Curry

A Tutorial Introduction

Draft of January 10, 2006

Sergio Antoy

Portland State University, U.S.A.
Email: antoy@cs.pdx.edu
Web: http://www.cs.pdx.edu/~antoy/

Michael Hanus

Christian-Albrechts-Universität Kiel, Germany
Email: mh@informatik.uni-kiel.de
Web: http://www.informatik.uni-kiel.de/~mh/

Contents

Preface

This book is about programming in Curry, a general-purpose declarative programming language that integrates functional with logic programming. Curry seamlessly combines the key features of functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchro- nization on logical variables).

This book is best used as an introduction to Curry. Curry is a rigorously defined programming language. The "Report" is a still evolving, but fairly stable, document that precisely defines the language, in particular both its syntax and operational semantics. However, the report is not best suited to the beginner, rather it may be consulted in conjunction with this tutorial for the sake of a completeness that is not sought here.

There are several implementations of Curry. The most usable at the time of the writing (Nov. 2002) is PAKCS. The examples and exercises in this book have been developed and executed using PAKCS.

Part I

Language Features

Chapter 1

Introduction

Curry is a universal programming language aiming at the amalgamation of the most important declarative programming paradigms, namely functional programming and logic programming. Curry combines in a seamless way features from functional programming (nested expressions, lazy evaluation, higher-order functions), logic programming (logical variables, partial data structures, built-in search), and concurrent programming (concurrent evaluation of constraints with synchronization on logical variables). Moreover, Curry provides additional features in comparison to the pure languages (compared to functional programming: search, computing with partial information; compared to logic programming: more efficient evaluation due to the deterministic evaluation of functions). Moreover, it also amalgamates the most important operational principles developed in the area of integrated functional logic languages: "residuation" and "narrowing" (see [4] for a survey on functional logic programming).

The development of Curry is an international initiative intended to provide a common platform for the research, teaching[1] and application of integrated functional logic languages.

This document is intended to provide a tutorial introduction into the features of Curry and their use in application programming. It is not a formal definition of Curry which can be found in [11].

[1] Actually, Curry has been successfully applied to teach functional and logic programming techniques in a single course without switching between different programming languages. More details about this aspect can be found in [5].

Chapter 2

Getting Started with Curry

There are different implementations of Curry available[1]. As such we can not describe the use of a Curry system in general. Some implementations are batch-oriented. In this case a Curry program is compiled into machine code and then executed. In this introduction we prefer an implementation that supports an interactive environment which provides faster program development by loading and testing programs within the integrated environment.

PAKCS (Portland Aachen Kiel Curry System) [10][2] contains such an interactive environment so that we show the use of this system here in order to get started with Curry. When you start the interactive environment of PAKCS (e.g., by typing "`curry2prolog`" as a shell command), you see something like the following output after the system's initialization:

```
 ______      __       _    _    ______   _______
|  __  |    /  \     | |  / /  |  ____| |  _____|   Portland Aachen Kiel
| |  | |   / /\ \    | |_/ /   | |      | |_____    Curry System
| |__| |  / /__\ \   |  _  |   | |      |_____  |
|  ____| / ______ \  | | \ \   | |____   _____| |   Version 1.6.0
|_|     /_/      \_\ |_|  \_\  |______| |_______|   October 2004

Curry2Prolog Compiler Environment (Version of 25/10/04)
(RWTH Aachen, CAU Kiel, Portland State University)

Bug reports: mh@informatik.uni-kiel.de

Type ":h" for help

prelude>
```

Now the system is ready and waits for some input. By typing "`:q`" (quit) you can always leave the system, but this is not what we intend to do now. The prefix of the current input line always shows the currently loaded module or program. In this case the module `prelude` is loaded during system startup. The standard system module `prelude` contains the definition

[1] Check the web page http://www.informatik.uni-kiel.de/~curry for details.
[2] http://www.informatik.uni-kiel.de/~pakcs

of several predefined functions and data structures which are available in all Curry programs. For instance, the standard arithmetic functions like `+`, `*` etc are predefined so that we can use the system as a simple calculator (the input typed by the user is underlined):

```
prelude> 3+5*4
Result: 23 ?
```

In this simple example you can already see the basic functionality of the environment: if you type an expression, the system evaluates this expression to a *value* (i.e., an expression without evaluable functions) and prints this value as the result (with a question mark, see below). Now hit the "enter" key and you are back to input line mode where you can type additional expressions to be evaluated. For instance, you can compare the values of two expressions with the usual comparison operators `>`, `<`, `<=`, `>=`:

```
prelude> 3+5*4 >= 3*(4+2)
Result: True ?
```

`==` and `/=` are the operators for equality and disequality and can be used on numbers as well as on other datatypes:

```
prelude> 4+3 == 8
Result: False ?
```

You may wonder why the system always puts a question mark after the result and then waits for further input. The reason is that Curry cannot only perform functional, thus, purely deterministic computations, which yields at most one result; Curry subsumes logic languages which are able to search for different results. Therefore, arbitrary expressions might deliver several solutions and PAKCS computes these solutions one after the other. Thus, it prints the first solution followed by a question mark and, if you type "`;`" (followed by the "enter" key), it computes and prints the next solution and so on. We will see later examples for this feature but ignore it for the moment. So, just hit the "enter" key after a result in order to ignore the computation of further solutions.

One may want to use Curry as more than a mere desk calculator. Therefore, we will discuss how to write programs in Curry. In general, a Curry *program* is a set of function definitions. The simplest sort of functions are those that do not depend on any input value, i.e., constant functions. For instance, a definition like

```
nine = 3*3
```

(such definitions are also called *rules* or *defining equations*) defines the name `nine` as equal to the value of `3*3`, i.e., `9`. This means that each occurrence of the name `nine` in an expression is replaced by the value `9`, i.e., the value of the expression "`4*nine`" is `36`.

Of course, it is more interesting to define functions depending on some input arguments. For instance, a function to compute the square value of a given number can be defined by

```
square x = x*x
```

Now it is time to make some remarks about the syntax of Curry (which is actually very similar to Haskell [12]). The names of functions and parameters usually start with a lowercase letter followed by letters, digits and underscores. The application of a function f to an expression e is denoted by juxtaposition, i.e., by "$f\ e$". An exception are the *infix operators* like `+` or `*` that can be written between their arguments to enable the standard mathematical notation for arithmetic expressions. Furthermore, these operators are defined with the usual associativity and precedence rules so that an expression like "`2+3+4*5`" is interpreted as `((2+3)+(4*5))`. However, one can also enclose expressions in parenthesis to enforce the intended grouping.

If we write the definitions of `nine` and `square` with a standard text editor into a file (note that each definition must be written on a separate line starting in the first column) named "`firstprog.curry`", we can load (and compile) the program into our environment by the command

```
prelude> :l firstprog
```

which reads and compiles the file "`firstprog.curry`" and makes all definitions in this program visible in the environment. After the successful processing of this program, the environment shows the prefix to the input line as

```
firstprog>
```

indicating that the program "`firstprog`" is currently loaded. Now we can use the definitions in this program in the expressions to be evaluated:

```
firstprog> square nine
Result: 81 ?
```

If we change our currently loaded program, we can easily reload the new version by typing "`:r`". For instance, if we add the definition "`two = 2`" to our file "`firstprog.curry`", we can reload the program as follows:

```
firstprog> :r
...
firstprog> square (square two)
Result: 16 ?
```

Functions containing only a single arithmetic expression in the right-hand side of their defining equations might be useful abstractions of complex expressions but are generally only of limited use. More interesting functions can be written using conditional expressions. A *conditional expression* has the general form "`if` c `then` e_1 `else` e_2" where c is a Boolean expression (yielding the value `True` or `False`). A conditional expression is evaluated by evaluating the condition c first. If its value is `True`, the value of the conditional is the value of e_1, otherwise it is the value of e_2. For instance, the following rule defines a function to compute the absolute value of a number:

```
abs x = if x>=0 then x else -x
```

Using recursive definitions, i.e., rules where the defined function occurs in a recursive call in the right-hand side, we can define functions whose evaluation requires a non-constant number

of evaluation steps. For instance, the following rule defines the factorial of a natural number [Program]:

```
fac n = if n==0 then 1
                else n * fac(n-1)
```

Note that function definitions can be put in several lines provided that the subsequent lines start in a column greater than the column where the left-hand side starts (this is also called the *layout* or *off-side* rule for separating definitions).

You might have noticed that functions are defined by rules like in mathematics without providing any type declarations. This does not mean that Curry is an untyped language. On the contrary, Curry is a *strongly typed language* which means that each entity in a program (e.g., functions, parameters) has a type and ill-typed combinations are detected by the compiler. For instance, expressions like "`3*True`" or "`fac False`" are rejected by the compiler. Although type annotations need not be written by the programmer, they are automatically inferred by the compiler using a *type inference algorithm*. Nevertheless, it is a good idea to write down the types of functions in order to provide at least a minimal documentation of the intended use of functions. For instance, the function `fac` maps integers into integers and so its type can be specified by

```
fac :: Int -> Int
```

(`Int` denotes the predefined type of integers; similarly `Bool` denotes the type of Boolean values). If one is interested in the type of a function or expression inferred by the type inference algorithm, one can show it using the command "`:t`" in PAKCS:

```
absfac> :t fac
fac :: Int -> Int
absfac> :t  abs 3
(abs 3) :: Int
```

A useful feature of Curry (as well as most functional and logic programming languages) is the ability to define functions in a *pattern-oriented style*. This means that we can put values like `True` or `False` in arguments of the left-hand side of a rule and define a function by using several rules. The rule that matches the pattern of left-hand side will be called. For instance, instead of defining the negation on Boolean values by the single rule

```
negate x = if x==True then False
                      else True
```

we can define it by using two rules, each with a different pattern (here we also add the type declaration):

```
negate :: Bool -> Bool
negate False = True
negate True  = False
```

The pattern-oriented notation becomes very useful in combination with more complex data structures, as we will see later.

One of the distinguishing features of Curry in comparison to functional languages is its ability to *search for solutions*, i.e., to compute values for the arguments of functions so that the functions can be evaluated. For instance, consider the following definitions of some functions on Boolean values contained in the prelude (note that Curry also allows functions defined as infix operators, i.e., "`x && y`" denotes the application of function `&&` to the arguments `x` and `y`):

```
False && _  =  False
True  && x  =  x

False || x  =  x
True  || _  =  True

not False   =  True
not True    =  False
```

The underscore "`_`" occurring in the rules for `&&` and `||` denotes an arbitrary value, i.e., such an *anonymous variable* is used for argument variables that occur only once in a rule.

We can use these definitions to compute the value of a Boolean expression:

```
prelude> True && (True || (not True))
Result: True ?
```

However, we can do more and use the same functions to compute Boolean values for some (initially unknown) arguments:

```
prelude> X && (Y || (not X))
Free variables in goal: X, Y
Result: True
Bindings:
X=True
Y=True ?
```

Note that the initial expression contains the *free variables* `X` and `Y` as arguments. Free variables in initial expressions must start with an uppercase letter and denote an "unknown" value. They are instantiated (i.e., replaced by some concrete values) so that the instantiated expression is evaluable. As we have seen above, replacing both `X` and `Y` by `True` makes the expression reducible to `True`. Therefore, the Curry system shows the result `True` together with the bindings (i.e., instantiations) of the free variables it has done to compute this value.

In general, there is more than one possibility to instantiate the arguments, e.g., the Boolean variables `X` and `Y` can be instantiated to `True` or `False`. This leads to different solutions which can be printed one after the other by typing "`;`" (followed by the "enter" key) after the question mark. Thus, we can show all solutions to the initial expression as follows:

```
prelude> X && (Y || (not X))
Free variables in goal: X, Y
Result: True
Bindings:
```

```
X=True
Y=True ? ;
Result: False
Bindings:
X=True
Y=False ? ;
Result: False
Bindings:
X=False
Y=Y ? ;
No more solutions.
```

The last solution shows that the initial expression has the value `True` provided that `X` is instantiated to `True` but `Y` can be arbitrary (i.e., `Y` is not instantiated). The final line indicates that there are no more solutions to the initial expression. This situation can also occur if functions are partially defined, i.e., there is a call to which no rule is applicable. For instance, assume that we define the function `pneg` by the single rule [Program]

```
pneg True = False
```

then there is no rule to evaluate the call "`pneg False`":

```
bool> pneg False
No more solutions.
```

As we have seen in the Boolean example above, Curry can evaluate expressions containing free variables by guessing values for the free variables so that the expression becomes evaluable (the concrete strategy used by Curry will be explained later, but don't worry: Curry is based on an optimal evaluation strategy [3] that performs these instantiations in a goal-oriented manner). However, we might not be interested to see all possible evaluations but only those that lead to a required result. For instance, we might be only interested to compute instantiations in a Boolean formula so that the formula becomes true. For this purpose, Curry offers *constraints*, i.e., formulas that are intended to be solved (instead of computing an overall value). One of the basic constraints supported by Curry is equality, i.e., "e_1 `=:=` e_2" denotes an *equational constraint* which is solvable whenever the expressions e_1 and e_2 (which must be of the same type) can be instantiated so that they are evaluable to the same value. For instance, the constraint "`1+4=:=5`" is solvable, and the constraint "`2+3=:=x`" is solvable if the variable `x` is instantiated to `5`. Now we can compute positive solutions to a Boolean expression by solving a constraint containing `True` on one side:

```
prelude> (X && (Y || (not X))) =:= True
Free variables in goal: X, Y
Result: success
Bindings:
X=True
Y=True ? ;
No more solutions.
```

Note that "`success`" denotes the trivial, always satisfiable constraint, i.e., a result like "`success`" indicates that the constraint is satisfied with respect to the computed instantiations.

Curry allows the definition of functions by several rules and is able to search for several solutions. We can combine both features to define functions that yield more than one result for a given input. Such functions are called *non-deterministic* or *set-valued functions*. A simple example for a set-valued function is the following function `choose` which yields non-deterministically one of its arguments as a result [Program]:

```
choose x y = x
choose x y = y
```

With this function we could have several results for a particular call:

```
choose> choose 1 3
Result: 1 ? ;
Result: 3 ? ;
No more solutions.
```

We can use `choose` to define other set-valued functions:

```
one23 = choose 1 (choose 2 3)
```

Thus, a call to `one23` delivers one of the results `1`, `2`, or `3`. Such a function might be useful to specify the domain of values for which we want to solve a constraint. For instance, to search for values $x \in \{1, 2, 3\}$ satisfying the equation $x + x = x * x$, we can solve this constraint (c_1 `&` c_2 denotes the conjunction of the two constraints c_1 and c_2):

```
choose> X=:=one23 & X+X=:=X*X
Free variables in goal: X
Result: success
Bindings:
X=2 ?
```

Set-valued functions are often a reasonable alternative to flexible functions in order to search for solutions. The advantages of set-valued functions will become clear when we have discussed the (demand-driven) evaluation strategy in more detail.

This chapter is intended to provide a broad overview of the main features of Curry and the use of an interactive programming environment so that one can easily try the subsequent examples. In the next chapter, we will discuss the features of Curry in more detail.

Chapter 3

Main Features of Curry

3.1 Overview

The major elements declared in program are *functions* and *data structures*.

- A *function* defines a computation similar to an expression. However, the expression computed by a function has a name and is often parameterized. These characteristics enable you to execute the same computation, possibly with different parameters, over and over in the same program by simply invoking the computation's name and setting the values of its parameters. A function also provides a *procedural abstraction*. Rather than coding a computation by means of a possibly complicated expression, you can factor out portions of this computation and abstract them by their names.

- A *data structure* is a way to organize data. For example, you can record the movements of your bank account in a column in which deposits are positive numbers and withdrawals are negative numbers. Or you can record the same movements in two columns, one for deposits and another for withdrawals, in which all numbers are positive. With the second option, the columns rather than the signs specialize the meaning of the numbers. The way in which information is organized may ease some computations, such as retrieving portions of information, and is intimately related, through pattern matching, to the way in which functions are coded.

