CSI 3120
Concepts of Programming Languages

Prolog and LISP

Prolog

- Logic Programming
- A Review of Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

Introduction

- Programs in logic languages are expressed in a form of symbolic logic
- Use a logical inferencing process to produce results
- *Declarative* rather that *procedural*:
  - Only specification of *results* are stated (not detailed *procedures* for producing them)

Proposition

- A logical statement that may or may not be true
  - Consists of objects and relationships of objects to each other

Symbolic Logic

- Logic which can be used for the basic needs of formal logic:
  - Express propositions
  - Express relationships between propositions
  - Describe how new propositions can be inferred from other propositions
- Particular form of symbolic logic used for logic programming called *predicate calculus*

Object Representation

- Objects in propositions are represented by simple terms: either constants or variables
- *Constant*: a symbol that represents an object
- *Variable*: a symbol that can represent different objects at different times
  - Different from variables in imperative languages

Compound Terms

- *Atomic propositions* consist of compound terms
- *Compound term*: one element of a mathematical relation, written like a mathematical function
  - Mathematical function is a mapping
  - Can be written as a table

Parts of a Compound Term

- Compound term composed of two parts
  - Functor: function symbol that names the relationship
  - Ordered list of parameters (tuple)
- Examples:

```
student(jon)

like(seth, OSX)

like(nick, windows)

like(jim, linux)
```

Forms of a Proposition

- Propositions can be stated in two forms:
  - *Fact*: proposition is assumed to be true
  - *Query*: truth of proposition is to be determined
- Compound proposition:
  - Have two or more atomic propositions
  - Propositions are connected by operators

Logical Operators

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg$ a	not a
conjunction	$\cap$	a $\cap$ b	a and b
disjunction	$\cup$	a $\cup$ b	a or b
equivalence	$\equiv$	a $\equiv$ b	a is equivalent to b
implication	$\supset$ $\subset$	a $\supset$ b a $\subset$ b	a implies b b implies a

Quantifiers

Name	Example	Meaning
universal	$\forall X.P$	For all X, P is true
existential	$\exists X.P$	There exists a value of X such that P is true

Clausal Form

- Too many ways to state the same thing
- Use a standard form for propositions
- *Clausal form*:
  - $B_1 \cup B_2 \cup \ldots \cup B_n \subset A_1 \cap A_2 \cap \ldots \cap A_m$
  - means if all the As are true, then at least one B is true
- *Antecedent*: right side
- *Consequent*: left side

Predicate Calculus and Proving Theorems

- A use of propositions is to discover new theorems that can be inferred from known axioms and theorems
- *Resolution*: an inference principle that allows inferred propositions to be computed from given propositions

Resolution

- *Unification*: finding values for variables in propositions that allows matching process to succeed
- *Instantiation*: assigning temporary values to variables to allow unification to succeed
- After instantiating a variable with a value, if matching fails, may need to *backtrack* and instantiate with a different value

Proof by Contradiction

- *Hypotheses*: a set of pertinent propositions
- *Goal*: negation of theorem stated as a proposition
- Theorem is proved by finding an inconsistency

Theorem Proving

- Basis for logic programming
- When propositions used for resolution, only restricted form can be used
- *Horn clause* - can have only two forms
  - *Headed*: single atomic proposition on left side
  - *Headless*: empty left side (used to state facts)
- Most propositions can be stated as Horn clauses

Overview of Logic Programming

- Declarative semantics
  - There is a simple way to determine the meaning of each statement
  - Simpler than the semantics of imperative languages
- Programming is nonprocedural
  - Programs do not state now a result is to be computed, but rather the form of the result

Example: Sorting a List

- Describe the characteristics of a sorted list, not the process of rearranging a list

sort(old_list, new_list) $\subset$ permute (old_list, new_list) $\cap$ sorted (new_list)

sorted (list) $\subset \forall$j such that 1$\leq$ j < n, list(j) $\leq$ list (j+1)

