4.1 Characteristics of Functional Programming Languages

Chapter 3 examined imperative languages, those languages with a design that is an abstraction of the underlying von Neumann architecture. While imperative languages continue to be the most widely used languages, renewed interest in functional languages has occurred partly due to the increasing popularity of multi-threaded programming. A function written in a pure functional programming language is thread safe; it always produces the correct result even when executed as part of multiple concurrent threads.

Functional programming languages, also known as applicative programming languages, have a mathematically based design, specifically, lambda calculus. A mathematical function maps a value in one set, known as the range, into a value in another set, known as the domain, and always produces the same domain value for a specific value in the range. There is no notion of a variable in mathematics that is used to model a memory location. A statement such as: $x = 2x + 1$ makes no sense in mathematics. Although all functional languages have some imperative features, a functional language is considered pure if it does not contain variables that model memory locations. In addition, there is no assignment statement in a pure functional language because the purpose of an assignment statement is to modify a variable. Similarly, since data is modified by assignments, the data in a functional language is immutable.

If you are comfortable with writing in an imperative style, you may be baffled by the concept of programming without variables. Imperative programmers use variables to represent the state of a running program, for example, a variable to keep track of the number of times a loop has been executed. In fact, a state is created by the execution of a functional program, but this state is implicit. Rather than storing values in programmer named variables, computations are stored in memory implicitly allocated during the program execution. For example, compare these two implementations of a factorial function, one in an imperative style and one in a functional style. The imperative style uses the variables result and $i$, and assignment statements to repeatedly modify these variables. Note that the iteration is controlled by the iteration variable, $i$. In the absence of variables, recursion is used in the functional implementation. The recursive function calculates temporary values that are implicitly stored in memory and used to calculate the final result.

```c
/* Imperative Style */
/* Variables used to store results and control iteration */
int factorial(int n)
{
    int result = 1, i;
    for (i = 2; i <= n; i++)
        result = result * i;
    return result;
```
/* Functional Style */
/* No variables, only arguments. */
/* Recursion instead of iteration. */

```c
int factorial(int n)
{
    if (n == 0) return 1;
    if (n == 1) return 1;
    return n * factorial(n - 1);
}
```

In addition to recursion, function composition, the application of one function to the results of another, plays a major role in functional programming. For example, consider the two implementations of a function to compute the distance between two cartesian coordinates. The imperative style relies on variables to hold intermediary results. The functional style relies on function composition to apply functions to the intermediary results.

/* Imperative Style */
/* Variables and assignment statements */
float distance(float x1, float y1,
               float x2, float y2)
{
    float x1minusx2 = x1 - x2;
    float y1minusy2 = y1 - y2;
    float xsqr1 = x1minusx2 * x1minusx2;
    float ysqr2 = y1minusy2 * y1minusy2;
    float sumsqrs = xsqr1 + ysqr2;
    return sqrtf(sumsqrs);
}

/* Functional Style */
/* Uses function composition */
float distance(float x1, float y1, float x2, float y2)
{
    return sqrtf(sum(square(subtract(x1, x2)),
                    square(subtract(y1, y2))));
}

The remainder of this section discusses specific characteristics of functional programming languages and the advantages of functional programming.

### 4.1.1 Higher Order Functions
Higher order function takes functions as parameters or return a function as a result or possibly both. Although imperative languages can provide similar functionality, for example C allows function pointers to be passed to other functions, syntactically these can be much cumbersome than in functional languages.

Let’s take a look at higher order functions in the functional programming languages Lisp and Haskell. The functional programming language Lisp has a built-in function called mapcar that takes as input a function and one or more list parameters and applies the function to successive elements of each list. This type of function is called an apply-to-all function. For example, the call below demonstrates the use of mapcar within the lisp interpreter. The > symbol is the lisp prompt. The parameters passed to the mapcar function are first, and ((a b) (c d)). The list parameter contains two sublists, (a b) and (c d). The first function is applied to each sublist to obtain the first element of those sublists, a and c. The results are returned in a list, (a c).

```lisp
(mapcar 'first '(((a b) (c d)))
(A C)
```

Here is an example in the functional programming language Haskell of a function that returns a function as a result. The function composition operator (.) takes two functions and returns a function that is the composition of those functions. For example, f . g produces a function that applies g to the function parameters and then f to the result of g. In the example below, Prelude> is the Haskell interpreter prompt. The first statement defines a function square that accepts a single parameter, x, and returns the square of that result. Note that the parameter, x, is not within parentheses; in Haskell, parentheses are only need to override the default evaluation order. The second statement defines a function add1. The third statement demonstrates the use of the function composition operator to create a function called sqradd1. The function is called in the final line; notice that it first computes 3 + 1 (by applying add1) and then squares that value (by applying square).

```haskell
Prelude> let square x = x * x
Prelude> let add1 x = x + 1
Prelude> let sqradd1 = square . add1
Prelude> sqradd1 3
16
```

talk about function currying?

