COMPILER DESIGN

1. INTRODUCTION

Instructor: Sushil Nepal, Assistant Professor

Block: 9-308

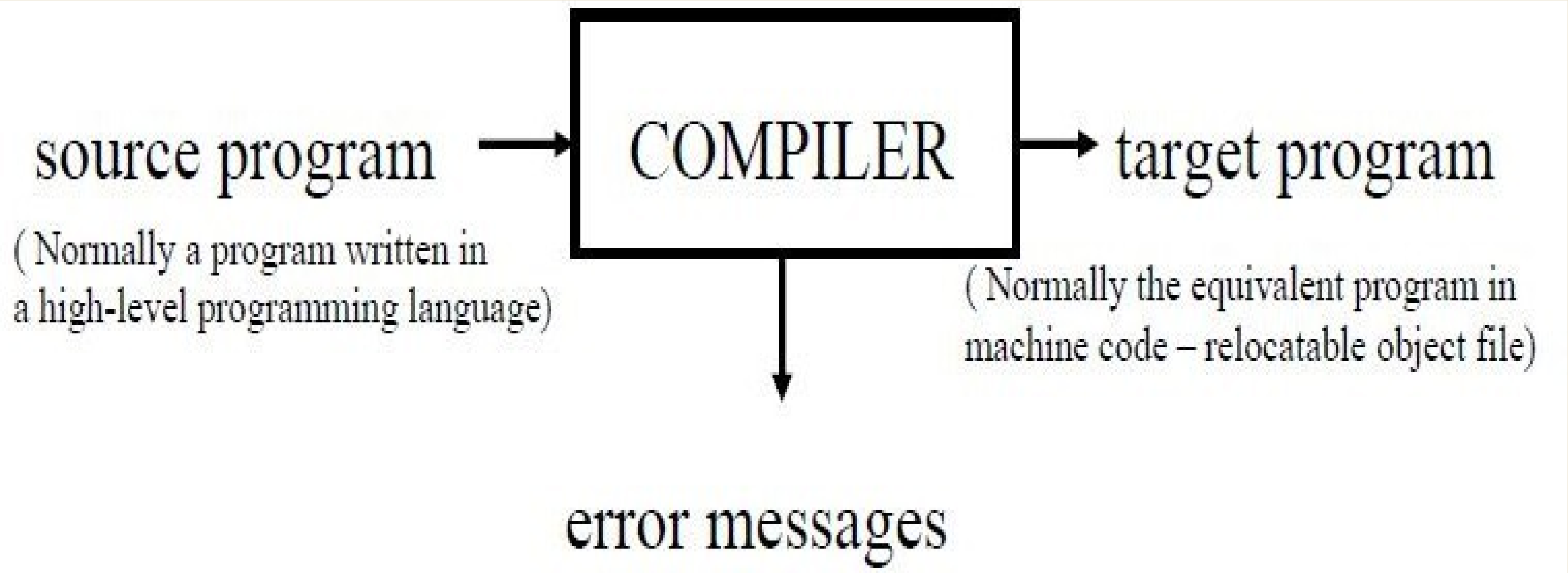
Email: sushilnepal@ku.edu.np

Contact: 9851-151617

What is a compiler?

Compiler

- A program that takes as input a program written in one language (source language) and translates it into a functionally equivalent program in another language (target language)
- Source – Usually high level languages like C, C++, Java
- Target – Low level languages like assembly or machine code
- During translation – Also reports errors and warnings to help the programmer


source program
COMPILER
target program
(Normally a program written in a high-level programming language)
(Normally the equivalent program in machine code – relocatable object file)
error messages

History of Compiler's Development

- Towards the end of 1950s, machine-independent programming languages were first proposed. Subsequently several experimental compilers were developed.
- The first compiler was written by ***Grace Hopper***, in 1952, for A-0 programming language.
- The FORTRAN team led by ***John Backus*** at IBM is considered first complete compiler in 1957.
- COBOL was an early language to be compiled on multiple architecture in 1960.
- Early compilers were written in assembly language.
- First self-hosting compiler capable of compiling its own source code in HLL was created for Lisp by ***Tim Hart*** and ***Mike Levin*** at MIT in 1962.
- Since 1970's compilers were developed for its own language such as Pascal and C.