The Origins of Prolog

- University of Aix-Marseille (Calmerauer & Roussel)
  - Natural language processing
- University of Edinburgh (Kowalski)
  - Automated theorem proving

Terms

- This book uses the Edinburgh syntax of Prolog
- *Term*: a constant, variable, or structure
- *Constant*: an atom or an integer
- *Atom*: symbolic value of Prolog
- Atom consists of either:
  - a string of letters, digits, and underscores beginning with a lowercase letter
  - a string of printable ASCII characters delimited by apostrophes

Terms: Variables and Structures

- *Variable*: any string of letters, digits, and underscores beginning with an uppercase letter
- *Instantiation*: binding of a variable to a value
  - Lasts only as long as it takes to satisfy one complete goal
- *Structure*: represents atomic proposition

  functor (*parameter list*)

Fact Statements

- Used for the hypotheses
- Headless Horn clauses

```
female(shelley).

male(bill).

father(bill, jake).
```

Rule Statements

- Used for the hypotheses
- Headed Horn clause
- Right side: *antecedent* (***if*** part)
  - May be single term or conjunction
- Left side: *consequent* (***then*** part)
  - Must be single term
- *Conjunction*: multiple terms separated by logical AND operations (implied)

Example Rules

```
ancestor(mary,shelley):- mother(mary,shelley).
```

- Can use variables (*universal objects*) to generalize meaning:

```
parent(X,Y):- mother(X,Y).

parent(X,Y):- father(X,Y).

grandparent(X,Z):- parent(X,Y), parent(Y,Z).
```

Goal Statements

- For theorem proving, theorem is in form of proposition that we want system to prove or disprove – *goal statement*
- Same format as headless Horn

```
man(fred)
```

- Conjunctive propositions and propositions with variables also legal goals

```
father(X, mike)
```

Inferencing Process of Prolog

- Queries are called goals
- If a goal is a compound proposition, each of the facts is a subgoal
- To prove a goal is true, must find a chain of inference rules and/or facts. For goal Q:

```
P2 :- P1
P3 :- P2
...
Q :- Pn
```

- Process of proving a subgoal called matching, satisfying, or resolution

Approaches

- *Matching* is the process of proving a proposition
- Proving a subgoal is called *satisfying* the subgoal
- *Bottom-up resolution, forward chaining*
  - Begin with facts and rules of database and attempt to find sequence that leads to goal
  - Works well with a large set of possibly correct answers
- *Top-down resolution, backward chaining*
  - Begin with goal and attempt to find sequence that leads to set of facts in database
  - Works well with a small set of possibly correct answers
- Prolog implementations use backward chaining

Subgoal Strategies

- When goal has more than one subgoal, can use either
  - Depth-first search: find a complete proof for the first subgoal before working on others
  - Breadth-first search: work on all subgoals in parallel
- Prolog uses depth-first search
  - Can be done with fewer computer resources

Backtracking

- With a goal with multiple subgoals, if fail to show truth of one of subgoals, reconsider previous subgoal to find an alternative solution: *backtracking*
- Begin search where previous search left off
- Can take lots of time and space because may find all possible proofs to every subgoal

Simple Arithmetic

- Prolog supports integer variables and integer arithmetic
- **`is`** operator: takes an arithmetic expression as right operand and variable as left operand

```
A is B / 17 + C
```

- Not the same as an assignment statement!
  - The following is illegal:

```
Sum is Sum + Number.
```

Example

```
speed(ford,100).
speed(chevy,105).
speed(dodge,95).
speed(volvo,80).
time(ford,20).
time(chevy,21).
time(dodge,24).
time(volvo,24).
distance(X,Y) :-   speed(X,Speed),
                   time(X,Time),
                   Y is Speed * Time.
```

