Preface

Overview

This book aims to provide a gentle introduction to functional programming. It is based on the premises that functional programming provides pedagogic insights into many aspects of computing and offers practical techniques for general problem solving.

The approach taken is to start with pure λ calculus, Alonzo Church's elegent but simple formalism for computation, and add syntactic layers for function definitions, booleans, integers, recursion, types, characters, lists and strings to build a highish level functional notation. Along the way, a variety of topics are discussed including arithmetic, linear list and binary tree processing, and alternative evaluation strategies. Finally, functional programming in Standard ML and COMMON LISP, using techniques developed throughout the book, are explored.

The material is presented sequentially. Each chapter depends on previous chapters. Within chapters, substantial use is made of worked examples. Each chapter ends with exercises which are based directly on ideas and techniques from that chapter. Specimen answers are included at the end of the book.

Readership

This book is intended for people who have taken a first course in an imperative programming language like Pascal, FORTRAN or C and have written programs using arrays and sub-programs. There are no mathematical prerequisites and no prior experience with functional programming is required.

The material from this book has been taught to third year undergraduate Computer Science students and to post graduate Knowledge Based Systems MSc students.

Approach

This book does not try to present functional programming as a complete paradigm for computing. Thus, there is no material on the formal semantics of functional languages or on transformation and implementation techniques. These topics are ably covered in other books. By analogy, one does not buy a book on COBOL programming in anticipation of chapters on COBOL's denotational semantics or on how to write COBOL compilers.

However, a number of topics which might deserve more thorough treatment are ommited or skimmed. In particular, there might be more discussion of types and typing schemes, especially abstract data types and polymorphic typing, which are barely mentioned here. I feel that these really deserve a book to themselves but hope that their coverage is adequate for what is primarily an introductory text. There is no mention of mutual recursion which is conceptually simple but technically rather fiddly to present. Finally, there is no discussion of assignment in a functional context.

Within the book, the λ calculus is the primary vehicle for developing functional programming. I was trained in a tradition which saw λ calculus as a solid base for understanding computing and my own teaching experience confirms this. Many books on functional programming cover the λ calculus but the presentation tends to be relatively brief and theoretically oriented. In my experience, students whose first language is imperative find functions, substitution and recursion conceptually difficult. Consequently, I have given a fair amount of space to a relatively informal treatment of these topics and include many worked examples. Functional afficionados may find this somewhat tedious. However, this is an introductory text.

The functional notation developed in the book does not correspond to any one implemented language. One of the book's objectives is to explore different approaches within functional programming and no single language encompasses these. In particular, no language offers different reduction strategies.

The final chapters consider functional programming in Standard ML and COMMON LISP. Standard ML is a modern functional language with succinct syntax and semantics based on sound theoretical principles. It is a pleasing language to program in and its use is increasing within education and research. SML's main pedagogic disadvantage is that it lacks normal order reduction and so the low-level λ calculus representations discussed in earlier chapters cannot be fully investigated in it.

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.