Application Areas

- The techniques used in compiler design can be applicable to many problems in computer science
- Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs
- Techniques used in a parser can be used in a query processing system such as SQL

Application Areas

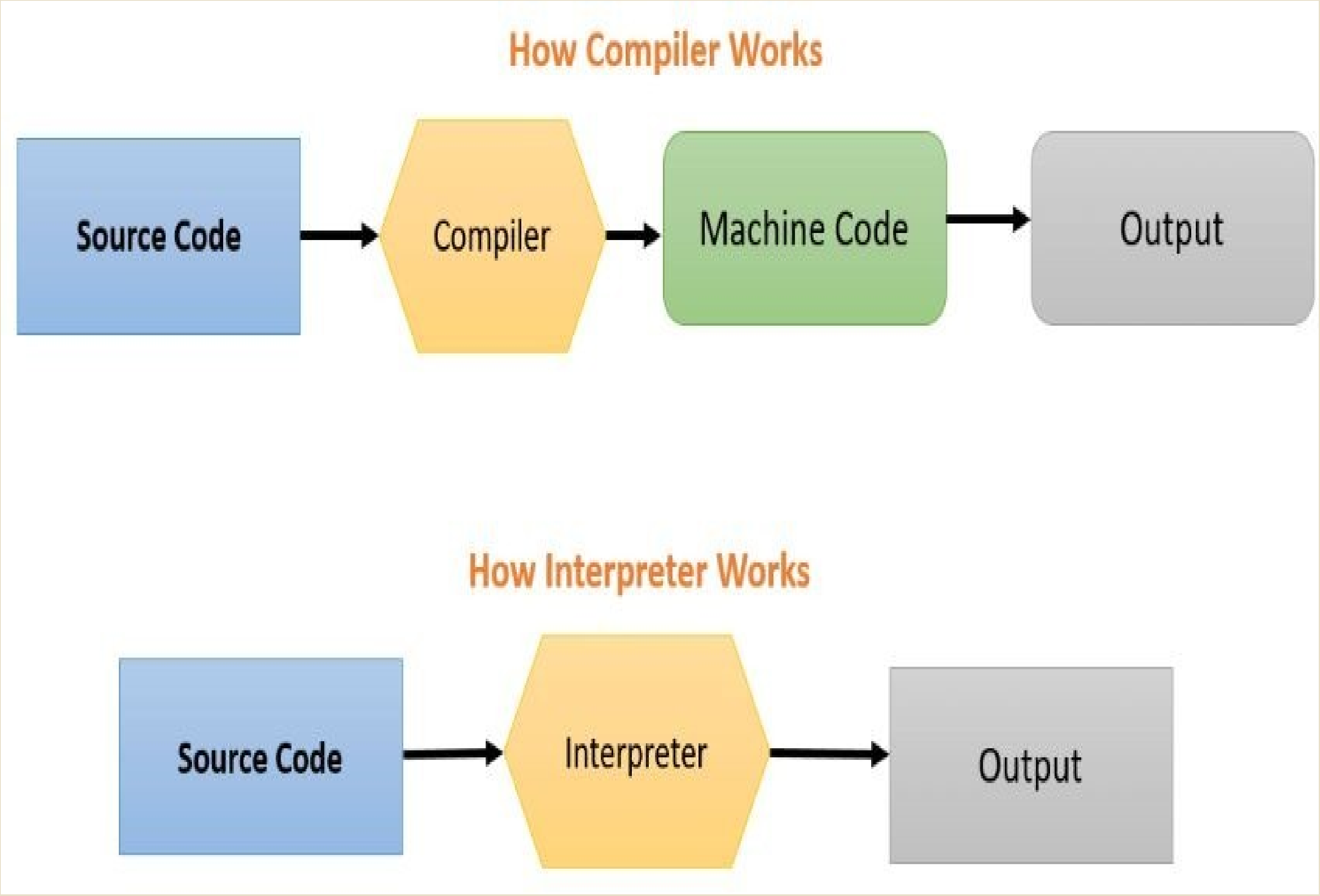
- Many software having a complex front-end may need techniques used in compiler design
  - Eg: A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
- Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems

Interpreter

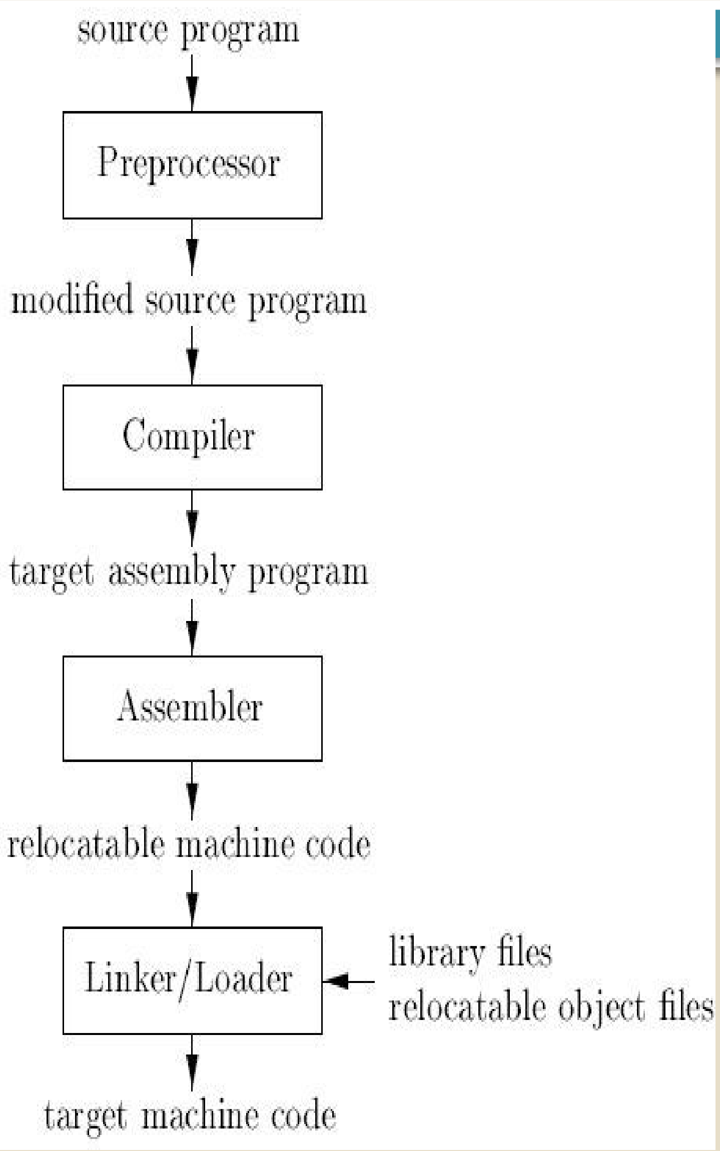
- An interpreter is another common kind of language processor
- Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user
- Both compiler and interpreters do the same job which is converting higher level programming language to machine code

Interpreter

- Compiler transforms code written in a HLL into the machine code, at once, before program runs
- Interpreter converts each HLL program statement, one by one, into the machine code, during run time
- Compiled code runs faster while interpreted code runs slower
- Compiler displays all errors after compilation, on the other hand, the Interpreter displays errors of each line one by one


How Compiler Works
Source Code
Compiler
Machine Code
Output
How Interpreter Works
Source Code
Interpreter
Output

A language processing system



Cousins of Compiler

1. Preprocessor

- Programs which translate source code into simpler or slightly lower level source code, for compilation by another compiler
- Performs a pre-compilation of the source program to expand any macro definitions
- A preprocessor may allow a user to define macros that are short hands for longer constructs
- A preprocessor may include header files into the program context.

Eg of a macro in C:

```
#define square(x) ((x)*(x))  //Function like micros
#define PI 3.14 //Object like micros
```

Cousins of Compiler

2. Assembler

- Programs written to automate the translation of assembly language into machine language
- An assembly language is one where mnemonics are used
- Mnemonics are symbols used for each machine instruction which make it easier to write/read programs compared to those written in machine language
- Mnemonics are subsequently translated into machine language

Cousins of Compiler

3. Linker/Loader

- If the target program is machine code, loaders are used to load the target code into memory for execution
- Linkers are used to link target program with the libraries