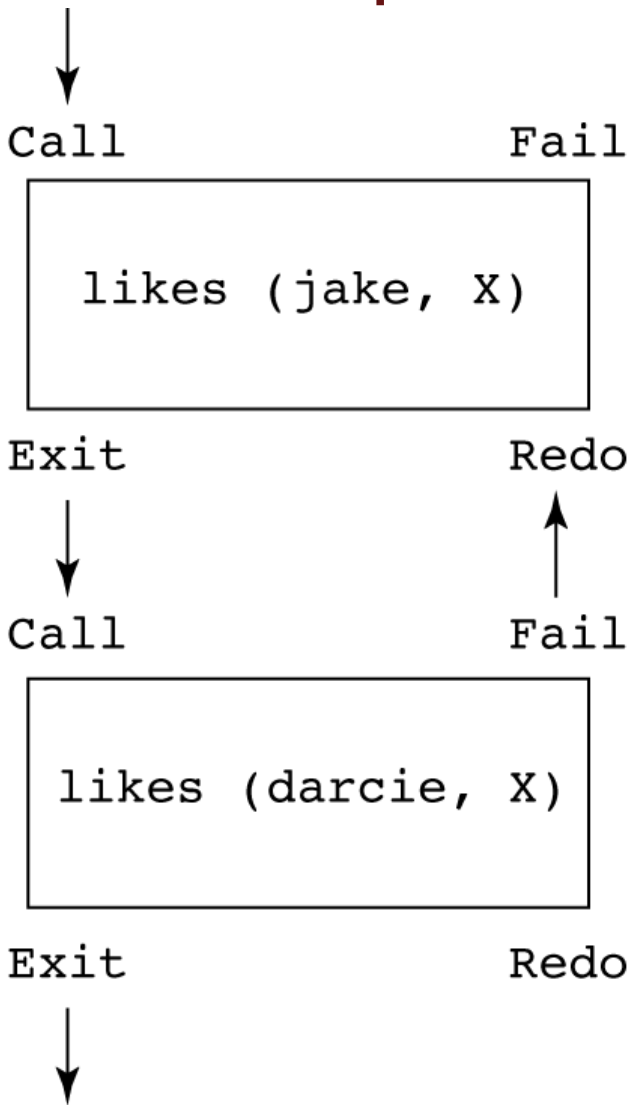
A query: `distance(chevy, Chevy_Distance).`

Trace

- Built-in structure that displays instantiations at each step
- *Tracing model* of execution - four events:
  - *Call* (beginning of attempt to satisfy goal)
  - *Exit* (when a goal has been satisfied)
  - *Redo* (when backtrack occurs)
  - *Fail* (when goal fails)

Example

```
likes(jake,chocolate).

likes(jake, apricots).

likes(darcie, licorice).

likes(darcie, apricots).


trace.

likes(jake, X), likes(darcie, X).
(1)  1 Call: likes(jake, _0)?

(1)  1 Exit: likes(jake, chocolate)

(2)  1 Call: likes(darcie, chocolate)?

(2)  1 Fail: likes(darcie, chocolate)

(1)  1 Redo: likes(jake, _0)?

(1)  1 Exit: likes(jake, apricots)

(3)  1 Call: likes(darcie, apricots)?

(3)  1 Exit: likes(darcie, apricots)

X = apricots
```



List Structures

- Other basic data structure (besides atomic propositions we have already seen): list
- *List* is a sequence of any number of elements
- Elements can be atoms, atomic propositions, or other terms (including other lists)

```
[apple, prune, grape, kumquat]
```

`[]` *(empty list)*

`[X | Y]` *(head X and tail Y)*

Append Example

```
append([], List, List).

append([Head | List_1], List_2, [Head | List_3]) :-
            append (List_1, List_2, List_3).
```

More Examples

```
reverse([], []).

reverse([Head | Tail], List) :-
  reverse (Tail, Result),
           append (Result, [Head], List).
```

```
member(Element, [Element | _]).

member(Element, [_ | List]) :-
  member(Element, List).
```

The underscore character means an anonymous variable—it means we do not care what instantiation it might get from unification

Deficiencies of Prolog

- Resolution order control
  - In a pure logic programming environment, the order of attempted matches is nondeterministic and all matches would be attempted concurrently
- The closed-world assumption
  - The only knowledge is what is in the database
- The negation problem
  - Anything not stated in the database is assumed to be false
- Intrinsic limitations
  - It is easy to state a sort process in logic, but difficult to actually do — it doesn't know how to sort

Applications of Logic Programming

- Relational database management systems
- Expert systems
- Natural language processing

Question set

1. Describe how a multiple-processor machine could be used to implement resolution. Could Prolog, as currently defined, use this method?

