Menoufia University

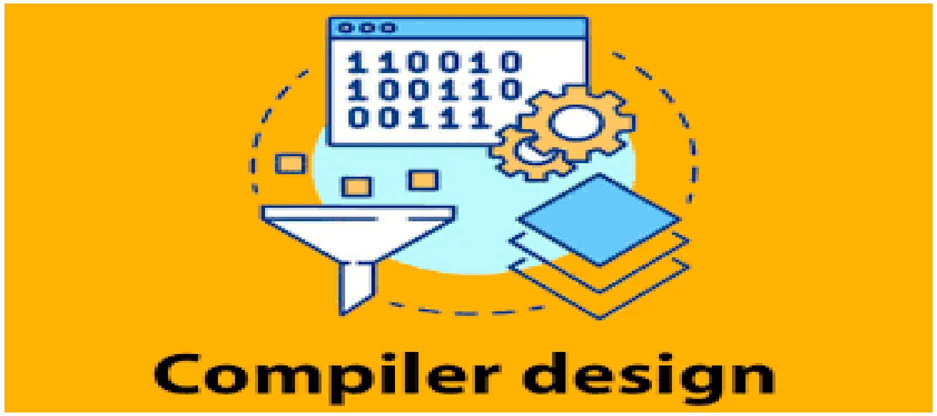
Faculty of computers & Information

**Computer Science Department.**

# Compiler Design

## 4 Year – first Semester

## Lecture 1



**DR. Eman Meslhy Mohamed**

**Lecturer at Computer Science department**

**2023-2024**

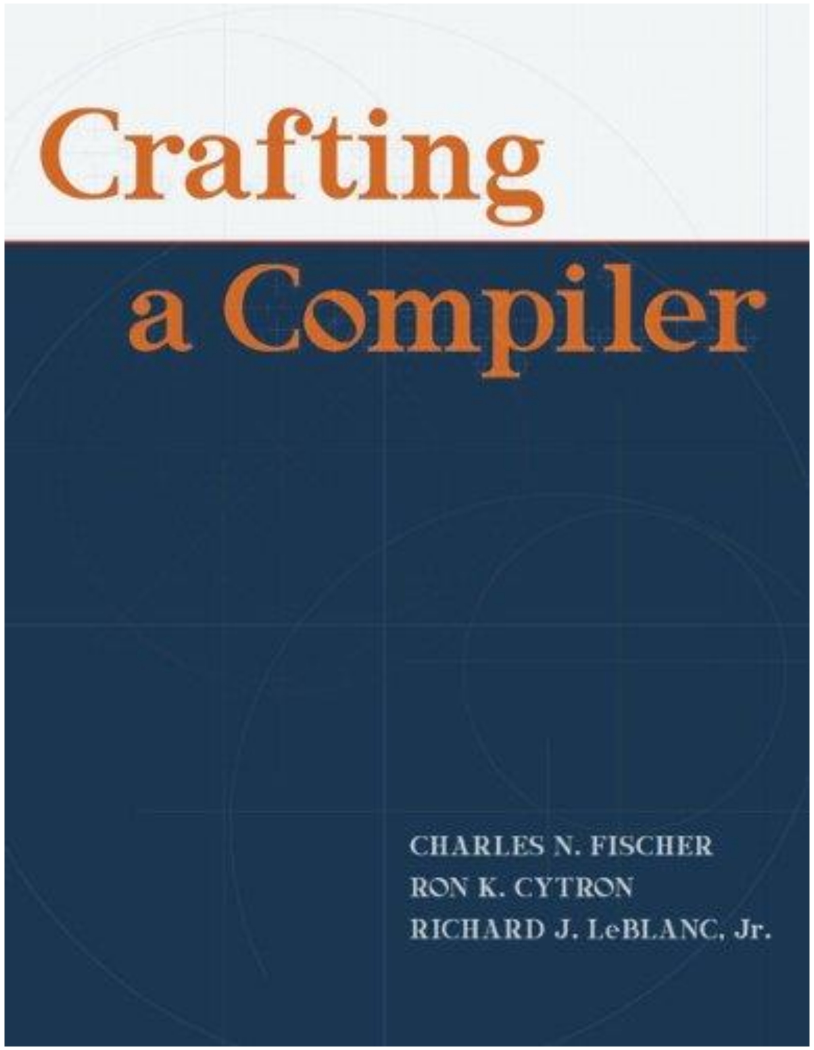
## Basic Course Information

- Course Name: **Compiler Design**
- Course Code: **CS471.**
- Level: **1st Semester, 4th Year/ B.Sc.**
- Course Credit: **3 credits**
- Instructor:
    - **Dr. Eman Meslhy**

