Introduction to Computer Science
Lecture 6: PROGRAMMING LANGUAGES

Tian-Li Yu
Taiwan Evolutionary Intelligence Laboratory (TEIL)
Department of Electrical Engineering
National Taiwan University

tianliyu@cc.ee.ntu.edu.tw

Slides made by Tian-Li Yu, Jie-Wei Wu, and Chu-Yu Hsu
PL Generations

1st
- Machine Instructions

2nd
- Assembly (Mnemonic system of MIs)

3rd
- Fortran
- Cobol
- Basic
- C/C++
- Java

4th
- SQL
- SAS
Assembler: Translating MIs to Assembly

<table>
<thead>
<tr>
<th>Machine instructions</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>156C</td>
<td>LD R5, Price</td>
</tr>
<tr>
<td>166D</td>
<td>LD R6, ShippingCharge</td>
</tr>
<tr>
<td>5056</td>
<td>ADDI R0, R5, R6</td>
</tr>
<tr>
<td>306E</td>
<td>ST R0, TotalCost</td>
</tr>
<tr>
<td>C000</td>
<td>HTL</td>
</tr>
</tbody>
</table>

- **Mnemonic names for op-codes**
- **Identifiers**: Descriptive names for memory locations, chosen by the programmer
3rd Generation Languages (3GL)

- Characteristics of assembly
  - Machine dependent
  - One-to-one mapping
  - Assembler

- High-level primitives
- Machines independent (virtually)
- One primitive to many MI mapping
- Compiler & interpreter
Languages and Issues

- Natural vs. formal languages
  - Formal language → formal grammar

- Portability
  - Theoretically: different compilers
  - Reality: Minor modifications
Programming Paradigms

- **Functional**: LISP, ML, Schema
- **Object-oriented**: Smalltalk, Visual Basic, Java, C++, C#
- **Imperative**: FORTRAN, BASIC, C, Ada, Pascal
- **Declarative**: COBOL, ALGOL, APL, GPSS, Prolog

Timeline:
- 1950
- 1960
- 1970
- 1980
- 1990
- 2000

Authors: Tian-Li Yu
Imperative vs. Declarative

- **Imperative paradigm**
  - Procedural
  - Approaching a problem by finding an algorithm to solve the problem.

- **Declarative paradigm**
  - Implemented a general problem solver
  - Approaching a problem by finding a formal description of the problem.
  - Will talk more about this later.
Functional Paradigm

Inputs:
- Old_balance
- Credits
- Debits

Output:
- New_balance

Find_sum

Find_sum

Find_diff
Functional vs. Imperative

(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))

Temp_balance ← Old_balance + Credit
Total_debits ← sum of all Debits
Balance ← Temp_balance – Total_debits

(Find_Quotiant (Find_sum Numbers) (Find_count Numbers))

Sum ← sum of all Numbers
Count ← # of Numbers
Quotiant ← Sum / Count
Object-Oriented Paradigm

- OOP (object-oriented programming)
- Abstraction
- Information hiding
  - Encapsulation
  - Polymorphism
- Inheritance

References:
More about Imperative Paradigm

- Variables and data types
- Data structure
- Constants and literals
- Assignment and operators
- Control
- Comments
Variables and Data Types

- Integer
- Real (floating-point)
- Character
- Boolean

FORTRAN

```
INTEGER  a, b
REAL    c, d
BYTE    e, f
LOGICAL g, h
```

Pascal

```
a, b: integer;
c, d: real;
e, f: char;
g, h: boolean;
```

C/C++ (Java)

```
int a, b;
float c, d;
char e, f;
bool g, h;
```
Imperative Paradigm

Data Structure

- Homogeneous array
- Heterogeneous array

FORTRAN

```
INTEGER a(6,3)
```

Pascal

```
a: array[0..5,0..2] of integer;
```

C/C++

```
int a[5][2];
```

C/C++

```
struct{
  char Name[25];
  int Age;
  float SkillRating;
} Employee;
```
Constant and Literals

- a ← b + 645;
  - 645 is a literal

- const int a=645;
- final int a=645;

A constant cannot be a l-value.
  - a=b+c;

a ← b + 645;
Assignment and Operators

- Operator precedence
- Operator overloading

APL
a ← b + c;

Ada, Pascal
a := b + c;

C/C++ (Java)
a = b + c;
Control

- Old-fashion: goto

```plaintext
goto 40
20 print "passed."
goto 70
40 if (grade < 60) goto 60
goto 20
60 print "failed."
70 stop
```

