Welcome to CS560
Programming Languages

Dr. Amie Souter

What is a programming language?

- Break into groups of 5.
- Define: programming language
- List as many programming languages as you can.

What is a programming language?

- **Bruce MacLellan's definition:**
  - A language that is intended for the expression of computer programs and that is capable of expressing any computer program.
- **Ravi Sethi's definition:**
  - Notations used for specifying, organizing, and reasoning about computations.
- **Donald Knuth's definition:**
  - *Programming is the art of telling another human being what one wants the computer to do.*

Some other definitions

- Programming language: An artificial language that is used to generate or to express computer programs. *Note:* The language may be a high-level language, an assembly language, or a machine language. (glossary telecommunication terms)
- A formal **language** in which computer **programs** are written. (hyperdictionary.com)
- A notational system for describing computation in machine-readable and human-readable form. (Louden)
- Any notation for the description of algorithms and data structures that can be implemented on a computer. (Pratt)
- Languages are collections of abstractions: collections of common patterns that programmers can combine into working programs.

Why Are There So Many Programming Languages?

- Evolution
  - We have learned better ways of doing things over time
- Socio-economic factors
  - Proprietary interests, commercial advantage
- Special purposes
- Special hardware
- Personal preference

What Makes a Language Successful?

- Easy to learn
  - Basic, Pascal
- Easy to express things
  - "Powerful"
  - Easy to use once known
  - C, Lisp
- Easy to implement
  - Small machines, limited resources
  - Basic, Forth

What Makes a Language Successful?

- Efficient compilers
  - Generate small / fast code
  - Fortran
- Backing of a powerful sponsor
  - Ada, Visual Basic
- Wide dissemination at a minimal cost
  - Pascal, Java
- Inertia
  - Fortran, Cobol

Course Information

- Instructor: Dr. Amie Souter
  - E-mail: souter@cs.drexel.edu
  - Office: Korman 255
  - Office Hours: Monday, Wednesday 2:30pm – 3:30pm
- Teaching assistant: Jeff Abrahamson
  - E-mail: jeffa@cs.drexel.edu
  - Office: Korman 222
- Class web page:
  - http://wwww.cs.drexel.edu/~souter/cs560/w04

Assignments and Exams

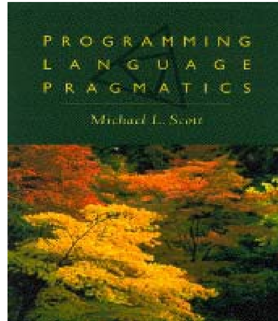
- Programming & Homework assignments (50%)
  - Written exercises
  - Group programming assignments
- Midterm exam (20%)
  - Closed book, closed notes
- Final exam (25%)
  - Closed book, closed notes
  - Comprehensive, although focused on the second half of the course
- Attendance and Class Participation (5%)
  - Attendance is expected
  - Active participation is also excepted

Late Submission Policy

- YOU ARE GRANTED 2 LATE DAYS
  - Use them as you see fit.
  - Send email to TA to let him know.
  - After 2 days are used up, assignments will not be accepted.
  - Read course syllabus for details!!

Required Textbook

- Programming Language Pragmatics,
  by Michael L. Scott, Morgan Kaufmann Press, 2000.



Questionnaire

Assignment 0

- Optional assignment.
  - www.cs.drexel.edu/~souter/cs560/w04
    - Assignments

Why do we have programming languages? – 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 implementor's point of view
- This course tries to balance coverage of these two angles. We will talk about language features for their own sake, and about how they can be implemented.

Why Study Programming Languages?

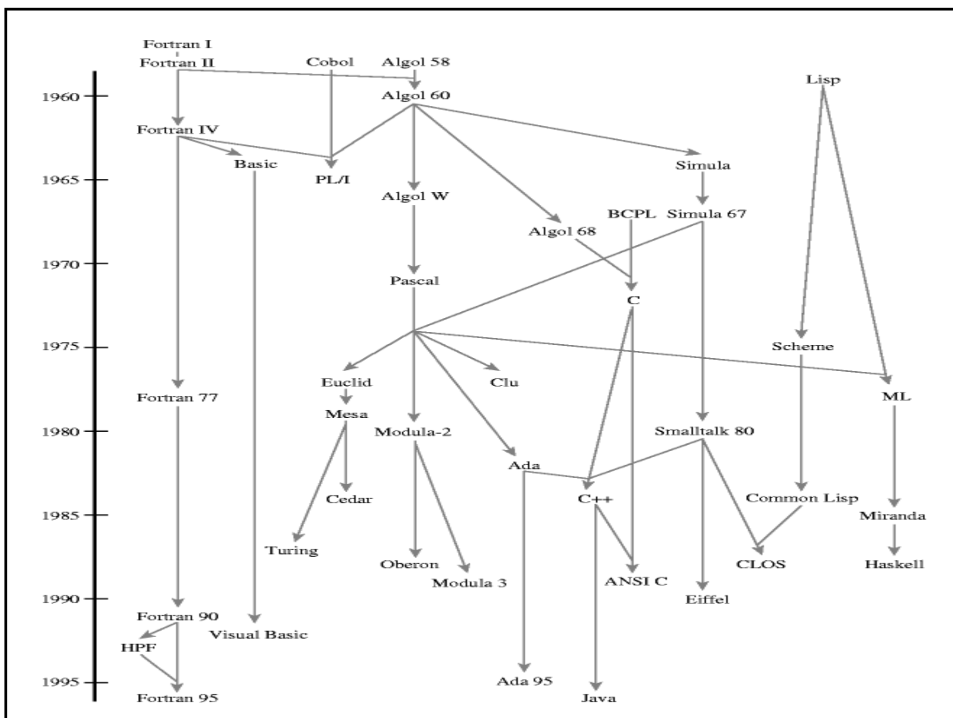
- Help you choose a language
  - Systems programming → C, Modula-3
  - Scientific computations → Fortran, Ada
  - Embedded systems → C, Ada, Modula-2
  - Symbolic computing → Lisp, Scheme, ML
- Make it easier to learn new languages
  - Some languages are similar
  - Some concepts are even more similar: iteration, recursion, abstraction

