LISP was one of the earliest languages with an approximation to a functional subset. It has a significant loyal following, particularly in the Artificial Intelligence community, and is programmed using many functional techniques. Here, COMMON LISP was chosen as a widely used modern LISP. Like SML, it lacks normal order reduction. Unlike SML, it combines minimal syntax with baroque semantics, having grown piecemeal since the late 1950's.

Acknowledgements

I had the good fortune to be taught Computer Science at the University of Essex from 1970 to 1973. There I attended courses on the theory of computing with Mike Brady and John Laski, which covered the λ calculus, recursive function theory and LISP, and on programming languages with Tony Brooker, which also covered LISP. Subsequently, I was a postgraduate student at St Andrew's University from 1974 to 1977 where I learnt about functional language design and implementation from Tony Davie and Dave Turner. I would like to thank all these people for an excellent education.

I would also like to thank my colleagues at Napier College, Glasgow University and Heriot-Watt University with whom I have argued about many of the ideas in this book, in particular Ken Barclay, Bill Findlay, John Patterson, David Watt and Stuart Anderson.

I would, of course, like to thank everyone who has helped directly with this book:

- Paul Chisholm for patiently and thoroughly checking much of the material: his help has been invaluable.
- David Marwick for checking an early draft of chapter 1 and Graeme Ritchie for checking an early draft of chapter 10.
- Peter King, Chris Miller, Donald Pattie, Ian Crorie and Patrick McAndrew, in the Department of Computer Science, Heriot-Watt University, who provided and maintained the UNIX† facilities used to prepare this book.
- Mike Parkinson and Stephen Troth at Addison-Wesley for their help in the development of this book, and Andrew McGettrick and Jan van Leeuwen for their editorial guidance.

I would particularly like to thank Allison King at Addison-Wesley.

Finally, I would like to thank my students.

I alone am responsible for bodges and blunders lurking within these pages. If you spot any then please let me know.

Greg Michaelson, Edinburgh, 1988

1. INTRODUCTION

1.1. Introduction

Functional programming is an approach to programming based on function calls as the primary programming construct. It provides practical approaches to problem solving in general and insights into many aspects of computing. In particular, with its roots in the theory of computing, it forms a bridge between formal methods in computing and their application.

In this chapter we are going to look at how functional programming differs from traditional imperative programming. We will then consider functional programming's origins in the theory of computing and survey its relevance to contemporary computing theory and practise. Finally, we will discuss the role of the λ (lambda) calculus as a basis for functional programming.

---

† UNIX is a trademark of Bell Laboratories.

1.2. Names and values in programming

Some of the simplest computing machines are electronic calculators. These are used to carry out arithmetic calculations with numbers. The main limitation to a calculator is that there is no way of generalising a calculation so that it can be used again and again with different values. Thus, to carry out the same calculation with different values, the whole calculation has to be re-entered each time.

Programming languages provide names to stand for arbitrary values. We write a program using names to stand for values in general. We then run the program with the names taking on particular values from the input. The program does not have to change to be used with different values: we simply change the values in the input and the computer system makes sure that they are associated with the right names in the program.

As we will see, the main difference between imperative programming languages, like Pascal, FORTRAN and COBOL, and functional programming languages, like SML and Miranda, lies in the rules governing the association of names and values.

1.3. Names and values in imperative and functional languages

Traditional programming languages are based around the idea of a **variable** as a changeable association between a name and values. These languages are said to be **imperative** because they consist of sequences of commands:

```
<command1> ;
<command2> ;
<command3> ;
 ...
```

Typically, each command consists of an **assignment** which changes a variables' value. This involves working out the value of an expression and associating the result with a name:

```
<name> := <expression>
```

In a program, each command's expression may refer to other variables whose values may have been changed by preceding commands. This enables values to be passed from command to command.

Functional languages are based on structured function calls. A functional program is an expression consisting of a function call which calls other functions in turn:

```
<function1>(<function2>(<function3> ... ) ... ))
```

Thus, each function receives values from and passes new values back to the calling function. This is known as function **composition** or **nesting.**

In imperative languages, commands may change the value associated with a name by a previous command so each name may be and usually will be associated with different values while a program is running.

