Programming Languages
— An Overview —

COMP 524: Programming Language Concepts
Björn B. Brandenburg
The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Tuesday, January 12, 2010
A Brief History of Modern Computing

Early computers required rewiring.
- For example, ENIAC (Electronic Numerical Integrator and Computer, 1946) programed with **patch cords**.
- Reprogramming took weeks.
- Used to compute artillery tables.

Von Neumann: stored program computers.
- Innovation: **program is data**.
- Program stored in core memory.
- Allowed for “rapid” reprogramming.

Early programming.
- Programmers wrote bare machine code.
- Essentially, **strings of zeros and ones**.
- Created with punchcards.
Machine Code

Limitations.
- Hard for humans to read and write.
- Very error-prone.
- Slow development.
Assembly Code

Idea: use the computer to simplify programming!
- Possible since programs are data.
- Computer transforms human-readable input into machine code.

First step: direct mapping.
- Use mnemonic abbreviations for instructions.
  - One abbreviations for each instruction.
  - Also encode operands.
- Computer assembles real program by mapping each line to its machine code equivalent, thus creating a new program.
- Assemblers are still in use today.
Assembly Code

Idea: use the computer to simplify programming!
- Possible since programs are data.
- Computer transforms human-readable input into machine code.

First step: direct mapping.
- Use mnemonic abbreviations for instructions.
  - One abbreviations for each instruction.
  - Also encode operands.
- Computer assembles real program by mapping each line to its machine code equivalent, thus creating a new program.
- Assemblers are still in use today.

Example: Intel x86-32 machine code and assembly language of javac program.

- Computer transforms human-readable input into machine code.
- Possible since programs are data.
- First step: direct mapping.
  - Use mnemonic abbreviations for instructions.
    - One abbreviations for each instruction.
    - Also encode operands.
  - Computer assembles real program by mapping each line to its machine code equivalent, thus creating a new program.
  - Assemblers are still in use today.

Example: Intel x86-32 machine code and assembly language of javac program.
Towards Higher-Level Languages

Limitations of assembly code.
- Still hard to read.
- **No error checking.**
- Machine specific, **not portable.**
  - Hardware architecture changed frequently in the early days.
- Tedious to write.
  - **Macros** somewhat alleviate this.

Desired: higher-level representation.
- Machine independent.
- More like **mathematical formulas.**
  - Usable by scientists.
- **Catch common errors.**

---

**Macro expansion:**
Programmer defines parametrized abbreviation; assembler replaces each occurrence of abbreviation with definition.

**Example:**
A macro with two parameters on Linux. Implements the write system call.

```
.macro write str, str_size
  movl  $4, %eax
  movl  $1, %ebx
  movl  \str, %ecx
  movl  \str_size, %edx
  int   $0x80
.endm
```

Subsequently, strings can be output with **write** `<address of string>`, `<length>` instead of the whole system call sequence.

High-Level Language

Key properties.
- Provides **facilities for data and control flow abstraction**.
- Machine-independent specification.
- One high-level statement typically corresponds to many machine instructions.
- **Human-friendly** syntax.
- Programming model / semantics not defined in terms of machine capabilities.

Translation to machine code.
- Checked and translated by **compiler**.
  - Alternatively, interpreted (next lecture).
- Initially, slower than handwritten assembly code.
- Today, compiler-generated code outperforms most human-written assembly code.
Early High-Level Languages

**FORTRAN**
- John Backus (IBM), 1954.
- *Formula Translating System*
- For numerical computing.
- Focus: **efficiency**.

**LISP**
- *List Processor*.
- For symbolic computing.
- Focus: **abstraction**.

**ALGOL**
- *Algorithmic Language*
- For specification of algorithms.
- Focus: **clear** and **elegant design**.

**COBOL**
- Grace Hopper (US Navy), 1959.
- *Common Business-Oriented Language*.
- For data processing in businesses.
- Focus: **english-like syntax**.
Early High-Level Languages

FORTRAN
- John Backus (IBM), 1954.
- *Formula Translating System*
- For numerical computing.
- Focus: **efficiency**.

LISP
- *List Processor*.
- For symbolic computing.
- Focus: **abstraction**.

FORTRAN, LISP, and COBOL are still in wide-spread use today! (in revised forms)

ALGOL
- *Algorithmic Language*
- For specification of algorithms.
- Focus: **clear and elegant design**.

COBOL
- Grace Hopper (US Navy), 1959.
- *Common Business-Oriented Language*.
- For data processing in businesses.
- Focus: **english-like syntax**.

