

# CS421 Lecture 1: Course Introduction<sup>1</sup>

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

May 30, 2006

---

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter. Figures from Concepts of Programming Languages, Seventh Edition, by Robert W. Sebesta

Outline  
Objectives for Today  
Administrivia  
Grading and Homeworks  
Motivation and History  
Language Design  
Language Implementation

# Welcome to CS 421!

## Outline

Objectives for Today  
Administrivia  
Grading and Homeworks  
Motivation and History  
Language Design  
Language Implementation

Objectives for Today

Administrivia

Grading and Homeworks

Motivation and History

Language Design

Language Implementation

## Today's Objectives

Your goal for this lecture is to . . .

- ▶ learn about who is teaching the class;
- ▶ learn some of the course mechanics;
- ▶ find out why this is important material to understand;
- ▶ learn some about the history of programming languages;
- ▶ learn how to classify and evaluate languages;
- ▶ gain a high-level understanding of how languages are executed.

## Contact Info – Mark Hills (Instructor)

- ▶ Office: 2111A SC
- ▶ Office Hours: Thursdays 2:00 PM - 3:30 PM (or by appointment)
- ▶ Email: [mhills@cs.uiuc.edu](mailto:mhills@cs.uiuc.edu)

## Contact Info – Ben Hildore (TA)

- ▶ Office: TBD
- ▶ Office Hours (On-Campus): TBA
- ▶ Office Hours (I2CS): Tuesday 6:30 PM - 7:30 PM, Thursday 1:00 PM - 2:00 PM
- ▶ Email: [bhillo2@uiuc.edu](mailto:bhilldo2@uiuc.edu)

## Course References

### No required textbook

- ▶ Compilers: Principles, Techniques, and Tools ("The Dragon Book"), by Aho, Sethi, and Ullman
- ▶ Programming Language Pragmatics (2nd Edition), by Michael L. Scott
- ▶ Concepts of Programming Languages (7th Edition), by Robert W. Sebesta
- ▶ Essentials of Programming Languages (2nd Edition), by Friedman, Wand, and Haynes
- ▶ Other books, papers, etc will be mentioned as we go...

## Electronic Resources

**Course Web Page** <http://www.cs.uiuc.edu/class/cs421/>.

You are expected to check the announce page daily.

**News group** `class.cs421`

You should read the news group daily also.

**Languages** OCaml, a dialect of ML.

See <http://caml.inria.fr/ocaml>

**Computer Lab** EWS labs: <http://www.ews.uiuc.edu/>

*NOTE: Contact the TAs if you do not have an EWS account.*



## Course Website

- ▶ *Main* has important news, links elsewhere
- ▶ *Policy* has rules governing the course – read this!
- ▶ *Lectures* has links to the lecture slides, plus old lectures
- ▶ *MPs* has info on the homeworks
- ▶ *Exams* has info on exams, past exams, and useful info to help prepare
- ▶ *Unit Projects* has info on projects for students registered for 4 hours
- ▶ *Resources* has links to helpful information
- ▶ *FAQ* has answers to some common course questions

## CS421 Best Practices

- ▶ Check the course web page and course news group daily.
- ▶ Attend lectures regularly and participate actively.
- ▶ Come to office hours and ask questions!
- ▶ Do the programming assignments thoroughly: they are critical
- ▶ Start an MP the day after it is out, not day before it is due!!
- ▶ **Take responsibility and initiative in learning material** — work out practice questions; do some OCaml programming; read another text (e.g., *EOPL* or *Scott*)

## Course Grading

- ▶ MPs (Machine Problems): 35% of grade, submitted electronically
- ▶ Midterm: 25%, in class June 29th
- ▶ Final: 40%, August 5th

## Course Homework

- ▶ You may discuss homeworks and their solutions with others
- ▶ You may not leave the discussion with a written solution
- ▶ You must write your own solution
- ▶ You may **not** look at another written solution when writing your own – but you *may* look at provided examples from class and other similar examples

## Unit Project

- ▶ For students registered for 4 hours
- ▶ Can work in groups of up to 3 people
- ▶ Potential projects detailed on course web page
- ▶ Fairly flexible – you should pick something you are interested in, to make it more fun
- ▶ Project proposals are due by **June 23**, but can be in earlier if you want to get a head start

## Why Study Programming Languages?

- ▶ Understand efficiency tradeoffs of different language features
- ▶ Reduce bugs by understanding language semantics better
- ▶ Make better choices of languages for specific tasks
- ▶ Use existing languages in new ways
- ▶ Learn new languages more quickly
- ▶ It's fun!

