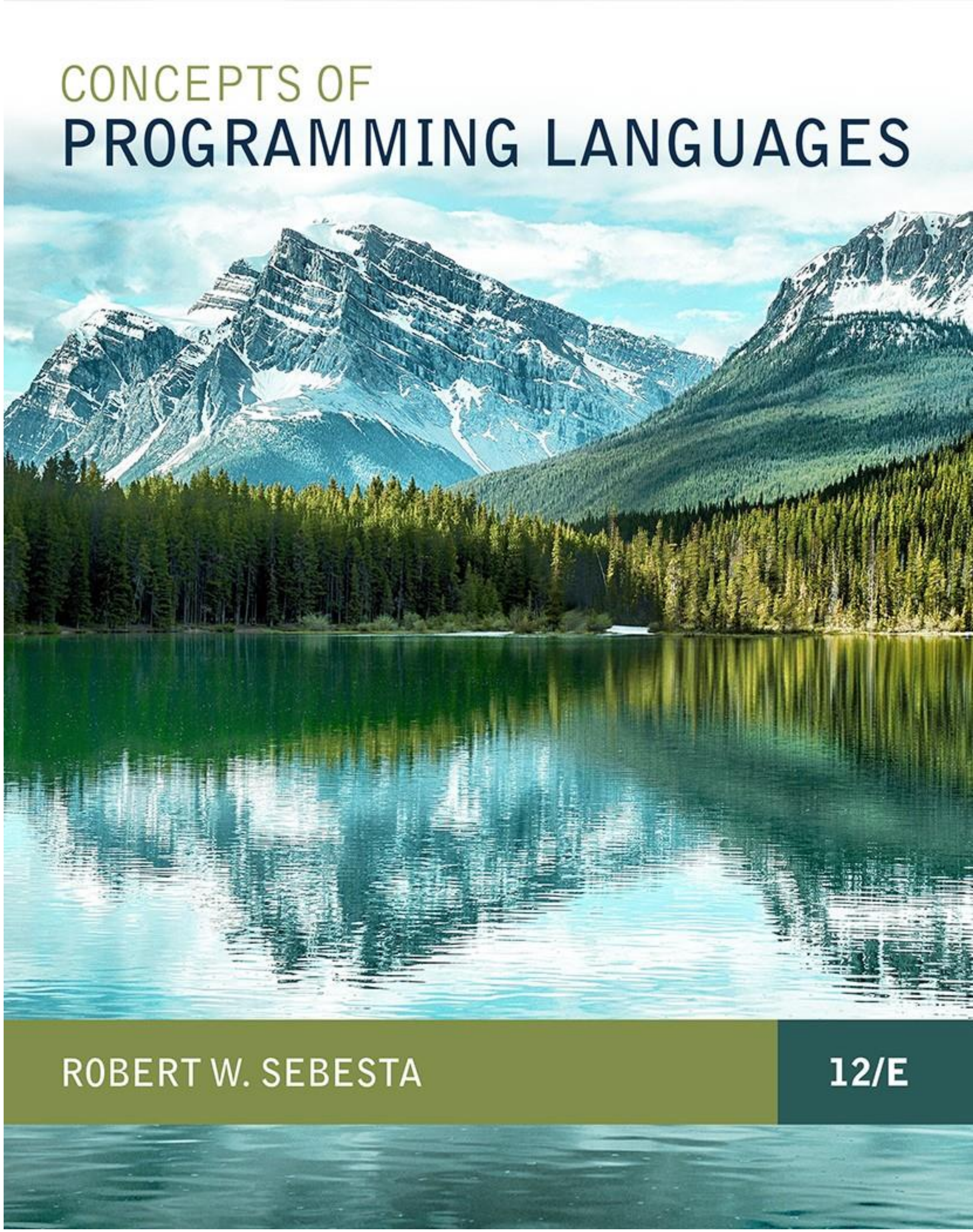
Chapter 15

Functional Programming Languages



ISBN 0-321-49362-1

Chapter 15 Topics

- Introduction
- Mathematical Functions
- Fundamentals of Functional Programming Languages
- The First Functional Programming Language: LISP
- Introduction to Scheme
- Common LISP
- ML
- Haskell
- F#
- Support for Functional Programming in Primarily Imperative Languages
- Comparison of Functional and Imperative Languages

