Programming Language and Paradigms

Introduction

Tasanawan Soonklang
Department of Computing, Silpakorn University
Chapter 1 Topics

- A brief history of programming languages
- Why study programming languages?
- The art of language design
- Language evaluation criteria
- Programming language paradigms
- Implementation methods
- Programming environments
A Brief History of Programming Languages
**Brief history: Machine language**

- Machine language – the sequence of bits that directly controls a processor
- Add, compare, move data from one place to another, and so forth at appropriate times

```
55 89 e5 53 83 ec 04 83 e4 f0 e8 31 00 00 00 89 c3 e8 2a 00
00 00 39 c3 74 10 8d b6 00 00 00 00 39 c3 7e 13 29 c3 39 c3
75 f6 89 1c 24 e8 6e 00 00 00 8b 5d fc c9 c3 29 d8 eb eb 90
```

GCD program in machine language for the x86 (Pentium) instruction set, expressed as hexadecimal (base 16) numbers
Brief history: Assembly language

- Assembly language – expressed with mnemonic abbreviations, a less error-prone notation

```
pushl %ebp
movl %esp, %ebp
pushl %ebx
subl $4, %esp
andl $-16, %esp
call getint
movl %eax, %ebx
call getint
cmpl %eax, %ebx
je C
A: cmpl %eax, %ebx
cmpl %eax, %ebx
je C
D: subl %ebx, %eax
B:
cmpl %eax, %ebx
subl %4, %esp
jne A
andl $-16, %esp
movl %ebx, (%esp)
call getint
call put int
movl %eax, %ebx
movl -4(%ebp), %ebx
leave
ret
ret
ret
je C
D: subl %ebx, %eax
B:
```

GCD program in assembly language for the x86.
Brief history: Assembly language

- One-to-one correspondence between mnemonics and machine language instructions
- Assembler – system program for translating from mnemonics to machine language
- Machine-dependent language – rewrite programs for every new machine
- Difficult to read and write large programs
Brief history: high-level language

• Fortran – first high-level language in the mid-1950s
• Machine-independent language
• Compiler – system program for translating from high-level language to assembly or machine language
• Not one-to-one correspondence between source and target operations.
The diagram illustrates the evolution of programming languages from 1956 to 1992. Key languages include Algol 60, Fortran I, COBOL, PL/I, Smalltalk, Pascal, Prolog, Fortran 77, ML, C (K&R), Smalltalk 80, Ada 83, SML, and Common Lisp.
Why Study Programming Languages?
Why study programming languages?

• Understand obscure features
• Choose among alternative ways to express things
• Simulate useful features in languages that lack them
• Make it easier to learn new languages
• Help you choose a language
Why study programming languages?

- Understand obscure features
  - In C, help you understand unions, arrays & pointers, separate compilation, catch and throw
  - In C++, help you understand multiple inheritance, * operator
  - In Common Lisp, help you understand first-class functions/closures, streams, catch and throw, symbol internals
Why study programming languages?

• Choose among alternative ways to express things
  – understand implementation costs:
  – based on knowledge of what will be done underneath:
    • use simple arithmetic equal (use \( x^2 \) instead of \( x^{**2} \))
    • use C pointers or Pascal "with" statement to factor address calculations
    • avoid call by value with large data items in Pascal
    • avoid the use of call by name in Algol 60
    • choose between computation and table lookup (e.g. for cardinality operator in C or C++)
Why study programming languages?

• Simulate useful features in languages that lack them
  – lack of named constants and enumerations in Fortran
    use variables that are initialized once, then never changed
  – lack of modules in C and Pascal
    use comments and programmer discipline
  – lack of suitable control structures in Fortran
    use comments and programmer discipline for control structures
Why study programming languages?

• Make it easier to learn new languages
  – some languages are similar; easy to walk down family tree
  – concepts have even more similarity;
  – if you think in terms of iteration, recursion, abstraction (for example), you will find it easier to assimilate the syntax and semantic details of a new language than if you try to pick it up in a vacuum.
Why study programming languages?