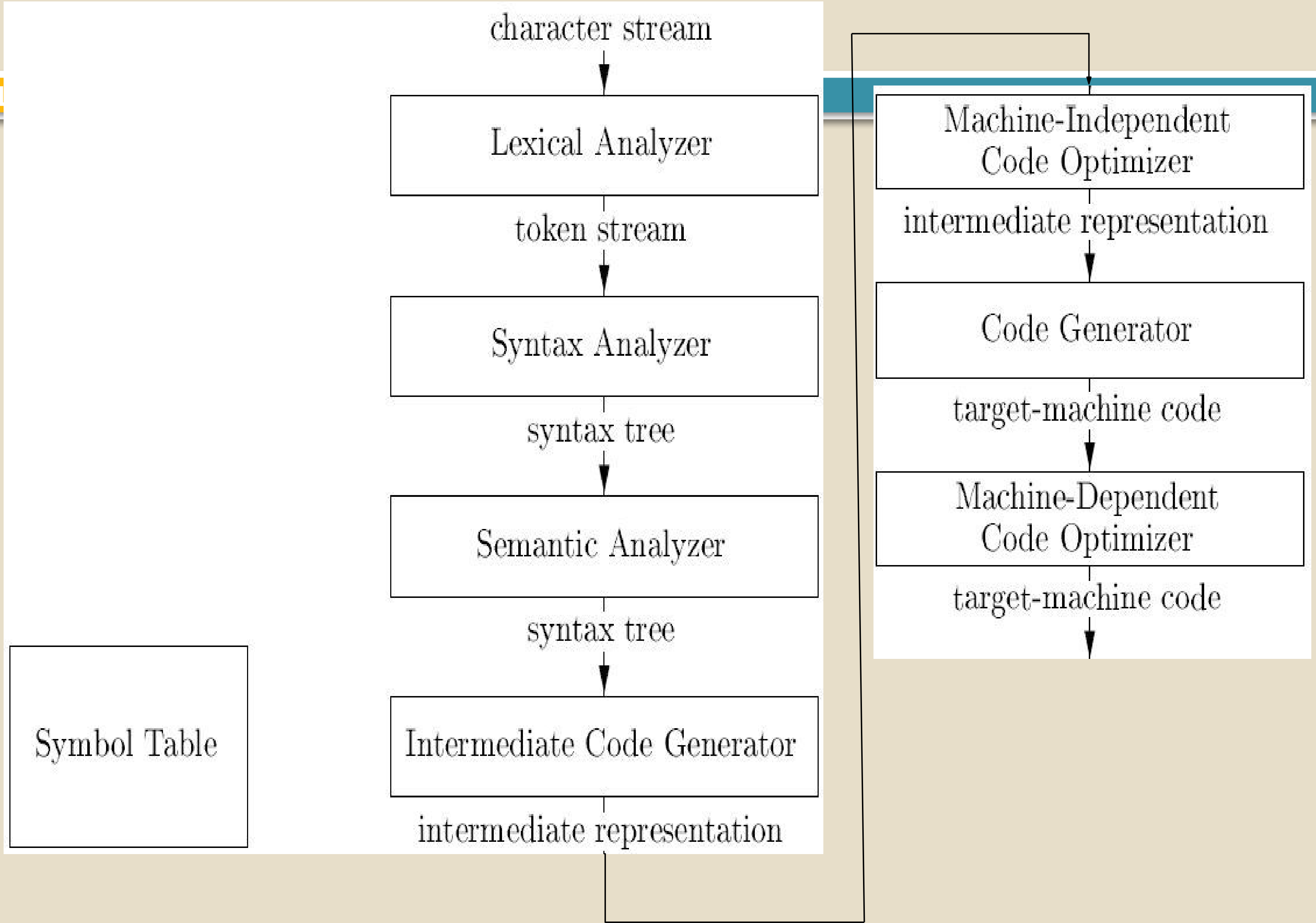
Phases of Compilation

- Two main phases of compiling process
  - Analysis
  - Synthesis
- **Analysis**
  - Breaks up the source program into pieces and creates a language independent intermediate representation(IR) of program
  - The analysis part also collects information about the source program and stores it in a data structure called a ‘**symbol table**’, which is passed along with the intermediate representation to the synthesis part