In imperative languages, the same name may be associated with different values.

In functional languages, names are only introduced as the formal parameters of functions and given values by function calls with actual parameters. Once a formal parameter is associated with an actual parameter value there is no way for it to be associated with a new value. There is no concept of a command which changes the value associated with a name through assignment. Thus, there is no concept of a command sequence or command repetition to enable successive changes to values associated with names.

In functional languages, a name is only ever associated with one value.

1.4. Execution order in imperative and functional languages

In imperative languages, the order in which commands are carried out is usually crucial. Values are passed from command to command by references to common variables and one command may change a variable's value before that variable is used in the next command. Thus, if the order in which commands are carried out is changed then the behaviour of the whole program may change.

For example, in the program fragment to swap `X` and `Y`:

```
T := X ;
X := Y ;
Y := T
```

`T`'s value depends on `X`'s value, `X`'s value depends on `Y`'s value and `Y`'s value depends on `T`'s value. Thus, any change in the sequence completely changes what happens. For example:

```
X := Y ;
T := X ;
Y := T
```

sets `X` to `Y` and:

```
T := X ;
Y := T ;
X := Y
```

sets `Y` to `X`.

Of course, not all command sequences have fixed execution orders. If the commands' expressions do not refer to each others' names then the order does not matter. However, most programs depend on the precise sequencing of commands.

Imperative languages have fixed execution orders.

In functional languages, function calls cannot change the values associated with common names. Hence, the order in which nested function calls are carried out does not matter because function calls cannot interact with each other.

For example, suppose we write functions in a Pascalish style:

```
FUNCTION F( X,Y,Z:INTEGER):INTEGER ;
BEGIN ...  END
FUNCTION A(P:INTEGER):INTEGER ;
BEGIN ...  END
FUNCTION B(Q:INTEGER):INTEGER ;
BEGIN ...  END
FUNCTION C(R:INTEGER):INTEGER ;
BEGIN ...  END
```

In a functional language, in the function call:

```
F(A(D),B(D),C(D))
```

the order in which `A(D), B(D)` and `C(D)` are carried out does not matter because the functions `A, B` and `C` cannot change their common actual parameter `D.`

In functional languages, there is no necessary execution order.

Of course, functional programs must be executed in some order - all programs are - but the order does not affect the final result. As we shall see, this execution order independence is one of the strengths of functional languages and has led to their use in a wide variety of formal and practical applications.

1.5. Repetition in imperative and functional languages

In imperative languages, commands may change the values associated with a names by previous commands so a new name is not necessarily introduced for each new command. Thus, in order to carry out several commands several times, those commands need not be duplicated. Instead, the same commands are repeated. Hence, each name may be and usually will be associated with different values while a program is running.

For example, in order to find the sum of the N elements of array A we do not write:

```
SUM1 := A[1] ;
SUM2 := SUM1 + A[2] ;
SUM3 := SUM2 + A[3] ;
 ...
```

Instead of creating N new SUMs and refering to each element of A explicitly we write a loop that reuses one name for the sum, say SUM, and another that indicates successive array elements, say I:

```
I := 0 ;
SUM := 0 ;
WHILE I < N DO
BEGIN
     I := I + 1 ;
     SUM := SUM + A[I]
END
```

In imperative languages, new values may be associated with the same name through command repetition.

In functional languages, because the same names cannot be reused with different values, nested function calls are used to create new versions of the names for new values. Similarly, because command repetition cannot be used to change the values associated with names, **recursive** function calls are used to repeatedly create new versions of names associated with new values. Here, a function calls itself to create new versions of its formal parameters which are then bound to new actual parameter values.

