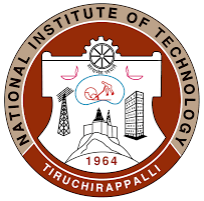


CSPC31: Principles of Programming Languages

Dr. R. Bala Krishnan
Asst. Prof.
Dept. of CSE
NIT, Trichy – 620 015
Ph: 999 470 4853
E-Mail: balakrishnan@nitt.edu

Books

- **Text Books**
  - ✓ **Robert W. Sebesta, *"Concepts of Programming Languages"*, Tenth Edition, Addison Wesley, 2012.**
  - ✓ Michael L. Scott, *"Programming Language Pragmatics"*, Third Edition, Morgan Kaufmann, 2009.
- **Reference Books**
  - ✓ Allen B Tucker, and Robert E Noonan, *"Programming Languages – Principles and Paradigms"*, Second Edition, Tata McGraw Hill, 2007.
  - ✓ R. Kent Dybvig, *"The Scheme Programming Language"*, Fourth Edition, MIT Press, 2009.
  - ✓ Jeffrey D. Ullman, *"Elements of ML Programming"*, Second Edition, Prentice Hall, 1998.
  - ✓ Richard A. O'Keefe, *"The Craft of Prolog"*, MIT Press, 2009.
  - ✓ W. F. Clocksin, C. S. Mellish, *"Programming in Prolog: Using the ISO Standard"*, Fifth Edition, Springer, 2003.

Chapters

Chapter No.	Title
1.	Preliminaries
~~2.~~	~~Evolution of the Major Programming Languages~~
3.	Describing Syntax and Semantics
4.	Lexical and Syntax Analysis
~~5.~~	~~Names, Binding, Type Checking and Scopes~~
6.	Data Types
7.	Expressions and Assignment Statements
8.	Statement-Level Control Structures
9.	Subprograms
10.	Implementing Subprograms
~~11.~~	~~Abstract Data Types and Encapsulation Constructs~~
12.	Support for Object-Oriented Programming
13.	Concurrency
14.	Exception Handling and Event Handling
15.	Functional Programming Languages
16.	Logic Programming Languages

Chapter 15 – Functional Programming Languages

Introduction

- Imperative languages are designed for von Neumann architecture
- Functional programming paradigm, which is based on mathematical functions, is the design basis of the most important nonimperative styles of languages
- Purely functional programs are easier to understand, both during and after development, largely because the meanings of expressions are independent of their context (one characterizing feature of a pure functional programming language is that neither expressions nor functions have side effects)
- One of the fundamental characteristics of programs written in imperative languages is that they have state, which changes throughout the execution process
    - This state is represented by the program's variables
    - The author and all readers of the program must understand the uses of its variables and how the program's state changes through execution
    - For a large program, this is a daunting task
- This is one problem with programs written in an imperative language that is not present in a program written in a pure functional language, for such programs have neither variables nor state

Introduction

- LISP began as a pure functional language but soon acquired some important imperative features that increased its execution efficiency
  - It is still the most important of the functional languages, at least in the sense that it is the only one that has achieved widespread use
  - It dominates in the areas of knowledge representation, machine learning, intelligent training systems, and the modeling of speech
- Scheme is a small, static-scoped dialect of LISP
- Scheme has been widely used to teach functional programming
- It is also used in some universities to teach introductory programming courses
- Functional programming languages are now being used in areas such as database processing, financial modeling, statistical analysis, and bio-informatics

Mathematical Functions

- A mathematical function is a mapping of members of one set, called the domain set, to another set, called the range set
- A function definition specifies the domain and range sets, either explicitly or implicitly, along with the mapping
- The mapping is described by an expression or, in some cases, by a table
- Functions are often applied to a particular element of the domain set, given as a parameter to the function
- Note that the domain set may be the cross product of several sets (reflecting that there can be more than one parameter)
- A function yields an element of the range set

Fundamental Characterisitics of Mathematical Functions

Sl. No.	Mathematical Functions	Imperative Programming Languages
1.	Evaluation order of their mapping expressions is controlled by recursion and conditional expressions	Evaluation order is controlled by the sequencing and iterative repetition
2.	As they have no side effects and cannot depend on any external values, they always map a particular element of the domain to the same element of the range	A subprogram may depend on the current values of several nonlocal or global variables. This makes it difficult to determine statically what values the subprogram will produce and what side effects it will have on a particular execution
3.	In mathematics, there is no such thing as a variable that models a memory location. Hence, there is no concept of the state of a function. A mathematical function maps its parameter(s) to a value (or values), rather than specifying a sequence of operations on values in memory to produce a value	Local variables in functions maintain the state of the function. Computation is accomplished by evaluating expressions in assignment statements that change the state of the program