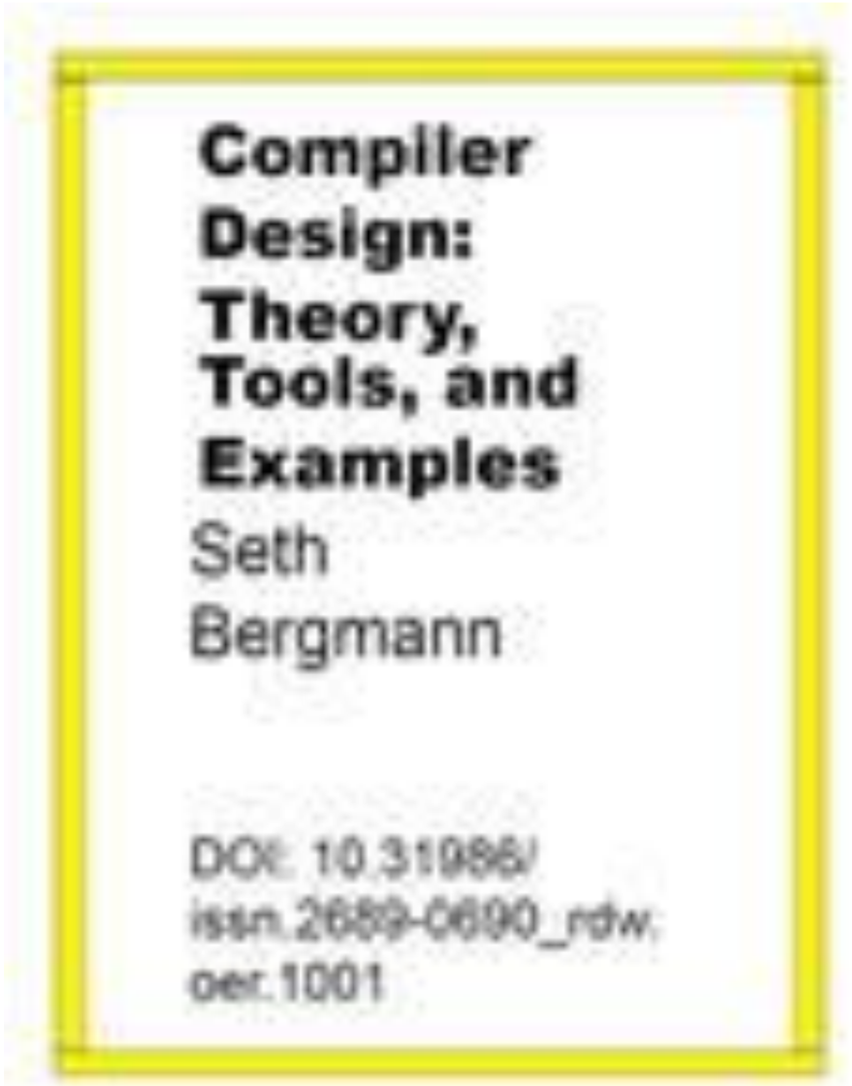
# Basic Course Information

| Course Assessment Tools | Degree |
|---|---|
| Mid-Term | 15 |
| Assignments & Section Work | Project ➔ 10<br>Section➔ 5<br>Sheets 10 ➔ on teams' group<br>Quiz 5 (bouns) ➔ on teams' group |
| Final | 60 |

## Selected References


Crafting a Compiler

CHARLES N. FISCHER
RON K. CYTRON
RICHARD J. LeBLANC, Jr.


Compiler Design: Theory, Tools, and Examples

Seth Bergmann

DOI: 10.31986/issn.2689-0690_rdw.oer.1001

## Course Objectives

- Able to convert any instruction of a program from source language to target language.
- Recognize what happens at each and every phase of a compiler.
- Specify and analyze the lexical, syntactic and semantic structures of advanced language features.
- Understand the different types of parsing techniques.
- Understand the concepts related to compilers and should be able to deploy their knowledge in various related fields.

## Course Contents

- Introduction
- Lexical Analysis
- Syntax Analysis
- Top-Down Parsing
- Bottom-Up Parsing
- Code Generation
- Optimization

## Introduction

## Outline

- **Introduction to Compiler.**
- Phases of Compilers.
- Implementation Techniques.

# Introduction to Compiler

## 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

**Compiler**

on in memory to another,

The function of the compiler is to accept statements such as those above and translate them into sequences of machine language operations

**if (array6[loc]<MAX) sum = 0; else array6[loc] = 0;**

## What is a Compiler?

- **A compiler** is a computer program that translates the code written in one language (high-level language) into an into **equivalent** other language (low-level or machine language).
- 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**.

