Introduction to Computer Science
Lecture 6: Programming Languages 程式語言

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

Assembler: Translating MIs to Assembly

- 1st
  Machine instructions
- 2nd
  Assembly 組合語言

156C	⟶	LD R5, Price
166D	⟶	LD R6, ShippingCharge
5056	⟶	ADDI R0, R5, R6
306E	⟶	ST R0, TotalCost
C000	⟶	HTL

- 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

2nd

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

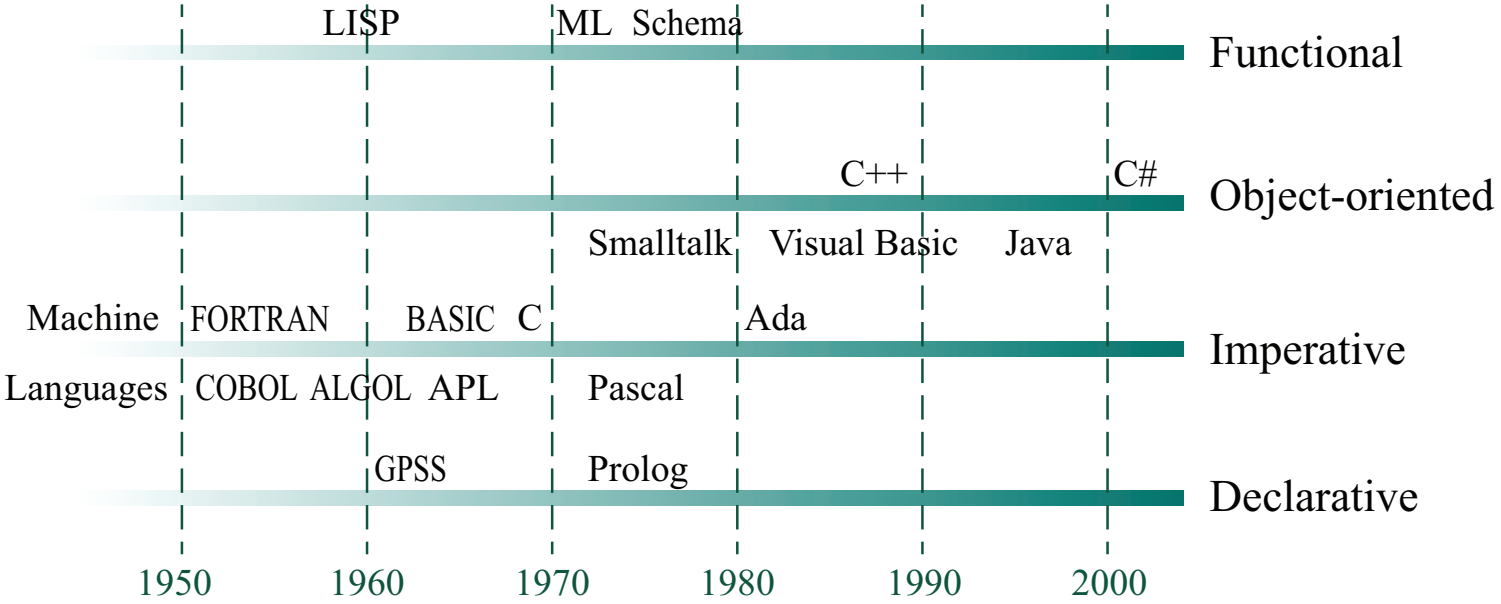
3rd

編譯　解譯　↳ 將程式碼轉換成機器看得懂的過程

Languages and Issues

- Natural vs. formal languages
  - Formal language → formal grammar
- Portability 可攜性 → 程式有多容易到不同平台執行
  - Theoretically: different compilers
  - Reality: Minor modifications

