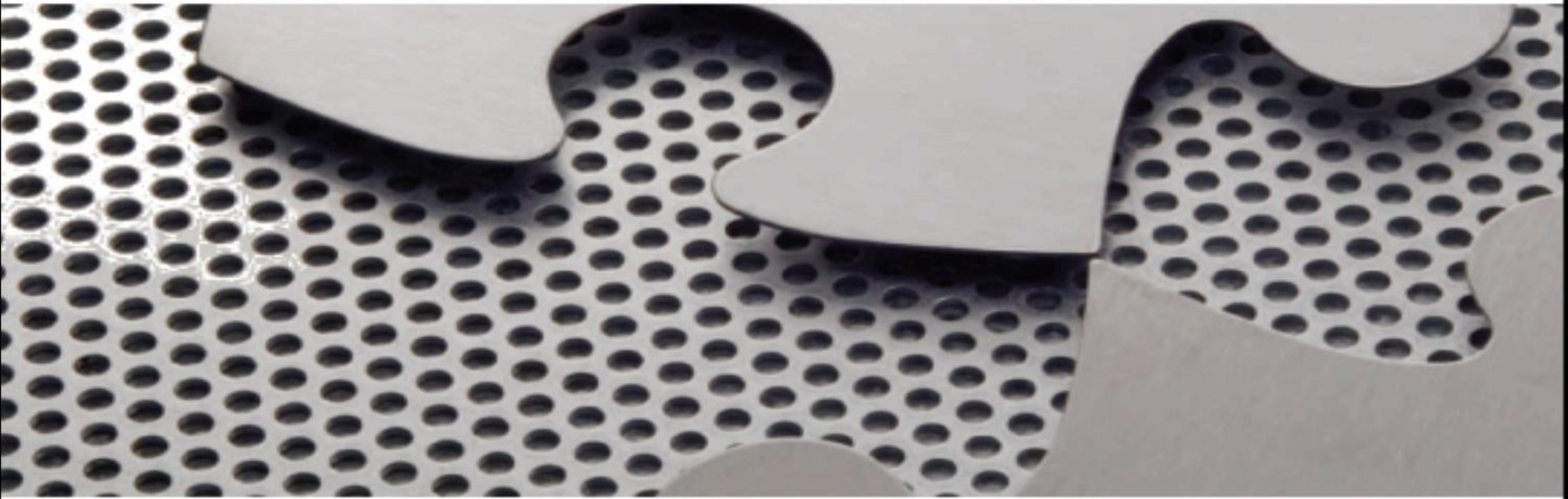
Programming Languages
Third Edition



Chapter 3

Functional Programming

Objectives

- Understand the concepts of functional programming
- Become familiar with Scheme
- Become familiar with ML
- Understand delayed evaluation
- Become familiar with Haskell
- Understand the mathematics of functional programming

Background

- Several different styles of programming, including:
  - Functional programming
  - Logic programming
  - Object-oriented programming
- Different languages have evolved to support each style of programming
  - Each type of language rests on a distinct model of computation, which is different from the von Neumann model

Background (cont'd.)

- Functional programming:
  - Provides a uniform view of programs as functions
  - Treats functions as data
  - Provides prevention of side effects
- Functional programming languages generally have simpler semantics and a simpler model of computation
  - Useful for rapid prototyping, artificial intelligence, mathematical proof systems, and logic applications

Background (cont'd.)

- Until recently, most functional languages suffered from inefficient execution
  - Most were originally interpreted instead of compiled
- Today, functional languages are very attractive for general programming
  - They lend themselves very well to parallel execution
  - May be more efficient than imperative languages on multicore hardware architectures
  - Have mature application libraries

Background (cont'd.)

- Despite these advantages, functional languages have not become mainstream languages for several reasons:
  - Programmers learn imperative or object-oriented languages first
  - OO languages provide a strong organizing principle for structuring code that mirrors the everyday experience of real objects
- Functional methods such as recursion, functional abstraction, and higher-order functions have become part of many programming languages

Programs as Functions

- A program is a description of specific computation
- If we ignore the "how" and focus on the result, or the "what" of the computation, the program becomes a virtual black box that transforms input into output
  - A program is thus essentially equivalent to a mathematical function
- **Function**: a rule that associates to each *x* from set of *X* of values a unique *y* from a set *Y* of values

Programs as Functions (cont'd.)

- In mathematical terminology, the function can be written as $y=f(x)$ or $f:X \rightarrow Y$
- **Domain** of f: the set X
- **Range** of f: the set Y
- **Independent variable**: the x in $f(x)$, representing any value from the set X
- **Dependent variable**: the y from the set Y, defined by $y=f(x)$
- **Partial function**: occurs when f is not defined for all x in X

Programs as Functions (cont'd.)

- **Total function**: a function that is defined for all x in the set X
- Programs, procedures, and functions can all be represented by the mathematical concept of a function
  - At the program level, *x* represents the input, and *y* represents the output
  - At the procedure or function level, *x* represents the parameters, and *y* represents the returned values

Programs as Functions (cont'd.)

- **Functional definition**: describes how a value is to be computed using formal parameters
- **Functional application**: a call to a defined function using actual parameters, or the values that the formal parameters assume for a particular computation
- In math, there is not always a clear distinction between a parameter and a variable
  - The term independent variable is often used for parameters

Programs as Functions (cont'd.)

- A major difference between imperative programming and functional programming is the concept of a variable
  - In math, variables always stand for actual values
  - In imperative programming languages, variables refer to memory locations that store values
- Assignment statements allow memory locations to be reset with new values
  - In math, there are no concepts of memory location and assignment

Programs as Functions (cont'd.)

- Functional programming takes a mathematical approach to the concept of a variable
  - Variables are bound to values, not memory locations
  - A variable's value cannot change, which eliminates assignment as an available operation
- Most functional programming languages retain some notion of assignment
  - It is possible to create a **pure functional program** that takes a strictly mathematical approach to variables

Programs as Functions (cont'd.)

- Lack of assignment makes loops impossible
  - A loop requires a control variable whose value changes as the loop executes
  - Recursion is used instead of loops
- There is no notion of the internal state of a function
  - Its value depends only on the values of its arguments (and possibly nonlocal variables)
- A function's value cannot depend on the order of evaluation of its arguments
  - An advantage for concurrent applications

Programs as Functions (cont'd.)

```
void gcd( int u, int v, int* x)
{ int y, t, z;
  z = u ; y = v;
  while (y != 0)
  { t = y;
    y = z % y;
    z = t;
  }
  *x = z;
}
```

(a) Imperative version using a loop

```
int gcd( int u,  int v)
{ if (v == 0) return u;
  else return gcd(v, u % v);
}
```

(b) Functional version with recursion

Figure 3.1 C code for a greatest common divisor calculation

Programs as Functions (cont'd.)