Simple Functions

- Function definitions are often written as a function name, followed by a list of parameters in parentheses, followed by the mapping expression
- For example, cube(x) = x * x * x, where x is a **real number**
- In this definition, the **domain and range sets are the real numbers**
- The symbol = is used to mean "is defined as"
- The parameter x can represent any member of the domain set, but it is fixed to represent one specific element during evaluation of the function expression
- This is one way the parameters of mathematical functions differ from the variables in imperative languages
- Function applications are specified by pairing the function name with a particular element of the domain set
- The range element is obtained by evaluating the function-mapping expression with the domain element substituted for the occurrences of the parameter
- Once again, it is important to note that during evaluation, the mapping of a function contains no unbound parameters, where a bound parameter is a name for a particular value

Simple Functions

- Every occurrence of a parameter is bound to a value from the domain set and is a constant during evaluation
- For example, consider the following evaluation of cube(x):

  cube (**2.0**) = 2.0 * 2.0 * 2.0 = 8

- The parameter x is bound to 2.0 during the evaluation and there are no unbound parameters
- Furthermore, x is a constant (its value cannot be changed) during the evaluation
- Early theoretical work on functions separated the task of defining a function from that of naming the function
- Lambda notation provides a method for defining nameless functions
- A lambda expression specifies the parameters and the mapping of a function
- **The lambda expression is the function itself, which is nameless**
- For example, consider the following lambda expression:

  λ(x)x * x * x

Simple Functions

- As stated earlier, before evaluation a parameter represents any member of the domain set, but during evaluation it is bound to a particular member
- When a lambda expression is evaluated for a given parameter, the expression is said to be applied to that parameter
- The mechanics of such an application are the same as for any function evaluation
- Application of the example lambda expression is denoted as in the following example:

$$\lambda((x)x * x * x)(2)$$

  which results in the value 8

- Lambda expressions, like other function definitions, can have more than one parameter

Functional Forms

- A higher-order function, or functional form, is one that either takes one or more functions as parameters or yields a function as its result, or both
- One common kind of functional form is function composition, which has two functional parameters and yields a function whose value is the first actual parameter function applied to the result of the second
- Function composition is written as an expression, using ° as an operator, as in

$$h \equiv f \circ g$$

For example, if

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

then h is defined as

$$h(x) \equiv f(g(x)), \text{ or } h(x) \equiv (3 * x) + 2$$

Functional Forms

- Apply-to-all is a functional form that takes a single function as a parameter
- If applied to a list of arguments, apply-to-all applies its functional parameter to each of the values in the list argument and collects the results in a list or sequence
- Apply-to-all is denoted by α

Let
$h(x) \equiv x * x$

then

$\alpha(h, (2, 3, 4))$ yields $(4, 9, 16)$

Fundamentals of Functional Programming Languages

- Objective of the design of a functional programming language is to mimic mathematical functions to the greatest extent possible
- This results in an approach to problem solving that is fundamentally different from approaches used with imperative languages
- In an imperative language, an expression is evaluated and the result is stored in a memory location, which is represented as a variable in a program
- This is the purpose of assignment statements

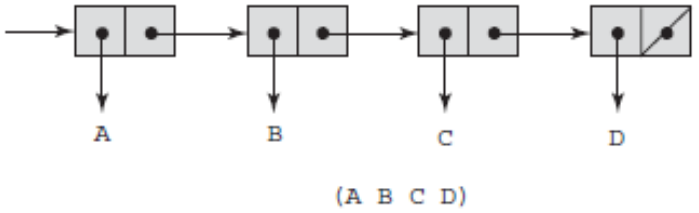
(x + y) / (a - b)

- A purely functional programming language does not use variables or assignment statements, thus freeing the programmer from concerns related to the memory cells, or state, of the program
- Without variables, iterative constructs are not possible, for they are controlled by variables
- Repetition must be specified with recursion rather than with iteration
- Without variables, the execution of a purely functional program has no state in the sense of operational and denotational semantics
- The execution of a function always produces the same result when given the same parameters -> Referential Transparency

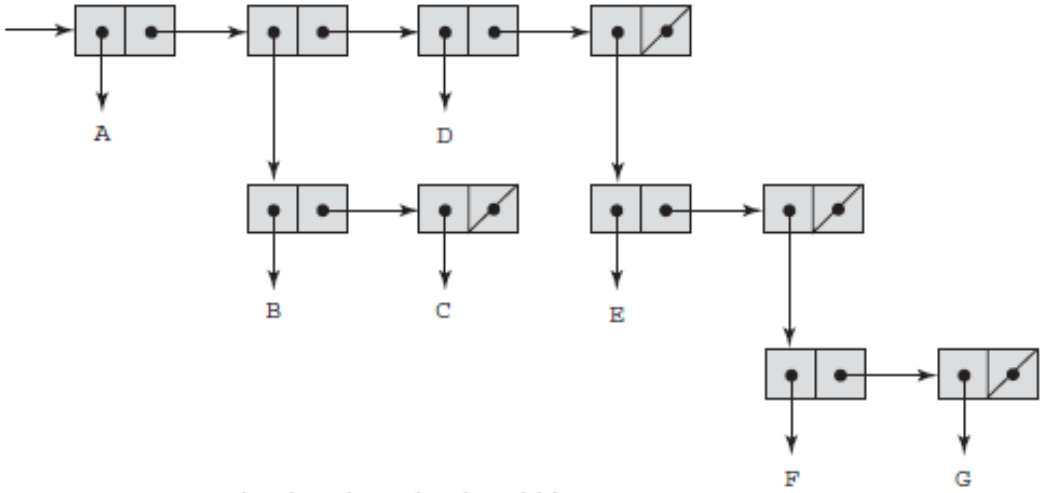
First Functional Programming Language: LISP