For example, we might write a function in a Pascalish style to sum an array:

```
FUNCTION SUM(A:ARRAY [1..N] OF INTEGER; I,N:INTEGER):INTEGER;
BEGIN
     IF I > N THEN
      SUM := 0
     ELSE
      SUM := A[I] + SUM(A,I+1,N)
END
```

Thus, for the function call:

```
SUM(B,1,M)
```

the sum is found through successive recursive calls to SUM:

```
B[1] + SUM(B,2,M) =

B[1] + B[2] + SUM(B,3,M)
```

```
B[1] + B[2] + ... + B[M] + SUM(B,M+1,M)

B[1] + B[2] + ... + B[M] + 0
```

Here, each recursive call to `SUM` creates new local versions of `A, I` and `N,` and the previous versions become inaccessable. At the end of each recursive call, the new local variables are lost, the partial sum is returned to the previous call and the previous local variables come back into use.

In functional languages, new values are associated with new names through recursive function call nesting.

1.6. Data structures in functional languages

In imperative languages, array elements and record fields are changed by successive assignments. In functional languages, because there is no assignment, sub-structures in data structures cannot be changed one at a time. Instead, it is necessary to write down a whole structure with explicit changes to the appropriate sub-structure.

Functional languages provide explicit representations for data structures.

Functional languages do not provide arrays because without assignment there is no easy way to access an arbitrary element. Certainly, writing out an entire array with a change to one element would be ludicrously unwieldy. Instead nested data structures like lists are provided. These are based on recursive notations where operations on a whole structure are described in terms of recursive operations on sub-structures. The representations for nested data structures are often very similar to the nested function call notation. Indeed, in LISP the same representation is used for functions and data structures.

This ability to represent entire data structures has a number of advantages. It provides a standard format for displaying structures which greatly simplifies program debugging and final output as there is no need to write special printing sub-programs for each distinct type of structure. It also provides a standard format for storing data structures which can remove the need to write special file I/O sub-programs for distinct types of structure.

A related difference lies in the absence of global structures in functional languages. In imperative languages, if a program manipulates single distinct data structures then it is usual to declare them as globals at the top level of a program. Their sub-structures may then be accessed and modified directly through assignment within sub-programs without passing them as parameters.

In functional languages, because there is no assignment, it is not possible to change independently sub-structures of global structures. Instead, entire data structures are passed explicitly as actual parameters to functions for sub-structure changes and the entire changed structure is then passed back again to the calling function. This means that function calls in a functional program are larger than their equivalents in an imperative program because of these additional parameters. However, it has the advantage of ensuring that structure manipulation by functions is always explicit in the function definitions and calls. This makes it easier to see the flow of data in programs.

1.7. Functions as values

In many imperative languages, sub-programs may be passed as actual parameters to other sub-programs but it is rare for an imperative language to allow sub-programs to be passed back as results. In functional languages, functions may construct new functions and pass them on to other functions.

Functional languages allow functions to be treated as values.

For example, the following contrived, illegal, Pascalish function returns an arithmetic function depending on its parameter:

```
TYPE OPTYPE = (ADD,SUB,MULT,QUOT);
```

```
FUNCTION ARITH(OP:OPTYPE):FUNCTION;
 FUNCTION SUM(X,Y:INTEGER):INTEGER; BEGIN SUM := X+Y END;
 FUNCTION DIFF(X,Y:INTEGER):INTEGER; BEGIN DIFF := X-Y END;
 FUNCTION TIMES(X,Y:INTEGER):INTEGER; BEGIN TIMES := X*Y END;
 FUNCTION DIVIDE(X,Y:INTEGER):INTEGER; BEGIN DIVIDE := X DIV Y END;
BEGIN
     CASE OP OF
       ADD: ARITH := SUM;
       SUB: ARITH := DIFF;
      MULT: ARITH := TIMES;
      QUOT: ARITH := DIVIDE;
     END
END
```

Thus:

```
ARITH(ADD)
```

returns the `FUNCTION:`

```
SUM,
```

and:

```
ARITH(SUB)
```

returns the `FUNCTION`

```
DIFF
```

and so on. Thus, we might add two numbers with:

```
ARITH(ADD)(3,4)
```

and so on. This is illegal in many imperative languages because it is not possible to construct functions of type ‘function’.

As we shall see, the ability to manipulate functions as values gives functional languages substantial power and flexibility.