- Not recommended in modern programming
  - Modern programming

```plaintext
if (grade < 60)
    then print "failed."
else print "passed."
```
Control Structures

if (B) S1
else S2;

while (B)
S1;

switch (N)
{
case C1: S1; break;
case C2: S2; break;
case C3: S3; break;
};
for (int Count = 1; Count < 4; Count++)
    body;
Comments

- C/C++, Java

```c
a = b + c; // This is an end-of-line comment

/*
 * This is a block comment
 */
a = b + c;

/**
 * This is a documentation comment
 */
a = b + c;
```
Calling Procedures

Calling program unit requests procedure.

Calling program unit continues.

Control is transferred to procedure.

Procedure is executed.

Control is returned to calling environment when procedure is completed.
Terminology

Starting the head with the term “void” is the way that a C programmer specifies that the program unit is a procedure rather than a function. We will learn about functions shortly.

```c
void ProjectPopulation (float GrowthRate){
    int Year;
    Population[0] = 100.0;
    for (Year = 0; Year <= 10; Year++)
        Population[Year+1] = Population[Year] + (Population[Year]*GrowthRate);
}
```

The former parameter list. Note that C, as with many programming languages, requires that the data type of each parameter be specified.

These statements describe how the populations are to be computed and stored in the global array named Population.
Terminology (contd.)

- Procedure’s header
- Local vs. global variables
- Formal vs. actual parameters
- Passing parameters
  - Call by value (passed by value)
  - Call by reference (passed by reference)
  - Call by address: variant of call-by-reference.
Call by Value

**procedure** `Demo(Actual)`

```plaintext
 Demo(Actual);
```

**a.** When the procedure is called, a copy of data is given to the procedure

<table>
<thead>
<tr>
<th>Calling environment</th>
<th>Procedure's environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
</tr>
</tbody>
</table>

**b.** and the procedure manipulates its copy.

<table>
<thead>
<tr>
<th>Calling environment</th>
<th>Procedure's environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

**c.** Thus, when the procedure has terminated, the calling environment has not changed.

<table>
<thead>
<tr>
<th>Calling environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
</tbody>
</table>
**Call by Reference**

**procedure** Demo(Formal)
Formal ← Formal + 1;

Demo(Actual);

**C/C++**

```c
void Demo(int& Formal) {
    Formal = Formal + 1;
}
```

**a.** When the procedure is called, the formal parameter becomes a reference to the actual parameter.

**b.** Thus, changes directed by the procedure are made to the actual parameter.

**c.** and are, therefore, preserved after the procedure has terminated.
Functions vs. Procedures

- A program unit similar to a procedure unit except that a value is transferred back to the calling program unit as “the value of the function.”

```c
float CylinderVolume (float Radius, float Height){
    float Volume;
    Volume = 3.14 * Radius * Radius * Height;
    return Volume;
}
```

- The function header begins with the type of the data that will be returned.
- This declares a local variable named Volume.
- Compute the volume of the cylinder
- This declares a local variable named Volume.
- Terminate the function and return the value of the variable Volume.
The Translation Process

- Lexical analyzer: identifying tokens.
- Parser: identifying syntax & semantics.
Syntax Diagrams for Algebra

Expression

Term

Factor

Term

Factor

Expression

+  

-  

x  

y  

z
Grammar for Algebra

- Expression → Term | Term + Expression
  | Term - Expression
- Term → Factor | Factor * Term | Factor / Term
- Factor → x | y | z

Starting: Expression
Nonterminals: Expression, Term, Factor
Terminals: x, y, z
Parse Tree

\[ x + y \times z \]
Ambiguity

- if \(B_1\) then if \(B_2\) then \(S_1\) else \(S_2\)
Coercion: implicit conversion between data types

Strongly typed: no coercion, data types have to agree with each other.

Code optimization
- \( x = y + z; \)
- \( w = x + z; \)
- \( w = y + (z \ll 1); \)
Object-Oriented Paradigm

**OOP**

- **Object**
  - Active program unit containing both data and procedures

- **Class**
  - A template from which objects are constructed
  - An object is an instance of the class.

- **Instance variables & methods (member functions)**

- **Constructors**
  - Special method used to initialize a new object when it is first constructed.