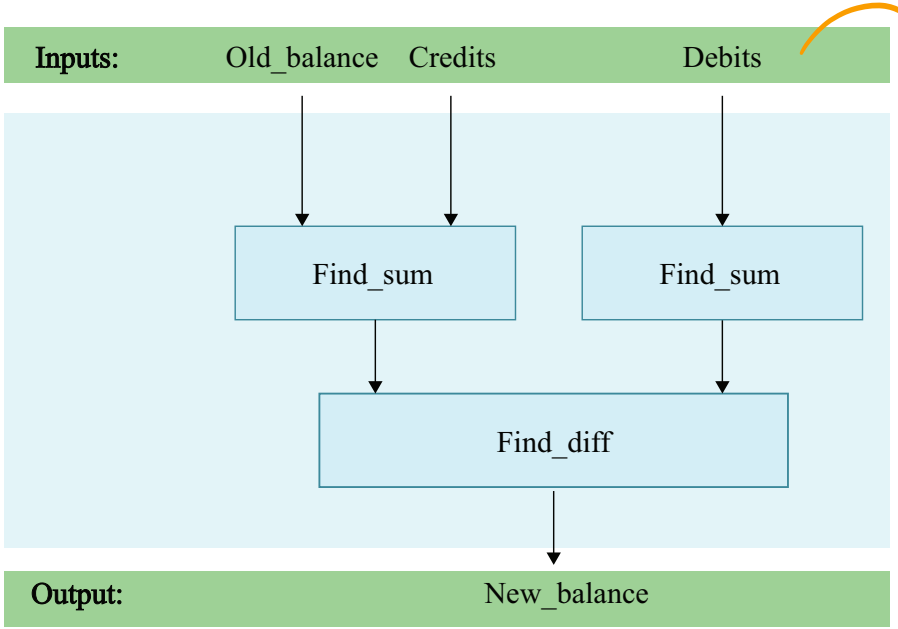
Programming Paradigms



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



會計方程式

Functional vs. Imperative

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

↳ (f x y) ≡ f(x, y)

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

Functional (LISP)

Imperative

Object-Oriented Paradigm

- OOP (object-oriented programming)
- Abstraction
- Information hiding ⇒ 幫助程式設計師避免犯錯
  - Encapsulation 封裝
  - Polymorphism 多型
- Inheritance 繼承

- References:
  - http://www.codeproject.com/KB/architecture/OOP_Concepts_and_manymore.aspx
  - http://en.wikipedia.org/wiki/Object-oriented_programming

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;
```

Data Structure

- Homogeneous array 同質陣列
- Heterogeneous array

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

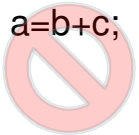
Pascal
```
a:  array[0..5,0..2] of integer;
```

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. ⇒不能在左邊
  - a=b+c;



b
Left

Assignment and Operators

APL
```
a <- b + c;
```

Ada, Pascal
```
a := b + c;
```

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

- Operator precedence
- Operator overloading

Control

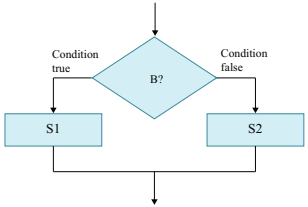
- Old-fashion: goto ⇒ 不好用

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

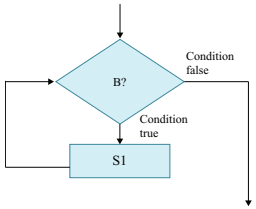
- Not recommended in modern programming
  - Modern programming

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

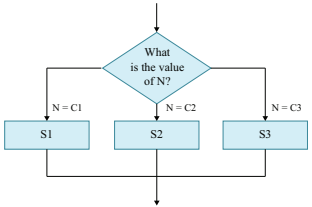
Control Structures




```
if (B) S1
   else S2;
```

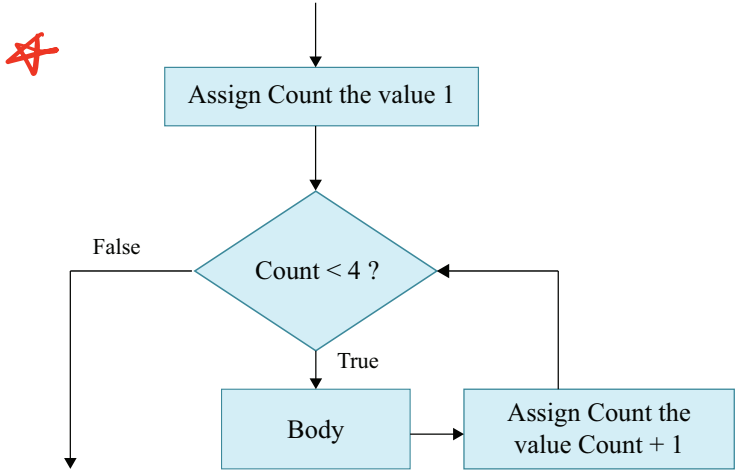



```
while (B)
  S1;
```




```
switch  (N)
{ case C1:  S1;break;
  case C2:  S2;break;
  case C3:  S3;break;
};
```

Control Structures (contd.)