- Lists are specified in LISP by delimiting their elements with parentheses
- The elements of simple lists are restricted to atoms, as in: (A B C D)
- Nested list structures are also specified by parentheses. For example, the list: (A (B C) D (E (F G)))

The first is the atom A; the second is the sublist (B C); the third is the atom D; the fourth is the sublist (E (F G)), which has as its second element the sublist (F G).
Internally, a list is usually stored as linked list structure in which each node has two pointers, one to reference the data of the node and the other to form the linked list. A list is referenced by a pointer to its first element.
The last element of a list has no successor, so its link is nil



(A B C D)



(A (B C) D (E (F G)))

Introduction

- Function calls were specified in a prefix list form originally called Cambridge Polish, as in the following:

  (function_name argument$_1$ …… argument$_n$)

- For example, if **+** is a function that takes two or more numeric parameters, then the following two expressions evaluate to 12 and 20, respectively:

  (+ 5 7)

  (+ 3 4 7 6)

  (function_name (LAMBDA (arg1 … argn) expression))

- Nameless functions are sometimes useful in functional programming
- For example, consider a function whose action is to produce a function for immediate application to a parameter list
- The produced function has no need for a name, for it is applied only at the point of its construction
- Another feature of early LISP systems that was apparently accidental was the use of dynamic scoping
- Functions were evaluated in the environments of their callers
- No one at the time knew much about scoping, and there may have been little thought given to the choice
- Dynamic scoping was used for most dialects of LISP before 1975

Introduction to Scheme

- It is characterized by its small size, its exclusive use of static scoping, and its treatment of functions as first-class entities
- As first-class entities, Scheme functions can be the values of expressions, elements of lists, passed as parameters, and returned from functions
- Early versions of LISP did not provide all of these capabilities
- Scheme includes primitive functions for the basic arithmetic operations
- These are +, −, *, and /, for add, subtract, multiply, and divide
- * and + can have zero or more parameters
- If * is given no parameters, it returns 1
- if + is given no parameters, it returns 0
- + adds all of its parameters together
- * multiplies all its parameters together
- / and − can have two or more parameters
- In the case of subtraction, all but the first parameter are subtracted from the first
- Division is similar to subtraction

Expression	*Value*
42	42
(* 3 7)	21
(+ 5 7 8)	20
(- 5 6)	-1
(- 15 7 2)	6
(- 24 (* 4 3))	12

```
(display (* 3 7))
```
Output: 21

https://www.jdoodle.com/execute-scheme-online/

Introduction to Scheme

- There are a large number of other numeric functions in Scheme, among them MODULO, ROUND, MAX, MIN, LOG, SIN, and SQRT
- SQRT returns the square root of its numeric parameter, if the parameter's value is not negative
- If the parameter is negative, SQRT yields a complex number
- In Scheme, note that we use uppercase letters for all reserved words and predefined functions
- The official definition of the language specifies that there is no distinction between uppercase and lowercase in these
- However, some implementations, for example DrRacket's teaching languages, require lowercase for reserved words and predefined functions
- If a function has a fixed number of parameters, such as SQRT, the number of parameters in the call must match that number
- If not, the interpreter will produce an error message

Defining Functions

- A Scheme program is a collection of function definitions
- Consequently, knowing how to define these functions is a prerequisite to writing the simplest program
- In Scheme, a nameless function actually includes the word LAMBDA, and is called a lambda expression

(LAMBDA (x) (* x x))

is a nameless function that returns the square of its given numeric parameter

- This function can be applied in the same way that named functions are: by placing it in the beginning of a list that contains the actual parameters
- For example, the following expression yields 49:

((LAMBDA (**x**) (* x x)) 7)

```
(display ((lambda (x) (* x x)) 7))
Output: 49
```

- In this expression, x is called a bound variable within the lambda expression
- A bound variable never changes in the expression after being bound to an actual parameter value at the time evaluation of the lambda expression begins
- Lambda expressions can have any number of parameters
- For example, we could have the following:

(LAMBDA (**a b c x**) (+ (* a x x) (* b x) c))

Introduction to Scheme

- Scheme special form function DEFINE serves two fundamental needs of Scheme programming:
  - To bind a name to a value
  - To bind a name to a lambda expression
- The form of DEFINE that binds a name to a value may make it appear that DEFINE can be used to create imperative language – style variables
- However, these name bindings create named values, not variables

```
(define pi 3.14159)
(define two_pi (* 2 pi))
(display pi)
(newline)
(display two_pi)
```