## Study of Programming Languages

- ▶ Design and Organization
  - ▶ Syntax: How a program is written
  - ▶ Semantics: What a program means
  - ▶ Implementation: How a program runs
- ▶ Major Language Features
  - ▶ Imperative/Applicative/Rule-based
  - ▶ Sequential/Concurrent

## Early History

- ▶ 1940's: programming meant modifying the *hardware*; programmers had intimate contact with the computer
- ▶ early 1950's: Early low-level languages developed
  - ▶ machine code
  - ▶ assembly language
- ▶ late 1950's - early 1960's: First higher-level languages arrive
  - ▶ Fortran
  - ▶ LISP
  - ▶ COBOL
  - ▶ ALGOL



## Structured Programming, Module Systems, and the $\lambda$ Calculus

- ▶ late 1960's - 1970's: Structured programming
  - ▶ Pascal
  - ▶ C
- ▶ late 1970's - early 1980's: Data abstraction/module systems
  - ▶ Modula-2
  - ▶ CLU
  - ▶ Ada
- ▶ 1960's:  $\lambda$  calculus really starts to influence languages
  - ▶ ISWIM (Landin)
  - ▶ ML (late 1970's)

## Object Orientation, Dynamic Languages, and the Internet

- ▶ 1980's: Object oriented languages start to catch on
  - ▶ Smalltalk
  - ▶ C++
  - ▶ Extensions to existing languages: Objective C, Object Pascal, etc
- ▶ 1990's: The Internet starts to impact languages more
  - ▶ Java
- ▶ 1990's - 2000's: Dynamic languages become more popular
  - ▶ Web scripting languages (JavaScript, etc)
  - ▶ Perl, Python, PHP
  - ▶ Some new interest in Lisp

## Imperative Languages

- ▶ Main focus: machine state the set of values stored in memory locations
- ▶ Command-driven: Each statement uses current state to compute a new state
- ▶ Syntax: S1; S2; S3; ...
- ▶ Example languages: C, Pascal, FORTRAN, COBOL, CLU, Ada

## Imperative Languages, C

```
1 while (b > 0) {  
2     a = a * 2;  
3     b = b - 1;  
4 }
```

## Imperative Languages, CLU<sup>2</sup>

```
1 Poly = cluster is create, degree, coeff, ....
2   rep = record[coeffs: array[int], lo, hi: int]
3   create = proc(c,n : int) returns (Poly)
4     A : array[int] := array[int]$create(0)
5     A[n] := c
6     return(up(rep${coeffs: A, lo: n, hi: n}))
7   end create
```

---

<sup>2</sup>from Sam Kamin's *Programming Languages: An Interpreter Based Approach*

## Object-Oriented Languages

Classes are complex data types grouped with operations (methods) for creating, examining, and modifying elements (objects); subclasses include (inherit) the objects and methods from superclasses

- ▶ Computation is based on objects sending messages (methods applied to arguments) to other objects
- ▶ Syntax: Varies, object  $\leftarrow$  method(args)
- ▶ Example languages: Java, C++, Smalltalk, Simula-67, Beta

## Object-Oriented Languages, C++

```
1 class Square {  
2 public:  
3     int x,y,h,l;  
4     Square() { x = y = h = l = 0; }  
5 };
```

## Object-Oriented Languages, Beta

```
1 (#  
2 do  
3   'Hello world!' -> putline  
4 #)
```



## Applicative/Functional Languages

Programs as functions that take arguments and return values;  
arguments and returned values may be functions

