NCCU CS 碩專班
Advanced Programming Languages
高等程式語言

Instructor:
資科系
陳 恭 副教授
Spring 2006

Lecture 1
Contacts

• Instructor: Dr. Kung Chen
  – E-mail: chenk@cs.nccu.edu.tw
  – Office: 大仁樓200210
  – Office Hours: Tuesday: 11am-12am

• Teaching assistants: 資科系研究生
  – E-mail: g9403@cs.nccu.edu.tw
  – g9405@cs.nccu.edu.tw
  – Office: 資科系程式語言與軟體方法實驗室

• Class web page:
  – http://www.cs.nccu.edu/~chenk/Courses/APL
What shall we study?
Don’t Get Confused by the Course Name

A “principles” course aims to teach the underlying concepts behind all programming languages (Concept-driven).
<table>
<thead>
<tr>
<th>Programming languages you have used</th>
<th>Programming languages you've heard of</th>
</tr>
</thead>
<tbody>
<tr>
<td>• C, Java, C++, C#</td>
<td>• Lisp, …</td>
</tr>
<tr>
<td>• Basic, …</td>
<td></td>
</tr>
<tr>
<td>• …</td>
<td></td>
</tr>
</tbody>
</table>
• Why are there so many programming languages?

"I speak Spanish to God, Italian to women, French to men, and German to my horse."

— Emperor Charles V (1500-1558)
Why So Many Languages?

• Application domains have distinctive (and conflicting) needs

• Examples:
  - Scientific Computing: high performance
  - Business: report generation
  - Artificial intelligence: symbolic computation
  - Systems programming: low-level access
  - Real-time systems: timing constraints
  - Special purpose languages
  - ...
Why So Many Languages? (Cont’d)

- Evolution: Our understanding of *programming* evolves, so do languages grow!
- The advance of *programming methods* usually lead to new programming languages:
  - Structured programming
    - Pascal
  - Object-Oriented Programming
    - Simula 67, Smalltalk, C++
  - Functional programming
- Socio-economic factors
  - Proprietary interests, commercial advantage
- Personal preference
Different Programming Language Design Philosophies

If all you have is a hammer, then everything looks like a nail.
What is a Programming Language?

A language for programming?

什麼是一個程式語言？
其目的為何？

What is Programming?

Writing programs?

What is a Program?
What is Programming?

Using computers to solve problems!
What is a program?

• A program is computer coding that:
  • Takes input
  • Performs some calculation on the input
  • Displays output

Is a program a function?
What is a Function?

- \( y = 2 \times x \)
- \( \text{twice}(x) = 2 \times x \)

\( \text{twice}(x) = x + x \)

;; a formula to compute twice; not the only one
What is a Function?

A function is a mapping (set) from one set to another set. $f : A \rightarrow B$ is a subset of $\{ (a, b) | a \text{ in } A, b \text{ in } B \}$

- $\forall x \in A, \exists! y \in B$, such that $f(x) = y$
- No one-to-many mapping.
A program computes a function

\textbf{gcd}(i, j): greatest common divisor

program gcd (input, output);
var i, j : integer;
begin
  read (i, j);
  while i <> j do
    if i > j then i := i \(- j
    \) else j := j \(- i
    \);
  writeln (i)
end.
What does the Following Program Computes?

```
r := x ;
q := 0 ;
while r ≥ y do
begin
  r := r - y ;
  q := q + 1
end ;
```

Integer division?

\[
x / y = q, \text{ remainder } = r
\]
Partial and Total Functions

- Total function: \( f(x) \) has a value for every \( x \)
- Partial function: \( g(x) \) does not have a value for every \( x \)

\[\neg (\forall a \in A, \exists! b \in B, \text{such that } f(a) = b)\]

Source: Prof. John Mitchell of Stanford University
Value of an expression may be **undefined**
- Undefined operation, e.g., division by zero
  - $3/0$ has no value
  - implementation may halt with error condition
- **Nontermination**
  - $f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$
  - this is a *partial* function: not defined on all arguments
  - cannot be detected at compile-time; this is halting problem
- These two cases are
  - “Mathematically” equivalent
  - Operationally different

Source: Prof. John Mitchell of Stanford University
Partial and Total Functions

• Total function $f : A \rightarrow B$ is a subset $f \subseteq A \times B$ with
  - For every $x \in A$, there is some $y \in B$ with $\langle x, y \rangle \in f$ (total)
  - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ (single-valued)

• Partial function $f : A \rightarrow B$ is a subset $f \subseteq A \times B$ with
  - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ (single-valued)

• Programs define partial functions for two reasons
  - partial operations (like division)
  - nontermination
    
    \[
    f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2) \]

Source: Prof. John Mitchell of Stanford University
Why Study Programming Languages?

