G6021 Comparative Programming

Ian Mackie

Organisation

Please check Canvas for information each week.

- Lectures
- Labs (check Sussex Direct for times)
- Extra sessions/Helpdesk
- Exam, 15 credits

To get the most out of this module:

- Keep up-to-date with the material and lab exercises
- Check the solutions of the previous lab each week
- Read the extra material on web page
- Prepare for the labs

You will enjoy this module whether you like it or not!

Objectives and Learning Outcomes

- Understand the role of programming languages in the software development process.
- Describe the main programming paradigms.
- Identify the main components of a programming language.
- Describe the main implementation techniques for programming languages.
- Distinguish between different kinds of syntactic and semantic descriptions.
- Most importantly: introduces you to the basic techniques of declarative and functional programming.

Introduction

- Programming languages are tools for writing software.
- The languages that exist today are the result of an evolution process which is likely to continue in the future.
- We will study the main concepts in programming languages and paradigms of programming in order to:
  - Be able to choose the most suitable language for each application.
  - Increase our ability to learn new languages.
  - Design new languages (programming languages, user-interfaces for software systems, etc.).

Programming Languages and Software Engineering

- Programming languages are used in several phases in the software development process: in the implementation phase, obviously, but also in the design phase (decomposition into modules etc.).

- A design method is a guideline for producing a design (e.g. *top-down design, object-oriented design*). Some languages provide better support for some design methods than others.

PL and SE, continued

- Some of the first programming languages, such as Fortran, did not support any specific design method.
- Later languages were designed to support a specific design method: e.g. Pascal supports top-down programming development and structured programming, Lisp and Haskell support functional design, Prolog supports symbolic and logical reasoning, Smalltalk and Java support object-oriented design and programming. (To name but a very few...)

To summarise:

- If the design method is not compatible with the language the programming effort increases.
- When they are compatible, the design abstractions can easily be mapped into program components.

Programming Paradigms

A programming language may enforce a particular style of programming, called a *programming paradigm*.

- **Imperative Languages**: Programs are decomposed into *computation steps* and routines are used to modularise the program. Typical features include: variables, assignment, iteration in the form of loops (For-loop, While-loop, recursion) and procedures. Fortran, Pascal and C are examples. Java has an imperative subset.

- **Functional Languages**: Based on the mathematical theory of functions. The focus is on *what* is computed rather than *how* it should be computed. They emphasise the use of expressions which are evaluated by simplification. Haskell, SML, Caml, Clean, are functional languages. Exercise: what does *referential transparency* mean?

Programming Paradigms, continued

- **Object-oriented Languages**: Emphasise the definition of hierarchies of objects. Smalltalk, Java, are object-oriented languages.
- **Logic Languages**: Programs describe a problem rather than defining an algorithmic implementation. The most well-known logic programming language is Prolog. Constraint logic programming languages combine logic programming and constraint-solving.

Definition of a programming language

A language has three main components:

1. *Syntax:* defines the form of programs; how expressions, commands, declarations are built and put together to form a program.
2. *Semantics:* gives the *meaning* of programs; how they behave when they are executed.
3. *Implementation:* a software system that can read a program and execute it in a machine, plus a set of tools (editors, debuggers, etc).

Implementing a Programming Language

We are concerned with *high level* programming languages which are (more or less) machine independent. Such languages can be implemented by:

- **Compiling** programs into machine language,
- **Interpreting** programs,
- A **Hybrid Method** which combines compilation and interpretation.

Syntax

The *syntax* is concerned with the form of programs. Given by:

- an *alphabet*: the set of characters that can be used,
- a *set of rules*: indicating how to form expressions, commands, etc.

We have to distinguish between *concrete syntax* and *abstract syntax*.

- Concrete Syntax: describes which chains of characters are well-formed programs.
- Abstract Syntax: describes the syntax trees, to which a semantics is associated.

To specify the syntax of a language we use *grammars*.
A grammar is given by:

- An alphabet $V = V_T \cup V_{NT}$.
- Rules
- Initial Symbol.

Example: Arithmetic expressions

Concrete syntax:

$$
\begin{array}{lcl}
\mathit{Exp} & ::= & \mathit{Num} \mid \mathit{Exp}\ \mathit{Op}\ \mathit{Exp} \\
\mathit{Op} & ::= & + \mid - \mid * \mid \mathit{div} \\
\mathit{Num} & ::= & \mathit{Digit} \mid \mathit{Digit}\ \mathit{Num} \\
\mathit{Digit} & ::= & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{array}
$$

Problem: Ambiguity. How is $1 - 2 - 3$ read?

Abstract Syntax:

$$
\begin{array}{lcl}
e & ::= & n \mid \mathit{op}(e, e) \\
\mathit{op} & ::= & + \mid - \mid * \mid \mathit{div}
\end{array}
$$

This grammar defines trees, not strings.

Remarks:

- The abstract syntax is not ambiguous.
- We will always work with the abstract syntax of the language and study the semantics of the main constructs of programming languages.

Semantics

- The *semantics* of a language defines the meaning of programs that is, how they behave when they are executed on a computer.
- Different languages have different syntax and different semantics for similar constructs, but variations in syntax are often superficial. It is important to appreciate the differences in meaning of apparently similar constructs.

There are two kinds of semantics:

- *Static Semantics* (for example typing)
- *Dynamic Semantics* (meaning of the program)

Static Semantics: Typing.

- The goal is to detect (before the actual execution of the program) programs that are syntactically correct but will give errors during execution.
- Example: `true && 37`
- We will study type systems later in the module.

Dynamic Semantics (or just Semantics)

Specifies the meaning of programs.
Informal definitions, often given by English explanations in language manuals, are often imprecise and incomplete. Formal semantics are important for:

- the implementation of the language: the behaviour of each construct is specified, providing an abstraction of the execution process which is independent of the machine.
- programmers: a formal semantics provides tools or techniques to reason about programs and prove properties of programs.
- language designers: a formal semantics allows to detect ambiguities in the constructs and suggests improvements and new constructs (e.g. influence of the study of the λ-calculus in the design of functional languages).

However, formal semantics descriptions can be complex, so usually only a part of the language is formally defined.

Styles of Semantics

- Denotational Semantics: The meaning of expressions (and in general, the meaning of the constructs in the language) is given in an abstract, mathematical way (using a mathematical model for the language). The semantics describes the *effect* of each construct.

- Axiomatic Semantics: Uses axioms and deduction rules in a specific logic. Predicates or assertions are given before and after each construct, describing the constraints on program variables before and after the execution of the statement (*precondition, post-condition*).

Styles of Semantics, continued

- Operational Semantics: The meaning of each construct is given in terms of computation steps. The behaviour of the program during execution can be described using a *transition system* (abstract machine, structural operational semantics).

Remarks:

- Each style has its advantages, they are complementary.
- Operational semantics is very useful for the implementation of the language and for proving correctness of compiler optimisations.
- Denotational semantics and axiomatic semantics are useful to reason and prove properties of programs.

This module

The rest of the module:

1. A study of the main components of imperative, functional, object, and logic based languages. We will look at foundations of these languages, as well as practical aspects.
2. Study each paradigm as a model of computation and a programming language.
3. Illustrate some of the most important applications of formal methods to date (type checking).
4. Practical work (labs) will be heavily biased towards a functional language.