Output:
3.14159
6.28318

- DEFINE is analogous to a declaration of a named constant in an imperative language

```
final float PI = 3.14159;
final float TWO_PI = 2.0 * PI;
```

https://www.jdoodle.com/execute-scheme-online/

Introduction to Scheme

- Second use of the DEFINE function is to bind a lambda expression to a name
- In this case, the lambda expression is abbreviated by removing the word LAMBDA
- To bind a name to a lambda expression, DEFINE takes two lists as parameters
- The first parameter is the prototype of a function call, with the function name followed by the formal parameters, together in a list
- The second list contains an expression to which the name is to be bound
- The general form of such a DEFINE is:

```
(DEFINE (function_name parameters)
(expression)
)
```

```
(define (square number) (* number number))
(display (square 5))
Output: 25
```

Introduction to Scheme

```
(define (sum a b) (display "x + y = ") (display (+ a b)))
(sum 10 25)
(newline)
```

(or)

```
(define (sum a b)
(display "x + y = ")
(display (+ a b)))
(sum 10 25)
(newline)
```

Output:
x + y = 35

https://www.jdoodle.com/execute-scheme-online/

Introduction to Scheme

- To illustrate the difference between primitive functions and the DEFINE special form, consider the following:

(DEFINE x 10)

- If DEFINE were a primitive function, EVAL's first action on this expression would be to evaluate the two parameters of DEFINE
- If x were not already bound to a value, this would be an error
- Furthermore, if x were already defined, it would also be an error, because this DEFINE would attempt to redefine x, which is illegal
- Remember, x is the name of a value; it is not a variable in the imperative sense
- Following is another example of a function
- It computes the length of the hypotenuse (the longest side) of a right triangle, given the lengths of the two other sides

```
(DEFINE (hypotenuse side1 side2)
(SQRT(+(square side1)(square side2)))
)
```

```
(define (square number) (* number number))
```

- Notice that hypotenuse uses square, which was defined previously

Output Functions

- Scheme includes a formatted output function, PRINTF, which is similar to the printf function of C
- Numeric Predicate Functions:
  - A predicate function is one that returns a Boolean value (some representation of either true or false)
  - Scheme includes a collection of predicate functions for numeric data

Notice that the names for all predefined predicate functions that have words for names end with question marks. In Scheme, the two Boolean values are #T and #F (or #t and #f), although some implementations use the empty list for false.5The Scheme predefined predicate functions return the empty list, (), for false. When a list is interpreted as a Boolean, any nonempty list evaluates to true; the empty list evaluates to false. This is similar to the interpretation of integers in C as Boolean values; zero evaluates to false and any nonzero value evaluates to true. In the interest of readability, all of our example predicate functions in this chapter return #F, rather than (). The NOT function is used to invert the logic of a Boolean expression.

Function	*Meaning*
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
EVEN?	Is it an even number?
ODD?	Is it an odd number?
ZERO?	Is it zero?

(display (even? 6))
Output: #t

Output Functions

- Scheme uses three different constructs for control flow:
  - One similar to the selection construct of the imperative languages
  - Two based on the evaluation control used in mathematical functions
- Scheme two-way selector function, named IF, has three parameters: a predicate expression, a then expression, and an else expression
- A call to IF has the form:

(IF predicate then_expression else_expression)

```
(DEFINE (factorial n)
    (IF (<= n 1)
        1
        (* n (factorial (– n 1)))
))
```

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

- Third Scheme control mechanism is recursion, which is used, as in mathematics, to specify repetition

List Functions

- Suppose we have a function that has two parameters, an atom and a list, and the purpose of the function is to determine whether the given atom is in the given list
- Neither the atom nor the list should be evaluated; they are literal data to be examined
- To avoid evaluating a parameter, it is first given as a parameter to the primitive function QUOTE, which simply returns it without change

(QUOTE A) **returns A**
(QUOTE (A B C)) **returns (A B C)**
(CAR '(A B C)) **returns A**
(CAR '((A B) C D)) **returns (A B)**
(CAR 'A) **is an error because A is not a list**
(CAR '(A)) **returns A**
(CAR '()) **is an error**
(CDR '(A B C)) **returns (B C)**
(CDR '((A B) C D)) **returns (C D)**
(CDR 'A) **is an error**
(CDR '(A)) **returns ()**
(CDR '()) **is an error**

```
(define (second a_list) (car (cdr a_list)))
(display (second '(A B C)))
```
Output: B