# Introduction to Compiler

## Example

**Source Program**

```
a = b + c * d;
```

↓

**Compiler**

↓

**Object Program**

```
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
MOV a,temp2         // Move temp2 to a, the final result
```

# Quiz 1

**a = (b+c) * (c-d);**

- ld r1,b
- add r1,c
- ld r2,c
- sub r2,d
- mr r1,r2
- sto r1,a

- ld r1,b
- add r1,c
- Sto r1, Temp1
- ld r1,c
- sub r1,d
- mr r1,Temp1
- sto r1,a

# Introduction to Compiler

## Example

- 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;**

```
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,x          // x = 2 * x
    B     L1            // Repeat loop
L3:
```

## Quiz 2

**for (i=1; i<=10; i++) a = a+i;**

- ld r1,1
- ld r2,a
- loop:
- cmp r1,='10'
- brh Done
- ar r2,r1
- incr r1
- jmp loop
- Done:

- ld r1,="10"
- Sto r1, Temp1
- Mov i="1"
- Lod r1, a
- loop:
- cmp i,Temp1
- brh Done
- add r1,i
- incr i
- Sto r1, a
- jmp loop
- Done:

# *Introduction to Compiler*

## High-Level Languages over Machine or Assembly Language

1. Machine language (and even assembly language) is **difficult to work** with and **difficult to maintain**.

2. With a high-level language you have a much greater degree of **machine independence** and **portability** from one kind of computer to another.

3. In High-level languages, the programmer **does not have complete control of the machine's resources** (registers, interrupts, I/O buffers).

4. The compiler may generate **inefficient machine** language programs.

5. **Additional software**, the compiler, is needed in order to use a high-level language.

## Compiler & Interpreter

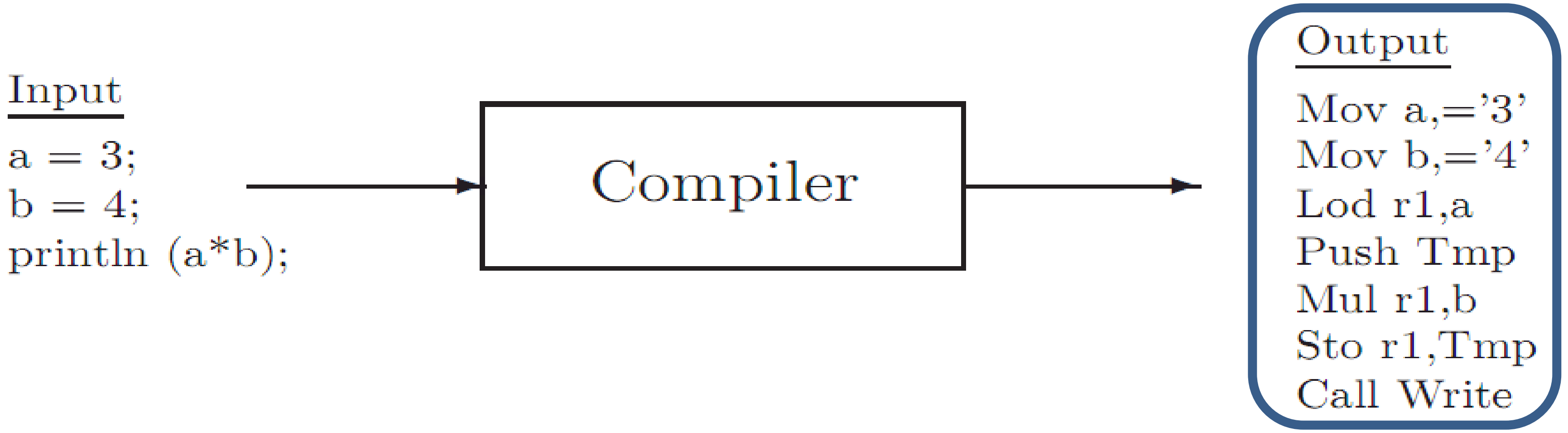
- **Interpreter** – is a computer program that interprets and executes the code written in one language (high-level language) **one by one**.

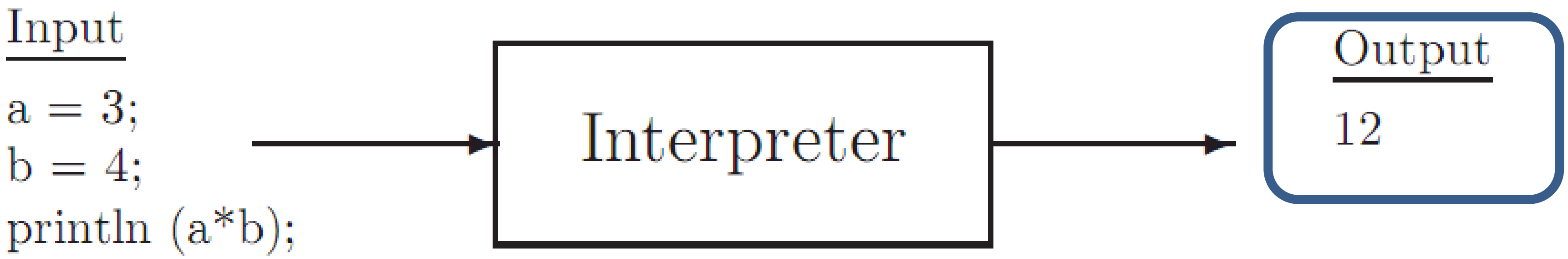
- **A compiler** is a computer program that **completely** translates the code written in one language (high-level language) into an into **equivalent** other language (low-level or machine language), which then runs on the computer hardware.

