Functional Programming Languages

Learning Objective

This session will introduce students to the functional category of programming languages. Students will gain an understanding of how this language category differs from imperative and object-oriented languages. The session will also present the basic concepts of functional languages, their relative advantages and disadvantages, and some examples of actual code.

Overview

We have spent the majority of this course looking at two of the four categories of programming languages: imperative and object oriented. This is probably as it should be, since the vast majority of software development has been, and is being done, using programming languages that fall into one of these categories. This is not to say that the remaining two categories, functional and logic programming languages, are not worth studying. Much innovative work has been done in these two categories, and powerful programming languages have been developed using the functional and logic paradigms.

There are at least two reasons why functional and logic programming languages may become increasingly important. The first reason is related to the availability of more powerful hardware. We saw early in this course that the popularity of imperative languages was driven by their natural fit to the von Neumann computer architecture. With the slow (by today's standards) processor speeds of early machines, programming languages were primarily imperative in order to maximize run-time processing efficiency. Hardware has now advanced far beyond that point, with true hardware multiprocessors and vast amounts of memory reasonably priced. Other hardware developments have made large-scale integration of non-traditional computing circuitry possible, circuitry that mimics the human brain. These developments make functional and logic languages more attractive.

The second reason for renewed interest in functional and logic languages is the increasing demand for "smart" devices - everything from appliances to weapon systems. These devices are called on to solve problems that are by their very nature non-procedural; constructs such as "do loops" simply aren't a very good approach to building, for example, a voice-recognition system.

We can't hope to do justice to the complicated subject of functional and logic programming language this course, but we will attempt to cover some highlights of each. This session will focus on functional languages, and the final session will focus on logic languages.

Essence of Imperative vs. Non-Imperative PL

Recall that imperative programming languages are a natural extension (abstraction) of the von Neumann architecture: fetch, execute, store... There is no *a priori* reason for this approach to programming-language design; other approaches are possible, and perhaps, desirable. However, given that most hardware platforms are von Neumann-like, implementing non-imperative programming languages generally involves a translation from a non-von Neumann paradigm to

the von Neumann paradigm. This added complication plus the imperative paradigm being somewhat more natural to use, probably accounts for much of the popularity of the imperative approach.

It should be apparent that declarative (non-imperative) languages work at a higher level than imperative languages, e.g., the fact that the programmer doesn't need to be concerned with memory locations. Declarative languages are by nature more *conceptual* than imperative languages

> *Imperative languages specify how a computation is performed by sequences of changes to the computer's store, while declarative languages specify what is to be computed.*

Just as imperative languages are implemented in various ways (e.g., procedural languages, object-oriented languages, etc.), declarative (non-imperative) languages are also implemented in different ways. Two implementation approaches for non-imperative languages: implementations are based on the mathematical notion of a function (implemented in functional languages), and implementations based on logic (implemented in logic languages). Now, let's look more closely at functional languages.

What is a Functional Language?

Functional programming languages are of interest for at least three reasons:

- The notation is concise, allowing for short, elegant programs
- Mathematical function theory is well understood, allowing for easy, automated translation of program specifications into runnable code
- Functional programs are easily adapted to multiprocessor machines

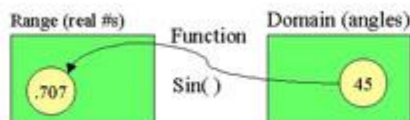
As an example of the third bullet, consider the functional statement:

    (+(*wx)(-yz))

which in standard mathematical form is (w*x) + (y-z). It is apparent that the two functions *wx and -yz could be evaluated on different processors.

A&V define a function as an *"association of a certain object (or objects) from one set (the* ***range****) with each object from another set (the* ***domain****).*"

Consider the simple mathematical sine function f(x) = sin(x). This function maps (associates) the domain set of angles x to the range set of real numbers between the values of -1 and +1. Note how the notation for this works: the range element is determined by paring the domain element with the function name, e.g., sin(45) = 0.707. The mapping is shown here. Note how the function sin( ) maps (associates) the domain object called 45 (degrees) into the range object 0.707. For the purposes of our discussion, we say that the function returns an ***element*** of the range set.



As an abstraction, it is not really necessary to name the mapping function. This was shown by Alonzo Church in 1941 by his invention of lambda-

calculus. A ***lambda expression*** specifies the parameter and the mapping of a function, without naming the function.

Some of the distinguishing characteristics of functional programming languages are:

- Programs are constructed as the composition of functions
- Functions are supported as first-class objects
- There are no, or few, side effects
- Clean and simple semantics are possible.

The second bullet (functions are supported as first-class objects) simply means that the argument to a function may be another function, i.e., embedded functions are allowed. It is also possible for functions to call themselves, i.e., recursion is allowed. In fact, recursion is an important and common practice in functional languages.

The third bullet (few or no side effects) is true because functional languages emphasize values, not memory locations.