- **Destructors vs. garbage collection**
An Example of Class

Instance variable
Constructor assigns a value to Remaining Power when an object is created.

```java
class LaserClass {
    int RemainingPower;

    LaserClass (InitialPower)
    { RemainingPower = InitialPower; }

    void turnRight ()
    { ... }

    void turnLeft ()
    { ... }

    void fire ()
    { ... }
}
```
Encapsulation

- A way of restricting access to the internal components of an object
- Bundling of data with the methods operating on that data.

Examples: private vs. public, getter & setter
**Polymorphism**

- Allows method calls to be interpreted by the object that receives the call.
- Allows different data types to be handled using a uniform interface.

```java
Circle circle;
Rectangle rect;
circle.draw();
rect.draw();
```
Inheritance

- Allows new classes to be defined in terms of previously defined classes.

```c
Class Base;
Class Circle : Base;
Class Rectangle : Base;

Base *base;
Circle circle;
Rectangle rect;
base = & circle;
base -> draw();
base = & rect;
base -> draw();
```
**Concurrency**

**Mutual Exclusion**: A method for ensuring that data can be accessed by only one process at a time.

**Monitor**: A data item augmented with the ability to control access to itself.

![Diagram showing the process of calling a procedure and the mutual exclusion](image-url)
Declarative Programming

Resolution
- Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
- \((P \text{ OR } Q) \text{ AND } (R \text{ OR } \neg Q)\) resolves to \((P \text{ OR } R)\)
- **Resolvent**: A new statement deduced by resolution
- **Clause form**: A statement whose elementary components are connected by OR

Unification
- Assigning a value to a variable so that two clauses would be the same.
- **Unify**(\(\text{Father}(\text{Mark}, \text{John}), \text{Father}(x, \text{John})\)) results in \(x\) is Mark.
Proof by Resolution (Refutation)

- We know that \((P \lor Q) \land (R \lor \neg Q) \land \neg R\) is true (\(KB\), knowledge base).
- We want to prove that \(P\) is true.
- Prove by showing that \(KB \land \neg p\) is unsatisfiable (empty clause).

\[
P \lor Q \\
P \lor R \\
\neg R \\
\neg P \\
\text{empty clause}
\]
Variables: first letter capitalized (exactly contrary to common logics).

Constants: first letter uncapitalized.

Facts:
- Consists of a single predicate
- predicateName(arguments).
  - parent(bill, mary).

Rules:
- conclusion :- premise.
  - :- means “if”
  - faster(X,Z) :- faster(X,Y), faster(Y,Z).

Operators:
“is”, ==, =, <, >, +, -, *, /, =>, <=
Gnu Prolog


- Interactive mode
  - Under the prompt `?-`, type `[user]`.
  - When finished, type `Ctrl-D`

- Comments
  - `/* */` or `%`

- Chinese incompatible.

- You may consult `*.pl` (a pure text file)
Prolog Examples

female(mary).
female(sue).
male(bill).
male(john).

parent(mary,john).
parent(bill,john).
parent(mary,sue).
parent(bill,sue).

mother(X,Y):−female(X),parent(X,Y).
father(X,Y):−male(X),parent(X,Y).

son(X,Y):−male(X),parent(Y,X).
daughter(X,Y):−female(X),parent(Y,X).

sibling(X,Y):−X\=Y,parent(Z,X),parent(Z,Y).
Prolog Examples

- Factorial again.
- If we want Prolog to compute factorials, we need to tell it what factorials are.

```prolog
factorial(0,1).
factorial(N,F) :-
    N > 0,
    N1 is N - 1,
    factorial(N1,F1),
    F is N * F1.

| ?- factorial(5,W).
  W=120 ?
```
Fibonacci Revisited

f(0,1).
f(1,1).

f(N,F) :-
    N>0,
    N1 is N-1,
    N2 is N-2,
    f(N1,F1),
    f(N2,F2),
    F is F1 + F2.

f(N,F) :- c(N,_,_,F).
c(0,0,0,1).
c(1,0,1,1).
c(2,1,1,2).
c(N,P1,P2,P3) :-
    N>2,
    N1 is N-1,
    c(N1, P0, P1, P2),
    P2 is P0+P1,
    P3 is P1+P2.

How about f(40,W)?
Ordered Clauses

factorial(0,1).

factorial(N,F) :-
    N > 0,
    factorial(N1,F1),
    N1 is N-1,
    F is N * F1.

?-factorial(3,W).

Try these commands:
- listing.
- trace.
- notrace.

This wouldn’t work, why?