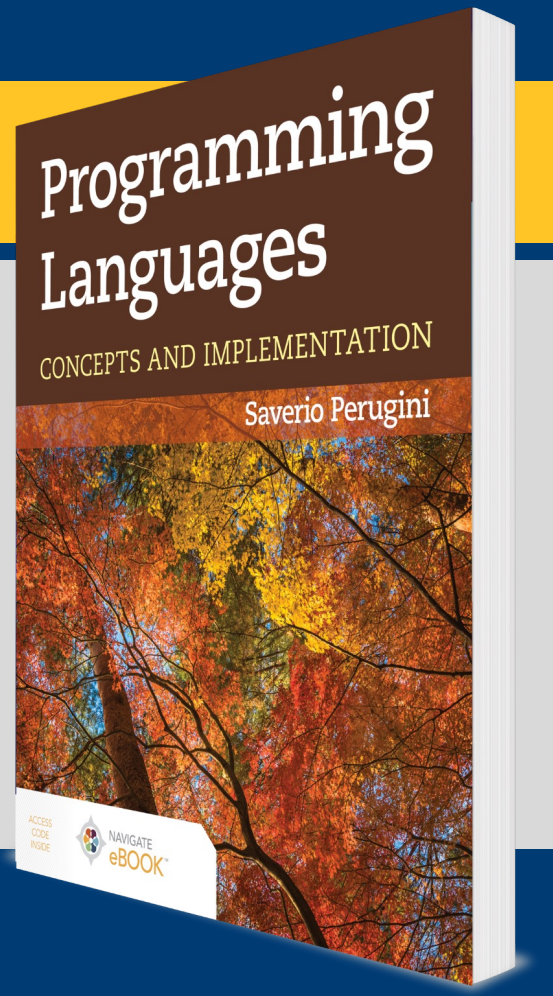
CHAPTER 8

Currying and Higher-Order Functions



Chapter 8: Currying and Higher-Order Functions

[T]here are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

—Tony Hoare, 1980 ACM A. M. Turing Award Lecture

The concept of *static typing* leads to *type inference* and *type signatures* for functions (all of which are covered in Chapter 7), which lead to the concepts of *currying* and *partial function application*, which we discuss in this chapter. All of these concepts are integrated in the context of higher-order functions, which also provide us with tools and techniques for constructing well-designed and -factored software systems, including interpreters (which we build in Chapters 10–12).

Outline

- **8.1 Chapter Objectives**
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

1.2 Chapter Objectives

- Explore the programming language concepts of *partial function application* and *currying*.
- Describe *higher-order functions* and their relationships to curried functions, which together support the development of well-designed, concise, elegant, and reusable software.

Outline

- 8.1 Chapter Objectives
- **8.2 Partial Function Application**
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

8.2 Partial Function Application (1 of 3)

With *partial function application*, for any function $f(p_1, p_2, ..., p_n)$,

$$f(a_1) = g(p_2, p_3, \cdots, p_n)$$

such that

$$g(a_2, a_3, \cdots, a_n) = f(a_1, a_2, a_3, \cdots, a_n)$$

Table 8.1 Type Signatures and λ-Calculus for a Variety of Higher-Order Functions

Concept	Function	Type Signature	λ-Calculus
com fun appl	`apply`	$: (((a \times b \times c) \to d) \times a \times b \times c) \to d$	$= \lambda(f,x,y,z).f(x,y,z)$
part fun appl 1	`papply1`	$: (((a \times b \times c) \to d) \times a) \to ((b \times c) \to d)$	$= \lambda(f,x).\lambda(y,z).f(x,y,z)$
part fun appl n	`papply`	$: (((a \times b \times c) \to d) \times a) \to ((b \times c) \to d)$	$= \lambda(f,x).\lambda(y,z).f(x,y,z)$
		$(((a \times b \times c) \to d) \times a \times b) \to (c \to d)$	$= \lambda(f,x,y).\lambda(z).f(x,y,z)$
		$(((a \times b \times c) \to d) \times a \times b \times c) \to (\{\} \to d)$	$= \lambda(f,x,y,z).\lambda().f(x,y,z)$
currying	`curry`	$: ((a \times b \times c) \to d) \to (a \to (b \to (c \to d)))$	$= \lambda(f).\lambda(x).\lambda(y).\lambda(z).f(x,y,z)$
uncurrying	`uncurry`	$: (a \to (b \to (c \to d))) \to ((a \times b \times c) \to d)$	$= \lambda(f).\ \lambda\ l.f(car\ l)(cadr\ l)(caddr\ l)$

Each signature assumes a ternary function $f : (a \times b \times c) \to d$. All of these functions except `apply` return a function. In other words, all but `apply` are closed operators.

