Programming Languages

— An Overview —

COMP 524: Programming Language Concepts
Björn B. Brandenburg

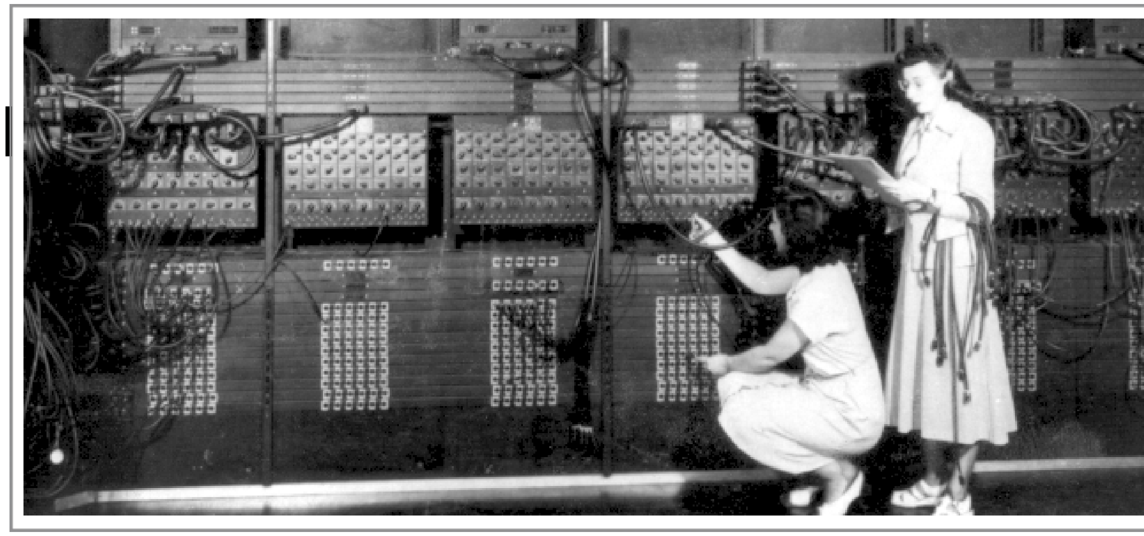
The University of North Carolina at Chapel Hill

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

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.

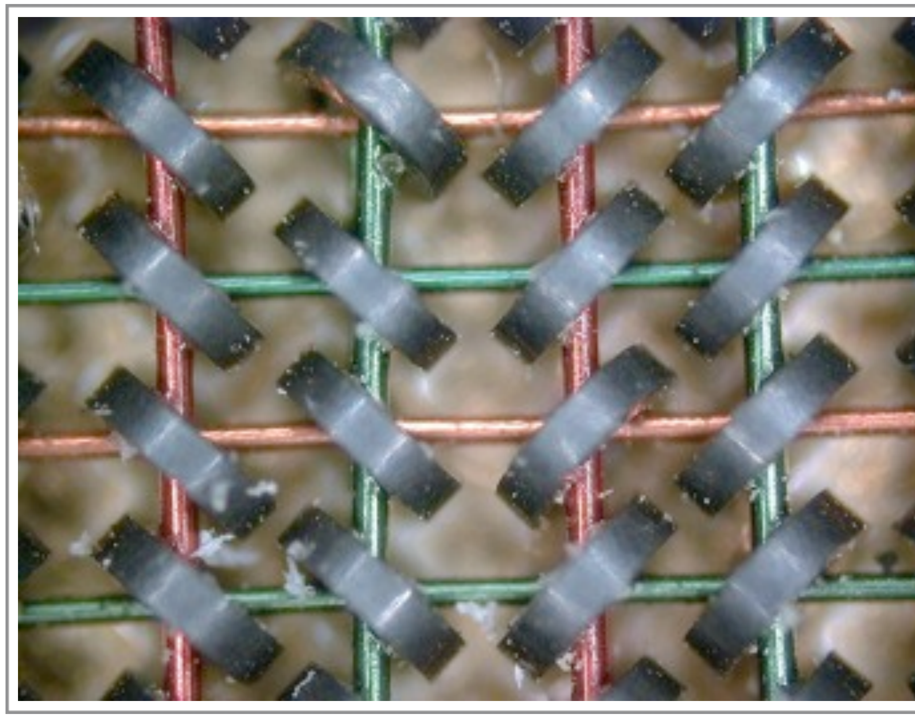


Von Neumann: stored program computers.