List Functions

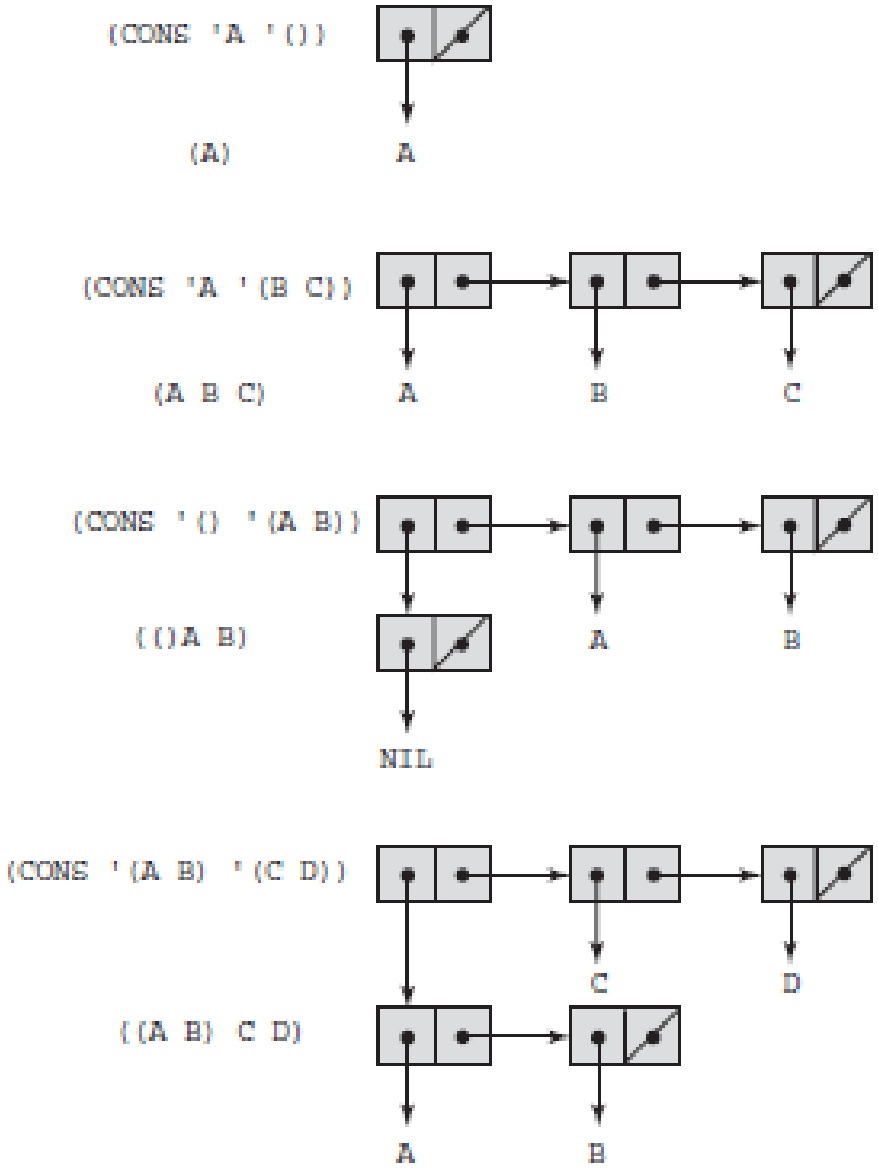
- Some of the most commonly used functional compositions in Scheme are built in as single functions
- For example:
- (**CA**AR x) is equivalent to (**CAR**(CAR x))
- (**CA**DR x) is equivalent to (**CAR** (CDR x))
- (CADDAR x) is equivalent to (CAR (CDR (CDR (CAR x))))
- Any combination of A's and D's, up to four, are legal between the 'C' and the 'R' in the function's name

```
(CADDAR '((A B (C) D) E)) =
(CAR (CDR (CDR (CAR '((A B (C) D) E))))) =
(CAR (CDR (CDR '(A B (C) D)))) =
(CAR (CDR '(B (C) D))) =
(CAR '((C) D)) =
(C)
```

List Functions

- Following are example calls to CONS:
  (CONS 'A '()) returns (A)
  (CONS 'A '(B C)) returns (A B C)
  (CONS '() '(A B)) returns (() A B)
  (CONS '(A B) '(C D)) returns ((A B) C D)
- Note that CONS is, in a sense, the inverse of CAR and CDR
- CAR and CDR take a list apart, and CONS constructs a new list from given list parts
- The two parameters to CONS become the CAR and CDR of the new list
- Thus, if a_list is a list, then (CONS (CAR a_list) (CDR a_list)) returns a list with the same structure and same elements as a_list
- If the result of (CONS 'A 'B) is displayed, it would appear as (A . B)
  - This dotted pair indicates that instead of an atom and a pointer or a pointer and a pointer, this cell has two atoms
- LIST is a function that constructs a list from a variable number of parameters
- It is a shorthand version of nested CONS functions, as illustrated in the following: (LIST 'apple 'orange 'grape) **returns (apple orange grape)**
- Using CONS, the call to LIST above is written as follows:
  (CONS 'apple (CONS 'orange (CONS 'grape '())))

List Functions



Predicate Functions for Symbolic Atoms and Lists

- Scheme has three fundamental predicate functions, EQ?, NULL?, and LIST?, for symbolic atoms and lists
- The EQ? function takes two expressions as parameters, although it is usually used with two symbolic atom parameters
- It returns #T if both parameters have the same pointer value—that is, they point to the same atom or list; otherwise, it returns #F
- If the two parameters are symbolic atoms, EQ? returns #T if they are the same symbols; otherwise #F

```
(EQ? 'A 'A) returns #T
(EQ? 'A 'B) returns #F
(EQ? 'A '(A B)) returns #F
(EQ? '(A B) '(A B)) returns #F or #T
(EQ? 3.4 (+ 3 0.4)) returns #F or #T
```