Table 8.2 Definitions of `papply1` and `papply` in Scheme

```
(define papply1
  (lambda (fun arg)
    (lambda x
      (apply fun (cons arg x)))))
```

```
(define papply
  (lambda (fun . args)
    (lambda x
      (apply fun (append args x)))))
```

8.2 Partial Function Application (2 of 3)

More generally, with *partial function application*, for any function $f(p_1, p_2, ..., p_n)$,

$$f(a_1, a_2, \cdots, a_m) = g(p_{m+1}, p_{m+2}, \cdots, p_n)$$

where $m \leqslant n$, such that

$$g(a_{m+1}, a_{m+2}, \cdots, a_n) = f(a_1, a_2, \cdots, a_m, a_{m+1}, a_{m+2}, \cdots, a_n)$$

8.2 Partial Function Application (3 of 3)

More formally, assuming an *n*-ary function *f*, where *n* > 0:

$$\underbrace{papply\ (\cdots\ (papply\ (\ \overbrace{papply(f,\ 1)}^{(n-1)-\text{ary function}}\ ,\ 2),\ \cdots),\ n)}_{\text{argumentless function fixpoint}}$$

$(n-1)-\text{ary function}$: $(n-2)-\text{ary-function}$

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- **8.3 Currying**
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- 8.6 Thematic Takeaways

8.3 Currying

- *Currying* refers to converting an *n*-ary function into one that accepts only one argument and returns a function, which also accepts only one argument and returns a function that accepts only one argument, and so on.
- For now, we can think of a curried function as one that permits transparent partial function application (i.e., without calling `papply1` or `papply`).
- In other words, a curried function (or a function written in curried form, as discussed next) can be partially applied without calling `papply1` or `papply`.
- Later, we see that a curried function is not being partially applied at all.

8.3.1 Curried Form in Haskell (1 of 4)

- Consider the following two definitions of a power function (i.e., a function that computes a base *b* raised to an exponent *e*, b^e) in Haskell:

```
1   Prelude> :{
2   Prelude| powucf(0, _) = 1
3   Prelude| powucf(1, b) = b
4   Prelude| powucf(_, 0) = 0
5   Prelude| powucf(e, b) = b * powucf(e-1, b)
6   Prelude|
7   Prelude| powcf 0 _ = 1
8   Prelude| powcf 1 b = b
9   Prelude| powcf _ 0 = 0
10  Prelude| powcf e b = b * powcf (e-1) b
11  Prelude| :}
```

- These two definitions are almost the same.
- But the types of these functions are different.

- The definition of the `powucf` function has a comma between each parameter in the tuple of parameters, and that tuple is enclosed in parentheses
- There are no commas and parentheses in the parameters tuple in the definition of the `powcf` function.

8.3.1 Curried Form in Haskell (2 of 4)

```
12  Prelude> :type powucf
13  powucf :: (Num a, Num b, Eq a, Eq b) => (a, b) -> b
14  Prelude>
15  Prelude> :type powcf
16  powcf :: (Num t1, Num t2, Eq t1, Eq t2) => t1 -> t2 -> t2
```

- The type of the `powucf` function states that it accepts a tuple of values of a type in the Num class and returns a value of a type in the `Num` class.
- In contrast, the type of the `powcf` function indicates that it accepts a value of a type in the `Num` class and returns a function mapping a value of a type in the `Num` class to a value of the same type in the `Num` class.

8.3.1 Curried Form in Haskell (3 of 4)

The definition of `powcf` is written in *curried form*, meaning that it accepts only one argument and returns a function, also with only one argument:

```
17  Prelude> square = powcf 2
18  Prelude>
19  Prelude> :type square
20  square :: (Num t2, Eq t2) => t2 -> t2
21  Prelude>
22  Prelude> cube = powcf 3
23  Prelude>
24  Prelude> :type cube
25  cube :: (Num t2, Eq t2) => t2 -> t2
26  Prelude>
27  Prelude> (powcf 2) 3
28  9
29  Prelude> square 3
30  9
31  Prelude> (powcf 3) 3
32  27
33  Prelude> cube 3
34  27
```

8.3.1 Curried Form in Haskell (4 of 4)

- By contrast, the definition of `powucf` is written in *uncurried form*, meaning that it must be invoked with arguments for all of its parameters with parentheses around the argument list and commas between individual arguments.
- `powucf` cannot be partially applied, without the use of `papply1` or `papply`
  - It must be completely applied:

```
35  Prelude> powucf(2,3)
36  9
37  Prelude>
38  Prelude> powucf(2)
39
40  <interactive>:36:1: error:
41      Non type-variable argument in the constraint: Num (a, b)
42      (Use FlexibleContexts to permit this)
43      When checking the inferred type
44        it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b
45  Prelude>
46  Prelude> powucf 2
47
48  <interactive>:38:1: error:
49      Non type-variable argument in the constraint: Num (a, b)
50      (Use FlexibleContexts to permit this)
51      When checking the inferred type
52        it :: forall a b. (Eq a, Eq b, Num a, Num b, Num (a, b)) => b
```

8.3.2 Currying and Uncurrying (1 of 2)

In general, currying transforms a function $f_{uncurried}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \rightarrow r$$

into a function $f_{curried}$ with the type signature

$$p_1 \rightarrow (p_2 \rightarrow (\cdots \rightarrow (p_n \rightarrow r) \cdots))$$

such that

$$f_{uncurried}(a_1, a_2, \cdots, a_n) = (\cdots((f_{curried}(a_1))(a_2))\cdots)(a_n)$$

8.3.2 Currying and Uncurrying (2 of 2)

Currying $f_{uncurried}$ and running the resulting $f_{curried}$ function has the same effect as progressively partially applying $f_{uncurried}$.

