COS 140: Foundations of Computer Science

Programming Languages

Fall 2018

Computer Science Foundations

Problem

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- Assembly language *much* better than machine language for programming
  - Mnemonics for op codes (e.g., ADD)
  - Symbolic addresses (memory and registers)
  - Rudimentary control structures via macros in some assemblers: if, loop
- But still basically one-to-one correspondence with machine language
- *Very* low-level

Computer Science Foundations

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

Problem

- Many instructions needed to do one conceptual step – E.g., want to set C = A + B – something like:

```
LD R1,A
LD R2,B
ADD R1,R2    ; result in R1, say
ST R1,C
```

- Requires programmer to think at very low level
- Tedious to program
- Prone to errors
- No type checking
- No automatic optimization
- Solution: *High-level programming languages*

Computer Science Foundations

What is a Programming Language?

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- A way to communicate with the computer.
  - Allows users to think about the computer in a way that is natural for them.
  - *Formal language* so it can be easily interpreted by the computer.

First steps

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- FORTRAN – 1957 (John Backus)
- LISP – 1958 (John McCarthy)
- COBOL – 1959 (Grace Hopper)
- Algol – 1960 (proposed 1958; John Backus, Peter Naur, others)

Issues for the Study of Programming Languages

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- Constructs that are available (or needed) in programming languages.
- Specifics of existing languages (to understand ramifications of design decisions, not to simply learn the language).
- Paradigms for programming languages.
- Formal methods for describing syntax and semantics.
- Implementation issues for interpreting the languages by the computer and supporting constructs.

Computer Science Foundations

Why are There So Many Languages?

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- Limitations of current languages give rise to new languages.
- New technology (speed, cost of computers as well as language implementation technology) makes new languages possible.
- Different languages are suited for different tasks (even among "general purpose languages").

Computer Science Foundations

Some Languages in Use

Introduction
- Problem
- What is a Programming Language?
- First steps
- Issues
- Why so many?
- Example languages

Programming Language Paradigms

Language Translation

Evaluation

Abstraction

Next Up

- General-purpose languages: C, C^{++}, Java, Ada, Visual Basic, Python, Lisp
- Languages for specific domains and tasks:
  - Scientific applications - FORTRAN (and now C/C^{++})
  - Business applications - COBOL
  - Artificial intelligence - Lisp and Scheme, Prolog
  - Systems programming - C, PL/I
  - Scripting languages - tcl, Perl, PHP
  - Teaching programming - Pascal, Modula
  - Web-oriented languages - JavaScript, PHP, Java
  - Simulation: Simula, GPSS, SNOBOL
  - Statistical analysis: SAS
  - Mathematics: APL (also Mathematica/Mathcad “languages”; Lisp for symbolic computation)
  - Mobile apps: Python, Java, Objective-C, Swift

Paradigms of Programming Languages

Introduction
Programming Language Paradigms
- Imperative Languages
- Functional Languages
- Logic Languages
- Object-Oriented Languages

Language Translation
Evaluation
Abstraction
Next Up

- High-level languages (as opposed to assembly language) give users an abstraction from the details of the machine and the CPU.
- *Paradigm*: way of thinking about how the programming language works.
- Paradigms in general:
  - Give the paradigm-holder a way of looking at the world.
  - Promote certain ways of thinking.
  - Make other ways of thinking more difficult.

Imperative Languages

Introduction
Programming Language Paradigms
- Imperative Languages
- Functional Languages
- Logic Languages
- Object-Oriented Languages

Language Translation
Evaluation
Abstraction
Next Up

- Based on von Neumann architecture.
  - Data has a location in memory (variables).
  - *Assignment* allows data to be stored at some location.
  - *Iteration* as a way of doing repetitive steps – corresponds to executing a sequence of machine instructions multiple times in a loop.
- Model is an abstraction of the actual machine $\Rightarrow$ helps with efficient programming and systems programming
- Examples: C, Python, Pascal, FORTRAN

Computer Science Foundations

Functional Languages

Introduction
Programming Language Paradigms
- Imperative Languages
- Functional Languages
- Logic Languages
- Object-Oriented Languages

Language Translation
Evaluation
Abstraction
Next Up

- Modeled on functions from mathematics.
  - Apply *functions* to values - not necessarily memory locations.
  - *Recursion* is method of iteration.
- Ignore constraints of von Neumann architecture.
- Assumes that people think in terms of mathematical functions "naturally".
- Examples: Lisp, Scheme, ML

Computer Science Foundations

Logic Languages

Introduction
Programming Language Paradigms
- Imperative Languages
- Functional Languages
- Logic Languages
- Object-Oriented Languages