# Introduction to Compiler

## Example

```
a = 3;
b = 4;
println (a*b);
```





## Example

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

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

*Compiler Output*

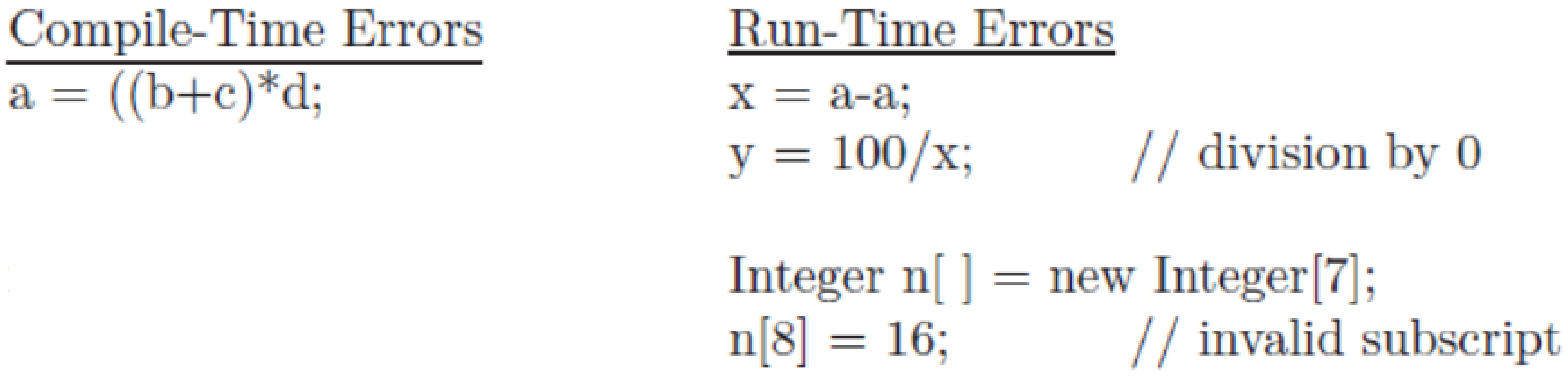
```
      LOD    r1,='4'
      STO    r1,Temp1
      MOV    i,='1'
L1:   CMP    i,temp1
      BH     L2
      LOD    r1,i
      MUL    r1,='3'
      STO    r1,Temp2
      PUSH   Temp2
      CALL   Print
      B      L1
L2:
```

*Interpreter Output*

```
3 6 9 12
```

## Compile Time & Run Time

- **Compile time**: The time at which a source program is compiled.
- **Run time**: The time at which the resulting object program is loaded and executed.



```
Compile-Time Errors          Run-Time Errors
a = ((b+c)*d;                x = a-a;
                             y = 100/x;          // division by 0

                             Integer n[ ] = new Integer[7];
                             n[8] = 16;          // invalid subscript
```

# Quiz 3

```
(a)   a = b+c = 3;
                    → Compile time error
(b) if (x<3)
            a = 2
         else a = x;
                    → Compile time error
(c) if (a>0) x = 20;
      else if (a<0) x = 10;
      else x = x/a;
                         → Run time error
(d) MyClass x [] = new MyClass[100];
x[100] = new MyClass;
                            →Run time error
```

## Compiler Language -- Big C notation

- It is important to remember that a **compiler is a program**, and it must be written in some language (machine, assembly, high-level).
- In describing this program, 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.
    3. The language in which the compiler is written, or the language in which it exists.

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

**A Java compiler for the Mac.**

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

**A compiler which translates Java programs to Mac machine language, and which runs on a Sun machine.**

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

**A compiler which translates PC machine language programs to Java, written in Ada**

## Example

**Using the big C notation to show each of the following compilers:**

1. An Ada compiler which runs on the PC and compiles to the PC machine language.
2. An Ada compiler which compiles to the PC machine language, but which is written in Ada.
3. An Ada compiler which compiles to the PC machine language, but which runs on a Sun.