- **Referential transparency**: the property whereby a function's value depends only on the values of its variables (and nonlocal variables)
- Examples:
  - `gcd` function is referentially transparent
  - `rand` function is not because it depends on the state of the machine and previous calls to itself
- A referentially transparent function with no parameters must always return the same value
  - Thus it is no different than a constant

Programs as Functions (cont'd.)

- Referential transparency and the lack of assignment make the semantics straightforward
- **Value semantics**: semantics in which names are associated only to values, not memory locations
- Lack of local state in functional programming makes it opposite of OO programming, wherein computation proceeds by changing the local state of objects
- In functional programming, functions must be general language objects, viewed as values themselves

Programs as Functions (cont'd.)

- In functional programming, functions are **first-class data values**
  - Functions can be computed by other functions
  - Functions can be parameters to other functions
- **Composition**: essential operation on functions
  - A function takes two functions as parameters and produces another function as its returned value
- In math, the composition operator *o* is defined:

  If $f: X \rightarrow Y$ and $g: Y \rightarrow Z$, then $g \circ f: X \rightarrow Z$ is given by

  $(g \circ f)(x) = g(f(x))$

Programs as Functions (cont'd.)

- Qualities of functional program languages and functional programs:
  - All procedures are functions that distinguish incoming values (parameters) from outgoing values (results)
  - In pure functional programming, there are no assignments
  - In pure functional programming, there are no loops
  - Value of a function depends only on its parameters, not on order of evaluation or execution path
  - Functions are first-class data values

Scheme: A Dialect of Lisp

- **Lisp** (**LISt Processing**): first language that contained many of the features of modern functional languages
  - Based on the lambda calculus
- Features included:
  - Uniform representation of programs and data using a single general structure: the list
  - Definition of the language using an interpreter written in the same language (**metacircular interpreter**)
  - Automatic memory management by the runtime system

Scheme: A Dialect of Lisp (cont'd.)

- No single standard evolved for Lisp, and there are many variations
- Two dialects that use static scoping and a more uniform treatment of functions have become standard:
  - Common Lisp
  - Scheme

The Elements of Scheme

- All programs and data in Scheme are considered expressions
- Two types of expressions:
  - **Atoms**: like literal constants and identifiers of an imperative language
  - **Parenthesized expression**: a sequence of zero or more expressions separated by spaces and surrounded by parentheses
- Syntax is expressed in **extended Backus-Naur form** notation

The Elements of Scheme (cont'd.)

Table 3.1 Symbols used in an extended Backus-Naur form grammar

Symbol	Use
→	Means "is defined as"
|	Indicates an alternative
{ }	Enclose an item that may be seen zero or more times
' '	Enclose a literal item

The Elements of Scheme (cont'd.)

- Syntax of Scheme:

  `expression → atom | '(' {expression} ')'`

  `atom → number | string | symbol | character | boolean`

- When parenthesized expressions are viewed as data, they are called lists
- **Evaluation rule**: the meaning of a Scheme expression
- An **environment** in Scheme is a symbol table that associates identifiers with values

The Elements of Scheme (cont'd.)

- Standard evaluation rule for Scheme expressions:
  - Atomic literals evaluate to themselves
  - Symbols other than keywords are treated as identifiers or variables that are looked up in the current environment and replaced by values found there
  - A parenthesized expression or list is evaluated in one of two ways:
    - If the first item is a keyword, a special rule is applied to evaluate the rest of the expression
    - An expression starting with a keyword is called a **special form**

The Elements of Scheme (cont'd.)

- Otherwise, the parenthesized expression is a function application
- Each expression within the parentheses is evaluated recursively
- The first expression must evaluate to a function, which is then applied to remaining values (its arguments)

- The Scheme evaluation rule implies that all expressions must be written in prefix form
  - Example: `(+ 2 3)`
    - + is a function, and it is applied to the values 2 and 3, to return the value 5

The Elements of Scheme (cont'd.)

- Evaluation rule also implies that the value of a function (as an object) is clearly distinguished from a call to the function
  - Function is represented by the first expression in an application
  - Function call is surrounded by parentheses
- Evaluation rule represents applicative order evaluation:
  - All subexpressions are evaluated first
  - A corresponding expression tree is evaluated from leaves to root

The Elements of Scheme (cont'd.)

C	Scheme
`3 + 4 * 5`	`(+ 3 (* 4 5))`
`(a == b) && (a != 0)`	`(and (= a b) (not (= a 0)))`
`gcd(10,35)`	`(gcd 10 35)`
`gcd`	`gcd`
`getchar()`	`(read-char)`

Figure 3.2 Some expressions in C and Scheme

The Elements of Scheme (cont'd.)

- Example: `(* (+ 2 3) (+ 4 5 ))`
  - Two additions are evaluated first, then the multiplication

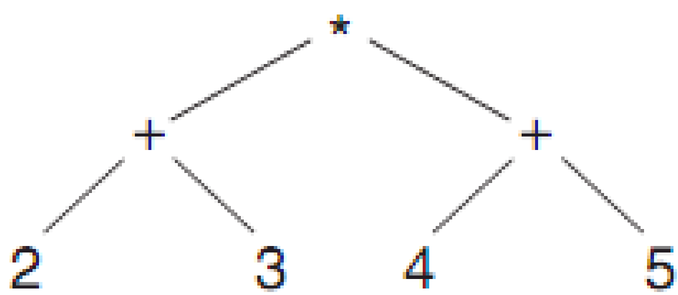


Figure 3.3 Expression tree for Scheme expression

The Elements of Scheme (cont'd.)

- A problem arises when data are represented directly in a program, such as a list of numbers
- Example: `(2.1 2.2 3.1)`
  - Scheme will try to evaluate it as a function call
  - Must prevent this and consider it to be a list literal, using a special form with the keyword `quote`
- Example: `(quote (2.1 2.2 3.1))`
- Rule for evaluating a `quote` special form is to simply return the expression following quote without evaluating it

The Elements of Scheme (cont'd.)