This section describes in some detail both of these features and a number of related concepts. Curry has some additional features not described in this section. Since they are useful to support particular programming tasks, we introduce them later when we discuss such programming techniques.

3.2 Expressions

A function can be regarded as a parameterized expression with a name. Thus, we begin by explaining what an expression is and how it is used. Most expressions are built from simpler subexpressions, a situation that calls for a recursive, or inductive, definition.

An *expression* is either a symbol or literal value or is the application of an expression to another expression.

A symbol or literal value is referred to as an *atom*. For example, numbers and the Boolean symbols "`True`" and "`False`" are examples of atoms. Atoms constitute the most elementary expressions. These elementary expressions can be combined to create more complex expressions, e.g., "`2 + 3`" or "`not True`". The combination is referred to as a *function application*. Since a function application is a very common activity, it is convenient to denote it as simply as possible. This convenience is obtained to the extreme by writing the two expressions one near the other as in "`not True`". This notation is referred to as *juxtaposition*.

In the above expressions, the symbols "`+`" and "`not`" are operations. Both are predefined in the `prelude`. Although conceptually the symbols "`+`" and "`not`" are alike, syntactically they differ. The symbol "`+`" is a *infix operator* as in the ordinary mathematical notation. Infix operators have a *precedence* and an *associativity* so that the expression "`2 + 3 * 4`" is understood as "`2 + (3 * 4)`" and the expression "`4 - 3 - 2`" is understood as "`(4 - 3) - 2`". The precedence and associativity of an infix symbol are defined in a program by a declaration. The following declarations, from the `prelude`, define these parameters for some ordinary arithmetic operations:

```
infixl 7 *, `div`, `mod`
infixl 6 +, -
infix  4  <, >, <=, >=
```

For example, the precedence of the addition and subtraction operators is 6 and their associativity is left. The relational operators have precedence 4 and are not associative. Operators with a higher precedence bind stronger, i.e., the expression "`4 < 2 + 3`" is interpreted as "`4 < (2 + 3)`".

Infix declarations must always occur at the beginning of a program. The precedence of an operator is an integer between 0 and 9 inclusive. The associativity of an operator is either *left*, denoted by the keyword "`infixl`" or *right*, denoted by the keyword "`infixr`". Non-associative infix operators are declared using the keyword "`infix`".

Most often, an infix operator is any user-defined sequence of characters taken from the set "`~!@#$%^&*+-=<>?./|\:`". Alphanumeric identifiers can be defined and used as infix operators if they are surrounded by backquotes, as "`` `div` ``" and "`` `mod` ``" in the previous declaration. For example, for any integer value x, the following expression evaluates to x itself.

x `` `div` 2 * 2 + `` x `` `mod` 2 ``

Non-infix symbols are *prefix*. They are applied by prefixing them to their arguments as in "`not True`".

Exercise 1 Define a predicate, read as "factors" and denoted by the infix operator "`./.`", that tells whether an integer is a factor of another integer. The predicate should work for every

input and `0` should not be a factor of any integer. The operator should be non-associative and have precedence 7. [Answer]

A symbol, whether infix or prefix, can only be applied to values of an appropriate type. As one would expect, the Boolean negation operator can be applied only to a Boolean value. For example, the expression "`not 2`" is an error. The compiler/interpreter would report that the expression is incorrectly typed. We will discuss types in more detail after presenting data declarations.

The application of an expression to another is a binary operation. The expression that is being applied is referred to as the *function* of the application. The other expression is referred to as the *argument*. Thus, in "`not True`", "`not`" is the function and "`True`" is the argument. The situation is slightly more complicated for infix operations. The reading of "`2+3`" is that the function "`+`" is applied to the expression "`2`". The result is a function which is further applied to the expression "`3`".

Expressions can also be conditional, i.e., depend on the value of a Boolean expression. Such *conditional expressions* have the form "`if` b `then` e_1 `else` e_2". The value of this expression is the value of e_1 if b evaluates to `True`, or the value of e_2 if b evaluates to `False`. Thus, the value of "`if 3>4 then 2*2 else 3*4`" is `12`.

3.3 Predefined Types

A *type* is a set of values. Ubiquitous types, such as integers or characters, are predefined by most programming languages. Curry makes no exception. These types are referred to as *builtin* and are denoted with a familiar, somewhat special, syntax. Both the availability of builtin types and their characteristics may depend on a specific implementation of Curry. The following table summarizes some types available in PAKCS.

Type	**Declaration**	**Examples**
Integer	`Int`	`...,-2,-1,0,1,2,...`
Boolean	`Bool`	`False, True`
Character	`Char`	`'a','b','c',...,'\n',...`
String	`String`	`"hello", "world"`
List of τ	`[`τ`]`	`[], [0,1,2], 0:1:2:[]`
Success	`Success`	`success`
Unit	`()`	`()`

The details of these types are found in the PAKCS User Manual. Below, we only outline a few crucial characteristics of the builtin types. The integers have arbitrary precision. Some frequently used non-printable characters are denoted, as in other popular programming languages, by escape sequences, e.g., *newline* is denoted by `\n`. The type *List* represents sequences of values. This type is polymorphic, i.e., for any type τ, the type list of τ, denoted by "`[`τ`]`", is a type whose instances are sequences of instances of τ. The last two examples in the *List* row of the table denote a list of integers, their type denoted by "`[Int]`". The notation of lists will be further discussed later. The type *Success* has no visible literal values

and is intended to denote the result of successfully solved constraints. Hence, expressions of type *Success* are also called *constraints*. Usually, they occur in conditions and are checked for satisfiability. The symbol *success* is a predefined function that denotes an always satisfiable constraint. The symbol "`()`" denotes the unit type as well as the only element of this type. The unit type is useful in situations where the return value of a function is not important. Another useful type available in PAKCS, the *tuple*, will be described later.

3.4 Predefined Operations

Many frequently-used functions and infix operators, similar to frequently-used types, are predefined in Curry. Some of these can be found in the "`prelude`", a Curry source program automatically loaded when the compiler/interpreter starts. A few others are so fundamental that they are built into the language. Some of these functions and operators are shown in the following table.

Description	**Ident.**	**Fix.**	**Prec.**	**Type**
Boolean equality	`==`		4	`a -> a -> Bool`
Constrained equality	`=:=`		4	`a -> a -> Success`
Boolean conjunction	`&&`	R	3	`Bool -> Bool -> Bool`
Boolean disjunction	`||`	R	2	`Bool -> Bool -> Bool`
Sequential conjunction	`&>`	R	0	`Success -> Success -> Success`
Parallel conjunction	`&`	R	0	`Success -> Success -> Success`

The *Boolean equality* applied to expressions u and v, i.e., u `==` v, returns "`True`" if and only if u and v can be evaluated to the same *value*—a precise definition will be given later. If the evaluation of u and/or v ends in an expression that still contains functions, e.g., `1 ‘div‘ 0` the computation *fails* and neither "`True`" nor "`False`" is returned.

The *constrained equality* applied to expressions u and v, i.e., u `=:=` v, succeeds if and only if u and v can be evaluated to the same *value*—a precise definition will be given later. Otherwise, the computation *fails* and no value is returned. A key difference between the Boolean and the constrained equalities is how they evaluate expressions containing variables. This will be discussed in some detail in Section 3.13.1.

The *Boolean conjunction* applied to expressions u and v, i.e., u `&&` v, returns "`True`" if and only if u and v can be evaluated to "`True`".

The *Boolean disjunction* applied to expressions u and v, i.e., u `||` v, returns "`True`" if and only if u or v can be evaluated to "`True`".

The *sequential conjunction* applied to expressions u and v, i.e., u `&>` v, sequentially evaluates first u and then v. If both succeeds, the evaluation succeeds; otherwise it fails.

The *parallel conjunction* applied to expressions u and v, i.e., u `&` v, evaluates u and v concurrently. If both succeeds, the evaluation succeeds; otherwise it fails.

Curry predefines many more functions and operations, e.g., the standard arithmetic and relational operators on numbers. A complete list can be found both in the Report and the

"prelude".

3.5 Functions

3.5.1 Basic concepts

A program function abstracts a function in the mathematical sense. A function is a device that takes arguments and returns a result. The result is obtained by evaluating an expression which generally involves the function's arguments. The following function computes the *square* of a number.

```
square x = x * x
```

The symbols "square" is the name or *identifier* of the function. The symbol "x" is the function's *argument*. The above declaration is referred to as a *rewrite rule*, or simply a rule, defining a function. The portion of the declaration to the left of the symbol "=" is the rule's *left-hand side*. The expression "x * x" is the rule's *right-hand side*.

When the "square" symbol is applied to an expression, e.g., "2 + 3", this expression is *bound* to the argument "x". The result of the application is "(2 + 3) * (2 + 3)", i.e., the body in which the argument is replaced by its binding. Thus:

```
prelude> square (2+3)
Result: 25 ?
```

Functions can be *anonymous*, i.e., without a name. An anonymous function is useful when a function is referenced only once. In this case, the reference to the function can be replaced by the expression defining the function. In the following example:

```
result = (\x -> x * x) (2+3)
```

the value of result is 25. It is obtained by applying the expression (\x -> x * x), an anonymous function, to (2+3), its argument. An anonymous function definition has the following structure:

args -> *left-hand side*

A more motivating example of anonymous function is presented in Section 3.10

The evaluation of any expression, in particular of a function application, is *lazy*. This means that the computation of any expression, including the subexpressions of a larger expression, is delayed until the expression's value is actually needed. The exact meaning of "actually needed" is quite technical, but the intuitive meaning suffices for our purposes. Many programming languages, such as C and Java, adopt this evaluation strategy, under the name of *short circuit*, only for boolean expressions.

We will discuss this issue in more detail later. Although the lazy evaluation strategy is conceptually simpler than any other strategy, many traditional programming languages evaluate the arguments of a function call eagerly, i.e., before applying a function to its arguments. This fact is sometimes a source of confusion for the beginner.

3.5.2 Pattern Matching

The definition of a function can be broken into several rules. A single rule would suffice in many cases. However, several rules allows a definition style, called *pattern matching*, which is easier to code and understand. This feature allows a function to dispatch the expression to be returned depending on the values of its arguments. The following example shows the definition of the boolean negation function "`not`":

```
not True = False
not False = True
```

The above definition is equivalent to the following one which does not use pattern matching but relies on a conditional expression:

```
not x = if x == True then False else True
```

Pattern matching is particularly convenient for functions that operate on algebraic datatypes. We will further discuss this aspect after discussing data declarations.

3.5.3 Conditions

Each rule defining a function can include one or more *conditions*. In the most general case, a rule has the following structure:

$$\begin{aligned} functId\ \ arg_1 \ldots arg_m \mid\ & cond_1 = expr_1 \\ \mid\ & \ldots \ = \ \ldots \\ \mid\ & cond_n = expr_n \end{aligned}$$

A condition is tested after binding the arguments of a call to the corresponding arguments in the left-hand side of the rule. The function is applied to the arguments only if the condition holds. A condition is an expression of type *Boolean* or *Success*. All the conditions of a rule must be of the same type. When the conditions of a rule are Boolean expressions, the conditions are tested in their textual order. Thus, the first right-hand side with a condition evaluable to `True` is taken. Furthermore, the last condition can be the reserved word "`otherwise`" which always holds, i.e., it holds regardless of any value of the arguments. The following example shows a plausible definition of the maximum of two numbers:

```
max x y | x < y     = y
        | otherwise = x
```

When the conditions of a rule are *constraints* (i.e., expressions of type *Success*), all conditions are checked and those right-hand sides with a satisfiable condition (possibly more than one) are considered for evaluation.

3.5.4 Non-determinism

Functions can be *non-deterministic*. Non-deterministic functions are not functions in the mathematical sense because they can return different values for the same input. For example, a hospital's information system defines which days a doctor is on-call with a non-deterministic function:

```
oncall Joan = Monday
oncall Joan = Wednesday
oncall Richard = Monday
oncall Luc = Tuesday
...
```

The value of "`oncall Joan`" can be either "`Monday`" or "`Wednesday`". The programmer cannot select which of the two values will be computed. Non-deterministic functions support a programming style similar to that of logic programs, while preserving some advantages of functional programs such as expression nesting and lazy evaluation. In particular, some strong properties concerning the evaluation of ordinary function hold also for non-deterministic functions [2]. For example, suppose that "`today`" holds which day of the week is today. A predicate, "`available`", telling whether its argument, a doctor, is available at the current time is coded as:

```
available x | oncall x == today = True
            | otherwise         = False
```

Without non-determinism, coding "`oncall`" would require some data structure, e.g., the list of days in which each doctor is on-call, and defining "`available`" would become more complicated.

Non-determinism is a powerful feature. In programming, as in other aspects of life, power must be exercised with some care. A non-deterministic program is appropriate only if all its possible outputs are equally desirable. If some outputs are more desirable than others, the program should be (more) deterministic. In this case, non-determinism could be conveniently used internally by the program to generate plausible results which can then be selected according to desirability.

Exercise 2 In a manufacturing plant two specialized tasks, `cut` and `polish`, are executed only by specialized workers, `Alex`, `Bert` and `Chuck`. Not every worker can execute every task. Only `Alex` and `Bert` are able to `cut`, whereas only `Bert` and `Chuck` are able to `polish`. Code a non-deterministic function, `assign`, that assigns to a task a worker that can execute it. [Answer]

3.6 User-defined Types

A *type* is a set of values. Some common types, presented in Section 3.3, are built into the language and the programmer does not declare them. All other types used in a program must be declared by the programmer. The classification of some types as builtin vs. user-defined is only a matter of convenience. Builtin and user-defined types are conceptually very similar. In fact, the declaration of some builtin types could have been left to the programmer. For example:

```
data Boolean = False | True
```

is exactly how the builtin "`Boolean`" type would be declared if it were not builtin. In this declaration, the identifier "`Boolean`" is referred to as a *type constructor*, whereas the iden-

tifiers "`False`" and "`True`" are referred to as *data constructors*. The following declarations, very similar to the previous one, define plausible types "`WeekDay`" and "`PrimaryColor`".

```
data WeekDay = Monday | Tuesday | Wednesday | Thursday | Friday
data PrimaryColor = Red | Green | Blue
```

All these types are finite, i.e., they define a finite set of values, and resemble enumerated types in the Pascal or C languages.

The declaration of an infinite type is similar, but as one should expect, must be (directly or indirectly) recursive. The following declaration defines a binary tree of integers. We recall that the typical definition of this type says that a *binary tree* is either a *leaf* or it is a *branch* consisting of two binary trees. Not surprisingly, this definition is recursive which accounts for an infinity of trees. The words "leaf" and "branch" are conventional names used to distinguish the two kinds of trees and have no other implicit meaning. Often, branches include a *decoration*, a value of some other arbitrary type. If a tree T is a branch, the two trees in the branch are referred to as the left and right children of T. A declaration defining binary trees where the decoration is an integer follows:

```
data IntTree = Leaf | Branch Int IntTree IntTree
```

All the following expressions are values of type "`IntTree`":

```
Leaf
Branch 0 Leaf Leaf
Branch 7 (Branch 5 Leaf Leaf) (Branch 9 Leaf Leaf)
```

The first tree is a leaf and therefore it contains no decoration. The second tree contains a single decoration, "`0`", and two children both of which are leaves. The third tree contains three decorations. Binary trees are interesting because many efficient searching and sorting algorithms are based on them.

User-defined types can be parameterized by means of other types similar to the builtin type list introduced in Section 3.3. These types are called *polymorphic*. For example, if the type of the decoration of a binary tree is made a parameter of the type of the tree, the result is a polymorphic binary tree. This is achieved by the following declaration [Program]:

```
data BinTree a = Leaf | Branch a (BinTree a) (BinTree a)
```

The identifier "`a`" is a *type variable*. Observe that the type variable not only defines the type of the decoration, but also the type of the subtrees occurring in a branch. In other words, the type that parameterizes a tree also parameterizes the children of a tree. The type variable can be implicitly or explicitly bound to some type, e.g., "`Int`" or "`WeekDay`" defined earlier. For example, a function that looks for the string "`Curry`" in a tree of strings is defined as [Program]:

```
findCurry Leaf = False
findCurry (Branch x l r) = x == "Curry" || findCurry l || findCurry r
```

The type of the argument of function "`findCurry`" is "`BinTree String`". The binding of type "`String`" to the type variable of the definition of the polymorphic type "`BinTree`" is

automatically inferred from the definition of function "`findCurry`".

