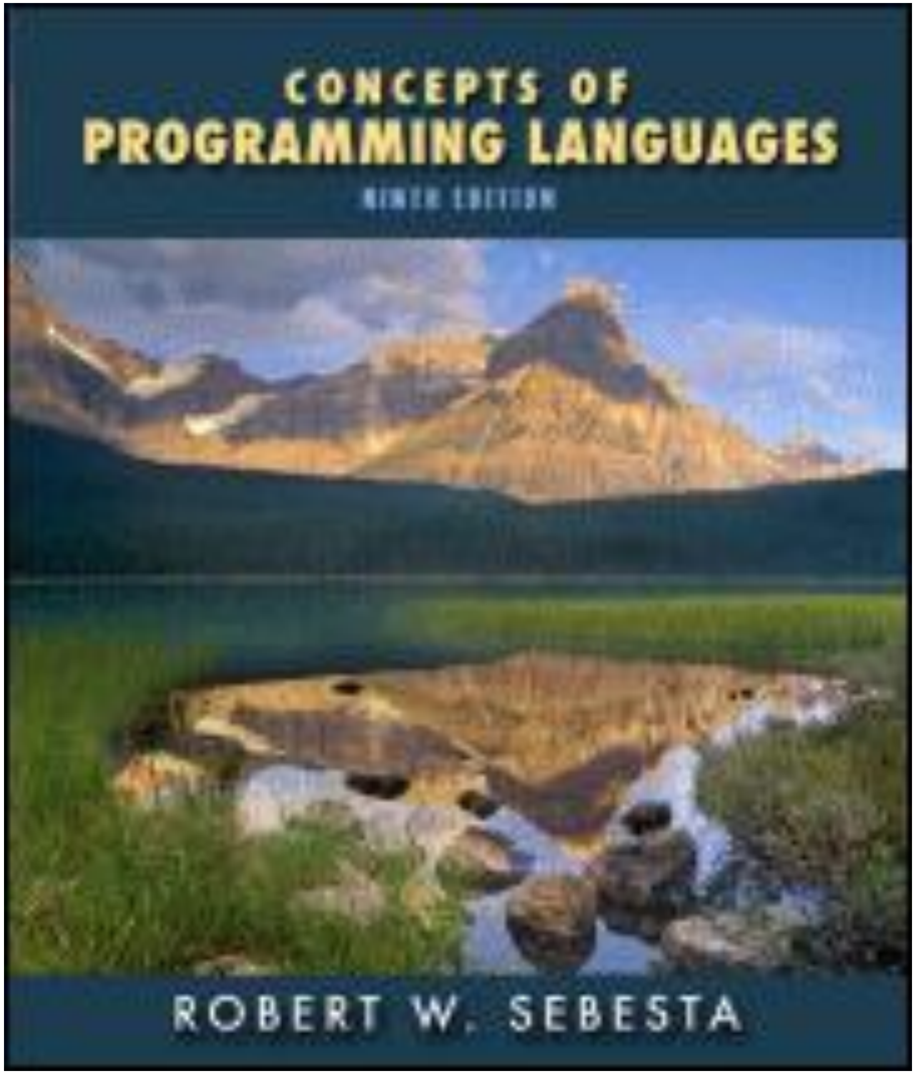
Chapter 1

Preliminaries



Ch01 - Preliminaries

1.1 Reasons for Studying Concepts of Programming Languages

1.2 Programming Domains

1.3 Language Evaluation Criteria

1.4 Influences on Language Design

1.5 Language Categories

1.6 Language Design Trade-Offs*

1.7 Implementation Methods

1.8 Programming Environments*

1.1 Reasons for Studying Programming Languages

- Increased ability to express ideas
- Improved background for choosing appropriate languages
- Increased ability to learn new languages
- Better understanding of significance of implementation
- Better use of languages that are already known
- Overall advancement of computing

1.2 Programming Domains

- Scientific applications
  - Large number of floating point computations
  - Fortran
- Business applications
  - Produce reports, use decimal numbers and characters
  - COBOL
- Artificial intelligence
  - Symbols rather than numbers manipulated
  - LISP

1.2 Programming Domains

- Systems programming
  - Need efficiency because of continuous use
  - C
- Web Software
  - Eclectic collection of languages: markup (e.g. HTML), scripting (e.g. Javascript), general-purpose (e.g. Java)

1.3 Language Evaluation Criteria

- Readability: the ease with which programs can be read and understood
- Writability: the ease with which a language can be used to create programs
- Reliability: conformance to specifications (i.e. performs to its specifications)
- Cost: the ultimate total cost

1.3 Language Evaluation Criteria

Readability

- Overall simplicity
  - A manageable set of features and constructs
  - Few feature multiplicity (means of doing the same operation), e.g. x++; ++x; x+=1; x=x+1;
  - Minimal operator overloading, e.g. &x, x&y; int&
- Orthogonality
  - A relatively small set of primitive constructs can be combined in a relatively small number of ways
  - Every possible combination is legal

1.3 Language Evaluation Criteria

- Orthogonality
  - Non-orthogonal features in C++

	call by value	call by reference
Non-array	void p(int)	void p(int&)
Array	✘	void p(int(&)[3])
		void p(int*)

	return by value	return by reference
Non-array	int p()	int& p()
Array	✘	int (&p())[3]
		int* p()

1.3 Language Evaluation Criteria

- Orthogonality
  - Non-orthogonal features in C++

	pointer	reference	array
pointer	int**	✘ int&*	int(*)[]
reference	int*&	✘ int& &	int(&)[]
array	int*[]	✘ int&[]	int[][]

  - Too little orthogonality
    a lot of exceptions to remember
  - Too much orthogonality
    complex language unless the number of primitives is small, e.g. functional language