The primary characteristic of functional languages is that evaluation occurs through recursion and conditional expressions, rather than through sequencing and iteration as in imperative languages. In functional programming languages, the details of variables and assignments are hidden from the programmer; there are no operational or denotational semantics to be concerned with. Hence, the semantics of functional languages are much simpler than for imperative programming languages.

A functional programming language is comprised of four components: 1) a set of ***primitive functions***, 2) a set of functional forms for constructing complex functions, 3) a function application operation, and 4) structure(s) for representing data.

Some Functional Language Examples

There are several functional, or near-functional, programming languages. Examples include LISP (**Lis**t **P**rocessor), Scheme, APL (**A P**rogramming **L**anguage), ML (**M**eta**L**anguage), SASL, KRC, Haskell, and Miranda. APL, Scheme, and ML, it should be noted, only have some functional-language features. Miranda is probably the only commercial functional language.

LISP was developed by Dartmouth professor John McCarthy. His explicit intent was to develop an artificial intelligence (AI) language; imperative languages deal with numbers, but McCarthy wanted a language that could mimic words and concepts, much like a human brain. Not surprisingly, Alonzo Church, of lambda fame, was McCarthy's thesis advisor.

LISP is an old language, second in age only to FORTRAN for languages that are still in use. The first implementation of LISP was for the IBM 704.

LISP has only one data type: the ***atom***, which is a number or a string. The original LISP had six built-in functions:

- **cons** (adds a new element to the beginning of a list)
- **cond** (a control statement)
- **car** (returns the first element of a list)
- **cdr** (returns all but the first element of a list)
- **eq** (tests for equality between two atoms)
- **atom** (returns "true" if the argument is an atom)

It may not be apparent how a list processor such as LISP can be used, particularly for AI applications. There are at least two characteristics of LISP, however, that make it ideal for such applications. First, extremely complex data structures can be defined using lists. These structures might be used, for example, to characterize the symptoms of various diseases, for a medical AI application. Second, a list can be constructed of function calls, including recursion, to search for solutions to a problem.

The most successful applications of LISP have been in the AI areas of robotics and knowledge engineering (especially expert systems). Some examples of expert systems written in LISP include MACSYMA (a program that can solve mathematical problems symbolically), EXPERT (an expert system for use in endocrinology, ophthalmology, and rheumatology) MYCIN (expert system used to diagnose infectious blood diseases), and DENDRAL (expert system used to analyze chemical, nuclear, and magnetic-resonance data). Please refer to the URL in the references below for descriptions of these expert systems.

Scheme is one of the most popular dialects of LISP in use today. Scheme is available free of charge for a variety of platforms, and has a thriving user community.

Wrap Up

We've seen that declarative (non-imperative) programming languages have a definite software-engineering niche. Non-imperative programming languages generally are functional or logic-based; this session, we've focused on functional languages and have seen that they offer several advantages over imperative languages:

- very simple syntax,
- potential for concurrent execution,
- easy representation of non-mathematical data, e.g., words and concepts, making functional languages suitable for AI applications

Next session, we'll examine logic languages, the last category of programming languages in our sessions.

Logic Programming Languages

Learning Objective

This session introduces students to logic programming languages. Students will gain an understanding of how this language category differs from imperative, object-oriented, and functional programming languages. The session will present the basic concepts of logic languages, their relative advantages and disadvantages, and some examples of actual code.

Overview

At this point, we have looked at three of the four categories of programming languages: imperative, object-oriented, and functional. This week we'll look at the fourth category: logic programming languages. We'll see that logic languages are similar to functional languages in that both categories are declarative; however, the similarity ends there.

In particular, you should understand how lists of clauses are built, and how forward and backward chaining is used to develop or prove inferences.

What is a Logic Language?

Logic languages, like functional languages, are known as declarative (non-imperative) programming languages (PLs). Declarative languages have a number of distinctive characteristics:

- Programs are written by specifying relations or functions
- No program variables are assigned values
- The language's compiler or interpreter manages memory
- Declarative languages are more abstract than imperative languages with respect to their interactions with the CPU

Similar to how functional programs are written (*declared*) as a set of functions, logic programs are written as a set of rules and axioms. Recalling the definition of axiom:
axiom: *a statement accepted as true as the basis for argument*

leads to the concept of logic programming:

The basic concept of a logic program is to assemble a list of rules and axioms, and then use those rules and axioms to deduce new facts, or to prove or disprove assertions.

Logic is one of the oldest branches of mathematics, having roots that go clear back to Aristotle in the fourth century B.C. Aristotle's logic was assembled into an ancient document under the title *Organon*

"*All students work hard"* and *Some students eat a lot"* leads to the logical deduction that "*Some who eat a lot work hard."* This is a very simple example of how new truths can be deduced from a given set of axioms and rules.

