What is a Compiler?

Recall from your study of assembly language or computer organization the kinds of instructions that the computer's CPU is capable of executing.

(1) add two numbers stored in memory,
(2) move numbers from one location in memory to another,
(3) move information between the CPU and memory.

COMPILER

The function of the compiler is to accept statements and translate them into sequences of machine language operations which, if loaded into memory and executed, would carry out the intended computation.

COMPILER - Definition

A software translator which accepts, as input, a program written in a particular high-level language and produces, as output, an equivalent program in machine language for a particular machine.

COMPILER

✓The input program is known as the **source program**, and its language is the **source language**.

✓The output program is known as the **object program**, and its language is the **object language**.

✓A compiler translates source language programs into equivalent object language programs.

COMPILER – Examples

- A Java compiler for the Apple Macintosh
- A COBOL compiler for the SUN
- A C++ compiler for the Apple Macintosh

What is a Compiler? - Sample Problem

Show the output of a Java native code compiler, in any typical assembly language, for the following Java input string:

```
while (x<a+b) x = 2*x;
```

Solution:

```
L1:     LOD R1,A            // Load A into reg. 1
        ADD R1,B            // Add B to reg. 1
        STO R1,Temp1        // Temp1 = A + B
        CMP X,Temp1         // Test for while condition
        BL L2               // Continue with loop if
                            // X<Temp1
        B L3                // Terminate loop
L2:     LOD R1,2
        MUL R1,X
        STO R1,Temp2
        LOD R1,Temp2
        STO Temp2,X         // X = 2*X
        B L1                // Repeat loop
L3:
```

What is a Compiler?

If a portion of the input to a Java compiler looked like this:

$$A = B + C * D;$$

the output corresponding to this input might look something like this:

```
LOD   R1,C        // Load the value of C into reg 1
MUL   R1,D        // Multiply the value of D by reg 1
STO   R1,TEMP1    // Store the result in TEMP1
LOD   R1,B        // Load the value of B into reg 1
ADD   R1,TEMP1    // Add value of Temp1 to register 1
STO   R1,TEMP2    // Store the result in TEMP2
LOD   R1,TEMP2
STO   R1,TEMP2    // Move TEMP2 to A, the final result
```

What is a Compiler?

In designing a compiler, the primary concern is that the object program be semantically equivalent to the source program

(i.e. that they mean the same thing, or produce the same output for a given input).

Object program efficiency is important, but not as important as correct code generation.

The advantages of a high-level language over machine or assembly language

- Machine language (and even assembly language) is difficult to work withand difficult to maintain.
- With a high-level language you have a much greater degree of machine independence and portability from one kind of computer to another (as long as the other machine has a compiler for that language).

The disadvantages of high-level languages

- The programmer doesn't have complete control of the machine's resources (registers, interrupts, I/O buffers).
- The compiler may generate inefficient machine language programs.
- Additional software – the compiler – is needed in order to use a high-level language.

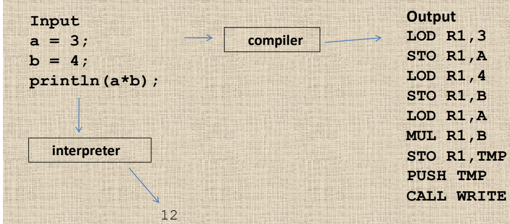
The advantages of a high-level language over machine or assembly language

- You don't have to retrain application programmers every time a new machine (with a new instruction set) is introduced.
- High-level languages may support data abstraction (through data structures) and program abstraction (procedures and functions).

Interpreter

An interpreter is software which serves a purpose very similar to that of a compiler.

The input to an interpreter is a program written in a high-level language, but rather than generating a machine language program, the interpreter actually carries out the computations specified in the source program.



COMPILER

- ✓There is a distinction between **compile time**, the time at which a source program is compiled, and **run time**, the time at which the resulting object program is loaded and executed.
- ✓Syntax errors are reported by the compiler at compile time as compile-time errors. Other kinds of errors not generally detected by the compiler are called run-time errors.