• Help you choose a language
  – C vs. Modula-3 vs. C++ for systems programming
  – Fortran vs. APL vs. Ada for numerical computations
  – Ada vs. Modula-2 for embedded systems
  – Common Lisp vs. Scheme vs. ML for symbolic data manipulation
  – Java vs. C/CORBA for networked PC programs
The Art of Language Design
What is a language for?

• Way of thinking – way of expressing algorithms
• Languages from the user's point of view
• Abstraction of virtual machine – way of specifying what you want the hardware to do without getting down into the bits
• Languages from the implementer's point of view
Why are there so many?

• **Evolution** - learn better ways of doing things over time
  – goto-based control flow (Fortran)
  – structured programming (Pascal, C)
  – object-oriented structure (C++, Java)

• **Special purpose**
  – symbolic data
  – character strings
  – low-level system programming
  – reasoning, logical relation

• **Socio-economic factors** - proprietary interests, commercial advantage

• **Personal preference** - diverse ideas about what is pleasant to use

• **Special hardware**
What makes a language successful?

- **Expressive power** – easy to express things, to use once fluent (C, APL, Algol-68, Perl)
- **Ease of use for novice** – easy to learn (BASIC, Pascal, LOGO)
- **Ease of implementation** – (BASIC, Forth)
- **Standardization** – (C, Java)
- **Open source** - wide dissemination without cost (Pascal, Java)
- **Excellent compilers** – possible to compile to very good (fast/small) code (Fortran)
- **Patronage** - backing of a powerful sponsor (COBOL, PL/1, Ada, Visual Basic)
Language Evaluation Criteria
Language Evaluation Criteria

- **Readability**: the ease with which programs can be read and understood
- **Writability**: the ease with which a language can be used to create programs
- **Reliability**: conformance to specifications (i.e., performs to its specifications under all conditions)
Evaluation Criteria: Others

- Cost
  - the ultimate total cost
- Portability
  - the ease with which programs can be moved from one implementation to another
- Generality
  - the applicability to a wide range of applications
- Well-definedness
  - the completeness and precision of the language’s official definition
Evaluation Criteria: Readability

• **Overall simplicity**
  – A manageable set of features and constructs
  – Few feature multiplicity (means of doing the same operation)
  – Minimal operator overloading

• **Orthogonality**
  – A relatively small set of primitive constructs can be combined in a relatively small number of ways
  – Every possible combination is legal
  – Lack of orthogonality leads to exceptions to rules
  – Makes the language easy to learn and read
  – Meaning is context independent
Evaluation Criteria: Readability

• **Control statements**
  – The presence of well-known control structures (e.g., *while* statement)

• **Data types and structures**
  – The presence of adequate facilities for defining data structures

• **Syntax considerations**
  – Identifier forms: flexible composition
  – Special words and methods of forming compound statements
  – Form and meaning: self-descriptive constructs, meaningful keywords
Evaluation Criteria: Writability

- **Simplicity and orthogonality**
  - Few constructs, a small number of primitives, a small set of rules for combining them

- **Support for abstraction**
  - The ability to define and use complex structures or operations in ways that allow details to be ignored

- **Expressivity**
  - A set of relatively convenient ways of specifying operations
  - Example: the inclusion of `for` statement in many modern languages
Evaluation Criteria: Reliability

- **Type checking**
  - Testing for type errors

- **Exception handling**
  - Intercept run-time errors and take corrective measures

- **Aliasing**
  - Presence of two or more distinct referencing methods for the same memory location

- **Readability and writability**
  - A language that does not support “natural” ways of expressing an algorithm will necessarily use “unnatural” approaches, and hence reduced reliability
Evaluation Criteria: Cost

- Training programmers to use language
- Writing programs
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs
## Language Characteristics & Criteria