Inversely, uncurrying transforms a function $f_{curried}$ with the type signature

$$p_1 \to (p_2 \to (\cdots \to (p_n \to r) \cdots))$$

into a function $f_{uncurried}$ with the type signature

$$(p_1 \times p_2 \times \cdots \times p_n) \to r$$

such that

$$f_{uncurried}(a_1, a_2, \cdots, a_n) = (\cdots((f_{curried}(a_1))(a_2))\cdots)(a_n)$$

8.3.3 The `curry` and `uncurry` Functions in Haskell

The built-in Haskell functions `curry` and `uncurry` are used to convert a binary function between uncurried and curried forms:

```
73  Prelude> :type curry
74  curry :: ((a,b) -> c) -> a -> b -> c
75  Prelude>
76  Prelude> :type uncurry
77  uncurry :: (a -> b -> c) -> (a,b) -> c
```

Table 8.3 Definitions of `curry` and `uncurry` in Curried Form in Haskell for Binary Functions

```
curry :: ((a,b)->c) -> a->b->c   uncurry :: (a->b->c) -> ((a,b)->c)
curry f a b = f (a,b)            uncurry f (a,b) = f a b
```

Notice that the definitions of `curry` and `uncurry` in Haskell are written in curried form.

Table 8.4 Definitions of `curry` and `uncurry` in Scheme for Binary Functions

```
(define curry
  (lambda (fun_ucf)
    (lambda (x)
       (lambda (y)
         (fun_ucf x y)))))
```

```
(define uncurry
   (lambda (fun_cf)
      (lambda args ; (x y)
         ((fun_cf (car args)) (cadr args)))))
                          ;; x           y
```

8.3.4 Flexibility in Curried Functions

- A curried function is more flexible than its uncurried analog because it can effectively be invoked in *n* ways, where *n* is the number of arguments its uncurried analog accepts:
  - The one and only way its uncurried analog is invoked (i.e., with all arguments as a complete application)
  - The one and only way it itself can be invoked (i.e., with only one argument)
  - *n* - 2 other ways corresponding to implicit partial applications of each returned function
- Any curried function can *effectively* be invoked with arguments for any prefix, akin to partial function application, including all of the parameters of its uncurried analog, without parentheses around the list of arguments or commas between individual arguments.

8.3.5 All Built-in Functions in Haskell Are Curried (1 of 2)

- This is why Haskell is referred to as a *fully* curried language.
- This is not the case in ML.
- Consider our final definition of `mergesort` in Haskell given in online Appendix C.
  - Neither the `mergesort` function nor the `compop` function is curried.
  - Thus, we cannot pass in the built-in < or > operators, because they are curried.

Converting an Infix Operator to a Prefix Operator

- When passing an operator as an argument to a function, the passed operator must be a prefix operator.
- Since the operators < and > are infix operators, we cannot pass them to this version of `mergesort` without first converting them to prefix operators.
- We can convert an infix operator to a prefix operator in Haskell either by wrapping it in a user-defined function or by enclosing it within parentheses:

```
44  Prelude> :type (+)
45  (+) :: Num a => a -> a -> a
46  Prelude>
47  Prelude> (+) 7 2
48  9
49  Prelude> add1 = (+) 1
50  Prelude>
51  Prelude> :type add1
52  add1 :: Num a => a -> a
53  Prelude>
54  Prelude> add1 9
55  10
```

8.3.5 All Built-in Functions in Haskell Are Curried (2 of 2)

The function `mergesort` is an ideal candidate for currying because by applying it in curried form with the < or > operators, we get back ascending-sort and descending-sort functions, respectively:

```
126  Prelude> (curry mergesort) (<) [9,8,7,6,5,4,3,2,1]
127  [1,2,3,4,5,6,7,8,9]
128  Prelude>
129  Prelude> ascending_sort = (curry mergesort) (<)
130  Prelude>
131  Prelude> :type ascending_sort
132  ascending_sort :: Ord a => [a] -> [a]
133  Prelude>
134  Prelude> ascending_sort [9,8,7,6,5,4,3,2,1]
135  [1,2,3,4,5,6,7,8,9]
136  Prelude>
137  Prelude> (curry mergesort) (>) [1,2,3,4,5,6,7,8,9]
138  [9,8,7,6,5,4,3,2,1]
139  Prelude>
140  Prelude> descending_sort = (curry mergesort) (>)
141  Prelude>
142  Prelude> :type descending_sort
143  descending_sort :: Ord a => [a] -> [a]
144  Prelude>
145  Prelude> descending_sort [1,2,3,4,5,6,7,8,9]
146  [9,8,7,6,5,4,3,2,1]
```