2. Write a Prolog description of your family tree (based only on facts), going back to your grandparents and including all descendants. Be sure to include all relationships.

3. Write the following English conditional statements as Prolog headed Horn clauses:

   a) If Fred is the father of Mike, then Fred is an ancestor of Mike.

   b) If Mike is the father of Joe and Mike is the father of Mary, then Mary is the sister of Joe.

   c) If Mike is the brother of Fred and Fred is the father of Mary, then Mike is the uncle of Mary.

Summary

- Symbolic logic provides basis for logic programming
- Logic programs should be nonprocedural
- Prolog statements are facts, rules, or goals
- Resolution is the primary activity of a Prolog interpreter
- Although there are a number of drawbacks with the current state of logic programming it has been used in a number of areas

LISP(s)

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

Introduction

- The design of the imperative languages is based directly on the *von Neumann architecture*
  - Efficiency is the primary concern, rather than the suitability of the language for software development
- The design of the functional languages is based on *mathematical functions*
  - A solid theoretical basis that is also closer to the user, but relatively unconcerned with the architecture of the machines on which programs will run

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 *lambda expression* specifies the parameter(s) and the mapping of a function in the following form

```
λ(x) x * x * x
```

for the function `cube(x) = x * x * x`

Lambda Expressions

- Lambda expressions describe nameless functions
- Lambda expressions are applied to parameter(s) by placing the parameter(s) after the expression

  e.g., `(λ(x) x * x * x)(2)`

  which evaluates to `8`

Functional Forms

- A higher-order function, or *functional form*, is one that either takes functions as parameters or yields a function as its result, or both

Function Composition

- A functional form that takes two functions as parameters and yields a function whose value is the first actual parameter function applied to the application of the second

  Form: `h ≡ f ° g`

  which means `h (x) ≡ f ( g ( x))`

  For `f (x) ≡ x + 2` and `g (x) ≡ 3 * x`,

  `h ≡ f ° g` yields `(3 * x)+ 2`

Apply-to-all

- A functional form that takes a single function as a parameter and yields a list of values obtained by applying the given function to each element of a list of parameters

  Form: α

  For `h(x) ≡ x * x`

  α`(h, (2, 3, 4))` yields `(4, 9, 16)`

Fundamentals of Functional Programming Languages

- The objective of the design of a FPL is to mimic mathematical functions to the greatest extent possible
- The basic process of computation is fundamentally different in a FPL than in an imperative language
  - In an imperative language, operations are done and the results are stored in variables for later use
  - Management of variables is a constant concern and source of complexity for imperative programming
- In an FPL, variables are not necessary, as is the case in mathematics
- *Referential Transparency* - In an FPL, the evaluation of a function always produces the same result given the same parameters

Lisp Data Types and Structures

- *Data object types*: originally only atoms and lists
- *List form*: parenthesized collections of sublists and/or atoms

  e.g., `(A B (C D) E)`
- Originally, Lisp was a typeless language
- Lisp lists are stored internally as single-linked lists