- Innovation: **program is data**.
- Program stored in core memory.
- Allowed for "rapid" reprogramming.

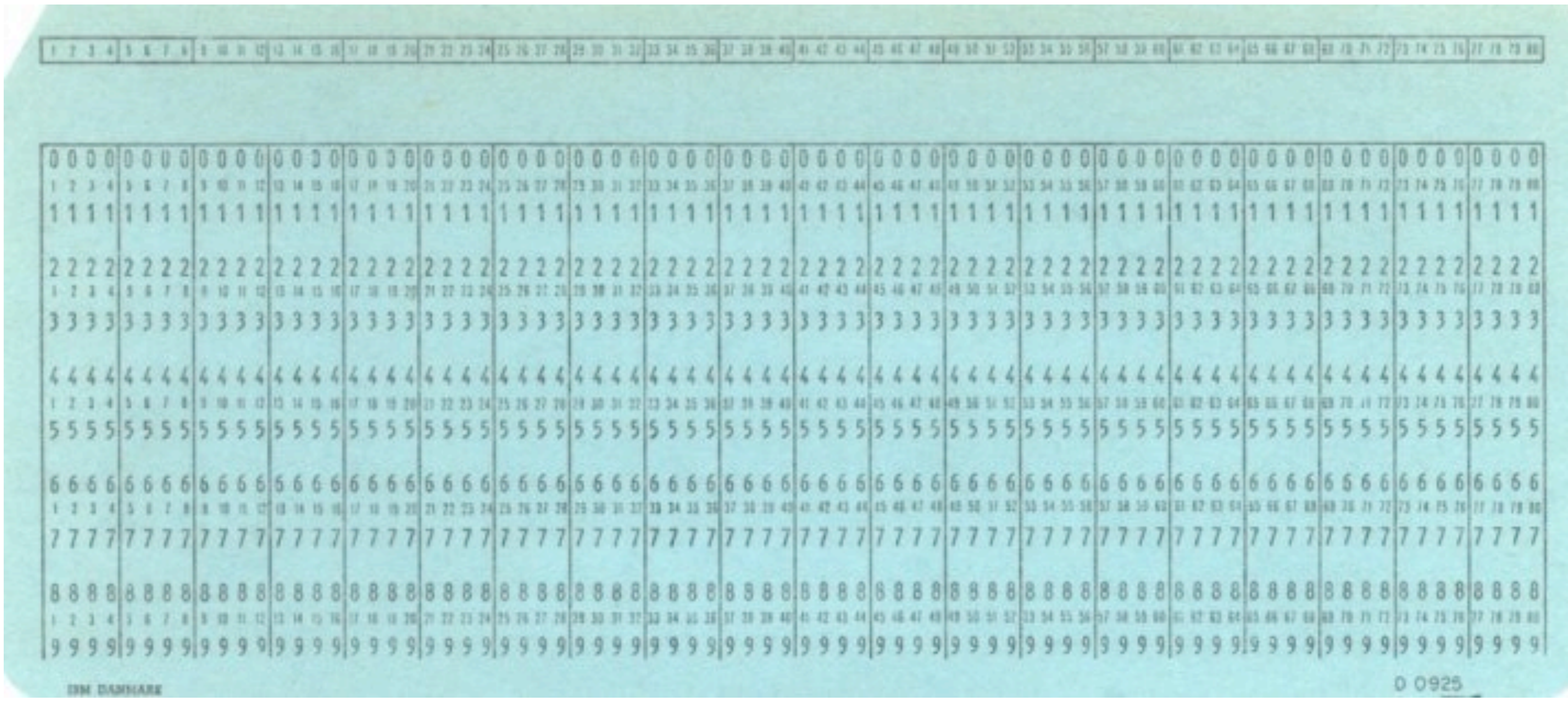
Early programming.