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Readability</td>
</tr>
<tr>
<td>Simplicity &amp; orthogonality</td>
<td>✓</td>
</tr>
<tr>
<td>Control statements</td>
<td>✓</td>
</tr>
<tr>
<td>Data types and structure</td>
<td>✓</td>
</tr>
<tr>
<td>Syntax design</td>
<td>✓</td>
</tr>
<tr>
<td>Support for abstraction</td>
<td>✓</td>
</tr>
<tr>
<td>Expressivity</td>
<td></td>
</tr>
<tr>
<td>Type checking</td>
<td></td>
</tr>
<tr>
<td>Exception handling</td>
<td></td>
</tr>
<tr>
<td>Restricted aliasing</td>
<td></td>
</tr>
</tbody>
</table>
Language Design Trade-Offs

• Reliability vs. cost of execution
  – Conflicting criteria
  – Example: Java demands all references to array elements be checked for proper indexing but that leads to increased execution costs

• Readability vs. writability
  – Another conflicting criteria
  – Example: APL provides many powerful operators (and a large number of new symbols), allowing complex computations to be written in a compact program but at the cost of poor readability

• Writability (flexibility) vs. reliability
  – Another conflicting criteria
  – Example: C++ pointers are powerful and very flexible but not reliably used
Programming Language Paradigms
Programming Paradigms

• **Imperative**
  – Central features are variables, assignment statements, and iteration
  – Examples: C, Pascal

• **Object-oriented**
  – Data abstraction (Encapsulate data objects with processing), inheritance, dynamic type binding
  – Examples: Java, C++

• **Functional**
  – Main means of making computations is by applying functions to given parameters
  – Examples: LISP, Scheme

• **Logic**
  – Rule-based (rules are specified in no particular order)
  – Example: Prolog

• **Markup**
  – New; not a programming per se, but used to specify the layout of information in Web documents
  – Examples: XHTML, XML
Programming Paradigms

• Imperative
  – Von Neumann (Fortran, Pascal, Basic, C)
  – Scripting (Perl, Python, JavaScript, PHP)
  – Object-oriented (Smalltalk, Eiffel, C++)

• Declarative
  – Functional (Scheme, ML, pure Lisp, FP)
  – Logic, constraint-based (Prolog, VisiCalc, RPG)
Programming Paradigms: Alternatives

• Imperative
  – Procedural (C)
    • Block-Structured (Pascal, Ada)
  – Object-based (Ada)
    • Object-oriented (Ada, Object-Pascal, C++, Java)
  – Parallel Processing (Ada, Pascal-S, Occam, C-Linda)

• Declarative
  – Logic (Prolog)
  – Functional (LISP, Scheme)
  – Database (SQL)
Example of GCD program

```c
int gcd(int a, int b) {
    while (a!=b) {
        if (a>b) a = a - b;
        else b = b - a;
    }
    return a;
}  //C
```

```scheme
(define gcd
  (lambda (a b)
    (cond ((= a b) a)
          ((> a b) (gcd (- a b) b))
          (else (gcd (- b a) a))))));
```

```prolog
gcd(A,B,G) :- A = B, G=A.
gcd(A,B,G) :- A > B, C is A-B, gcd(C,B,G).
gcd(A,B,G) :- B > A, C is B-A,
gcd(C,A,G).
%Prolog
```
Programming Paradigms: Emerging

- Event-driven/Visual
  - Continuous loop that responds to events
  - Code is executed upon activation of events
  - Subcategory of imperative
  - Examples: Visual Basic .NET, Java

- Concurrent
  - Cooperating processes
  - Examples: High Performance Fortran
Programming Domains

• **Scientific applications**
  – Large number of floating point computations
  – Fortran

• **Business applications**
  – Produce reports, use decimal numbers and characters
  – COBOL

• **Artificial intelligence**
  – Symbols rather than numbers manipulated
  – LISP

• **Systems programming**
  – Need efficiency because of continuous use
  – C

• **Web Programming**
  – Eclectic collection of languages: markup (e.g., XHTML), scripting (e.g., PHP), general-purpose (e.g., Java)
A Brief Historical Lineage of Some Key Programming Languages
Implementation Methods
Layered View of Computer