```
for (int Count = 1; Count < 4; Count++)
     body;
```

Comments

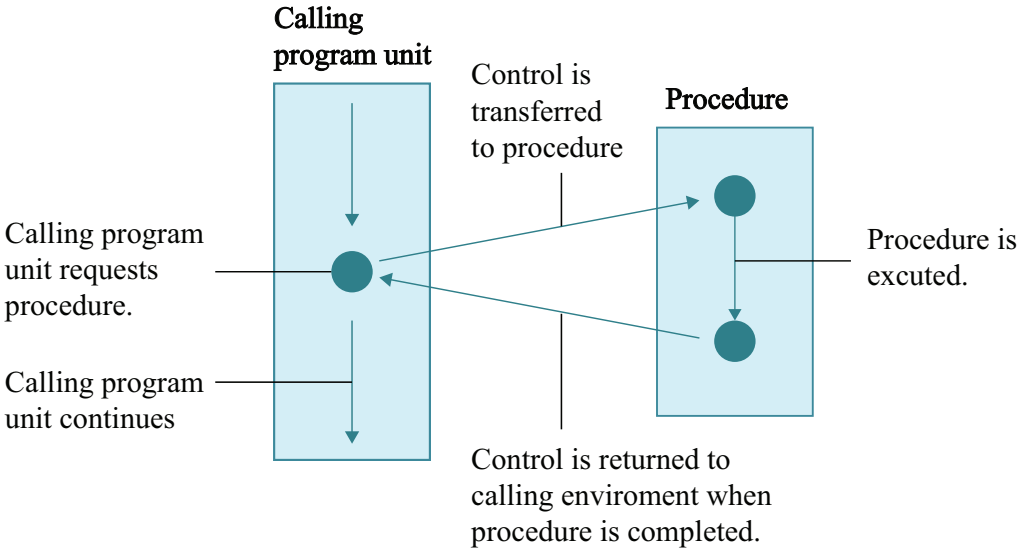
- C/C++, Java

```
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



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.

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

```
void  ProjectPopulation  (float GrowthRate){

int Year;

Population[0] = 100.0;
for (Year = 0; Year =< 10; Year++)
   Population[Year+1] = Population[Year] + (Population[Year]*GrowthRate);

}
```

This declares a local variable named Year.

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

Terminology (contd.)

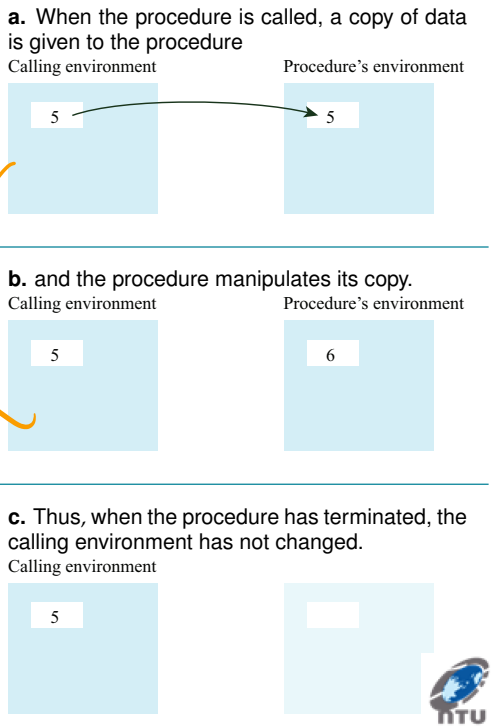
- Procedure's header
- Local vs. global variables
- Formal vs. actual parameters
- Passing parameters 參數傳遞
  - Call by value (passed by value)
  - Call by reference (passed by reference)
  - Call by address: variant of call-by-reference.

