Design & Implementation Overview

COS 301

Fall 2017
Outline

1. Influences on language design
2. Language as VM
3. Compilation
4. Interpretation
5. Hybrid implementation
6. Preprocessors
7. Programming environments
Influences on language design
Ontological commitments

Imperative languages

- Architecture:
  - Memory cells $\rightsquigarrow$ variables
  - Data movement (memory $\rightarrow$ memory, CPU $\rightarrow$ memory) $\rightsquigarrow$ assignment
  - Sequential machine instruction execution $\rightsquigarrow$ sequential statements
  - Conditional execution $\rightsquigarrow$ if–then–else constructs, goto
  - Iteration via conditionals + jump $\rightsquigarrow$ loops
Ontological commitments

Functional languages

- Variables “standing for” value \( \rightarrow \text{binding} \), not pointers/addresses
- Function application \( \rightarrow \) produce new values \( \rightarrow \)
  - No notion of “executing” sequential statements
  - No statements, only functions (with values)
  - Function composition as major characteristic
  - Recursion as primary way of repeating function application
Ontological commitments

Object-oriented languages

- World consists of objects \(\leadsto\) classes, instances, inheritance, instantiation
- No notion of address: variables hold value or \textit{references}
- \(\exists\) classes of objects \(\leadsto\) inheritance, instantiation
- Instance variables as properties or relations to other objects
- Objects \textit{affordances} (things they can do) \(\leadsto\) methods
Factors affecting design: early (< mid-1960s)

- Computer time extremely valuable
  - $\gg$ programmers’ time
  - $\leadsto$ languages tailored toward machine, not humans
- Computers relatively slow
  - Thousands–millions of instructions/s (kIPS – MIPS)
  - E.g.: IBM 360 mainframe, mid-60s, \(~34\) kIPS – \(~17\) MIPS
  - $\leadsto$ extreme concern for efficiency
    - $\leadsto$ compilation rather than interpretation
    - $\leadsto$ simple languages
- Relatively simple applications $\leadsto$ small programs
Factors affecting design: late 60s–mid-70s

- Cheaper processors $\sim$ cost of programmer time $\gg$ computer time
- Demand for capable/sophisticated software applications $\sim$
  - More programming time
  - Larger programs $\sim$ harder to design, debug, maintain

Result:
- Focus on
  - Human-friendly languages
  - Languages supporting design, debugging, maintenance
- **Structured programming:**
  - Top-down design
  - Stepwise refinement
  - More sophisticated control structures
- Prominence of ALGOL-like languages (PL/I, C, Pascal, etc.)
Factors affecting design: more recently

- Data abstraction (Modula-2, Ada, etc.)
- Object-orientation
  - Revived early work on CLU, Smalltalk, etc.
  - C++, Objective-C, Java . . .
- More powerful computers
  - More sophisticated compilers possible
  - Practical interpreters
- Widespread availability of multi-core systems, clusters

new languages (C*, StarLisp, Parallel Euclid, . . .)
Language as VM
Virtual machine

- Programming language $\Rightarrow$ virtual machine
- VM can be implemented as a compiler, interpreter or a hybrid
Virtual machine: layers
Compilation
Compilation

- **Compiler**: program that translates HLL ⇒ *object code*
  
- **Link editor**:
  - Gathers multiple object modules (e.g., subprograms, libraries)
  - Patches (links) unresolved references in object modules
  - ⇒ *executable*

- **Loader**:
  - Part of OS
  - Allocates (virtual) memory
  - Loads (copies) executable file into (virtual) memory
  - May treat parts of executable differently
  - May create memory not present in executable (heap, uninitialized data)
Overview

- Source (HLL) program → lexical analyzer
  - ⇒ lexical units
  - Updates symbol table
- Lexical units → syntax analyzer
  - Checks syntax for errors
  - Updates symbol table
  - ⇒ parse tree
- Parse tree → intermediate code generator
  - Semantic analyzer
  - ⇒ intermediate code
  - Interacts with optimizer
- Intermediate code → code generator
  - ⇒ object code file
  - Machine language program
  - May have unresolved references
Process

Source Code

Lexical Analyzer

lexical units

Syntax Analyzer

parse tree

Symbol Table

Intermediate Code Generator

Optimizer

intermediate code

Code Generator

Object File

Design & Implementation Overview

Influences on language design

Language as VM

Compilation

Interpretation

Hybrid implementation

Preprocessors

Programming environments
Properties

- Fast execution
  - Running at native machine speed
  - Optimizer \( \Rightarrow \) often faster than hand-coded assembly
- Compiler has access to entire program at once
  - Can do global optimizations
  - Can have complex languages, easy forward references, etc.
- Possibly lengthy compilation time
  - Amortized over execution times, ameliorated by faster machines
  - But during debugging/rapid prototyping
    - compile–test cycle cumbersome
    - source level debugging somewhat difficult
    - hard to change part without recompiling whole
Interpretation
Interpretation

- **Interpreter**: Program that reads source code and carries out actions
- One of the very first: Lisp
- No translation of HLL to machine code
- Supports rapid prototyping
- Need significant runtime environment (i.e., the interpreter)
- Slower execution (10–100 times as slow as executable)
- Historically rare for traditional HLLs (though Lisp, Scheme)
- Now: Python, JavaScript, PHP, …
Hybrid implementation
Hybrid implementations

- Compromise between compilation and interpretation
- One way: HLL translated to intermediate language that is easy to interpret
  - Faster than pure interpretation
  - E.g., Perl, Java, Smalltalk, Microsoft Common Language Runtime
- Another way:
  - Allow both compiled and interpreted code
  - E.g., most Common Lisp systems, some of Perl
Just-in-Time (JIT) compilers

- Compile to byte code first (e.g., Java byte code)
- When subprograms called, byte code compiled to machine code
- Machine code kept for subsequent calls
- JIT used for Java, .NET languages
- Makes Java competitive with fully-compiled languages
Preprocessors
Preprocessors

- Preprocessor instructions:
  - handled immediately prior to compilation...
  - ...or prior to loading code in interpreter

- Types:
  - include other code (e.g., C’s `#include`)
  - macro commands (e.g., C’s `#define`)
  - templates (e.g., C++, for generic classes)
  - more complex macros: e.g., Lisp’s `defmacro`
Programming environments
Programming environments

- Collection of tools used for software development
- Compilers, editors, debuggers, profilers, linkers, etc.
- E.g., Unix
  - Command line tools (e.g., make, grep, awk, sed, gcc)
  - Editors (e.g., Emacs, vi) and IDEs
Integrated development environments (IDEs)

Includes a compiler, linker, debugger, editor, and build automator

May also include source control system, class browser, object inspector, profiler, etc.

Some support multiple languages, others single language

Examples:

- PyCharm
- Eclipse, Emacs
- Netbeans
- Xcode
- MonoDevelop
- Lisp machine, modern Lisp and Scheme IDEs (e.g., Allegro, PLT Scheme/Racket)
Microsoft .NET

- Collection of languages, technologies, development environment
- Most common: C++, C#, VB... – dozens available
- Large, complex visual environment (though command line available)
- .NET SDK available as free download
- Output language: machine-independent byte code for the Common Language Runtime
NetBeans

- Java answer to .NET
- Used for Java, but also supports C, PHP, Ruby, C++, others
- Written in Java
- Extensible via modules