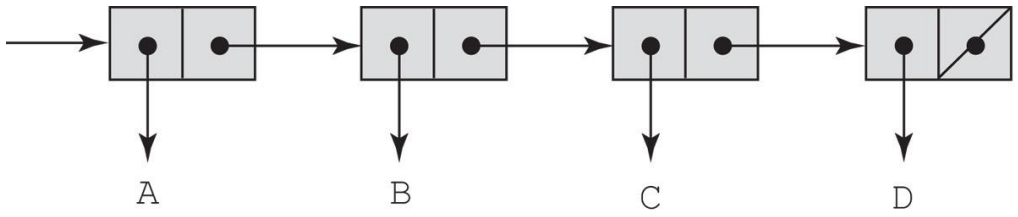
Lisp Interpretation

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

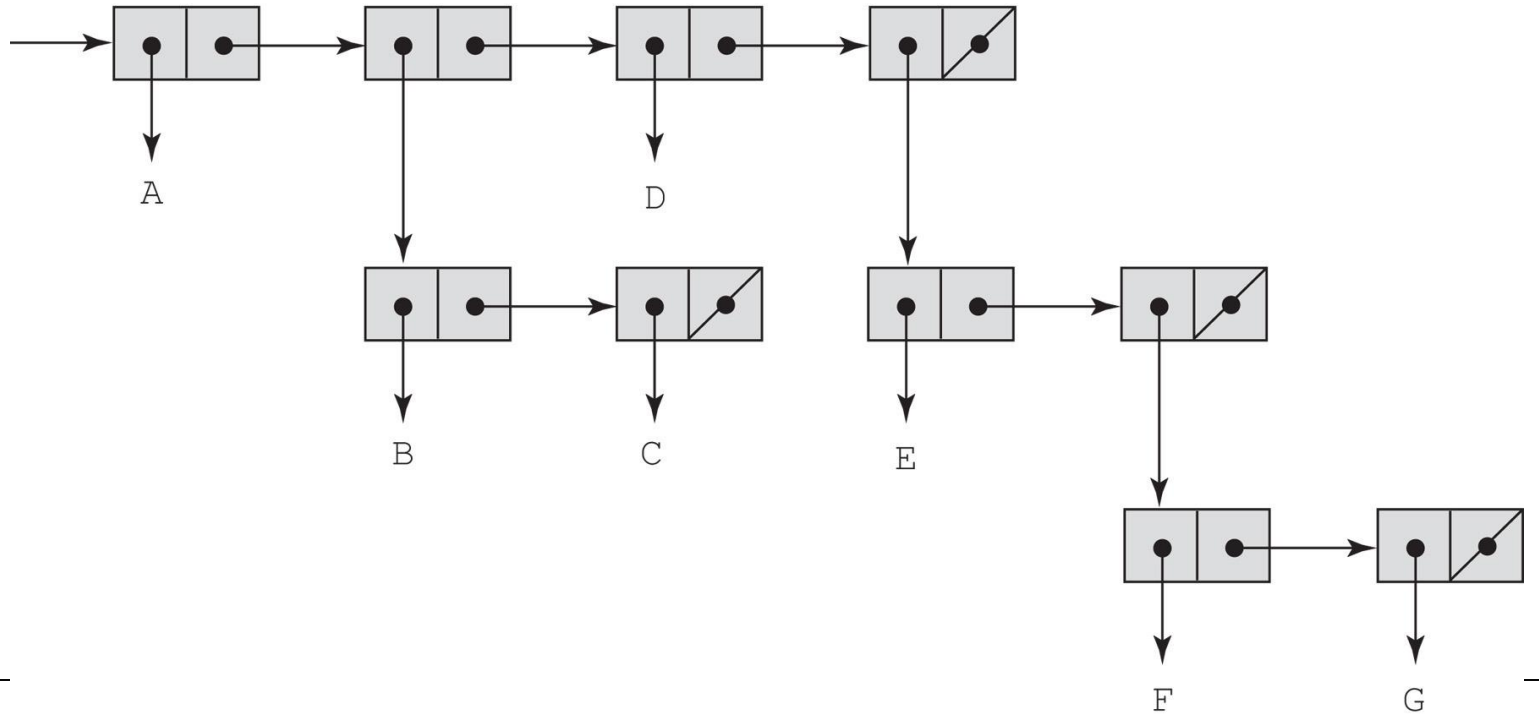
  e.g., If the list `(A B C)` is interpreted as data it is a simple list of three atoms, `A`, `B`, and `C` If it is interpreted as a function application, it means that the function named `A` is applied to the two parameters, `B` and `C`