• Increased capacity to express programming concepts
  – A language is a “conceptual universe” (Perlis)
  – 語言影響我們的思考! Good language aids programmer in writing good programs.

• 工欲善其事必先利其器!
  – Improved background for choosing appropriate languages
  – Greater ability to learn new languages

• Understand significance & cost of implementation

• Ability to design new languages
But I will *never* design a programming language!

- Many system programs are like languages
  - command shells
  - programmable editors
  - XML schema
- Many system programs are like compilers
  - read & analyze configuration files and command line options
- Easier to use and design such things once you know about ‘real’ languages
- Martin Fowler, Language-Oriented Programming

你們一定會遇到一些需要 PL knowledge 的場合
A Little Language: 規則描述語言範例

TABLE 門診醫令清單:
BEGIN
資料格式:  $$=12;
醫事服務機構代號:  $$ MEMBER{HospitalID.HospID};
費用年月:  today()-date($$)<=day(732);
申報類別:  $$ IN {1,2};
案件分類:  $$=門診點數清單.案件分類;
流水號:  $$=門診點數清單.流水號;
藥品項目代號:
  IF $$.醫令類別 IN {0,2,7,8} THEN $$ MEMBER{NHIFeepay.Code}
  ELSE IF ($$.醫令類別=1) THEN $$ MEMBER{DrugsCode.Code}
  ELSE IF ($$.醫令類別=3) THEN $$ MEMBER{SpecialStuff.Code}
  ELSE IF ($$.醫令類別=4) THEN $$ MEMBER{NHIFeepay.Code,}
$$=SUM 申請金額 FROM 點數清單
WHERE 點數清單.案件分類 IN {02,03,C1,D1,D4,E1};

$$ = COUNT 點數清單 WHERE 點數清單.案件分類=05;
END
規則檢查程式產生器系統流程概述

1. 檢查程式產生器製作

2. 產生平衡檢查程式

3. 執行檢查程式進行檢核
Some Basic Concepts of Programming Languages

• Classification of PL’s
  - Levels of Abstraction
  - Programming Paradigms

• Aspects of PL’s
  - Syntax vs. Semantics
  - Pragmatics: Implementation
  - Preferred Programming methods
Levels of Abstractions

```c
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- High level: C / Java / Pascal
- Low level: Assembly / Bytecode
- Machine Language

```
muli $2, $5, 4
add  $2, $4, $2
lw   $15, 0($2)
...
```
Abstraction!
Abstraction Levels

Abstract 也有 摘要 的意思
History of Ideas: Abstraction

• Abstraction = detached from concrete details
• Abstraction necessary to build software systems

• Modes of abstraction
  - Via languages/compilers:
    • Higher-level code, few machine dependencies
  - Via subroutines
    • Abstract interface to behavior
  - Via modules
    • Export interfaces; hide implementation
  - Via abstract data types
    • Bundle data with its operations
Example: Quicksort

- 39 23 17 90 33 72 46 79 11 52 64 5 71

- Pick middle element as pivot: 46

Partition list

23 17 5 33 39 11 46

79 72 52 64 90 71

- quick sort the less than list
  ... (recursive call)...

  5 11 17 23 33 39

- quick sort the greater than list
  ... (recursive call)...

  52 64 71 72 79 90
Quicksort in C