1.3 Language Evaluation Criteria

- Syntax design

  Methods of forming compound statements

  Fortran: A brief introduction (References: 1 2)

  - Fortran example 1: Version A

```
    program factorial
!   implicit real(a-h,o-z),integer(i-n)
    implicit none                  ! declarations before statements
    integer :: n, fact             ! :: may be removed here
    print *, "Enter an integer >=0"
    read *, n
    print *, n, '!=', fact(n)
    end ! [program [factorial]]
```

1.3 Language Evaluation Criteria

- Fortran example 1: Version A (Cont'd)

```
function fact(n)
implicit none
integer :: fact,n,i
fact = 1
do i = 2,n,1                 ! default step size = 1
    fact = fact*i            ! favt = fact*i
end do                       ! or, enddo
end ! [function [fact]]
```

- Drawback of implicit declaration

  Assume that fact is misspelled as favt (see above).

  Without implicit none, the error will not be detected.

1.3 Language Evaluation Criteria

- Note: Other iterative statements in Fortran

```
    do 10 i = 2,n,1
       fact = fact*i
10  continue
```

```
    i = 2
    do 30 while (i<=n)
       fact = fact*i
30     i = i+1
```

```
    i = 2
20  if (i<=n) then
       fact = fact*i
       i = i+1
       goto 20
    end if
```

```
    i = 2
    do while (i<=n)
       fact = fact*i
       i = i+1
    end do
```

1.3 Language Evaluation Criteria

- Note: Fortran IV/77 uses fixed format

  Col. 1 : !,*,c,d for comments

  Col. 1-5 : statement label (optional)

  Col. 6 : continuation, any character except '0'

  Col. 7-72 : statements

  Col. 73-80: sequence number

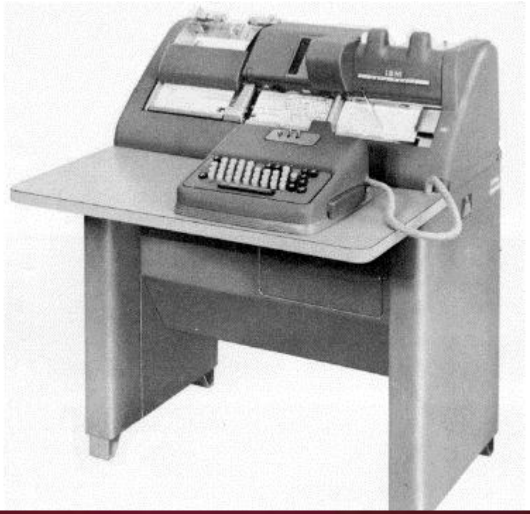
- Note: Fortran 90/95 uses free format, but still line-oriented

  Col. 1-132

```
i = 2; do while (i<=n); fact = fact*i; i = &
i+1; end &       ! ; is a separator
do
```

1.3 Language Evaluation Criteria

- Coding form, card punch & punch card



1.3 Language Evaluation Criteria

- Using G95

  Let factA.f95 be the file that contains the preceding Fortran Program.

  Sample run

  ```
  bsd> g95 factA.f95
  bsd> ./a.out
   Enter an integer >=0
  10
   10 != 3628800
  bsd>
  ```

- Note: .f95 uses free format; .f uses fixed format.

1.3 Language Evaluation Criteria

- Drawback of Version A

A.f95
```
program factorial
...                              ! only the return type is known
integer :: n,fact                ! no information about parameters
...
print *, n, '!=', fact(n,n)      ! Case 1: A single file
end program factorial            ! May issue a warning message
```

B.f95
```
function fact(n)                 ! Case 2: Two separate files
...                              ! No warning message
end function fact                ! bsd> g95 A.f95 B.f95
```

In either case, the program still compiles and works.

1.3 Language Evaluation Criteria

- Fortran example 1: Version B (in 1 file or 2 files)

```
  program factorial
  implicit none
  integer :: n
! integer :: fact      ! don't declare this
  interface
  function fact(n)
  integer :: n,fact
  end function fact
  end interface
  print *, "Enter an integer >=0"
  read *, n
  print *, n, '!=', fact(n)   ! fact(n,n) causes an error
  end program factorial
```

```
! external function
! may be in a separate file
```

```
function fact(n)
implicit none
integer :: fact,n,i
fact = 1
do i = 2,n,1
    fact = fact*i
end do
end function fact
```

1.3 Language Evaluation Criteria

- Fortran example 1: Version C (in 1 file)

```
  program factorial
  implicit none
  integer :: n
! Integer :: fact     ! don't declare this
  print *, "Enter an integer >=0"
  read *, n
  print *, n, '!=',fact(n)
  contains
! Definition of local function fact  (only one nesting level allowed)
  end program factorial
```

```
function fact(n)
implicit none
integer :: fact,n,i
fact = 1
do i = 2,n,1
   fact = fact*i
end do
end function fact
```

1.3 Language Evaluation Criteria

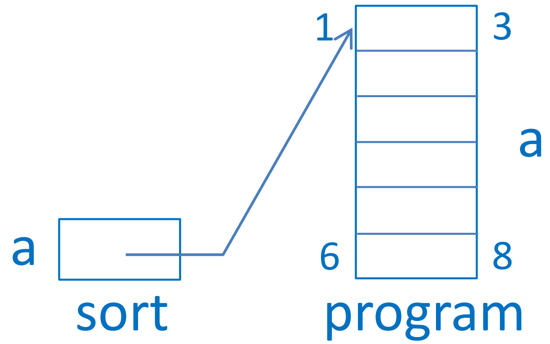
- Fortran example 2: Version A

```
program sorting
implicit none                        ! dimension(6) = dimension(1:6)
integer :: i                         ! or, integer :: a(3:8)
integer, dimension(3:8) :: a
print *, 'Enter 6 integers'
read *, a                            ! key in 6 integers in one or more lines
! read *, a(3), a(4), a(5), a(6), a(7), a(8)
! read *, (a(i), i = 3,8)            ! implied do loop
! read *, a(3:8)                     ! array bounds may be omitted
                                     ! a = a(:) = a(3:) = a(:8) = a(3:8)
                                     ! array section, e.g. a(5:) = a(5:8)
```