- Loops are provided by recursive call
- Selection is provided by special forms:
  - `if` form: like an `if-else` construct
  - `cond` form: like an `if-elseif` construct; `cond` stands for conditional expression

```
(if (= a 0) 0           ; if a = 0 then return 0
    (/ 1 a))            ; else return 1/a

(cond((= a 0) 0)        ; if a=0 then return 0
     ((= a 1) 1)        ; elsif a=1 then return 1
     (else (/ 1 a)))    ; else return 1/a
```

The Elements of Scheme (cont'd.)

- Neither the `if` nor the `cond` special form obey the standard evaluation rule
  - If they did, all arguments would be evaluated each time, rendering them useless as control mechanisms
  - Arguments to special forms are **delayed** until the appropriate moment
- Scheme function applications use pass by value, while special forms in Scheme and Lisp use delayed evaluation

The Elements of Scheme (cont'd.)

- Special form `let`: binds a variable to a value within an expression
  - Example: `(let ((a 2) (b 3)) (+ 1 b))`
    - First expression in a `let` is a **binding list**
- `let` provides a local environment and scope for a set of variable names
  - Similar to temporary variable declarations in block-structured languages
  - Values of the variables can be accessed only within the `let` form, not outside it

The Elements of Scheme (cont'd.)

- `lambda` special form: creates a function with the specified formal parameters and a body of code to be evaluated when the function is applied
  - Example:

    ```
    (lambda (radius) (* 3.14 (* radius radius)))
    ```
  - Can apply the function to an argument by wrapping it and the argument in another set of parentheses:

    ```
    ((lambda (radius) (* 3.14 (* radius radius)))
    10)
    ```

The Elements of Scheme (cont'd.)

- Can bind a name to a `lambda` within a `let`:

```
(let ((circlearea (lambda (radius) (* 3.14 (*
radius radius))))) (circlearea 10))
```

- `let` cannot be used to define recursive functions since `let` bindings cannot refer to themselves or each other
- `letrec` special form: works like a `let` but allows arbitrary recursive references within the binding list

```
(letrec ((factorial (lambda (n) (if (= n 0) 1 (*
  n (factorial (- n 1)))))))) (factorial 10))
```

The Elements of Scheme (cont'd.)

- `let` and `letrec` forms create variables visible within the scope and lifetime of the `let` or `letrec`
- `define` special form: creates a global binding of a variable visible in the top-level environment

Dynamic Type Checking

- Scheme's semantics include dynamic or latent type checking
  - Only values, not variables, have data types
  - Types of values are not checked until necessary at runtime
- Automatic type checking happens right before a primitive function, such as +
- Arguments to programmer-defined functions are not automatically checked
- If wrong type, Scheme halts with an error message

Dynamic Type Checking (cont'd.)

- Can use built-in type recognition functions such as `number?` and `procedure?` to check a value's type
  - This slows down programmer productivity and the code's execution speed

Tail and Non-Tail Recursion

- Because of runtime overhead for procedure calls, loops are always preferable to recursion in imperative languages
- **Tail recursive**: when the recursive steps are the last steps in any function
  - Scheme compiler translates this to code that executes as a loop with no additional overhead for function calls other than the top-level call
  - Eliminates the performance hit of recursion

Tail and Non-Tail Recursion (cont'd.)

Non-Tail Recursive `factorial`	Tail Recursive `factorial`

`> (define factorial`
`    (lambda (n)`
`      (if (= n 1)`
`          1`
`          (* n (factorial (- n 1))))))`
 	
`> (factorial 6)`
`720`
	`> (define factorial`
	`    (lambda (n result)`
	`      (if (= n 1)`
	`          result`
	`          (factorial (- n 1) (* n`
	`                      result)))))`
	`> (factorial 6 1)`
	`720`

Figure 3.4 Tail recursive and non-tail recursive functions

Tail and Non-Tail Recursion (cont'd.)

- Non-tail recursive function example in Figure 3.4:
  - *After* each recursive call, the value returned by the call must be multiplied by n (the argument to the previous call)
  - Requires a runtime stack to track the value of this argument for each call as the recursion unwinds
  - Entails a linear growth of memory and a substantial performance hit

Tail and Non-Tail Recursion (cont'd.)

- Tail recursive function example in Figure 3.4:
  - All the work of computing values is done when the arguments are evaluated *before* each recursive call
  - Argument result is used to accumulate intermediate products on the way down through the recursive calls
  - No work remains to be done after each recursive call, so no runtime stack is necessary to remember arguments of previous calls

Data Structures in Scheme

- Basic data structure in Scheme is the list
  - Can represent a sequence, a record, or any other structure
- Scheme also supports structured types for vectors (one-dimensional arrays) and strings
- List functions:
  - `car`: accesses the head of the list
  - `cdr`: returns the tail of the list (minus the head)
  - `cons`: adds a new head to an existing list

Data Structures in Scheme (cont'd.)

- Example: a list representation of a binary search tree

```
("horse" ("cow" () ("dog" () ()))
("zebra" ("yak" () ()) () ))
```

- A tree node is a list of three items (name left right)

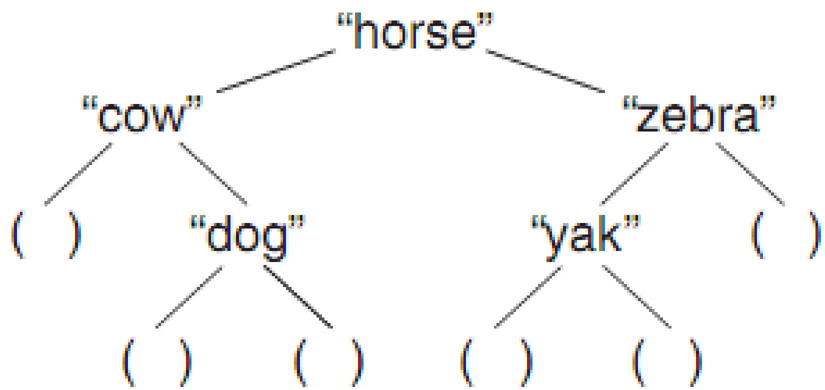


Figure 3.5 A binary search tree containing string data

Data Structures in Scheme (cont'd.)

