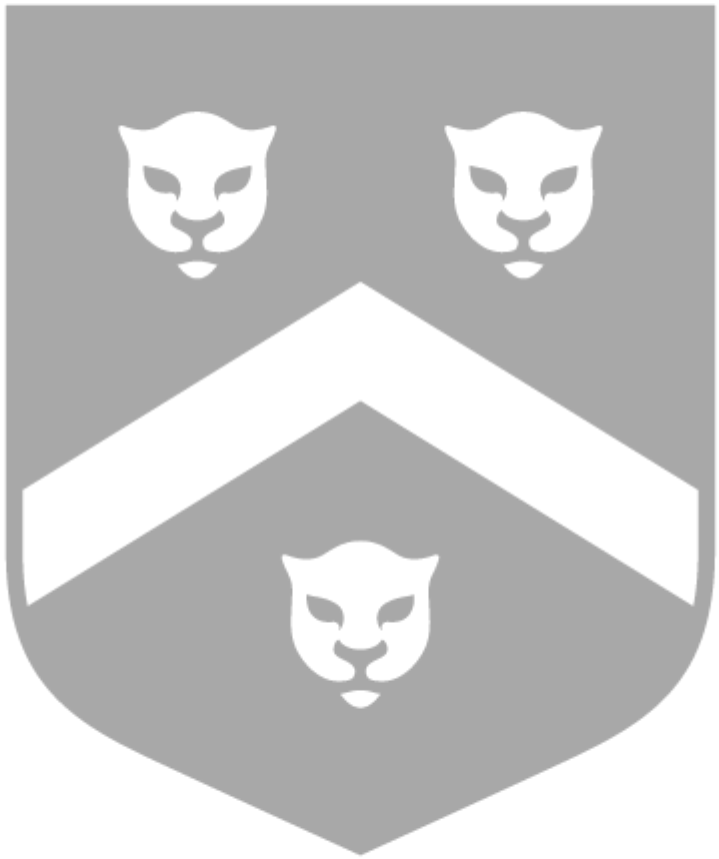
Intro to Programming Languages



Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

May 12, 2023

Programming Languages

Programming Languages Overview

- Purpose:
  - Discover language design successes and failures.
  - Discover how languages are designed and implemented.
- Several real languages will be programmed but:
  - Course is not intended to teach programming
  - Experience the key programming language elements that are common to or distinguish two classes of languages.
  - Assume you can already program in at least one object oriented language that uses Java style syntax.
  - C# used as a recent object oriented, threaded and networking language.
  - F# used as a very high-level language mainly for studying and implementing interpretive language.

Overview, continued

- A programming language is the problem-solving tool that computer science uses for human expression of computing solutions.
  - Ideas are expressed in a language.
- The Sapir-Whorf linguistic theory states that the structure of language defines the boundaries of thought.
  - New ideas often require new language, for example: algebra.
- A given language can impede or facilitate certain modes of thought.
- All programming languages are capable of solving any computable problem – computer languages are equivalent.
  - No programming language can prevent a problem solution.
  - A given language can subtly influence the class of solutions examined and the quality of a program.

Motivation

The Amazing Variety

- There are very many, very different languages
- A list that used to be posted occasionally on `comp.lang.misc` had over 2300 published languages in 1995
- Often grouped into four families:
  - Imperative
  - Functional
  - Logic
  - Object-oriented

Imperative Languages

Example (a factorial function in C)

```
int fact(int n) {
  int sofar = 1;
  while (n>0) sofar *= n--;
  return sofar;
}
```

- Hallmarks of imperative languages:
  - Assignment and side-effects
  - Iteration
  - Order of execution is critical

Functional Languages

Example (a factorial function in ML)

```
let rec factorial x =
  if x <= 0 then 1 else x * factorial (x-1)
```

- Hallmarks of functional languages:
  - Single-valued variables
  - Heavy use of recursion
  - Functions are first-class citizens, can be used as parameters, function results, etc.
  - Minimal use of assignments and side-effects