1.3 Language Evaluation Criteria

- Fortran example 2: Version A (Cont'd)

```
call sort(a)
print *, a                          ! output 6 integers in a line
contains
subroutine sort(a)                  ! sorting 6-element array
implicit none
integer, dimension(6) :: a          ! explicit shape array
! may use other indices
! e.g. dimension(2:7)
! but, must agree with the actual
! parameter in size, i.e. 6 elements
! otherwise, error.
```



1.3 Language Evaluation Criteria

- Fortran example 2: Version A (Cont'd)

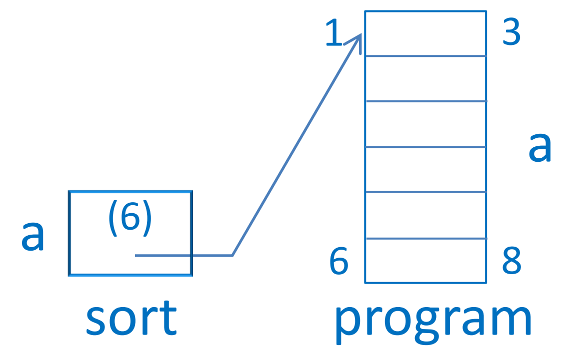
```
integer :: i,j,z
outer: do i = 1,5                    ! bubble sort
   inner: do j = 6,i+1,-1
               if (a(j)<a(j-1)) then
                  z = a(j); a(j) = a(j-1); a(j-1) = z
               end if
   end do inner
end do outer
end subroutine sort
end program sorting
```

1.3 Language Evaluation Criteria

- Fortran example 2: Version B

```
program sorting
implicit none
integer :: i
integer, dimension(3:8) :: a
print *, 'Enter 6 integers'
read *, a
call sort(a)
print *, a
contains
subroutine sort(a)          ! sorting array of varied size
implicit none
```

Have to pass the array’s shape
i.e. # of dimensions +
# of elements in each dimension
Here, the shape of a is (6).



1.3 Language Evaluation Criteria

- Fortran example 2: Version B (Cont'd)

```
integer, dimension(:) :: a          ! assumed-shape array
integer :: i,j,z
outer: do i = 1, ubound(a,1)-1      ! upper bound of 1st dimension
   inner: do j = ubound(a,1), i+1, -1
            if (a(j)<a(j-1)) then
               z = a(j); a(j-1) = a(j); a(j) = z
            end if
   end do inner                     ! may not specify the upper bound
end do outer                        ! but, may specify the lower bound
end subroutine sort                 ! e.g. dimension(2:)
end program sorting                 ! then, ubound(a,1) = 7 (= 2+6-1)
```

1.3 Language Evaluation Criteria

- Fortran example 3: Version A

```
program matrix
implicit none                      ! column-major order
integer :: i,j                     ! let the input be 1 2 3 4 5 6
integer, dimension(2,3) :: a       ! then, the array a is:
print *, 'Enter 6 integers'        ! 1 3 5
read *, a                          ! 2 4 6
call interchange(a,1,2)
do i = 1,2          ! as before, array bounds may be omitted
   print *, a(i,:)  ! a(i,:) = a(i,1:) = a(i,:3) = a(i,1:3)
end do              ! a = a(:,:) = a(1:2,1:3) = ...
contains
```

1.3 Language Evaluation Criteria

- Fortran example 3: Version A (Cont'd)

```
subroutine interchange(a,i,j)
implicit none
integer :: i,j
integer, dimension(:,:) :: a
integer, dimension(size(a,2)) :: t    ! or, ubound(a,2)
t = a(i,:)                            ! lbound(a,2) = 1 implies
a(i,:) = a(j,:)                       ! size(a,2) = ubound(a,2)
a(j,:) = t          ! or, a(j,1:ubound(a,2))
end subroutine interchange

end program matrix
```

1.3 Language Evaluation Criteria

- Fortran example 3: Version B

```
! rewrite subroutine interchange of version A as follows
subroutine interchange(a,i,j)
implicit none
integer :: i,j
integer, dimension(:,:) :: a
a(i,:) = a(i,:) + a(j,:)
a(j,:) = a(i,:) - a(j,:)
a(i,:) = a(i,:) - a(j,:)
end subroutine interchange
```

1.3 Language Evaluation Criteria

- Syntax design

  Form and meaning

  - Language constructs with distinct semantics should not have similar syntax.
  - Example – Fortran IV/77

    Computed goto n = 2

    goto (10,20,30), n

    Assigned goto assign 20 to n

    goto n, (10,20,30)

    where n is an integer variable

    These are obsolescent features in Fortran 90/95.

1.3 Language Evaluation Criteria

- Example – C++

```
&  &&     bitwise vs logical
|  ||     bitwise vs logical
```

C++ provides alternative names for some operators.

```
&     bitand      &&    and      { }    <% %>
|     bitor       ||    or       [ ]    <:  :>
^     xor                        #      %:
~     compl       !     not
&=    and_eq
|=    or_eq
^=    xor_eq
!=    not_eq
```

1.3 Language Evaluation Criteria

- Example – C/C++

  The static keyword is overloaded in C/C++.

  Related to lifetime

  - Local static objects, e.g. void p() { static int x=2; }
  - Static data members

  Related to scope (or linkage)

  - Global static objects and functions

  Related to neither lifetime nor scope

  - Static member functions

    Meaning: They can only access static data members.

