Lecture 1: Introduction and Overview
Introduction

- Eight lectures covering:
  - Basic programming language concepts
  - Organization of compilers for modern programming languages
  - Introduction to the theory of formal languages and automata

- Text:
  - *Programming Language Pragmatics* by Michael L. Smith
Abstractions...

- Eliminate detail unnecessary for solving a particular problem
  - Complexity is hidden
- Often build upon one another
  - Allow us to solve increasingly complex problems
- Modern software's complexity is without precedent
  - Abstractions are absolutely necessary to manage this complexity

<table>
<thead>
<tr>
<th>Abstraction</th>
<th>Abstracts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digital logic</td>
<td>Transistors</td>
</tr>
<tr>
<td>Computer architecture</td>
<td>Digital logic</td>
</tr>
<tr>
<td>Assembly language</td>
<td>Machine language</td>
</tr>
<tr>
<td>Operating systems</td>
<td>Resource allocation (time, storage, etc.)</td>
</tr>
<tr>
<td>Computer communications</td>
<td>Physical networks, protocols</td>
</tr>
</tbody>
</table>
Languages as abstraction

- Human languages are a tool for abstracting thought
  - “When I am warm I turn on the fan”
    - Communicates a simple intention, whereas the cognitive and neurological conditions from which the intention arose are most likely too complex for anyone to understand
    - Meaning of this statement is left to the understanding of the individual who utters it and the individuals who hear it

- Programming languages are a tool for abstracting computation
  - if (temperature() > 30.0) { turn_on_fan(); }
    - Involves a complex, but concrete sequence of actions:
      - read the thermostat; convert the reading to an IEEE floating point value on the centigrade scale; compare the value to 30.0; if greater then send a command to a PCI card which sends a signal to a relay which then turns on the fan
    - Meaning of this statement is fixed by the formal semantics of the programming language and by the implementations of the functions temperature() and turn_on_fan()
How do programming languages abstract computation?

1) Provide a notation for the expression of algorithms that:
   - Is (mostly) independent of the machine on which the algorithm will execute
   - Provides high-level features that focus the programmer’s attention more on the algorithm and less on:
     - How the algorithm will be implemented in a machine language
     - Implementing auxiliary algorithms, e.g., hash tables
   - Allows the programmer to build her own abstractions (subroutines, modules, libraries, classes, etc.) thus allowing the continuation of the “complexity management by layering” principle
How do programming languages abstract computation?

- 2) Hide low-level details of the target architecture
  - Assembly language instruction names, register names, argument ordering, ...
  - Mapping of language elements to assembly language
    - Arithmetic expressions,
    - Conditionals,
    - Procedure calling conventions, ...

```java
if (a < b + 10) {
    do_1();
} else {
    do_2();
}
```

```sparc
add   %11,10,%12
cmp   %10,%12
bge   .L1;   nop
call  do_1;   nop
ba    .L2;   nop  
.L1:   call    do_2;   nop  
.L2:   ...
```

```mips
addi   $t2,$t1,10
bge    $t0,$t2,L1
call   do_1
b     L2  
L1:    call    do_2  
L2:    ...
```
How do programming languages abstract computation?

3) Provide primitives, subroutines, and run-time support for common programming chores

Examples
- Reading and writing files
- Character string handling (comparison, substring detection, etc.)
- Dynamic memory allocation (new, malloc, etc.)
- Reclamation of unused memory (garbage collection)
- Sorting
How do programming languages abstract computation?

4) Provide features that encourage or enforce a particular style of algorithm or software development

<table>
<thead>
<tr>
<th>Structured programming</th>
<th>Object Oriented Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subroutines</td>
<td>Classes</td>
</tr>
<tr>
<td>Nested variable scopes</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Loops</td>
<td>Polymorphism</td>
</tr>
<tr>
<td>Restricted forms of the “goto” statement</td>
<td></td>
</tr>
</tbody>
</table>
Categories of programming languages

- All languages fall into one of the following two categories:
  - **Imperative languages** require programmers to describe, step-by-step, how an algorithm should do its work
    - Real world analogy » A recipe is a type of imperative program that tells a cook how to prepare a dish
  - **Declarative languages** allow programmers to describe what an algorithm should do without having to specify exactly how it should do it
    - Real world analogy » The recent Pfand law is a declarative program that tells retailers that they must institute a recycling program for the einweg Flasche and Dosen that they sell, without telling them exactly how to do it
Imperative Languages

- The von Neumann languages
  - Include Fortran, Pascal, Basic, C
  - Are a reflection of the von Neumann computer architectures on which their programs run
  - Execute statements which alter the program’s state (variables/memory)
    - Sometimes called computing by side effects
  - Example: summing the first n integers in C
    - for(sum=0,i=1;i<=n;i++) { sum += i; }