8.3.6 Supporting Curried Form Through First-Class Closures

- Any language with first-class closures can be used to define functions in curried form.
- For instance, because Python supports first-class closures, we can define the `pow` function in curried form in Python:

```
>>> def pfa_pow(e):
...     def pow_e(b):
...         if e == 0:
...             return 1
...         elif e == 1:
...             return b
...         elif b == 0:
...             return 0
...         else: return b * (pfa_pow(e-1)(b))
...     return pow_e
...
>>> pfa_pow(2)(3)
9
>>> square = pfa_pow(2)
>>>
>>> square(3)
9
>>> pfa_pow(3)(3)
27
>>> cube = pfa_pow(3)
>>>
>>> cube(3)
27
```

Here the curried form is too tightly woven into the function definition (like Scheme, but unlike ML/Haskell). Moreover, `pfa_pow` cannot be completely applied (again, like Scheme, but unlike ML/Haskell).

8.3.7 ML Analogs (Curried Form in ML)

Same as in Haskell

```
fun pow 0 _ = 1
|   pow 1 b = b
|   pow _ 0 = 0
|   pow e b = b * pow (e-1) b;

(*
int * int -> int
=>
int -> int -> int
 *)
val square = pow 2;
val cube = pow 3;
```

- Not all built-in ML functions are curried as in Haskell.
  - e.g., map is curried, while Int.+ is uncurried.
- Also, there are no built-in curry and uncurry functions in ML.

8.4 Putting It All Together: Higher-Order Functions

- Curried functions open up new possibilities in programming, especially with respect to higher-order functions (HOFs).
  - A HOF, such as `map` in Scheme, is a function that either accepts functions as arguments or returns a function as a value, or both.
  - Such functions capture common, typically recursive, programming patterns as functions.
- They provide the glue that enables us to combine simple functions to make more complex functions.
- Most HOFs are curried, which makes them powerful and flexible.
- Writing a program to solve a problem with HOFs requires:
  - Creative insight to discern the applicability of a HOF approach to solving a problem
  - The ability to decompose the problem and develop atomic functions at an appropriate level of granularity to foster:
    - A solution to the problem at hand by composing atomic functions with HOFs
    - The possibility of recomposing the constituent functions with HOFs to solve alternative problems in a similar manner

8.4.1 Functional Mapping

- The `map` function in ML and Haskell accepts only a unary function and returns a function that accepts a list and applies the unary function to each element of the list, and returns a list of the results.
- The HOF `map` is also built into both ML and Haskell and is curried in both.

Mapping in Haskell (left) Vis-à-Vis ML (right)

Haskell

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

square n = n*n

ans = ourmap square [1,2,3,4,5,6]

squarelist lon = map square lon

ans2 = squarelist [1,2,3,4,5,6]
```

vis-à-vis

```
squarelist = map square
```

ML

```
fun ourmap f nil = nil
| ourmap f (x::xs) = (f x)::ourmap f xs;

fun square x = x*x;

ourmap square [1,2,3,4,5,6];

fun squarelist lon = map square lon;

squarealist [1,2,3,4,5,6];
```

vis-à-vis

```
val squaralist = map square;
```

8.4.2 Functional Composition

- The function composition operator that accepts only two unary functions and returns a function that invokes the two in succession.
- In mathematics, $g \circ f = g(f(x))$, which means
  - "first apply f and then apply g" or
  - "g followed by f" or "g of f of x."
- The functional composition operator is . in Haskell and ○ in ML:

```
- (op ○);
val it = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b
- fun add3 x = x+3;
val add3 = fn : int -> int
- fun mult2 x = x*2;
val mult2 = fn : int -> int
- val add3_then_mult2 = mult2 ○ add3;
val add3_then_mult2 = fn : int -> int
- val mult2_then_add3 = add3 ○ mult2;
val mult2_then_add3 = fn : int -> int
- add3_then_mult2 4;
val it = 14 : int
- mult2_then_add3 4;
val it = 11 : int
```

8.4.3 Sections in Haskell (1 of 4)

In Haskell, any binary function or binary prefix operator (e.g., `div` and `mod`) can be converted into an equivalent infix operator by enclosing the name of the function in grave quotes (e.g., `` `div` ``):