- List can be visualized as a pair of values: the `car` and the `cdr`
  - List `L` is a pointer to a box of two pointers, one to its `car` and the other to its `cdr`

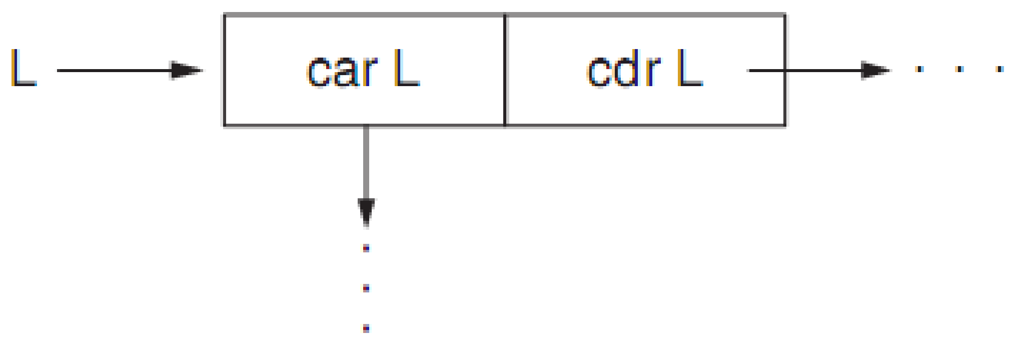


Figure 3.6 Visualizing a list with box and pointer notation

Data Structures in Scheme (cont'd.)

- **Box and pointer notation** for a simple list `(1 2 3)`
  - Black rectangle in the end box stands for the empty list ( )

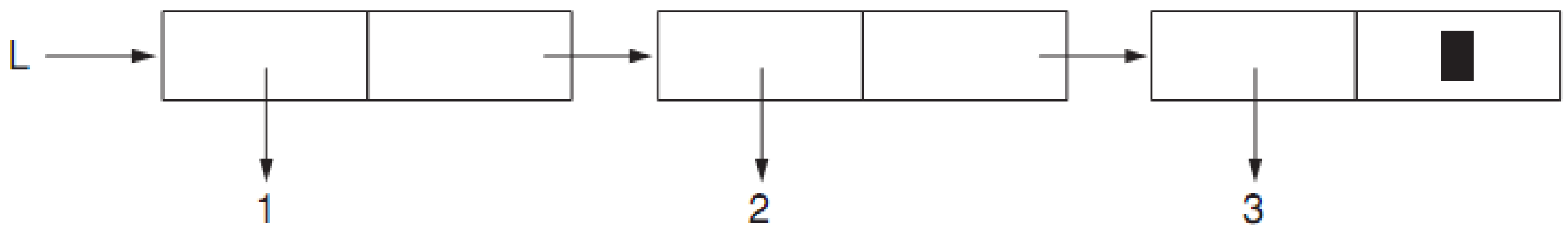


Figure 3.7 Box and pointer notation for the list `(1 2 3)`

Data Structures in Scheme (cont'd.)

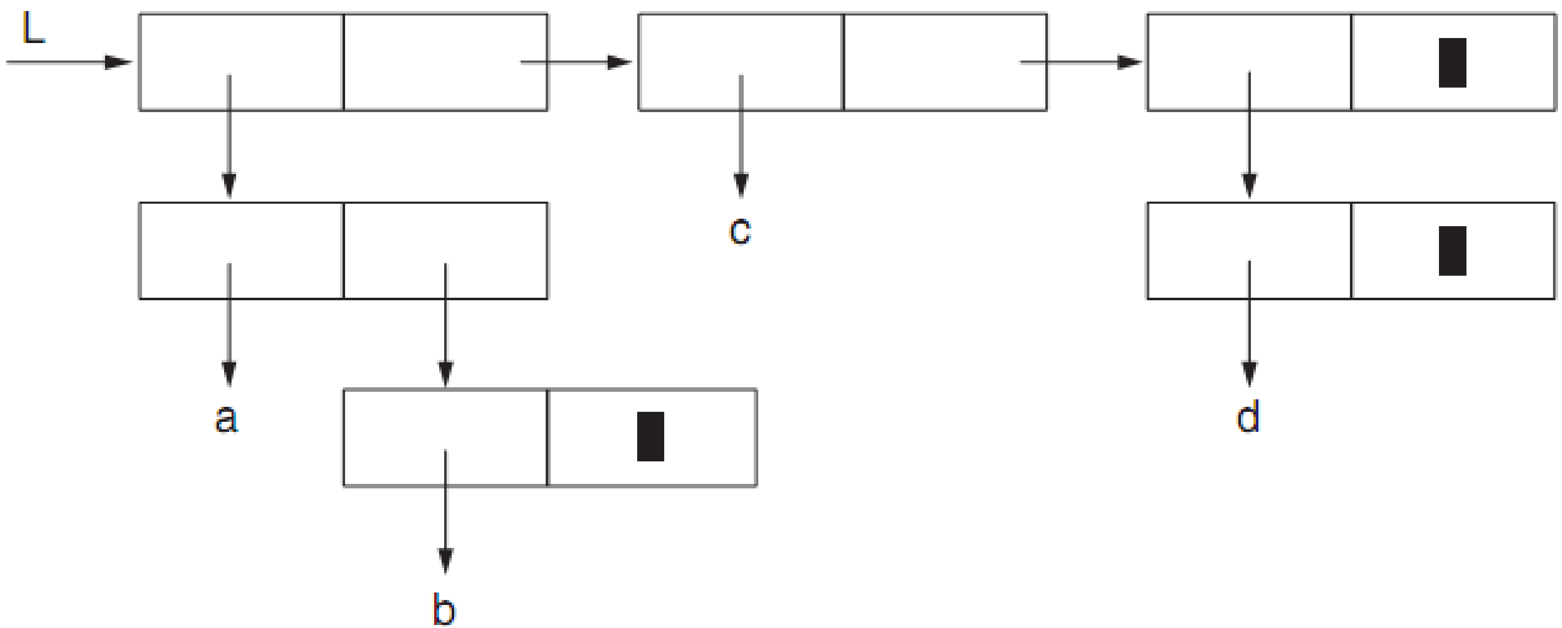


Figure 3.8 Box and pointer notation for the list `L = ((a b) c (d))`

Data Structures in Scheme (cont'd.)

- All the basic list manipulation operations can be written as functions using the primitives `car`, `cdr`, `cons`, and `null?`
  - `null?` returns true if the list is empty or false otherwise

Programming Techniques in Scheme

- Scheme relies on recursion to perform loops and other repetitive operations
  - To apply repeated operations to a list, “cdr down and cons up”: apply the operation recursively to the tail of a list and then use the `cons` operator to construct a new list with the current result
- Example:

```
(define square-list (lambda (L)
  (if (null? L) '()
    (cons (* (car L) (car L)) (square-list
      (cdr L))))))
```

Higher-Order Functions

- **Higher-order functions**: functions that take other functions as parameters and functions that return functions as values
- Example: function with a function parameter that returns a function value

```
(define make-double (lambda (f)
         (lambda (x) (f x x))))
```

- Can now create functions using this:

```
(define square (make-double *))
(define double (make-double +))
```

Higher-Order Functions (cont'd.)