- As the fourth example indicates, the result of comparing lists with EQ? is not consistent
- The reason for this is that two lists that are exactly the same often are not duplicated in memory
- At the time the Scheme system creates a list, it checks to see whether there is already such a list
- If there is, the new list is nothing more than a pointer to the existing list. In these cases, the two lists will be judged equal by EQ?
- However, in some cases, it may be difficult to detect the presence of an identical list, in which case a new list is created -> In this scenario, EQ? yields #F

Predicate Functions for Symbolic Atoms and Lists

- As we have seen, EQ? works for symbolic atoms but does not necessarily work for numeric atoms
- The = predicate works for numeric atoms but not symbolic atoms
- As discussed previously, EQ? also does not work reliably for list parameters
- Sometimes it is convenient to be able to test two atoms for equality when it is not known whether they are symbolic or numeric
- For this purpose, Scheme has a different predicate, EQV?, which works on both numeric and symbolic atoms

```
(EQV? 'A 'A) returns #T
(EQV? 'A 'B) returns #F
(EQV? 3 3) returns #T
(EQV? 'A 3) returns #F
(EQV? 3.4 (+ 3 0.4)) returns #T
(EQV? 3.0 3) returns #F
```

- Notice that the last example demonstrates that floating-point values are different from integer values
- EQV? is not a pointer comparison, it is a value comparison
- primary reason to use EQ? or = rather than EQV? when it is possible is that EQ? and = are faster than EQV?

Predicate Functions for Symbolic Atoms and Lists

- LIST? predicate function returns #T if its single argument is a list and #F otherwise, as in the following examples:

```
(LIST? '(X Y)) returns #T
(LIST? 'X) returns #F
(LIST? '()) returns #T
```

- The NULL? function tests its parameter to determine whether it is the empty list and returns #T if it is
- Consider the following examples:

```
(NULL? '(A B)) returns #F
(NULL? '()) returns #T
(NULL? 'A) returns #F
(NULL? '(())) returns #F
```

- The last call yields #F because the parameter is not the empty list
- Rather, it is a list containing a single element, the empty list

Example Scheme Functions

- Consider the problem of membership of a given atom in a given list that does not include sublists -> Such a list is called a simple list
- There are three cases that must be handled in the function:
  - An empty input list
  - A match between the atom and the CAR of the list
  - A mismatch between the atom and the CAR of the list, which causes the recursive call
- These three are the three parameters to COND, with the last being the default case that is triggered by an ELSE predicate
- The complete function follows:
- If the function is named member, it could be used as follows:

(**member** 'B '(A B C)) **returns #T**

(**member** 'B '(A C D E)) **returns #F**

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

Example Scheme Functions

- As another example, consider the problem of determining whether two given lists are equal
- If the two lists are simple, the solution is relatively easy, although some programming techniques with which the reader may not be familiar are involved
- A predicate function, equalsimp, for comparing simple lists is shown here:

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

(display (equalsimp ‘(a b c) ‘(a b d)))

- (append '(A B) '(C D R)) returns (A B C D R)
- (append '((A B) C) '(D (E F))) returns ((A B) C D (E F))

Example Scheme Functions

- Another commonly needed list operation is that of constructing a new list that contains all of the elements of two given list arguments
- This is usually implemented as a Scheme function named append

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

(**append** '(A B) '(C D R)) **returns (A B C D R)**

(**append** '((A B) C) '(D (E F))) **returns ((A B) C D (E F))**

Example Scheme Functions

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
             (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2))
))

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

Example Scheme Functions

```
(DEFINE (guess list1 list2)
  (COND
    ((NULL? list1) '())
    ((member (CAR list1) list2)
             (CONS (CAR list1) (guess (CDR list1) list2)))
    (ELSE (guess (CDR list1) list2))
))

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

- guess yields a simple list that contains the common elements of its two parameter lists
- So, if the parameter lists represent sets, guess computes a list that represents the intersection of those two sets

LET

- LET is a function that creates a local scope in which names are temporarily bound to the values of expressions
- It is often used to factor out the common subexpressions from more complicated expressions
- These names can then be used in the evaluation of another expression, but they cannot be rebound to new values in LET
- The mathematical definitions of the real (as opposed to complex) roots of the quadratic equation ax2 + bx + c are as follows:

  root1 = (**-b + sqrt(b^2 - 4ac))/2a** and root2 = (**-b - sqrt(b^2 - 4ac))/2a**