서론

- 명령형 언어는 *von Neumann* 구조를 기반으로 설계됨
  - 소프트웨어 개발에 적합한 언어 관점보다는 효율성이 주 관심
- 함수형 언어는 수학 함수에 기반하여 설계

수학 함수

- 수학함수는 정의역 집합의 원소를 치역 집합으로의 사상(mapping)이다.
- 함수정의는 함수명, 매개변수리스트, 사상표현식으로 구성

  (예) 함수 cube(x) ≡ x * x * x, 단 x는 실수
- 특성
  - 사상표현식의 제어는 재귀나 조건식
  - 부작용(side effect) 없음
  - 함수의 상태라는 개념이 없다.

람다 표현식(Lambda expressions)

- 람다 표현식은 이름이 없는 함수를 기술하는 방법

  (예) λ(x) x * x * x

- 람다 표현식의 적용

  –표현식 이후에 매개변수 값을 위치시켜 적용

  (예) (λ(x) x * x * x)(2)

  ➔ 평가결과는 8

합성 함수(function composition)

- 두 개의 함수 매개변수를 취하여 두 번째 실 매개변수 함수의 결과에 첫 번째 실 매개변수 함수를 적용한 결과를 값으로 갖는 함수를 산출

  Form: $h \equiv f \circ g$

  which means $h(x) \equiv f(g(x))$

  (예) For $f(x) \equiv x + 2$ and $g(x) \equiv 3 * x$,

  $h \equiv f \circ g$ yields $(3 * x) + 2$

모두-적용(Apply-to-all) 함수

- 매개변수로서 단일 함수를 취하는 범함수 형태;
- 인자 리스트에 적용했다면, 모두-적용 함수는 인자 리스트에 있는 각 값에 범함수 매개변수를 적용하여 결과를 리스트나 수열로 수집

Form: α

For `h(x) ≡ x * x`

α`(h, (2, 3, 4))` yields `(4, 9, 16)`

함수형 프로그래밍 언어의 원리

- FPL의 설계 목적은 가능한 최대로 수학 함수를 모방하자는 것임
- FPL에서 계산의 기본 처리과정은 명령형 언어에서 사용한 방법과 근본적으로 다르다.
  - 명령형 언어에서는, 연산이 수행되고 결과를 변수에 저장하여 나중에 사용하도록 함
  - 변수의 관리는 지속적인 관심의 대상이며 명령형 언어의 복잡성의 원인이 된다.
- FPL에서는, 수학 함수처럼 변수가 필수적이 아님
- 참조 투명성(*Referential Transparency)* – FPL에서, 함수의 평가는 동일한 매개변수가 주어졌을 때 항상 동일한 결과를 내준다.