Language Translation
Evaluation
Abstraction
Next Up

- Based on some form of formal logic.
  - Expressions written in logical formalism.
  - Processing done as *theorem proving*.
- Assumes that people think in terms of first order predicate calculus “naturally”.
- Example: Prolog

Object-Oriented Languages

Introduction
Programming Language Paradigms
- Imperative Languages
- Functional Languages
- Logic Languages
- Object-Oriented Languages

Language Translation
Evaluation
Abstraction
Next Up

- Data and related functions are grouped together as *objects*.
  - Processing is tied to specific data types.
  - Similarities and differences between types of data, including what you want to do with them, becomes focus.
- Can be a paradigm for a whole language or an add-on to an existing language.
- Examples: Smalltalk, C++, Java, Lisp/CLOS, Python, Perl, Visual Basic

Computer Science Foundations

Language Translation

Introduction
Programming Language Paradigms
Language Translation
- Compiling
- Compilation steps
- Interpretation

Evaluation
Abstraction
Next Up

- Needed to change the high-level language into instructions the computer can carry out.
- Two types: compiling and interpreting

Translation by Compiling

Introduction
Programming Language Paradigms
Language Translation
- Compiling
- Compilation steps
- Interpretation

Evaluation
Abstraction
Next Up

- Creates a machine language program that carries out the program in the higher-level language.
- Need to have access to much of the program to make necessary decisions.
  - May need to re-compile large portions (or all) of a program to make small changes.
- Compiled code runs fast because it is at the machine level. (This code can also be *optimized*.)

Introduction
Programming Language Paradigms
Language Translation
- Compiling
- Compilation steps
- Interpretation

Evaluation
Abstraction
Next Up

Steps of Compilation

1. Lexical analysis - break program into lexical units and classify by type
2. Syntactic analysis - identify structure of statement or find syntax errors
3. Intermediate code generation - produce code that can be used by humans and machines
4. Optimization - make intermediate code more efficient by finding specific patterns, applying refinements
5. Machine code generation - converts intermediate code to machine code
6. Linking - linker links machine code with necessary system calls, libraries, etc.
7. Executable image - machine instructions + system calls

The language is designed so that all steps can be automated.

Translation by Interpretation

Introduction
Programming Language Paradigms
Language Translation
- Compiling
- Compilation steps
- Interpretation

Evaluation
Abstraction
Next Up

- Interpreter carries out high-level commands directly.
- Debugging is easier than with compiler because source code which produced the error is available.
- Don't have to recompile to make small changes.
- Slower for execution because must interpret commands each time used and cannot optimize.
- Cannot use knowledge of whole program, so language must have simple structure.

How to Evaluate a Language

Introduction
Programming Language Paradigms
Language Translation
Evaluation
- How to evaluate
- Evaluation criteria

Abstraction
Next Up

- Use agreed-upon criteria.
- There may be a trade-off between different criteria.
- Must be applied depending on the use of the language (users, project, etc.).

Some Criteria for Evaluating Languages

Introduction
Programming Language Paradigms
Language Translation
Evaluation
- How to evaluate
- Evaluation criteria

Abstraction
Next Up

- Writability/Readability
  - Is it simple and natural?
  - Does it allow the user to do what is needed?
- Orthogonality
  - Are there a small number of primitive constructs?
  - Can all constructs be used in the same way?
  - Can take this too far. Still may need special cases and want to make sure that don't have too many options.

Some Criteria for Evaluating Languages (cont'd)

Introduction
Programming Language Paradigms
Language Translation
Evaluation
- How to evaluate
- Evaluation criteria

Abstraction
Next Up

- Are appropriate *control structures* and *data structures* provided by the language?
- Does the syntax help the programmer write clearly instead of posing obstacles to clear writing?
- Do features exist which increase the likelihood that code will not contain errors (type checking, etc.)?
- Is the language portable?
- What is the cost of using the language (including: training programmers, writing code, compiling and executing code, maintaining code)?

Abstraction in Programming Languages

Introduction
Programming Language Paradigms
Language Translation
Evaluation
Abstraction
Next Up

- Programming languages abstract the details of the machine from the user.
- Some constructs follow abstraction in processing that most people use (e.g., conditionals, loops).
- Some constructs help users build abstractions which can be used throughout the program.
  - *Subroutines* - allow user to abstract processing.
  - User-defined *data types* - allow user to abstract data by functional type.
  - *Data encapsulation* - allow user to group together by function data and ways to process it.
  - *Data hiding* - allow only the routines that must access data to access it.

Computer Science Foundations

What's next in this section?

Introduction
Programming Language Paradigms
Language Translation
Evaluation
Abstraction
Next Up

- Variables and data types
- Control structures
- Backus–Naur form and parsing

Computer Science Foundations