ALGOL was highly influential and (revised versions) were the *de-facto* standard for the description of algorithms for most of the 20th century.
Definition

What is a Programming Language?

- Java? Yes.
- HTML? No.
- Javascript? Yes.
- LaTeX? Yes.

A programming language is a formal language that is both:

- universal (any computable function can be defined)
- implementable (on existing hardware platforms).

Turing-complete: can simulate any Turing machine.
(of course, real hardware has space constraints)
To be of practical interest, a language should also:

“Naturally” express algorithms.

- With respect to its intended problem domain.
- This is often achieved by mimicking existing notation or adopting core concepts (e.g., function definitions, predicates).
- In essence, a language must appeal to its intended users to be successful.

Be efficiently implementable.

- Acceptable definitions of “efficient” vary by problem domain.
- For example, in high-performance computing, there is typically no “efficient enough.”
- In contrast, in work on artificial intelligence, efficiency was often only a secondary concern in the past.
Programming Language Spectrum

Declarative Languages
- focus on *what* the computer should do
  - Functional
    - (Ex: LISP/Scheme, ML, Haskell)
  - Logic and constraint-based
    - (Ex: Prolog)
  - Dataflow
    - (Ex: Id, Val)

Imperative Languages
- focus on *how* the computer should do
  - Procedural / Von Neumann
    - (Ex: Fortran, Pascal, C)
  - Object-Oriented
    - (Ex: Smalltalk, Eiffel, C++, Java)
  - Scripting
    - (Ex: Shell, TCL, Perl, Python)

“do what I mean” — “do exactly what I say”
**Programming Languages**

**Declarative Languages**

- **Functional**
  - (Ex: LISP/Scheme, ML, Haskell)

- **Logic and constraint-based**
  - (Ex: Prolog)

- **Dataflow**
  - (Ex: Id, Val)

**Imperative Languages**

- **Procedural / Von Neumann**
  - (Ex: Fortran, Pascal, C)

- **Object-Oriented**
  - (Ex: Smalltalk, Eiffel, C++, Java)

- **Scripting**
  - (Ex: Shell, TCL, Perl, Python)

---

**“do what I mean”**

- **Procedural Languages**:
  
  Direct evolution from assembly (and thus how computers work internally):
  
  a program is a **sequential computation** that **directly manipulates** simple **typed data** (memory locations); abstraction is achieved by calling **subroutines** as service providers.

**“do exactly what I say”**
Object-Oriented Languages:

Human-inspired model: problems are solved by a team of **objects that collaborate** by **sending messages** to each other.

Objects represent “**subcontractors**” that do one job (possibly with the help of other “experts”) and **encapsulate** all related **state**.

The benefit of object-orientation is twofold: that large, **complex problems** can be **decomposed** in a “**natural**” way; and message passing can be compiled into **efficient procedural code**.
Functional Languages:

Mathematics-inspired model: program defined in terms of *mathematical functions* (equivalences).

There is **no concept of memory**: functions simply map values onto other values.

There is **no concept of time**: mathematical functions just are; there is no “before” and “after.”

There is **no concept of state**: functions are only defined in terms of their arguments and other functions.

The computer’s job is to **compute the result of applying the program (a function) to the input**.

How this is done is not specified in the program. **Control flow is implicit** and based on recursion.

---

Declarative Languages focus on **what** the computer should do:

- **Functional** (Ex: LISP/Scheme, ML, Haskell)
- **Logic and constraint-based** (Ex: Prolog)
- **Dataflow** (Ex: Id, Val)

“**do what I mean**”

---

“**do exactly what I say**”
Programming Language Spectrum

Declarative Languages
focus on **what** the computer should do

- **Functional**
  (Ex: LISP/Scheme, ML, Haskell)
- **Logic and constraint-based**
  (Ex: Prolog)
- **Dataflow**
  (Ex: Id, Val)

“do what I mean”

Imperative Languages
focus on **how** the computer should do it

- **Procedural / Von Neumann**
  (Ex: Fortran, Pascal, C)
- **Object-Oriented**
  (Ex: Smalltalk, Eiffel, C++, Java)
- **Functional**
  (Ex: LISP/Scheme, ML, Haskell)
- **Logic and constraint-based**
  (Ex: Prolog)
- **Dataflow**
  (Ex: Id, Val)
- **Scripting**
  (Ex: Shell, TCL, Perl, Python)

“do exactly what I say”

Logic Languages:
Inspired by propositional logic. Program is defined in terms of

- **facts** (the “knowledge base”),
- **rules** (implications, “if X then also Y”), and a **goal** (query, “is Y true?”, “what makes Y true?”).