1.3 Language Evaluation Criteria

- Example (Cont'd)

  On linkage in C/C++

  External linkage

  The name is visible in every translation unit of the program. e.g. functions and global objects (possibly declared extern)

  Internal linkage

  The name is visible only in the translation unit in which it is declared, e.g. functions and global objects declared static

  No linkage

  The name is visible only in the scope in which it's declared e.g. local objects

1.3 Language Evaluation Criteria

- Example (Cont'd)

File 1

```
/*extern*/ int x = 1;          // external linkage, definition
/*extern*/ void p();           // external linkage, declaration
int main() { p(); }
```

File 2

```
#include <iostream>
extern int x;                  // external linkage, declaration
/*extern*/ void p()            // external linkage, definition
{
    int y = 2;                 // no linkage
    std::cout << x+y;
}
```

1.3 Language Evaluation Criteria

- Example (Cont'd)

File 1

```
static int x = 1;            // internal linkage
/*extern*/ void p();         // external linkage
int main() { p(); }          // linking error
```

File 2

```
#include <iostream>
extern int x;                // external linkage
static void p()              // internal linkage
{
    int y = 2;               // no linkage
    std::cout << x+y;        // linking error
}
```

1.3 Language Evaluation Criteria

- Example (Cont'd)

  C++ unnamed namespace

  - C++ prefers unnamed namespace to global static.
  - Members of an unnamed namespace are visible only in the containing translation unit.
  - Members of an unnamed namespace are referred to without qualification.
  - Examples:

```
namespace { int x = 1; }
int main() { cout << x; }
```

```
namespace { int x = 1; }
int x = 2;
int main()
{
    cout << x;    // ambiguous
    cout << ::x;
}
```

1.3 Language Evaluation Criteria

- Example (Cont'd)

File 1

```
namespace { int x = 1; }
void p();
int main() { p(); }          // linking error
```

File 2

```
#include <iostream>
extern int x;
namespace
{
    void p()  {  int y = 2; std::cout << x+y;  }    // linking error
}
```

1.3 Language Evaluation Criteria

Writability

- Simplicity and orthogonality
  - Few constructs, a small number of primitives, a small set of rules for combining them
- Support for abstraction
  - The ability to define and use complex structures or operations in ways that allow details to be ignored, i.e. data abstraction and procedural abstraction.
- Expressivity
  - Example: the inclusion of for statement in many modern languages

1.3 Language Evaluation Criteria

Reliability

- Type checking
  - Testing for type errors
- Exception handling
  - Intercept run-time errors and take corrective measures
- Aliasing
  - Presence of two or more distinct referencing methods for the same memory location
- Readability and writability

1.3 Language Evaluation Criteria

Cost

- Training programmers to use language
- Writing programs
- Compiling programs
- Executing programs
- Language implementation system: availability of free compilers
- Reliability: poor reliability leads to high costs
- Maintaining programs

1.3 Language Evaluation Criteria

Others

- Well-definedness
  - The completeness and precision of the language's official definition
- Portability
  - The ease with which programs can be moved from one implementation to another
  - One factor: sizes of primitive types
  - For portability, some languages specify sizes of primitive types, e.g. Java.

1.3 Language Evaluation Criteria

- For efficiency, most languages don't specify sizes of primitive types.

  E.g. C, C++

  Plain ints have the natural size suggested by CPU.

  $2 \leq$ sizeof(short) $\leq$ sizeof(int) $\leq$ sizeof(long)

  $\leq$ sizeof(long long) $\geq$ 8

  sizeof(signed T) = sizeof(unsigned T)

  sizeof(float) $\leq$ sizeof(double) $\leq$ sizeof(long double)

1.4 Influences on Language Design

- Computer Architecture
  - Well-known computer architecture: Von Neumann
  - Imperative languages, most dominant, because of von Neumann computers
    - Data and programs stored in memory
    - Memory is separated from CPU
    - Instructions and data are piped from memory to CPU
  - Basis for imperative languages
    - Variables model memory cells
    - Assignment statements model piping
    - Iteration is efficient

1.4 Influences on Language Design

- Programming Methodologies
  - Late 1960s: People efficiency became important; readability, better control structures
    - structured programming
      gotos considered harmful
    - top-down design and step-wise refinement
  - Late 1970s: Process-oriented to data-oriented
    - data abstraction
  - Middle 1980s: Object-oriented programming
    - Data abstraction + inheritance + polymorphism

1.5 Language Categories

- Imperative
  - Central features are variables, assignment statements, and iteration
  - Examples: C, Pascal
- Functional
  - Based on lambda calculus – a study of functions
  - Variables denote constant values

    $f(x) = x^2 + 3x - 5 \quad \Rightarrow \quad f(7) = 7^2 + 3 \cdot 7 - 5$
  - No assignment and iteration (purely functional)
  - Recursion only (purely functional)
  - Examples: impure: LISP, Scheme, SML; pure: Haskell

1.5 Language Categories

- Imperative style

```
function fact(n)
implicit none
integer :: fact,n,i
fact = 1
do i = 2,n,1
    fact = fact*i
end do
end function fact
```

n	3
i	~~2~~ ~~3~~ 4
fact	~~1~~ ~~2~~ 6

O(n) time, O(1) space

1.5 Language Categories

- Functional style

```
recursive function fact(n) result(r)
implicit none
integer :: n,r          ! can't declare fact
if (n==0) then          ! fact's type = r's type
    r = 1               ! initialize r
else
    r = n*fact(n-1)     ! initialize r
end if
end function
```

n	0
r	1
n	1
r	1
n	2
r	2
n	3
r	6

O(n) time, O(n) space

- Recursion isn't allowed in Fortran IV/77.