Call by Value

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

```
Demo(Actual);
```

不會動到 Calling environment



Call by Reference

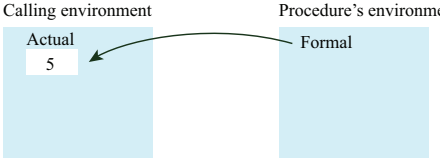
procedure Demo(*Formal*)
Formal ← *Formal* + 1;

Demo(*Actual*);

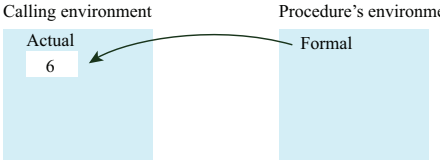
C/C++

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

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



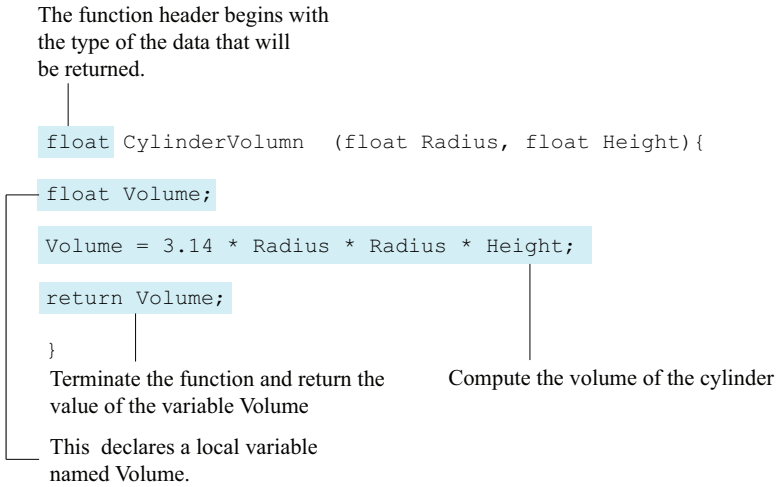
b. Thus, changes directed by the procedure are made to the actual parameter



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.”

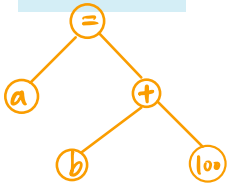


The Translation Process

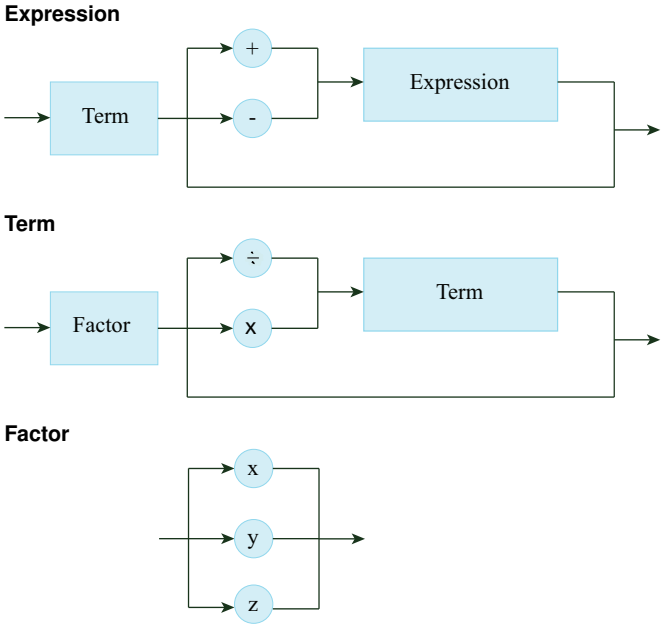
- Lexical analyzer: identifying tokens.
- Parser: identifying syntax & semantics.

ex: a+b=100

分辨"a""+""b""="100"



Syntax Diagrams for Algebra



Grammar for Algebra

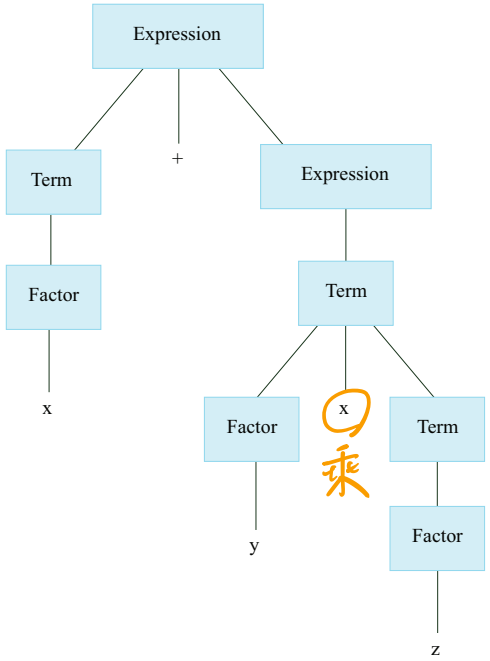
- Expression → Term | Term + Expression
  | Term - Expression
  Term → Factor | Factor * Term | Factor / Term
  Factor → **x** | **y** | **z**