Why Study Programming Languages?

- Make better use of the languages you know
  - Improve ability to develop effective algorithms
  - Understand obscure features
  - Understand implementation costs
  - Simulate useful features in languages that do not support them
- Prepare for further study in language design and implementation (compilers)
- Help understand other system software (assemblers, linkers, debuggers)

Historic Perspective

- When did the earliest high-level languages appear?
  - Continuous evolution since the 1950s (Fortran, Cobol)
- When the Department of Defense did a survey as part of its efforts to develop Ada in the 1970s, how many languages did it find in use?
  - Over 500 languages were being used in various defense projects



Attributes of a Good Language

- Clarity, simplicity, and unity
  - All are desirable, but in conjunction
  - Using a very simple language in an application for which it is not well suited may produce lack of clarity
- Orthogonality
  - Every combination of features is meaningful
  - Example: data types and return values are not orthogonal in C, but are in ML
- Naturalness
  - program structure reflects the logical structure of algorithm

Attributes of a Good Language

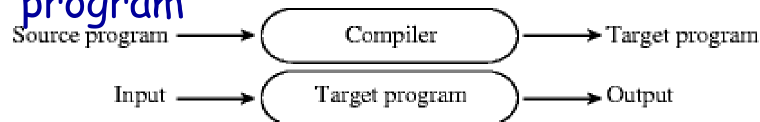
- Support for abstraction
  - program data reflects problem being solved
- Reliability of programs
  - Does the program behave the same way every time it is run with the same input data?
- Cost of use:
  - Cost of program creation
  - Cost of program translation
  - Cost of program execution
  - Cost of program maintenance

Spectrum of Languages

- Imperative ("how should the computer do it?")
  - Von Neumann: Fortran, Basic, Pascal, C
    - Computing via "side-effects" (modification of variables)
  - Object-oriented: Smalltalk, Eiffel, C++, Java
    - Interactions between objects, each having an *internal state* and *functions* which manage that state
- Declarative ("what should the computer do?")
  - Functional: Lisp, Scheme, ML, Haskell
    - Program ↔ application of functions from inputs to outputs
    - Inspired from *lambda-calculus* (Alonzo Church)
  - Logic, constraint-based: Prolog
    - Specify constraints / relationships, find values that satisfy them
    - Based on *propositional logic*

Compilation and Interpretation

- *A* **compiler** is a *program* that **translates** high-level source programs into target program



- An interpreter is a program that **executes** another program

Groups

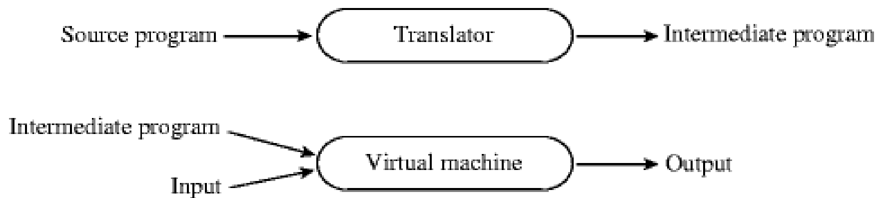
- List two compiled languages.
  - Why are these languages compiled?
- List two interpreted languages.
  - Why are these languages interpreted?
- **Class discussion:** Half the class will represent the advantages of **interpretation**, the others the advantages of **compilation**. Which is better?