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



Magnetic core memory. Each core is one bit.
Source: Wikimedia Commons
Credit: H.J. Sommer III, Professor of Mechanical Engineering, Penn State University

Machine Code



A punch card.
Source: Wikimedia Commons

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.

```
08049580 <main>:
 8049580:       55                      push   %ebp
 8049581:       89 e5                   mov    %esp,%ebp
 8049583:       31 c0                   xor    %eax,%eax
 8049585:       57                      push   %edi
 8049586:       31 ff                   xor    %edi,%edi
 8049588:       56                      push   %esi
 8049589:       31 f6                   xor    %esi,%esi
 804958b:       53                      push   %ebx
 804958c:       e8 7f 34 00 00          call   804ca10 <__i686.get_pc_thunk.bx>
 8049591:       81 c3 b3 92 00 00       add    $0x92b3,%ebx
 8049597:       81 ec bc 20 00 00       sub    $0x20bc,%esp
 804959d:       89 bd 74 df ff ff       mov    %edi,-0x208c(%ebp)
 80495a3:       83 e4 f0                and    $0xfffffff0,%esp
 80495a6:       89 85 64 df ff ff       mov    %eax,-0x209c(%ebp)
 80495ac:       31 c0                   xor    %eax,%eax
 80495ae:       89 85 60 df ff ff       mov    %eax,-0x20a0(%ebp)
 80495b4:       8d 83 c4 c0 ff ff       lea    -0x3f3c(%ebx),%eax
 80495ba:       89 04 24                mov    %eax,(%esp)
 80495bd:       e8 96 fb ff ff          call   8049158 <getenv@plt>
 80495c2:       85 c0                   test   %eax,%eax
 80495c4:       0f 85 fa 02 00 00       jne    80498c4 <main+0x344>
 80495ca:       8b 45 08                mov    0x8(%ebp),%eax
 80495cd:       89 04 24                mov    %eax,(%esp)
 80495d0:       8b 45 0c                mov    0xc(%ebp),%eax
 80495d3:       89 44 24 04             mov    %eax,0x4(%esp)
 80495d7:       8d 85 74 df ff ff       lea    -0x208c(%ebp),%eax
 80495dd:       89 44 24 08             mov    %eax,0x8(%esp)
 80495e1:       e8 5a 0e 00 00          call   804a440 <SelectVersion>
 80495e6:       8b 45 08                mov    0x8(%ebp),%eax
 80495e9:       8d 04 85 04 00 00 00    lea    0x4(,%eax,4),%eax
 80495f0:       89 04 24                mov    %eax,(%esp)
 80495f3:       e8 90 fe ff ff          call   8049488 <JLI_MemAlloc@plt>
 80495f8:       31 d2                   xor    %edx,%edx
 80495fa:       89 c1                   mov    %eax,%ecx
 80495fc:       8b 45 08                mov    0x8(%ebp),%eax
 80495ff:       40                      inc    %eax
 8049600:       39 c6                   cmp    %eax,%esi
 8049602:       7d 1e                   jge    8049622 <main+0xa2>
 8049604:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
 804960a:       8d bf 00 00 00 00       lea    0x0(%edi),%edi
 8049610:       8b 45 0c                mov    0xc(%ebp),%eax
```

Assembly Code

Idea: use the c
programming!

➡ Possible sinc

➡ Computer **transforms** human-readable input into machine code.

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

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.