The computer’s job is to **construct a proof** based on the given **axioms** (facts + rules).
Programming Language Spectrum

**Imperative Languages**

- **Procedural / Von Neumann**
  (Ex: Fortran, Pascal, C)
- **Object-Oriented**
  (Ex: Smalltalk, Eiffel, C++, Java)
- **Scripting**
  (Ex: Shell, TCL, Perl, Python)

“do exactly what I say”

**Declarative Languages**

- **Logic and constraint-based**
  (Ex: Prolog)
- **Dataflow**
  (Ex: Id, Val)

“do what I mean”

**Dataflow Languages**

Similar to gate networks (hardware).

Tokens (units of data) are *streamed* through a network of primitive functional units.

“Unix pipes + loops + multiple inputs / outputs.”

Tuesday, January 12, 2010
Scripting Languages:

Fuzzy category of high-level languages that focus heavily on developer productivity (“rapid development”).

Often used for integration of components (“glue languages”), more recently for web development.

Traditionally imperative model, but there is a trend to include object-oriented and functional design elements.
Programming Language Spectrum

Declarative Languages
focus on **what** the computer should do

- **Functional**
  (Ex: LISP/Scheme, ML, Haskell)

- **Logic and constraint-based**
  (Ex: Prolog)

- **Dataflow**
  (Ex: Id, Val)

Imperative Languages
focus on **how** the computer should do it

- **Procedural / Von Neumann**
  (Ex: Fortran, Pascal, C)

- **Object-Oriented**
  (Ex: Smalltalk, Eiffel, C++, Java)

- **Scripting**
  (Ex: Shell, TCL, Perl, Python)

**Note:** this is a very coarse-grained view.
- most real-world languages are not **pure** (i.e., they mix categories).
- there exist many **sub-categories** (e.g., synchronous reactive FP).
Design Considerations

What are the primary use cases?

Communicate ideas.
- Programs are read more often than written.
- Maintenance costs.

Exactly specify algorithms.
- Succinct and precise.
- No ambiguity.

Create useful programs.
- Development must be economically viable.

- Readability
- Expressivity
- Writability
- Reliability

What are the primary use cases?
Readability Factors

What does this code fragment do?

Simplicity.
- Limited number of concepts / variants.

Orthogonality.
- Are concepts independent of each other?
- Lack of special cases.

Syntax design.
- Identifier restrictions (e.g., hyphen vs minus).
- Terseness; frequency of operator symbols.
  ‣ For example, \(|x|\) vs. \(x.length()\).
  ‣ But: \(x.add(y.times(z))\) vs. \(x + y * z\).

Explicit constraints.
- Assumptions made explicit and checked.
- Enforced “design by contract.”

Java: many ways to increment.

++x; x++; x = x + 1; x += 1;

Java:
ArrayList<int> vs. ArrayList<Integer>

Example: variable name for “global input database file”

FORTRAN 77: GIDBFL (max 6 chars.) vs. LISP: *input-database-file*

Eiffel keywords:
invariant, require, ensure
Eiffel: Checked Constraints Example

```
indexing ... class COUNTER

feature decrement is
   -- Decrease counter by one.
   require item > 0
   do
      item := item - 1
   ensure item = old item - 1
   end

... invariant
   item >= 0

end
```

Example: Expressivity

QuickSort in Haskell

\[
\text{qsort } \ [\ ] = [\ ]
\]
\[
\text{qsort } (x:xs) = \text{qsort } \text{lt}_x ++ [x] ++ \text{qsort } \text{ge}_x
\]

where
\[
\text{lt}_x = [y \mid y \leftarrow xs, y < x]
\]
\[
\text{ge}_x = [y \mid y \leftarrow xs, y \geq x]
\]

(we will discuss Haskell in detail later in the semester)