4.1.2 First Class Functions

Functions are first class if they can be treated just like other values in the program. For example, a first class function can be constructed at run-time, passed as a parameter, returned from a function, and assigned into a variable.
Entities in a program that can be treated this way are called first-class values or first-class objects. Note that the term object in this definition does not necessarily imply an object in an object-oriented language.

Although most imperative languages do not contain first class functions, they provide features that support the functionality. For example, Java supports the creation of a functor, which is an object whose purpose is to encapsulate usually only one method. These objects can then be passed to other methods in order to invoke the method the functor encapsulates. The C programming language supports the creation of function pointers that can be passed as parameters, but does not allow the dynamic creation of functions.

The Lisp and Haskell examples in the previous section show how functions can be passed as values to other functions and how functions can be created at runtime. As further evidence that functions are just like other values in the program, in Haskell all values have a type, including functions. The code below displays the definition of a variable and then a use of :type to determine its type is Integer. The sqradd1 function also has a type, from integer to integer, as can also be seen in the example.

Prelude> let x = 3
Prelude> :type x
x :: Integer
Prelude> :t sqradd1
sqradd1 :: Integer -> Integer

4.1.3 Referential Transparency

An expression is referentially transparent if it always yields the same value whenever it is evaluated. Any functions that are called as part of the expression evaluation must be pure functions. Pure functions must not depend upon the state of the running program (for example, the values of global variables) or input from an I/O device. In addition, pure functions must not modify state information or output to an I/O device. In other words, these functions cause no side effects. The result of the function should only be dependent upon its arguments.

Although the absence of variables in pure functional languages does not guarantee referential transparency, this does limit the possibility of side effects. For example, compare the two C functions below. The first function is pure; its output is based only on its input. In addition, it does not modify any variables. Thus, any call to the increment function is referentially transparent. On the other hand, the addCount function computes a result using both its argument and the statically allocated variable k. The variable is allocated space that is initialized to 1 when the program begins execution. Each time the function is called, 1 is added to k. Thus, the result of this function is dependent upon the value of the argument and the number of previous calls to addCount.
/* Pure function: output is only dependent upon input */
int increment(int n)
{
    return n + 1;
}

/* Impure function: output is dependent upon the input */
/* and state (value of k). Function modifies state. */
int addCount(int n)
{
    static int k = 1;
    k = k + 1
    return n + k;
}

The importance of referential transparency is that it allows programmers to reason about program behavior, for example, in proving program correctness and debugging code. In addition, referential transparency assists the optimizing compiler in eliminating the recalculation of expressions when it is known that a recalculation will always produce the same result. Most programming languages, even functional languages, do not guarantee that expressions in the language are referentially transparent because of the incorporation of imperative style features such as assignment statements. However, the functions in the language Haskell and Miranda are pure and all functional languages can be used in mostly a purely functional style.

### 4.1.4 Implicit Parametric Polymorphism

Polymorphic subroutines are those that can work with multiple types of data. Many programming languages support either parametric or subtype polymorphism. In *explicit parametric polymorphism*, code, such as a class, takes a parameter that specifies the type of data for which the code is customized. Explicit parametric polymorphism is named *generics* in Java, Ada, Eiffel, C# and Visual Basic. This functionality is provided via *templates* in C++.

Many functional languages provide *implicit parametric polymorphism*. Functions in the language can operate on different types of data, but without the programmer being required to indicate the type upon which the function will operate. In Lisp dialects, this is implemented by examining the type of the data before application of a built-in function to determine whether the function can be applied to operands of that type. For example, consider the Lisp implementation of a max function. The Lisp function, *defun*, is used to define a function named *mymax* that takes two arguments. (> a b) causes the > function to be applied to a and b. Notice that Lisp requires that the name of the function, >, is inside of the parenthesis with the parameters. The Lisp interpreter will determine
immediately before application of the > function whether this function can be applied to operands of that type. Delaying the type checking until runtime in this manner is called *dynamic type checking*. The lines below the definition display the result of calling mymax within the Lisp interpreter (the interpreter prompt is >). Notice that this function can be invoked on both integer and floating point data.

(\texttt{(\textbf{defun mymax} a b) (if (\textbf{>} a b) a b)})
\texttt{> (mymax 4 10)}
\texttt{10}
\texttt{> (mymax 5.2 3.1)}
\texttt{5.2000000000000002}

Haskell also supports implicit parametric polymorphism. Below is the Haskell definition of \texttt{mymax} and invocations of the function within the Haskell interpreter. (The *\texttt{Main>} is the interpreter prompt.) \texttt{mymax a b} provides the name of the function and the number of parameters. The code on the right side of the = is the body of the function. Notice the invocations of mymax function on integer, string and character data. The last command uses \texttt{:type} to display the type of the mymax function. \texttt{Ord a} indicates that the type of the parameters to mymax must be ordered datatypes; specifically, the data must be of a type to which the \texttt{>} operator can be applied. The programmer did not supply a type signature so Haskell inferred the type of the arguments and the return type based upon the operations involved in the function definition. This allows the Haskell compiler to perform *static type checking* by comparing the types of the data used in the call to the types specified in the inferred type signature.

