

Introduction

Lloyd Allison

Bernd Meyer

*Computer Science and Software Engineering,
Monash University, Semester 2, 2005.*

Our Requirements to be a Programming Language

- **Universal:** If a problem is computable, then it must be programmable in that language. Said to be “Turing complete”. This requires:
 - recursion and/or
 - iteration.
- **Implementable in a computer:** Every well formed program can be executed.

Why Study Programming Languages?

- Programming languages are at the core of computer science—they are the principal tool of the programmer.
- Increased vocabulary of programming constructs and capacity to express ideas:

The limits of my language are the limits of my world

— Ludwig Wittgenstein.

A good programming language is a conceptual universe for thinking about programming

— Alan Perlis.

- Better understanding of the significance of implementation and how to best exploit existing languages.
- Allows informed choice of appropriate programming language.
- Increased ability to learn future programming languages.
- Improved ability to design new computer languages and application interfaces.
- Good example of the synergy between theory and practice.

Why Study Programming Languages? (Cont.)

The ideas do not just apply to programming languages but also to any language for specifying data and information processing (i.e. arguably to the input – output of any program!).

This has become even more important with the advent of the web and meta-mark up languages like XML and style sheet languages like XSL.

What Programming Languages Provide

- An underlying computation model, e.g. the von Neumann architecture, the lambda calculus, predicate calculus, concurrent evaluation, etc..
- Data types and operations, e.g. reals, integers, lists and records.
- Abstraction facilities, e.g. functions or procedures, abstract data types (classes).
- Checking and enforcement, e.g. type checking, array-bounds checking, protection of private data.

9

Evolution of Software Architectures

Programming language development does not occur in a vacuum. It is affected by the underlying hardware, the operating environment and the application domains.

There have been three main eras:

- Mainframe Era (1940s–1970s) First batch, then interactive processing. (Although the mainframe is far from dead in 2005.)
- Personal Computer Era (1970s– 1980s) GUIs, also embedded system environments. Rise of OO languages.
- Networking Era (from late 1980s) Security, distributed computing, internet programming and document processing, untrained programmers. Java, scripting languages.

10

Application Domains

1960s

Application	Major languages	Other languages
Business	COBOL	Assembler
Scientific	FORTRAN	Algol, BASIC, APL
System AI	Assembler LISP	JOVIAL, Forth SNOBOL

Today

Application	Major languages	Other languages
Business	COBOL, C++, spreadsheets, Java	C, PL/1, 4GLs
Scientific	FORTRAN, C, C++, Java	BASIC
System AI	C, C++, Java LISP, Prolog	Ada, Modula
Publishing	HTML, XML, TeX, Postscript, PDF, word processing	
Process	UNIX shell, TCL, Perl, Java/ECMA script	AWK, Marvel, Python
New paradigms	ML, Haskell, Smalltalk	Eiffel

11

What Makes a Good Language?

- Clarity, simplicity and unity: a small number of different orthogonal concepts with simple rules for combining them.
- Naturalness for the application.
- Support for abstraction.
- Ease of program verification.
- Programming environment.
- Cost of use Cost includes:
 - cost of program execution (efficiency)
 - cost of program translation
 - cost of program creation
 - cost of program maintenance and modification.
- Portability

Of course simply being better doesn't mean that a language will be adopted!

12

Syntax versus semantics.

Colorless green ideas sleep furiously — Noam Chomsky

Birds is garden full — Freya Beyer (age 3)

The syntax of a language specifies how elements in the language combine to form valid “sentences.”

The semantics of a language specifies what the elements in the language mean.

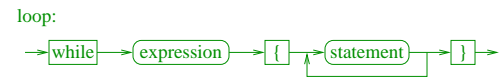
E.g. $I = I + 1$ means very different things in C and in Prolog.

Programming language syntax is usually specified using a variant of context-free grammars.

One common notation is BNF (Backus-Naur Form). For example,

$$\langle loop \rangle ::= \text{while} \langle expr \rangle \{ \langle statement \rangle^+ \}$$

which has the corresponding syntax diagram



Programming language semantics can be specified using:

- an operational semantics, i.e. a simplified execution model.
- a denotational semantics, i.e. in terms of mathematical functions.
- an axiomatic semantics, i.e. in terms of mathematical logic.

Implementation

Varies between interpretation and compilation.