- Runtime environment of functional languages is more complicated than the stack-based environment of a standard block-structured imperative language
- **Garbage collection**: automatic memory management technique to return memory used by functions

Static (Lexical) Scoping

- Early dialects of Lisp were dynamically scoped
- Modern dialects, including Scheme and Common Lisp, are statically scoped
- **Static scope** (or **lexical scope**): the area of a program in which a variable declaration is visible
  - For static scoping, the meaning or value of a variable can be determined by reading the source code
  - For dynamic scoping, the meaning depends on the runtime context

Static (Lexical) Scoping (cont'd.)

- Declaration of variables can be nested in block-structured languages
- Scope of a variable extends to the end of the block in which it is declared, including any nested blocks (unless it is redeclared within a nesting block)

```
> (let ((a 2) (b 3))
    (let ((a (+ a b)))
      (+ a b)))
8
```

Static (Lexical) Scoping (cont'd.)

- **Free variable**: a variable referenced within a function that is not also a formal parameter to that function and is not bound within a nested function
- **Bound variable**: a variable within a function that is also a formal parameter to that function
- Lexical scoping fixes the meaning of free variables in one place in the code, making a program easier to read and verify than dynamic scoping

Symbolic Information Processing and Metalinguistic Power

- **Metalinguistic power**: the capacity to build, manipulate, and transform lists of symbols that are then evaluated as programs
- Example: `let` form is actually syntactic sugar for the application of a `lambda` form to its arguments

```
(let ((a 3) (b 4))          ((lambda (a b) (* a b)) 3 4)
  (* a b))
```

Figure 3.9 `let` as syntactic sugar for the application of `lambda`

ML: Functional Programming with Static Typing

- **ML** (or **MetaLanguage**): a functional programming language quite different from the dialects of Lisp
  – Has more Algol-like syntax, which avoids the use of many parentheses
  – Is statically typed, allows for type-checking
- Advantages:
  – Makes the language more secure since more errors are found prior to execution
  – Improves efficiency by making type-checking at runtime unnecessary

ML: Functional Programming with Static Typing (cont'd.)

- ML was first developed in the late 1970s for proving the correctness of programs
  - Part of the Edinburgh Logic for Computable Functions (LCF) system
- Was later combined with the HOPE language and named Standard ML, or SML
- Current standard reflects another revision in 1997, called SML97, or ML97

The Elements of ML

- In ML, the basic program is a function declaration
- `fun`: reserved word that introduces a function declaration
- Parentheses are almost completely unnecessary since the meaning of items can be determined based solely on their position

```
> fun fact (n: int): int = if n = 0 then 1
else n * fact (n - 1);
val fact = fn: int -> int
```

The Elements of ML (cont'd.)

- A declared function can be called by its name:

```
> fact 5;
val it = 120 : int
```

- ML responds with the returned value and its type
  - `it` is the name of the current expression under evaluation
- Values can be defined using the `val` keyword

```
> val Pi = 3.14159;
val Pi = 3.14159 : real
```

The Elements of ML (cont'd.)

- Arithmetic operators are written as infix operators
  - Different from the prefix notation of Lisp
  - Operator precedence and associativity are an issue
  - ML adheres to the standard math conventions for arithmetic operators
- Can turn infix operators into prefix operators using the op keyword:

```
> op + (2 , op * ( 3,4));
val it = 14 : int
```

The Elements of ML (cont'd.)

- Note that binary arithmetic operators take pairs of integers as their argument
  - Pairs are elements of the Cartesian product type, or **tuple type** `int * int`

```
> (2,3);
val it = (2,3) : int * int
> op +;
val it = fn : int * int -> int
```

The Elements of ML (cont'd.)

- In ML, programs are not themselves lists, as they are in Lisp
- A list in ML is indicated by square brackets, with elements separated by commas
  - A list's elements must all have the same type

```
> [1,2,3];
val it = [1,2,3] : int list
```

- To mix data types, must use a **tuple**:

```
> (1,2,3.1);
val it = (1,2,3.1) : int * int * real
```

The Elements of ML (cont'd.)

- The operator `::` corresponds to `cons` in Scheme, for constructing a list out of an element (the head) and a previously constructed list (the tail)
  - Every list is constructed by a series of applications of the `::` operator, wherein `[]` is the empty list

```
> 1 :: 2 :: 3 :: [];
val it = [1,2,3] : int list
```

- **Type variable**: denoted by `'a`

```
> op :: ;
val it = fn : 'a * 'a list -> 'a list
```

The Elements of ML (cont'd.)

- ML operators hd (for **head**) and `tl` (for **tail**) correspond to Scheme's `car` and `cdr` operators

```
> hd [1,2,3];
val it = 1 : int
> tl [1,2,3];
val it = [2,3] : int list
```

- ML's pattern-matching ability makes these functions unnecessary
  - Can use `h::t` to identify the head and tail of a list

The Elements of ML (cont'd.)

- Pattern matching can eliminate most uses of `if` expressions
- Example: recursive factorial function using pattern matching:

```
fun fact 0 = 1 | fact n = n * fact (n - 1);
```

- Patterns can also contain wildcards, written as the underscore character

```
fun hd (h::_) = h | hd [] = raise Empty;
```

The Elements of ML (cont'd.)

- Because of its strong typing, you must manually convert between data types using a conversion function

```
> fun square x: real = x * x;
val square = fn : real -> real
> square (real 2);
val it = 4.0 : real
```

- ML does not allow overloading of functions

The Elements of ML (cont'd.)

- **`rev`** function: built-in function that reverses a list
- ML makes a strong distinction between types that can be compared for equality and types that cannot
  - Real numbers cannot be compared for equality
- When a polymorphic function definition involves an equality comparison, the type variables can only range over the **equality types**, written with two quotes

```
> op =;
val it = fn : ''a * ''a -> bool
```

The Elements of ML (cont'd.)

- **Structure**: ML's version of the library package
  - Includes several standard predefined resources useful for input and output
  - Examples: `TextIO` structure and `inputLine` and `output` functions
- `unit` type in ML is similar to the `void` type of C
  - Has one value `()` that represents "no actual value"
- Can convert between strings and numbers with `toString` and `fromString` functions

The Elements of ML (cont'd.)