\texttt{mymax a b = if a > b then a else b}
*\texttt{Main>} mymax 4 10
\texttt{10}
*\texttt{Main>} mymax "abc" "aac"
\texttt{abc}  
*\texttt{Main>} mymax 'a' 'c'
\texttt{c}  
*\texttt{Main>} :type mymax
\texttt{mymax :: Ord a \Rightarrow a \Rightarrow a \Rightarrow a}

Finally, note that implicit parametric polymorphism is not the same as function overloading. In function overloading, multiple functions with the same name and the same number of parameters (but with different types) provide similar functionality but on different types of data. With parametric polymorphism, there is only one function that can be invoked using different types of data.

### 4.1.5 List Types

Earlier we noted the use of recursion in functional programming languages and higher order functions such as apply-to-all functions. These mechanisms are ideally suited for use with list structures and most functional programming languages provide a list type and many built-in list operations.
For example, consider the list structure in the Lisp programming language. A list in Lisp contains s-expressions separated by whitespace enclosed within parenthesis. An s-expression is a sequence of characters and sublists. For example, (a b c), ((a b) c) and (+ 3 4) are s-expressions. The code below shows how the Lisp first and rest functions can be used to obtain the first element and the remaining elements of a list. The quote (') is needed to cause the s-expression elements to be treated as data; otherwise, the Lisp interpreter will assume that a is a function name and the remaining elements are the parameters to the a function. (You can see by the Lisp interpreter response that Lisp is case-insensitive, thus a is equivalent to A.)

```
> (first '(a b (c d) e))
A
> (rest '(a b (c d) e))
(B (C D) E)
```

The function definition of mymin below also causes the creation of an s-expression that represents the function. Lisp is a homoiconic language which means that a structure in the language (the list) is used to represent functions in the language. The #’ syntax in the example below causes the function associated with the symbol to be used. Notice that the second and third functions are being used to grab the second and third elements of the function definition!

```
> (defun mymin(a b) (if (< a b) a b))
MYMIN
> (second #'mymin)
MYMIN
> (third #'mymin)
(A B)
```

Unlike Lisp, the elements of a list in the functional language ML must all be of the same type. Lists in ML are comma-separated elements of the same type enclosed within square brackets. ML provides numerous list manipulation functions including hd to obtain the first element of the list, tl to obtain all but the first element of the list and the list concatenation operator, @. The example below demonstrates the creation of two lists of strings and the use of the hd, tl and @ operations. The dash (-) is the ML interpreter prompt. val is used to define a value, for example a list, and bind a variable, for example females, to that value.

```
- val females = ["Carol", "Marcia", "Jan", "Cindy"];
val females = ["Carol","Marcia","Jan","Cindy"] : string list
- val males = ["Mike", "Greg", "Peter", "Bobby"];
val males = ["Mike","Greg","Peter","Bobby"] : string list
- val family = males @ females;
val family = ["Mike","Greg","Peter","Bobby","Carol","Marcia","Jan","Cindy"]
: string list
- hd family;
```
4.1.6 Benefits of Functional Programming

Functional programming languages offer numerous benefits. For one, the presence of higher order functions and use of recursion and pattern matching, allow programmers to write very powerful code with a small number of statements. The programs are easier to understanding (after going through the language learning curve) than comparable programs in imperative languages and easier to prove correct.

Functions that are pure provide benefits for testing and debugging. Pure functions return a result based only upon the arguments of the function and do not modify variables that would be used by another function. In addition, the behavior of a function does not depend upon the code executed before that function. Thus, functions can be tested and debugged in isolation of other functions.

The extensive polymorphism available in functional languages eases the task of writing reusable code. Most of the functional languages employ implicit parametric polymorphism - functions naturally accept arguments of any type that can be handled by the operations applied to them by the function. Even in languages that perform static typing such as Haskell, the programmer can omit the type of the function parameters and Haskell will infer an appropriate polymorphic type.

If all functions in the program are pure, then concurrency comes easily. For example, the two calls to `computeSomeValue` below can be executed simultaneously without concern that the second call is dependent upon some state produced by the first call. These calls are referentially transparent – they will always yield the same value when given a specific input. Thus, the calls to `computeSomeValue` can be executed in a different order or at the same time.

```plaintext
num1 = computeSomeValue(input1);
num2 = computeSomeValue(input2);
result = computeResult(num1, num2);
```

Functional languages relieve the programmer of the difficulties of memory management. Data representation is hidden by the language and programmers do not need to be concerned about any limitations imposed by the underlying architecture. The memory allocation needed for temporaries is implicit and these languages provide automatic garbage collection to reclaim memory that is no longer needed.