1.5 Language Categories

- Recursive functions in Fortran 90/95 can't be written as

```
function fact(n)              ! Error: Unexpected array reference
integer :: n,fact
if (n==0) then;  fact = 1; else; fact = n*fact(n-1); end if
end function
```

- Iterative functions in Fortran 90/95 can be written as

```
function fact(n) result(r)
integer :: r,n
r = 1
do while (n>0) ; r = r*n; n = n-1; end do
end function
```

1.5 Language Categories

- Logic
  - Based on math logic
  - Variables denote constant values
    - $\forall x\ (x > 0 \rightarrow -x < 0)$

      Let $x = 2$, then $2 > 0 \rightarrow -2 < 0$
  - No assignment and iteration; Recursion only
  - Programming in relations, rather than functions
    - ∀x likes(x,C++)

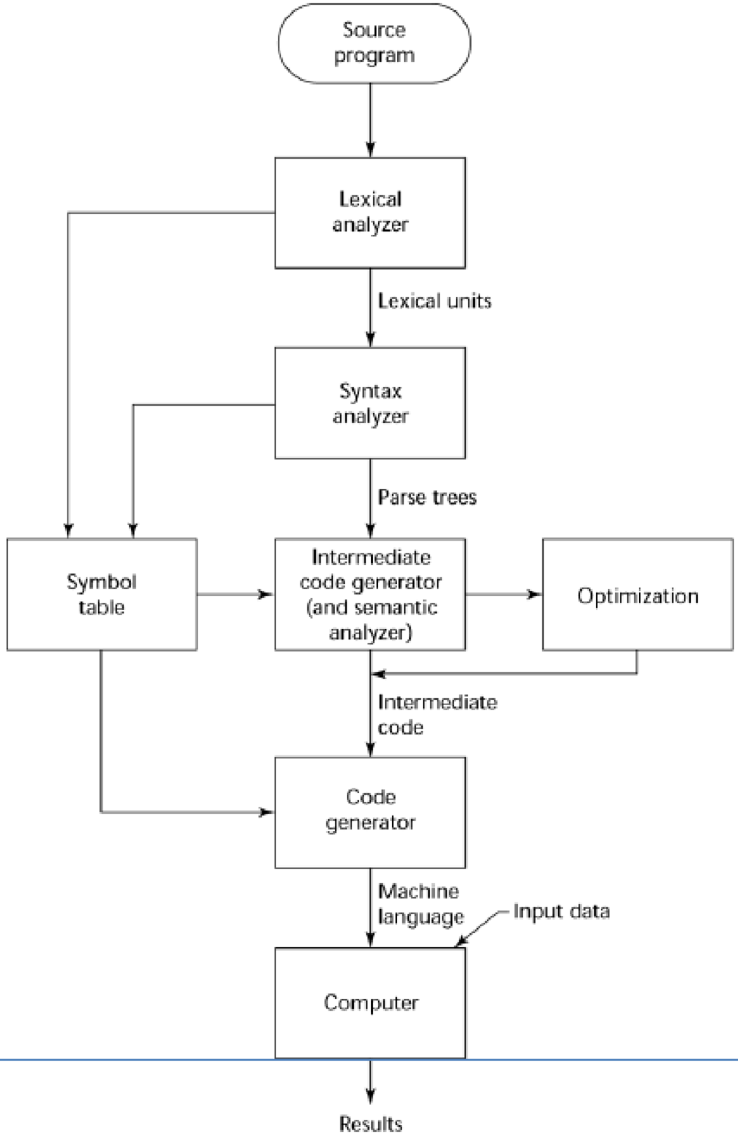
      likes(John,Mary)
  - Programming in logic (rule-based)
    - ∀x (likes(x,C++) → crazy(x))
  - Example: Prolog

1.5 Language Categories

- Object-oriented
  - Data abstraction, inheritance, late binding
  - Examples: Java, C++
- Markup
  - New
  - Not a programming per se, but used to specify the layout of information in Web documents
  - Examples: XHTML, XML

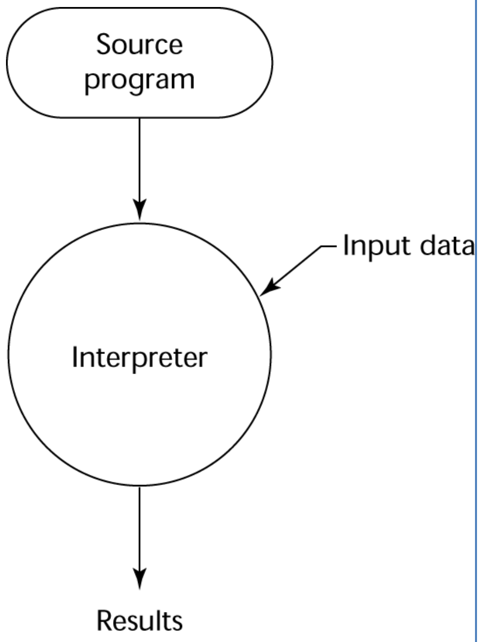
1.7 Implementation Methods

- Compilation
  - Lexical analyzer
    x = 3y+4
  - Syntax analyzer
    x = y+4)*z
  - Semantic analyzer
    int x = 5;
    *x = 7;   // type error
  - Compile once, fast execution
  - Platform dependent
  - Examples – C/C++



1.7 Implementation Methods

- Pure interpretation
  - Platform independent
  - Interpret a program each time it is executed
  - Slower execution - interpret a statement each time it is executed
  - Some might do some preprocessing to speed up execution.
  - Examples
    Early languages (APL, SNOBOL, Lisp)
    JavaScript