- The first Lisp interpreter appeared only as a demonstration of the universality of the computational capabilities of the notation

Internal representation of two Lisp lists

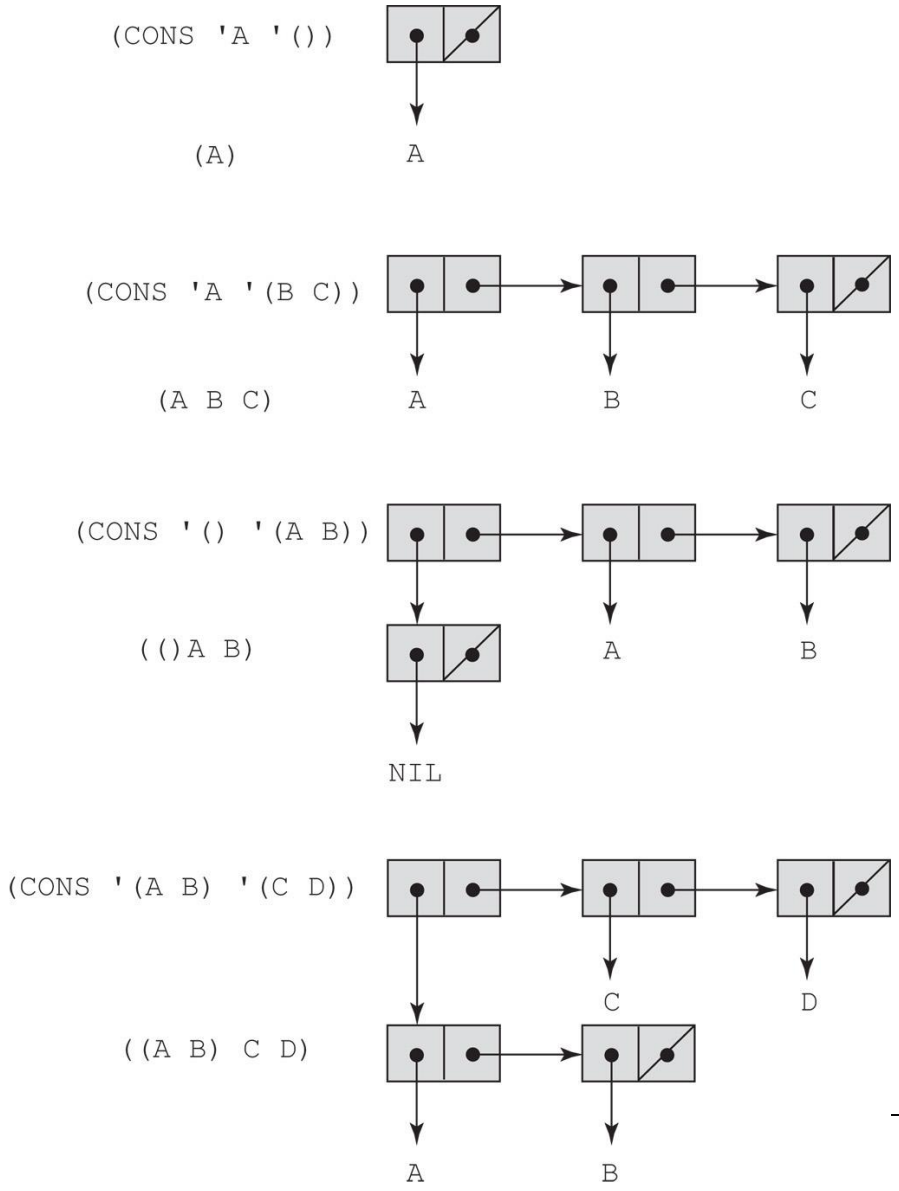


(A B C D)