Virtual computer – the OS and language implementation which are layered over machine interface of a computer
Implementation Methods

• **Compilation**
  – Programs are translated into machine language

• **Pure Interpretation**
  – Programs are interpreted by another program known as an interpreter

• **Hybrid /Mixing**
  – A compromise between compilers and pure interpreters
Implementation Methods

• Interpretation:
  – Greater flexibility
  – Better diagnostics (error messages)

• Compilation
  – Better performance
Implementation Methods

• Compilation

• Interpretation

• Mixing
Implementation Methods

- Library routines and linking
- Post-compilation assembly
Overview of Compilation
Overview of Compilation

- Lexical analysis
- Syntax analysis
- Semantic Analysis & intermediate code generation
- Target code generation
- Code improvement
Overview of Compilation

• Lexical and Syntax Analysis

GCD Program (in C)

```c
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i - j;
        else j = j - i;
    }
    putint(i);
}
```
Overview of Compilation

• Lexical Analysis
  • Scanning and parsing recognize the structure of the program, groups characters into tokens, the smallest meaningful units of the program

GCD Program Tokens

```c
int main ( ) {
  int i = getint ( ), j = getint ( );
  while ( i != j ) {
    if ( i > j ) i = i - j ;
    else j = j - i ;
  }
  Putint ( i ) ;
}
```
Overview of Compilation

- Syntax Analysis
  - Context-Free Grammar and Parsing
    - Parsing organizes tokens into a parse tree that represents higher-level constructs in terms of their constituents
    - Potentially recursive rules known as context-free grammar define the ways in which these constituents combine
Overview of Compilation

• Context-Free Grammar and Parsing

Example (\texttt{while} loop in C)

\[
\text{iteration\text{-}statement} \rightarrow \text{while ( expression ) statement}
\]

statement, in turn, is often a list enclosed in braces:
\[
\text{statement} \rightarrow \text{compound\text{-}statement}
\]
\[
\text{compound\text{-}statement} \rightarrow \{ \text{block\text{-}item\text{-}list opt} \}
\]

where
\[
\text{block\text{-}item\text{-}list opt} \rightarrow \text{block\text{-}item\text{-}list}
\]
or
\[
\text{block\text{-}item\text{-}list opt} \rightarrow \epsilon
\]
and
\[
\text{block\text{-}item\text{-}list} \rightarrow \text{block\text{-}item}
\]
\[
\text{block\text{-}item\text{-}list} \rightarrow \text{block\text{-}item\text{-}list block\text{-}item}
\]
\[
\text{block\text{-}item} \rightarrow \text{declaration}
\]
\[
\text{block\text{-}item} \rightarrow \text{statement}
\]
Overview of Compilation

• Context-Free Grammar and Parsing

GCD Program Parse Tree
Overview of Compilation

- Context-Free Grammar and Parsing

```
declaration
  declaration-specifiers init-declarator-list_opt ;
    typeSpecifier declaration-specifiers_opt init-declarator-list
    int e init-declarator-list , init-declarator
      init-declarator declarator = initializer
        declarator = initializer
          pointer_opt direct-declarator assignment-expression e ident(j)
            postfix-expression
              postfix-expression ( )
                ident(getint) argument-expression-list_opt
                  e
```
Overview of Compilation

• Context-Free Grammar and Parsing
Overview of Compilation

• Syntax Tree - GCD Program Parse Tree
Programming Environments
Programming Environments

• The collection of tools used in software development

• UNIX
  – An older operating system and tool collection
  – Nowadays often used through a GUI (e.g., CDE, KDE, or GNOME) that run on top of UNIX

• Borland JBuilder
  – An integrated development environment for Java

• Microsoft Visual Studio.NET
  – A large, complex visual environment
  – Used to program in C#, Visual BASIC.NET, Jscript, J#, or C++
# Programming Environments