1.7 Implementation Methods

```
// A simple JavaScript example
<html>
<body>
<h2>The following is generated by a Javascript program.</h2>
<br>

<script language="JavaScript">
document.write("<h3>Click the button to compute factorial</h3><br>")
function f(n)
{
    var r=1                          // local variable; w/o var, global
    for (var i=2;i<=n;i++) r*=i      // ; isn't needed here
    return r                         // it is a separator
}
```

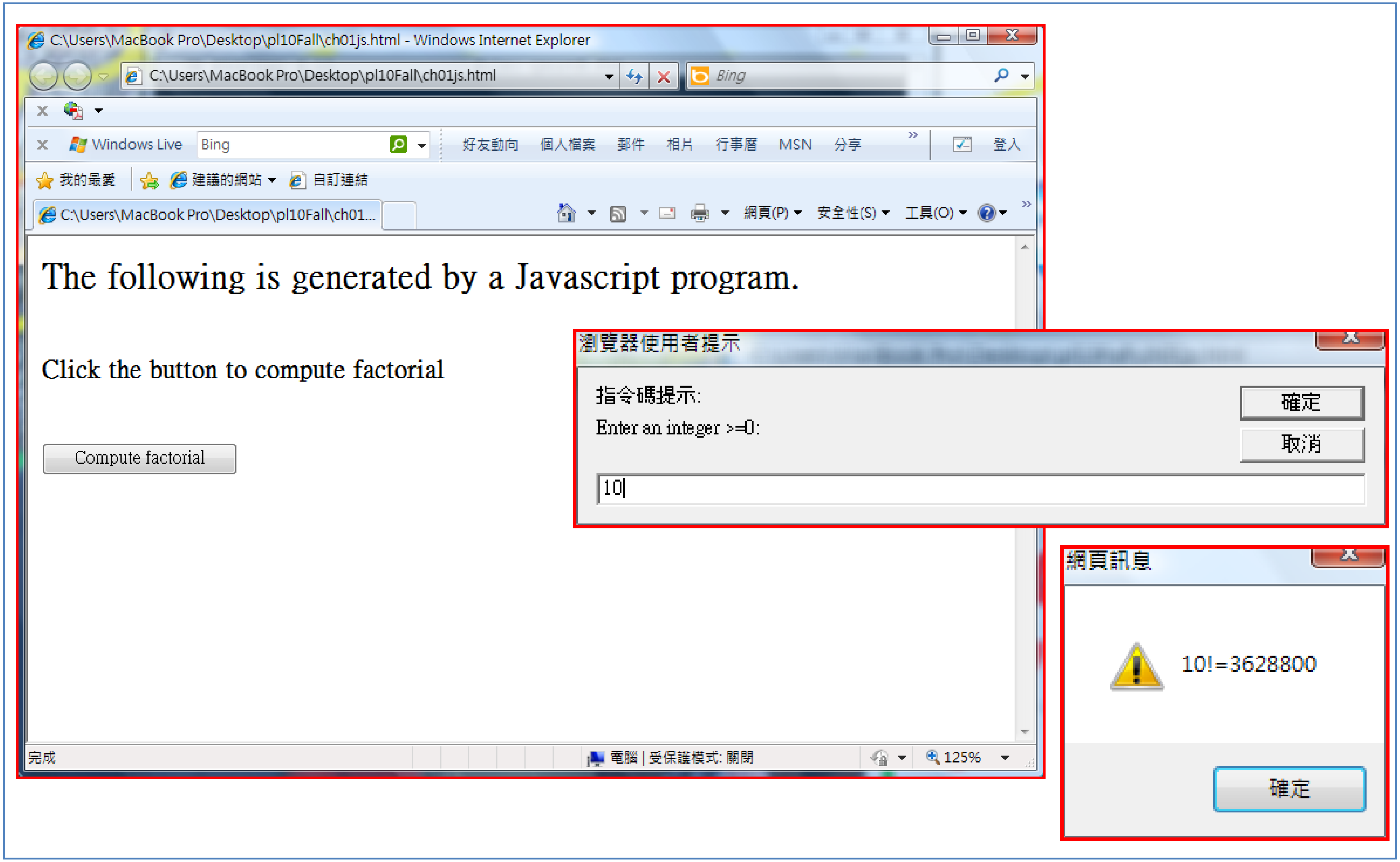
Html interpreter

Javascript interpreter

1.7 Implementation Methods

```
function main()
{
    var n=+prompt("Enter an integer >=0: ")  // convert to number
    if (n>=0) alert(n+"!="+f(n))
    else alert("Illegal input")
}
</script>
<form>
<input type=button value="Compute factorial" onClick=main()>
</form>
</body>
</html>
```

1.7 Implementation Methods



1.7 Implementation Methods

- var n = +prompt("Enter an integer >=0: ")

input	n	typeof n	n>=0
12	12	number	true
hello	NaN	number	false

NaN = Not a Number

"number" is double-precision floating-point number

- var n = prompt("Enter an integer >=0: ")

input	n	typeof n	n>=0
12	"12"	string	true
hello	"hello"	string	false

Still work, but inefficient: the test i <= n requires n be converted to a number

1.7 Implementation Methods

- Comment

  JavaScript is an untyped language, i.e. the variables have no fixed types.