(1) $C^{Ada \rightarrow PC}_{PC}$

(2) $C^{Ada \rightarrow PC}_{Ada}$

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

## Quiz 3

- A compiler which translates COBOL source programs to PC machine language and runs on a PC

- A compiler, written in Java, which translates Sun machine language programs to Java.

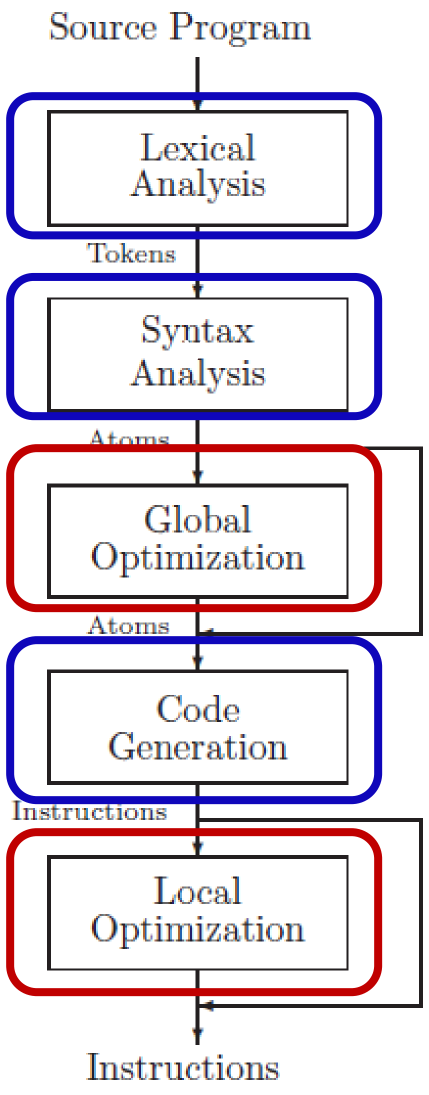
## ***Outline***

- Introduction to Compiler.
- **Phases of Compilers.**
- Implementation Techniques.

# Phases of Compilers

## Phases of Compilers

- Lexical Analysis (Scanner)
- Syntax Analysis Phase
- Global Optimization
- Code Generation
- Local Optimization



# Phases of Compilers

## 1. Lexical Analysis (Scanner)

- The first phase of a compiler is called lexical analysis which generate **a stream of tokens** as output from source program.
- Also, it build a **symbol table** to store all identifiers used in the source program.

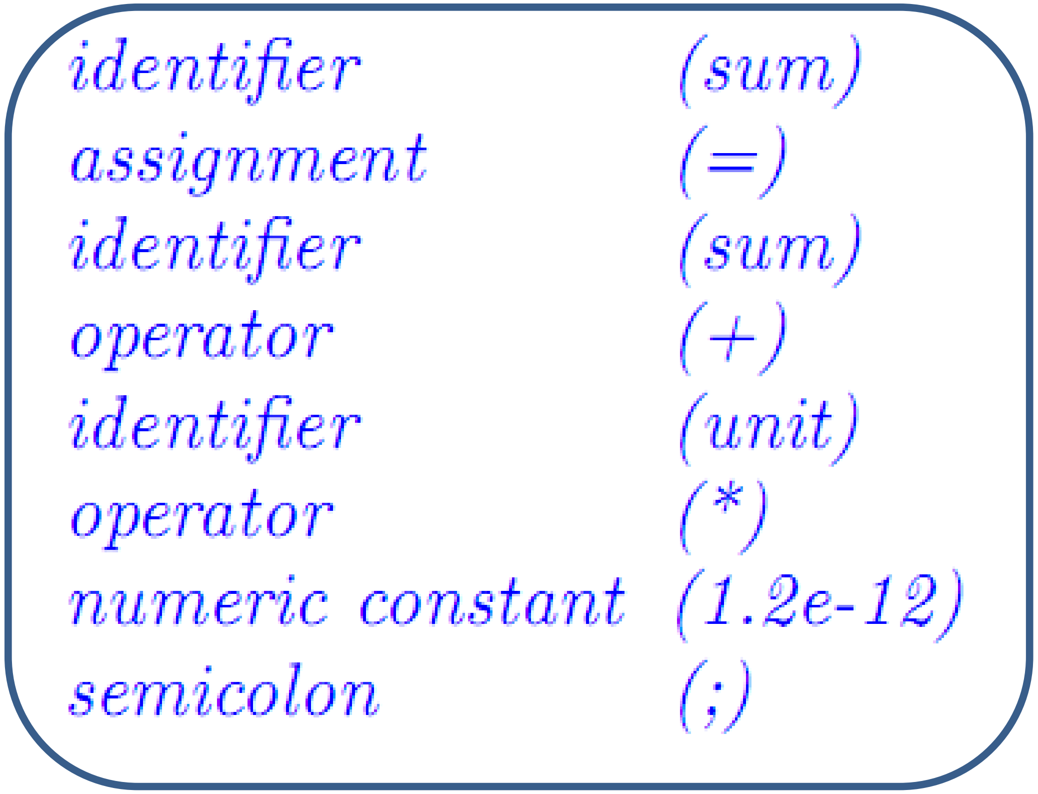
  - **key words** - while, void, if, for, ...
  - **identifiers** - declared by the programmer
  - **operators** - +, -, *, /, =, ==, ...
  - **numeric constants** - numbers such as 124, 12.35, 0.09E-23, etc.
  - **character constants** - single characters or strings of characters
  - **special characters** - characters used as delimiters such as . ( ) , ; :
  - **comments** - ignored by subsequent phases. These must be identified by the scanner, but are not included in the output.

