

CSC324H, Fall 2009

Principles of Programming Languages (Week 1)

Instructor: Yilan Gu
yilan@cdf.toronto.edu

<http://www.cs.toronto.edu/~yilan/324f09/>

Today

- **Administrative Information**
- **Introduction of Programming Languages**
- **Programming Languages Paradigms**
- **Formal Specifications of Programming Languages**

Many thanks to ...

- Lecture notes are prepared partially based on materials provided by previous CSC324 instructors (Diane Horton, Suzanne Stevenson, Eric Joanis, Sheila McIlraith, Wael Aboelsaadat, Eric Hsu and Anya Tafliovich).
- Thanks particularly to Wael Aboelsaadat, Gary Baumgartner, Eric Hsu, Ali Muhammad and Anya Tafliovich
- Slides copyright © Wael Aboelsaadat 2004, modified by Yilan Gu 2009/09

Course Information

- Class web site:
 - <http://www.cs.toronto.edu/~yilan/324f09/>
 - Course information sheet for rules, grading, important dates,...
 - Visit useful websites for additional info on PLs we study
- One theoretical assignment and three programming assignments
 - A1 (paper copy) will be submitted right before to tutorial
 - A2-A5 assignments should be submitted online
 - Silent policy and grace day policy
- Evaluation:
 - Five assignments (6%, 8%, 8%, 8%, 8%)
 - Two term test (10% each), held in tutorial hours
 - Final exam 42% (need to get 33.33% to pass the course)

Course Contents

- **Introduction and History of Programming Languages (PLs)**
- **Formal Specification of PLs**
- **Functional Programming (Scheme)**
- **Logic Programming (Prolog)**
- **Procedure Design**

Introduction and Levels of Programming Languages

Course Goals

- **Programming language culture:**
 - Learn what is important about various languages to ensure an appropriate language is selected for a given task/domain
 - Understand the ideas and programming methods better
 - Understand the languages you use by comparison with other languages
 - Appreciate history and diversity of ideas in programming
 - Be prepared for new problem-solving paradigms
- **Critical thought:**
 - Properties of language, not documentation
- **Language and implementation:**
 - Recognize the cost of presenting an abstract view of machine
 - Understand trade-offs in programming language design
 - Be prepared for CSC488-Compilers and Interpreters

Introduction: what is a PL?

"a language intended for use by a **person** to express a **process** by which a **computer** can solve a problem"

-- Hope and Jipping

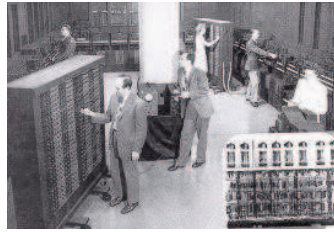
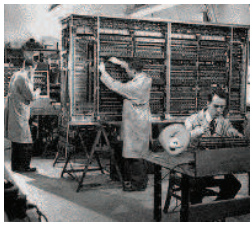
"a set of conventions for communicating an algorithm"

-- E. Horowitz

"the art of programming is the art of organizing complexity"

-- E. Dijkstra, 1972

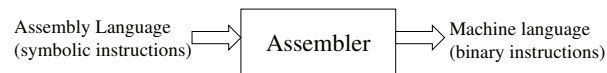
PLS: programming then...



Line	Instruction	Op Code	Operand
0	11001	000	
1	11111	010	
2	11111	110	
3	11111	010	
4	11111	001	
5	11111	011	
6	11111	010	
7	11111	110	
8	00111	010	
9	00111	001	
10	00111	110	
11	11111	010	
12	11111	011	
13	11111	110	
14	00111	010	
15	00111	110	
16	01011	000	
17	11111	110	
18	11111	110	
19	00111	001	
20	00111	110	
21	00111	110	
22	11011	000	
23	11011	000	
24	11011	000	
25	11011	000	
26	11011	000	
27	11011	000	

PLS: assembly language

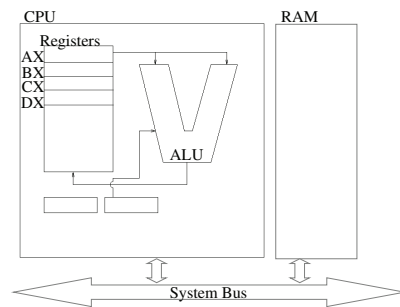
- Assembly language consist of a set of instructions that are in one-to-one corresponds with machine language