## Tools

<table>
<thead>
<tr>
<th>Type</th>
<th>Unix examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Editors</td>
<td><code>vi, emacs</code></td>
</tr>
<tr>
<td>Pretty printers</td>
<td><code>cb, indent</code></td>
</tr>
<tr>
<td>Pre-processors (esp. macros)</td>
<td><code>cpp, m4, watfor</code></td>
</tr>
<tr>
<td>Debuggers</td>
<td><code>adb, sdb, dbx, gdb</code></td>
</tr>
<tr>
<td>Style checkers</td>
<td><code>lint, purify</code></td>
</tr>
<tr>
<td>Module management</td>
<td><code>make</code></td>
</tr>
<tr>
<td>Version management</td>
<td><code>sccs, rcs</code></td>
</tr>
<tr>
<td>Assemblers</td>
<td><code>as</code></td>
</tr>
<tr>
<td>Link editors, loaders</td>
<td><code>Id, Id-so</code></td>
</tr>
<tr>
<td>Perusal tools</td>
<td><code>More, less, od, nm</code></td>
</tr>
<tr>
<td>Program cross-reference</td>
<td><code>ctags</code></td>
</tr>
</tbody>
</table>
Programming Languages: Trend
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Oct 2010</td>
<td>Oct 2009</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td>Java</td>
<td>18.166%</td>
<td>-0.48%</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td></td>
<td>C</td>
<td>17.177%</td>
<td>-0.33%</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>↑</td>
<td>C++</td>
<td>9.802%</td>
<td>-0.08%</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>↓</td>
<td>PHP</td>
<td>8.323%</td>
<td>-2.03%</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>(Visual) Basic</td>
<td>5.650%</td>
<td>-3.04%</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td></td>
<td>C#</td>
<td>4.963%</td>
<td>+0.55%</td>
<td>A</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td></td>
<td>Python</td>
<td>4.860%</td>
<td>+0.96%</td>
<td>A</td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>↑↑↑↑</td>
<td>Objective-C</td>
<td>3.706%</td>
<td>+2.54%</td>
<td>A</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>↓</td>
<td>Perl</td>
<td>2.310%</td>
<td>-1.45%</td>
<td>A</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td>Ruby</td>
<td>1.941%</td>
<td>-0.51%</td>
<td>A</td>
</tr>
<tr>
<td>11</td>
<td>9</td>
<td>↓</td>
<td>JavaScript</td>
<td>1.659%</td>
<td>-1.37%</td>
<td>A</td>
</tr>
<tr>
<td>12</td>
<td>11</td>
<td>↓</td>
<td>Delphi</td>
<td>1.558%</td>
<td>-0.58%</td>
<td>A</td>
</tr>
<tr>
<td>13</td>
<td>17</td>
<td>↑↑↑↑</td>
<td>Lisp</td>
<td>1.084%</td>
<td>+0.48%</td>
<td>A-</td>
</tr>
<tr>
<td>14</td>
<td>24</td>
<td>↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑</td>
<td>Transact-SQL</td>
<td>0.820%</td>
<td>+0.42%</td>
<td>A-</td>
</tr>
<tr>
<td>15</td>
<td>15</td>
<td></td>
<td>Pascal</td>
<td>0.771%</td>
<td>+0.10%</td>
<td>A-</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
<td>↑</td>
<td>RPG (OS/400)</td>
<td>0.708%</td>
<td>+0.12%</td>
<td>A-</td>
</tr>
<tr>
<td>17</td>
<td>29</td>
<td>↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑</td>
<td>Ada</td>
<td>0.704%</td>
<td>+0.40%</td>
<td>A-</td>
</tr>
<tr>
<td>18</td>
<td>14</td>
<td>↓</td>
<td>SAS</td>
<td>0.664%</td>
<td>-0.14%</td>
<td>B</td>
</tr>
<tr>
<td>19</td>
<td>19</td>
<td></td>
<td>MATLAB</td>
<td>0.627%</td>
<td>-0.05%</td>
<td>B</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑↑</td>
<td>Go</td>
<td>0.626%</td>
<td>-0.63%</td>
<td>B</td>
</tr>
</tbody>
</table>
References

Books


Interesting links