Phases of Compilation

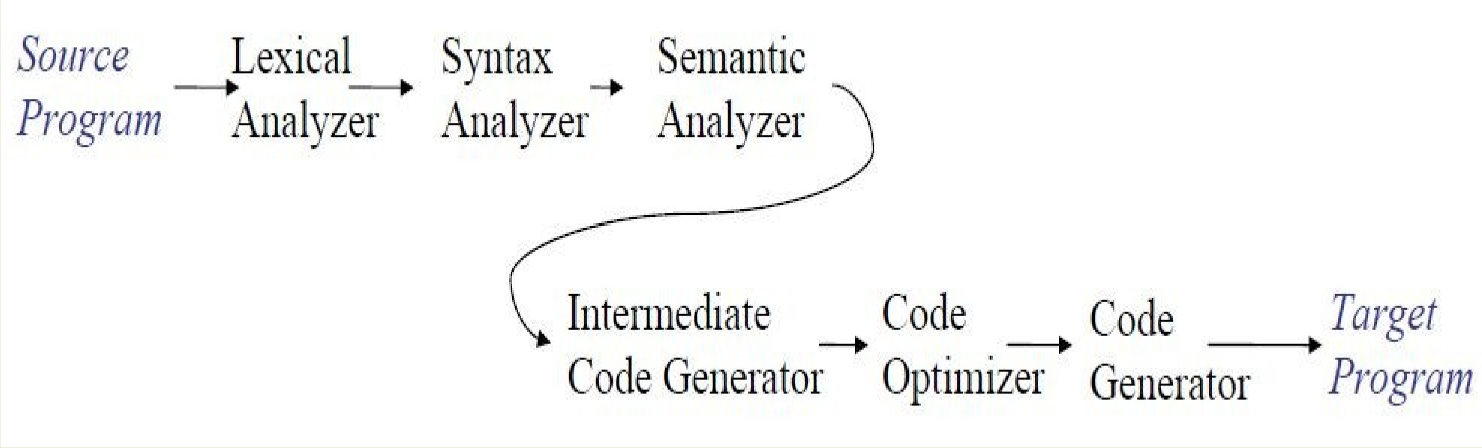
- **Synthesis**
  - Constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called as the front end of the compiler whereas the synthesis part is called as the back end.

Phases of Compiler



Phases of Compiler

- Each phase transforms the source program from one representation into another representation
- They communicate with **error handlers** and **symbol table**



Phases of Compiler

- **Analysis**
  - Lexical Analysis
  - Syntax Analysis
  - Semantic Analysis

- **Synthesis**
  - Intermediate Code Generation
  - Code Optimization
  - Code Generation

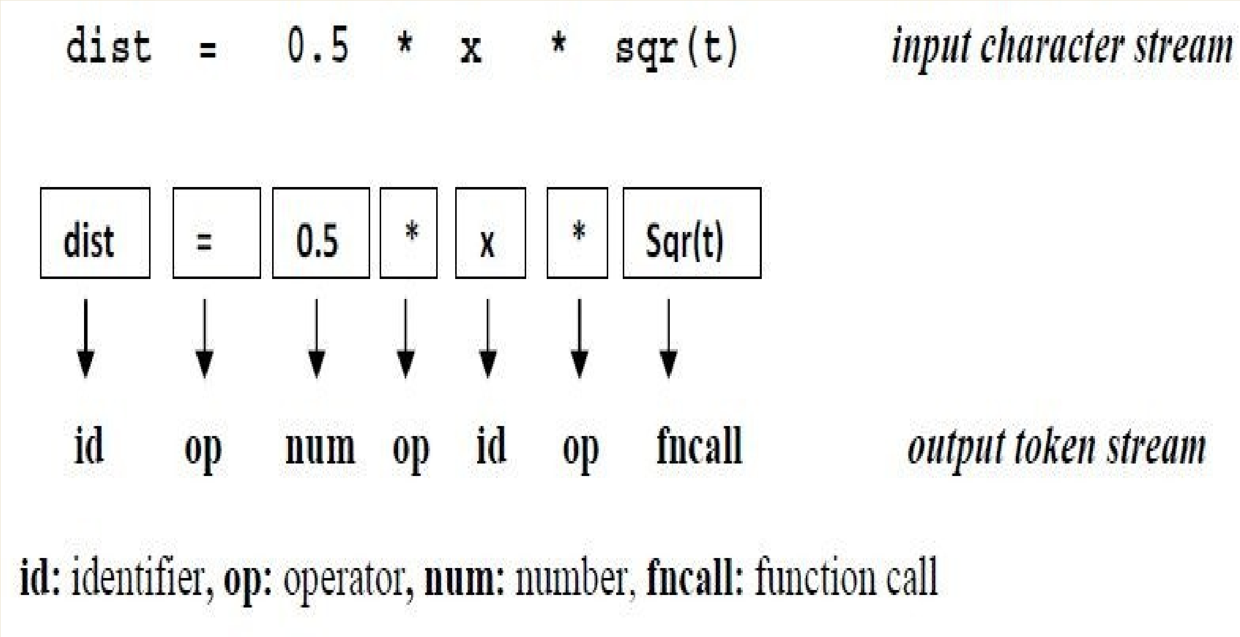
Lexical Analysis (Scanning)