Imperative Languages

- The object oriented languages
  - Include Smalltalk, Eiffel, C++, Java
  - Are similar to the von Neumann languages with the addition of objects

- Objects:
  - Contain their own internal state (member variables) and functions which operate on that state (methods)
  - Computation is organized as interactions between objects (one object calling another’s methods)

- Most object oriented languages provide facilities based on objects that encourage object oriented programming
  - Encapsulation, inheritance, and polymorphism
  - We’ll talk about these more in a future lecture
Declarative Languages

- The functional languages
  - Include Lisp/Scheme, ML, Haskell
  - Are a reflection of Church’s theory of recursive functions (lambda calculus)
  - Computation carried out as functions return values based on the (possibly recursive) evaluation of other functions
    - Mechanism known as reduction
  - No side effects!
    - Permits equational reasoning, easier formal proofs of program correctness, etc.
  - Example: summing the first n integers in SML
    - `fun sum (n) = if n <= 1 then n else n + sum(n-1)`
Declarative Languages

- The logic languages
  - Include Prolog, SQL, and Microsoft Excel
  - Are a reflection of the theory of propositional logic
  - Computation is an attempt to find values which satisfy a set of logical relationships
    - Mechanism most commonly used to find these values is known as resolution and unification
  - Example: summing the first n integers in Prolog
    - sum(1,1).
    - sum(N,S) :- N1 is N-1, sum(N1,S1), S is S1+N.
A historical perspective: Machine languages

- First machines programmed directly in machine language or machine code
- Tedious, but machine time more expensive than programmer time

MIPS machine code for a program that computes the GCD of two integers

```
27bdff0 0bf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c
00401825 10820008 0064082a 10200003 00000000 10000000 00832023
00641823 1483ffe1 0064082a 0c1002b2 00000000 8fa00014 27bd0020
03e00008 00001025
```
A historical perspective: Assembly languages

- Programs became more complex
  - Too difficult, time-consuming, and expensive to write them in machine code
- Assembly languages developed
  - Human readable
  - Originally provided a one-to-one correspondence between machine language instructions and assembly language instructions
  - Eventually “macro” facilities were added to further speed software development by providing a primitive form of code-reuse
  - The assembler was the program which translated an assembly language program into machine code which the machine could run

```assembly
addiu sp, sp, -32
sw ra, 20(sp)
jal getint
nop
jal getint
sw v0, 28(sp)
lw a0, 28(sp)
move v1, v0
beq a0, v0, D
slt at, v1, a0
A: beq at, zero, B
nop
b
C
subu a0, a0, v1
B: subu v1, v1, a0
C: bne a0, v1, A
slt at, v1, a0
D: jal putint
nop
lw ra, 20(sp)
addiu sp, sp, 32
jr ra
move v0, zero
```

Assembler
A historical perspective: High-level languages

- Programs became even more complex
  - Too difficult, time-consuming, and expensive to write in assembly language
  - Too difficult to move from one machine to another with a different assembly language
- High-level languages developed
  - Mid 1950's Fortran was designed and implemented
    - Allowed numerical computations to be expressed in a form similar to mathematical formulae
  - The compiler was the program which translated a high-level program into an assembly language or machine language program
    - Originally, good programmers could write faster assembly language programs than the compiler
  - Other high-level languages followed Fortran in the late 50's and early 60's
    - Lisp: first functional language, based on recursive function theory
    - Algol: first block-structured language

```c
int gcd (int i, int j) {
    while (i != j) {
        if (i > j)
            i = i - j;
        else
            j = j - i;
    }
    printf("%d\n",i);
}
```
Running high-level language programs

- **Compilation**
  - Program is translated into assembly language or directly into machine language
  - Compiled programs can be made to run relatively fast
  - Fortran, C, C++, ...

- **Interpretation**
  - Program is read by another program and executed one source language element at a time
  - Slower than compiled programs
  - Interpreters are (usually) easier to implement than compilers, are more flexible, and can provide excellent debugging and diagnostic support
  - Java, Python, Perl,...
Designing a Compiler

- Compilers are well-studied, but very complex programs.
- Complexity is managed by dividing the compilers work into independent stages or phases.
- Typically a phase analyzes a representation of a program, and translates that representation into another that is more appropriate for the next phase.
- Design of these intermediate program representations are critical to the successful implementation of a compiler.
The Compilation Process
(Phases)

Scanner (lexical analysis)
Read program and convert character stream into tokens.

Parser (syntax analysis)
Read stream of tokens and generate a parse tree.

Semantic analysis
Traverse parse tree, checking non-syntactic rules.

Intermediate code generation
Traverse parse tree again, emitting intermediate code.

Optimization
Examine intermediate code, attempting to improve it.

Target code generation
Translate intermediate code to assembly / machine code.