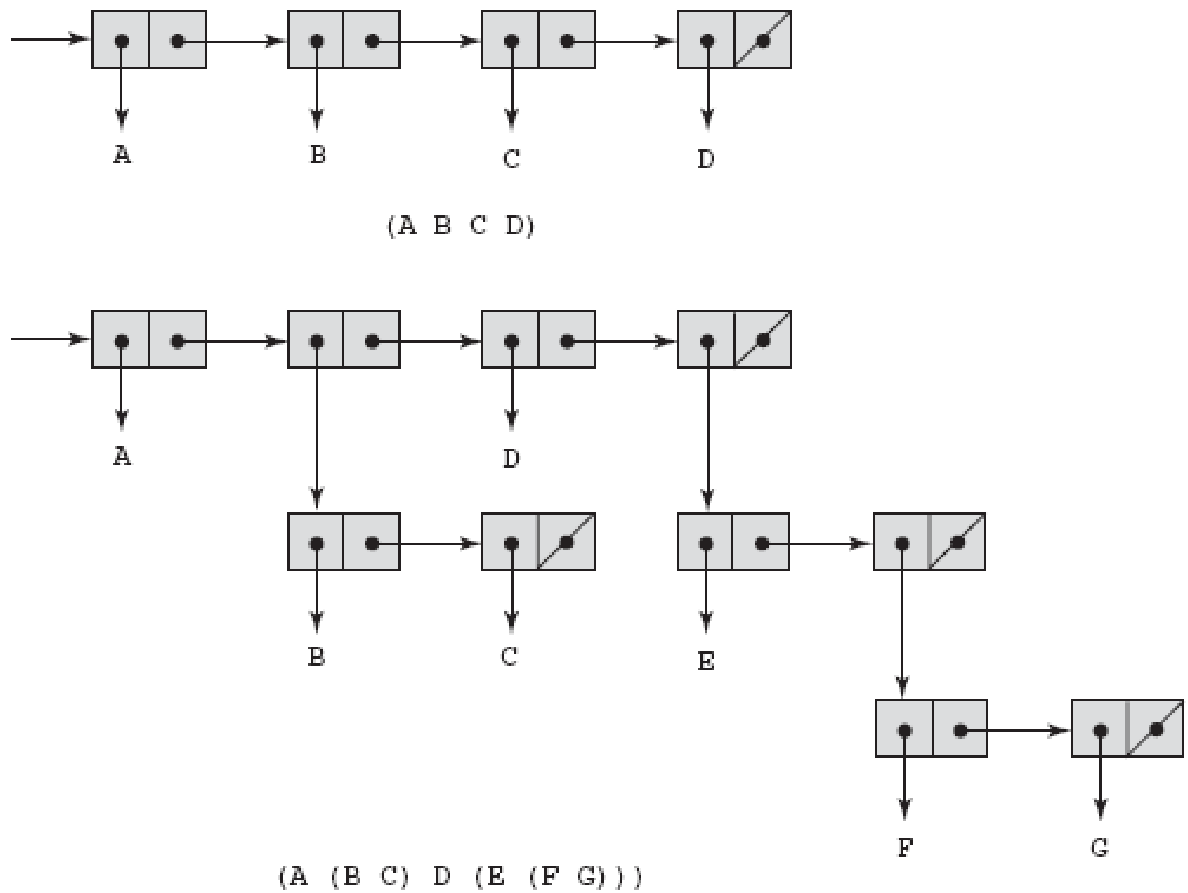
LISP 데이터 타입과 구조

- *데이터 객체 타입*: 원자, 리스트
- *리스트의 형식* : (서브리스트 and/or 원자)

  e.g., `(A B (C D) E)`
- 원래 LISP는 타입이 없는 언어였음
- 리스트는 내부적으로 단일 연결 리스트로 저장
- 리스트 원소들은 쌍으로 구성
  - 첫 부분: 원소의 데이터; 원자나 중첩 리스트에 대한 포인터
  - 두 번째 부분: 원자에 대한 포인터; 다른 원자에 대한 포인터 또는 널리스트

Figure 15.1

Internal representation of two LISP lists



Lisp Interpretation

- Lambda notation is used to specify functions and function definitions. Function applications and data have the same form.

  e.g., If the list `(A B C)` is interpreted as data it is

  a simple list of three atoms, `A`, `B`, and `C`

  If it is interpreted as a function application,

  it means that the function named `A` is

  applied to the two parameters, `B` and `C`

Scheme의 기원

- 1970년대 중반, LISP의 파생어; cleaner, more modern, and simpler version than the contemporary dialects of LISP
- 정적 영역 규칙 사용
- 함수는 일등급 개체로 취급
  – 표현식의 값, 리스트의 원소
  – 변수로 대입, 매개변수로 전달, 함수로부터 반환

Scheme 인터프리터

- In interactive mode, the Scheme interpreter is an infinite read-evaluate-print loop (REPL)
  - This form of interpreter is also used by Python and Ruby
- Expressions are interpreted by the function `EVAL`
- Literals evaluate to themselves

Primitive Function Evaluation

- Parameters are evaluated, in no particular order
- The values of the parameters are substituted into the function body
- The function body is evaluated
- The value of the last expression in the body is the value of the function