- **Expression sequence**: a semicolon-separated sequence of expressions surrounded by parentheses, whose value is the value of the last expression listed

```
> fun printstuff () =
      ( output(stdOut,"Hello\n");
        output(stdOut,"World!\n")
      );
val printstuff = fn : unit -> unit
> printstuff ();
Hello
World!
val it = () : unit
```

Data Structures in ML

- ML has a rich set of data types, including enumerated types, records, tuples, and lists
- `type` keyword: gives a synonym to an existing data type
- `datatype` keyword produces a user-defined data type
- **Value constructors** (or **data constructors**): names used in the construction of data types that can be used as patterns
  - Vertical bar is used to indicate alternative values

Data Structures in ML (cont'd.)

- Example of a value constructor:

```
> fun heading North = 0.0 |
      heading East = 90.0 |
      heading South = 180.0 |
      heading West = 270.0 ;
val heading = fn : Direction -> real
```

- Binary search tree can be declared with `datatype`:

```
> datatype 'a BST = Nil | Node of 'a * 'a BST * 'a BST;
```

Higher-Order Functions and Currying in ML

- `fn` keyword: denotes a function expression and is followed by `=>`
  - Can be used to build anonymous functions and function return values
  - `fun` definition is just syntactic sugar for the use of an `fn` expression
- Example:

```
fun square x = x * x;
```

is equivalent to:

```
val square = fn x => x * x;
```

Higher-Order Functions and Currying in ML (cont'd.)

- `rec` keyword: used to declare a recursive function when using `fn`
  - Similar to Scheme `letrec`

```
val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

- Function composition can be done with the letter o

```
> val double_square = double o square;
val double_square = fn : int -> int
> double_square 3;
val it = 18 : int
```

Higher-Order Functions and Currying in ML (cont'd.)

- **Currying**: a process in which a function of multiple parameters is viewed as a higher-order function of a single parameter that returns a function of the remaining parameters
  - A function to which this process is applied is said to be **curried**
- Can use a tuple to get an "uncurried" version of a function or two separate parameters to get a curried version

Higher-Order Functions and Currying in ML (cont'd.)

- A language is said to be **fully curried** if function definitions are automatically treated as curried and all multiparameter built-in functions are curried
  - ML is not fully curried since all built-in binary operators are defined as taking tuples

Delayed Evaluation

- In a language with an applicative order evaluation rule, all parameters to user-defined functions are evaluated at the time of a call
- Examples that do not use applicative order evaluation:
  - Boolean special forms `and` and `or`
  - `if` special form
- Short-circuit evaluation of Boolean expressions allows a result without evaluating the second parameter

Delayed Evaluation (cont'd.)

- Delayed evaluation is necessary for `if` special form
- Example: `(if a b c)`
  - Evaluation of b and c must be delayed until the result of a is known; then either b or c is evaluated, but not both
- Must distinguish between forms that use standard evaluation (function applications) and those that do not (special forms)
- Using applicative order evaluation for functions makes semantics and implementation easier

Delayed Evaluation (cont'd.)

- **Nonstrict**: a property of a function in which delayed evaluation leads to a well-defined result, even though subexpressions or parameters may be undefined
- Languages with the property that functions are strict are easier to implement, although nonstrictness can be a desirable property
- Algol60 included delayed execution in its pass by name parameter passing convention
  - A parameter is evaluated only when it is actually used in the code of a called procedure

Delayed Evaluation (cont'd.)

- Example: Algol60 delayed execution

```
function p(x: boolean; y: integer): integer;
begin
  if x then p := 1
  else p := y;
end;
```

- When called as `p(true, 1 div 0)`, it returns 1 since y is never reached in the code of p
  - The undefined expression `1 div 0` is never computed

Delayed Evaluation (cont'd.)

- In a language with function values, it is possible to delay evaluation of a parameter by enclosing it in a function "shell" (a function with no parameters)
- Example: C pass by name equivalent

```
typedef int (*IntProc) ();
int divByZero ()
{  return 1 / 0;
}
int p(int x, IntProc y)
{ if (x) return 1;
  else return y();
}
```

Delayed Evaluation (cont'd.)

- Such "shell" procedures are sometimes referred to as **pass by name thunks**, or just **thunks**
- In Scheme and ML, the `lambda` and `fn` function value constructors can be used to surround parameters with function shells
- Example:

```
(define (p x y) (if x 1 (y)))
```

which can be called as follows:

```
(p #T (lambda () (/ 1 0)))
```

Delayed Evaluation (cont'd.)

- `delay` special form: delays evaluation of its arguments and returns an object like a `lambda` "shell" or **promise** to evaluate its arguments
- `force` special form: causes its parameter, a delayed object, to be evaluated
- Previous function can now be written as:

```
(define ( p x y) (if x 1 (force y)))
```

and called as:

```
(p #T (delay (/ 1 0)))
```

Delayed Evaluation (cont'd.)

- Delayed evaluation can introduce inefficiency when the same delayed expression is repeatedly evaluated
- Scheme uses a **memoization** process to store the value of the delayed object the first time it is forced and then return this value for each subsequent call to force
  - This is sometimes referred to as **pass by need**

Delayed Evaluation (cont'd.)

- **Lazy evaluation**: only evaluate an expression once it is actually needed
- This can be achieved in a functional language without explicit calls to `delay` and `force`
- Required runtime rules for lazy evaluation:
  - All arguments to user-defined functions are delayed
  - All bindings of local names in `let` and `letrec` expressions are delayed
  - All arguments to constructor functions are delayed

Delayed Evaluation (cont'd.)

- Required runtime rules for lazy evaluation (cont'd.):
  - All arguments to other predefined functions are forced
  - All function-valued arguments are forced
  - All conditions in selection forms are forced
- Lists that obey lazy evaluation may be called **streams**
- Primary example of a functional language with lazy evaluation is Haskell

Delayed Evaluation (cont'd.)

- **Generator-filter programming**: a style of functional programming in which computation is separated into procedures that generate streams and other procedures that take streams as arguments
- **Generators**: procedures that generate streams
- **Filters**: procedures that modify streams
- **Same-fringe** problem for lists: two lists have the same fringe if they contain the same non-null atoms in the same order

Delayed Evaluation (cont'd.)

- Example: these lists have the same fringe:
  `((2 (3)) 4)` and `(2 (34 ()))`
- To determine if two lists have the same fringe, must **flatten** them to just lists of their atoms
- `flatten` function: can be viewed as a filter; reduces a list to a list of its atoms
- Lazy evaluation will compute only enough of the flattened lists as necessary before their elements disagree

Delayed Evaluation (cont'd.)

- Delayed evaluation complicates the semantics and increases complexity in the runtime environment
  - Delayed evaluation has been described as a form of parallelism, with `delay` as a form of process suspension and `force` as a kind of process continuation
- Side effects, in particular assignment, do not mix well with lazy evaluation

Haskell – A Fully Curried Lazy Language with Overloading

- **Haskell**: a pure functional language developed in the late 1980s
- Builds on and extends a series of purely functional lazy languages
- Contains a number of novel features, including function overloading and a mechanism called **monads** for dealing with side effects such as I/O

Elements of Haskell

- Haskell's syntax is very similar to that of ML
  - Uses a **layout rule** with indentation and line formatting to resolve ambiguities
- Differences from ML:
  - Cannot redefine any predefined functions
  - `cons` operator is written as a single colon
  - Types are given using a double colon
  - Pattern matching does not require the use of the `.` symbol
  - List concatenation is given by the ++ operator

Elements of Haskell (cont'd.)

- Haskell is a fully curried language, with all predefined operators curried
- **Section** construct: allows a binary operator to be partially applied to either argument using parentheses
- Examples:
  - `plus2 = (2 +)` defines a function that adds 2 to its argument on the left
  - `times3 = (* 3)` defines a function that multiples 3 times its argument on the right

Elements of Haskell (cont'd.)

- Infix functions can be turned into prefix functions by surrounding them with parentheses