- **Examples:**

- Adding 3 numbers (-3,-4 & 10) and multiply result by 6

```
MOV AX, -3
MOV BX, -4
ADD AX, BX
MOV AX, DX
MOV BX, 10
ADD AX, BX
MOV AX, DX
MUL AX, 6
```



- **What's the problem?**

- Very detailed, tedious, error-prone and machine-specific

PLS: machine language

- Operations are simple for things like
 - Move constants of mem location 08125 to register 7
 - Jump to line 85
 - Skip the next instruction ...
- Instructions are encoded as numbers
- No variables
- Programming requires deep understanding of the machine architecture
- Programs are not portable because the instructions and their encoding are machine-specific
- Programs are extremely hard too write, debug and read

PLS: assembly language (cont.)

- Operations and operands have symbolic names
- Can use macros as shorthand for common sequences of code
- An assembler translates into machine code
- Still machine dependent
- Almost as hard to write as machine code

PL History: what is a PL?

“The main idea is to treat a program as a piece of literature, addressed to human beings rather than to a computer.”

Donald Knuth

<http://www-cs-faculty.stanford.edu/~knuth/lp.html>

Want to write a code in a high-level language (that can be understand easily by human beings.

What kinds of PLs Do You Know?



PLs: a language diagram

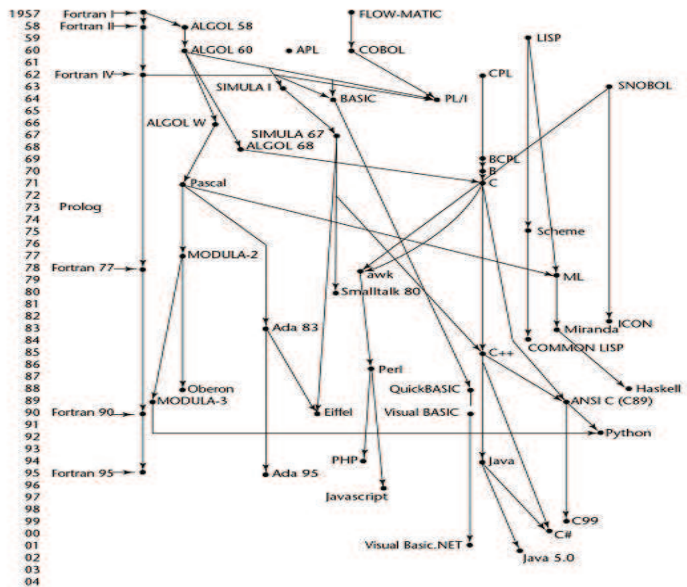


Figure 2.1 Genealogy of common high-level programming languages from Sebesta, Concepts of programming languages

PLs: high-level languages

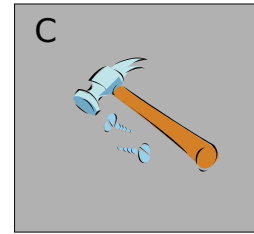
- Examples: C, Lisp, Java, Fortran, ...
- Have higher level constructs. Example:


```
if (x == 3)
    <some instrns>
else
    <other instrns>
```
- Language usually supports type checking and other checks that help detect bugs.
- Programs are much easier to write, debug, and read.
- Programs are now machine independent.
- Programs may still be “language-implementation dependent”.
- Before the first Fortran compiler (1957), it was commonly believed that any compiler would produce code so terribly inefficient to be useless.

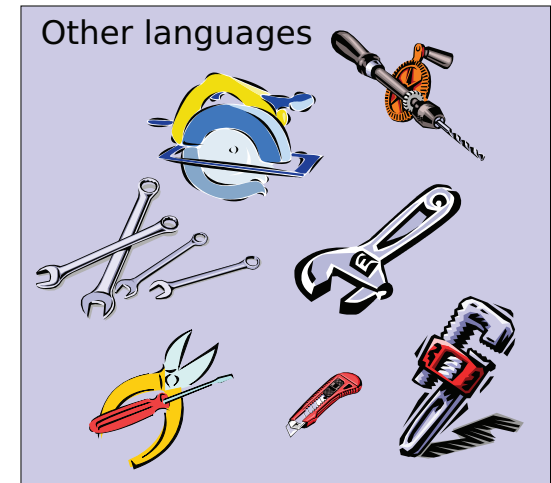
PLS: why are there so many PLS?