```c
qsort( a, lo, hi ) int a[ ], hi, lo;
{   int h, l, p, t;
    if (lo < hi)
    {   l = lo;   h = hi;   p = a[hi];
        do
        {  while ((l < h) && (a[l] <= p))   l =  l  + 1;
            while ((h > l) && (a[h] >= p))   h = h – 1 ;
            if (l < h) { t = a[l];  a[l] = a[h];  a[h] = t; }   
        } while (l < h);
        t = a[l];   a[l] = a[hi];  a[hi] = t;
        qsort( a, lo, l-1 );   qsort( a, l+1, hi );
    }
}
```

For array of integers only.
Quicksort in Haskell

qsort :: [Int] -> [Int]
qsort [] = []
qsort (x:xs) = qsort lt ++ [x] ++ qsort greq
where
    lt = [y | y <- xs, y < x]
    greq = [y | y <- xs, y >= x]

Polymorphic
qsort :: Ord a => [a] -> [a]
Programming Paradigms

Are there any fundamental differences among Fortran, C, Pascal, COBOL, etc.?
Example: Sum up twice each number from 1 to N

\[ \sum_{k=1}^{n} 2k \]

**Fortran:**
```
SUM = 0
DO 10 K = 1, N
    SUM = SUM + 2*K
10  CONTINUE
```

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

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

**COBOL:**
```
MOVE 0 TO SUM
PERFORM Total VARYING K FROM 1 BY 1 UNTIL K=N
Total.  ADD 2*K TO SUM
```
Programming Paradigms

• Way of thinking about programming
• View of a program
  • 對什麼是程式、計算(computation)有不同的認知
    (model)導致不同典範的程式語言
    - Imperative (命令式)
    - Functional (函數式)
    - Logic (邏輯式)
    - Object-Oriented (物件導向式)
Paradigms affect...

- ...how you see the problem
- ...how you express the problem
- ...how you go about solving the problem
- ...the solutions you consider
- ...the solutions you can implement
The Previous Example in Functional Languages

Scheme (Lisp): Prefix, simple syntax

(set sum (total n))

(define (total k)
  (cond
    ( (= k 1) 2)
    (t (+ (* 2 k) (total (- k 1)))
  )
)

Haskell

sum k
  | k==0       = 0
  | otherwise = 2*k + sum (k-1)
Imperative Languages

• 最早也是最普遍的（von Neumann model）
• 視一程式為一系列指揮電腦的指令
  主要的概念是變數、指定序述（assignment）與控制
  流程—program modifies state
• Examples: Fortran, Cobol, Basic, C, Pascal, Ada, etc
An Example: Appending two linked lists

```c
struct node *nalloc(void)
{
    return((struct node *) malloc(sizeof(struct node)));  
}

struct node *copy(node *n)
{
    node *temp = nalloc();
    temp->data = n->data;
    return temp;
}

struct node *append(node *l1, node *l2)
{
    node *temp = NULL;
    if (l1) {
        temp = copy(l1);
        temp->next = append(l1->next,l2);
    }
    else if (l2) {
        temp = copy(l2);
        temp->next = append(l1,l2->next);
    }
    return temp;
}
```
Functional Languages

- Based on mathematical functions, emphasizing concise and correct program proofs.
- View a program as a series of mathematical functions, program execution proceeds through function call simplification.
- Examples: Lisp, SML, Haskell etc

```
(define append (l1 l2)
  (cond ((null l1) l2)
       (t (cons (car l1) (append (cdr l1) l2))
  )
)
```

```
fun append nil l = l
  | append (h::t) l = h::(append t l);
```

Pattern matching
Logic Languages

- 基於 First-order 邏輯推論
- 視一程式為一系列邏輯事實與推論規則
  程式的執行透過邏輯推論的化簡來進行
- Examples: Prolog

```prolog
append([], K, K).
append([X|K], L, [X|M]) :- append(K, L, M).
```
Object-Oriented Languages

- 擬人(物)化: 物件 = 資料 + 程式碼
- 視一程式為一組互動的物件的定義
  程式的執行透過物件間訊息傳遞來完成
- Examples: Smalltalk, Java, Simula 67, C++, etc