```
Prelude> add x y = x+y
Prelude>
Prelude> :type add
add :: Num a => a -> a -> a
Prelude>
Prelude> 3 `add` 4
7
Prelude> 7 `div` 2
3
Prelude> div 7 2
3
Prelude> 7 `div` 2
3
Prelude> mod 7 2
1
Prelude> 7 `mod` 2
1
```

8.4.3 Sections in Haskell (2 of 4)

Parenthesizing an infix operator in Haskell converts it to the equivalent curried prefix operator:

```
Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude> (+) (1,2)
<interactive>:12:1: error:
    Non type-variable argument in the constraint: Num (a, b)
    (Use FlexibleContexts to permit this)
    When checking the inferred type
      it :: forall a b. (Num a, Num b, Num (a, b)) =>
                         (a, b) -> (a, b)
Prelude> (+) 1 2
3
```

8.4.3 Sections in Haskell (3 of 4)

- An operator in Haskell can be partially applied only if it is both curried and invocable in prefix form:

```
Prelude> :type (+) 1
(1 +) :: Num a => a -> a
```

- This convention also permits one of the arguments to be included in the parentheses
  - Which both converts the infix binary operator to a prefix binary operator and partially applies it in one stroke:

```
Prelude> :type (1+)
(1 +) :: Num a => a -> a
Prelude> (1+) 3
4
Prelude> :type (+3)
flip (+) 3 :: Num a => a -> a
Prelude> (+3) 1
4
```

Uses of Sections

1. Constructing simple and succinct functions. Example: `(+3)`
2. Declaring the type of an operator (because an operator itself is not a valid expression in Haskell). Example: `(+) :: Num a => a -> a -> a`
3. Passing a function to a higher-order function. Example: `map (+1)[1,2,3,4]`

8.4.3 Sections in Haskell (4 of 4)

- The same is not possible in ML because built-in operators (e.g., `+` and `*`) are not curried in ML.
- To convert an infix operator (e.g., `+` and `*`) to the equivalent prefix operator in ML, we must enclose the operator in parentheses (as in Haskell) and also include the lexeme `op` after the opening parenthesis:

```
- val add3_then_mult2 = (op o) (mult2, add3);
val add3_then_mult2 = fn : int -> int
```

- The concepts of mapping, functional composition, and sections are interrelated:

```
Prelude> inclist = map ((+) 1)
Prelude>
Prelude> :type inclist
inclist :: Num b => [b] -> [b]
Prelude>
Prelude> inclist [1,2,3,4,5,6]
[2,3,4,5,6,7]
```

8.4.4 Folding Lists

- The built-in ML and Haskell functions `foldl` (“fold left”) and `foldr` (“fold right”), like `map`, capture a common pattern of recursion.
- These list folding functions are helpful for defining a variety of functions.
- The functions `foldl` and `foldr` both accept only
  - a prefix binary function (sometimes called the folding function or the combining function),
  - a base value (i.e., the base of the recursion), and
  - a list, in that order:

```
Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
Prelude> :type foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
```

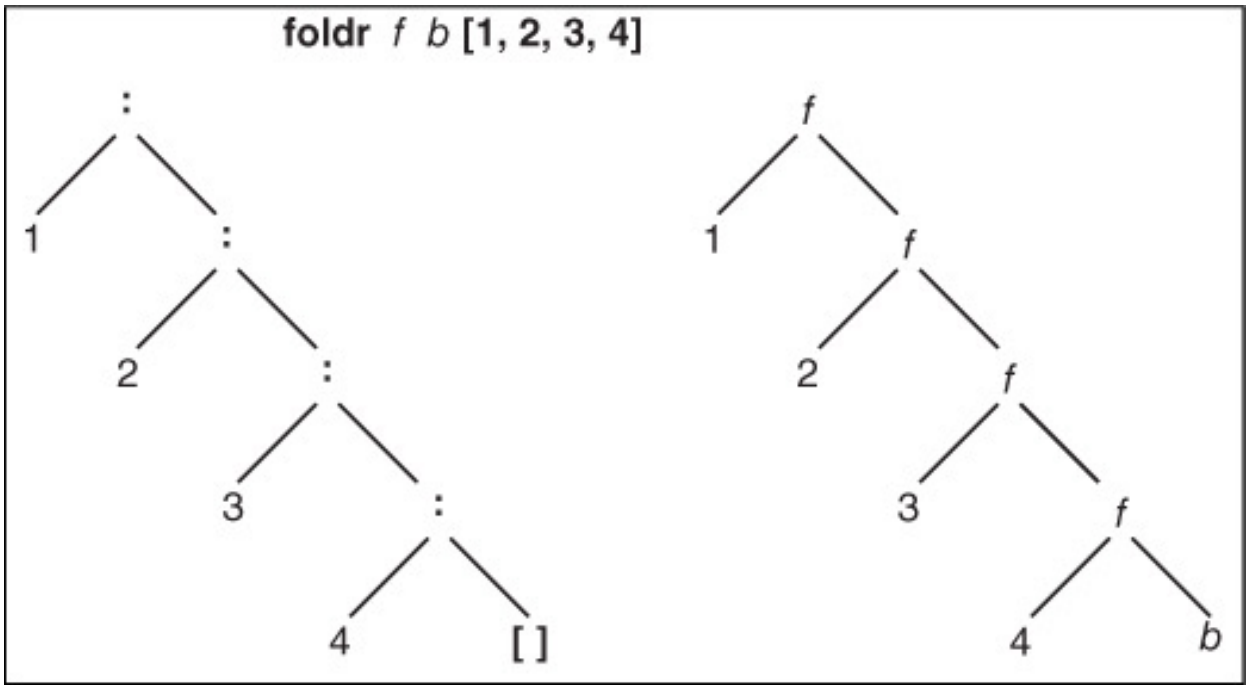
`foldr` in Haskell

The function `foldr` folds a function, given an initial value, across a list from right to left:

$$\texttt{foldr} \oplus v\,[e_0, e_1, \cdots e_n] = e_0 \oplus (e_1 \oplus (\cdots (e_{n-1} \oplus (e_n \oplus v)) \cdots))$$

where $\oplus$ is a symbol representing an operator.

Figure 8.1 `foldr` Using the Right-associative : cons Operator



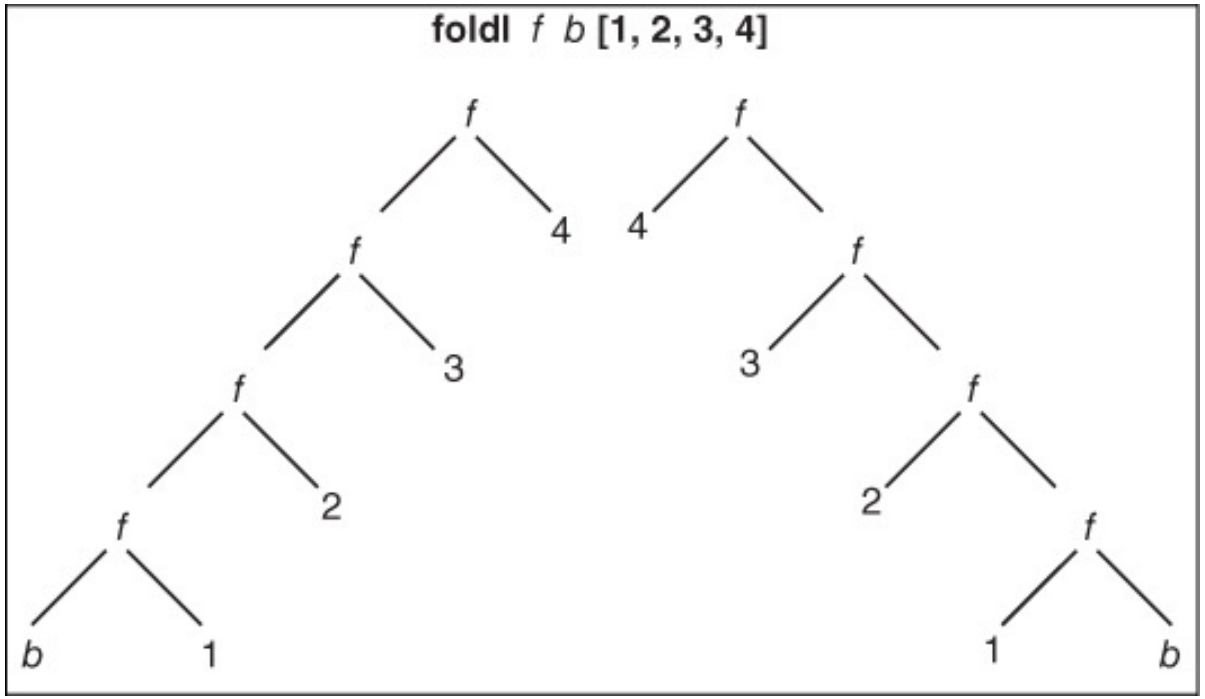
`foldl` in Haskell

The function `foldl` folds a function, given an initial value, across a list from left to right:

$$\texttt{foldl} \oplus v\,[e_0, e_1, \cdots e_n] = ((\cdots((v \oplus e_0) \oplus e_1)\cdots) \oplus e_{n-1}) \oplus e_n$$

where $\oplus$ is a symbol representing an operator. Notice that the initial value v appears on the left-hand side of the operator with `foldl` and on the right-hand side with `foldr`.

Figure 8.2 `foldl` in Haskell (left) Vis-à-Vis `foldl` in ML (right)



`foldr` in ML

- The types of `foldr` in ML and Haskell are the same.