A program can be run using an interpreter: a program which executes the program directly (no executable is created). An interpreter acts as a software simulation of a computer which understand the high-level program constructs rather than machine instructions. E.g. Basic, HTML.

Programs can be compiled or translated into machine code which is directly executed on the computer. Eg. FORTRAN, C, C++.

Often a hybrid abstract machine-based approach is used in which programs are compiled into lower-level abstract machine code which is then interpreted. Eg. Prolog, Java.

Layers of Virtual Machines

Applications can be understood in terms of a hierarchy of virtual machines.

Each layer has its own language(s) and programs are interpreted or compiled into the language of the layer below.

For example consider a web application written in Java:

Web Application (implemented in Java: i.e. an interpreter written in Java)
Java (compiled into Java Abstract Machine)
Java Abstract Machine (interpreter written in C)
C (compiled into Assembly language)
Assembly language (compiled into machine code)
Machine code (interpreted by the firmware computer)
Firmware (interpreted by microcode executed by the actual computer)
Actual Computer Microcode (implemented by physical devices)

Functional Programming Languages

Functional languages are so-called because pure functions are the basic building blocks from which programs are constructed.

Functional languages are inspired by mathematical functions.

Execution is based on function application, not assignment to memory locations.

Functions are first-class objects and may be higher-order and/or recursive.

1930s Alonzo Church developed the λ -calculus.

1958 John McCarthy developed LISP.

1965 Peter Landin developed ISWIM.

1980s Robin Milner developed ML.

early 1980s David Turner developed Miranda.

late 1980s Paul Hudak et al developed Haskell.

Many ideas pioneered with functional languages have moved into more traditional languages:

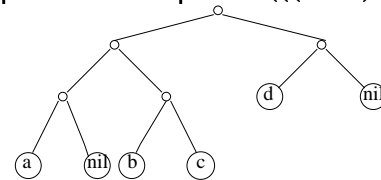
- Complex types
- Higher-order functions
- Automatic memory management (garbage collector)

17

Lisp (List processing)

Lisp is typeless: its only has S-expressions.

This tree represents the S-expression $((a.nil).(b.c)).(d.nil)$



The following function returns the intersection of two lists:

```
(define '(
  (intersect
    (lambda (m n)
      (cond
        ((null m) nil)
        ((member (car m) n)
         (cons (car m)
               (intersect (cdr m) n)
                )
        )
      )
    )
  )
  (t (intersect (cdr m) n)
    )
  )
))
```

18

The programming language ML

ML is a widely used functional programming language.

(Standard) ML was originally developed by Robert Harper, Dave MacQueen and Robin Milner for theorem proving.

It is now recognised as a simple, elegant high-level language particularly useful for symbolic computation.

ML has the following characteristics:

- First-class higher-order functions.
- Strict functions (call by value), invoked by pattern matching.
- Polymorphic static typing.
- Implicit memory management.
- Referential transparency.
- Module system.
- Exception handling.

19

Running ML

We shall use ML97 and the SML/NJ (Standard ML of New Jersey) implementation.

SML/NJ is installed on ra-clay.cc.monash.edu.au and the Linux machines.

You can download a SML/NJ for your home computer from

<http://www.smlnj.org/>

20

Running ML programs

ML is interactive. You can enter expressions followed by semicolon:

```
- 2+2;
```

ML evaluates the expression returning the value and its type.

```
val it = 4 : int
```

Note that `it` is a special variable which is set to the value of any expression typed in interactive mode.

Expressions can take up more than one line:

```
- 3.9 -  
= 3.2;  
val it = 0.7 : real
```

So if you get the prompt `'='` it means that you haven't finished typing your expression.

If you have a large (or even small) program to run repeatedly, you don't want to type it again every time. Suppose I have an ML program in the file `prog.ml`. I can load the program from the ML system by typing

```
- use "prog.ml";
```

Any pathname can appear in the string.

You leave the system by typing `control-D`.

Summary

- What are programming languages and why they were invented.
- Historical overview.
- Introduction to Functional Languages and ML.

Homework

- The languages PL/1 and Ada were designed to be universal programming languages. How well did they succeed? Give a possible explanation.
- Give two applications that C is suited to and two that it is unsuitable for.
- Use ML to evaluate $3.0 + 4.0$ and $3 + 4.0$. Explain the answer.