Compilation and Interpretation

- Interpretation
  - Greater flexibility, better diagnostics
  - Allow program features (data types or sizes) to depend on the input
  - Lisp, Prolog: program can write new pieces of itself and execute them on the fly
  - Java: compilation, then interpretation (JIT)
- Compilation
  - Better performance
  - Many decisions are made only once, at compile time, not at every run time
  - Fortran, C

Mixing Compilation and Interpretation

- Fuzzy difference:
  - A language is **interpreted** when the initial translation is *simple*
  - A language is **compiled** when the translation process is *complicated*

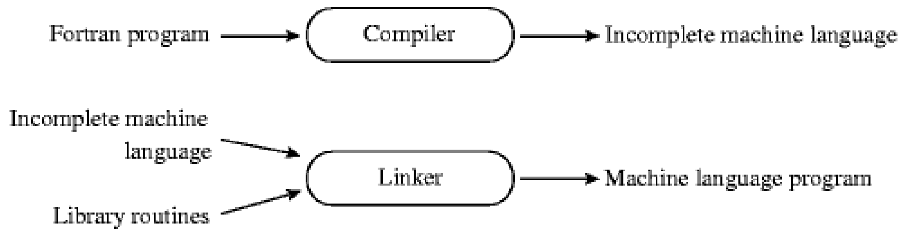


Preprocessing

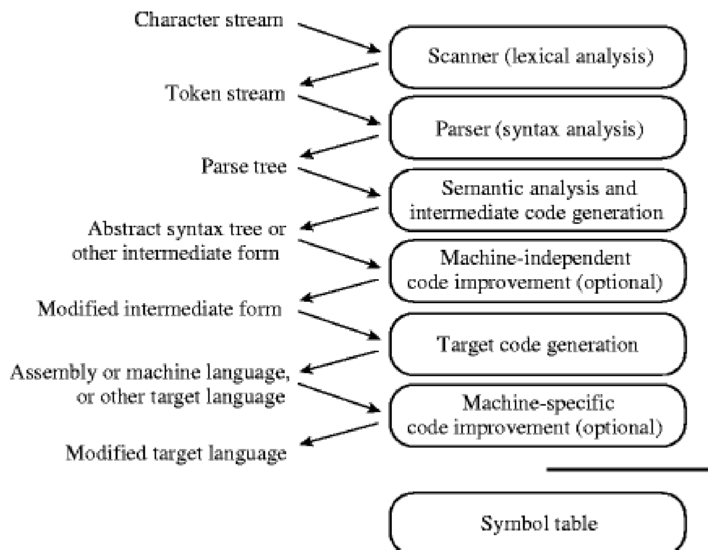
- Macros
  - #define <macro> <replacement name>
  - `#define FALSE 0`
  - `#define max(A,B) ( (A) > (B) ? (A):(B))`

Linking

- Libraries of subroutines



Phases of Compilation



Example

- Example program:

```
read A
read B
sum := A + B
write sum
write sum / 2
```


Lexical Analysis

- Tokens: scanner extracts token (smallest meaningful units)

  id = letter (letter | digit) * [except "read" and "write"]
  literal = digit digit *
  ":=", "+", "-", "*", "/", "(", ")"
  $$$ [end of file]

Syntax Analysis

- Grammar in EBNF

```
<pgm>         ->  <statement list> $$$
<stmt list>   ->  <stmt list> <stmt> | E
<stmt>        ->  id := <expr> | read <id> | write <expr>
<expr>        ->  <term> | <expr> <add op> <term>
<term>        ->  <factor> | <term> <mult op> <factor>
<factor>      ->  ( <expr> ) | id | literal
<add op>      ->  + | -
<mult op>     ->  * | /
```


Semantic Analysis

- Discovers meaning in a program
- Static semantic analysis (at compile time):
  - Identifiers declared before use
  - Subroutine calls provide correct number and type of arguments
- Dynamic semantics (cannot be checked at compile time, so generate code to check at run time):
  - Pointers dereferenced only when refer to a valid object
  - Array subscript expressions lie within bounds
- Simplifies the parse tree → syntax tree
- Maintains symbol table, attaches attributes

Code Generation

- Intermediate code:

```
read
pop A
read
pop B
push A
push B
add
pop sum
push sum
write
push sum
push 2
div
write
```

read A
read B
sum := A + B
write sum
write sum / 2

Code Generation

- Target code:

```
        .data
A:      .long 0
B:      .long 0
sum:    .long 0
        .text
main:   jsr read
        movl    d0,d1
        movl    d1,A
        jsr read
        movl    d0,d1
        movl    d1,B
        movl    A,d1
```

```
movl    B,d2
addl    d1,d2
movl    d1,sum
movl    sum,d1
movl    d1,d0
jsr write
movl    sum,d1
movl    #2,d2
divsl   d1,d2
movl    d1,d0
jsr write
```

Back-end

- Target code generation
  - Generate assembly or machine language
  - Traverses the syntax tree to generate elementary operations: loads and stores, arithmetic operations, tests and branches
- Code improvement (optional)
  - A.k.a "optimization"
  - Transform program into a new version with same functionality, but more efficient (faster, uses less memory)

Summary

- What is a programming language?
- Why study programming languages?
- What kinds of programming languages exist?
- How is a programming languages implemented?

Reading & HW

- Chapter 1
- Exercise 1.3 in the book