# Phases of Compilers

## Example

- 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 ;**

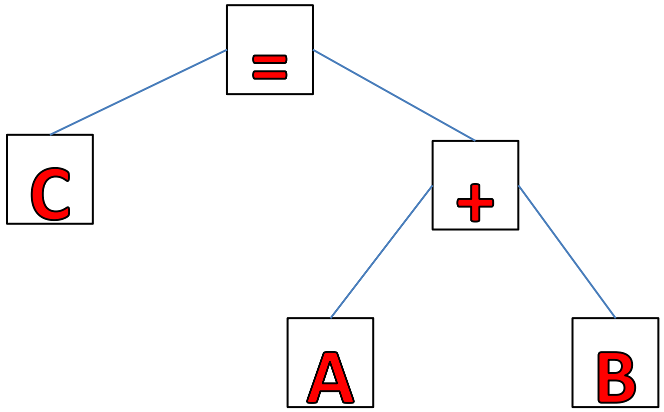


| | |
|---|---|
| *identifier* | *(sum)* |
| *assignment* | *(=)* |
| *identifier* | *(sum)* |
| *operator* | *(+)* |
| *identifier* | *(unit)* |
| *operator* | *(*)* |
| *numeric constant* | *(1.2e-12)* |
| *semicolon* | *(;)* |

# Phases of Compilers

## 2. Syntax Analysis

- The syntax analysis phase is often called the **parser**.
- The **parser** will check for the program **syntax error** and determine the underlying structure of the source program.
- The output of this phase may be a **stream of atoms** or a collection of **syntax trees**.
- For example, **(ADD, A, B, C):** The meaning of the following atom would be to add A and B, and store the result into C.



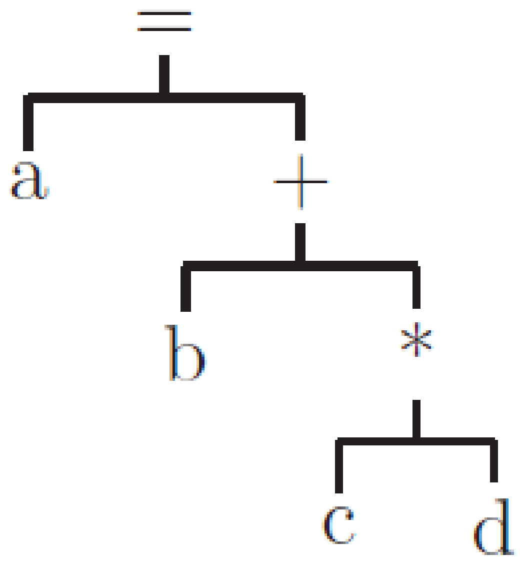
# Phases of Compilers

## Example

- Show the atoms and syntax tree corresponding to the following Java statement:

**a = b + c * d ;**

```
(MULT, c, d, temp1)
(ADD, b, temp1, temp2)
(MOVE, temp2, a)
```

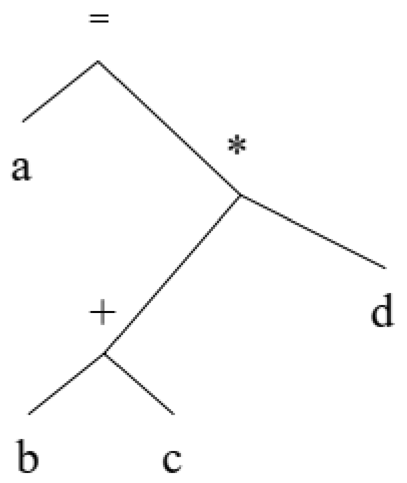


**Atoms**

**Syntax Tress**

```
a  =  (b+c)  *  d;
```

(ADD, b, c, T1)
(MUL, T1, d, T2)
(MOV, T2, , a)