Primitive Functions & LAMBDA Expressions

- Primitive Arithmetic Functions : **+, -, *, /, ABS, SQRT, REMAINDER, MIN, MAX**
  e.g., `(+ 5 2)` yields `7`

- Lambda Expressions
  - Form is based on λ notation
    e.g., `(LAMBDA (x) (* x x))`
    `x` is called a bound variable

- Lambda expressions can be applied to parameters
  e.g., `((LAMBDA (x) (* x x)) 7)`

- `LAMBDA` expressions can have any number of parameters
  `(LAMBDA (a b x) (+ (* a x x) (* b x)))`

특수형 함수: DEFINE

- `DEFINE` – Two forms:
  1. To bind a symbol to an expression

     e.g., `(DEFINE pi 3.141593)`

     Example use: `(DEFINE two_pi (* 2 pi))`

     These symbols are not variables – they are like the names bound by Java's **`final`** declarations
  2. To bind names to lambda expressions (`LAMBDA` is implicit)

     e.g., `(DEFINE (square x) (* x x))`

     Example use: `(square 5)`

  – The evaluation process for `DEFINE` is different! The first parameter is never evaluated. The second parameter is evaluated and bound to the first parameter.

Output Functions

- `(DISPLAY expression)`
- `(NEWLINE)`

Output Functions

- Usually not needed, because the interpreter always displays the result of a function evaluated at the top level (not nested)
- Scheme has `PRINTF`, which is similar to the `printf` function of C
- Note: explicit input and output are not part of the pure functional programming model, because input operations change the state of the program and output operations are side effects

수치 술어 함수

- 술어 함수는 불리안 값을 반환하는 함수
- `#T` (or `#t`) is true and `#F` (or `#f`) is false (sometimes `()` is used for false)
- =, <>, >, <, >=, <=
- `EVEN?`, `ODD?`, `ZERO?`, `NEGATIVE?`

- The `NOT` function inverts the logic of a Boolean expression

제어 흐름

- 이중선택 구조 – IF

  (IF predicate then_exp else_exp)

  (Ex)
  ```
  (IF (<> count 0)
          (/ sum count)
      )
  ```

- 다중선택 구조 – COND function (8장에서 다룸)

  (Ex)
  ```
  DEFINE (leap? year)
      (COND
        ((ZERO? (MODULO year 400)) #T)
        ((ZERO? (MODULO year 100)) #F)
        (ELSE (ZERO? (MODULO year 4)))
      ))
  ```

리스트 함수

- QUOTE – takes one parameter; returns the parameter without evaluation
  - QUOTE is required because the Scheme interpreter, named EVAL, always evaluates parameters to function applications before applying the function. QUOTE is used to avoid parameter evaluation when it is not appropriate
  - QUOTE can be abbreviated with the apostrophe prefix operator

    '(A B) is equivalent to (QUOTE (A B))

- Recall that CAR, CDR, and CONS were covered in Chapter 6

리스트 함수(continued)

- Examples:

`(CAR '((A B) C D))` returns `(A B)`

`(CAR 'A)` is an error

`(CDR '((A B) C D))` returns `(C D)`

`(CDR 'A)` is an error

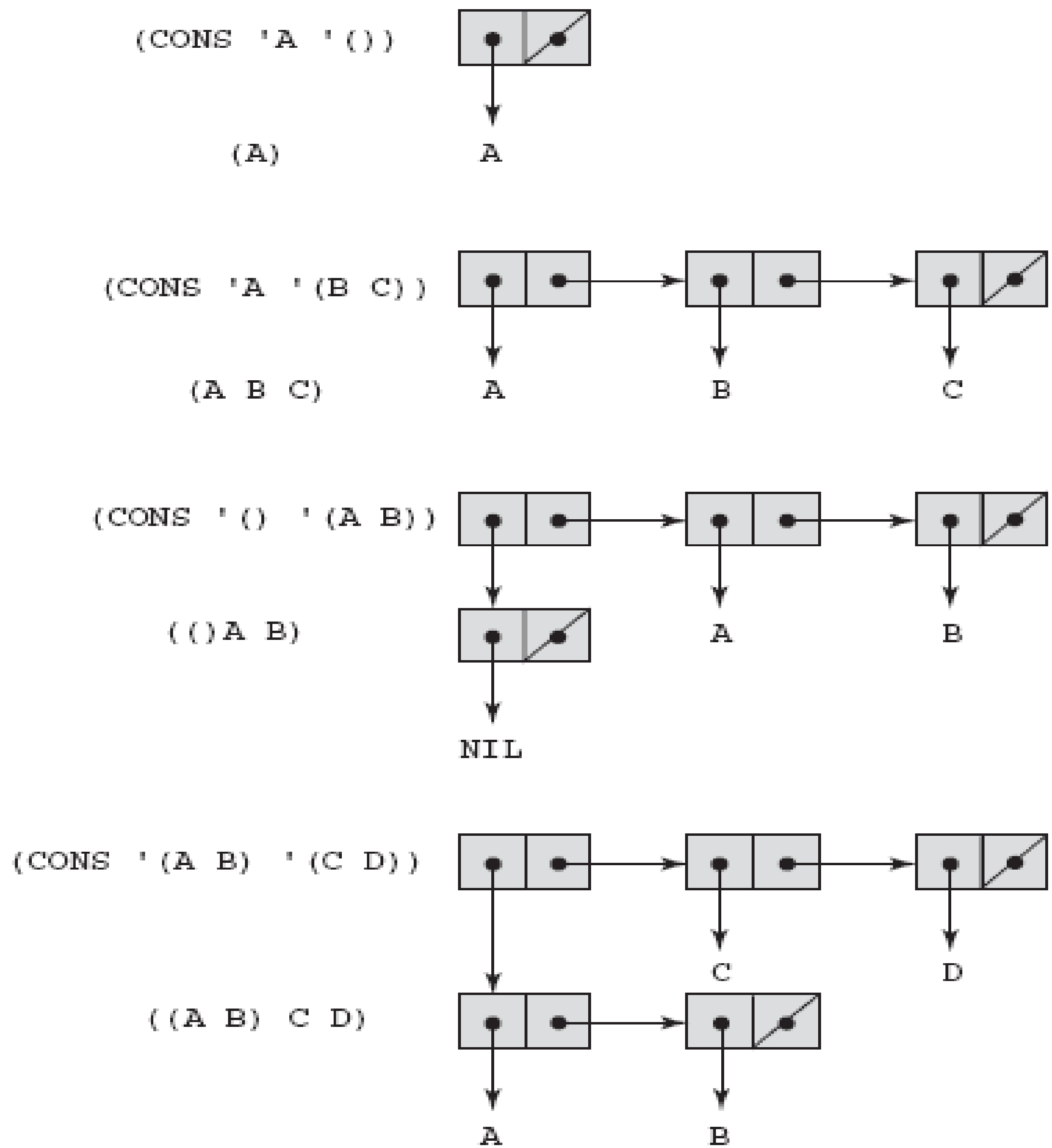
`(CDR '(A))` returns `()`

`(CONS '() '(A B))` returns `(() A B)`

`(CONS '(A B) '(C D))` returns `((A B) C D)`

Figure 15.2

The result of several CONS operations



리스트 함수 (continued)

- `LIST` is a function for building a list from any number of parameters

  `(LIST 'apple 'orange 'grape)` returns

  `(apple orange grape)`

술어 함수: EQ?

- EQ? takes two expressions as parameters (usually two atoms); it returns #T if both parameters have the same pointer value; otherwise #F

  (EQ? 'A 'A) yields #T

  (EQ? 'A 'B) yields #F

  (EQ? 'A '(A B)) yields #F

  (EQ? '(A B) '(A B)) yields #T or #F

  (EQ? 3.4 (+ 3 0.4))) yields #T or #F

술어 함수: `EQV?`

- `EQV?` is like `EQ?`, except that it works for both symbolic and numeric atoms; it is a value comparison, not a pointer comparison

  `(EQV? 3 3)` yields `#T`

  `(EQV? 'A 3)` yields `#F`

  `(EQV 3.4 (+ 3 0.4))` yields `#T`

  `(EQV? 3.0 3)` yields `#F` (floats and integers are different)