Another Functional Language

Example (a factorial function in Lisp)

```
(defun fact (x)
  (if (<= x 0) 1 (* x (fact (- x 1)))))
```

- Looks very different from ML
  - Fully-parenthesized, prefix syntax
- But ML and Lisp are closely related
  - Single-valued variables: no assignment
  - Heavy use of recursion: no iteration

Logic Languages

Example (a factorial function in Prolog)

```
fact(X,1) :-
  X =:= 1.
fact(X,Fact) :-
  X > 1,
  NewX is X - 1,
  fact(NewX,NF),
  Fact is X * NF.
```

- Hallmarks of logical languages:
  - Program expressed as rules in formal logic
  - Execution attempts to prove a result based upon rules

Object-Oriented Languages

Example (a factorial function object in C#)

```
public class Int {
    private int n;
    public Int(int n) { this.n = n; }
    public int N
    {
        get { return this.n; }
        set { this.n = value; }
    }
    public Int getFact()
    { return new Int(fact(n)); }
    private int fact(int n)
   {
        if (n <= 0) return 1;
        else return n * fact(n - 1);
    }
}
```

Object-Oriented Languages

- Hallmarks of object-oriented languages:
  - Usually imperative, plus. . .
  - Constructs to help programmers use “objects” – little bundles of data that know how to do things to themselves

Strengths and Weaknesses

- The different language groups show an advantage on different kinds of problems
- Decide for yourself at the end of the semester, after experimenting with them
- For now, one comment: don't jump to conclusions based on factorial!
  - Functional languages do well on such functions
  - Imperative languages, a bit less well
  - Logic languages, considerably less well
  - Object-oriented languages need larger examples

About Those Families

- There are many other language family terms (not exhaustive and sometimes overlapping)
  - Applicative, concurrent, constraint, declarative, definitional, procedural, scripting, single-assignment, . . .
- Some languages straddle families
- Others are so unique that assigning them to a family is pointless

Example: Forth Factorial

Example (a factorial function in Forth)

```
: FACTORIAL
   1 SWAP BEGIN ?DUP WHILE TUCK * SWAP 1- REPEAT ;
```

- A stack-oriented language
- Postscript language used by printers is similar
- Could be called *imperative*, but has little in common with most imperative languages

Example: APL Factorial

Example (a factorial function in APL)

FACTORIAL$\leftarrow \{\times/\iota X\}$

- An APL expression that computes X's factorial
- Expands X into a vector of the integers 1..X, then multiplies them all together
- (You would not really do it that way in APL, since there is a predefined factorial operator: !X)
- Could be called *functional*, but has little in common with most functional languages

The Odd Controversies

- Programming languages are the subject of many heated debates:
  - Partisan arguments
  - Language standards
  - Fundamental definitions

Language Partisans

- There is a lot of argument about the relative merits of different languages
- Every language has partisans, who praise it in extreme terms and defend it against all detractors
- To experience some of this, explore newsgroups: `comp.lang.*` or `/r/programming`

Language Standards

- The documents that define language standards are often drafted by international committees
- Can be a slow, complicated and rancorous process
- C++ 98, 03, 07/TR1, 11, 14, 17, 20

Basic Definitions

- Some terms refer to fuzzy concepts: all those language family names, for example
- No problem, just remember they are fuzzy
  - Bad: Is X really an *object-oriented* language?
  - Good: What aspects of X support an *object-oriented* style of programming?
- Some crisp concepts have conflicting terminology: one person's *argument* is another person's *actual parameter*

The Intriguing Evolution

- Programming languages are evolving rapidly
  - New languages are being invented
  - Old ones are developing new dialects

New Languages

- A clean slate: no need to maintain compatibility with an existing body of code
- But never entirely new any more: always using ideas from earlier designs
- Some become widely used, others do not
- Whether widely used or not, they can serve as a source of ideas for the next generation

