Introduction to Programming

Aims and Objectives

To give students the grounding that makes it possible to approach problems and solve them on the computer.

The aspects covered range across:

— Modelling a given problem domain appropriately

— Designing a solution

— Implementing the solution in a high-level programming language

Contents

Two paradigms are used as vehicles to carry the ideas and execute practicals for this course – the functional and the imperative.

1. The Functional Paradigm

The central issue here is to be able to use the computer as a high-level tool for problem solving. The paradigm conveyed may be simply expressed as:

> *Programs are functions from input to output.*
> *Functions are defined in terms of expressions of arbitrary type.*
> *Understanding functions requires nothing more than substitution.*
> *Programs compute by rewriting the head of a function to the body until nothing can be further rewritten.*

A modern non-strict functional language with a polymorphic type system is the medium for this part. The currently used language is the internationally standardized language, Haskell [].

Important ideas that are to be covered include:

1.1 Standard Constructs

Function and type definition, block structure.
Guarded equations, pattern matching.
Special syntax for lists, comprehension.

1.2 Standard Data Types

Fluency is to be achieved in the standard data-types: numbers, boolean, character, tuple, list.
List programs in an algebraic vein.
Lists in the context of general collections – sets, bags, lists, tuples. (MF?)

1.3 λ-calculus

A direct way for denoting functions.

1.4 First-Classness

All values are uniformly treated and conceptualized.

1.5 Higher Order Functions

Use of first-class, higher-order functions to capture large classes of computations in a simple way. An understanding of the benefits that accrue – modularity, flexibility, brevity, elegance.

1.6 Laziness

The use of infinite data-structures to separate control from action.

1.7 Type discipline

Polymorphism: The use of generic types to model and capture large classes of data-structures by factorizing common patterns.

Inference: The types of expressions may be determined by simple examination of the program text. Understanding such rules.

User-defined types: User-defined types as

— a means to model

— a means to extend the language

— a means to understand the built-in types in a uniform framework.

Concrete types: Types are concrete. i.e. values that are read or written by the system correspond directly to the abstractions that they represent. More specifically, unlike abstract types which are defined in terms of admissable operations, concrete types are defined by **directly** specifying the set of possible values.

1.8 Recursion

Recursive definitions as

— a means of looping indefinitely

— a structural counterpart to recursive data-type definitions

— a means to understand induction in a more general framework than just for natural numbers

1.9 Operational Semantics

Functional programs execute by rewriting.
λ-calculus as a rewriting system
Reduction, confluence, reasons for preferring normal-order reduction.

1.10 Type Classes

Values are to types as types are to classes. Only elementary ideas.

2. The Imperative Paradigm

The imperative paradigm is smoothly introduced as follows:

Worlds	The Timeless World	World of Time
Domain	Mathematics	Programming
Syntax	Expressions	Statements
Semantics	Values	Objects
Explicit	Data Structures	Control Structure
Think with	Input-Output relations	State Change
Abstractions	Functions	Procedures
Relation	Denote programs $\Rightarrow$	$\Leftarrow$ Implement functions

In the following we spell out some of the points of how FP translates into Imp P. The examples may be analogized from say how one would teach assembly language to someone who undestands structured programming.

2.1 Semantic relations

The central relation is that imperative programming's denotational semantics is FP, FP's operational semantics is imperative programming.

2.2 Operational Thinking

IN FP data-dependency implicitly determines sequencing whereas in Imp P it is done explicitly. Advantages and disadvantages of operational thinking.

2.3 Environment

In imperative programming there is a single implicit environment – memory. In FP there are multiple environments; which could be explicit to the point of first-classness (the value of variables bound in environments could be other environments). Use of environments to model data abstraction, various object frameworks, module systems.

2.4 Semi-Explicit Continuation

Explicit in the sense that goto-labels can be dealt with first-classly (as in assembly), but not explicit in the sense of capturing the entire future of a computation – dynamic execution of a code block may be

’concave’.

2.5 Recursion iteration equivalence

General principles as well as scheme semantics of tail-recursion.

2.6 Type Issues

Monomorphic, polymorphic and latent typing: translating one into another.

2.7 Guile

A variety of vehicles have been used for the imperative paradigm, eg. Pascal, C, Java, Tcl. The current choice is Scheme in the guile dialect because it gives a full support for the functional and the imperative paradigm. In fact Guile has been chosen over C because the single data structure in guile – s-expressions – is universal (aka XML) and thus imperative and functional thinking do not quarrel with data-structure issues.

Orthogonal kinds of abstractions, which are usually considered ‘advanced’, such as functional, higher-order functional, object-oriented, stream-based, data-driven, language extensions via eval, via macros, via C can be easily demonstrated. In fact, once guile has been learnt, it is much faster to pick up C in the subsequent semester.

Note: In addition to being a system programming and general purpose language Guile is also a scripting, extension and database programming language because it is the flagship language for FSF (The free software foundation).

3. Bibliography

1. Bird and Wadler; Introduction to Functional Programming; Prentice Hall
2. Bird; Algebra of Programs; Prentice Hall
3. Abelson and Sussman; Structure and Interpretation of Computer Programs; MIT Press
4. Friedmann and Haynes; Scheme and the Art of Programming; MIT Press
5. Thomas Myers; Equations Models and Programs; Prentice Hall
6. N Wirth; Algorithms + Data Structures = Programs
7. Reade; Functional Programming
8. Bornat; Programming from First Principles; Prentice Hall
9. Hall and Donnell; Discrete Maths with a computer; Springer Verlag
10. Guile Reference Manual; www.gnu.org