Compile-Time Errors	Run-Time Errors
`a = ((b+c)*d;`	`x = a-a;`
	`y = 100/x; // division by 0`
`if x<b fn1();`	`Integer n[] = new Integer[7];`
`else fn2();`	`n[8] = 16; // invalid subscript`

Sample Problem

Show the output of a Java native code compiler, in any typical assembly language, for the following Java input string:

```
a)
if (-a<b)
      y=-(a+b;
else
      x=2*(y-3);
b)
while (a<=b)
   if (b!=a)
      a=b;
```

Sample Problem

Show the compiler output and the interpreter output for the following Java source code:

```
for (i=1; i<=4; i++) System.out.println (i*3);
```

Solution:

Compiler

```
   LOD R1,1
   STO R1,I
L1:CMP I,4
   BH L2          {Branch if i>Temp1}
   LOD R1,I
   MUL R1,3
   STO R1,Temp1
   PUSH Temp1
   CALL Println
   LOD R1,1
   ADD R1,I       {Add 1 to i}
   STO R1,I
   B L1
L2:
```

Interpreter

```
3 6 9 12
```

Comparison Symbols	Comparison code	Complement code
==	BE	BNE
<	BL	BHE
>	BH	BLE
<=	BLE	BH
>=	BHE	BL
!=	BNE	BE

COMPILER

A compiler is a program, and it must be written in some language (machine, assembly, high-level). We are dealing with three languages:

1. the **source language**, i.e. the input to the compiler,
2. the **object language**, i.e. the output of the compiler, and
3. the **language in which the compiler is written**, or the language in which it exists, since it might have been translated into a language foreign to the one in which it was originally written.

COMPILER

A Java compiler that runs on the Mac

$$C^{Java \rightarrow Mac}_{Mac}$$

A Java compiler that generates Mac programs and runs on a Sun computer

$$C^{Java \rightarrow Mac}_{Sun}$$

A compiler that translates PC programs into Java and is written in Ada.

$$C^{PC \rightarrow Java}_{Ada}$$

The Phases of a Compiler

A compiler consists of at least three phases:

(1) Lexical analysis,
(2) Syntax analysis, and
(3) Code generation.

In addition, there could be other optimization phases employed to produce efficient object programs.

(4) Global optimization
(5) Local optimization

Sample Problem

Using the big C notation, show each of the following compilers:

a) An Ada compiler which runs on the PC and compiles to the PC machine language.
b) An Ada compiler which compiles to the PC machine language, but which is written in Ada.
c) An Ada compiler which compiles to the PC machine language, but runs on a Sun.

a) $C^{Ada \rightarrow PC}_{PC}$

b) $C^{Ada \rightarrow PC}_{Ada}$

C) $C^{Ada \rightarrow PC}_{Sun}$

Lexical Analysis

Words in the source program are converted to a sequence of tokens representing entities such as

(1) Key words - while, void, if, for, ...

(2) Identifiers - declared by the programmer

(3) Operators - +, -, ∗, /, =, ==, ...

(4) Numeric constants - numbers such as 124, 12.35, 0.09E-23, etc.

Lexical Analysis

Words in the source program are converted to a sequence of tokens representing entities such as

(6) Special characters - characters used as delimiters such as . ( ) , ; :

(7) Comments - ignored by subsequent phases. These must be identified by the scanner, but are not included in the output.

Syntax Analysis

Checks for syntax errors in the source program, using, as input, tokens put out by the lexical phase and producing, as output, a stream of atoms or syntax trees.

Atom
A record put out by the syntax analysis phase of a compiler which specifies a primitive operation and operands.

Example:
- MULT, ADD, and MOVE could represent atomic operations for multiplication, addition, and moving data in memory.
- Each operation could have 0 or more operands also listed in the atom: (operation, operand1, operand2, operand3).
- The meaning of the following atom would be to add A and B, and store the result into C:

```
(ADD,  A,  B,  C)
```

Sample Problem

Show the token classes, or "words", put out by the lexical analysis phase corresponding to this Java source input:

```
sum = sum + unit * /* accumulate sum */ 1.2e-12 ;
```

Solution:

```
identifier          (sum)
assignment          (=)
identifier          (sum)
operator            (+)
identifier          (unit)
operator            (*)
numeric constant    (1.2e-12)
```

Sample Problem 1.2 (b)
Show atoms corresponding to the following Java statement:

```
A = B + C * D ;
```

Solution:

```
(MULT,C,D,TEMP1)
(ADD,B,TEMP1,TEMP2)
(MOVE,TEMP2,A)
```

Sample Problem 1.2 (c)
Show atoms corresponding to the Java statement:

```
while (A<=B) A = A + 1;
```

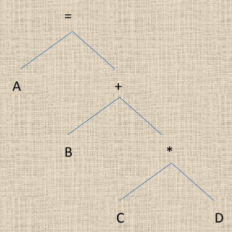
Solution:

```
(LBL, L1)
(TEST, A, <=, B, L2)
(JMP, L3)
(LBL, L2)
(ADD, A, 1, A)
(JMP, L1)
(LBL, L3)
```

Syntax tree

A tree data structure showing the structure of a source program or statement, in which the leaves represent operands, and the internal nodes represent operations or control structures.