```
> (+) 2 3
5
> (*) 4 5
20
```

- Haskell has anonymous functions or lambda forms, with the backslash representing the lambda

```
> (\x -> x * x ) 3
9
```

Higher-Order Functions and List Comprehensions

- Haskell includes many predefined higher-order functions, such as `map`, that are all in curried form
- It has built-in lists and tuples, type synonyms, and user-defined polymorphic types

```
type ListFn a = [a] -> [a]
type Salary = Float
type Pair a b = (a,b)
data BST a = Nil | Node a (BST a) (BST a)
```

Higher-Order Functions and List Comprehensions (cont'd.)

- Type variables are written without the quote of ML and are written after the data type name, not before
- `data` keyword replaces ML's `datatype` keyword
- Type and constructor names must be uppercase, while function and value names must be lowercase
- Functions on new data type can use data constructors as patterns, as in ML

```
flatten:: BST a -> [a]
flatten Nil = []
flatten (Node val left right) =
          (flatten left) ++ [val] ++ (flatten right)
```

Higher-Order Functions and List Comprehensions (cont'd.)

- **List comprehension**: a special notation for operations applied to lists
- Example: squaring a list of integers

```
square_list lis = [ x * x | x <- lis]
```

- This is syntactic sugar for:

```
square_list_positive lis = [ x * x | x <- lis, x > 0]
```

Lazy Evaluation and Infinite Lists

- Haskell is a lazy language – no value is computed unless it is actually needed
  - Lists in Haskell are the same as streams and can be potentially infinite
- Haskell has several shorthand notations for infinite lists, such as `[n..]`, which means a list of integers beginning with n
- `take` function: extracts the first n items from a list
- `drop` function: discards the first n items from a list

Type Classes and Overloaded Functions

- Haskell allows overloading of functions
- **Type class**:
  - A set of types that all define certain functions
  - Specifies the names and types (called signatures) of the functions that every type belonging to it must define
  - Similar to Java interfaces

```
class Num a where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs            :: a -> a
```

Type Classes and Overloaded Functions (cont'd.)

- **Instance definition**: contains the actual working definitions for each of the required functions

```
instance Num Int where
    (+)             = primPlusInt
    (-)             = primMinusInt
    negate          = primNegInt
    (*)             = primMulInt
    abs             = absReal
```

- Many type classes themselves are defined to be part of other type classes
  – This dependency is called **type class inheritance**

Type Classes and Overloaded Functions (cont'd.)

- Type inheritance relies upon a hierarchy of type classes
- `Eq` and `Show` classes are the base classes
  - All predefined Haskell types are instances of the `Show` class
  - `Eq` class establishes the ability of two values of a member type to be compared using == operator

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x == y      = not (x/=y)
    x /= y      = not (x==y)
```