A polymorphic type such as "`BinTree`" can be specialized by binding its variable to a specific type by an explicit declaration as follows [Program]:

```
type IntTree = BinTree Int
```

where "`type`" is a reserved word of the language. This declaration defines "`IntTree`" as a synonym of "`BinTree Int`". The synonym can be used in *type declarations* to improve readability. The following example defines a function that tallies all the decorations of a tree of integers [Program]:

```
total :: IntTree -> Int
total Leaf = 0
total (Branch x l r) = x + total l + total r
```

Exercise 3 Pretend that list is not a builtin type, with special syntax, of the language. Define your own type list. Define two functions on this type, one to count how many elements are in a list, the other to find whether some element is in a list. [Answer]

3.7 Lists

The type list is builtin or predefined by the language. This type could be easily defined by the programmer, see Exercise 3, except that the language allows the representation of lists in a special notation which is more agile than that that would be available to the programmer. The following statement defines important concepts of a list:

> A *list* is either *nil* or it is a *cons* consisting of an element, referred to as the *head* of the list, and another list, referred to as the *tail* of the list.

The nil list is denoted by "`[]`", which is read "nil". A cons list, with head h and tail t is denoted by "h`:`t". The infix operator "`:`", which is read "cons", is right associative with precedence 5. A list can also be denoted by enumerating its elements, e.g., "`[u,v,w]`" is a list containing three elements, "`u`", "`v`" and "`w`", i.e., it is just another notation for "`u:v:w:[]`". The number of elements is arbitrary. The elements are enclosed in brackets and separated by commas.

The following functions concatenate two lists and reverse a list, respectively. The "`prelude`" defines the first one as the infix operator "`++`" and the second one, much more efficiently, as the operation "`reverse`".

```
conc []     ys = ys
conc (x:xs) ys = x : conc xs ys

rev []     = []
rev (x:xs) = conc (rev xs) [x]
```

Several *ad hoc* notations available for lists are described in Sections 4.2.3 and 4.2.4.

A key advantage of these special notations for lists is a reduction of the number of parentheses needed to represent list expressions in a program. This claim can be easily verified by comparing the builtin notation with the ordinary notation which was the subject of Exercise 3.

3.8 Strings

Although "`String`" is a predefined type (see Section 3.3), there are no special operations on strings. The reason is that "`String`" is just another name for "`[Char]`", i.e., strings are considered as lists of characters. In addition, Curry provides a handy notation for string constants, i.e., the string constant

```
"hello world"
```

is identical to the character list

```
['h','e','l','l','o',' ','w','o','r','l','d']
```

Thus, any operation applicable to arbitrary lists can also be applied to strings. For instance, the prelude defines an infix operator "`++`" to concatenate lists and the function "`reverse`" to reverse the order of all lists elements (similarly to `conc` and `rev` in Section 3.7). Thus, we can also use them to operate on strings:

```
prelude> "Hi"++"Hi"
Result: "HiHi" ?
prelude> reverse "hello"
Result: "olleh" ?
```

3.9 Tuple

The word "tuple" is a generic name for a family of related types. A tuple in a program is similar to a tuple in mathematics, i.e., a fixed length sequence of values of possibly different types. Examples of tuples are pairs and triples. They could be defined by the programmer as follows:

```
data Pair a b = Pair a b
data Triple a b c = Triple a b c
```

These types are polymorphic. Observe the two occurrences of the identifiers "`Pair`" and "`Triple`" in the above declarations. The occurrence to the left names a type constructor, whereas the occurrence to the right names a data constructor. These symbols are *overloaded*. However, this kind of overloading causes no problems since type expressions are clearly separated from value expressions. The type variables "`a`", "`b`"... can be bound to different types.

For example, the information system of a "Big & Tall" shoe store declares a function that defines the largest size and width of each model [Program]:

```
data Width = C | D | E | EE | EEE | EEEE
largest "New Balance 495" = Pair 13 EEE
largest "Adidas Comfort"  = Pair 15 EE
...
```

The language predefines tuples and denotes them with a special notation similar to the standard mathematical notation. Using predefined tuples, the above function is coded as:

```
largest "New Balance 495" = (13,EEE)
largest "Adidas Comfort"  = (15,EE)
...
```

Tuples are denoted by a fixed-length sequence of comma-separated values between parentheses. There is no explicit data constructor identifier. The type of a tuple is represented as a tuple as well, e.g., the type of "`largest`" is reported by the interpreter as:

```
BigTall> :t largest
largest :: String -> (Int,Width)
BigTall>
```

3.10 Higher-Order Computations

The arguments of a function can be functions themselves. This feature is banned or restricted by many programming languages. E.g., in C only a *pointer* to a function can be passed as a parameter to another function. For the same purpose, C++ uses *templates* and Java uses *interfaces*. In Curry, no special construct or concept is necessary.

A function that takes an argument of function type is referred to as a *higher-order* function. Loosely speaking, a higher-order function is computation parameterized by another computation. We show the power of this feature with a simple example. The function "`sort`", shown below, takes a list of numbers and sorts them in ascending order. On non-empty arguments, the function "`sort`" recursively sorts the tail and inserts the head at the right place in the sorted tail. This algorithm becomes inefficient as lists grow longer, but it is easy to understand [Program]:

```
sort []     = []
sort (x:xs) = insert x (sort xs)

insert x [] = [x]
insert x (y:ys) | x <= y    = x : y : ys
                | otherwise = y : insert x ys
```

To sort a list in descending order or to sort a list of a different type, a new function must be coded.

An alternative is to code a sort function where the ordering criterion is an argument. The overall structure of the function is the same. The new argument, the first one of each function, is denoted by "`f`". This argument is a function that takes two arguments and

returns "`True`" if and only if the first argument must appear before the second argument in the output list [Program]:

```
sort _ []     = []
sort f (x:xs) = insert f x (sort f xs)

insert _ x [] = [x]
insert f x (y:ys) | f x y     = x : y : ys
                  | otherwise = y : insert f x ys
```

For example:

```
HOInsertionSort> sort (<=) [3,5,1,2,6,8,9,7]
Result: [1,2,3,5,6,7,8,9] ?
HOInsertionSort> sort (>) [3,5,1,2,6,8,9,7]
Result: [9,8,7,6,5,3,2,1] ?
```

In the above expressions, the operators "`<=`" and "`>`" are the functional arguments. The parentheses around them are necessary, since these functions are identified by infix operators. Without parentheses, the expression "`sort <= [3,5,1,2,6,8,9,7]`" would test whether the left argument of "`<=`" is smaller than the right argument, which is meaningless.

Observe that the first version of the "`sort`" function constrains the elements of the input list to be numbers, since these elements are arguments of "`<=`". In the second, higher-order version, the type of the elements of the input list is unconstrained. Thus, the function can be applied to lists of any type as long as a suitable ordering criterion for the type of the list elements is provided.

Higher-order computations involve a functional argument. Sometimes, the corresponding argument in a call, which is a function, is referenced only in the call itself. In this case, it is appropriate to use an anonymous function. For example, suppose that an elementary school information system represents classes with a grade and a section. The grade is a number in the range 1 through 5 and the section is a letter, `a`, `b` ... The following ordering criterion sorts the classes in a "natural" (lexicographic) order [Program]:

```
sortClasses l = sort lex l
   where lex (x,y) (u,v) = x<u || x==u && ord y <= ord v
```

A more compact and informative formulation uses an anonymous function as follows [Program]:

```
sortClasses l = sort (\(x,y) (u,v) -> x<u || x==u && ord y <= ord v) l
```

Observe that pattern matching is normally used in the definition of the above anonymous function.

3.11 Lazy Evaluation

The *evaluation* of an expression t is the process of obtaining a value v from t.

A *value* is an expression consisting only of builtin literals and/or data constructors

and/or variables.

The value v is obtained from t by replacing an instance of the left-hand side of a rule with the corresponding instance of the right-hand side. For example, referring to the function *square* defined in Section 3.5.1:

```
square x = x * x
```

an instance of *square* x is replaced with the corresponding instance of $x * x$. For example, $4 +$ *square* $(2 + 3)$ is replaced by $4 + (2 + 3) * (2 + 3)$.

The evaluation of an expression t proceeds replacement after replacement until an expression v in which no more replacements are possible is obtained. If v is not a value, the evaluation fails, otherwise v is the result of a computation of t. For example, the following function *head* computes the first element of a (non-null) list:

```
head (x:_) = x
```

An attempt to evaluate "`head []`" fails, since no replacement is possible and the expression is not a value since it contains a function.

Often, an expression may contain several distinct replaceable subexpressions, e.g., from $(2 + 3) * (2 + 3)$ we can obtain both $5 * (2 + 3)$ and $(2 + 3) * 5$. Even a single subexpression may allow several distinct replacements when non-deterministic functions are involved. The order in which different subexpressions of an expression are replaced is not determined by a program. The choice is made by an evaluation *strategy*. The semantics of the language guarantees that any value obtainable from an expression is eventually obtained. This property is referred to as the *completeness* of the evaluation. To ensure this completeness, expressions must be evaluated lazily. A lazy strategy is a strategy that evaluates a subexpression only if its evaluation is unavoidable to obtain a result. The following example clarifies this delicate point.

The following function computes the list of all the integers beginning with some initial value n [Program]:

```
from n = n : from (n+1)
```

An attempt to evaluate "`from 1`" aborts with a memory overflow since the "result" would be the infinite term:

```
[1,2,3,...
```

However, the function "`from`" is perfectly legal. The following function returns the n-th element of a list:

```
nth n (x:xs) = if n==1 then x else nth (n-1) xs
```

The expression "`nth 3 (from 1)`" evaluates to `3` despite the fact that "`from 1`" has no (finite) value:

```
lazy> nth 3 (from 1)
Result: 3 ?
```

The reason is that only the third element of "`from 1`" is needed for the result. All the other elements, in particular the infinite sequence of elements past the third one, do not need to be evaluated.

Infinite data structures are an asset in the conjunction with lazy evaluation. Programs that use infinite structures are often simpler than programs for the same problem that use finite structures. E.g., a function that computes a (finite) prefix of "`[1,2,3,...`" is more complicated than "`from`". Furthermore, the functions of the program are less interpedendent and consequently more reusable. E.g., the following function, initially applied to `0` and `1`, computes the (infinite) sequence of the Fibonacci numbers:

```
fibolist x0 x1 = x0 : fibolist x1 (x0+x1)
```

The function "`nth`" can be reused to compute the n-th Fibonacci number through the evaluation of the expression "`nth` n `(fibolist 0 1)`", e.g.:

```
lazy> nth 5 (fibolist 0 1)
Result: 3 ?
```

The evaluation strategy of the PAKCS compiler/interpreter, which is used for all our examples, is lazy, but incomplete. The strategy evaluates non-deterministic choices sequentially instead of concurrently.

All the occurrences of same variables are shared. This design decision has implications both on the efficiency and the result of a computation. For example, consider again the following definition:

```
square x = x * x
```

The evaluation of say `square` t goes through $t * t$. Without sharing, t would be evaluated twice, each evaluation independent of the other. If t has only one value, the double evaluation would be a waste. If t has more then one value, this condition will be discussed in Section 3.12.1, sharing produces the same value for both occurrences.

3.12 Local Definitions

The syntax of Curry implicitly associates a scope to each identifier, whether a function, a type, a variable, etc. Roughly speaking the scope of an identifier is where in a program the identifier can be used. For example, the scope of a variable occurring in the left-hand side of a rule is the rule itself, which includes the right-hand side and the condition, if any. In the following code:

```
square x = x * x
cube   x = x * square x
```

the variable identified by "`x`" in the definition of "`square`" is completely separated from the variable identified by "`x`" as well in the definition of "`cube`". Although these variables share the same name, they are completely independent of each other.

Curry is *statically scoped*, which means that the scope of an identifier is a static property

of a program, i.e., the scope depends on the textual layout of a program rather than on an execution of the program.

> The *scope* of an identifier is the region of text of a program in which the identifier can be referenced.

In most cases, the programmer has no control on the scope of an identifier—and this is a good thing. The scope rules are designed to make the job of the programmer as easy and safe as possible. The context in which an identifier occurs determines the identifier's scope. However, there are a couple of situations where the programmer can limit, by mean of syntactical constructs provided by the language, the scope of an identifier. Limiting the scope of an identifier is convenient in some situations. For example, it prevents potential name clashes and/or it makes it clearer that a function is introduced only to simplify the definition of another function. A limited scope, which is referred to as a *local scope*, is the subject of this section.

Curry has two syntactic constructs for defining a local scope: the "`where`" clause and the "`let`" clause. They are explained next.

3.12.1 `Where` clauses

A "`where`" clause creates a scope nested within a rewrite rule. The following example defines an infix operator, "`**`", for integer exponentiation [Program]:

```
infixl 8 **
a ** b | b >= 0 = accum 1 a b
   where accum x y z | z == 0    = x
                     | otherwise = accum aux (y * y) (z `div` 2)
           where aux = if (z `mod` 2 == 1) then x * y else x
```

For example, `2**5` $= 2^5 = 32$. There are several noteworthy points in the above code fragment. The scope of the function "`accum`" is limited to the rewrite rule of "`**`". This is convenient since the purpose of the former is only to simplify the definition of the latter. There would be no gain in making the function "`accum`" accessible from other portions of a program. The function "`accum`" is *nested* inside the function "`**`", which is *nesting* "`accum`".

The rewrite rule defining "`accum`" is conditional. Pattern matching of the arguments and non-determinism can occur as well in local scopes. Finally, there is yet another local scope nested within the rewrite rule of the function "`accum`". The identifier "`aux`" is defined in this scope and can be referenced from either condition or right-hand side of the rewrite rule of the function "`accum`".

The right-hand side of the rewrite rule defining "`aux`" references the variables "`x`", "`y`" and "`z`" that are arguments of "`accum`" rather than "`aux`" itself. This is not surprising since the scope of these variables is the rewrite rule of "`accum`" and "`aux`" is defined within this rule.

The identifier "`aux`" takes no arguments. Because it occurs in a local scope, "`aux`" is considered a local *variable* instead of a *nullary function*. The language does not make this distiction for non-local identifiers, i.e., identifiers defined at the top level. The evaluation of local variables differs from that of local functions. All the occurrences of a variable, whether

or not local, share the same value. This policy may affect both the efficiency of a program execution and the result of computations involving non-deterministic functions. The following example clarifies this subtle point [Program]:

```
coin = 0
coin = 1

g = (x,x) where x = coin

f = (coin,coin)
```

The values of "`g`" are `(0,0)` and `(1,1)` only, whereas the values of "`f`" also include `(0,1)` and `(1,0)`. The reason of this difference is that the two occurrences of "`coin`" in the rule of "`f`" are evaluated independently, hence they may have different values, whereas the two occurrences of "`x`" in the rule of "`g`" are "shared," hence they have the same value.

There is one final important aspect of local scoping. A local scope can declare an identifier already declared in a nesting scope—a condition referred to as *shadowing*. An example of showing is shown below:

```
f x = x where x = 0
```

The variable "`x`" introduced in the `where` clause *shadows* the variable with the same name introduced in the rewrite rule left-hand side. The occurrence of "`x`" in the right-hand side is bound to the former. Hence, the value "`f 1`" is `0`. This situation may be a source of confusion for the beginner. The PAKCS compiler/interpreter detects this situation and warns the programmer as follows [Program]:

```
prelude> :l shadow
Parsing 'shadow.curry'...
prelude.curry: compilation completed.
Warning: shadow.curry:1:15   shadowing symbol "x"
Warning: shadow.curry:1:3   unreferenced variable "x"
shadow.curry: compilation completed with 2 warnings.
translating to ./shadow.flc ...
...
```

The first warning reports that the identifier in line 1, column 15, the variable "`x`" in the local scope, shadows some identifier(s) with the same name. The second warning reports that the identifier in line 1, column 3, the variable "`x`" argument of "`f`", is not used. This is a consequence of its shadowing and gives an important clue that the occurrence of "`x`" in the right-hand side of the rewrite rule of "`f`" is bound to the local variable rather than the argument.