```cpp
template <class T>
void LList<T>::Append(const LList<T>& L)
{
    Node *p=L.First;
    while(p != 0)
    {
        Insert(p->Data);  p=p->Next;
    }
}
```
Declarative vs. Non-declarative

- Languages can also be classified as
- Imperative (Non-Declarative)
  - You tell the computer exactly how to solve a problem
- Declarative
  - You ask the computer to find the solution for you
- A spectrum between “what” and “how”: It’s a matter of “degree”
Scripting Languages

- Shell scripts?
- Popular scripting languages
  - Perl
  - JavaScript
  - Tcl/Tk
  - VBScript, Python, Ruby, …

“… Scripting languages represent a very different style of programming than system programming languages. They are designed for "gluing" applications, and use typeless approaches to achieve a higher level of programming and more rapid application development….” --Ousterhout
Domain-Specific Languages (DSL)

- A.K.A. Application-Oriented Languages

“A programming language tailored specifically for an application domain: it is not general purpose, but rather captures precisely the semantics of the domain, no more and no less.”
Popular DSL’s

- **SQL** (for Database table manipulation)
- **Lexx and Yacc** (for program lexing and parsing)
- **PERL** (for text/file manipulation)
- **VHDL** (for hardware description)
- **TeX and LaTex** (for document layout)
- **HTML/SGML** (for document “markup”)
- **Postscript** (for low-level graphics)
- **Open GL** (for high-level 3D graphics)
- **Tcl/Tk** (for GUI scripting)
- **Macromedia Director** (for animation)
- **Mathematica/Matlab** (for symbolic computation)
Aspects of Programming Languages

- Syntax
- Semantics
- Pragmatics
- Preferred Programming Methods
Syntax and Semantics of PL

• 語法 (Syntax) — 外表形式 (form)
  - (x=1) vs. (x==1) in C

• 語意 (Semantics) — 內在意義 (meaning), 執行結果

```
x = x + 1;
```

```
R-val
00000001
```

```
L-val
00000001
```

```
11000000
```

```
00000010
```

• C:   \( x = x + 1; \)  \( x++; \)
Pascal: \( x := x + 1; \)
Syntax: Famous FORTRAN example

```
Do 10 I = 1.5       // should be 1,5
     a(I) = X + B(I)
10 continue
```

- FORTRAN ignores all spaces
- so this was interpreted to
- do10i = 1.5
- destroyed a rocket!
Semantics of PL’s are complicated to get right

• 語意 (Semantics)— 不容易精準而無誤的表示清楚

What does the following print if “x==1“?

C: printf(“%d %d\n”, x++, ++x);
Motivating Semantics

- When can we safely replace one expression with another expression?

```c
int k, x;
k = f(x) + f(x);
Can we replace it with
k = 2 * f(x);
??
```

Example:

```c
int myOR (int x, int y)
{
    return x || y;
}

Is
    myOR(a, b)
Equivalent to
    a || b
??
```
Semantics

• More complex and involved than syntax

• Approaches
  - Natural language documents: Language Reference manual and Language Specification
    The Java Language Specification
  - Interpreter/compiler
  - Formal notations

  1. Axiomatic Semantics
  2. Denotational Semantics
  3. Operational Semantics
Pragmatics

• Use of a language
• Implementation
  – Compilation
  – Interpretation
  – Optimization
Implementation – Compilation

**Translation Time (Compile time)**

vs.

**Run Time**

\[
B \times 2 - 4.0 \times A \times C
\]

\[
R1 := B
\]

\[
R1 := R1 \times R1
\]

\[
R2 := 4.0
\]

\[
R2 := R2 \times R3
\]

\[
R3 := A
\]

\[
R2 := R2 \times R3
\]

\[
R3 := C
\]

\[
R2 := R2 \times R3
\]

\[
R1 := R1 \times R2
\]
Implementation – Interpretation

Interpretation Time :

\[
\text{function } \text{eval}(E) = \\
\quad \text{if } E \text{ is the constant } c \text{ then } c \\
\quad \text{elseif } E \text{ is the sum of } E_1 \text{ and } E_2 \\
\quad \quad \text{then } \text{eval}(E_1) + \text{eval}(E_2) \\
\quad \text{elseif } E \text{ is the product of } E_1 \text{ and } E_2 \\
\quad \quad \text{then } \text{eval}(E_1) \times \text{eval}(E_2) \\
\quad \ldots
\]
Implementation

• To compile or interpret? Pros and cons for each.

• Compilation can be more efficient than interpretation.

• Interpretation can be more flexible than compilation.

• 混合式 (Hybrid): Intermediate code, Java Virtual Machine
Hybrid Approach

例: The Java Virtual Machine
Anatomy of a Computer

Program (character stream, string)
Lexical Analyzer (Scanner)
Token Stream
Lexical Analyzer (Scanner) (斷字)

Program: a stream of characters (a big string)

```
2 3 4 * ( 1 1 + - 2 2 )
```

Token stream

Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op
Lexical Analyzer (Scanner)

Num(234) mul_op lpar_op Num(11) add_op Num(-22) rpar_op

Lexical errors:
18..23 + val#ue
Lexical Analyzer (Scanner)

Lexical errors:

18..23 + val#ue

- Not a number
- Variable names cannot have ‘#’ character
Anatomy of a Computer

Program (character stream)

Lexical Analyzer (Scanner)

Token Stream

Syntax Analyzer (Parser)

Parse Tree
Syntax Analyzer (Parser)