Aristotle made a nice start in defining the field of logic, but work by mathematicians in the last century or so demonstrated the gaps in his work. More recent work has led to the development of ***propositional,*** or ***predicate, calculus***, which is a set of rules for calculating the truth values of **propositions**. Note that the truth values are binary: "true" or "false."

Logic programming languages are based on a subset of predicate (**propositional**) calculus. A breakthrough in logic programming language came in 1965, when J.Alan Robinson published the theory of ***resolution:*** "resolution is an inference rule that allows inferred **propositions** to be computed from given **propositions**, thus providing a method with potential application to automatic theorem proving."

Logic-based programs are constructed using:

- a series of ***axioms*** (i.e., facts)
- rules of inference
- a theorem or query (a **proposition**) that is to be proved

Propositions can be presented in either of two modes:

- Mode 1: The **proposition** is stated to be "true"
- Mode 2: The "truth" of the **proposition** is to be determined

The reason that logic programming languages are categorized as declarative should now be apparent: programs written in a logic language consist of declarations rather than assignments and control-flow statements. Like functional languages, the semantics of logic languages are much simpler than imperative languages, since the meaning of **propositions** is self-contained. Also, programming languages are non-procedural, in that the programmer specifies the desired result, rather than specifying the precise details of a computation.

Logic Language Implementation

Prolog is <u>the</u> logic programming language, although other languages such as the Gödel programming language are gaining popularity. Prolog was invented by Alain Colmerauer and Philippe Roussel at the University of Marseille in the early 1970s, and was an outgrowth of work done at the University of Edinburgh by Robert Kowalski on automatic theorem proving.

The work in developing a Prolog application is primarily the development of the list of clauses. As mentioned before, an application may require thousands of facts and rules ("clauses"). Once the list of clauses is built, the program can be run repeatedly to produce many new inferred **propositions**, or to verify the truthfulness of submitted **propositions**.

Logic Language Applications

Several factors should be kept in mind regarding logic language applications:

- *Memory requirements*: useful logic program applications may have thousands of rules, requiring large amounts of memory to store the facts and rules.
- *Processor speed*: large lists of facts and rules will result in a huge number of relationships that must be tested when inferring new relationships.
- *Potential of Distributed Processing*: Since facts and rules do not need to be searched in any particular order, logic programming is a natural candidate for distributed processing, i.e., a logic application can be implemented across many networked platforms.

Three categories of applications that benefit from logic programming languages: relational database systems, expert systems, and natural language systems.

For those familiar with RDBMS, the power of logic languages should be apparent, i.e., logic PLs allow very complex and sophisticated query engines to be built.

Expert and natural-language systems fall under the category of artificial intelligence, or AI. In principle, the power of logic languages in building expert systems based on the idea of using human experts to build a large list of clauses, and then allow non-experts to query the system. In practice, it often turns out that the clause list may be missing one or more facts or rules necessary to infer a proposition, so that no definitive answer is produced. AI computer scientists are working to overcome this deficiency.

Natural language processing uses logic languages to specify rules of grammar, so that natural-language utterances can be parsed. Again, the practical problem is in obtaining a comprehensive list of clauses. The "grammar checker" in some word-processing programs can be implemented with logic language.

In summary, although logic programming languages have been around for a long time, they have yet to fulfill their potential. This may change, however, with the availability of distributed computing and the drastic reduction in the cost of memory.

Wrap Up

In the Earlier sessions we considered the reasons to study programming languages; let's take this opportunity to reiterate them:

- ***Increased capacity to express ideas***: We've looked at many concepts in this course: the distinction between syntax, static and dynamic semantics, orthogonality, the object-oriented paradigm, static versus dynamic binding, and pointers, just to name a few. An understanding of these concepts should enhance your capabilities as software developers and managers.
- ***As background for choosing an appropriate language:*** We've looked at many of the pros and cons of various programming languages. Again, in your roles as software developers and managers, this knowledge should enhance your ability to assess the

choice (by yourself or with others) of the right language for the job, and to make that choice objectively, rather than by simply picking the latest "cool" language.

- ***Improved ability to learn other languages:*** We have seen repeatedly that the same concepts are used in many languages - such as control methods. If you are involved with hands-on coding (or with reviewing the code of others), your ability to learn new languages should be enhanced.
- ***Better ability to design new languages:*** The concept of programming language is becoming blurred. Database engines, for example, are beginning to incorporate object-oriented and AI concepts. You may well find yourself working on a project that incorporates some aspect of language design.
- ***Overall advancement in computing:*** As we said in the first session, understanding what we have now will enable software engineers and IT managers to make incremental (or quantum) improvements to the field of computing.

- ## Recommended Readings:

- ***Programming Languages: Paradigm and Practice***
  by Doris Appleby and Julius J. VandeKopple; 444 pages,
  Published by McGraw-Hill Companies; 1997; ISBN 0-07-005315-4
- ***Article***
  Hudak, P. (1989). Conception, evolution, and application of functional programming languages, *ACM computing surveys* 21(3): 359-411.
-