3.12.2 `Let` clauses

A "`let`" clause creates a scope nested within an expression. The concept is very similar to a "`where`" clause, but the granularity of the scope is finer. For example, the program for integer exponentiation presented earlier can be coded using "`let`" clauses as well [Program]:

```
infixl 8 **
a ** b | b >= 0 =
  let accum x y z | z == 0    = x
                  | otherwise =
                       let aux = if (z `mod` 2 == 1) then x * y else x
                       in  accum aux (y * y) (z `div` 2)
  in  accum 1 a b
```

Using a "`let`" declaration is more appropriate than a "`where`" declaration for the definition of operation "`aux`". With a "`let`" declaration, the scope of the identifier "`aux`" is the right-hand side of the second conditional rule of the function "`accum`" instead of the whole rule.

3.12.3 Layout

By contrast to most languages, Curry programs do not use a printable character to separate syntactic constructs, e.g., one rewrite rule from the next. Similar to Haskell, Curry programs use a combination of an end-of-line and the indentation of the next line, if any. A Curry construct, e.g., a "`data`" declaration or a rewrite rule, terminates at the end of a line, unless the following line is more indented. For example, consider the following layout:

```
f = g
 h...
```

Since "`f`" starts in column 1 and "`h`" starts in column 2, the right-hand side of the rule defining "`f`" consists in the application of "`g`" to "`h`" to "`...`" By contrast, with the following layout:

```
f = g
h...
```

the right-hand side of the rule defining "`f`" consists of "`g`" only. Since "`h`" starts in the same column as "`f`", this line is intended as a new declaration.

The layout style described above goes under the name "off-side rule". The examples of Sections 3.12.1 and 3.12.2 shows how the off-side rule applies to "`where`" and "`let`" clauses.

3.13 Variables

Most of the programs discussed so far are functional. They declare data and/or define functions. An execution of the program is the functional-like evaluation of an expression. Curry is a *functional logic* programming language. It adds two crucial features to the model outlined above: non-determinism, which was discussed in Section 3.5.4, and *logic variables*, which are discussed in this section.

3.13.1 Logic variables

A logic variable differs from the variables introduced by the left-hand side of a rewrite rule. A variable introduced by the left-hand side of a rewrite rule stands for any expression (of an appropriate type). For example, the following definition:

```
head (x:xs) = x
```

is read as "for all expressions x and xs the head of (the list) `(x:xs)` is `x`." Since Curry is strongly typed, the type of `xs` must be list, otherwise the program would be invalid, but no other conditions are imposed on `xs`.

A *logic variable* either is a variable occurring in an expression typed by the user at the interpreter prompt or it is a variable in the condition and/or right-hand side of a rewrite rule which does not occur in the left-hand side. We show an example of both. The operation "`==`", called *Boolean equality*, is predefined in Curry. Hence, one can (attempt to) evaluate:

```
prelude> Z==2
```

Every variable in a query, such as "`Z`" in the above example, is a logic variable that initially is not bound to any value. We will discuss shortly why queries with variables may be useful and how variables are handled.

The second kind of logic variable is shown in the following example:

```
path a z = edge a b && path b z   where b free
```

The intuition behind the names tells that in a graph there exists a path from a node a to a node z if there exists an edge from the node a to some node b and a path from the node b to the node z. In the definition, both "`a`" and "`z`" are ordinary (rule) variables, whereas "`b`" is a logic variable. Variables, such as "`b`", which occur in the condition and/or right-hand side of a rule, but not in the left-hand side, are also called *extra variables*. Extra variables must be explicitly declared "`free`" in a "`where`" or "`let`" clause as shown in the example.

3.13.2 Evaluation

The evaluation of expressions containing logic variables is a delicate issue and the single most important feature of functional logic languages. There are two approaches to deal with the evaluation of expressions containing logic variables: *residuation* and *narrowing*.

Let e be an expression to evaluate and v a variable occurring in e. Suppose that e cannot be evaluated because the value of v is not known. Residuation suspends the evaluation of e. If it is possible, we will address this possibility shortly, some other expression f is evaluated in hopes that the evaluation of f will bind a value to v. If and when this happens, the evaluation of e resumes. If the expression f does not exists, e is said to *flounder* and the evaluation of e fails. For example, this is what would happen for the query we showed earlier:

```
prelude> Z==2+2
Free variables in goal: Z
*** Goal suspended!
```

By contrast to residuation, if e cannot be evaluated because the value of v is not known, narrowing guesses a value for v. The guessed value is uninformed except that only values that make it possible to continue the computation are chosen.

The operation "`=:=`", called *constrained equality*, is predefined in Curry. This operation

is similar to the Boolean equality discussed earlier except for two important differences. The first difference is the type returned by the operation. The constrained equality returns the type *Success*. This type has no visible constructors. An expression of type *Success* either succeeds or fail. There exists predefined operations, "`success`" and "`failed`", to encode successes and failures in a program. The second difference is that operation "`=:=`" narrows instead of residuating. Thus:

```
prelude> Z=:=2+2
Free variables in goal: Z
Result: success
Bindings:
Z=4 ?
```

3.13.3 Flex vs. Rigid

Operations that residuate are called *rigid*, whereas operations that narrow are called *flexible*. By default, all operations are flexible except I/O actions, i.e., those that return the type "`IO ...`" (see Section 3.14). Apart from I/O actions, most primitive operations, like arithmetic operations, are rigid since guessing is not a reasonable option for them. The programmer can override the default behavior with an explicit *eval* declaration. For example, the `prelude` defines a list concatenation operation as follows:

```
infixr 5 ++
...
(++)          :: [a] -> [a] -> [a]
[]     ++ ys  = ys
(x:xs) ++ ys  = x : xs ++ ys
```

Since "`++`" is flexible by default, we can use it to search for a list satisfying a particular property:

```
prelude> X ++ [3,4] =:= [1,2,3,4]
Free variables in goal: X
Result: success
Bindings:
X=[1,2] ?
```

The following operation is defined by the same rules as "`++`", but it is also declared *rigid*:

```
infixr 5 +++
(+++) eval rigid
(+++)          :: [a] -> [a] -> [a]
[]     +++ ys  = ys
(x:xs) +++ ys  = x : xs +++ ys
```

Obviously, operation "`+++`" concatenates lists as well, since it is defined by the same rules, but it behaves differently from "`++`" when its first argument is a variable. For instance, the query similar to the previous flounders:

```
rigidappend> X +++ [3,4] =:= [1,2,3,4]
```

```
Free variables in goal: X
*** Goal suspended!
```

For *ground expressions*, i.e., expressions without logic variables, the flex/rigid status of a function make no difference. Rigid user-defined functions are useful in the context of concurrent/distributed object-oriented programming.

3.13.4 Programming

Often, programming with variables leads to conceptually simple, terse and elegant code at the cost of an acceptable loss of efficiency. The logic variables of a program and/or a query are not much different from the variables that are typically used to solve algebra or geometry problems. In both cases, some unknown entities of the problem are related to each other by expressions involving functions. Narrowing allows us to evaluate these expressions—and in the process to find values for the variables. The simplest application, and the most familiar for those used to solve algebra and geometry problems with variables, is when the expression to evaluate is an equation.

In later chapters, we will discuss some problems that are conveniently solved if one uses variables in computations. Here we want to present a simple, but non-trivial, motivating example. The problem is to parse a string that represents an expression. To keep the example small, our expressions are functional terms whose syntax is defined by:

```
term       ::= identifier
             | identifier '(' args ')'
args       ::= term
             | term ',' args
identifier ::= any non-null string of alphabetic characters
```

For example, `"f(g(a,b))"` is a term described by the above syntax. When a term is represented as a string, answering questions such as how many arguments "`g`" has, is more complicated and less efficient than it needs to be. A parser converts a term from its string representation into a data structure that makes easy and efficient answering questions of that kind. Thus, the first step to build a parser is to design a suitable type to represent a term. Our choice is: [Program]:

```
data Term = Term String [Term]
```

Note that the occurrence of "`Term`" to right of the "`=`" character is a data constructor, whereas the two other occurrences are type constructors. The "`Term`" identifier is *overloaded* by the declaration.

Using this data structure, we represent a function identifier with a string and the function's arguments with a list of terms. For example, the term `"f(g(a,b))"` would be represented as "`Term "f" [Term "g" [Term "a" [],Term "b" []]]`". The following operation parses a term [Program]:

```
parseTerm s | s =:= fun ++ "(" ++ args ++ ")" &
              all isAlpha fun =:= True = Term fun (parseArgs args)
            | all isAlpha s =:= True = Term s []
```

```
  where fun, args free
```

The elements of the program most relevant to our discussion are the variables "`fun`" and "`args`". The first condition of the operation "`parseTerm`" instantiates these variables and ensures that "`fun`" is an alphabetic identifier. The operation "`isAlpha`", defined in the library "`Char`", ensures that its argument, a character, is alphabetic. The operation "`all`" is defined in the "`prelude`". The combination "`all isAlpha`" ensures that all the characters of a string are alphabetic.

If a term has arguments, these arguments are parsed by the operation "`parseArgs`". The overall design of this operation is very similar to that of "`parseTerm`". In this case, though, a string is decomposed according to different criteria [Program]:

```
parseArgs s | s =:= term ++ "," ++ terms &
              parseTerm term =:= result = result : parseArgs terms
            | parseTerm s =:= result = [result]
  where term, terms, result free
```

One could code more efficient versions of this parser. This version is very simple to understand and it is the starting point for the design of more efficient versions that will be discussed later in this book.

3.14 Input/Output

As we have seen up to now, a Curry program is a set of datatype and function declarations. Functions associate result values to given input arguments. However, application programs must also interact with the "outside" world, i.e., they must read user input, files etc. Traditional programming languages addresses this problem by procedures with side effects, e.g., a procedure `read` that returns a user input when it is evaluated. Such procedures are problematic in the context of Curry. Firstly, the evaluation time of a function is difficult to control due to the lazy evaluation strategy (see Section 3.11). Secondly, the meaning of functions with side effects is unclear. For instance, if the function `readFirstNum` returns the first number in a particular file, the evaluation of the expression "`2*readFirstNum`" yields different values at different points of time (if the contents of the file changes).

Curry solves this problem with the "monadic I/O" concept much like that seen in the functional language Haskell [13]. In the monadic approach to I/O, a program interacting with the outside world is considered as a sequence of actions that change the state of the outside world. Thus, an interactive program computes actions which are applied to a given state of the world (this application is finally done by the operating system that executes a Curry program). As a consequence, the outside world is not directly accessible but can be only manipulated through actions that change the world. Conceptually, the world is encapsulated in an abstract datatype which provides actions to change the world. The type of such actions is "`IO t`" which is an abbreviation for

```
World -> (t,World)
```

where "`World`" denotes the type of all states of the outside world. If an action of type "`IO t`"

is applied to a particular world, it yields a value of type `t` and a new (changed) world.

For instance, `getChar` of type "`IO Char`" is an action which reads a character from the standard input whenever it is executed, i.e., applied to a world. Similarly, `putChar` of type "`Char -> IO ()`" is an action which takes a character and returns an action which, when applied to a world, puts this character to the standard output (and returns nothing, i.e., the unit type). The important point is that values of type `World` are not accessible to the programmer — she/he can only create and compose actions on the world.

Actions can only be sequentially composed, i.e., one can built a new action that consists of the sequential evaluation of two other actions. The predefined function

```
(>>) :: IO a -> IO b -> IO b
```

takes two actions as input and yields an action as the result. The resulting action consists of performing the first action followed by the second action, where the produced value of the first action is ignored. For instance, the value of the expression "`putChar 'a' >> putChar 'b'`" is an action which prints "`ab`" whenever it is executed. Using this composition operator, we can define a function `putStrLn` (which is actually predefined in the prelude) that takes a string and produces an action to print this string:

```
putStrLn []     = putChar '\n'
putStrLn (c:cs) = putChar c >> putStrLn cs
```

If two actions should be composed and the value of the first action should be taken into account before performing the second action, the actions can be also composed by the predefined function

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

where the second argument is a function taking the value produced by the first action as input and performs another action. For instance, the action

```
getChar >>= putChar
```

is of type "`IO ()`" and copies, when executed, a character from standard input to standard output. Actually, this composition operator is the only elementary one since the operator "`>>`" can be defined in terms of "`>>=`":

```
a1 >> a2  =  a1 >>= \_ -> a2
```

There is also a primitive "empty" action

```
return :: a -> IO a
```

that only returns the argument without changing the world. The prelude also defines the "empty" action which returns nothing (i.e., the unit type):

```
done :: IO ()
done = return ()
```

Using these primitives, we can define more complex interactive programs. For instance, an

I/O action thats copies all characters from the standard input to the standard output up to the first period can be defined as follows [Program]:

```
echo = getChar >>= \c -> if c=='.' then done else putChar c >> echo
```

Obviously, such a definition is not well readable. Therefore, Curry provides a special syntax extension for writing sequences of I/O actions, called the *do notation*. The do notation follows the layout style (see Section 3.12.3), i.e., a sequence of actions is vertically aligned so that

```
do putChar 'a'
   putChar 'b'
```

is the same as "`putChar 'a' >> putChar 'b'`", and

```
do c <- getChar
   putChar c
```

is just another notation for "`getChar >>= \c -> putChar c`". Thus, the do notation allows a more traditional style of writing interactive programs. For instance, the function `echo` defined above can be written in the do notation as follows:

```
echo = do c <- getChar
          if c=='.'
            then done
            else do putChar c
                    echo
```

As a further example, we show the definition of the I/O action `getLine` as defined in the prelude. `getLine` as an action that reads a line from the standard input and returns it:

```
getLine :: IO String
getLine = do c <- getChar
             if c=='\n'
                then return []
                else do cs <- getLine
                        return (c:cs)
```

Curry also provides predefined I/O actions for reading files and accessing other parts of the environment. For instance, "`readFile f`" is an action which returns the contents of file `f` and "`writeFile f s`" is an action writing string `s` into the file `f`. This allows us to define a function that copies a file with transforming all letters into uppercase ones in a very concise way (`toUpper` is defined in the standard character library `Char` and converts lowercase into uppercase letters) [Program]:

```
convertFile input output =
   do s <- readFile input
      writeFile output (map toUpper s)
```

The function `toUpper`, defined in the library "`Char`", takes a character. If the character is lower case and alphabetic, then it returns it in upper case, otherwise it returns it unchanged. The operation "`map`" is defined in the "`prelude`" and discussed in detail in Section 4.2.6.

The combination "`map toUpper`" transforms all the characters of a string to upper case.

The monadic approach to input/output has the advantage that there are no "hidden" side effects—any interaction with the outside world can be recognized by the `IO` type of the function. Thus, functions can be evaluated in any order and the only way to combine I/O actions is a sequential one, as one would expect also in other programming languages. However, there is one subtle point. If a function computes non-deterministically different I/O actions, like in the expression "`putStrLn (show coin)`" (see Section 3.12.1 for the definition of the non-deterministic function `coin`; `show` is a predefined function that converts any value into a string), then it is not clear which of the alternative actions should be applied to the world. Therefore, Curry requires that *non-determinism in I/O actions must not occur*. For instance, we get a runtime error if we evaluate the above expression:

```
localvar> putStr (show coin)
ERROR: non-determinism in I/O actions occurred!
```

One way to ensure the absence of such errors is the encapsulation of all search between I/O operations, e.g., by using the function `findall`.