- ▶ Programming consists of building the function that computes the answer; function application and composition main method of computation
- ▶ Syntax:  $P1(P2(P3 X))$
- ▶ Example languages: ML, LISP, Scheme, Haskell, Miranda, Clean, Erlang

## Applicative/Functional Languages, OCaml

```
1 # let twice f x = f(f(x));;  
2 # let inc x = x + 1;;  
3 # inc 5;;  
4 6  
5 # twice inc 5;;  
6 7
```

## Logic/Rule-based Languages

- ▶ Programs as sets of basic rules for decomposing problem
- ▶ Computation by deduction: search, unification and backtracking main components
- ▶ Syntax: Answer : – specification rule
- ▶ Example languages: Prolog, Datalog, BNF Parsing

```
1 - human(socrates).  
2 - mortal(X) :- human(X).  
3 -? mortal(Who).  
4 Who = socrates
```

## Themes in Language Development

- ▶ Languages have become more *abstract* – abstraction of data, memory, machine errors, execution, etc
- ▶ Languages have shifted performance burden to compilers – machines used to be expensive, now people are

## Evaluation Criteria<sup>3</sup>

- ▶ Readability
- ▶ Writability
- ▶ Reliability
- ▶ Cost

---

<sup>3</sup>From Concepts of Programming Languages, 7th Edition, by Robert W. Sebesta

## Readability

- ▶ Overall Simplicity
- ▶ Orthogonality
- ▶ Control Statements (Structured Control vs. Gotos)
- ▶ Flexible Data Types and Structures
- ▶ Flexible, Consistent Syntax (identifiers, keywords, consistency)

## Writability

- ▶ Simplicity and Orthogonality
- ▶ Support for *Process* and *Data* Abstractions
- ▶ Expressivity

## Reliability

- ▶ Type Checking
- ▶ Type Safety
- ▶ Exception Handling
- ▶ Aliasing
- ▶ Readability and Writability



## Cost

- ▶ Training
- ▶ Writing programs
- ▶ Programming environment availability
- ▶ Compilation
- ▶ Execution performance
- ▶ Language implementation/execution environment expense
- ▶ Reliability
- ▶ Maintenance
- ▶ Others: Portability? Generality?

## Goals can conflict...

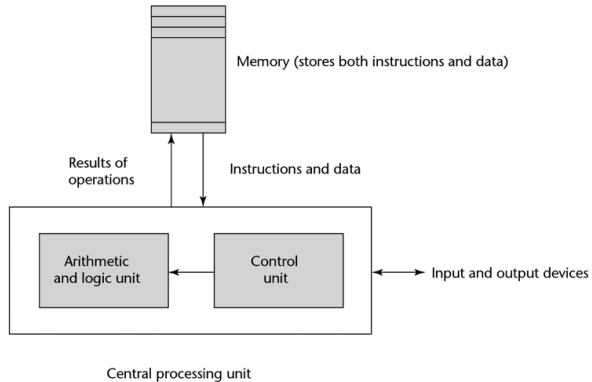
- ▶ Orthogonality is important, but too much can be confusing
- ▶ Sometimes more readable languages are not as writable, or vice versa (nested comments, overloaded operators, etc)
- ▶ Reliability and cost can be in tension – more reliable programs may be more expensive to execute

## Programming Language Implementation

- ▶ Develop layers of machines, each more primitive than the previous
- ▶ Translate between successive layers (compile or interpret)
- ▶ Ultimately, all programs execute in hardware

**Figure 1.1**

The von Neumann  
computer architecture

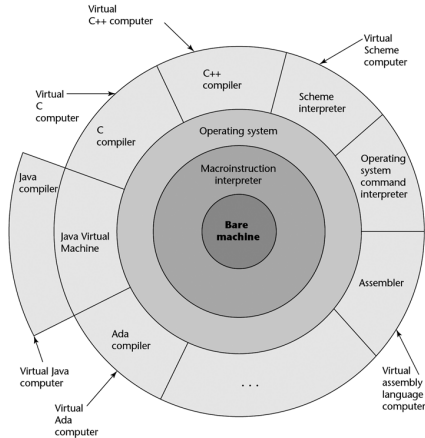


## Virtual Machines

- ▶ At first, programs written in assembly language (or at very first, machine language)
- ▶ Hand-coded to be very efficient
- ▶ Now, no longer write in native assembly language
- ▶ Use layers of software (eg operating system)
- ▶ Each layer makes a *virtual machine* in which the next layer is defined
- ▶ Compilers often define virtual machine layers: lambda calculus, continuations, graph reduction, etc. in functional languages, bytecode machines, etc