```
(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)))
        )
  (LIST (+ minus_b_over_2a root_part_over_2a)
            (- minus_b_over_2a root_part_over_2a))
```

- Because the names bound in the first part of a LET construct cannot be changed in the following expression, they are not the same as local variables in a block in an imperative language

LET

- LET is actually shorthand for a LAMBDA expression applied to a parameter
- The following two expressions are equivalent:

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

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

- In the first expression, 7 is bound to alpha with LET
- In the second, 7 is bound to alpha through the parameter of the LAMBDA expression

Tail Recursion in Scheme

- A function is tail recursive if its recursive call is the last operation in the function
- This means that the return value of the recursive call is the return value of the nonrecursive call to the function

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

- This function can be automatically converted by a compiler to use iteration, resulting in faster execution than in its recursive form
- However, many functions that use recursion for repetition are not tail recursive
- Programmers who were concerned with efficiency have discovered ways to rewrite some of these functions so that they are tail recursive
- One example of this uses an accumulating parameter and a helper function

Tail Recursion in Scheme

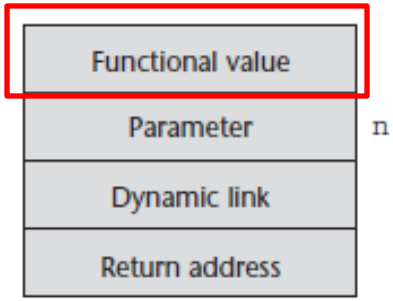
```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    (* n (factorial (- n 1))))
))
```

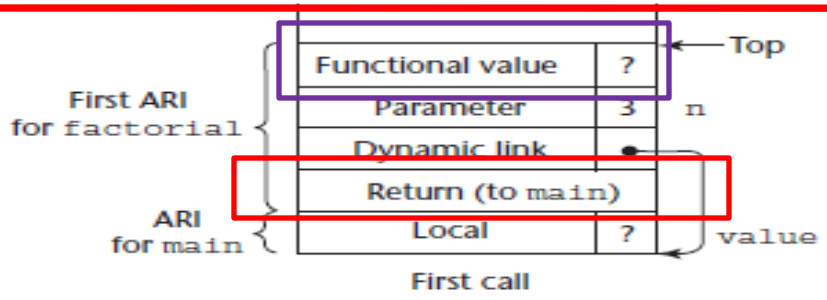
- The last operation of this function is the multiplication
- The function works by creating the list of numbers to be multiplied together and then doing the multiplications as the recursion unwinds to produce the result
- Each of these numbers is created by an activation of the function and each is stored in an activation record instance
- As the recursion unwinds the numbers are multiplied together
- This factorial function can be rewritten with an auxiliary helper function, which uses a parameter to accumulate the partial factorial
- The helper function, which is tail recursive, also takes factorial's parameter

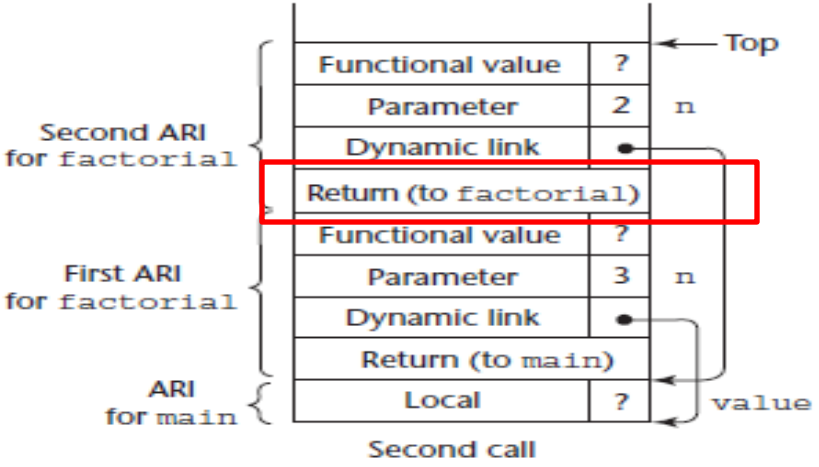
Miscellaneous

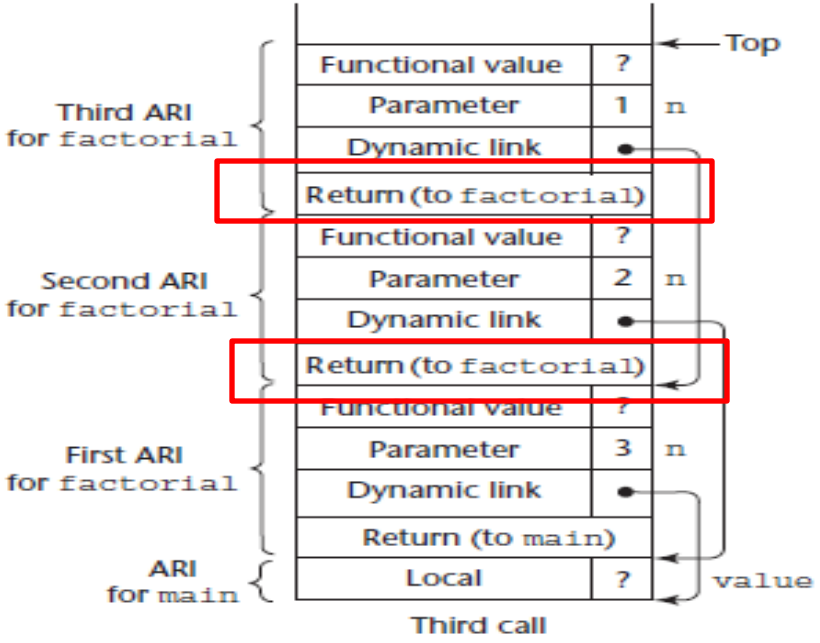
With Recursion