Exercise 4 Define an I/O action `filelength` that reads requests the name of a file from the user and prints the length of the file, i.e., the number of characters contained in this file. [Answer]

Part II

Programming with Curry

Chapter 4

Programming in Curry

4.1 Overview

Lists and *trees* are datatypes frequently used in programming.

- A *list* abstracts a sequence of elements. The elements of a list are implicitly ordered by the list structure. Therefore, a list is a convenient representation for queues, stacks and other linear structures. As list can also be used for representing collections, typically unordered, such as a set, by ignoring or hiding the implicit order of the elements.
- A *tree* ...

This section describes in some detail both these datatypes and how they help solve some typical problems, e.g., sorting a collection of elements or searching for an element in a collection.

4.2 Lists

4.2.1 Notation

A *List* is a simple algebraic polymorphic datatype defined by two constructors conventionally referred to as *Nil* and *Cons*. Within the Curry language, the datatype "*List of a*" would be declared as:

```
data List a = Nil | Cons a (List a)
```

Because lists are one of the most frequently used types in functional, logic and functional logic programs, many languages offer several special notations for lists. In Curry, the type "*List of a*", where *a* is a type variable that stands for any type, is predefined and denoted by `[a]`. Likewise, `[]` denotes the constructor *Nil*, the empty list, and "`:`" denotes the constructor *Cons*, which takes an element of type `a` and a list of `a`'s. Thus, with a syntax that is *not* legal in Curry, but is quite expressive, the above declaration would look like:

```
data [a] = [] | a : [a]
```

The expression `(u:v)` denotes the list with the first element `u` followed by the list `v`. The infix operator "`:`", which read "cons", is predefined, right associative and has precedence 5. This implies that `u:v:w` is parsed as `u:(v:w)`.

A list can also be denoted by enumerating its elements, e.g., "`[u,v,w]`" is a list containing three elements, "`u`", "`v`" and "`w`", i.e., it is just another notation for "`u:v:w:[]`". This notation can be used with any number of elements. The elements are enclosed in brackets and separated by commas. This notation has several advantages over the standard algebraic notation: lists stand out in a program and references to lists are textually shorter. In particular, the number of parentheses occurring in the text is reduced. This claim can be easily verified by comparing the builtin notation with the ordinary notation.

The type list is polymorphic, which means that different lists can have elements of different types. However, all the elements of a particular list must have the same type. The following annotated examples show this point [Program]:

```
-- list of integers
digits = [0,1,2,3,4,5,6,7,8,9]

-- list of characters, equivalent to "Pakcs", print with putStr
string = ['P','a','k','c','s']

-- list of list of integers
matrix = [[1,0,2],[3,7,2],[2,8,1],[3,3,4]]
```

Other special notations available for lists are described in Sections 4.2.3 and 4.2.4.

4.2.2 Inductive Definitions

Many elementary functions on lists are defined by an induction similar to that available for the naturals. The cases of the induction are conveniently defined by different rules using pattern matching. For lists, the base case involves defining a function for `[]` whereas the inductive case involves defining the function for a list `(u:v)` under the assumption that the value of the function for `v` is available. In a program, this is expressed by a recursive call. The function that counts the number of elements of a list is emblematic in this respect:

```
len []    = 0
len (u:v) = 1 + len v
```

For computing the length of a list, the value of `u` is irrelevant and `u` should be replaced by an anonymous variable in the above definition.

Exercise 5 Code an inductively defined function that takes a list of integers and returns the sum of all the integers in the list. Hint: the function should return `0` for an empty list. [Answer]

The `prelude` defines many useful functions on lists, e.g., "`++`" for concatenation, "`!!`" for indexing, i.e., `(l!!i)` is the i-th (starting from 0) element of `l`, etc. We will use some of these functions, after providing a brief explanation, in this section. We might also re-define some functions already available in the `prelude` or other libraries when they make good examples.

E.g., the function `len` discussed above is equivalent to the function `length` of the `prelude`. In Section 4.2.5, we will present the most important list functions available in the `prelude`.

Functions inductively defined are easy to code, understand and evaluate. Sometimes they may be inefficient. Below are two definitions of a function to reverse a list. For long lists, the second one is much more efficient.

```
slowRev [] = []
slowRev (u:v) = slowRev v ++ [u]

fastRev l = aux l []
  where aux [] r = r
        aux (u:v) r = aux v (u:r)
```

A function inductively defined performs a "traversal" of its argument. During this traversal some computation is performed on each element of the list—this is referred to *visiting* a *cons*—and the result combined with a recursive invocation of the function. Loosely speaking, the visit can be performed either *before* the recursive call, or *after*, or *both*. The following example shows how to subtract the minimum element of a list of integers from all the elements of the list. The function performs a single traversal of its argument. The minimum of the list is computed (as much as feasible) before the recursive call. The subtraction is computed after the recursive calls (otherwise the minimum could not be known) [Program]:

```
submin [] = []
submin (x:xs) = fst (aux (x:xs) x)
  where aux []     m = ([],m)
        aux (y:ys) m = let (zs,n) = aux ys (min y m)
                       in (y-n:zs,n)
```

The function `fst`, which returns the first element of a pair, is defined in the `prelude`. The function `min`, which returns the minimum of two integers, is defined in the `Integer` library.

More complicated computations may lead to more complicated inductive definitions. A discussion on the structure and the design of inductively defined function is in [1].

Exercise 6 Code an inductively defined function that transposes a matrix represented by a list of lists (all of the same length). [Answer]

There are a couple of noteworthy alternatives to directly defining inductive functions. One involves higher-order list functions. Some of these functions are presented in Section 4.2.6. The other involves narrowing. Lists are a particularly fertile ground for narrowing. Below are two definitions of the function that computes the last element of a list. The first definition is inductive, whereas the second is narrowing-based.

```
inductLast [x] = x
inductLast (x:y:z) = inductLast (y:z)

narrowLast x | x =:= y++[e] = e where y,e free
```

4.2.3 Ranges

A special notation is available to define lists containing *ranges* of integers. The most common of this notation is "`[`e_1`..`e_2`]`" which denotes the list "`[`$e_1, e_1+1, e_1+2, \cdots, e_2$`]`". For example:

```
prelude> [2..5]
Result: [2,3,4,5] ?
prelude>
```

Similarly, the expression "`[`e`..]`" denotes the *infinite* list of all the integers starting from e. This list cannot be printed in its entirety, but it can be used in a program if only a finite portion of the list is needed, because the evaluation strategy is lazy.

The elements in the lists defined by the above expressions are consecutive, i.e., the distance between adjacent elements is one. The above expressions can be generalized to produce lists where the distance between adjacent elements is a constant greater than one. This distance is inferred from the first two elements of the expression. For example:

```
prelude> [2,6..20]
Result: [2,6,10,14,18] ?
prelude>
```

Likewise, "`[2,6..]`" generates the infinite list "`[2,6,10,14,...]`".

Ranges can be defined using ordinary functions. The `prelude` defines four functions whose names start with `enumFrom`. These functions define in the ordinary syntax the notations for ranges.

4.2.4 Comprehensions

Another useful notation involving lists goes under the name of *list comprehension*. A list comprehension is a notation to construct a list from one or more other lists called *generators*. It goes without saying that ranges are simple generators. For example, the infinite sequence of square and triangular numbers are obtained as follows [Program]:

```
squares   = [x * x | x <- [0..]]
triangles = [x * (x+1) `div` 2 | x <- [0..]]
```

A *generator* is an expression of the form *var* `<-` *list*. Generators can be nested and/or combined with *guards*. A *guard* is a Boolean expression that filters the elements produced by the generator. For example, if `isPrime` is a predicate telling whether an integer greater than 2 is a prime number, the following comprehension is the sequence of the prime numbers [Program]:

```
primes = [x | x <- [2..], isPrime x]
```

In this example, the guard is the Boolean expression `(isPrime x)`. The elements produced by the generator are passed to the comprehension if and only if the guard holds.

Generators are considered to be nested from left to right. The following example shows how to compute pairs where the second component is not greater than the first [Program]:

```
lexPairs = [(x,y) | x <- [0..3], y <- [x..3]]
```

This simple example shows that the second generator `(y<-[x..3])` is nested within the first one, since it references the generated elements.

Exercise 7 Compute the Fibonacci sequence using a list comprehension. Hint: compute a list of *pairs* of numbers where each pair contains two *consecutive* Fibonacci numbers. [Answer]

4.2.5 Basic Functions

The PAKCS compiler/interpreter of Curry is distributed with the `prelude`, a collection of primitive and fundamental types and functions, and with several libraries. The `prelude` and some of these libraries contain useful list functions. In this section, we informally discuss some of these functions. The `currydoc` documentation utility, which is distributed with PAKCS, should be used for an exhaustive up-to-date consultation of the content of these libraries.

Name	**Description**	**Example(s)**
`head`	First element of a list	`head [1,2]` = `1`; `head []` fails
`tail`	All the elements but the first	`tail [1,2]` = `[2]`; `tail []` fails
`length`	Length	`length [1,2]` = `2`
`null`	Tell whether it is nil	`null [1,2]` = `False`
`++`	Concatenate two lists	`[1,2]++[3]` = `[1,2,3]`
`!!`	n-th element of a list	`[1,2]!!1` = `[2]`; `[1,2]!!4` fails
`reverse`	Reverse the order of the elements	`reverse [1,2]` = `[2,1]`
`concat`	Concatenate all the lists of a list	`concat [[1,2],[3]]` = `[1,2,3]`
`take`	List of the first n elements	`take 2 [1,2,3]` = `[1,2]`
`drop`	All elements but the first n	`drop 2 [1,2,3]` = `[3]`
`and`	Boolean conjunction	`and [True,False,True]` = `False`
`or`	Boolean disjunction	`or [True,False,True]` = `True`
`elem`	Whether a value is in a list	`elem 2 [1,3,5]` = `False`
`nub`	Remove duplicates	`nub [1,2,2]` = `[1,2]`
`delete`	Remove the first occurrence of a value	`delete 2 [2,1,2]` = `[1,2]`; `delete 2 [1]` = `[1]`

Many more functions that operate on lists are defined in the libraries of the PAKCS distribution (e.g., see the library `List` which contains the definition of `nub` and `delete` discussed above). The above table is intended to give only a feeling of what is available.

4.2.6 Higher-order Functions

Lists are commonly used to represent collections of elements. Some computations of a list can be expressed by repeatedly applying another, somewhat simpler, computation to all the elements of the collection. This section discusses some frequently occurring situations of this kind.

The simplest case is when a list, which we refer to as the *result list*, is obtained from another list, which we refer to as the *argument list*, by applying the same function, say `f`, to

all the elements of the argument list. This is easily accomplished by defining a new function, say `flist` since its analogy to `f`, as follows:

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

Although trivial, the definition of `flist` can be avoided altogether using the function `map`, provided by the prelude. The function `map` is higher-order in that it takes as an argument the function, in this example `f`, that is applied to all the arguments of the list. Thus, the function `flist` defined above is the same as `map f`.

The following code, taken from the `prelude`, shows the type and the definition of `map`:

```
map          :: (a->b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

It can be seen that the first argument of `map` is a function from any type a to any type b. The second argument of `map` is a list whose elements must have, of course, type a. The result is a list of type b. For example, suppose that `isEven` is a function telling whether an integer is even. Then, the expression `(map isEven [0,1,2,3])` evaluates to `[True,False,True,False]`.

A second frequently used higher-order function on lists is `filter`. As the name suggests, `filter` is used to filter the elements of a list that satisfy some criterion expressed by a predicate.

The following code, taken from the `prelude`, shows the type and the definition of `filter`:

```
filter          :: (a -> Bool) -> [a] -> [a]
filter _ []     = []
filter p (x:xs) = if p x then x : filter p xs else filter p xs
```

It can be seen that the first argument of `filter` is a function from any type a to `Bool`, i.e., a predicate. The second argument of `map` is a list whose elements must have, of course, type a. The result is again a list of type a. The elements of the result are the elements of the second argument that satisfy the predicate. For example, as before, suppose that `isEven` is a function telling whether an integer is even. Then, the expression `(filter isEven [0,1,2,3])` evaluates to `[0,2]`.

The last higher-order function operating on lists that we describe in this section is used to "combine together" all the elements of a list. For example, a function that adds all the elements of a list of integers can be defined using a higher-order function and the ordinary addition on integers. Several options should be considered, e.g., whether the elements of a list are composed starting with the first or the last one, whether the list can be empty and thus a default value must be supplied, etc. The prelude contains a family of functions, referred to as *folds* for this purpose. The names of these functions starts with "`fold`".

The following code, taken from the `prelude`, shows the type and the definition of `foldr`:

```
foldr            :: (a->b->b) -> b -> [a] -> b
```

```
foldr _ z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

For example, functions that compute the sum, product and maximum of all the elements of a list of integers are easily defined through `foldr` as follows [Program]:

```
sumList  = foldr (+) 0
prodList = foldr (*) 1
maxList  = \l -> foldr max (head l) (tail l)
```

The last function is more complicated than the previous two, because it is meaningful only for non-empty lists. The function `foldr1`, defined in the `prelude`, would simplify our definition of `maxList`.

4.2.7 `findall`

There is a predefined function, `findall`, that is similar to a list comprehension and generates a list of values from an expression that generates a value. By contrast to comprehensions, though, the generator of `findall` is a constraint, i.e., a function returning `Success`, and the order in which the elements are generated is less deterministic. The function `findall` is often used to find all the solutions of a search problem.

For example, consider the problem of computing all the subsets of a set. Let us represent a set with a list. This representation requires some care both to avoid duplicate elements in a list and to ensure that the order of the elements in a list cannot be observed. We ignore these conditions since they are irrelevant to our example. The following non-deterministic function returns a subset of a set [Program]:

```
subset []     = []
subset (x:xs) = x:subset xs
subset (_:xs) =   subset xs
```

Now, using `findall` we can easily compute the set of all subsets of a set [Program]:

```
allSubsets set = findall \x -> subset set =:= x
```

The intuitive reading of the above fragment is "Find all x's such that x is a subset of *set*."

In the above example, the constraint may generate more than one value because the function `subset` is non-deterministic. A second situation in which a constraint may generate more than one value is when its evaluation involves narrowing steps. For example, the prerequisites for the undergraduate Computer Science courses at Portland State is abstracted by 16 rules as follows [Program]:

```
isPrereqOf 162 = 161
isPrereqOf 163 = 162
isPrereqOf 200 = 162
...
isPrereqOf 303 = 252
isPrereqOf 303 = 300
isPrereqOf 350 = 252
```

The meaning is that, e.g., `162` is a direct prerequisite of both `163` and `200` and that, e.g., both `252` and `300` are direct prerequisites of `303`.

The function to compute *all* the direct prerequites of a course and the function to compute *all* courses that a course gives access to (somewhat the inverse of the former) are shown below [Program]:

```
allIsPrereqOf     course = findall \p -> isPrereqOf course =:= p
allGivesAccessTo  course = findall \c -> isPrereqOf c =:= course
```

The evaluation of `findall` does not instantiate the free variables, if any, in the constraint argument unless they are local to the constraint itself, i.e., they are declared by a `let` block. The reason is that this seems to be the most sensible semantics.

4.2.8 Narrowing

Narrowing is a convenient programming feature when dealing with lists. Lists are frequently used to represent collections of elements. Sometimes the problem is to find in a list either elements or sublists that satisfy certain relationships. The programmer can either code functions to compute these elements or express the relationships using variables for these elements and let narrowing compute the elements by instantiating the variables. Generally, the latter leads to simpler and more declarative programs.