1.8. Origins of functional languages

Functional programming has its roots in mathematical logic. Informal logical systems have been in use for over 2000 years but the first modern formalisations were by Hamilton, De Morgan and Boole in the mid 19th century. Within their work we now distinguish the **propositional** and the **predicate calculus.**

The propositional calculus is a system with `true` and `false` as basic values and with `and, or, not` and so on as basic operations. Names are used to stand for arbitrary truth values. Within propositional calculus it is possible to **prove** whether or not an arbitrary expression is a **theorem,** always true, by starting with **axioms,** elementary expressions which are always true, and applying **rules of inference** to construct new theorems from axioms and existing theorems. Propositional calculus provides a foundation for more powerfull logical systems. It is also used to describe digital electronics where on and off signals are represented as `true` and `false,` and electronic circuits are represented as logical expressions.

The predicate calculus extends propositional calculus to enable expressions involving non-logical values like numbers or sets or strings. This is through the introduction of **predicates** to generalise logical expressions to describe properties of non-logical values, and **functions** to generalise operations on non-logical values. It also introduces the idea of

quantifiers to describe properties of sequences of values, for example all of a sequence having a property, **universal quantification**, or one of a sequence having a property, **existential quantification**. Additional axioms and rules of inference are provided for quantified expressions.

Predicate calculus may be applied to different problem areas through the development of appropriate predicates, functions, axioms and rules of inference. For example, number theoretic predicate calculus is used to reason about numbers. Functions are provided for arithmetic and predicates are provided for comparing numbers. Predicate calculus also forms the basis of logic programming in languages like Prolog and of expert systems based on logical inference.

Note that within the propositional and predicate calculi, associations between names and values are unchanging and expressions have no necessary evaluation order.

The late 19th century also saw Peano's development of a formal system for **number theory**. This introduced numbers in terms of `0` and the successor function, so any number is that number of successors of `0.` Proofs in the system were based on a form of **induction** which is akin to recursion.

At the turn of the century, Russell and Whitehead attempted to derive mathematical truth directly from logical truth in their "Principia Mathematica." They were, in effect, trying to construct a logical description for mathematics. Subsequently, the German mathematician Hilbert proposed a 'Program' to demonstrate that 'Principia' really did describe totally mathematics. He required proof that the 'Principia' descripton of mathematics was **consistent,** did not allow any contradictions to be proved, and **complete,** allowed every true statement of number theory to be proved. Alas, in 1931, Godel showed that any system powerful enough to describe arithmetic was necessarily incomplete.

However, Hilbert's 'Program' had promoted vigorous investigation into the **theory of computability**, to try to develop formal systems to describe computations in general. In 1936, three distinct formal approaches to computability were proposed: Turing's **Turing machines,** Kleene's **recursive function theory** (based on work of Hilbert's from 1925) and Church's **λ calculus.** Each is well defined in terms of a simple set of primitive operations and a simple set of rules for structuring operations. Most important, each has a proof theory.

All the above approaches have been shown formally to be equivalent to each other and also to generalised von Neumann machines - digital computers. This implies that a result from one system will have equivalent results in equivalent systems and that any system may be used to model any other system. In particular, any results will apply to digital computer languages and any may be used to describe computer languages. Contrariwise, computer languages may be used to describe and hence implement any of these systems. Church hypothesised that all descriptions of computability are equivalent. While **Church's thesis** cannot be proved formally, every subsequent description of computability has been proved to be equivalent to existing descriptions.

An important difference between Turing machines, and recursive functions and λ calculus is that the Turing machine approach concentrated on computation as mechanised symbol manipulation based on assignment and time ordered evaluation. Recursive function theory and λ calculus emphasised computation as structured function application. They both are evaluation order independent.

Turing demonstrated that it is impossible to tell whether or not an arbitrary Turing machine will halt - the **halting problem** is **unsolvable** This also applies to the λ calculus and recursive function theory, so there is no way of telling if evaluation of an arbitrary λ expression or recursive function call will ever terminate. However, Church and Rosser showed for the λ calculus that if different evaluation orders do terminate then the results will be the same. They also showed that one particular evaluation order is more likely to lead to termination than any other. This has important implications for computing because it may be more efficient to carry out some parts of a program in one order and other parts in another. In particular, if a language is evaluation order independent then it may be possible to carry out program parts in parallel.