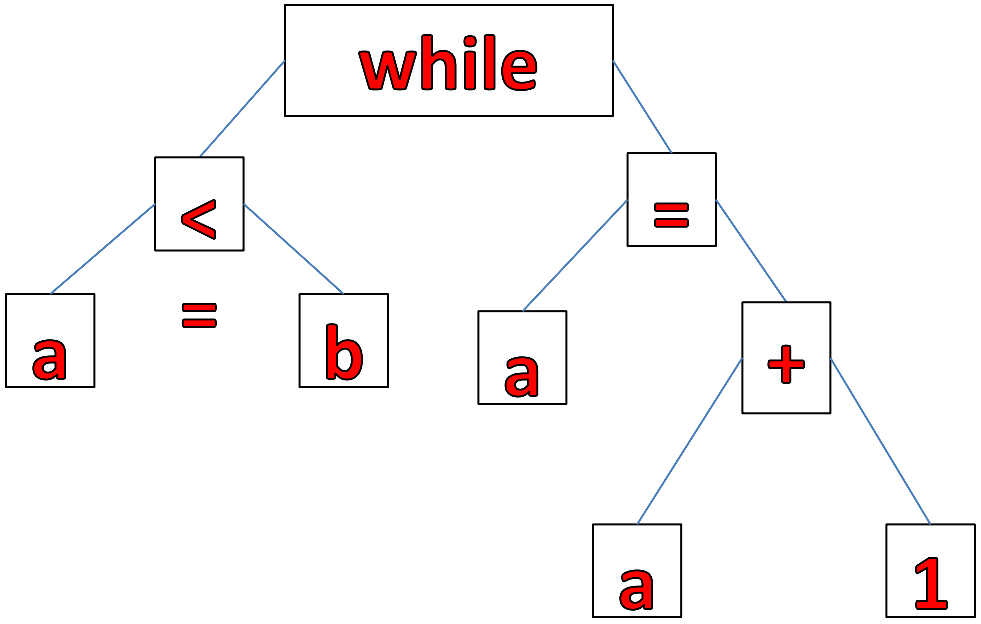
# Phases of Compilers

## Example

- Show the atoms and syntax tree corresponding to the following Java statement:

**while (a <= b) a = a + 1;**

```
(LBL, L1)
(Test, a, <=, b, L2)
(JMP, L3)
(LBL, L2)
(ADD, a, 1, a)
(JMP, L1)
(LBL, L3)
```



**Atoms**

**Syntax Tress**

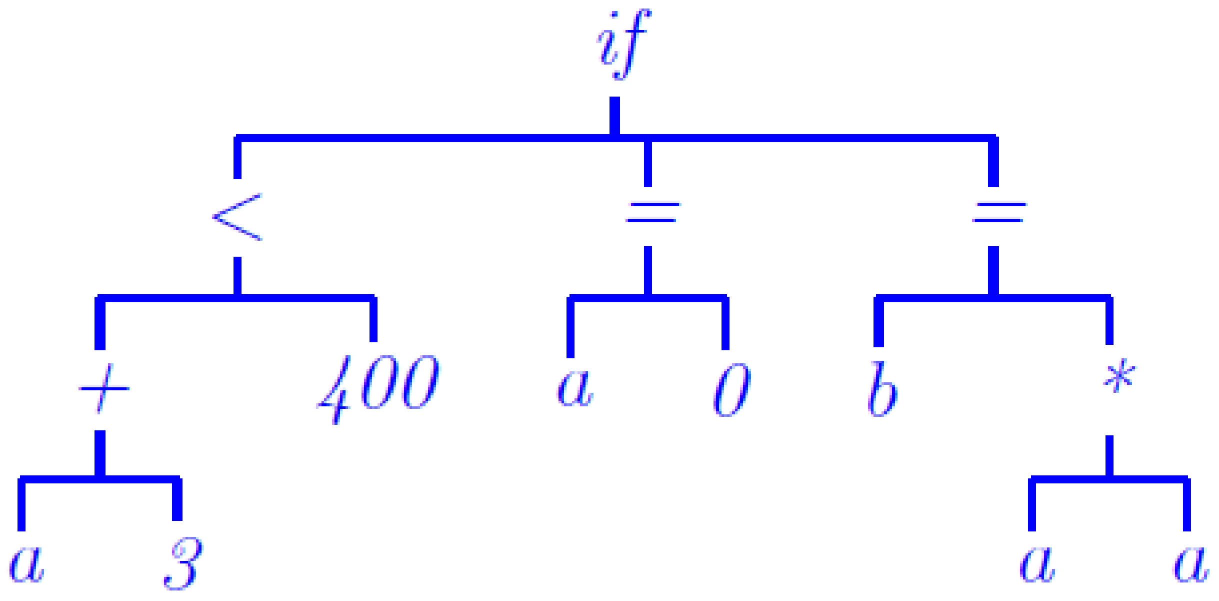
# Phases of Compilers

## Example

- Show the syntax tree corresponding to the following Java statement:

**if (a+3 < 400) a =0; else b = a*a;**

**Note:** In syntax trees, each **interior node** represents an **operation or control structure** and each **leaf node** represents an **operand**.