- The stream of characters making the source program is read from left to right and grouped into **tokens**
- **Tokens** are sequence of characters that have a collective meaning
- Examples of tokens are identifiers, reserved words, operators, special symbols, etc

Lexical Analysis (Scanning)

newval := oldval + 12	=>	tokens:	newval	identifier
			:=	assignment operator
			oldval	identifier
			+	add operator
			12	a number

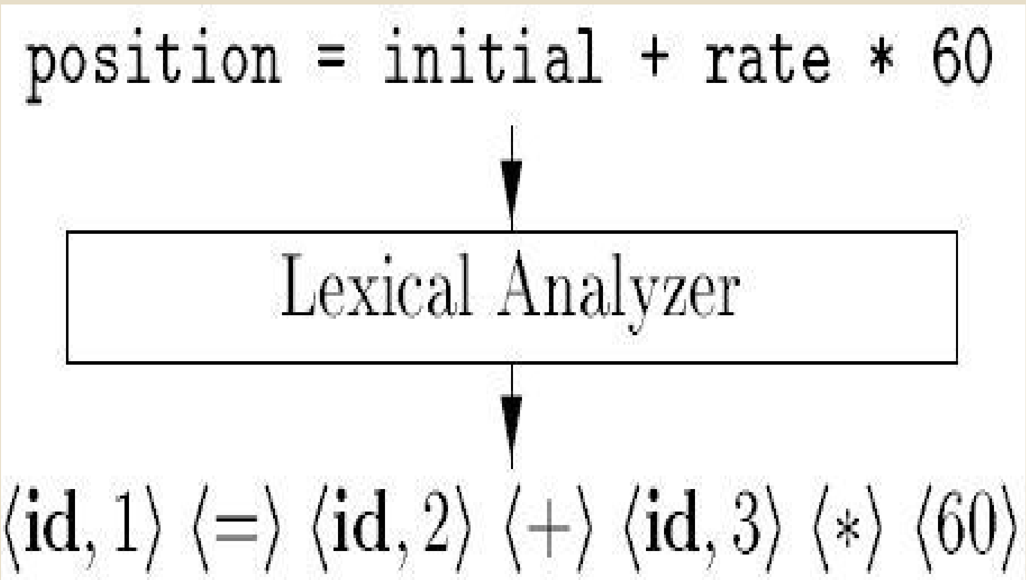
Lexical Analysis (Scanning)



Lexical Analysis (Scanning)

- The other important task of the lexical analyzer is to build a symbol table
- This is a table of all the identifiers (variable names, procedures, and constants) used in the program
- When an identifier is first recognized by the analyzer, it is inserted into the symbol table, along with information about its type, where it is to be stored, and so forth
- This information is used in subsequent passes of the compiler

Lexical Analysis (Scanning)



1	position	...
2	initial	...
3	rate	...

SYMBOL TABLE

Syntax Analysis(Parsing)

- Tokens found during scanning are grouped together using **context free grammar**
- The grammar is a set of rules that define valid structures in programming languages
- Each token is associated with a specific rule and grouped accordingly
- This process is called **parsing**
- The output of this phase is parse (syntax) tree or derivation

Syntax Analysis (Parsing)

- If the program follows the rules of the language, then it is syntactically correct
- When the parser encounters a mistake, it issues a warning or error message and tries to continue
- When the parser reaches the end of the token stream, it will tell the compiler that either the program is grammatically correct and compiling can continue or the program contains too many errors and compiling must be aborted
- If a parse tree is reached where there are only tokens, the corresponding statements is valid.