For example, consider a program that plays the game of poker. A hand is represented by a list of 5 cards. Suppose that the problem is to find whether 4 of the 5 cards are all of the same kind, i.e., the hand is a four-of-a-kind. A narrowing-based solution removes one card from the hand so that the remaining 4 cards are all of the same rank. The following function takes a hand. If the hand is a four-of-a-kind, the function returns the kind or rank of the four cards, otherwise it fails [Program]:

```
fourConstraint hand | hand =:= x++y:z & map rank (x++z) =:= [r,r,r,r]
                    = r
                    where x,y,z,r free
```

The card removed from the hand is represented by `y`. This card is non-deterministically selected by solving the constraint "`hand =:= x++y:z`". The remaining cards are represented by `x` and `z`. They are uniquely determined by the selection of `y`, and vice versa. Additionally, the condition of the rule imposes that all the cards in `x` and `z` have the same rank, represented by `r`. The rank, too, is non-deterministically selected by solving the constraint "`map rank (x++z) =:= [r,r,r,r]`". If the condition succeeds, there is obviously a unique value for all these variables.

The advantage of the narrowing-based approach over more conventional approaches is that no instructions need to be coded both to isolate the card that does not contribute to the four-of-a-kind nor to find the kind of the four.

As we said, when a hand is not a four-of-a-kind the above function fails. In general, failures are undesirable for normal conditions such as not having a four-of-a-kind hand. The function `findall` described in Section 4.2.7 is used to construct a list of all the results produced by `fourConstraint` when applied to a hand. Obviously, this list can contain either zero or one

value only. The following function prints whether a hand is a four-of-a-kind without ever failing [Program]:

```
isFour hand = putStrLn (if sorry then "Sorry" else "Four "++(show rank))
              where score = findall (\r -> fourConstraint hand =:= r)
                    sorry = score == []
                    rank  = head score
```

The above example is typical of situations in which a collection contains elements that must satisfy a certain conditions. Since lists are implicitly ordered, conditions involving the *position* of elements in a collection can also be conveniently expressed using narrowing. We will see an example of this kind in a program to solve the n-queens puzzle.

Exercise 8 Similar to the example just discussed, code a function that tells whether a hand in a game of poker is a *full house*. Hint: `Cards.curry` defines suits, ranks, etc. [Answer].

4.3 Trees

...

Part III

Applications & Libraries

Chapter 5

Web Programming

5.1 Overview

Due to the ubiquity of the world wide web (WWW or "web" for short), many applications offer web-based interfaces in order to support convenient access to them. This chapter describes how one can implement web-based interfaces in Curry. We will see that the functional and logic programming features of Curry are quite useful in providing a high-level programming interface for such applications so that Curry can also be used as a language for "web scripting," i.e., for writing web interfaces in a concise manner.

This chapter requires some basic knowledge about the structure of HTML, the "Hypertext Markup Language" for describing the general form and layout of documents presented by web browsers. Up-to-date information about HTML is available from the World Wide Web Consortium (W3C).

The approach to web programming described in this chapter is based on the library "`HTML`" contained in the PAKCS distribution. Details about the ideas and the implementation of this library can also be found in [8].

5.2 Representing HTML Documents in Curry

HTML is a language for specifying the structure and layout of web documents. We also say "HTML document" for a text written in the syntax of HTML. Basically, an HTML document consists of the following elements:

- elementary text
- *tags* with other HTML elements as contents, like headers (`h1`, `h2`,...), lists (`ul`, `ol`,...), etc.
- tags without contents, like line breaks (`br`), images (`img`), etc.

The plain syntax of HTML, which is interpreted by a web browser when displaying HTML documents, requires tags be enclosed in pointed brackets (`<`$\cdots$`>`). The contents of a tag is written between an opening and a closing tag where the closing tag has the same name as the opening tag but is preceded by a slash. Tags can also contain *attributes* to attach specific

information to tags. If present, attributes are written in the form "*name*=*value*" after the opening tag's name and before its right bracket.

For instance, "`i`" and "`b`" are tags to specify that their contents should be set using an italic and bold font, respectively. Thus, the HTML text

```
This is the <i>italic</i> and the <b>bold</b> font.
```

would be displayed by a web browser as this:

This is the *italic* and the **bold** font.

Tags without contents have no closing tag. An example is the tag for including images in web documents, where the attribute "`src`" specifies the file containing the picture and "`alt`" specifies a text to be displayed as an alternative to the picture:

```
<img src="picture.jpg" alt="Picture">
```

A program with a web interface must generate HTML documents that are displayed in the client's browser. In principle, we can do this in Curry by printing the text of the HTML document directly, as in:

```
writeHTML = do
  putStrLn "This is the "
  putStrLn "<i>italic</i> and the "
  putStrLn "<b>bold</b> font."
```

If the program becomes more complex and generates the HTML text by various functions, there is the risk that the generated HTML text is syntactically not correct. For instance, the tags with contents must be properly nested, i.e., the following text is not valid in HTML (although browser can display it but may become confused by illegal HTML documents):

```
This is <b>bold and also <i>italic</b></i>.
```

To avoid such problems in applications programs, one can introduce an *abstraction layer* where HTML documents are modeled as terms of a specific datatype. Thus, a web application program generates such abstract HTML documents instead of the concrete HTML text. This has the advantage that ill-formed web documents correspond to ill-formed expressions in Curry which would immediately be rejected by the compiler. The actual printing of the concrete HTML text is done by a wrapper function that translates an abstract HTML document into a string.

For representing abstract HTML documents in Curry, we define the following datatype of *HTML expressions*:

```
data HtmlExp = HtmlText   String
             | HtmlStruct String [(String,String)] [HtmlExp]
             | HtmlElem   String [(String,String)]
```

The constructor `HtmlText` corresponds to elementary text in an HTML document, whereas the constructors `HtmlStruct` and `HtmlElem` correspond to HTML elements with and without contents, respectively. The parameter of type "`[(String,String)]`" is the list of attributes,

i.e., name/value pairs.

For instance, our first HTML document above is represented with this datatype as the following list of HTML expressions:

```
[HtmlText "This is the ",
 HtmlStruct "i" [] [HtmlText "italic"],
 HtmlText " and the ",
 HtmlStruct "b" [] [HtmlText "bold"],
 HtmlText " font."]
```

Similarly, the image tag above is represented as follows:

```
HtmlElem "img" [("src","picture.jpg"),("alt","Picture")]
```

Obviously, we can specify any HTML document in this form but this becomes very tedious for a programmer. To avoid this, we define several functions as useful abbreviations of common HTML tags:

```
h1     hexps  = HtmlStruct "h1" [] hexps                  -- header 1
h2     hexps  = HtmlStruct "h2" [] hexps                  -- header 2
⋮
bold   hexps  = HtmlStruct "b"  [] hexps                  -- bold font
italic hexps  = HtmlStruct "i"  [] hexps                  -- italic font
hrule         = HtmlElem "hr" []                          -- horizontal rule
breakline     = HtmlElem "br" []                          -- line break
image src alt = HtmlElem "img" [("src",src),("alt",alt)]  -- image
...
```

Characters that have a special meaning in HTML, like "<", ">", "&", """, should be quoted in elementary HTML texts to avoid ill-formed HTML documents. Thus, we define a function "`htxt`" for writing strings as elementary HTML texts where the special characters are quoted by the function "`htmlQuote`":

```
htxt   :: String -> HtmlExp
htxt s = HtmlText (htmlQuote s)

htmlQuote :: String -> String
htmlQuote [] = []
htmlQuote (c:cs) | c=='<' = "&lt;"   ++ htmlQuote cs
                 | c=='>' = "&gt;"   ++ htmlQuote cs
                 | c=='&' = "&amp;"  ++ htmlQuote cs
                 | c=='"' = "&quot;" ++ htmlQuote cs
                 | otherwise = c : htmlQuote cs
```

Now we can represent our first HTML document above as follows:

```
[htxt "This is the ", italic [htxt "italic"],
 htxt " and the ", bold [htxt "bold"], htxt " font."]
```

All the definitions we have introduced so far are contained in the library "HTML" of the PAKCS distribution. Thus, to define abstract HTML documents in a program, one has to write the import declaration

```
import HTML
```

in the header of the Curry program. The HTML library defines also a wrapper function showHtmlExps to generate the concrete textual representation of an abstract HTML document. For instance, the value of

```
showHtmlExps [h1 [htxt "Hello World"], italic [htxt "Hello"], htxt " world!"]
```

is the string

```
<h1>Hello World</h1>
<i>Hello</i> world!
```

Furthermore, the HTML library defines another wrapper function showHtmlDoc to generate a complete HTML document with head and body parts where the first argument is the title of the document. For instance, the value of

```
showHtmlDoc "Hello" [h1 [htxt "Hello World"], italic [htxt "Hello"], htxt " world!"]
```

is the string

```
<html>
<head>
<title>Hello</title>
</head>
<body bgcolor="#ffffff">
<h1>Hello World</h1>
<i>Hello</i> world!
</body>
</html>
```

We can use these functions to write Curry programs that generate HTML documents. For instance, consider the generation of an HTML document that contains a list of all multiplications of digits, i.e., a line in this document should look as follows:

The product of **7** and **6** is **42**

First, we define a list of all triples containing such multiplications by the use of list comprehensions (compare Section 4.2.4):

```
multiplications = [ (x,y,x*y) | x <- [1..10], y <- [1..x] ]
```

Each triple is translated into a list of HTML expressions specifying the layout of a line:

```
mult2html :: (Int,Int,Int) -> [HtmlExp]
mult2html (x,y,z) =
 [htxt "The product of ", bold [htxt (show x)],
  htxt " and ", bold [htxt (show y)],
```

```
  htxt " is ", bold [htxt (show z)], breakline]
```

Now can use these definitions to define the complete HTML document (the prelude function `concatMap` applies a function that maps elements to lists to each element of a list and concatenates the result into a single list) [Program]:

```
html_multiplications =
 [h1 [htxt "Multiplication of Digits"]] ++ concatMap mult2html multiplications
```

For instance, we can use the latter function to store the HTML document in a file named "`multtable.html`" by evaluating the expression:

```
writeFile "multtable.html" (showHtmlDoc "Multiplication" html_multiplications)
```

Exercise 9 Define a function `boldItalic` to translate text files into HTML documents. The function has two arguments: the name of the input text file and the name of the file where the HTML document should be stored. The HTML document should have the same line structure as the input but the lines should be formatted in bold and italic, i.e, first line in bold, second in italic, third in bold, fourth in italic, etc. Hint: use the prelude function `lines` to split a string into a list of lines. [Answer]

5.3 Server-Side Web Scripts

We have seen so far how to write programs that create HTML documents. Such programs could be useful to transform existing data into a static set of HTML pages. However, this is not sufficient to create *dynamic web pages*, i.e., web pages whose contents is computed at the time they are requested by a client. The creation of dynamic web pages is supported by most web servers by so-called CGI (Common Gateway Interface) programs. If a web server is asked for a document with the suffix "`.cgi`" instead of "`.html`" (the exact behavior is defined in the configuration of the web server; see also Section 5.4 below), then the server does not return the contents of the corresponding file but executes the file (the "CGI program") and returns the standard output produced by this program. Thus, a CGI program must write an HTML document on its standard output. The CGI program can also take user input in an HTML form into account; this is described in Section 5.5.

To support the creation of dynamic HTML documents, the `HTML` library has a definition for HTML *forms* as follows:

```
data HtmlForm = HtmlForm String [FormParam] [HtmlExp]
```

The first argument is the title of the form (as in HTML documents) and the third argument is the contents of the form which can also contain elements for user input, as we will see later (see Section 5.5). The second argument is a list of optional parameters to extend the functionality of forms, like cookies, style sheets etc (see Section 5.8). Since they are seldom used in standard forms, the `HTML` library contains also the following function to specify forms without optional parameters:

```
form :: String -> [HtmlExp] -> HtmlForm
form title hexps = HtmlForm title [] hexps
```

Usually, the contents of dynamic HTML documents depend on the environment of the web server, e.g., information stored in the file system or databases. Thus, a dynamic web page is allowed to perform some I/O operations in order to compute the requested HTML document. As a consequence, a function to compute a dynamic web page must have the type "`IO HtmlForm`". For instance, if we want to write a CGI program that computes the above multiplications of digits on demand, we define the following function `multForm` (the right-associate operator "`$`" is defined with a low precedence in the prelude and denotes function application; it is often used to avoid brackets, e.g., the expression "`f $ g $ 3+4`" is equivalent to "`f (g (3+4))`") [Program]:

```
multForm :: IO HtmlForm
multForm = return $ form "Multiplication of Digits" html_multiplications
```

To see an application of accessing the server environment, we define a form that shows the current date and time of the server (the IO action "`getClockTime`", defined in the standard library `Time`, returns the current date and time in some internal representation which can be converted into a readable string by the function "`toDateString`") [Program]:

```
timeForm :: IO HtmlForm
timeForm =
 do time <- getClockTime
    return $ form "Current Server Time"
              [h1 [htxt $ "Current date and time: " ++ toDateString time]]
```

The installation of such web programs on a web server is described in the following section.

5.4 Installing Web Programs

Although the installation of CGI programs highly depends on the web server, in this section we will provide some hints so that you can execute the programs described in this chapter on your web server. Clearly, your web server must be configured to enable the execution of CGI programs. Fortunately, most web servers support CGI programs, though they will likely require special configuring by the system administrator. A web server can be configured to interpret any file ending with "`.cgi`" and execute it when requested, or it can be also configured to execute only CGI programs stored in a particular directory, e.g., "`cgi-bin`". Ask your system administrator for the instructions on CGI execution that are specific to your system.

If you have installed PAKCS on your system (i.e., the web server), the installation of a CGI program is quite simple. Assume you have written a Curry program "`myscript.curry`" containing a definition of a form function "`main`" of type "`IO HtmlForm`" (see previous section). Then you can compile it into an executable CGI program by the shell command

```
makecurrycgi myscript
```

(`makecurrycgi` is a shell script stored in the "`bin`" directory of PAKCS). This creates (after successful compilation) an executable program "`myscript.cgi`". If your main form function has a name different from the default `main`, you can provide it with option "`-m`". For instance, the command

```
makecurrycgi -m myForm myscript
```

creates a CGI program with form function "`myForm`". In general, the parameter following "`-m`" can be any Curry expression of type "`IO HtmlForm`".[1]

Similarly, the option "`-o`" can be used to install the CGI program under a different name. For instance, the command

```
makecurrycgi -o ~/cgi-bin/myscript.cgi myscript
```

installs the executable CGI program in the file "`~/cgi-bin/myscript.cgi`". Depending on the configuration of your web server, you can execute the CGI program by requesting the document with a URL like "`http://your.server.name/cgi-bin/myscript.cgi`" in your web browser.

5.5 Forms with User Input

In many applications, dynamic web pages should not only depend on the environment of the web server but also on the input provided by the client (i.e., the user contacting the server via its browser). In principle, this is possible since HTML includes also elements for user input (text fields, buttons, etc) which is sent to the web server when requesting a document. How can we access the user input in to a CGI program running on the server? The whole purpose of the Common Gateway Interface is to create a way to send information to the server and from the browswer, hence the name Common Gateway. Fortunately, it is not necessary to know all the details of CGI since the `HTML` library defines an abstraction layer to provide a comfortable access to user inputs. This abstraction layer exploits the functional and logic features of Curry and will be explained in this section.

The `HTML` library contains definitions of various input elements for HTML forms. For instance, the element "`textfield`" defines an HTML input element where the user can type a line of text:

```
textfield :: CgiRef -> String -> HtmlExp
```

The second argument is the initial contents and the first argument is the reference to this element (*CGI reference*). The reference is used in the CGI program to access the user's actual input when computing an answer to this form. Now, how is the type `CgiRef` defined? The surprising answer is: the type is abstract (i.e., its constructors are not exported by the `HTML` library) since it is not necessary to know any constructor! It is sufficient to use a logic variable when using a text field in an HTML form. For instance, we can define a form containing a string and an input field as follows:

[1] Since this expression is executed as the main program, all symbols of this expression must be exported from the Curry program.

```
rdForm = return $ form "Question"
           [htxt "Enter a string: ", textfield tref ""]
 where tref free
```

A `CgiRef` variable serves as a reference to the corresponding input field to access the user's input. Raw CGI requires concrete strings as references (attribute "`name`" of "`input`" tags) which is error-prone (since typos in these strings lead to run-time errors). However, the concrete strings are not important, and so the logic variables are sufficient. It is only important to use them when computing the answer to the client. For this purpose, the `HTML` library defines a *CGI environment* as a mapping from CGI references to strings:

```
type CgiEnv = CgiRef -> String
```

A CGI environment is used to collect the input of the user when computing the response. The computation of the response is done by an *event handler* that is attached to each button for submitting a form to the web server. Thus, an event handler has the type

```
CgiEnv -> IO HtmlForm
```

i.e., it is called with the current CGI environment and yields an I/O action that returns a form to be sent back to the client. Thus, the `HTML` library defines a button for submitting forms with the following type:

```
button :: String -> (CgiEnv -> IO HtmlForm) -> HtmlExp
```

The first argument is the text shown on the button and the second argument is the event handler called when the user clicks this submit button.