- We've learned better ways of doing things over time
- **Socio-economic factors: proprietary interests, commercial advantage**
- **Orientation toward special purposes**
- **Orientation toward special hardware**
- **Different ideas about what is pleasant to use**

PLS: high-level languages



• *Carpentry view:
If all you have is a
hammer, then
everything looks like a
nail!*



Digression: "A hammer is more than just a hammer. It's a personal tool that you get used to and you form a loyalty with. It becomes an extension of yourself."

<http://www.hammernet.com/romance.htm>

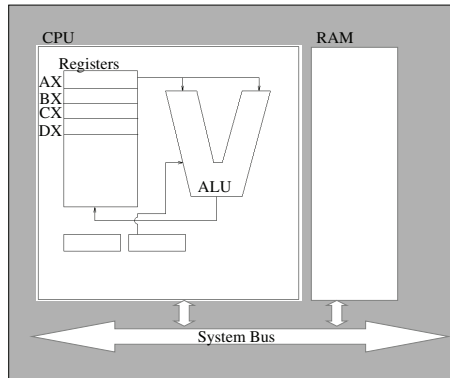
PLS: successful/popular languages - why?

- **Easy to learn**
 - BASIC, Pascal, Python
- **Easy to express things; Easy use once fluent; 'Powerful'**
 - C, Perl, Python, Scheme
- **Easy to implement**
 - Basic
- **Possible to compile to very good (fast/small) code**
 - Fortran
- **Backing of a powerful sponsor**
 - Ada, visual basic
- **Wide dissemination at minimal cost**
 - Pascal, java

Programming Language Paradigms

PL Paradigms: imperative

- **Underlying notion of an abstract machine**
 - Von Neumann architecture
 - Store (memory)
 - Accumulator (ALU)
 - Load/store into memory
 - Key operation: assignment



PL Paradigms: functional

- **Process of problem solution expressed as a sequence of operations on the data**
 - (Pure) value binding through parameter passing
 - No store accessible through names
 - No iteration
 - Key operation: function application (with recursion)

PL Paradigms: imperative examples

Fortran

```
SUM = 0
DO 11 K=1,N
SUM = SUM + 2 * K
11 CONTINUE
```

C

```
sum = 0;
for (k=1; k <= n; ++k)
    sum += 2*k;
```

Pascal

```
sum := 0;
for k:= 1 to n do
    sum := sum + 2 * k;
```

PL Paradigms: functional language example

Scheme

```
(define (sumble n)
  (if (= n 0)
      0
      (+ (* n 2) (sumble (- n 1)))))
)

(sum 4) evaluates to 20 = 2+4+6+8
```

PL Paradigms: logic

- **Program is a formal description of characteristics required of a problem solution**
 - Programs tell what should be not how to make it so
 - Solutions through a reasoning process called theorem proving
 - Key operations: unification

PL Paradigms: logic language example

```
sum(0,0).  
sum(N,S) :- NN is N - 1,  
           sum(NN, SS),  
           S is N * 2 + SS.
```

Prolog

```
?- sum(1,2).  
yes  
?- sum(2,4).  
no  
?-sum(20,S).  
S = 420  
?-sum(X,Y).  
X = 0 = Y
```

PL Paradigms: evolution

Problem → Algorithm → Assembly Code → Machine Code
|-----Assembler-----|
|---Imperative/Functional Languages---|
|-----Logic Languages-----|

Many paradigmatic conventions cut through these distinctions:

- Message-passing
- Object-orientation
- Event-handling
- concurrency/threading

What Makes a Good Language

- Easy to read and understand
 - Comments, names, syntax,
- Easy to learn
- Reflect intuition of the programmer
- Orthogonality
 - Small number of concepts combine regularly and systematically, without exceptions.
- Portability (*language standardization*)
- Meaning of a construct doesn't depend on context
- Simple syntax
- Naturalness for the intended applications

Good design demands good compromise

Language Specification

Language Specification: syntax vs. semantics

- **Syntax**
 - The structural rules of a language that determine the *form* of a program written in the language
 - Examples:
 - In C, variable names can be followed by two adjacent + symbols (Index++)
 - In Java, the main method must be defined as `public static void main(...)`
 - In C++/C, the if statement is written as `if(<expression>) <block> else <block>`
- **Semantics**
 - The *meaning* of the various language constructs in the context of a given program
 - Examples:
 - In C 'j = Index++;' *means* "increment Index after assigning its value to j"
 - In Java, defining a main method in a class *means* you can start the program by invoking that class from the command line.
 - In C++/C, the if statement *means* a selection construct that allows programmer to express one of two possible execution paths depending on some condition.