  Q: What would happen if we write

  var r = "1"

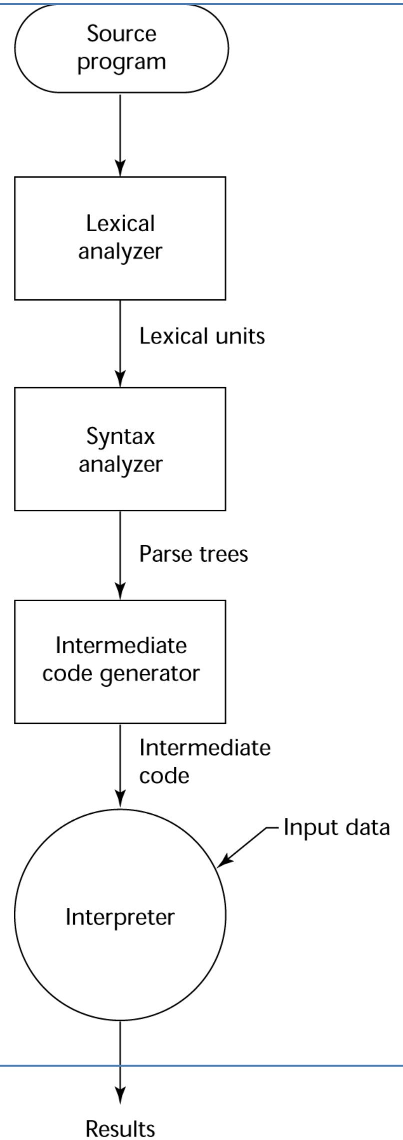
  in function f?

  A: If n = 0 or 1, the function returns a string, i.e. "1".

  If $n \geq 2$, the function returns a number, i.e. n!, because the 1st time the statement r *= i is executed, the type of r is changed from string to number.

1.7 Implementation Methods

- Hybrid Implementation Systems
  - Platform independent
  - Faster than pure interpretation
  - Example
    - Functional languages
      SECD machine
      Stack, **E**nvironment, **C**ode, **D**ump
      are internal registers of the machine.
    - Prolog
      Warren Abstract Machine (WAM)



1.7 Implementation Methods

- Example – Java

  A Java program is compiled to JVM (Java Virtual Machine) bytecode, which is then interpreted.

Java: A brief introduction

- // Java example 1

```
class demo {
    public static void main(String[] args)
    {
        System.out.println("Hello");
    }
}
```

1.7 Implementation Methods

- // Java example 1 (Cont'd)

  Sample run

  Let ex1.java be the file that contains this Java program

```
bsd> javac ex1.java        # Java compiler yields demo.class
bsd> java demo             # Java interpreter
Hello
bsd> javap demo            # Java disassembler
Compiled from "ex1.java"
class demo extends java.lang.Object {
    demo();
    public static void main(java.lang.String[]);
}
```

1.7 Implementation Methods

- // Java example 1 (Cont'd)

bsd> javap -c demo   # print out disassembled code

```
Compiled from "ex1.java"
class demo extends java.lang.Object {
demo();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return
public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   3:   ldc             #3; //String Hello
   5:   invokevirtual   #4; //Method java/io/PrintStream.println:(Ljava/lang/String;)V
   8:   return
}
```

1.7 Implementation Methods

- On Java package (analog to C++ namespace)

  Package level access control

  - Public class is accessible from other packages.
  - Package class is accessible only from the same package.

  A file may contain at most one public class.

  A file may contain any number of package classes.

  If a file contains a public class, then file name = class name

  Class level access control

  - private ⊂ package ⊂ protected ⊂ public
  - Package members can be accessed by code in the same package.

1.7 Implementation Methods

- // Java example 2

```
// File ex2.java contains two classes in an unnamed package
class demo {                    // package class
    public static void main(String[] args)
    {
        System.out.println(10+"! = "+factorial.f(10));
    }
}                               // access factorial's package member

class factorial {               // package class
    static int f(int n) { return n==0? 1: n*f(n-1); }
}                               // package member
```

1.7 Implementation Methods

- // Java example 2 (Cont'd)

  Sample run

  bsd> javac ex2.java # yield demo.class and factorial.class

  bsd> java demo

  10! = 3628800

- // Java example 3 on package naming convention

  •

  anonymous ∋ ex3.java mylib ⇐ directory = package

  package

  factorial.java ∈ mylib package

1.7 Implementation Methods

- // Java example 3

```
// File ./ex3.java
import mylib.factorial;   // or, import mylib.*;
class demo {              // ∵ factorial class is public
    public static void main(String[] args)
    {
        System.out.println(10+"! = "+factorial.f(10));
    }
}                         // ∵ the member f is public
// Alternatively, remove the import statement and write
// mylib.factorial.f(10)
```

1.7 Implementation Methods

- // Java example 3 (Cont'd)

```
// File ./mylib/factorial.java
package mylib;
public class factorial {                    // public class
    public static int f(int n) { return n==0? 1: n*f(n-1); }
}                                           // public member
```

Sample run

```
bsd> javac ex3.java
# yield ./demo.class and ./mylib/factorial.class
bsd> java demo
10! = 3628800
```

1.7 Implementation Methods

- Example

  Perl (Practical Extraction and Report Language)

  A Perl interpreter first compiles a Perl program to a code tree, and then executes the program by walking the tree.

Perl: A brief introduction (Reference: PerlIntro)

- Perl has three distinct namespaces for variables, denoted by the 1st character of the variables' names.

  $scalar (numbers, strings, references)

  @array

  %hash (i.e. associative array)

1.7 Implementation Methods

- ```
  # Perl example 1
  print "Enter an integer >=0: ";
  # ; is a terminator
  # () may be omitted, if a function is predefined before.

  $n = <stdin>;                # or, $n = <>;
  # $n is a string of the input line, including '\n'
  chomp $n;                    # or, together, chomp($n=<>);
  # remove '\n'; useful in print "$n! = $r\n";

  # On interpolation
  # print "$n! = $r\n";        # 5! = 120↲      ($n = "5", say)
  # print '$n! = $r\n';        # $n! = $r\n     (not interpolated)
  ```
  ```

1.7 Implementation Methods

- # Perl example 1 (Cont'd)