Example
A Syntax Tree for `A = B + C * D`



Global Optimization

- ✓Improvement of intermediate code in space and/or time.
- ✓It involves examining the sequence of atoms put out by the parser to find redundant or unnecessary instructions or inefficient code.

```
stmt1
go to label1
stmt2
stmt3
label1: stmt4
```

```
for (i=1; i<=100000; i++)
{ x = Math.sqrt (y); // square root method
System.out.println (x+i);
}
x = Math.sqrt (y); // loop invariant
for (i=1; i<=100000; i++)
System.out.println (x+i);
```

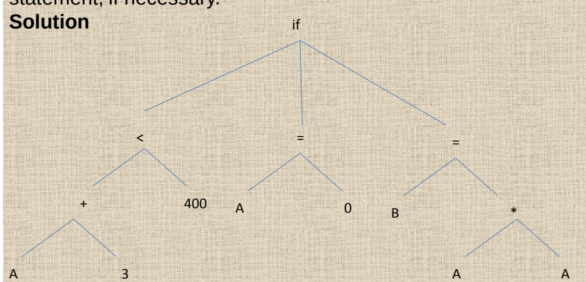
Sample Problem – Syntax tree

Show a syntax tree for the Java statement

`if (A+3<400) A = 0; else B = A*A;`

Assume that an if statement consists of three subtrees, one for the condition, one for the consequent statement, and one for the else statement, if necessary.

Solution



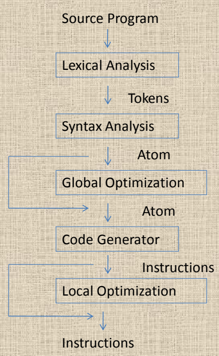
Local Optimization

- ➢Optimization applied to object code, usually by examining relatively small blocks of code.
- ➢It involves examining sequences of instructions put out by the code generator to find unnecessary or redundant instructions

```
LOD R1,A      // Load A into register 1
ADD R1,B      // Add B to register 1
STO R1,TEMP1  // Store the result in TEMP1*
LOD R1,TEMP1  // Load result into reg 1*
ADD R1,C      // Add C to register 1
STO R1,TEMP2   // Store the result in TEMP2

LOD R1,A      // Load A into register 1
ADD R1,B      // Add B to register 1
ADD R1,C /    / Add C to register 1
STO R1,TEMP   // Store the result in TEMP
```

The Phases of a Compiler



Sample Problem 1.2 (e)

Show assembly language instructions corresponding to the following atom string:

```
(ADD, A, B, Temp1)
(TEST, A, ==, B, L1)
(MOVE, Temp1, A)
(LBL, L1)
(MOVE, Temp1, B)
```

Solution:

```
    LOD R1,A
    ADD R1,B
    STO R1,Temp1    // ADD, A, B, Temp1
    CMP A,B
    BE L1           // TEST, A, ==, B, L1
    MOV A,Temp1     // MOVE, Temp1, A
L1: MOV B,Temp1     // MOVE, Temp1, B
```

Code Generation

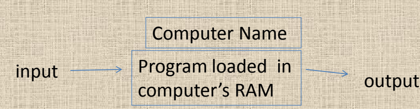
Produces machine language object code from syntax trees or atoms.

Example

An ADD atom might be translated to three machine language instructions:
(1) load the first operand into a register,
(2) add the second operand to that register, and
(3) store the result, as shown for the atom (ADD, A, B, Temp):

```
LOD R1,A        // Load A into reg. 1
ADD R1,B        // Add B to reg. 1
STO R1,Temp     // Store reg. 1 in Temp
```

Implementation Techniques



Notation for a Program Running on a Computer

1. The program loaded into the computer's memory will be a compiler.
2. A computer is capable of running only programs written in the machine language of that computer.

M

$C_S^{X\to Y} \longrightarrow \boxed{C_M^{S\to O}} \longrightarrow C_O^{X\to Y}$

Notation for a compiler being translated to a different language

3. The subscript language (the language in which it exists) of the executing compiler (the one inside the computer), M, must be the machine language of the computer on which it is running.
4. The subscript language of the input, S, must be the same as the source language of the executing compiler.
5. The subscript language of the output, O, must be the same as the object language of the executing compiler.