Today, λ calculus and recursive function theory are the backbones of functional programming but they have wider applications throughout computing.

1.9. Computing and theory of computing

The development of electronic digital computers in the 1940s and 1950s led to the introduction of high level languages to simplify programming. Computability theory had a direct impact on programming language design. For example, ALGOL 60, an early general purpose high level language and an ancestor of Pascal, had recursion and the λ calculus based call by name parameter passing mechanism

As computer use exploded in the 1960s, there was renewed interest in the application of formal ideas about computability to practical computing. In 1963, McCarthy proposed a mathematical basis for computation which was influenced by λ calculus and recursive function theory. This culminated in the LISP(LISt Processing) programming language. LISP is a very simple language based on recursive functions manipulating lists of words and numbers. Variables are not typed so there are no restrictions on which values may be associated with names. There is no necessary distinction between programs and data - a LISP program is a list. This makes it easy for programs to manipulate programs. LISP was one of the first programming languages with a rigorous definition but it is not a pure functional language and contains imperative as well as functional elements. It has had a lot of influence on functional language design and functional programming techniques. At present, LISP is used mainly within the Artificial Intelligence community but there is growing industrial interest in it. McCarthy also introduced techniques for proving recursive function based programs.

In the mid 1960s, Landin and Strachey both proposed the use of the λ calculus to model imperative languages. Landin's approach was based on an **operational** description of the λ calculus defined in terms of an **abstract interpreter** for it - the SECD machine. Having described the λ calculus, Landin then used it to construct an abstract interpreter for ALGOL 60. (McCarthy had also used an abstract interpreter to describe LISP). This approach formed the basis of the Vienna Definition Language(VDL) which was used to define IBM's PL/1. The SECD machine has been adapted to implement many functional languages on digital computers. Landin also developed the pure functional language ISWIM which influenced later languages.

Strachey's approach was to construct descriptions of imperative languages using a notation based on λ calculus so that every imperative language construct would have an equivalent function **denotation.** This approach was strengthened by Scott's **lattice theoretic** description for λ calculus. Currently, denotational semantics and its derivatives are used to give formal definitions of programming languages. Functional languages are closely related to λ calculus based semantic languages.

Since LISP many partially and fully functional languages have been developed. For example, POP-2 was developed in Edinburgh University by Popplestone & Burstall in 1971 as an updated LISP with a modern syntax and a pure functional subset. It has led to POP11 and and to POPLOG which combines POP11 and Prolog. SASL, developed by Turner at St Andrews University in 1974, is based strongly on λ calculus. Its offspring include KRC and Miranda, both from the University of Kent. Miranda is used as a general purpose language in research and teaching. Hope was developed by Burstall at Edinburgh University in 1980 and is used as the programming language for the ALICE parallel computer. ML was developed by Milner at Edinburgh University in 1979 as a language for the computer assisted proof system LCF. Standard ML is now used as a general purpose functional language. Like LISP, it has imperative extensions.

Interest in functional languages was increased by a paper by Backus in 1977. He claimed that computing was restricted by the structure of digital computers and imperative languages, and proposed the use of **Functional Programming(FP)** systems for program development. FP systems are very simple, consisting of basic atomic objects and operations, and rules for structuring them. They depend strongly on the use of functions which manipulate other functions as values. They have solid theoretical foundations and are well suited to program proof and refinement. They also have all the time order independence properties that we considered above. FP systems have somewhat tortuous syntax and are not as easy to use as other functional languages. However, Backhus' paper has been very influential in motivating the broader use of functional languages.