Machine Code

Instructions

Operands

```
08049580 <main>:
 8049580:                           push   %ebp
 8049581:                           mov    %esp,%
 8049583:                           xor    %eax,%eax
 8049585:                           push   %edi
           31 ff                    xor    %edi,%edi
           56                       push   %esi
           31 f6                    xor    %esi,%esi
           53                       push   %ebx
           e8 7f 34 00 00           call   804ca10 <__i686.get_pc_thunk.bx>
           81 c3 b3 92 00 00        add    $0x92b3,%ebx
           81 ec bc 20 00 00        sub    $0x20bc,%esp
           89 bd 74 df ff ff        mov    %edi,-0x208c(%ebp)
 80495a3:  83 e4 f0                 and    $0xfffffff0,%esp
 80495a6:  89 85 64 df ff ff        mov    %eax,-0x209c(%ebp)
 80495ac:  31 c0                    xor    %eax,%eax
 80495ae:  89 85 60 df ff ff        mov    %eax,-0x20a0(%ebp)
 80495b4:  8d 83 c4 c0 ff ff        lea    -0x3f3c(%ebx),%eax
 80495ba:  89 04 24                 mov    %eax,(%esp)
 80495bd:  e8 96 fb ff ff           call   8049158 <getenv@plt>
 80495c2:  85 c0                    test   %eax,%eax
 80495c4:  0f 85 fa 02 00 00        jne    80498c4 <main+0x344>
 80495ca:  8b 45 08                 mov    0x8(%ebp),%eax
 80495cd:  89 04 24                 mov    %eax,(%esp)
 80495d0:  8b 45 0c                 mov    0xc(%ebp),%eax
 80495d3:  89 44 24 04              mov    %eax,0x4(%esp)
 80495d7:  8d 85 74 df ff ff        lea    -0x208c(%ebp),%eax
 80495dd:  89 44 24 08              mov    %eax,0x8(%esp)
 80495e1:  e8 5a 0e 00 00           call   804a440 <SelectVersion>
 80495e6:  8b 45 08                 mov    0x8(%ebp),%eax
 80495e9:  8d 04 85 04 00 00 00     lea    0x4(,%eax,4),%eax
 80495f0:  89 04 24                 mov    %eax,(%esp)
 80495f3:  e8 90 fe ff ff           call   8049488 <JLI_MemAlloc@plt>
 80495f8:  31 d2                    xor    %edx,%edx
 80495fa:  89 c1                    mov    %eax,%ecx
 80495fc:  8b 45 08                 mov    0x8(%ebp),%eax
 80495ff:  40                       inc    %eax
 8049600:  39 c6                    cmp    %eax,%esi
 8049602:  7d 1e                    jge    8049622 <main+0xa2>
 8049604:  8d b6 00 00 00 00        lea    0x0(%esi),%esi
 804960a:  8d bf 00 00 00 00        lea    0x0(%edi),%edi
 8049610:  8b 45 0c                 mov    0xc(%ebp),%eax
```


Tuesday, January 12, 2010

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.

Source: http://www.ibm.com/developerworks/library/l-gas-nasm.html

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

- John McCarthy (MIT), 1958.
- *List Processor*.
- For symbolic computing.
- Focus: **abstraction**.

ALGOL

- John Backus (IBM), Friedrich Bauer (TU Munich), *etal.*, 1958.
- *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

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

FORTRAN

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

LISP

- John McCarthy (MIT), 1958.
- *List Processor*.
- For symbolic computing.
- Focus: **abstraction**.

ALGOL

- John Backus (IBM), Friedrich Bauer (TU Munich), *etal.*, 1958.
- *Algorithmic Language*
- For specification of algorithms.
- Focus: **clear** and **elegant design**.