```
num \* ( num \+ num )
```
```c
int * foo(i, j, k)
{
    int i;
    int j;
    for(i=0; i j) {
        fi(i>j)
            return j;
    }
}
```

Syntax Analyzer (Parser)

Syntax errors (parsing errors)

- Extra parentheses
- Missing increment
- Not an expression
- Not a keyword
int * foo(i, j, k)
    int i;
    int j;
    {
        int x;
        x = x + j + N;
        return j;
    }
Anatomy of a Computer

Program (character stream)
- Lexical Analyzer (Scanner)
- Token Stream
- Syntax Analyzer (Parser)
- Parse Tree
- Semantic Analyzer
- Intermediate Representation
- Code Optimizer
- Optimized Intermediate Representation
int sumcalc(int a, int b, int N)
{
    int i;
    int x, y;
    x = 0;
    y = 0;
    for(i = 0; i <= N; i++) {
        x = x + 4*a/b*i + (i+1)*(i+1);
        x = x + b*y;
    }
    return x;
}

int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}
Anatomy of a Computer

Program (character stream)
- Lexical Analyzer (Scanner)
  - Token Stream
  - Syntax Analyzer (Parser)
    - Parse Tree
    - Intermediate Representation
      - Code Optimizer
        - Optimized Intermediate Representation
          - Code Generator
            - Assembly code
int sumcalc(int a, int b, int N) 
{
    int i;
    int x, t, u, v;
    x = 0;
    u = ((a<<2)/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + v + t*t;
        v = v + u;
    }
    return x;
}

test:
    subu $fp, 16
    add $t9, zero, zero
    sll $t0, $a0, 2
    div $t7, $t0, $a1
    add $t6, zero, zero
    add $t5, zero, zero
lab1:
    addui $t8, $t5, 1
    mul $t0, $t8, $t8
    addu $t1, $t0, $t6
    addu $t9, t9, $t1
    addu $6, $6, $7
    addui $t5, $t5, 1
    ble $t5, $a3, lab1
    addu $v0, $t9, zero
    addu $fp, $t9, zero
    b $ra
Stages of an interpreter

- Parsing
- Syntax Tree

Program → Analysis → Abstract program → Interpreter

Input → Output
Our Focus in this Course

• Imperative Languages
  – Concepts and Principles

• Use Language Development Tools
  – Parser generator
  – Source-to-source translation
    • C++ → C
    • Java subset → C

• OO Languages
  – Java (C++, C#)

• Functional Languages
  – Scheme, A little Haskell
Motivating Semantics

• What does a program mean?
• How do we describe the meaning of a program?
• As a function: $\text{Sem[aProgram]}$: Input $\rightarrow$ Output
• Or …

Must first determine the meaning of
• “x = 1;”
• “x++”
• “++x”
• printf(…)
and its evaluation order

```c
int main()
{
    x = 1;
    printf("%d, %d\n", x++, ++x);
    return 0;
}
```
There are two declarations of x.

When we use “x” in main(), which declaration are we using?

Name resolution and name scope

Example:

```c
int x;
void foo (int y)
{
    float x;
    ... x ...
}
int main() {
    ...
    foo(X);
    // which declaration of x?
    ...
}
```
Motivating Semantics

- When can we safely replace one expression with another expression?

```c
int myOR (int x, int y)
{
    return x || y;
}
```

Is `myOR(a, b)` equivalent to `a || b`?

Example:

```c
int k, x;
k = f(x) + f(x);
```

Can we replace it with

```c
k = 2 * f(x);
```

??