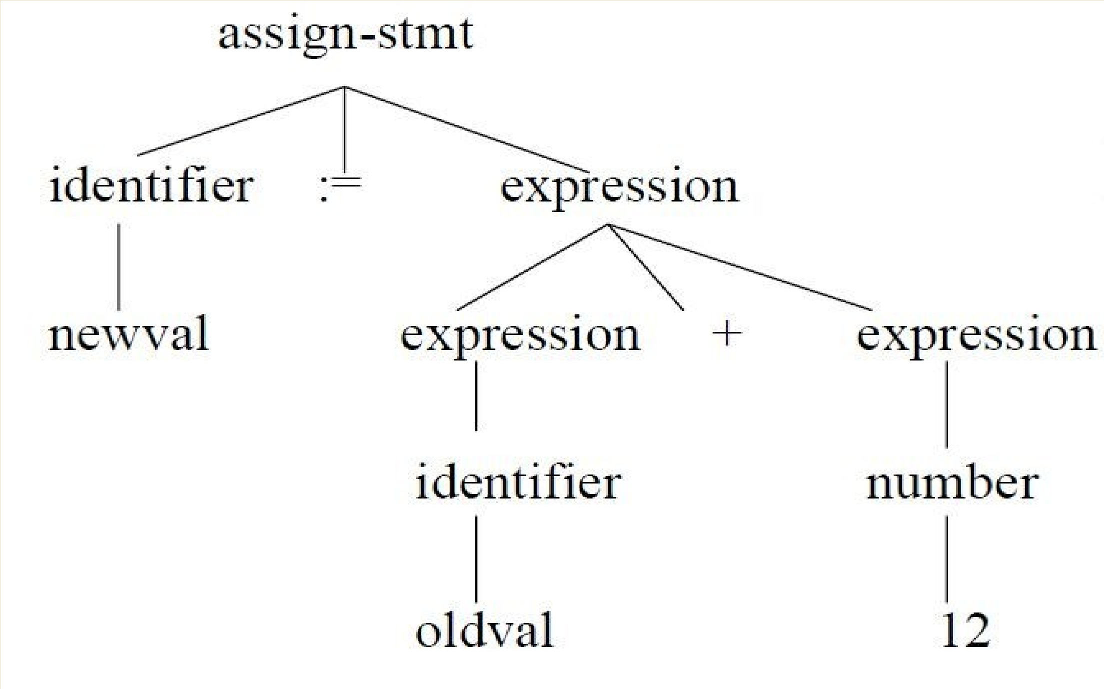
Syntax Analysis(Parsing)

- Given a CFG in Backus Naur Form (BNF):

```
assign-stmt -> identifier := expression
expression  -> identifier
expression  -> number
expression  -> expression + expression
```

Syntax Analysis(Parsing)

- Parse tree for : newval := oldval + 12

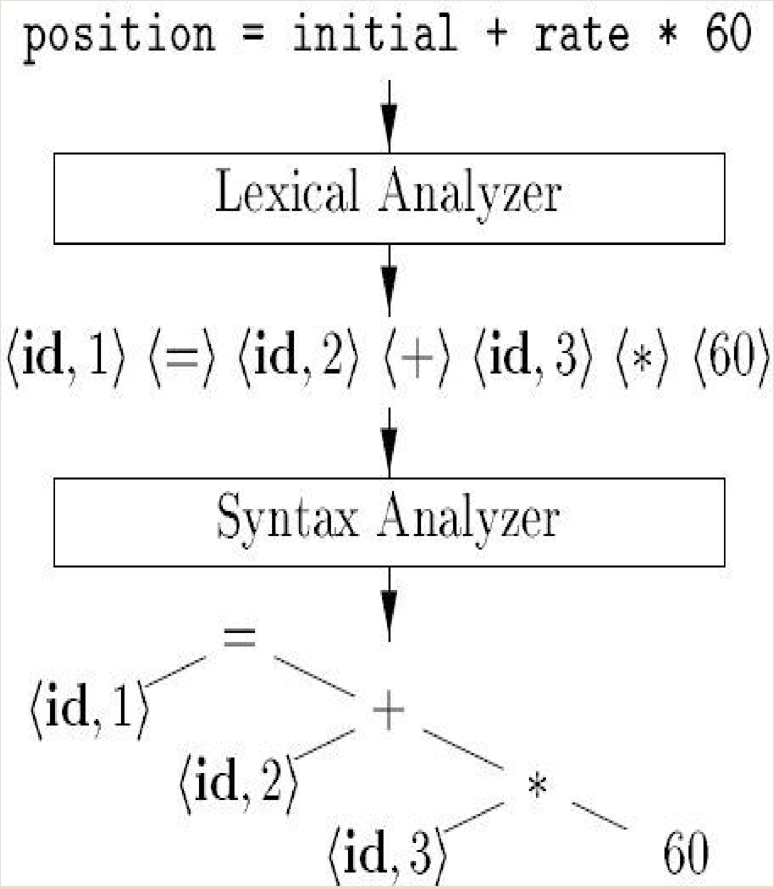


Syntax Analysis(Parsing)