COBOL

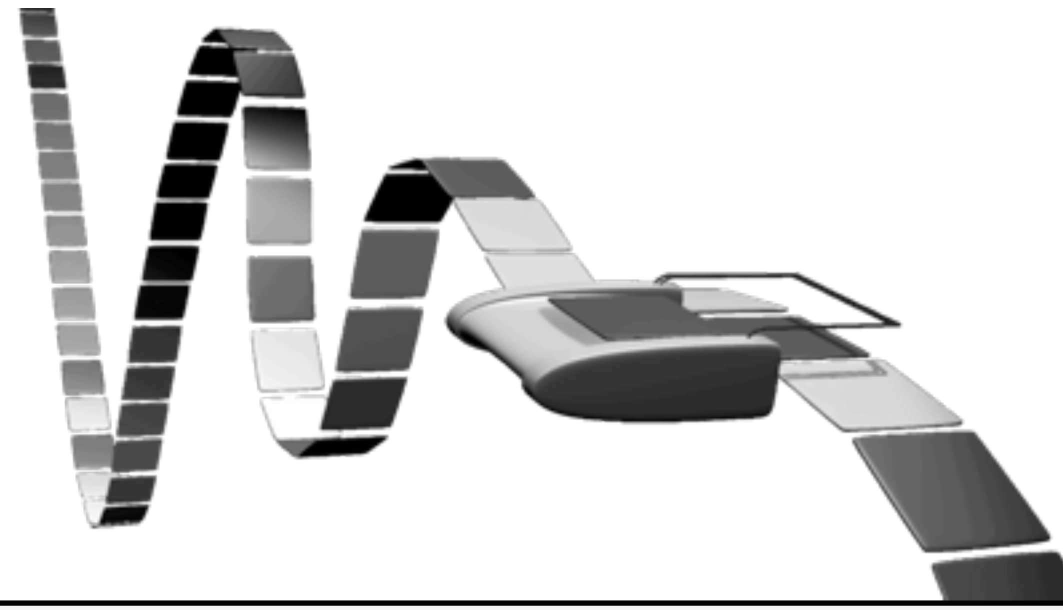
- Grace Hopper (US Navy), 1959.
- *Common Business-Oriented Language*.
- For data processing in businesses.
- Focus: **english-like syntax**.

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

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)

Illustration source: Wikimedia Commons

Practical Languages

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.

“do what I mean”

“do exactly what I say”

Tuesday, January 12, 2010

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”

Procedural Languagess:

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.

Program...n

Declarative La...

focus on **what** the computer should do | focus on **how** the computer should do it

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

Logic and constraint-based
(Ex: Prolog)

Dataflow
(Ex: Id, Val)

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"*

...nguage Spectrum

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**.

De...

focu...

(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)

"do what I mean"

"do exactly what I say"

Tuesday, January 12, 2010

Programming La

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)

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**.

"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)

Imperative Languages

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

Procedural / Von Neumann

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).

"do what I mean"

Tuesday, January 12, 2010

Programming Language Spectrum

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.”

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)

“do what I mean”

“do exactly what I say”

Prog age Spectrum

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.

Declarati

focus on **what**

F
(Ex: LISP/Sc

Logic and
(E

Dataflow
(Ex: Id, Val)

ative Languages

ow the computer should do it

cedural / Von Neumann
Ex: Fortran, Pascal, C)

Object-Oriented
Smalltalk, Eiffel, C++, Java)

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

"do what I mean"

"do exactly what I say"

Tuesday, January 12, 2010

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.