술어 함수: `LIST?` and `NULL?`

- `LIST?` takes one parameter; it returns `#T` if the parameter is a list; otherwise `#F`

  `(LIST? '())` yields `#T`

- `NULL?` takes one parameter; it returns `#T` if the parameter is the empty list; otherwise `#F`

  `(NULL? '(()))` yields `#F`

Example Scheme Function: `member`

- `member` takes an atom and a simple list; returns `#T` if the atom is in the list; `#F` otherwise

```
DEFINE (member atm a_list)
(COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    ((ELSE (member atm (CDR a_list)))
))
```

Example Scheme Function: `equalsimp`

- `equalsimp` takes two simple lists as parameters; returns `#T` if the two simple lists are equal; `#F` otherwise

```
(DEFINE (equalsimp list1 list2)
(COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp(CDR list1)(CDR list2)))
    (ELSE #F)
))
```

Example Scheme Function: `append`

- `append` takes two lists as parameters; returns the first parameter list with the elements of the second parameter list appended at the end

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1)
                (append (CDR list1) list2)))
))
```

Example Scheme Function: `LET`

- Recall that `LET` was discussed in Chapter 5
- `LET` is actually shorthand for a `LAMBDA` expression applied to a parameter

```
(LET ((alpha 7))(* 5 alpha))
```

is the same as:

```
((LAMBDA (alpha) (* 5 alpha)) 7)
```

LET Example

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
     (/ (SQRT (- (* b b) (* 4 a c)))(* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a))))
  (DISPLAY (+ minus_b_over_2a root_part_over_2a))
  (NEWLINE)
  (DISPLAY (- minus_b_over_2a root_part_over_2a))
))
```

Tail Recursion in Scheme

- Definition: A function is *tail recursive* if its recursive call is the last operation in the function
- Why the tail recursion is important?
  - tail recursions can be easily converted into loops
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- what is important is not the location of the call but the time of the call
- Scheme language definition requires that Scheme language systems convert all tail recursive functions to use iteration

Tail Recursion in Scheme – continued

- Example of rewriting a function to make it tail recursive, using helper a function

Original:

```
(DEFINE (factorial n)
   (IF (<= n 0)
      1
      (* n (factorial (- n 1)))
))
```

Tail recursive:

```
(DEFINE (facthelper n factpartial)
   (IF (<= n 0)
      factpartial
      facthelper((- n 1) (* n factpartial)))
))
(DEFINE (factorial n)
   (facthelper n 1))
```

Functional Sum

Imperative `sum`

```
int sum(int i, int j)
{ int k, temp;
  temp = 0;
  for(k = i; k <= j; k++)
    temp += k;
  return temp;
}
```

Functional `sum`

```
int sum(int i, int j)
{ if (i > j)
    return 0;
  else
    return i + sum(i+1, j);
}
```

Is this call tail recursive?

Though the recursive call is located at the end of the function body, this is not a tail recursion.

Accumulating Parameters

- Conversion to Tail-Recursion
  - Some recursive functions may be converted into tail-recursive ones using "accumulating parameters."
  - In this cases, helper functions may be needed.

```
int sum1(int i, int j, int sumSoFar)
{ if (i > j)
    return sumSoFar;
  else
    return sum1(i+1,j,sumSoFar+i);
};

int sum(int i, int j)
{ return sum1(i,j,0);
}
```

accumulating parameter

helper function (helping procedure)

This is a tail call !

범함수형 – 합성함수(Composition)

- Composition
  - If `h` is the composition of `f` and `g`, `h(x) = f(g(x))`

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE h x) (+ 2 (* 3 x))) (The composition)
```

  - In Scheme, the functional composition function `compose` can be written:

```
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
((compose CAR CDR) '((a b) c d)) yields c → CADR
((compose CDR CAR) '((a b) c d)) yields (b) → CDAR
  (DEFINE (third a_list)
    ((compose CAR (compose CDR CDR)) a_list))
```

  is equivalent to `CADDR`

모두-적용(Apply-to-All) 범함수 형태

- Apply to All – one form in Scheme is `map`
  - Applies the given function to all elements of the given list;

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list))
                (map fun (CDR a_list))))
))
```