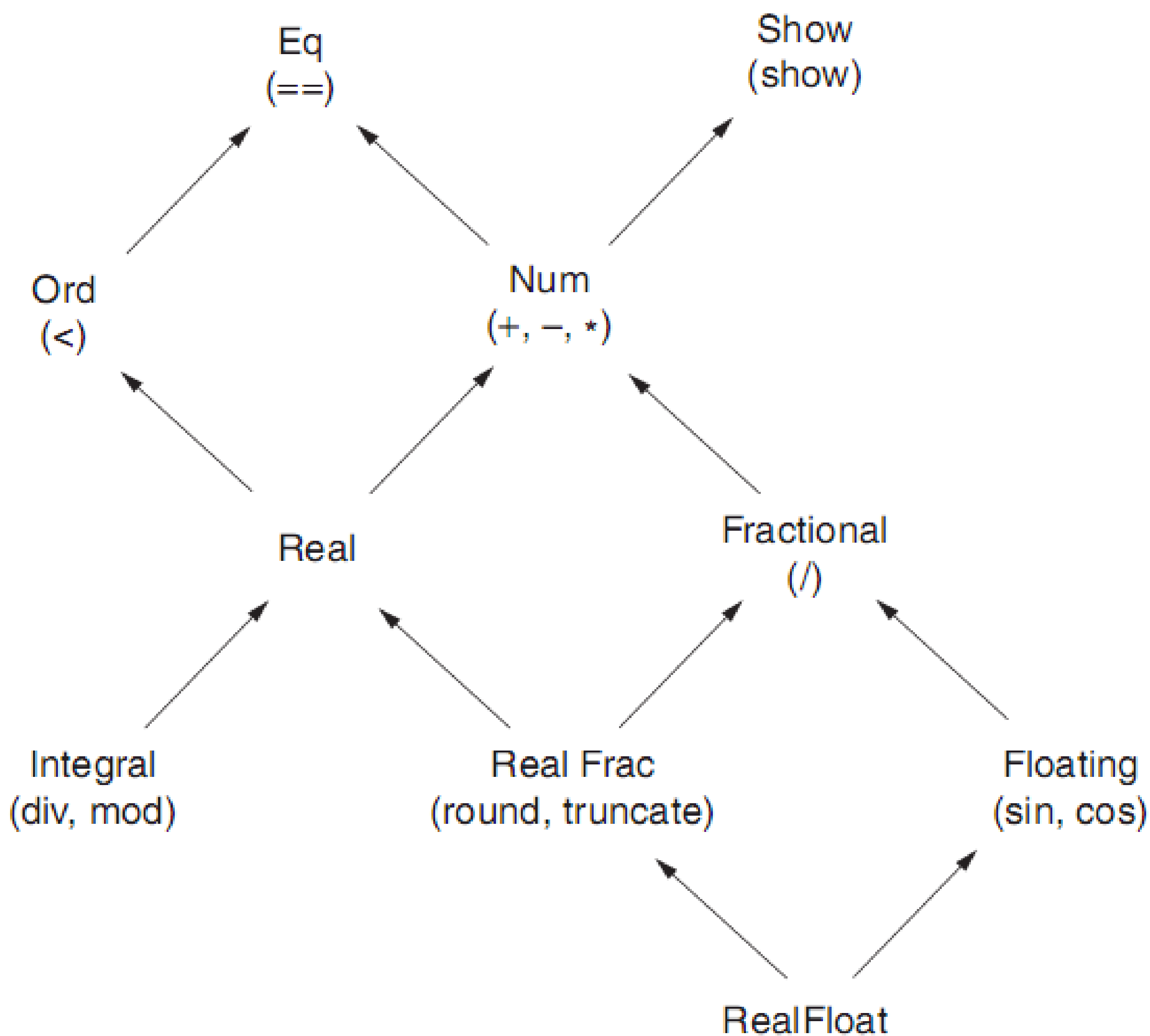


Figure 3.10 The numeric type class hierarchy in Haskell, with sample functions required by some of the classes in parentheses

The Mathematics of Functional Programming: Lambda Calculus

- **Lambda calculus**: invented by Alonzo Church in the 1930s
  - A mathematical formalism for expressing computation by functions
  - Can be used as a model for purely functional programming languages
- Many functional languages, including Lisp, ML and Haskell, were based on lambda calculus

Lambda Calculus (cont'd.)

- **Lambda abstraction**: the essential construct of lambda calculus: $(\lambda\, x\, .\, 1\ 1\ x)$
- Can be interpreted exactly as this Scheme lambda expression: `(lambda (x) (+ 1 x))`
  - An unnamed function of parameter x that adds 1 to x
- Basic operation of lambda calculus is the **application** of expressions such as the lambda abstraction

Lambda Calculus (cont'd.)

- This expression: $(\lambda\, x\, .\, +\, 1\, x)\, 2$
  - Represents the application of the function that adds 1 to x to the constant 2
- A **reduction rule** permits 2 to be substituted for x in the lambda, yielding this:

$$(\lambda\, x\, .\, +\, 1\, x)\, 2 \Rightarrow (+\, 1\, 2) \Rightarrow 3$$

Lambda Calculus (cont'd.)

- Syntax for lambda calculus:

$$
\begin{aligned}
exp \rightarrow\ & constant \\
|\ & variable \\
|\ & (exp\ exp) \\
|\ & (\lambda\ variable\ .\ exp)
\end{aligned}
$$

- Third rule represents function application
- Fourth rule gives lambda abstractions
- Lambda calculus as defined here is fully curried

Lambda Calculus (cont'd.)

- Lambda calculus variables do not occupy memory
- The set of constants and the set of variables are not specified by the grammar
  - It is more correct to speak of many **lambda calculi**
- In the expression $(\lambda x.E)$
  - x is **bound** by the lambda
  - The expression E is the scope of the binding
  - **Free occurrence**: any variable occurrence outside the scope
  - **Bound occurrence**: an occurrence that is not free

Lambda Calculus (cont'd.)

- Different occurrences of a variable can be bound by different lambdas
- Some occurrences of a variable may be bound, while others are free
- Can view lambda calculus as modeling functional programming:
  - A lambda abstraction as a function definition
  - Juxtaposition of two expressions as function application

Lambda Calculus (cont'd.)

- **Typed lambda calculus**: more restrictive form that includes the notion of data type, thus reducing the set of expressions that are allowed
- Precise rules must be given for transforming expressions
- **Substitution** (or **function application**): called **beta-reduction** in lambda calculus
- **Beta-abstraction**: reversing the process of substitution
- **Beta-conversion**: either beta-reduction or beta-abstraction

Lambda Calculus (cont'd.)

- **Name capture** problem: when doing beta-conversion and replacing variables that occur in nested scopes, an incorrect reduction may occur
  - Must change the name of the variable in the inner lambda abstraction (**alpha-conversion**)
- **Eta-conversion**: allows for the elimination of "redundant" lambda abstractions
  - Helpful in simplifying curried definitions in functional languages

Lambda Calculus (cont'd.)

- Applicative order evaluation (pass by value) vs. normal order evaluation (pass by name)
- Example: evaluate this expression: $((\lambda x.\ *\ x\ x)\ (+\ 2\ 3))$
  - Use applicative order; replacing (1 2 3) by its value and then applying beta-reduction gives:

$$((\lambda x.\ *\ x\ x)\ (+\ 2\ 3)) \Rightarrow ((\lambda x.\ *\ x\ x)\ 5) \Rightarrow (*\ 5\ 5) \Rightarrow 25$$

  - Use normal order; applying beta-reduction first and then evaluating gives:

$$((\lambda x.\ *\ x\ x)\ (+\ 2\ 3)) \Rightarrow (*\ (+\ 2\ 3)\ (+\ 2\ 3)) \Rightarrow (*\ 5\ 5) \Rightarrow 25$$

- Normal order evaluation is a kind of **delayed** evaluation

Lambda Calculus (cont'd.)

- Different results can occur, such as when parameter evaluation gives an undefined result
  - Normal order will still compute the correct value
  - Applicative order will give an undefined result
- Functions that can return a value even when parameters are undefined are said to be **nonstrict**
- Functions that are undefined when parameters are undefined are said to be **strict**
- **Church-Rosser theorem**: reduction sequences are essentially independent of the order in which they are performed

Lambda Calculus (cont'd.)

- **Fixed point**: a function that when passed to another function as an argument returns a function
- To define a recursive function in lambda calculus, we need a function Y for constructing a fixed point of the lambda expression for the function
  - Y is called a **fixed-point combinator**
- Because by its nature, Y will actually construct a solution that is in some sense the "smallest"; one can refer to the **least-fixed-point semantics** of recursive functions in lambda calculus