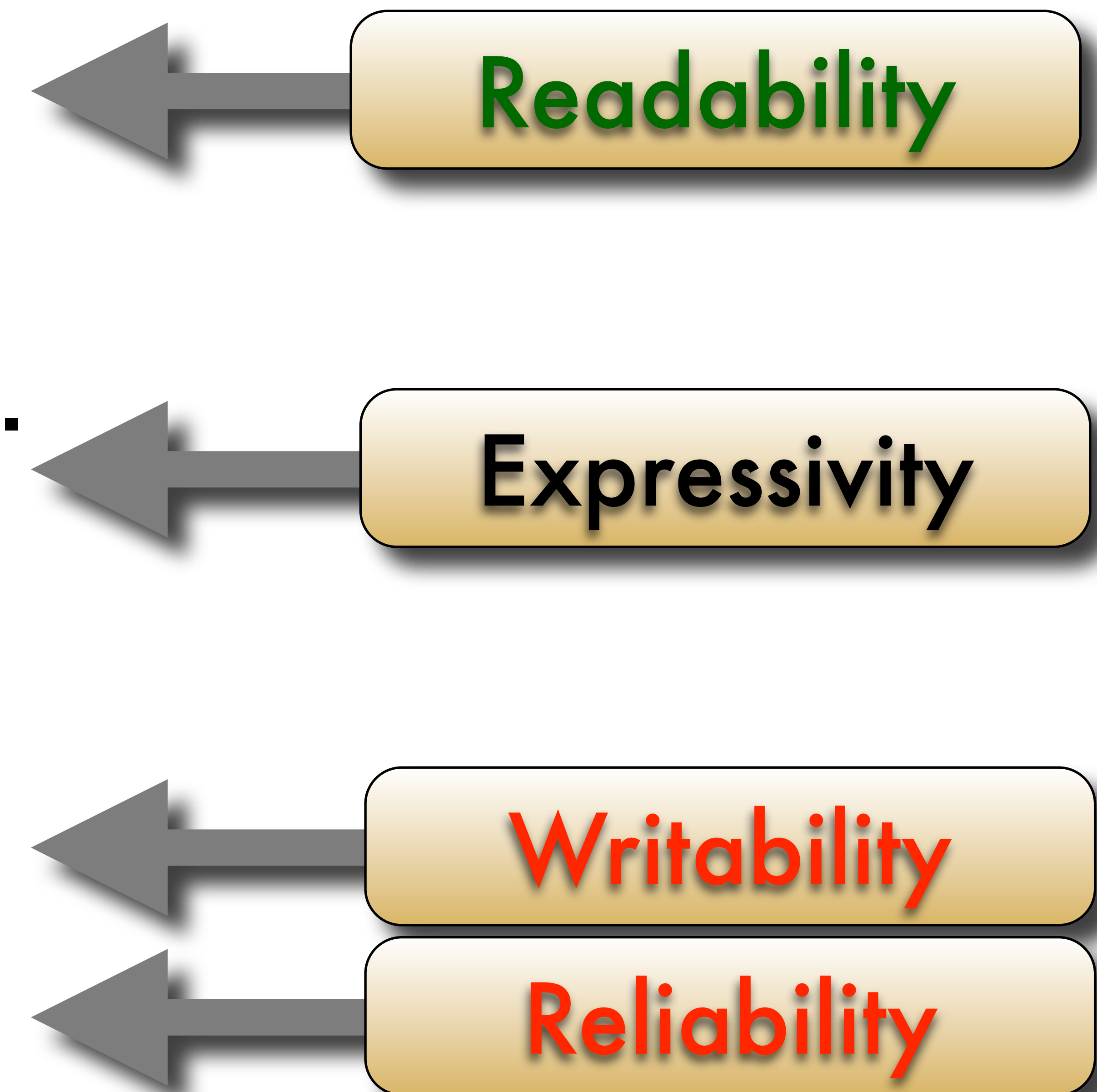
- ➡ Programs are read more often than written.
- ➡ Maintenance costs.

Exactly specify algorithms.

- ➡ Succinct and precise.
- ➡ No ambiguity.

Create useful programs.

- ➡ Development must be economically viable.



Readability Factors

What does this code fragment do?

Simplicity.

- Limited **number of concepts** / variants.

Java: many ways to increment.
`++x; x++; x = x + 1; x += 1;`

Orthogonality.

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

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

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`.

Example: variable name for "global input database file"

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

Explicit constraints.

- Assumptions made **explicit** and **checked**.
- **Enforced** "design by contract."

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
```

Precondition

Postcondition

Invariant

Source: http://archive.eiffel.com/eiffel/nutshell.html

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]
```

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

Quicksort in C

```
qsort( a, lo, hi ) int a[], hi, lo;{
  int h, w, p, t;
  if (lo < hi) {
    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 );
  }
}
```

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.

Only for int.

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

Quicksort in C

```
qsort( a, lo, hi ) int a[], hi, lo;{
  int h, w, p, t;
  if (lo < hi) {
    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 );
  }
}
```

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.

Example: detect use of uninitialized variables.