```
# convert $n to a number (similar to atoi/atof in C/C++)
$r = 1;
for (my $i=1;$i<=$n;$i++)  {        # local variable
    $r *= $i;                        # always enclose body in {}
}
print $n,'! = ',$r,"\n";
```

Sample run

Let ex1.pl be the file that contains this Perl program

```
bsd> perl ex1.pl
Enter an integer >=0: 10
10! = 3628800
```

1.7 Implementation Methods

- On list and array

  @a = (1,2,3,4,5);   # @a = (1..5);

  array   list

  A list is a constant; but an array is a variable.

  E.g.

  The following array operations can't be applied to lists.

  $i = push @a,9;   # push_back;   @a = (1,2,3,4,5,9), $i = 6

  $i = pop @a;   # pop_back;   @a = (1,2,3,4,5), $i = 9

  $i = shift @a;   # pop_front;   @a = (2,3,4,5), $i = 1

  $i = unshift @a,8;   # push_front;   @a = (8,2,3,4,5), $i = 5

  # of elements in @a

1.7 Implementation Methods

- On list and array (Cont'd)

```
@a = (1..$n);               # if $n is converted to 0, @a is empty
$r = 1;
for ($i=0;$i<$n;$i++)  { $r *= $a[$i]; }

# Given @array, $#array = the highest index of @array
for ($i=0;$i<=$#a;$i++) { $r *= $a[$i]; }

while ($#a>=0) {            # eventually, @a is empty and $#a = -1
    ($i,@a) = @a;           # list assignment
    $r *= $i;               # same as $i = shift @a;
}
```

1.7 Implementation Methods

- Iteration over a list or an array

```
foreach $i (1..$n)  { $r *= $i; }    # foreach = for
foreach $i (@a)  { $r *= $i; }
foreach (@a)  { $r *= $_; }          # predefined variable $_

# The loop variable references to the list or array elements.
foreach (@a)  { $_++; }              # C++11
                                     # int a[5]; for (int& i : a) i++;
print @a;                            # 23456    ($n = "5", say)
print "@a";                          # 2 3 4 5 6
# Array elements are separated by the value of $", which
# is a space by default.
```

1.7 Implementation Methods

- # Perl example 2

```
# Parameters are passed in the special array @_
sub f                                   # @_ = ($n) in this case
{                                       # my $x = shift;
    my $r = 1;                          # my ($x) = @_;
    for (1..$_[0]) { $r *= $_;  }       # for (1..$x) { $r *= $_; }
    return $r;
}                                       # default is the array @_
print "Enter an integer >=0: ";
$n = <>*1;                              # $n is a number
print "$n! = ", f($n), "\n";            # print "$n! = ", f $n, "\n"; ✗
                                        # ($n, "\n") is passed
```

1.7 Implementation Methods

- # Perl example 3

```
sub gcd
{
    my  ($a,$b) = @_;
    return $b==0? $a: gcd($b,$a%$b);   # must parenthesize
}
print "Enter two integers in a line: ";
($a,$b) = split ' ', <>;       # split the input line into an array
print gcd $a,$b;               # of strings separated by space
# Or, combining the preceding two lines into one:
# print gcd split ' ', <>;
```

1.7 Implementation Methods

- On hash (associative array)

```
%h = (Snoopy=>"dog",Garfield=>"cat", Pluto=>"dog");
# or, %h = ("Snoopy","dog","Garfiled","cat", "Pluto","dog");
$h{Micky} = "mouse";            # $h{"Micky"} = "mouse";
delete $h{Snoopy};
# Quotes are optional for keys when using => and {}

foreach (keys %h) {             # keys %h = a list of keys
    print "$_ => $h{$_}\n";     # values %h = a list of values
}
while (($key,$value) = each %h) {
    print "$key => $value\n";
}
```

```
Output:
Garfield => cat
Pluto => dog
Micky => mouse
```

1.7 Implementation Methods

- # Perl example 4

```
sub f          # if (!exists $h{"$n!"}) { $h{"$n!"}=$n==0? 1: $n*f($n-1); }
{
    my $n = shift;
     $h{"$n!"} = $n==0? 1: $n*f($n-1), if !exists $h{"$n!"};
    return $h{"$n!"};          # $n = <>*1 fails, if input 0
}
while (print("Enter an integer>=0: "), chomp($n=<>)) {
     print "$n! = ",f($n),"\n";
     print "Hash table created so far\n";
     foreach (keys %h) { print "$_ => $h{$_}\n";  }
}
#   On eof, $n=undef ⇒ chomp($n)=0, i.e. # of eoln removed
```

1.7 Implementation Methods

- Context

  There are two major contexts: scalar and list.

  An expression evaluates to a list in list context, but a scalar value in scalar context.

```
@a = (1,3,5);      # (1,3,5) in list context; @a = (1,3,5)
$a = (1,3,5);      # (1,3,5) in scalar context;
                   # $a = 5 (like C's comma expression)
@b = @a;           # @a in list context; @b = (1,3,5)
$b = @a;           # @a in scalar context
                   # $b = 3 (i.e. number of elements in @a)
@a = $b            # $b in list context; @a = (3)
```

1.7 Implementation Methods

- Context (Cont'd)

```
%h = (p=>1,l=>2,u=>3,t=>4,o=>5);
$c = %h;            # %h in scalar context; $c = "3/8"
                    # i.e. number of used buckets / number
                    # of allocated buckets
# The function scalar forces a scalar context.
print %h;           # l2u3p1o5t4
print scalar %h;    # 3/8
# An example
@a = (1..5);
$r = 1;
for ($i=0;$i<@a;$i++) { $r *= $a[$i]; }
```

1.7 Implementation Methods

- Compiled languages? Interpreted languages?
  - Theoretically, any language may be compiled or interpreted.
  - "C++ is a compiled language" is purely due to common implementation practice.
  - Many languages have been implemented using both compilers and interpreters