```
int factorial(int n) {
    <-------------- 1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <-------------- 2
 }
void main() {
  int value;
  value = factorial(3);
    <-------------- 3
 }
```

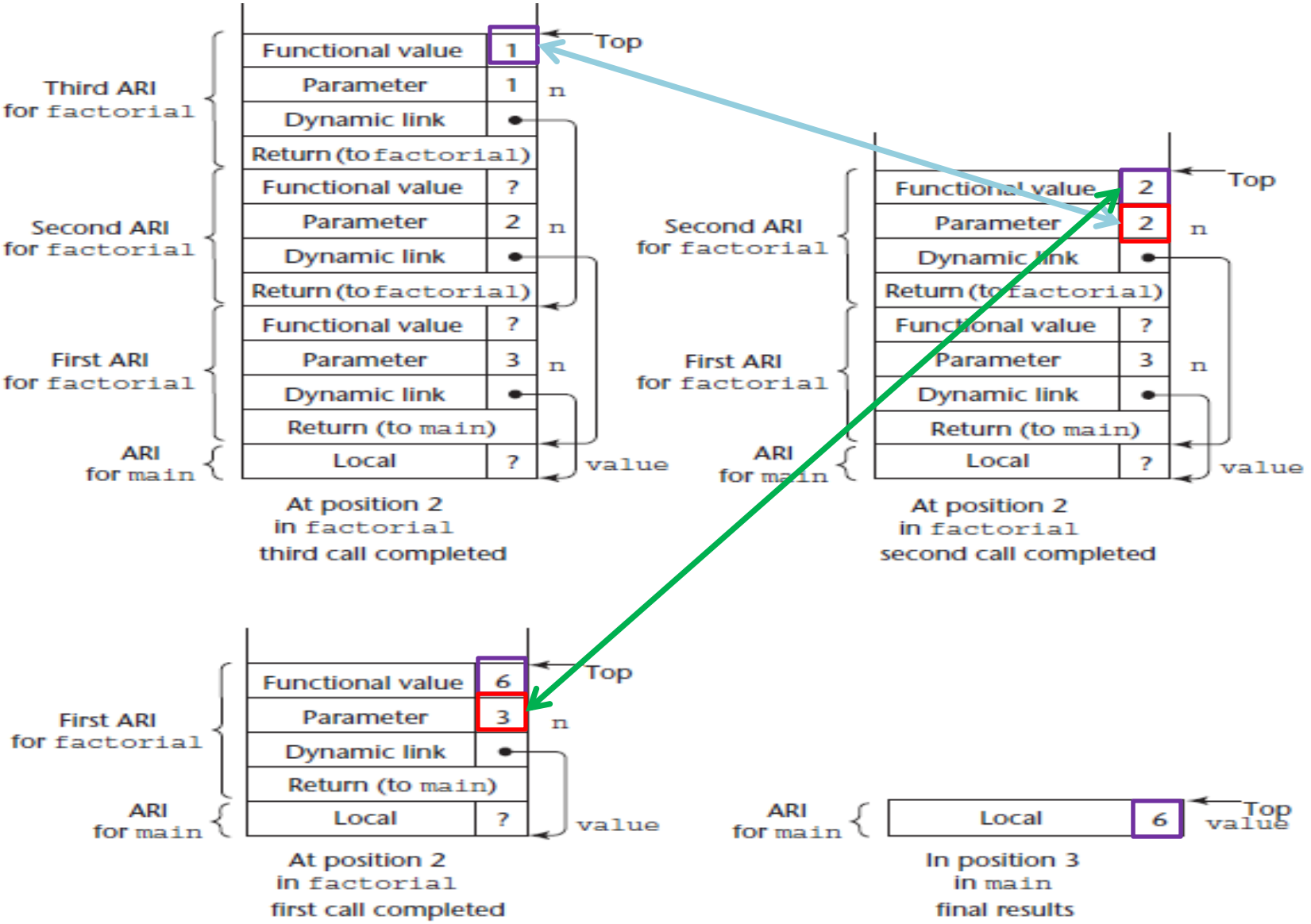








Miscellaneous



Tail Recursion in Scheme

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

- With these functions, the result is computed during the recursive calls, rather than as the recursion unwinds
- Because there is nothing useful in the activation record instances, they are not necessary
- Regardless of how many recursive calls are requested, only one activation record instance is necessary
- This makes the tail-recursive version far more efficient than the non–tail-recursive version
- The Scheme language definition requires that Scheme language processing systems convert all tail-recursive functions to replace that recursion with iteration
- Therefore, it is important, at least for efficiency's sake, to define functions that use recursion to specify repetition to be tail recursive
- Some optimizing compilers for some functional languages can even perform conversions of some non–tail-recursive functions to equivalent tail-recursive functions and then code these functions to use iteration instead of recursion for repetition

Thank You