## *Phases of Compilers*

### 3. Global Optimization

- The global optimization phase is optional. Its purpose is simply to make the object program **more efficient** in **space** and/or **time**.
- Examining the sequence of atoms put out by the parser to find **redundant** or **unnecessary** instructions or inefficient **code**.
- Since it is invoked before the code generator, this phase is often called **machine-independent** optimization.

# Phases of Compilers

## Example

- **Examples:**

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

**stmt2 and stmt3 can never be executed.**

```
y=16;
for (i=1; i<=100000; i++)
{ x = Math.sqrt (y); // square root method
System.out.println (x+i);
}
```

**the assignment to x need not be inside the loop since y does not change as the loop repeats**

$\Rightarrow$

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

**This would eliminate 99,999 unnecessary calls to the sqrt method at run time.**

- The reader is cautioned that global optimization can have a serious impact on run-time debugging.
- Most compilers can **turn optimization** on or off through debugging.

```
(a)    for  (i=1;  i<=10;  i++)
              { x = i + x;
                a[i] = a[i-1];
                y = b * 4;
              }

for  (i=1;  i<=10;  i++)
              { x = i + x;
                a[i] = a[i-1];
              }
          y = b * 4;
```

# *Phases of Compilers*

## 4. Code Generation

- Java compilers produce an intermediate form, known as **byte code**, which can be interpreted by the Java run-time environment.
- But, in this course, we will be **assuming** that our compiler is to produce native code for a particular machine.
- For 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.
  3. Store the result.

**(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
```

# Phases of Compilers

## Example

- Show assembly language instructions corresponding to the following atom string:

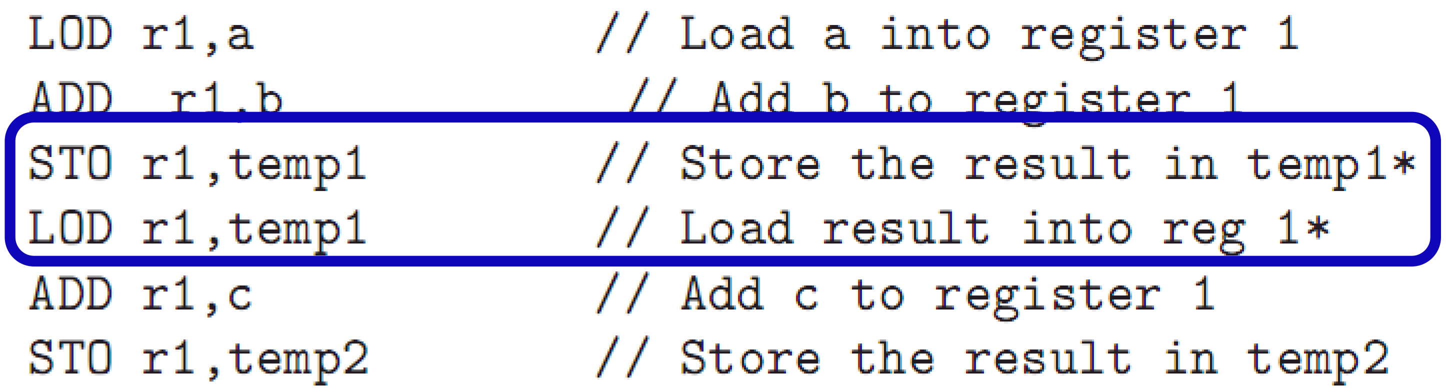
```
(ADD, a, b, temp1)
(TEST, a, ==, b, L1)
(MOVE, temp1, a)
(LBL, L1)
(MOVE, temp1, b)
```

```
       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
```

# Phases of Compilers

## 5. Local Optimization

- The local optimization phase is also **optional**.
- It involves examining sequences of instructions put out by the code generator to find **unnecessary** or **redundant instructions**.
- For this reason, local optimization is often called **machine-dependent** optimization.
- For Example, the expression **a + b + c** in the source program might result in the following instructions as code generator output:



```
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
```

```
LD   R1,A
MULT R1,B
ST   R1,Temp1
LD   R1,Temp1
ADD  R1,C
ST   R1,Temp2
```

```
LD   R1,A
MULT R1,B
ADD  R1,C
ST   R1,Temp2
```

## Assignments across Course

**After the completed each phase in our lecture, each group of students (2-3 students) should implement it regrading one of the available programming languages.**

THANKS

for your attention