Fortran

```
SUM = 0
DO 11 K=1,N
SUM = SUM + 2 * K
CONTINUE
11
```

C

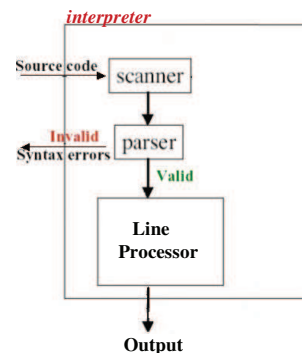
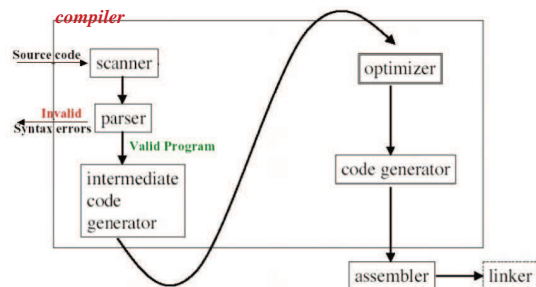
```
sum = 0;
for (k=1; k <= n; ++k)
    sum += 2*k;
```

Pascal

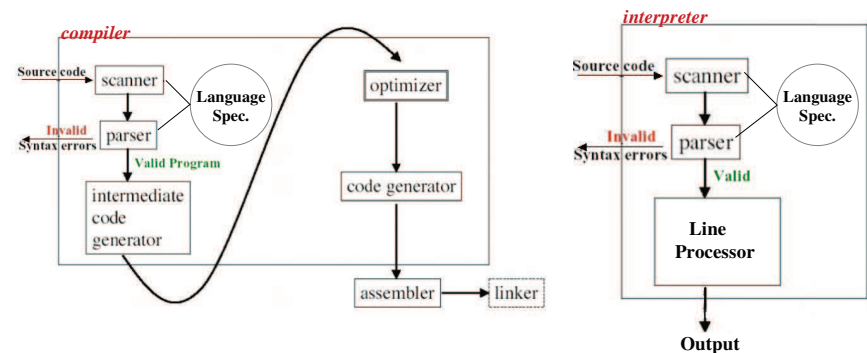
```
sum := 0;
for k:= 1 to n do
    sum := sum + 2 * k;
```

Language Specification: compilation vs. interpretation

- **Compilation**
 - Translation of a program written in a high-level PL into a form that is executable on the machine (*done by compiler*)
- **Interpretation**
 - A program is translated and executed one statement at a time (*done by interpreter*).



Language Specification: where is it used?



Language Specification: definitions

- **Language**
 - The collection of all valid strings (e.g., in human language, noun phrases) drawn from a finite alphabet (e.g., of characters)
- **Grammar**
 - Rules by which valid strings are formed
- **Recognizer**
 - Automaton (machine) able to recognize all valid strings (and reject invalid ones!)
- Languages can be specified by either a *grammar* or a *recognizer*

Language Specification : example

- Consider the ‘language’ of noun phrases
 - It was a sunny day.
 - We had a picnic in a lovely secluded park.
- A *grammar* for simple noun phrases:
 - $noun\text{-}phrase \rightarrow adjective\text{-}list\ noun$
 - $adjective\text{-}list \rightarrow adjective\ adjective^*$
- * Indicate zero or more times

Language Specification : example derivation

- It was a sunny day.
 - $noun\text{-}phrase \Rightarrow adjective\text{-}list\ day$
 - $adjective\text{-}list \Rightarrow sunny$
- Two *productions* were used to yield
 - $noun\text{-}phrase \Rightarrow sunny\ day$
- A *derivation* is a sequence of *productions* that begin with the *start* symbol (*noun-phrase* in this case) and derives a valid string in the language called the *yield*