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

The result of several CONS operations



Origins of Scheme

- A mid-1970s dialect of Lisp, designed to be a cleaner, more modern, and simpler version than the contemporary dialects of Lisp
- Uses only static scoping
- Functions are first-class entities
  - They can be the values of expressions and elements of lists
  - They can be assigned to variables, passed as parameters, and returned from functions

The Scheme Interpreter

- 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)))
```

Special Form Function: `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

- 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

Numeric Predicate Functions

- `#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

Control Flow

- Selection- the special form, `IF`

`(IF` predicate then_exp else_exp`)`

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

- Recall from Chapter 8 the `COND` function:

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

List Functions

- `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

List Functions (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)`

`(CONS 'A 'B)` returns `(A . B)` (a *dotted pair*)

List Functions (continued)

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

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

```
(apple orange grape)
```

Predicate Function: `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`

Predicate Function: `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)

Predicate Functions: `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 lis)) #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: `equal`

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

```
(DEFINE (equal list1 list2)
  (COND
    ((NOT (LIST? list1))(EQ? list1 list2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((equal (CAR list1) (CAR list2))
          (equal (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)))
  (LIST (+ minus_b_over_2a root_part_over_2a))
        (- 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
- A tail recursive function can be automatically converted by a compiler to use iteration, making it faster
- 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)
```

Functional Form - 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
 (DEFINE (third a_list)
    ((compose CAR (compose CDR CDR))
a_list))
```

---

is equivalent to `CADDR`

Functional Form – Apply-to-All

- 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)`

Functions That Build Code

- It is possible in Scheme to define a function that builds Scheme code and requests its interpretation

- This is possible because the interpreter is a user-available function, `EVAL`

Adding a List of Numbers

```
((DEFINE (adder a_list)
   (COND
     ((NULL? a_list) 0)
     (ELSE (EVAL (CONS '+ a_list)))
))
```

- The parameter is a list of numbers to be added; `adder` inserts a `+` operator and evaluates the resulting list
  - Use `CONS` to insert the atom `+` into the list of numbers.
  - Be sure that `+` is quoted to prevent evaluation
  - Submit the new list to `EVAL` for evaluation

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

Common Lisp (continued)

- Macros
  - Create their effect in two steps:
    - Expand the macro
    - Evaluate the expanded macro
- Some of the predefined functions of Common Lisp are actually macros
- Users can define their own macros with `DEFMACRO`

Common Lisp (continued)

- Backquote operator (`)
  - Similar to the Scheme's `QUOTE`, except that some parts of the parameter can be unquoted by preceding them with commas

    `` `(a (* 3 4) c) `` evaluates to `(a (* 3 4) c)`

    `` `(a ,(* 3 4) c) `` evaluates to `(a 12 c)`

Common Lisp (continued)

- Reader Macros
  - Lisp implementations have a front end called the *reader* that transforms Lisp into a code representation. Then macro calls are expanded into the code representation.
  - A reader macro is a special kind of macro that is expanded during the reader phase
  - A reader macro is a definition of a single character, which is expanded into its Lisp definition
  - An example of a reader macro is an apostrophe character, which is expanded into a call to `QUOTE`
  - Users can define their own reader macros as a kind of shorthand

Common Lisp (continued)

- Common Lisp has a symbol data type (similar to that of Ruby)
  - The reserved words are symbols that evaluate to themselves
  - Symbols are either bound or unbound
    - Parameter symbols are bound while the function is being evaluated
    - Symbols that are the names of imperative style variables that have been assigned values are bound
    - All other symbols are unbound

Member Revisited

- The member function could be written as:

```
member b [] = False

member b (a:x)=(a == b) || member b x
```

- However, this would only work if the parameter to squares was a perfect square; if not, it will keep generating them forever. The following version will always work:

```
member2 n (m:x)
     | m < n = member2 n x
     | m == n = True
     | otherwise = False
```

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
- Haskell is a lazy functional language supporting infinite lists and set comprehension.
- 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

Survey Results