Bootstrapping

The process of using a program as input to itself – as in compiler development – through a series of increasingly larger subsets of the source language.

Bootstrapping a compiler

$C_{Sun}^{Java\to Sun}$ We want this compiler

$C_{Sun}^{Sub\to Sun}$ $C_{Sub}^{Java\to Sun}$ We write these two small compilers

Sun

$C_{Sub}^{Java\to Sun} \longrightarrow \boxed{C_{Sun}^{Sub\to Sun}} \longrightarrow C_{Sun}^{Java\to Sun}$

Bootstrapping Java onto a Sun Computer

Sample Problem

Show the output of the following compilation using the big C notation.

Sun

$C_{Ada}^{Ada\to PC} \longrightarrow \boxed{C_{Sun}^{Ada\to Sun}} \longrightarrow$?

Solution

Sun

$C_{Ada}^{Ada\to PC} \longrightarrow \boxed{C_{Sun}^{Ada\to Sun}} \longrightarrow C_{Sun}^{Ada\to PC}$

Cross compiling

The process of generating a compiler for a new computer architecture, automatically.

We want this compiler $C_{Mac}^{Java\to Mac}$

We write this compiler $C_{Java}^{Java\to Mac}$

We already have this compiler $C_{Sun}^{Java\to Sun}$

Sun

$C_{Java}^{Java\to Mac} \longrightarrow \boxed{C_{Sun}^{Java\to Sun}} \longrightarrow C_{Sun}^{Java\to Mac}$

Sun

$C_{Java}^{Java\to Mac} \longrightarrow \boxed{C_{Sun}^{Java\to Mac}} \longrightarrow C_{Mac}^{Java\to Mac}$

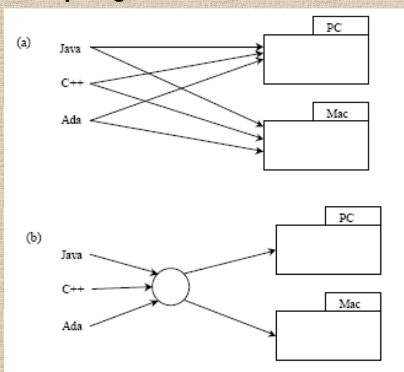
Compiling To Intermediate Form

- ✓ It is possible to compile to an intermediate form, which is a language somewhere between the source high-level language and machine language.
- ✓ The stream of atoms put out by the parser is a possible example of an intermediate form.
- ✓ The primary advantage of this method is that one needs only one translator for each high-level language to the intermediate form (each of these is called a front end) and only one translator (or interpreter) for the intermediate form on each computer (each of these is called a back end)

Decaf

```
/*cos(x) = 1 - x2/2 + x4/24 - x6/720 + ...*/
class Cosine
{
  public static void main (String [] args)
  {
    float cos, x, n, term, eps, alt;
   /* compute the cosine of x to within tolerance eps */
   /* use an alternating series */
    x = 3.14159;
    eps = 0.0001;
    n = 1;
    cos = 1;
    term = 1;
    alt = -1;
    while (term>eps)
    {
      term = term * x * x / n / (n+1);
      cos = cos + alt * term;
      alt = -alt;
      n = n + 2;
    }
  }
}
```

Compiling To Intermediate Form




```
Program → class identifier { public static
          void main (String[] identifier)
          CompoundStmt }
Declaration → Type IdentList ;
Type →  int
      | float
IdentList → identifier , IdentList
           | identifier
Stmt → AssignStmt
      | ForStmt
      | WhileStmt
      | IfStmt
      | CompoundStmt
      | Declaration
      | NullStmt
A→ssignStmt → AssignExpr ;
ForStmt → for ( OptAssignExpr;
                 OptBoolExpr ;
               OptAssignExpr ) Stmt
OptAssignExpr → AssignExpr
               | ε
OptBoolExpr → BoolExpr
              | ε
WhileStmt → while ( BoolExpr ) Stmt
IfStmt → if ( BoolExpr ) Stmt ElsePart
ElsePart → else Stmt
           | ε

CompoundStmt → { StmtList }
StmtList → StmtList Stmt
            | ε
NullStmt  → ;
BoolExpr → Expr Compare Expr
Compare → == | < | > | <= | >= | !=
Expr → AssignExpr
      | Rvalue
AssignExpr → identifier = Expr
Rvalue → Rvalue + Term
        | Rvalue - Term
        | Term
Term → Term * Factor
      | Term / Factor
      | Factor
Factor → ( Expr )
        | - Factor
        | + Factor
        | identifier
        | number
```