The actual event handlers can simply be defined as local functions attached to forms so that the `CgiRef` variables are in scope and need not be passed. To see a simple but complete example, we show the specification of a form where the user can enter a string and choose between two actions (reverse or duplicate the string) by two submit buttons (see Figure 5.1) [Program]:

```
rdForm :: IO HtmlForm
rdForm = return $ form "Question"
           [htxt "Enter a string: ", textfield tref "", hrule,
            button "Reverse string"   revhandler,
            button "Duplicate string" duphandler]
 where
   tref free

   revhandler env = return $ form "Answer"
     [h1 [htxt $ "Reversed input: " ++ reverse (env tref)]]

   duphandler env = return $ form "Answer"
     [h1 [htxt $ "Duplicated input: " ++ env tref ++ env tref]]
```

Note the simplicity of retrieving values entered into the form: since the event handlers are called with the appropriate environment containing these values (parameter "`env`"), they can

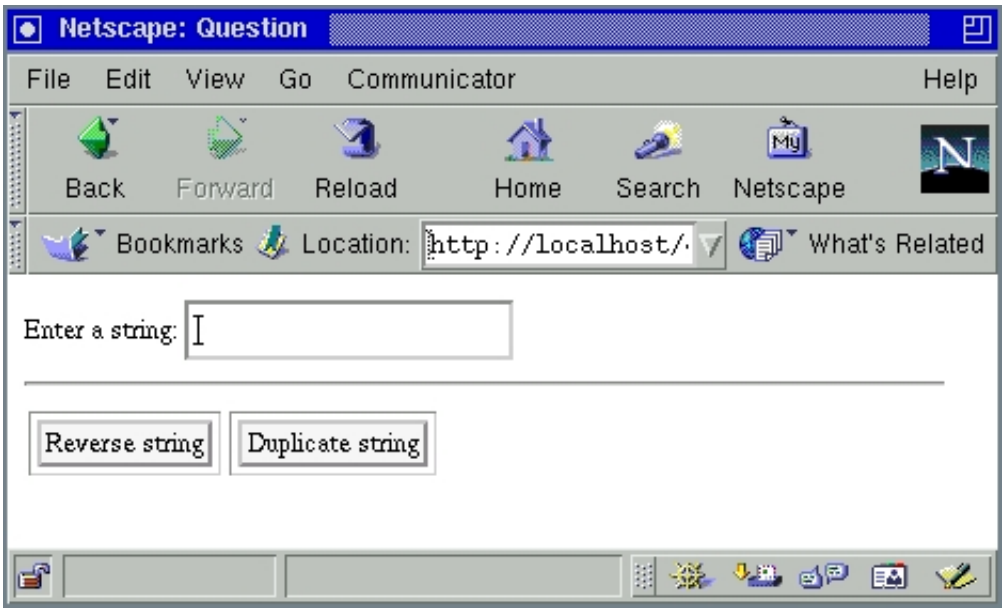

Figure 5.1: A simple string reverse/duplication form

easily access these values by applying the environment to the appropriate CGI reference, like "`(env tref)`".

5.6 Further Examples for Web Server Programming

Now we have seen all elements for writing CGI programs in Curry. In this section we will show by various examples how to use this programming interface. We will see that this programming model (i.e., logic variables for CGI references, associated event handlers depending on CGI environments) is sufficient to solve typical problems in web server programming in an appropriate way, like handling sequences of interactions or holding intermediate states between interactions.

5.6.1 Interaction Sequences

In the previous example, the interaction between the client and the web server is quite simple: the client sends a request by filling a form which is answered by the server with an HTML document containing the requested information. In realistic applications, it is often the case that the interaction is not finished by sending back the requested information but the client requests further (e.g., more detailed) information based on the received results. Thus, one has to deal with sequences of longer interactions between the client and the server.

Our programming model provides a direct support for interaction sequences. Since the answer provided by the event handler is an HTML form rather than an HTML expression, this answer can also contain further input elements and associated event handlers. By nesting event handlers, it is straightforward to implement bounded sequences of interactions. An example for this technique is shown in the next section.

A more interesting question is how to implement other control abstractions like arbitrary loops. For this purpose, we show the implementation of a simple number guessing game: the client has to guess a number known by the server (here: `42`), and for each number entered by the client the server responds whether this number is right, smaller or larger than the number to be guessed. If the guess is not right, the answer form contains an input field where the client can enter the next guess. Moreover, the number of guesses should also be counted and shown at the end.

As the typical approach in declarative languages, we implement looping constructs by recursion. Thus, the event handler computing the answer for the client contains a recursive call to the initial form which implements the interaction loop. The entire implementation of this number guessing game is as follows [Program]:

```
guessForm :: IO HtmlForm
guessForm = return $ form "Number Guessing" (guessInput 1)

guessInput :: Int -> [HtmlExp]
guessInput n =
  [htxt "Guess a natural number: ", textfield nref "",
   button "Check" (guessHandler n nref)]   where nref free

guessHandler :: Int -> CgiRef -> (CgiRef -> String) -> IO HtmlForm
guessHandler n nref env =
  let nr = readInt (env nref) in
  return $ form "Answer"
            (if nr==42
             then [h1 [htxt $ "Right! You needed "++show n++" guesses!"]]
             else [h1 [htxt $ if nr<42 then "Too small!"
                                       else "Too large!"],
                   hrule] ++ guessInput (n+1))
```

"`guessInput n`" is an HTML expression corresponding to the initial form which contains an input field for entering the client's guess. "`guessHandler`" is the associated event handler where the number of guesses and the CGI reference to the input field are the first and the second argument of the handler, respectively. It checks the number entered by the client (`readInt` is defined in the standard library `Read` and converts a string into a number) and returns the different answers depending on the client's guess. If the guess is not right, the `guessInput` is appended to the answer which implements the recursive call.

5.6.2 Handling Intermediate States

A nasty problem in many CGI applications is the handling of intermediate states due to the fact that HTTP is a stateless protocol. For instance, in electronic commerce applications, the clients have shopping baskets where the already selected items are stored, and the contents of these baskets must be kept between the interactions. Storing this information on the server side has several drawbacks. For instance, the client wants to identify himself only after he really orders the items, i.e., during the selection phase the server cannot uniquely associate the selections to a client. Furthermore, the client might not proceed with his selections so that

the server does not know whether the basket information can be deleted (which is necessary at some point to avoid a memory overflow). Therefore, it is often better to store such client-dependent information on the client side. For this purpose, one can have HTML forms with input elements of type "`hidden`" which have no visual representation but can be used to pass client-dependent information between interactions. "Raw" HTML/CGI programmers must explicitly handle these fields which is awkward and a source of many programming problems.

Our programming model offers a much simpler solution to this problem. By nesting event handlers (which is allowed in languages with lexical scoping like Curry), one can directly refer to input elements in previous forms. As a concrete example, we consider a sequence of HTML forms where the client enters his first name in the first form and his last name in the second form. The complete name is returned in the third form. This example can be implemented as follows [Program]:

```
nameForm = return $ form "First Name Form"
  [htxt "Enter your first name: ", textfield firstref "",
   button "Continue" fhandler]
 where
   firstref free

   fhandler _ = return $ form "Last Name Form"
                  [htxt "Enter your last name: ", textfield lastref "",
                   button "Continue" lhandler]
     where
       lastref free

       lhandler env = return $ form "Answer"
                      [htxt $ "Hi, " ++ env firstref ++ " " ++ env lastref]
```

Due to lexical scoping, the variable "`firstref`" is visible in the event handler "`lhandler`" without explicitly passing it as an argument.

5.6.3 Storing Information on the Server

We have seen how we can retrieve information from the server by CGI programs. This is possible by performing I/O actions on the server before computing the HTML form as the response to the client. In many applications, clients also want to store or update information on the server, e.g., by putting orders for books, flight tickets, etc. In this section we will see a small example that demonstrates how this can be done using the already known techniques.

Consider the implementation of a web form that counts and shows the number of visitors. Thus, each visitor updates the current visitor counter on the server. This can be easily implemented by storing the current visitor number in a file. For this purpose, we define an I/O action "`incVisitNumber`" that reads the number stored in this file, increments it, stores the incremented number in the file, and returns the incremented number (`doesFileExist` is an action defined in the library `Directory` that checks the existence of a file):

```
incVisitNumber :: IO Int
incVisitNumber = do
```

```
 existnumfile <- doesFileExist visitFile
 if existnumfile
   then do vfcont <- readFile visitFile
           writeVisitFile (readInt vfcont +1)
   else writeVisitFile 1

writeVisitFile n =
 do writeFile (visitFile++".new") (show n)
    system ("mv "++visitFile++".new "++visitFile)
    return n

visitFile = "numvisit"  -- file to store the current visitor number
```

Note the definition of `writeVisitFile`: it does not directly write the incremented number into the `visitFile` but it writes it into another file that is subsequently moved to the `visitFile`. This is necessary to avoid the overlapping of reading and writing actions on the same file due to the lazy evaluation of `readFile`.

Now the visitor form is simply obtained by calling `incVisitNumber` before generating the form [Program]:

```
visitorForm = do
  visitnum <- incVisitNumber
  return $ form "Access Count Form"
           [h1 [htxt $ "You are the " ++ show visitnum ++ ". visitor!"]]
```

5.6.4 Ensuring Exclusive Access

Since CGI programs are executed whenever a client accesses them, one has not much control on the order of their execution. In particular, the same CGI program can be executed in parallel if two clients accessing them simultaneously. This can cause a problem if both update the same information. For instance, an access to the visitor form above reads the current visitor number from the global `visitFile` and write the incremented number back. If the script is simultaneously executed by two clients, it may be the case that one update is lost (if both read the same number and write the same incremented number).

Multiple simultaneous accesses or updates can be avoided by ensuring the exclusive access to a resource on the web server between different processes running on the server. Although Curry has no direct features to support this,[2] it can be implemented by the use of the underlying operating system. For instance, Unix/Linux systems offer the command "`lockfile`" to ensure an exclusive access to a resource of the system. `lockfile` tries to create a given file (the argument to `lockfile`). If the file cannot be created (since it has been already created by another process), the `lockfile` command waits and retries after some time. Using `lockfile`, we can implement a generic function "`exclusiveIO`" that takes a name for a global lock file and exclusively executes an I/O action (the second parameter), i.e., it ensures that two processes using the same lock file do not execute the action at the same time:

[2] It could be implemented in Curry by the use of ports but this will be discussed later.

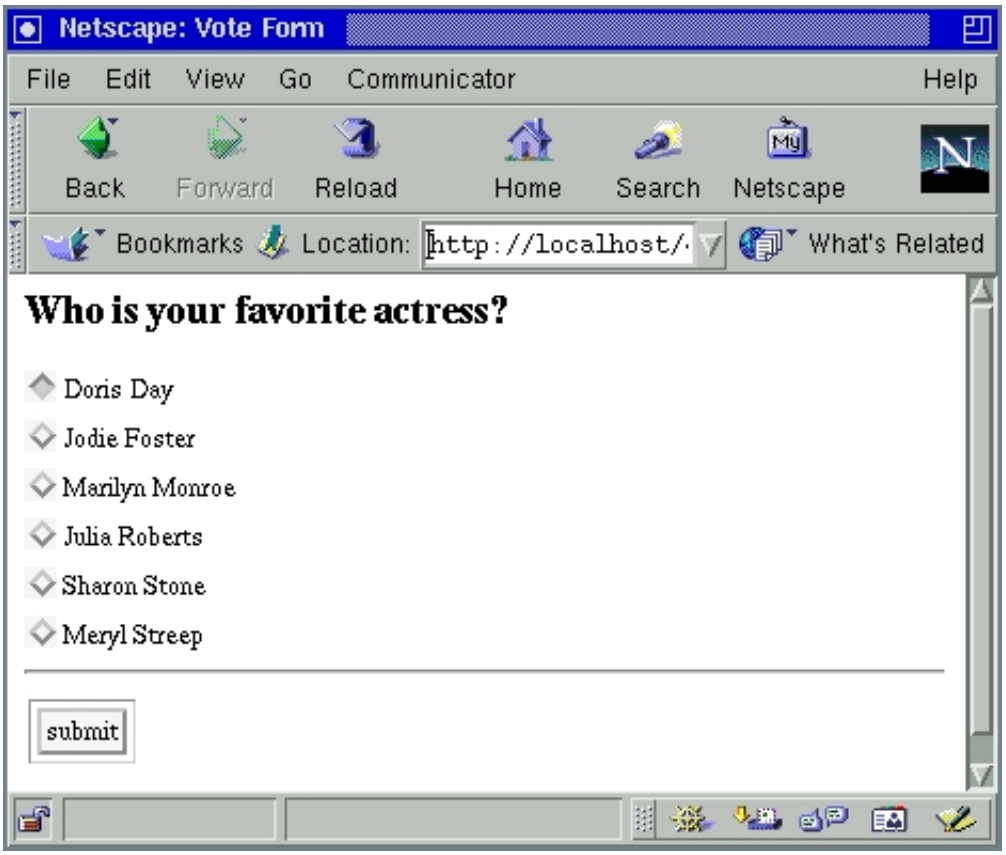

Figure 5.2: A web questionnaire

```
exclusiveIO :: String -> IO a -> IO a
exclusiveIO lockfile action =
  do system ("lockfile "++lockfile)
     actionResult <- action
     system ("rm -f "++lockfile)
     return actionResult
```

Now it is straightforward to extend our visitor form in order to ensure the exclusive update of the visitor counter. This is done by replacing the expression `incVisitNumber` in the definition of `visitorForm` by the following expression [Complete program]:

```
exclusiveIO (visitFile++".lock") incVisitNumber
```

5.6.5 Example: A Web Questionnaire

This section shows an example for web programming where the formerly discussed techniques are applied. Consider the implementation of a web-based questionnaire which allows the clients to vote on a particular topic. Figure 5.2 shows an example of such a questionnaire. The votes are stored on the web server. The current votings are shown after a client submits a vote (see Figure 5.3).

