Chapter 6

Functional and Logical Programming Languages

Functional Programming Language 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

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

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

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: $h \equiv f \circ g$

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

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

$h \equiv f \circ g$ 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 $h\ (x) \equiv x * x$

$\alpha(\ h, (2, 3, 4))$ 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

Applications of Functional Languages

- APL is used for throw-away programs
- LISP is used for artificial intelligence
  - Knowledge representation
  - Machine learning
  - Natural language processing
  - Modeling of speech and vision
- Scheme is used to teach introductory programming at some universities

- Inefficient execution
- Programs can automatically be made concurrent

Logic Programming Languages

Topics

- Introduction
- A Brief Introduction to Predicate Calculus
- Predicate Calculus and Proving Theorems
- An Overview of Logic Programming
- The Origins of Prolog
- The Basic Elements of Prolog
- Deficiencies of Prolog
- Applications of Logic Programming

Introduction

- *Logic* programming languages, sometimes called *declarative* programming languages
- Express programs 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

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

The Origins of Prolog

- University of Aix-Marseille
    - Natural language processing
- University of Edinburgh
    - Automated theorem proving

Terms

- Edinburgh Syntax
- *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).
sibling(X,Y):- mother(M,X), mother(M,Y),
        father(F,X), father(F,Y).
```

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:

```
B :- A
C :- B
...
Q :- P
```

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

Approaches

- *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!

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.

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)

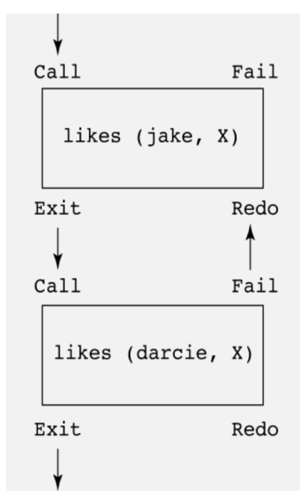
– *Redo* (when backtrack occurs)

– *Fail* (when goal fails)

Example

likes(jake,chocolate).
likes(jake,apricots).
likes(darcie,licorice).
likes(darcie,apricots).

trace.
likes(jake,X),
likes(darcie,X).



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).
```

Reverse Example

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

Deficiencies of Prolog

• Resolution order control

• The closed-world assumption

• The negation problem

• Intrinsic limitations

Applications of Logic Programming

• Relational database management systems

• Expert systems

• Natural language processing