QuickSort in C

\[
\text{qsort}( \ a, \ \text{lo}, \ \text{hi} \ ) \ \text{int} \ a[], \ \text{hi}, \ \text{lo};\{
\text{int} \ \text{w}, \ \text{h}, \ \text{p}, \ \text{t};
\text{if} \ (\text{lo} < \text{hi}) \{ \\
\text{w} = \text{lo}; \\
\text{h} = \text{hi}; \\
\text{p} = a[\text{hi}]; \\
\text{do} \{ \\
\text{while} \ ((\text{w} < \text{h}) \ \&\& \ (a[\text{w}] \leq p)) \\
\text{w} = \text{w}+1; \\
\text{while} \ ((\text{h} > \text{w}) \ \&\& \ (a[\text{h}] \geq p)) \\
\text{h} = \text{h}-1; \\
\text{if} \ (\text{w} < \text{h}) \{ \\
\text{t} = a[\text{w}]; \\
\text{a}[\text{w}] = a[\text{h}]; \\
\text{a}[\text{h}] = \text{t}; \\
\}
\} \text{while} \ (\text{w} < \text{h});
\text{t} = a[\text{w}]; \\
\text{a}[\text{w}] = a[\text{hi}]; \\
\text{a}[\text{hi}] = \text{t}; \\
\text{qsort} ( \ a, \ \text{lo}, \ \text{w}-1 \ ); \\
\text{qsort} ( \ a, \ \text{w}+1, \ \text{hi} \ ); \\
\}
\}
Example: Expressivity

Quicksort in Haskell

qsort [] = []
qsort (x:xs) = qsort lt_x ++ [x] ++ qsort ge_x
where
  lt_x = [y | y <- xs, y < x]
  ge_x = [y | y <- xs, y >= x]

For any ordered datatype.

(we will discuss Haskell in detail later in the semester)

Quicksort in C

```c
int qsort( a, lo, hi ) {
    if (lo < hi) {
        int w, h, p, t;
        w = lo;
        h = hi;
        p = a[hi];
        do {
            while ((w < h) && (a[w] <= p))
                w = w+1;
            while ((h > w) && (a[h] >= p))
                h = h-1;
            if (w < h) {
                t = a[w];
                a[w] = a[h];
                a[h] = t;
            }
        } while (w < h);
        t = a[w];
        a[w] = a[hi];
        a[hi] = t;
        qsort( a, lo, w-1 );
        qsort( a, w+1, hi );
    }
}
```

Quicksort in C

Only for int.

Quicksort in Haskell

Only for int.

For any ordered datatype.

(we will discuss Haskell in detail later in the semester)
Writability Factors

Facilities for abstraction
- Define each concept only **once**.

Repetition avoidance.
- **DRY principle**: “don’t repeat yourself”
- Code generation.
- Generic programming.
- Sparse type declarations, type inference.

Quality of development tools.
- **Efficiency** of compiler-generated code.
- Availability of **libraries**.
- Leniency of compiler / language system.
- Turnaround time of **edit-compile-test** cycle.
- Number of available compiler / tool chains.

Documentation.
- Availability and **quality**.

---

**Haskell**: allows numeric integration to be defined once for *any* function.

**Ruby**: The “Ruby on Rails” web framework drastically reduced the need for configuration files.

**D**: designed as a C successor, it has been hindered by the existence of incompatible compilers and libraries.

**gcc**: some warnings not used in Linux due to excessive false positives.

**Java**: javadoc support ensures standardized, indexable documentation.
Reliability Factors

Static error detection.
- Type checking.
- Constraint checking.
- Model-driven development.
- Model extraction.

Dynamic error detection.
- Array bounds checking.
- Integer overflow detection.

Ease of error handling.
- Structured exception handling.
- Error propagation.

Versioning of components.
- Avoid mismatch in assumptions.

Ease of testing.
- Unit testing support.
- Test case generation.

Example: detect use of uninitialized variables.

Model-checking is a technique to automatically prove safety and liveness properties.

C: lack of run-time checking has caused billions in damages due to security incidences.

In Erlang, processes can be linked: if one fails, then all linked processes are also terminated. This prevents “half-dead” systems.

Example: detect when interface has changed.

Haskell: the QuickCheck library aids debugging by automatically generating counter examples to invariants based on type signatures.

Ease of error handling.
- Structured exception handling.
- Error propagation.

Versioning of components.
- Avoid mismatch in assumptions.

Ease of testing.
- Unit testing support.
- Test case generation.
Language Design Tradeoff

program safety

developer productivity

program efficiency
Summary

History.
- Programming language development started with a desire for **higher levels of abstraction**.
- Compiling very high levels of abstraction into **efficient** machine code is challenging.

Programming Language Spectrum.
- Language design involves many tradeoffs.
- The result: many competing languages, all slightly different.
- Often variations on a theme.

Categories.
- **Declarative**: what to do.
  ‣ Functional, logic-based, dataflow.
- **Imperative**: how to do it.
  ‣ Procedural, object-oriented, scripting.