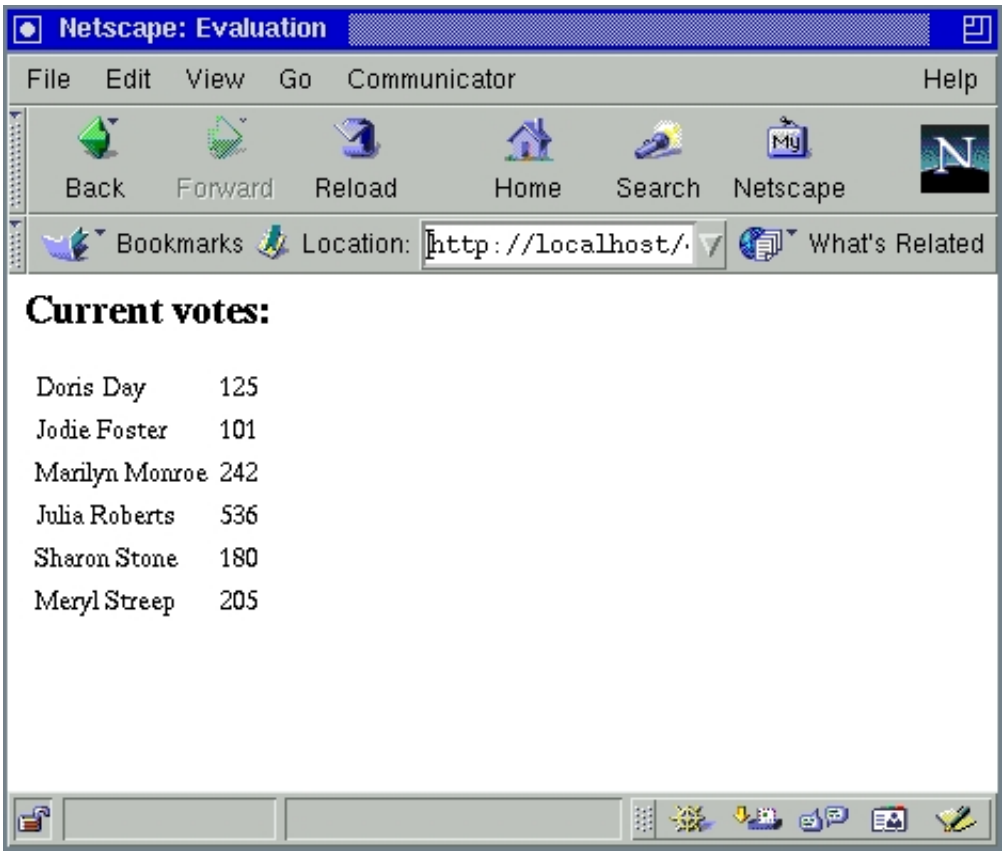

Figure 5.3: Answer to the web questionnaire

In order to provide an implementation that is easy to maintain, we define the main question and the choices for the answers as constants in our program so that they can be easily adapted to other questionnaires:

```
question = "Who is your favorite actress?"

choices = ["Doris Day","Jodie Foster","Marilyn Monroe",
           "Julia Roberts","Sharon Stone","Meryl Streep"]
```

The current votes are stored in a file on the web server. We define the name of this file as a constant in our program:

```
voteFile = "votes.data"
```

For the sake of simplicity, this file is a simple text file. If there are n choices for voting, the file has n lines where each line contains the textual representation of the number of votes for the corresponding choice. Thus, the following function defines an action that reads the vote file and returns the list of numbers in this file (the prelude function `lines` breaks a string into a list of lines where lines are separated by newline characters; `readNat` is defined in the standard library `Read` and interprets a string as a natural number):

```
readVoteFile :: IO [Int]
```

```
readVoteFile = do
  vfcont <- readFile voteFile
  return (map readNat (lines vfcont))
```

Similarly, `writeVoteFile` is an action that write a list of numbers into the vote file. Similarly to the definition of `writeVisitFile` in Section 5.6.3, the numbers are written into a new file that is moved to the vote file in order to avoid an overlapping between reading and writing the same file.

```
writeVoteFile :: [Int] -> IO ()
writeVoteFile nums = do
  writeFile (voteFile++".new") (concatMap (\n->show n++"\n") nums)
  system ("mv "++voteFile++".new "++voteFile)
  done
```

Using `writeVoteFile`, we define an action `initVoteFile` that initializes the vote file with `n` zeros if it does not exist:

```
initVoteFile :: Int -> IO ()
initVoteFile n = do
  existnumfile <- doesFileExist voteFile
  if existnumfile then done
                  else writeVoteFile (take n (repeat 0))
```

When a client submits a vote, we have to increment to corresponding number in the vote file. This can be easily done by a sequence of actions that initialize the vote file (if necessary), read the current votes and write the votes that are incremented by the function `incNth`:

```
incNumberInFile :: Int -> IO ()
incNumberInFile n = do
  initVoteFile (length choices)
  nums <- readVoteFile
  writeVoteFile (incNth nums n)

incNth :: [Int] -> Int -> [Int]
incNth []     _ = []
incNth (x:xs) n = if n==0 then (x+1):xs else x:incNth xs (n-1)
```

Now we have all auxiliary definitions that are necessary to define the web scripts. First, we show the definition of the HTML form "`evalForm`" that shows the current votes (which produces the result shown in Figure 5.3). Note that we ensure the exclusive access to the vote file by the use of the function "`exclusiveIO`" defined in Section 5.6.4 (the prelude function "`zip`" joins two lists into one list of pairs of corresponding elements):

```
evalForm :: IO HtmlForm
evalForm = do
  votes <- exclusiveIO (voteFile++".lock") readVoteFile
  return $ form "Evaluation"
   [h1 [htxt "Current votes:"],
    table (map (\(s,v)->[[htxt s],[htxt $ show v]])
```

```
                (zip choices votes))]
```

Now we can define our main form that allows the user to submit a vote (see Figure 5.2). It uses radio buttons as input elements. Radio buttons are lists of buttons where exactly one button can be turned on. Thus, all buttons have the same CGI reference but different values. When a form is submitted, the CGI environment maps the CGI reference to the value of the selected radio button. A complete radio button suite consists always of a main button (`radio_main`) which is initially on and some further buttons with the same CGI reference as the main button (`radio_others`) that are initially off. In our example, we associate to each button the index of the corresponding choice as a value. The event handler `questHandler` increments he appropriate vote number and returns the current votes with the form `evalForm` [Complete program]:

```
questForm = return $ form "Vote Form"
  ([h1 [htxt question],
    radio_main vref "0", htxt (' ':head choices), breakline] ++
   concatMap (\(i,s)->[radio_other vref (show i), htxt (' ':s), breakline])
             (zip [1..] (tail choices)) ++
   [hrule, button "submit" questHandler])
 where
   vref free

   questHandler env = do
     exclusiveIO (voteFile++".lock") (incNumberInFile (readNat (env vref)))
     evalForm
```

5.7 Finding Bugs

Since debugging of CGI programs can be quite tedious, here are some hints on how to debug CGI programs.

If the execution of the CGI program produces some run-time error (e.g., access to a non-existing files), the error messages are shown in the error log file of the web browser (ask your system administrator for the actual location of this file). Each execution of a Curry CGI program is also logged in this file. If you want to suppress the writing in the error log, you can generate the CGI program with "`makecurrycgi`" with the option "`-noerror`".

If the execution of the CGI program does not produce a run-time error but simply fails (e.g., because of an incompletely defined function are a unification failure), you will probably see the message "`No more solutions`" in the web browser instead of the expected HTML document. For the purpose of debugging, it is often useful to see the subexpressions where a reduction was not possible but failed. In this case, you can generate the CGI program by "`makecurrycgi`" with the option "`-debug`". This has the effect that some debugging code is inserted in the CGI program so that you can see the trace of all failed subexpressions in the browser (not formatted with HTML so that you should better view the source with your browser). Note that the debug option produces less efficient CGI programs so that it is better

to use this option only when necessary.

The use of logic variables as references to input elements in HTML forms ensures that typos in the name of references can be detected by the compiler (e.g., resulting in an "undeclared identifier" error message), in contrast to traditional approaches to CGI programming using plain strings as references. However, if we use the same logic variable for two different input elements, this is not detected by the compiler (which is not worse than traditional approaches where this is also not detected) but results in a run-time error that is not easy to understand due to the implementation of the `HTML` library in Curry. If the web script fails, i.e., the execution produces the message "`No more solutions`", compile it again with the option "`-debug`" as described above. If the new execution shows a failure of an expression like "`'1'=:='2'`" resulting in the subsequent failure of an equational constraint like `("FIELD_1" =:= ...)`, then you have probably used the same logic variable as references to two different input elements. Thus, you should check your source program for these possible errors.

5.8 Advanced Web Programming

This section discusses some further features which are useful for writing web applications in Curry. *Cookies* are useful to store information about the client between different web scripts. *URL parameters* can be exploited to write generic web scripts. *Style sheets* can be used to modify and add new presentation styles for web documents.

5.8.1 Cookies

Cookies are small pieces of information (represented by strings) that are stored on the client's machine when a client communicates to a web server via his browser. The web server can sent cookies to the client together with a requested web document. If the client wants to retrieve the same or another document from the web server, the client's browser sends the stored cookies together with the request for a document to the browser. Thus, cookies can be used to identify the client during a longer interaction with the web server (also across various web scripts stored on the same web browser). Cookies are another approach to handle intermediate state in web applications. The technique presented in Section 5.6.2 is only useful inside the same web script whereas cookies can be used as a link between different web scripts. However, cookies need special support on the browser's side and the client must enable cookies in his web browser. Fortunately, most web browsers support cookies since they are used in many web sites.

Basically, a cookie has a name and a value. Both parameters are of type string. Cookies can also have additional parameters to control their lifetime, validity for different web servers or regions on a web server etc (see definition of datatype "`CookieParam`" in the `HTML` library) which we will not describe here. As the default, a cookie is a valid during the client's browser session for all documents in the same directory or a subdirectory in which the cookie was set.

The `HTML` library provides two functions to set and retrieve cookies. As described above, a cookie is set by sending it with some web document. For doing so, there is the function

```
cookieForm :: String -> [(String,String)] -> [HtmlExp] -> HtmlForm
```

which behaves similarly to the function "`form`" but takes an additional parameter: a list of cookies, i.e., name/value pairs. These cookies are submitted with the form to the client's browser. To retrieve cookies (that are previously sent with a "`cookieForm`"), there is an I/O action

```
getCookies :: IO [(String,String)]
```

that returns the list of all cookies (i.e., name/value pairs) sent from the browser for the current CGI script.

As a simple example, we want to use cookies to write a web application where a user must identify himself and this identification is used in another independent script. The identification is done by setting a cookie of the form `("LOGINNAME",<name>)` where `<name>` is the user's name. We implement a "login form" that sets this cookie as follows [Program]:

```
loginForm = return $ form "Login"
      [htxt "Enter your name: ", textfield tref "", hrule,
       button "Login" handler
      ]
 where
   tref free

   handler env =
     return $ cookieForm "Logged In" [("LOGINNAME",env tref)]
                      [h2 [htxt $ env tref ++ ": thank you for visiting us"]]
```

The first form asks the user for his name. The cookie is set together with the acknowledgment form (function "`handler`").

Now we can write another web script that uses this cookie. This script shows the user's name or the string `"Not yet logged in"` if the user has not used the login form to set the cookie. Using the function `getCookies`, the implementation is quite simple (the function `lookup`, defined in the prelude, searches for a name in a name/value list; it returns "`Nothing`" of the name was not found and "`Just v`" if the first occurrence of the name in the list has the associated value `v`; the prelude function `maybe` processes these two cases) [Program]:

```
getNameForm =
   do cookies <- getCookies
      return $ form "Hello" $
       maybe [h1 [htxt "Not yet logged in"]]
             (\n->[h1 [htxt $ "Hello, " ++ n]])
             (lookup "LOGINNAME" cookies)
```

As mentioned above, cookies need special support on the client's side, i.e., the web browser of the client must support cookies. If cookies are essential for an application, one should check whether the client allows the setting of cookies. This can be done by trying to set a cookie and by checking whether this was successful. For instance, one can modify the above login script as folllows. The first form immediately sets a cookie with name "`SETCOOKIE`". Then

the handler checks whether this cookie has been sent by the client's browser. If this cookie is not received, it returns a form with the message "Sorry, can't set cookies." instead of the acknowledgment form which sets the cookie "LOGINNAME" [Program]:

```
loginForm = return $ cookieForm "Login" [("SETCOOKIE","")]
      [htxt "Enter your name: ", textfield tref "", hrule,
       button "Login" handler
      ]
 where
   tref free

   handler env = do
     cookies <- getCookies
     return $
       if lookup "SETCOOKIE" cookies == Nothing
       then form "No cookies" [h2 [htxt "Sorry, can't set cookies."]]
       else cookieForm "Logged In" [("LOGINNAME",env tref)]
                       [h2 [htxt $ env tref ++ ": thank you for visiting us"]]
```

5.8.2 URL Parameters

In some situations it is preferable to have generic web scripts that can be applied in various situations described by parameters. For instance, if we want to write a web application that allows the navigation through a hierarchical structure, one does not want to write a different script for each different level of the structure but it is preferable to write a single script that can be applied to different points in the structure. This is possible by attaching a parameter (a string) to the URL of a script. For instance, a URL can have the form "`http://myhost/script.cgi?parameter`" where "`http://myhost/script.cgi`" is the URL of the web script and "`parameter`" is an optional parameter that is passed to the script. A *URL parameter* can be retrieved inside a script by the I/O action

```
getUrlParameter :: IO String
```

which returns the part of the URL following the character "`?`". Note that an URL parameter should be "URL encoded" to avoid the appearance of characters with a special meaning. The `HTML` library provides the functions "`urlencoded2string`" and "`string2urlencoded`" to decode and encode such parameters, respectively.

As a simple example, we want to write a web script to navigate through a directory structure. The current directory is the URL parameter for this script. The script extracts this parameter by the use of `getUrlParameter` and shows all entries as a HTML list [Program]:

```
showDirForm = do
  param <- getUrlParameter
  let dir = if param=="" then "." else urlencoded2string param
  entries <- getDirectoryContents dir
  hexps <- mapIO (entry2html dir) entries
```

```
  return $ form "Browse Directory"
                [h1 [htxt $ "Directory: " ++ dir], ulist hexps]
```

The I/O action "`getDirectoryContents`" is defined in the system library `Directory` and returns the list of all entries in a directory. The function "`entry2html`" checks for an entry whether it is a directory. If this is the case, it returns a link to the same web script but with an extended parameter, otherwise it simply returns the entry name as an HTML text ("`doesDirectoryExist`" is defined in the library `Directory` and returns `True` if the argument is the name of a directory):

```
entry2html :: String -> String -> IO [HtmlExp]
entry2html dir e = do
  direx <- doesDirectoryExist (dir++"/"++e)
  if direx
   then return [href ("browsedir.cgi?" ++ string2urlencoded (dir++"/"++e))
                     [htxt e]]
   else return [htxt e]
```

Finally, the prelude function "`mapIO`" applies a mapping from elements into I/O actions to all elements of a list and collect all results in a list:

```
mapIO :: (a -> IO b) -> [a] -> IO [b]
mapIO _ []     = return []
mapIO f (x:xs) = do
  y  <- f x
  ys <- mapIO f xs
  return (y:ys)
```

5.8.3 Style Sheets

[To be completed.]

Chapter 6

Further Libraries for Application Programming

- Databases [9]
- Constraints
- GUI [7]
- XML
- Distributed Programming, ports [6]
- Metaprogramming

Bibliography

[1] S. Antoy. Definitional trees. In *Proc. of the 4th Intl. Conf. on Algebraic and Logic Programming*, pages 143–157. Springer LNCS 632, 1992.

[2] S. Antoy. Optimal non-deterministic functional logic computations. In *6th Int'l Conf. on Algebraic and Logic Programming (ALP'97)*, volume 1298, pages 16–30. Springer LNCS, 1997.

[3] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. *Journal of the ACM*, 47(4):776–822, 2000.

[4] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[5] M. Hanus. Teaching functional and logic programming with a single computation model. In *Proc. Ninth International Symposium on Programming Languages, Implementations, Logics, and Programs (PLILP'97)*, pages 335–350. Springer LNCS 1292, 1997.

[6] M. Hanus. Distributed programming in a multi-paradigm declarative language. In *Proc. of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, pages 376–395. Springer LNCS 1702, 1999.

[7] M. Hanus. A functional logic programming approach to graphical user interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pages 47–62. Springer LNCS 1753, 2000.

[8] M. Hanus. High-level server side web scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pages 76–92. Springer LNCS 1990, 2001.

[9] M. Hanus. Dynamic predicates in functional logic programs. In *Proc. 13th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2004)*, pages 62–73, Aachen (Germany), 2004. Technical Report AIB-2004-05, RWTH Aachen.

[10] M. Hanus, S. Antoy, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at http://www.informatik.uni-kiel.de/~pakcs/, 2004.

[11] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.8). Available at http://www.informatik.uni-kiel.de/~curry, 2003.

[12] S. Peyton Jones and J. Hughes. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, 1999.

[13] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, 1997.

Index