In addition to the development of functional languages, there is considerable research into formal descriptions of programming languages using techniques related to λ calculus and recursive function theory. This is both theoretical, to develop and extend formalisms and proof systems, and practical, to form the basis of programming methodologies and language implementations. Major areas where computing theory has practical applications include the precise specification of programs, the development of prototypes from specifications and the verification that implementations correspond to specifications. For example, the Vienna Development Method(VDM), Z and OBJ approaches to

program specification all use functional notations and functional language implementations are used for prototyping. Proof techniques related to recursive function theory, for example constructive type theory, are used to verify programs and to construct correct programs from specifications.

1.10. λ calculus

The λ calculus is a suprisingly simple yet powerful system. It is based on function **abstraction,** to generalise expressions through the introduction of names, and function **application,** to evaluate generalised expressions by giving names particular values.

The λ calculus has a number of properties which suit it well for describing programming languages. First of all, abstraction and application are all that are needed to develop representations for arbitrary programming language constructs. Thus, λ calculus can be treated as a universal machine code for programming languages. In particular, because the λ calculus is evaluation order independent it can be used to describe and investigate the implications of different evaluation orders in different programming languages. Furthermore, there are well developed proof techniques for the λ calculus and these can be applied to λ calculus language descriptions of other languages. Finally, because the lambda calculus is very simple it is relatively easy to implement. Thus a λ calculus description of language can be used as a prototype and run on a λ calculus implementation to try the language out.

1.11. Summary of rest of book

In this book we are going to use λ calculus to explore functional programming. Pure λ calculus does not look much like a programming language. Indeed, all it provides are names, function abstraction and function application. However, it is straightforward to develop new language constructs from this basis. Here we will use λ calculus to construct step by step a compact, general purpose functional programming notation.

In chapter 2, we will look at the pure λ calculus, examine its syntax and evaluation rules, and develop functions for representing pairs of objects. These will be used as building blocks in subsequent chapters. We will also introduce simplified notations for λ expressions and for function definitions.

In chapter 3, we will develop representations for boolean values and operations, numbers and conditional expressions.

In chapter 4, we will develop representations for recursive functions and use them to construct arithmetic operations.

In chapter 5, we will discuss types and introduce typed representations for boolean values, numbers and characters. We will also introduce notations for case definitions of functions.

In chapter 6, we will develop representations for lists and look at linear list processing.

In chapter 7, we will extend linear list processing techniques to composite values and look at nested structures like trees.

In chapter 8, we will discuss different evaluation orders and termination.

In chapter 9, we will look at functional programming in Standard ML using the techniques developed in earlier chapters.

In chapter 10, we will look at functional programming in LISP using the techniques developed in earlier chapters.

1.12. Notations

In this book, different typefaces are used for different purposes.

Text is in Times Roman. **New terms and important concepts are in bold Times Roman**.

`Programs and definitions are in Courier`.

Greek letters are used in naming λ calculus concepts:

α - alpha
β - beta
λ - lambda
η - eta

Syntactic constructs are defined using BNF **rules.** Each rule has a **rule name** consisting of one or more words within angle brackets `<` and `>`. A rule associates its name with a **rule body** consisting of a sequence of **symbols** and rule names. If there are different possible rule bodies for the same rule then they are separated by `|`s.

For example, binary numbers are based on the digits `1` and `0`:

```
<digit> ::= 1 | 0
```

and a binary number may be either a single digit or a digit followed by a number:

```
<binary> ::= <digit> | <digit> <binary>
```

1.13. Summary

In this chapter we have discussed the differences between imperative and functional programming and seen that:

- imperative languages are based on assignment sequences whereas functional langugaes are based on nested function calls
- in imperative languages, the same name may be associated with several values whereas in functional languages a name is only associated with one value
- imperative languages have fixed evaluation orders whereas functional languages need not
- in imperative languages, new values may be associated with the same name through command repetition whereas in functional languages new names are associated with new values through recursive function call nesting
- functional languages provide explicit data structure representations
- in functional languages, functions are values

We have also seen that:

- functional languages originate in mathematical logic and the theory of computing, in recursive function theory and λ calculus

2. LAMBDA CALCULUS

2.1. Introduction

In this chapter we are going to meet the λ calculus, which will be used as a basis for functional programming in the rest of the book.