Principles of Programming Languages I

Wael Aboelsaadat
wael@cs.toronto.edu
http://www.cs.toronto.edu/~wael/324.html

Today

- Administrivia
- History of Programming Languages
- Programming Languages Paradigms
- Formal Specifications of Programming Languages

Administrivia

- Class web site:
  - Course information sheet for rules, grading, important dates,…
  - Visit useful websites for additional info on PLs we study
- One theoretical assignment and four programming assignments
- Midterm on Oct 25th (15%) and final worth 45%
- Readings…

Course Contents

- Introduction & History of Programming Languages (PLs)
- Formal Specification of PL
- Functional Programming (Scheme, ML)
- Types
- Logic Programming (Prolog)
- Procedure Design
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
  - Understand the language you use (C, C++, Java) by comparison with other languages
  - Appreciate history!, 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 && Interpreters

Introduction & PL History

**PL History:** Von Neumann architecture

- How to specify a program?
Assembly language consists of a set of instructions that are in one-to-one correspondence with machine language.

Examples:
- Adding 3 numbers (-3, -4, & 10)
- 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

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/pl.html

"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

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.hammerart.com/women/htm
**PL History: 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

---

**PL History: successful/popular languages - why?**

- **Easy to learn**
  - BASIC, Pascal, LOGO
- **Easy to express things; Easy use once fluent; ‘Powerful’**
  - C, Perl
- **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

---

**PL 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: imperative examples**

**Fortran**

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

**C**

```
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: imperative vs. assembly**

```
int main() {
    int nIndex, nSum;
    for(nIndex=0; nIndex<10; nIndex++)
        nSum =+ 2 * nIndex;
}
```

```
.int main() {
    int nIndex, nSum;
    for(nIndex=0; nIndex<10; nIndex++)
        nSum =+ 2 * nIndex;
    }
```

```
.globl main
.type main, @function
main: push BP
    mov $9, AX
    mov SP, BP
    sub $8, SP
    and $-16, SP
    p2align 4,,15
.L6:
    dec AX
    jns .L6
    mov BP, SP
    pop BP
    ret
.size main, .-main
.ident "GCC: (GNU) 3.3.1"
```

```
Try this: gcc -O2 -S -c foo.c
```

**PL Paradigms: object oriented**

- Organizes a program to be operations on abstract representations of the data
  - Objects with data abstraction and information hiding
  - Object implementation is hidden from user
  - Actions performed on objects (messages)
  - Can combine with imperative or functional paradigm easily
  - Key operation: message passing
**PL Paradigms: object oriented example**

```java
class intSet : public Set
{
    public: intSet()
    {
        //inherits Set add_element(), Set del_element()
        //from Set class, defined as a set of objects
        public int sum()
        {
            int s = 0;
            SetEnumeration e = new SetEnumeration(this);
            while (e.hasMoreElements())
            {
                s += ((Integer)e.nextElement().intValue());
            }
            return s;
        }
    }
}
```

**PL Paradigms: functional language example**

```scheme
(define (square n)
    (if (< n 0)
        0
        (+ (* n 2) (square (- n 1))))
)
```

<s>```scheme
(define sum 4) evaluates to 20 = 4+6+8</s>

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

Prolog

?- sum(1,2).
  yes
?- sum(2,4).
  no
?- sum(26,5).
  S = 420
?- sum(X,Y).
  X = 0 = Y

PL Paradigms: visual languages

PL Paradigms: evolution

Problem → Algorithm → Assembly Code → Machine Code

------------------------ Assembler ------------------------
------------------- Imperative/Functional Languages -------
------------------- Logic Languages -----------------------
PL Design Criteria

• Readability
  – Comments, names, … syntax
  – Bad Perl: $string = "bballball";
    $string =~ s" (b)\1(a..)\1\2" $1$2";

• Writability

• Orthogonality
  – Small number of concepts combine regularly and systematically, without exceptions.

• Portability (language standardization)

• Abstraction

• Simplicity (e.g. assignment is always from right to left)
  – Good example: ‘pointers’ in Java

Good design demands good compromise

PL Design: how are new languages designed?

• Lots and lots of design choices and tradeoffs

• Need to define a variety of parameters
  – Syntax and Semantics
  – Naming and scope
  – Data types
  – Expressions, assignment and control flow
  – Subroutines
  – Data abstraction and object orientation
  – Exception handling and concurrency

PL Design: what a language designer needs to know?

• Applications

• Architecture

• Art of Programming

The AAAs of Language Design
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.

### Examples of code:

**Fortran**

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

**Pascal**

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

**C**