Machine-level optimization
Examine machine code, attempting to improve it.
Lexical analysis

- Program file is just a sequence of characters
- Wrong level of detail for syntax analysis
- Lexical analysis groups sequence of characters into **tokens**
- Tokens are the smallest “units of meaning” in the compilation process and are the foundation for parsing (syntax analysis)
- The compiler component that performs lexical analysis is called the **scanner** and is often automatically generated from a high-level specification
  - More on scanners in the next lecture...
Lexical analysis example

A GCD program in C

```c
int gcd (int i, int j) {
    while (i != j) {
        if (i > j)
            i = i - j;
        else
            j = j - i;
    }
    printf("%d\n",i);
}
```

Tokens

```c
int   gcd   (  
int   i    ,  
int   j    )  
{    while  (  
i   !=   j  
)  {    if  
    (  i  >   j  
    )  i  
    =  i  -  
    j ;    else  
    j = j  
-  i ;  
    }  printf  (  
"%d\n",  I  
)  ;  }
```
Syntactic analysis

- Lexical analysis generates a stream of tokens
- Wrong level of detail for semantic analysis and code generation
- Syntax analysis groups a string of tokens into parse trees guided by the context-free grammar that specifies the syntax of the language being compiled
  - conditional -> if ( expr ) block else block
- Parse trees represent the phrase structure of the program and are the foundation for semantic analysis and code generation
- The compiler component that performs syntax analysis is called the parser and is also often generated from a high-level specification
  - More on context-free grammars and parsing in upcoming lectures...
Syntax analysis example

Tokens

if ( i > j ) i = i - j;
else j = j - i ;}

conditional

if ( expr ) block else block

statement

id comp id

id = expr

statement

id = expr

id op id

i id op id

i j - i

i j - i

j i
Semantic analysis

- Determines the meaning of a program based on the parse tree representation.
- Enforces rules that are not governed by the program language syntax:
  - Consistent use of types, e.g.,
    - int a; char s[10]; s = s + a; illegal!
  - Every identifier is declared before used.
  - Subroutine calls provide the correct number and type of arguments.
  - Etc.
- Updates the symbol table, notating among other things, the types of variables, their sizes, and the scope in which they were declared.
Intermediate code generation

- Parse trees are the wrong level of detail for optimization and target code generation.
- Intermediate code generation transforms the parse tree into a sequence of intermediate language statements which embody the semantics of the source program.
- The intermediate language is just as powerful, but simpler than the high-level source language.
  - E.g., the intermediate language may only have one type of looping construct (goto) whereas the source language may have several (for, while, do, etc.).
- A simple intermediate language makes subsequent phases of the compiler easier to implement.
Target code generation

- Ultimate goal of the compilation process is to generate a program which the computer can run
  - This is the job of the target code generator

- Step 1: traverse the symbol table, assigning variables to locations in memory

- Step 2: traverse the parse tree or intermediate language program, emitting loads and stores for variable references, along with arithmetic operations, comparisons, branches, and subroutine calls
Intermediate code and/or target code is typically not as efficient as it could be.
- Limitation allows code generators to focus on code generation, not on code improvement.
- An optimizer can be called to improve the quality of intermediate and/or target code after each of these phases.
- The compiler component which improves the quality of generated code is called the optimizer.
- Optimizers are the most complicated part of the compiler.
- Optimization algorithms are often very elaborate, require substantial amounts of memory and time to run, and produce only small improvement in program size and/or run-time performance.

Two important optimizations:
- Register allocation - deciding which program variables can reside in registers at a particular point in the program’s execution.
- Dead code elimination - removing functions, blocks, etc., which won’t be executed by the program.
So why study programming languages and compilers?

According to Michael Scott:
- Understand obscure language features
- Choose among alternative ways to express things
- Make good use of debuggers, assemblers, linkers, and related tools
- Simulate useful features in languages that lack them

According to me:
- Compilers are large, complex programs: studying them helps you better understand “big software”
- Lots of programs contain “little programming languages”
  - Unix shells, Microsoft Office applications, etc.
  - It’s useful to know a bit about language design and implementation so that you can incorporate little languages into your own software
Roadmap of remaining lectures

- Next two lectures (Chapter 2 from text)
  - Lexical analysis
  - Syntactic analysis
  - Automata theory and automatic generation of scanners and parsers

- Final five lectures
  - Names, scopes and binding (Chapter 3)
  - Control flow (Chapter 6)
  - Subroutines and control abstraction (Chapter 8)
  - Building a runnable program (Chapter 9)
  - Object oriented programming (Chapter 10)
Programming language and compiler resources

- Website for our text
  - http://www.cs.rochester.edu/u/scott/pragmatics/

- Catalog of compiler construction tools
  - http://www.first.gmd.de/cogent/catalog/

- Conferences and journals
  - ACM Transactions on Programming Languages and Systems
  - ACM SIGPLAN Conference on Programming Language Design and Implementation
  - ACM SIGPLAN Conference on Programming Language Principles