**Figure 1.2**

Layered interface of virtual computers, provided by a typical computer system

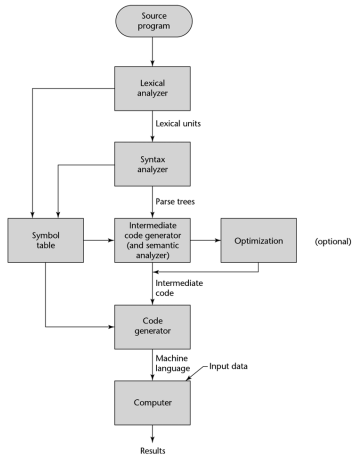


## What is a compiler?

A **compiler** from language  $L_1$  to language  $L_2$  is a program that takes an  $L_1$  program and for each piece of code in  $L_1$  generates a piece of code in  $L_2$  with the same meaning

Figure 1.3

The compilation process





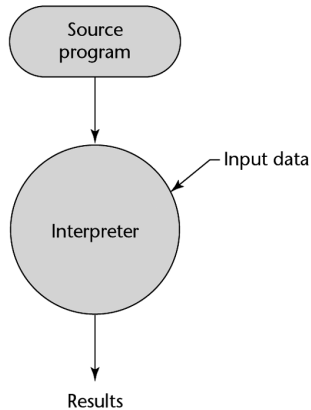
## What is an interpreter?

An **interpreter** of  $L_1$  in  $L_2$  is an  $L_2$  program that executes the meaning of a given  $L_1$  program

Interpreters and compilers must know about both the *syntax* of the language and its *semantics*, and must preserve the semantics (the meaning)

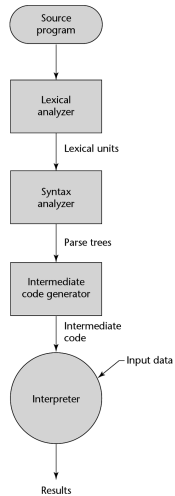
**Figure 1.4**

Pure interpretation



**Figure 1.5**

Hybrid implementation system



## Compiler Front-End

- ▶ Lex: Break the source into separate tokens
- ▶ Parse: Analyze phrase structure and apply semantic actions, usually to build an abstract syntax tree
- ▶ Semantic analysis: Determine what each phrase means, connect variable name to definition (typically with symbol tables), check types

## Compiler Back-End

- ▶ Translate to IR
- ▶ Optimize: Improve generated code, while maintaining original meaning
- ▶ Instruction Selection
- ▶ Register Allocation
- ▶ Emit final code

## Sample IR

Source:

```
1 X = A + B + C;
```

Three Address Code:

```
1 %tmp.1 = A + B;  
2 X = %tmp.1 + C;
```

## Sample Optimization

Source:

```
1 while (X != 0) {  
2     B = 5;  
3     C = X + B;  
4     X = ...  
5 }
```

Target:

```
1 if (X != 0) B = 5;  
2 while (X != 0) {  
3     C = X + B;  
4     X = ...  
5 }
```

## For Further Information

- ▶ For History, most books on languages. Sebesta Chapter 2 is good, as well as Chapter 2 of Louden's Programming Languages: Principles and Practice. The books for the History of Programming Languages workshops are good in-depth sources as well. Finally, IEEE Annals of the History of Computing has articles on this theme occasionally.
- ▶ Language Design principles can be found in the same sources. Obviously, there is disagreement about which features are best, so it's best to look at multiple sources.
- ▶ For compilers, Compilers: Principles, Techniques, and Tools by Aho, Sethi, and Ullman is the classic text. Cooper and Torczon's Engineering a Compiler is a more recent treatment.