Widely Used: Java

- Quick rise to popularity since 1995 release
- C# uses many ideas from Java and C++, plus some from Mesa, Modula, and other languages
- C++ uses most of C and extends it with ideas from Simula 67, Ada, Clu, ML and Algol 68
- C was derived from B, which was derived from BCPL, which was derived from CPL, which was derived from Algol 60

Not Widely Used: Algol

- One of the earliest languages: Algol 58, Algol 60, Algol 68
- Never widely used
- Introduced many ideas that were used in later languages, including
  - Block structure and scope
  - Recursive functions
  - Parameter passing by value

Dialects

- Experience with languages reveals their design weaknesses and leads to new dialects
- New ideas pass into new dialects of old languages

Some Dialects Of Fortran

- Original Fortran, IBM
- Major standards:
  - Fortran II
  - Fortran III
  - Fortran IV
  - Fortran 66
  - Fortran 77
  - Fortran 90
  - Fortran 95
  - Fortran 2003
  - Fortran 2008
  - Fortran 2018
- Deviations in each implementation
- Parallel processing
  - HPF
  - Fortran M
  - Vienna Fortran
- And many more. . .

The Connection To Programming Practice

- Languages influence programming practice
  - A language favors a particular programming style – a particular approach to algorithmic problem-solving
- Programming experience influences language design

Language Influences Programming Practice

- Languages often strongly favor a particular style of programming
  - Object-oriented languages: a style making heavy use of objects
  - Functional languages: a style using many small side-effect-free functions
  - Logic languages: a style using searches in a logically-defined problem space

Fighting the Language

- Languages favor a particular style, but do not force the programmer to follow it
- It is always possible to write in a style not favored by the language
- It is not usually a good idea. . .
  - C++ is not good for logic programming.
  - Prolog is not good for systems programming.

Imperative ML

ML makes it hard to use assignment and side-effects. But it is still possible:

Example

```
fun fact n =
  let
    val i = ref 1;
    val xn = ref n
  in
    while !xn>1 do (
      i := !i * !xn;
      xn := !xn - 1
    );
    !i
  end;
```

Non-object-oriented C#

C#, more than C++, tries to encourage you to adopt an object-oriented mode. But you can still put your whole program into static methods of a single class:

Example

```
class Fubar {
  public static void Main (string args[]) {
    // whole program here!
  }
}
```

Functional Pascal

Any imperative language that supports recursion can be used as a functional language:

Example

```
function ForLoop(Low, High: Integer): Boolean;
  begin
    if Low <= High then
      begin
        {for-loop body here}
        ForLoop := ForLoop(Low+1, High)
      end
    else
      ForLoop := True
  end;
```

Programming Influences Language Design

- Corrections to design problems make future dialects, as already noted
- Programming styles can emerge before there is a language that supports them
  - Programming with objects predates object-oriented languages
  - Automated theorem proving predates logic languages

Computer Architecture

- Language evolution drives and is driven by hardware evolution:
  - Call-stack support – languages with recursion
  - Parallel architectures – parallel languages
  - Internet – Java

Theory of Formal Languages

- Theory of formal languages is a core mathematical area of computer science
  - Regular grammars, finite-state automata – lexical structure of programming languages, scanner in a compiler
  - Context-free grammars, pushdown automata – phrase-level structure of programming languages, parser in a compiler
  - Turing machines – Turing-equivalence of programming languages

Turing Equivalence

- Languages have different strengths, but fundamentally they all have the same power
  - {problems solvable in Java}
    = {problems solvable in Fortran}
    = . . .
- And all have the same power as various mathematical models of computation
  - {problems solvable by Turing machine}
    = {problems solvable by lambda calculus}
    = . . .
- Church-Turing thesis: this is what "computability" means

Conclusion

Conclusion

- Why programming languages are worth studying (and this course worth taking):
  - The amazing variety
  - The odd controversies
  - The intriguing evolution
  - The connection to programming practice
  - The many other connections
- Plus. . . there is the fun of learning three new languages!