```c
j = Index++;
```

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: 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 derivation**

- It was a **sunny day**:
  
  `noun-phrase => adjective-list day
  adjective-list => sunny`

- Two **productions** were used to yield
  
  `noun-phrase => 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**

**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-phrase => adjective-list noun
  adjective-list => adjective adjective*`
  
  * Indicate zero or more times

**Language Specification : another derivation**

- We had a picnic in a **lovely secluded** park.
  
  `noun-phrase => adjective-list park
  adjective-list => lovely secluded`

- Each of the two above applications of productions is called a **sentential form**

- Here, two productions were used to yield
  
  `noun-phrase => lovely secluded park`

- The **yield** is lovely secluded park
Language Specification: using a recognizer

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

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

Language Specification: example 2

- A grammar for expressions:

  - expression → identifier
  - number
  - expression
  - (expression)
  - expression operator expression

- operator → +
  → -
  → *
  → /

- Let's look at the formula for a line: \( m \times x + b \)

  - expression ⇒ expression operator expression (using 5)
  - expression operator identifier (using 1)
  - expression operator identifier (using 6)
  - expression operator expression + identifier (using 5)
  - expression operator expression + identifier (using 1)
  - expression operator expression + identifier (using 8)
  - identifier + identifier (using 1)

  \[ m \times x + b \]
Language Specification: Language Specification:

for \( m \cdot x + b \)

Instead of

• A grammar for expressions

expression \( \rightarrow \) identifier \[1\]
\( \rightarrow \) number \[2\]
\( \rightarrow \) expression \[3\]
\( \rightarrow \) expression \[4\]
\( \rightarrow \) expression \cdot operator \cdot expression \[5\]
operator \( \rightarrow \) \[6\]
\( \rightarrow \) \[7\]
\( \rightarrow \) \[8\]
\( \rightarrow \) \[9\]

• \( m \cdot x + b \)

expression \( \Rightarrow \) expression \cdot operator \cdot expression \hspace{1cm} \text{(using 5)}
\( \Rightarrow \) expression \cdot operator \cdot expression \hspace{1cm} \text{(using 1)}
\( \Rightarrow \) identifier \cdot expression \hspace{1cm} \text{(using 8)}
\( \Rightarrow \) identifier \cdot expression \cdot operator \cdot expression \hspace{1cm} \text{(using 5)}
\( \Rightarrow \) identifier \cdot identifier \cdot expression \hspace{1cm} \text{(using 1)}
\( \Rightarrow \) identifier \cdot identifier \cdot identifier \hspace{1cm} \text{(using 1)}

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

• | means or

• To do: find out if this is an ambiguous grammar

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 = "/**" ... "*/"

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

interface_declaration = < modifier > "interface" identifier < "extends" interface_name < "," interface_name > " implements " interface_name < "," interface_name > " {" < field_declaration > " }"

field_declaration = [ doc_comment ] < method_declaration / constructor_declaration / variable_declaration > / static_initializer " ;"

method_declaration = < modifier > type_identifier "(" parameter_list ")" < "[ [ " [ statement_block ] " ] ]"

variable_declaration = < modifier > type_variable_declarator < ";" variable_declarator < ";" variable_declaration < ";"

modifier = "public" / "private" / "protected" / "static" / "final" / "transient" / "synchronized" / "abstract" / "threadsafe" / "transient"

package_name = identifier | ( package_name "." identifier )

identifier = "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

Readings...

• Lecture 1: 1,3.1,3.2,4.4
• Lecture 2: 4.1