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

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

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

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

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

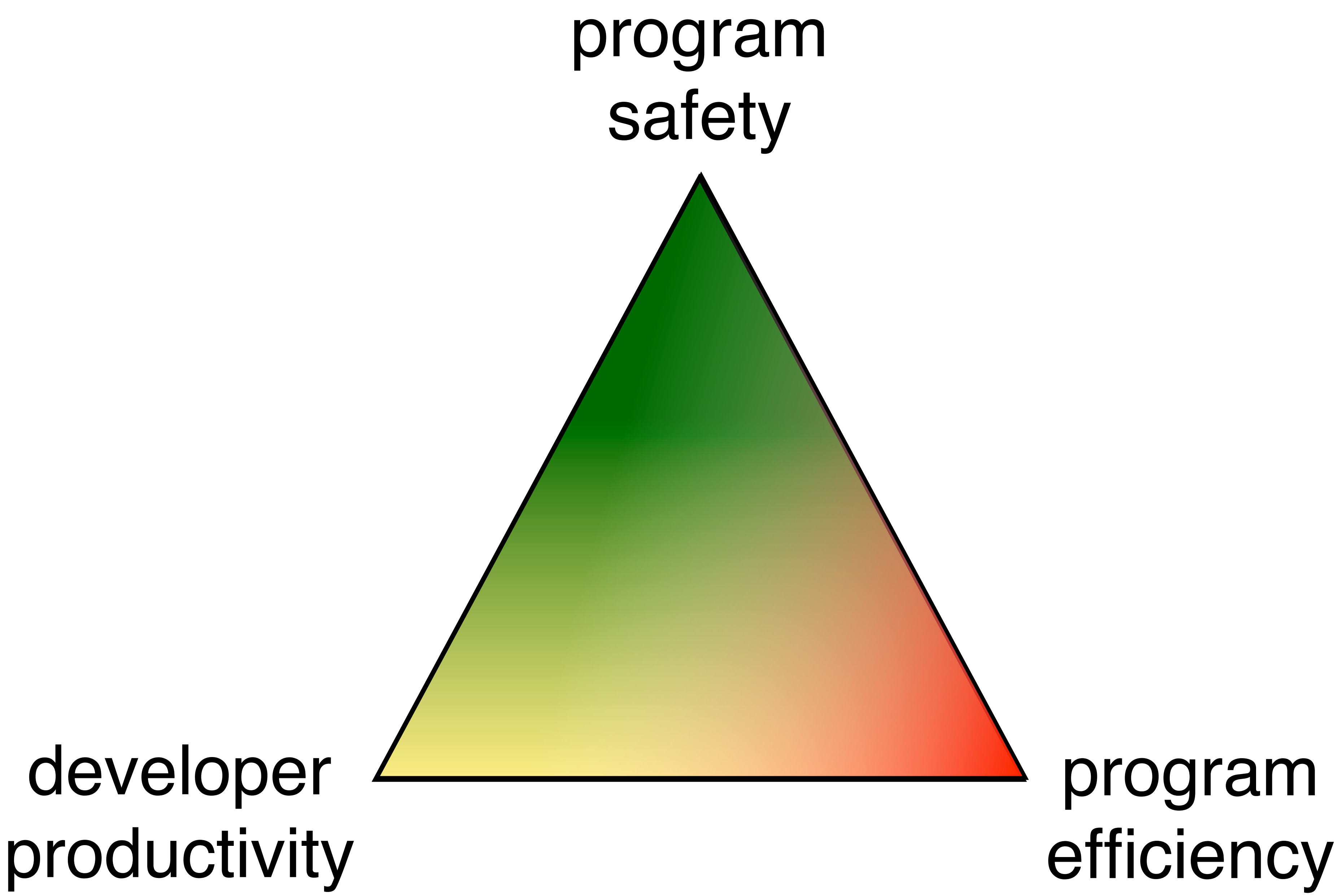
Versioning of components.
- Avoid mismatch in assumptions.

Example: detect when interface has changed.

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

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

Language Design Tradeoff



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.