`(map (LAMBDA (num) (* num num num)) '(3 4 2 6))` yields `(27 64 8 216)`

COMMON LISP

- A combination of many of the features of the popular dialects of LISP around in the early 1980s
- A large and complex language--the opposite of Scheme
- Features include:
  - records
  - arrays
  - complex numbers
  - character strings
  - powerful I/O capabilities
  - packages with access control
  - iterative control statements

ML

- A static-scoped functional language with syntax that is closer to Pascal than to Lisp
- Uses type declarations, but also does *type inferencing* to determine the types of undeclared variables
- It is strongly typed (whereas Scheme is essentially typeless) and has no type coercions
- Does not have imperative-style variables
- Its identifiers are untyped names for values
- Includes exception handling and a module facility for implementing abstract data types
- Includes lists and list operations

Haskell

- Similar to ML (syntax, static scoped, strongly typed, type inferencing)
- Different from ML (and most other functional languages) in that it is *purely* functional (e.g., no variables, no assignment statements, and no side effects of any kind)
- Most Important Features
  - Uses *lazy evaluation* (evaluate no subexpression until the value is needed)
  - Has *list comprehensions*, which allow it to deal with infinite lists

F#

- Based on Ocaml, which is a descendant of ML and Haskell
- Fundamentally a functional language, but with imperative features and supports OOP
- Has a full-featured IDE, an extensive library of utilities, and interoperates with other .NET languages
- Includes tuples, lists, discriminated unions, records, and both mutable and immutable arrays
- Supports generic sequences, whose values can be created with generators and through iteration

F# (continued)

- Why F# is Interesting:
  - It builds on previous functional languages
  - It supports virtually all programming methodologies in widespread use today
  - It is the first functional language that is designed for interoperability with other widely used languages
  - At its release, it had an elaborate and well-developed IDE and library of utility software

Support for Functional Programming in Primarily Imperative Languages

- Support for functional programming is increasingly creeping into imperative languages
  - Anonymous functions (lambda expressions)
    - JavaScript: leave the name out of a function definition
    - C#: `i => (i % 2) == 0` (returns true or false depending on whether the parameter is even or odd)
    - Python: **`lambda`** `a, b : 2 * a - b`

Support for Functional Programming in Primarily Imperative Languages (continued)

- Python supports the higher-order functions filter and map (often use lambda expressions as their first parameters)

```
map(lambda x : x ** 3, [2, 4, 6, 8])
```

Returns `[8, 64, 216, 512]`

- Python supports partial function applications

```
from operator import add
add5 = partial (add, 5)
```

(the first line imports add as a function)

Use: `add5(15)`

Support for Functional Programming in Primarily Imperative Languages (continued)

- Ruby Blocks
  - Are effectively subprograms that are sent to methods, which makes the method a higher-order subprogram
  - A block can be converted to a subprogram object with **`lambda`**

    ```
    times = lambda {|a, b| a * b}
    ```

    Use: `x = times.(3, 4)` (sets `x` to `12`)
  - Times can be curried with

    ```
    times5 = times.curry.(5)
    ```

    Use: `x5 = times5.(3)` (sets `x5` to `15`)

Comparing Functional and Imperative Languages

- Imperative Languages:
  - Efficient execution
  - Complex semantics
  - Complex syntax
  - Concurrency is programmer designed
- Functional Languages:
  - Simple semantics
  - Simple syntax
  - Less efficient execution
  - Programs can automatically be made concurrent

Summary

- Functional programming languages use function application, conditional expressions, recursion, and functional forms to control program execution
- Lisp began as a purely functional language and later included imperative features
- Scheme is a relatively simple dialect of Lisp that uses static scoping exclusively
- Common Lisp is a large Lisp-based language
- ML is a static-scoped and strongly typed functional language that uses type inference
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- F# is a .NET functional language that also supports imperative and object-oriented programming
- Some primarily imperative languages now incorporate some support for functional programming
- Purely functional languages have advantages over imperative alternatives, but still are not very widely used