

Introduction

Kim Marriott

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.

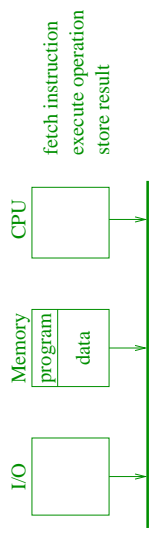
Why Study Programming Languages? (Cont.)

The ideas do not just apply to programming languages but also to any language for specifying data and information processing.

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.

Why Programming Languages?

Modern computers were first developed at Princeton in the 1940's by von Neumann et al.



Low-level languages were used to program them:

- **Machine language**

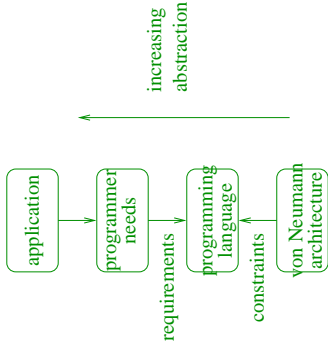
```
0010101010100001
0001001111100011
1011110011010101
```

- **Assembly language**

```
LOAD I
ADD J
STORE K
```

Both are an error-prone and clumsy way of specifying
 $K := I+J$.

The First Programming Languages



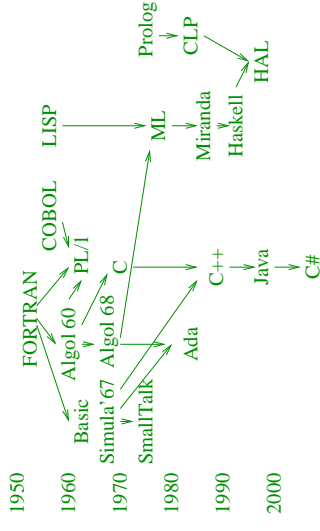
The first successful programming language was **FORTRAN** (FORmula TRANslation) developed by a team at IBM lead by John Backus in the mid 1950s.

FORTRAN was designed for numerical computing and introduced **symbolic expressions** and **sub-programs**.

Benefits of programming languages:

- New users and programs.
 - Portability.
 - Readability.
- However, **efficient** implementation was crucial to the acceptance of FORTRAN.

The Tower of Babel



Apart from FORTRAN some of the most influential languages have been:

COBOL: Developed in 1959-60 by Grace Hopper for data processing. Introduced **data descriptions** and **file handling**.

ALGOL 60: Developed in 1958-60 for numerical computing. Introduced **block structure** and **recursion**.

LISP: Developed in 1956-62 by John McCarthy for symbolic computing and AI. It was the first **functional programming** language.

SIMULA 67: Developed in 1967 by Ole-Johan Dahl and Kristen Nygaard for simulation. It was the precursor to **object oriented programming** introducing **hierarchically organized classes, objects with dynamic lookup and concurrency**.

PROLOG: Developed in 1972 by Alain Colmerauer for AI. It was the first **logic programming** language.

What Programming Languages Provide

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

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.
- **Personal Computer Era (1970s– mid 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.

Application Domains

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

1960s

What Makes a Good Language?

- **Clarity, simplicity and unity:** minimum 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

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

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

Today

Programming Language Specification

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**.

I.e. $I + 1$ means very different things in C and Prolog.

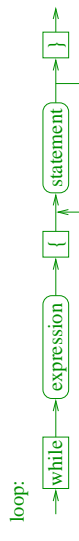
Programming Language Specification (Cont.)

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

One common notation is **BNF (Bachus-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**.

Programs can be run using an **interpreter**. This is a program which executes the program directly, acting as a software simulation of a computer which understands the high-level program constructs rather than machine instructions. Eg. Basic, HTML.

Programs can be **compiled** or translated into machine code which is directly executed on the computer. E.g. 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)

About CSE3322

The unit will be taught by Kim Marriott
Consultation 3:00pm–5:00pm on Fridays.

Lectures will be in S-7 on Tuesday at 4:00pm and on Friday
at 2:00pm in S-3.

There will be non-compulsory tutorial classes. These will be
held every fortnight starting in the second week of semester.

- Tuesday 12:00 noon Rm 109 Blg 19
- Thursday 1:00 pm Rm G13 Blg 19
- Friday 12:00 noon Rm G13 Blg 19

Syllabus of CSE3322

We will

- Overview the programming language landscape and the four major language paradigms: procedural, object-oriented, functional and logic.
- Examine functional programming in detail by learning ML.
- Examine the implementation of programming languages and compiler-generation tools, such as **flex** and **bison**, which facilitate this process.

Lectures will cover the following:

- programming language history, paradigms, concepts and issues (5 lectures);
- the functional programming language, ML (8 lectures);
- implementation of programming languages (11 lectures).

In addition there will be two revision lectures.

Textbooks

There are three main textbooks for this subject:

- J.D. Ullman. *Elements of ML Programming (2nd Ed.)*. Prentice Hall, 1998.
- D. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall, 1990.
- A. Aho, R. Sethi and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

Some material is also taken from

- M. Felleisen and D. Friedman. *The Little MLer*. MIT Press, 1997
- L.C. Paulson. *ML for the Working Programmer (2nd Ed.)*. Cambridge University Press, 1996.
- T.W.Pratt and M.V. Zelkowitz. *Programming Languages: Design and Implementation (4th Ed)*. Prentice Hall, 2001.
- J.C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2003.
- R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.
- R. Wilhelm and D. Maurer. *Compiler Design*. Addison-Wesley, 1995.

The lecture material will be loosely based on these books and will be available on the Web from the CSE3322 Courseware Page.

This lecture is based upon Pratt & Zelkowitz and Mitchell.

Assessment

There will be three assignments based on the course material.

- Assignment 1 (5%) - 12 noon Friday 16 August:
A small ML program.
- Assignment 2 (10%) - 12 noon Friday 12 September:
A larger ML program.
- Assignment 3 (15%) - 12 noon Friday 17 October:
A compiler for a simple language.

Students are referred to

http://www.csse.monash.edu.au/subjects/assign_submit.html

for information about assignment submission and plagiarism.

A 3-hour examination consisting of multiple choice questions and short answer questions will contribute the remaining 70% to the final grade.

Functional Programming Languages

Functional languages are so-called because **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**.

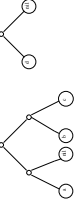
Many ideas pioneered with functional languages have moved into more traditional languages:

- **Complex types**
- **Higher-order functions**
- **Automatic memory management**

Lisp (List processing)

The programming language ML

Lisp is **typed**: There is only one datatype, the **S-expression**.



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

```
(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))  
        )  
      )  
    )  
  ))
```

This function returns the intersection of two lists.

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.**
- **Module system.**
- **Exception handling.**

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/>

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.