```c
int k, x;
k = f(x) + f(x);
```

Can we replace it with

```c
k = 2 * f(x);
```

??
Motivating Semantics

• What’s wrong with the function “dp”?

• What does “kp” points to after the execution of “do”?

• What’s wrong with the function “garbage”?

• Does it consume too many unneeded memory spaces?

Examples:

```c
int *dp(int i)
{
    int k, *kp;
    k = i;
    kp = &k;
    return kp;
}
```

```c
int garbage(int c) {
    toy_t *tp;
    int total = 0;
    tp = (toy_t *)
        malloc(c*sizeof(toy_t));
    tp->price = discount(Price);
    ...
    total = tp_price * c;
    return total;
}
```
Motivating Semantics

- A good command of semantics enables us to do more rigorous reasoning on the behaviors of our programs.

- What does the program on the right print?
What does a program mean?

• Compile and run
  – Implementation dependencies
  – Not useful for reasoning
  – 立法 (semantics) vs. 執法 (compiler)

• Informal Semantics
  – Natural language description of PL

• Formal Semantics
  – Description in terms of notation with formally understood meaning: sets, logic, algebra, …
Why not informal semantics?

Two types have compatible types if their types are the same [footnote: Two types need not be identical to be compatible.].

ANSI C Standard, 3.1.2.6

• What does “compatible” mean here?
• In practice, many people use informal semantics, yet try to use precise language and a little formal notations to help.
• We’ll study some formal semantics.
Formal Semantics: Operational Semantics

\[(x := a, s) \Rightarrow s[x \leftarrow A[a]s]\]

\[(\text{skip}, s) \Rightarrow s\]

\[
\frac{(S_1, s) \Rightarrow (S'_1, s')}{(S_1; S_2, s) \Rightarrow (S'_1; S_2, s')}\]

\[
\frac{(S_1, s) \Rightarrow s'}{(S_1; S_2, s) \Rightarrow (S_2, s')}\]

\[(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \Rightarrow (S_1, s)\]

\[\text{if } B[b]s = \text{tt}\]

\[(\text{if } b \text{ then } S_1 \text{ else } S_2, s) \Rightarrow (S_2, s)\]

\[\text{if } B[b]s = \text{ff}\]

\[(\text{while } b \text{ do } S, s) \Rightarrow\]

\[\text{(if } b \text{ then } (S; \text{while } b \text{ do } S) \text{ else skip, } s)\]
Why use a Functional Programming Language (FPL)?

• FP languages are *higher level* languages than Imperative and Object-Oriented ones such as C, C++ and Java

• Not constricted by the Von-Neuman bottleneck

• Imperative languages like C essentially reflect computer hardware and are based around:
  – CPU and Memory Store
  – Fetch-Execute Cycle
  – Similar paradigm to assembly except at a much higher level
Google is using FPL

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that
Microsoft is developing FPL!

F#

Combining the strong typing, scripting and productivity of ML with the efficiency, stability, libraries, cross-language working and tools of .NET.

About Textbooks

• **No single book**

• **Recommended Text:**
  - Ken Slonneger and Barry Kurtz, *Formal Syntax and Semantics of Programming Languages*, (Chap. 1)
  - Scheme & Haskell books

• **Articles, papers, websites**
  - You will be asked to read a few great papers in PL
  - Visit the course website
Assignments and Exams

• Homework assignments (40%)
  – Written exercises and small programs
  – More complex programs, in Scheme or Haskell and maybe others

• Midterm exam (20~25%)
  – Closed book, closed notes

• Final project (25~35%)
  – Format not determined yet
  – An essay or a group presentation & report
  – Plus a written quiz?

• Class participation (5~10%)
What determines a “good” language

The Roles of PL’s:

Human

↑

1. Making problem-solving convenient

↑

PL’s

↓

2. Making efficient use

↓

Computing Machines

Organizing Complexity
What determines a “good” language

• Formerly: Run-time performance
  – (Computers were more expensive than programmers)

• Now: Life cycle (human) cost is more important
  – Ease of designing, coding
  – Readability !!!
  – Supports Intention Revealing Programming
  – Debugging
  – Maintenance
  – Reusability
virtually all computers follow the *von Neumann* architecture

fetch-execute cycle: repeatedly
- fetch instructions/data from memory
- execute in CPU
- write results back to memory

imperative languages parallel this behavior
- variables (memory cells)
- assignments (changes to memory)
- sequential execution & iteration (fetch/execute cycle)

since features resemble the underlying implementation, tend to be efficient

declarative languages emphasize problem-solving approaches far-removed from the underlying hardware
- e.g., Prolog (logic): specify facts & rules, interpreter performs logical inference
  LISP/Scheme (functional): specify dynamic transformations to symbols & lists

tend to be more flexible and expressive, but not as efficient
A Word on Interpretation

• Data vs. Code (Program)
• It all depends on your interpretation
• The notion of “Stored program” by John Von Neumann