```
- foldr;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

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

- `foldr` has the same semantics in ML and Haskell.

```
- foldr (op -) 0 [1,2,3,4]; (* 1-(2-(3-(4-0))) *)
val it = ~2 : int

Prelude> foldr (-) 0 [1,2,3,4] -- 1-(2-(3-(4-0)))
-2
```

`foldl` in ML

- The types of `foldl` in ML and Haskell differ:

```
- foldl;
val it = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

Prelude> :type foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

- `foldl` has different semantics in ML and Haskell.
- In ML, the function `foldl` is computed as follows:

$$\texttt{foldl} \oplus v\,[x_0, x_1, \cdots x_n] = x_n \oplus (x_{n-1} \oplus (\cdots \oplus (x_1 \oplus (x_0 \oplus v)) \cdots))$$

- Unlike in Haskell, `foldl` in ML is the same as `foldr` in ML (or Haskell) with a reversed list:

```
- foldl (op -) 0 [1,2,3,4]; (* 4-(3-(2-(1-0))) *)
val it = 2 : int
- foldr (op -) 0 [4,3,2,1]; (* 4-(3-(2-(1-0))) *)
val it = 2 : int

Prelude> foldr (-) 0 [4,3,2,1] -- 4-(3-(2-(1-0)))
2
```

8.4.5 Crafting Cleverly Conceived Functions with Curried HOFs

- Curried HOFs are powerful programming abstractions that support the definition of functions succinctly.
- We demonstrate the construction of the following three functions using curried HOFs:
  - `implode`: a list-to-string conversion function (online Appendix B)
  - `string2int`: a function that converts a string representing a non-negative integer to the corresponding integer
  - `powerset`: a function that computes the powerset of a set represented as a list

`implode` (1 of 3)

Consider the following `explode` and `implode` functions from online Appendix B:

```
- explode;
val it = fn : string -> char list
- explode "apple";
val it = [#"a",#"p",#"p",#"l",#"e"] : char list
- implode;
val it = fn : char list -> string
- implode [#"a", #"p", #"p", #"l", #"e"];
val it = "apple" : string
- implode (explode "apple");
val it = "apple" : string
```

`implode` (2 of 3)

- We can define `implode` using HOFs:

```
- val implode = foldr (op ^) #"";
stdIn:1.29-1.31 Error: character constant not length 1
```

- However, the string concatenation operation `^` only concatenates strings, and not characters:

```
- "hello " ^ "world";
val it = "hello world" : string
- #"h" ^ #"e";
stdIn:6.1-6.12 Error: operator and operand don't agree
[tycon mismatch]
  operator domain: string * string
  operand:         char * char
  in expression:
    #"h" ^ #"e"
```

- We need a helper function that converts a value of type `char` to value of type `string`:

```
- str;
val it = fn : char -> string
```

`implode` (3 of 3)

Now we can use the HOFs `foldr`, `map`, and `o` (i.e., functional composition) to compose the atomic elements:

```
- (* parentheses unnecessary, but present for clarity *)
- val implode = (foldr op ^ "") o (map str);
val implode = fn : char list -> string
- val implode =  foldr op ^ "" o map str;
val implode = fn : char list -> string
- implode [#"a", #"p", #"p", #"l", #"e"];
val it = "apple" : string
- foldr op ^ "" (map str [#"a", #"p", #"p", #"l", #"e"]);
val it = "apple" : string
- foldr op ^ "" ["a", "p", "p", "l", "e"];
val it = "apple" : string
- "a" ^ ("p" ^ ("p" ^ ("l" ^ ("e" ^ ""))));
val it = "apple" : string
```

Notice: The functions `map` and `foldr` (and `foldl`) are defined in curried form in ML.

`string2int` (1 of 4)

- Let's implement a function that converts a string representing a nonnegative integer into the equivalent integer.

```
- (* has type string -> int *)

- string2int "0"
0
- string2int "123"
123
- string2int "321"
321
- string2int "5643452"
5643452
```

- We can use `explode` to decompose a string into a list of chars.
- We must recognize that, for example, 123 = (3 + 0) + (2 * 10) + (1 * 100).
- We start by defining a function that converts a `char` to an `int`:

```
- fun char2int c = ord c - ord #"0";
val char2int = fn : char -> int
```

`string2int` (2 of 4)

- Now we can define another helper function that invokes `char2int` and acts as an accumulator for the integer being computed:

```
- fun helper(c, sum) = char2int c + 10*sum;
val helper = fn : char * int -> int
```

- We are now ready to glue the elements together with `foldl`:

```
(* helper (#"3", helper (#"2", helper (#"1", 0))) *)
- foldl helper 0 (explode "123");
val it = 123 : int
```

`string2int` (3 of 4)

- Since we use `foldl` in ML, we can think of the characters of the reversed string as being processed from right to left.
- The function `helper` converts the current `char`acter to an `int` and then adds that value to the product of 10 times the running sum of the integer representation of the characters to the right of the current character:

```
- foldl helper 0 (explode "123");
val it = 123 : int
- foldl helper 0 [#"1",#"2",#"3"];
val it = 123 : int
- helper(#"3", helper(#"2", helper(#"1", 0)));
val it = 123 : int
- foldl (fn (c, sum) => char2int c + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int
- foldl (fn (c, sum) =>
    ord c - ord #"0" + 10*sum) 0 [#"1",#"2",#"3"];
val it = 123 : int
```

`string2int` (4 of 4)

- Thus, we have:

```
- fun string2int s = foldl helper 0 (explode s);
val string2int = fn : string -> int
```

- After inlining an anonymous function for `helper`, the final version of the function is:

```
- fun string2int s =
    foldl (fn (c, sum) => ord c - ord #"0" + 10*sum) 0 (explode s);
val string2int = fn : string -> int
- string2int "0";
val it = 0 : int
- string2int "1";
val it = 1 : int
- string2int "123";
val it = 123 : int
- string2int "321";
val it = 321 : int
- string2int "5643452";
val it = 5643452 : int
```

powerset (1 of 2)

The following code from online Appendix B is the definition of a `powerset` function:

```
$ cat powerset.sml
fun powerset(nil) = [nil]
|   powerset(x::xs) =
        let
            fun insertineach(_, nil) = nil
            |   insertineach(item, x::xs) =
                    (item::x)::insertineach(item, xs);
            val y = powerset(xs)
        in
            insertineach(x, y)@y
        end;
$
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
```

powerset (2 of 2)

Using the HOF `map`, we can make this definition more succinct:

```
$ cat powerset.sml
fun powerset nil = [nil]
|   powerset (x::xs) =
   let
      val temp = powerset xs
   in
      (map (fn excess => x::excess) temp) @ temp
   end;
$
$ sml powerset.sml
Standard ML of New Jersey (64-bit) v110.98
[opening powerset.sml]
val powerset = fn : 'a list -> 'a list list
- powerset [1];
val it = [[1],[]] : int list list
- powerset [1,2];
val it = [[1,2],[1],[2],[]] : int list list
- powerset [1,2,3];
val it = [[1,2,3],[1,2],[1,3],[1],[2,3],[2],[3],[]] : int list list
```

Use of the built-in HOF map in this revised definition obviates the need for the nested helper function `insertineach`.

Summary

- Higher-order functions support the capture and reuse of a pattern of recursion or, more generally, a pattern of control.
- Curried HOFs provide the glue that enables programmers to compose reusable atomic functions together in creative ways.
- The resulting functions can be used in concert to craft a malleable/reconfigurable program.
- What results is a general set of (reusable) tools resembling an API rather than a monolithic program.
- This style of modular programming makes programs easier to debug, maintain, and reuse (Hughes 1989).

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- **8.5 Analysis**
- 8.6 Thematic Takeaways

8.5 Analysis (1 of 3)

- Higher-order functions capture common, typically recursive, programming patterns as functions.
- When HOFs are curried, they can be used to automatically define atomic functions—rendering the HOFs more powerful.
- Curried HOFs help programmers define functions in a modular, succinct, and easily modifiable/reconfigurable fashion.
- In this style of programming, programs are composed of a series of concise function definitions that are defined through the application of (curried) HOFs
  - map;
  - functional composition: `o` in ML and `.` in Haskell; and
  - `foldl`/`foldr`.

8.5 Analysis (2 of 3)

- Programming becomes essentially the process of creating composable building blocks and combining them like LEGO(R) bricks in creative ways to solve a problem.
- The resulting programs are more concise, modular, and easily reconfigurable than programs where each individual function is defined literally (i.e., hardcoded).
- The challenge and creativity require determining the appropriate level of granularity of the atomic functions, figuring out how to automatically define them using (built-in) HOFs, and then combining them using other HOFs into a program so that they work in concert
- Resembles building a library or API more than an application program.
- Focus is more on identifying, developing, and using the appropriate higher-order abstractions than on solving the target problem.

8.5 Analysis (3 of 3)

- Once the abstractions and essential elements have crystallized, solving the problem at hand is an afterthought.
- The pay-off, of course, is that the resulting abstractions can be reused in different arrangements in new programs to solve future problems.

Outline

- 8.1 Chapter Objectives
- 8.2 Partial Function Application
- 8.3 Currying
- 8.4 Putting It All Together: Higher-Order Functions
- 8.5 Analysis
- **8.6 Thematic Takeaways**

8.6 Thematic Takeaways

- First-class, lexical closures are an important primitive construct for creating programming abstractions (e.g., partial function application and currying).
- Higher-order functions capture common, typically recursive, programming patterns as functions.
- Currying a higher-order function enhances its power because such a function can be used to automatically define new functions.
- Curried, higher-order functions also provide the glue that enables you to combine these atomic functions to construct more complex functions.

HOFs + Currying Concise Functions + Reconfigurable Programs

HOFs + Currying (Curried HOFs) ⇝ Modular Programming