$noun\text{-}phrase \rightarrow adjective\text{-}list\ noun$ $adjective\text{-}list \rightarrow adjective\ adjective^*$
--

Language Specification : another derivation

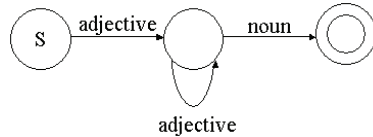
- We had a picnic in a lovely secluded park.
 - $noun\text{-}phrase \Rightarrow adjective\text{-}list\ park$
 - $adjective\text{-}list \Rightarrow lovely\ secluded$
- Each of the two above applications of *productions* is called a *sentential form*
- Here, two *productions* were used to yield
 - $noun\text{-}phrase \Rightarrow lovely\ secluded\ park$
- The *yield* is lovely secluded park

$noun\text{-}phrase \rightarrow adjective\text{-}list\ noun$ $adjective\text{-}list \rightarrow adjective\ adjective^*$
--

Language Specification : using a recognizer

- Here is a recognizer, or automaton, that will recognize the same language:

noun-phrase \rightarrow adjective-list noun
 adjective-list \rightarrow adjective adjective*



- This finite state machine recognizes our language of simple English noun phrases

Language Specification : example 2

- A grammar for expressions

expression \rightarrow identifier [1]
 \rightarrow number [2]
 \rightarrow - expression [3]
 \rightarrow (expression) [4]
 \rightarrow expression operator expression [5]
 operator \rightarrow + [6]
 \rightarrow - [7]
 \rightarrow * [8]
 \rightarrow / [9]

- Let's look at the formula for a line: $m*x + b$

expression \Rightarrow expression operator expression (using 5)
 \Rightarrow expression operator identifier (using 1)
 \Rightarrow expression + identifier (using 6)
 \Rightarrow expression operator expression + identifier (using 5)
 \Rightarrow expression operator identifier + identifier (using 1)
 \Rightarrow expression * identifier + identifier (using 8)
 \Rightarrow identifier * identifier + identifier (using 1)
 m * x + b

Language Specification : example 2

- A grammar for expressions

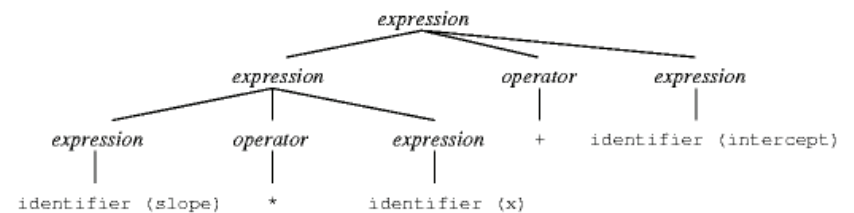
expression \rightarrow identifier
 \rightarrow number
 \rightarrow - expression
 \rightarrow (expression)
 \rightarrow expression operator expression
 operator \rightarrow +
 \rightarrow -
 \rightarrow *
 \rightarrow /

For example:
 x+y
 Index/5
 (x+(m*t))
 -10

Language Specification: a parse tree for $m*x + b$

- $m*x + b$

expression \Rightarrow expression operator expression (using 5)
 \Rightarrow expression operator identifier (using 1)
 \Rightarrow expression + identifier (using 6)
 \Rightarrow expression operator expression + identifier (using 5)
 \Rightarrow expression operator identifier + identifier (using 1)
 \Rightarrow expression * identifier + identifier (using 8)
 \Rightarrow identifier * identifier + identifier (using 1)
 m * x + b



Language Specification : example 2 – alternate derivation

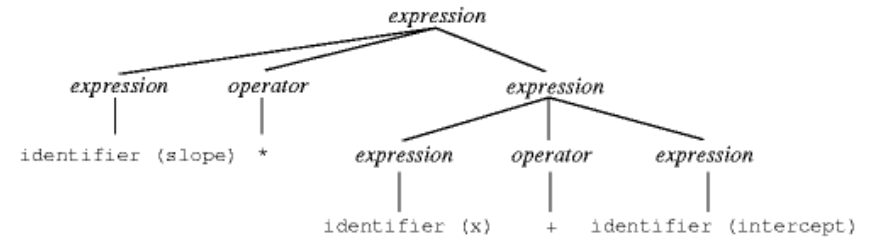
- A grammar for expressions

<i>expression</i>	→ <i>identifier</i>	[1]
	→ <i>number</i>	[2]
	→ <i>- expression</i>	[3]
	→ <i>(expression)</i>	[4]
	→ <i>expression operator expression</i>	[5]
<i>operator</i>	→ <i>+</i>	[6]
	→ <i>-</i>	[7]
	→ <i>*</i>	[8]
	→ <i>/</i>	[9]