- Starting: Expression
- Nonterminals: Expression, Term, Factor
- Terminals: **x**, **y**, **z**
  終端

Parse Tree

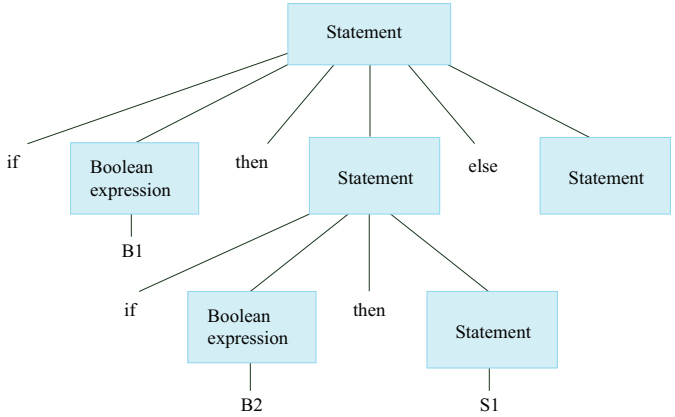
- $x + y \times z$

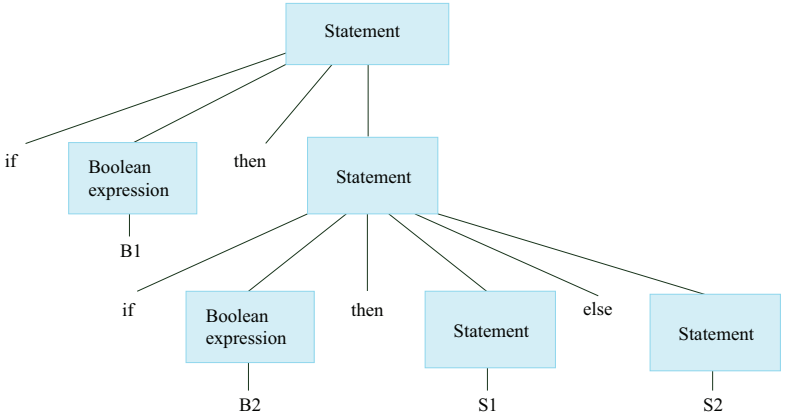
由上頁推導出



Ambiguity → 一句話有很多種文法

- **if** B1 **then if** B2 **then** S1 **else** S2 * 寫程式時，使用括號可避免此狀況





Code Generation

- Coercion: implicit conversion between data types
- Strongly typed: no coercion, data types have to agree with each other.

* C、C++屬於 weakly typed

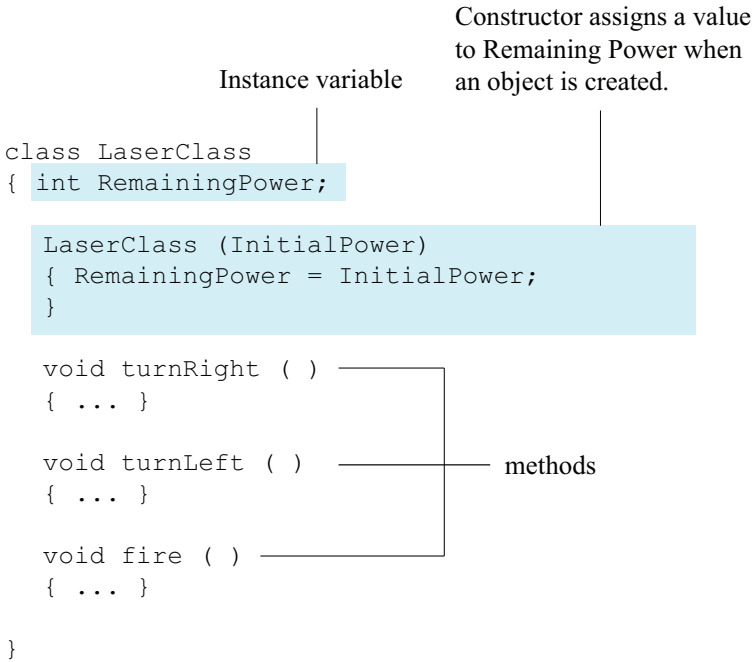
- Code optimization
  - x = y + z;
  - w = x + z;
  - w = y + (z << 1);

