = p
```

is equivalent to

```
p = (5,2)
x = fst p
y = snd p
```

where `fst p` is the Haskell expression for the first component of pair `p` and `snd` similarly returns the second component of a pair. The operations `fst` and `snd` are called *destructors* for pairs.

A function declaration with more than one pattern is equivalent to a function declaration that uses standard `if-then-else` and destructors. For example,

```
f [] = 0
f (x:xs) = x
```

is equivalent to

```
f z = if z == [] then 0 else head z
```

where `head` is the Haskell function that returns the first element of a list.

Questions:

(a) Write a function that does not use pattern matching and that is equivalent to

```
f (x,0) = x
f (0,y) = y
f (x,y) = x + y
```

Haskell pattern matching is applied in order. When the function `f` is applied to an argument `(a, b)`, the first clause is used if `b = 0`, the second clause if `b≠0` and `a=0`, and the third clause if `b≠0` and `a≠0`.

(b) Consider the following function:

```
eq(x,x) = True
eq(x,y) = False
```

Describe how you could translate Haskell functions that contain patterns with repeated variables, like `eq`, into functions without patterns, using destructors and `if-then-else` expressions. Give the resulting translation of the `eq` function.

(c) Why do you think the designers of Haskell prohibited repeated variables in patterns? (*Hint:* If `f, g :: Int -> Int`, then the expression `f == g` is not type-correct Haskell as the test for equality is not defined on function types.)

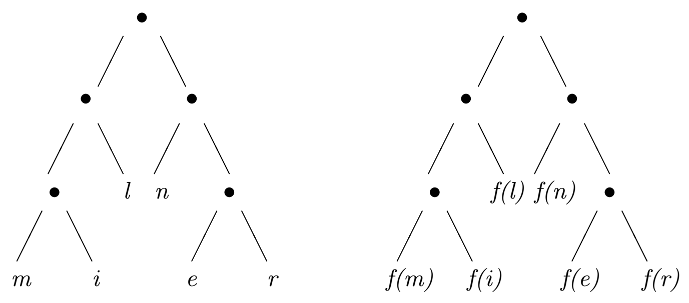
5.4 Haskell Map for Trees

(a) The binary tree data type

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

describes a binary tree for any type, but does not include the empty tree (i.e., each tree of this type must have at least a root node).

Write a function `maptree` that takes a function as an argument and returns a function that maps trees to trees by mapping the values at the leaves to new values, using the function passed in as a parameter. In more detail, if `f` is a function that can be applied to the leaves of tree `t` and `t` is the tree on the left, then `maptree f t` should result in the tree on the right:



For example, if `f` is the function `f x = x + 1` then

```
maptree f (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
```

should evaluate to `Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)`. Explain your definition in one or two sentences.

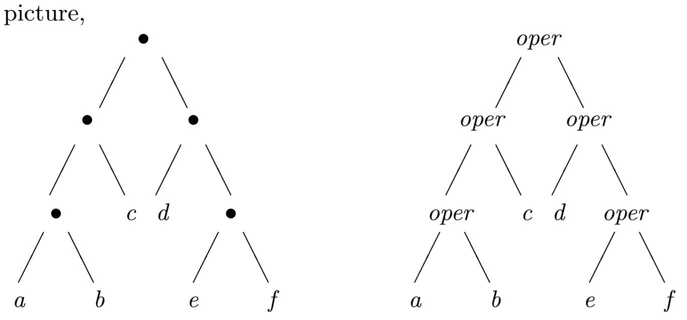
(b) What is the type Haskell gives to your function? Why is it not the type `(t -> t) -> Tree t -> Tree t`?

5.5 Haskell Reduce for Trees

Assume that the data type `tree` is defined as in problem 4. Write a function

```
reduce :: (a -> a -> a) -> Tree a -> a
```

that combines all the values of the leaves by using the binary operation passed as a parameter. In more detail, if `oper :: a -> a -> a` and `t` is the nonempty tree on the left in this picture,



then `reduce oper t` should be the result we obtain by evaluating the tree on the right. For example, we can reduce an integer tree with the plus function:

```
reduce (+) (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3))
= (1 + 2) + 3
= 6
```

Explain your definition of `reduce` in one or two sentences.

5.6 Currying

This problem asks you to show that the Haskell types `a -> b -> c` and `(a,b) -> c` are essentially equivalent.

(a) Define higher-order Haskell functions

```
curry :: ((a,b) -> c) -> (a -> (b -> c))
```

and

```
uncurry :: (a -> (b -> c)) -> ((a,b) -> c)
```

(b) For all functions `f :: (a,b) -> c` and `g :: a -> (b -> c)`, the following two equalities should hold (if you wrote the right functions):

```
uncurry(curry f) = f
curry(uncurry g) = g
```

Explain why each is true for the functions you have written. Your answer can be three or four sentences long. Try to give the main idea in a clear, succinct way. (We are more interested in insight than in number of words.) Be sure to consider termination behavior as well.

5.7 Disjoint Unions

A *union type* is a type that allows the values from two different types to be combined in a single type. For example, an expression of type `union(A, B)` might have a value of type `A` or a value of type `B`. The languages C and ML both have forms of union types.

(a) Here is a C program fragment written with a union type:

```
...
union IntString {
  int i;
  char *s;
} x;
int y;
if ( ...) x.i = 3 else x.s = ``here, fido'';
...
y = (x.i) + 5;
...
```

A C compiler will consider this program to be well typed. Despite the fact that the program type checks, the addition may not work as intended. Why not? Will the run-time system catch the problem?

(b) In ML, a union type `union(A,B)` would be written in the form `datatype UnionAB = tag_a of A | tag_b of B` and the preceding `if` statement could be written as

```
datatype IntString = tag_int of int | tag_str of string;
...
val x = if ...then tag_int(3) else tag_str(``here, fido'');
...
let val tag_int (m) = x in m + 5 end;
```

Can the same bug occur in this program? Will the run-time system catch the problem? The use of tags enables the compiler to give a useful warning message to the programmer, thereby helping the programmer to avoid the bug, even before running the program. What message is given and how does it help?

5.8 Lazy Evaluation and Functions

It is possible to evaluate function arguments at the time of the call (*eager evaluation*) or at the time they are used (*lazy evaluation*). Most programming languages (including ML) use eager evaluation, but we can simulate lazy evaluation in an eager language such as ML by using higher-order functions.

Consider a *sequence* data structure that starts with a known value and continues with a function (known as a *thunk*) to compute the rest of the sequence:

```
Prelude> datatype 'a Seq = Nil
             | Cons of 'a * (unit -> 'a Seq);

Prelude> fun head (Cons (x, _)) = x;
val head = fn : 'a Seq -> 'a

Prelude> fun tail (Cons (_, xs)) = xs();
val tail = fn : 'a Seq -> 'a Seq

Prelude> fun BadCons (x, xs) = Cons (x, fn() =>xs);
```

```
val BadCons = fn : 'a * 'a Seq -> 'a Seq
```

Note that `BadCons` does not actually work, as `xs` was already evaluated on entering the function. Instead of calling `BadCons(x, xs)`, you would need to use `Cons(x, fn() =>xs)` for lazy evaluation.

This lazy sequence data type provides a way to create infinite sequences, with each infinite sequence represented by a function that computes the next element in the sequence. For example, here is the sequence of infinitely many 1s:

```
Prelude> val ones = let fun f() = Cons(1,f) in f() end;
```

We can see how this works by defining a function that gets the nth element of a sequence and by looking at some elements of our infinite sequence:

```
Prelude> fun get(n,s) = if n=0 then head(s) else get(n-1,tail(s));
val get = fn : int * 'a Seq -> 'a
Prelude> get(0,ones);
val it = 1 : int
Prelude> get(5,ones);
val it = 1 : int
Prelude> get(245, ones);
val it = 1 : int
```

We can define the infinite sequence of all natural numbers by

```
Prelude> val natseq = let fun f(n)() = Cons(n,f(n+1)) in f(0)()
end;
```

Using sequences, we can represent a function as a potentially infinite sequence of ordered pairs. Here are two examples, written as infinite lists instead of as ML code (note that $\sim$ is a negative sign in ML):

$$\begin{aligned} \text{add1} &= (0,1) :: (\sim 1, 0) :: (1,2) :: (\sim 2, \sim 1) :: (2,3) :: \ldots \\ \text{double} &= (0,0) :: (\sim 1, \sim 2) :: (1,2) :: (\sim 2, \sim 4) :: (2,4) :: \ldots \end{aligned}$$

Here is ML code that constructs the infinite sequences and tests this representation of functions by applying the sequences of ordered pairs to sample function arguments.

```
Prelude> fun make_ints(f)=
      let
        fun make_pos (n) = Cons( (n, f(n)), fn()=>make_pos(n + 1))
        fun make_neg (n) = Cons( (n, f(n)), fn()=>make_neg(n - 1))
      in
        merge (make_pos (0), make_neg(~1))
      end;
val make_ints = fn : (int -> 'a) -> (int * 'a) Seq

Prelude> val add1 = make_ints (fn(x) => x+1);
val add1 = Cons ((0,1),fn) : (int * int) Seq

Prelude> val double = make_ints (fn(x) => 2*x);
val double = Cons ((0,0),fn) : (int * int) Seq

Prelude> fun apply (Cons( (x1,fx1), xs) , x2) =
      if (x1=x2) then fx1
        else apply(xs(), x2);
```

```
val apply = fn : (''a * 'b) Seq * ''a -> 'b

Prelude> apply(add1, ~4);
val it = ~3 : int
Prelude> apply(double, 7);
val it = 14 : int
```

(a) Write *merge* in ML. Merge should take two sequences and return a sequence containing the values in the original sequences, as used in the `make_ints` function.

(b) Using the representation of functions as a potentially infinite sequence of ordered pairs, write *compose* in ML. Compose should take a function f and a function g and return a function h such that $h(x) \mathcal{D} f(g(x))$.

(c) It is possible to represent a partial function whose domain is not the entire set of integers as a sequence. Under what conditions will your *compose* function `not halt`? Is this acceptable?