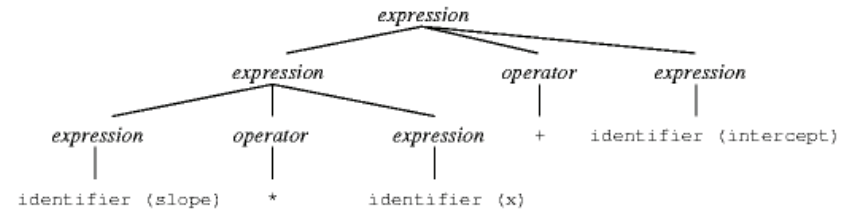
- $m * x + b$

$expression \Rightarrow expression \operatorname{operator} expression$ (using 5)
 $\Rightarrow expression \operatorname{operator} expression$ (using 1)
 $\Rightarrow identifier * expression$ (using 8)
 $\Rightarrow identifier * expression \operatorname{operator} expression$ (using 5)
 $\Rightarrow identifier * expression \operatorname{operator} identifier$ (using 1)
 $\Rightarrow identifier * identifier + expression$ (using 6)
 $\Rightarrow identifier * identifier + identifier$ (using 1)
 $m * x + b$

Language Specification: an alternate parse tree for $m * x + b$



Instead of



Language Specification: which parse is right?

- Hint: remember, we are dealing with syntax here, not semantics ...

Language Specification: ambiguity in grammars

- A grammar that allows multiple parses of a single input string is termed *ambiguous*.

Language Specification: how can our grammar for expressions be 'fixed'?

- A modified grammar for expressions

```
expression  → term
            → expression add-op term
term        → factor
            → term mult-op factor
factor      → identifier | number | - factor | ( expression )
add-op      → + | -
mult-op     → * | /
```

- | means or

- **To do:** find out if this is an ambiguous grammar

```
field_declaration ::= ( [ doc_comment ] ( method_declaration / constructor_declaration
variable_declaration ) ) / static_initializer / ";" .
```

```
method_declaration ::= < modifier > type identifier "(" [ parameter_list ] ")" < "[" "]" >
( statement_block / ";" ) .
```

blah blah blah.....

```
variable_declaration ::= < modifier > type variable_declarator < "," variable_declarator > ";" .
```

blah blah blah....

```
modifier ::= "public" | "private" | "protected" | "static" | "final" | "native" | "synchronized" |
"abstract" | "threadsafe" | "transient" .
```

```
package_name ::= identifier | ( package_name "." identifier ) .
```

blah blah blah...

```
identifier ::= "a..z,$_<" < "a..z,$_0..9,unicode character over 00C0" > .
```

Interested to read more:

- <http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/AJAVA.html>
- <http://www.cs.uiowa.edu/~fleck/JavaBNF.htm>

A more difficult example in *modified* BNF format: Java Grammar Rules

```
goal          ::= compilation_unit

compilation_unit ::= [ package_statement ] < import_statement > < type_declaration > .

package_statement ::= "package" package_name ";" .

import_statement ::= "import" ( ( package_name "." "*" ";" ) / ( class_name / interface_name )
) ";" .

type_declaration ::= [ doc_comment ] ( class_declaration / interface_declaration ) ";" .

doc_comment ::= "/*" "... text ..." "*/" .

class_declaration ::= < modifier > "class" identifier [ "extends" class_name ] [ "implements"
interface_name < "," interface_name > ] "{" < field_declaration > }" .

interface_declaration ::= < modifier > "interface" identifier [ "extends" interface_name < ","
interface_name > ] "{" < field_declaration > }" .
```

Readings...

- **Lecture 1:** Mitchell, Chapter 1, 2, 4.4
- **Lecture 2:** Mitchell, Chapter 4.1