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



Encapsulation

- Encapsulation 封裝
  - 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

- 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.

```
Circle();
Rectangle();
```

```
Circle circle;
Rectangle rect;

circle.draw();
rect.draw();
```

Inheritance

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

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

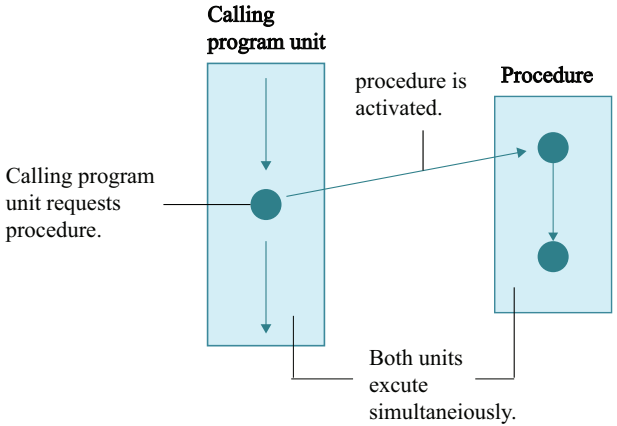
```
Base *base;
Circle circle;
Rectangle rect;

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

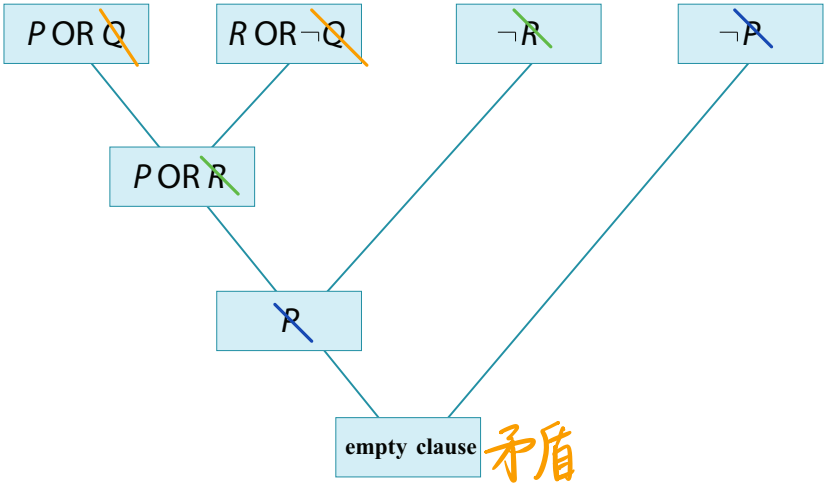


Declarative Programming

- **Resolution**
  - Combining two or more statements to produce a new statement (that is a logical consequence of the originals).
  - (P OR Q) AND (R OR $\neg Q$) resolves to (P OR R) (Q和not Q不可能同時成立)
  - **Resolvent**: A new statement deduced by resolution
  - **Clause form**: A statement whose elementary components are connected by `OR`
- **Unification** 統一化
  - Assigning a value to a variable so that two clauses would be the same.
  - `Unify(Father(Mark,John), Father(x,John))` results in `x is Mark`.

Proof by Resolution (Refutation)

- We know that (*P* OR *Q*) AND (*R* OR ¬*Q*) AND (¬*R*) is true (*KB*, knowledge base).
- We want to prove that *P* is true.
- Prove by showing that *KB* AND ¬*p* is unsatisfiable (empty clause).



Prolog

- **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”, ==,
  - =, <, >, +, -, *, /, =>, =<

大寫為變數
小寫為常數

可運算：term is evaluable ex: X is 1+2, X=3

↳ unification：等號左右要完全一樣 ex: 1+2 = 1+2

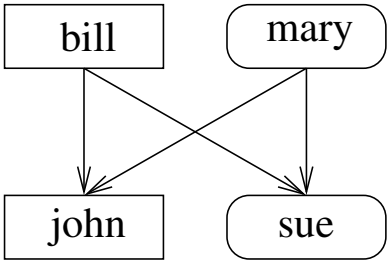
* term == term
ex: 1+2 == 2+1

* evaluable =:= evaluable
ex: 2+3 =:= 1+4

Gnu Prolog

- Gnu prolog http://www.gprolog.org/
- 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. 階乘
- If we want Prolog to compute factorials, we need to tell it what factorials are.

```
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(W,6).
```

Try these commands:

- `listing.`
- `trace.` → 逐步執行
- `notrace.`

This wouldn't work, why?