- Derivation:

  assign-stmt -> identifier := expression
  -> newval := expression
  -> newval := expression + expression
  -> newval := identifier + expression
  -> newval := oldval + expression
  -> newval := oldval + number
  -> newval := oldval + 12

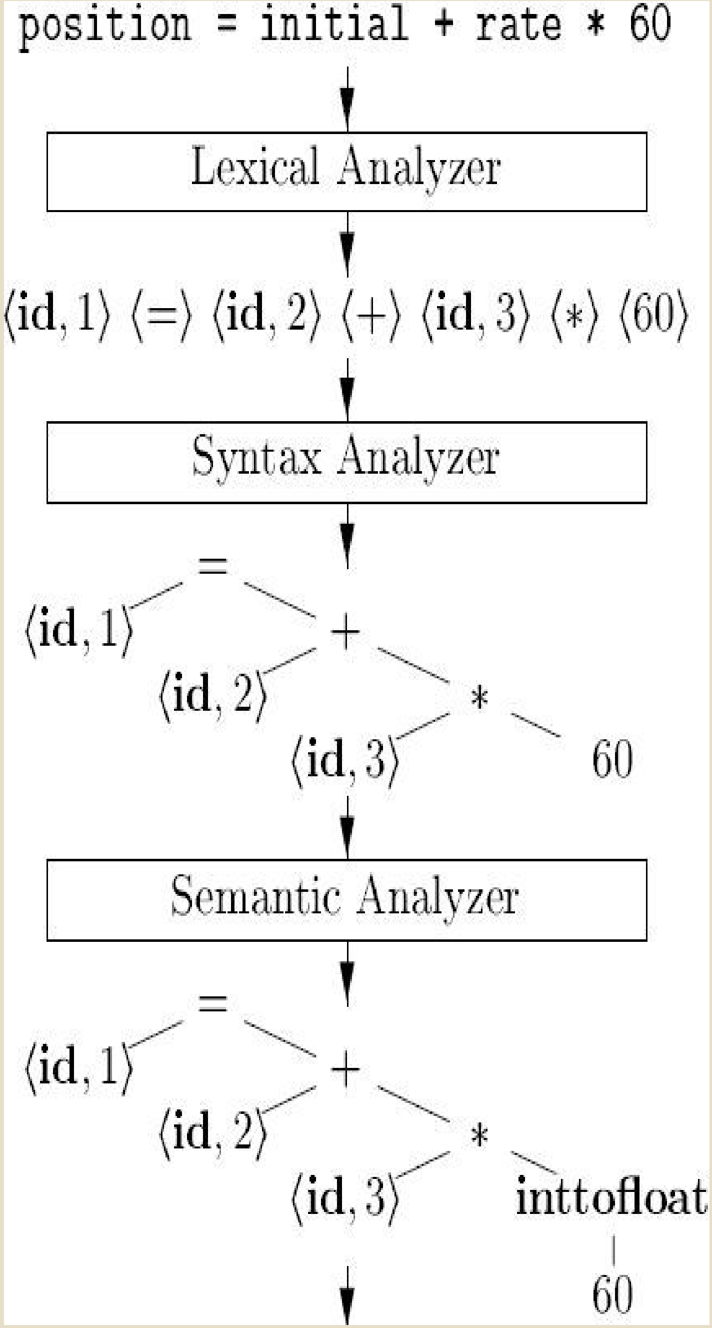
Syntax Analysis(Parsing)



Semantic Analysis

- The parse tree or derivation is next checked for semantic errors
- Semantic errors are statements that are syntactically correct but disobey the semantic rules of source language
- Detection of things such as undeclared variables, functions with improper arguments, access violation, incompatible operands, etc
- Type-checking is an important part of semantic analyzer
- Eg: int a[9], int b, int c;

  c = a * b; //Syntactically correct but semantically incorrect


position = initial + rate * 60
Lexical Analyzer
⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩
Syntax Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
60
Semantic Analyzer
=
⟨id, 1⟩
+
⟨id, 2⟩
*
⟨id, 3⟩
inttofloat
60

Intermediate Code Generation

- An intermediate language is often used by many compilers for analyzing and optimizing the source program
- Intermediate language should have two important properties:
  - It should be simple and easy to produce
  - It should be easy to translate target program
- Intermediate codes are generally machine (architecture) independent
- But the level of intermediate codes is close to the level of machine codes

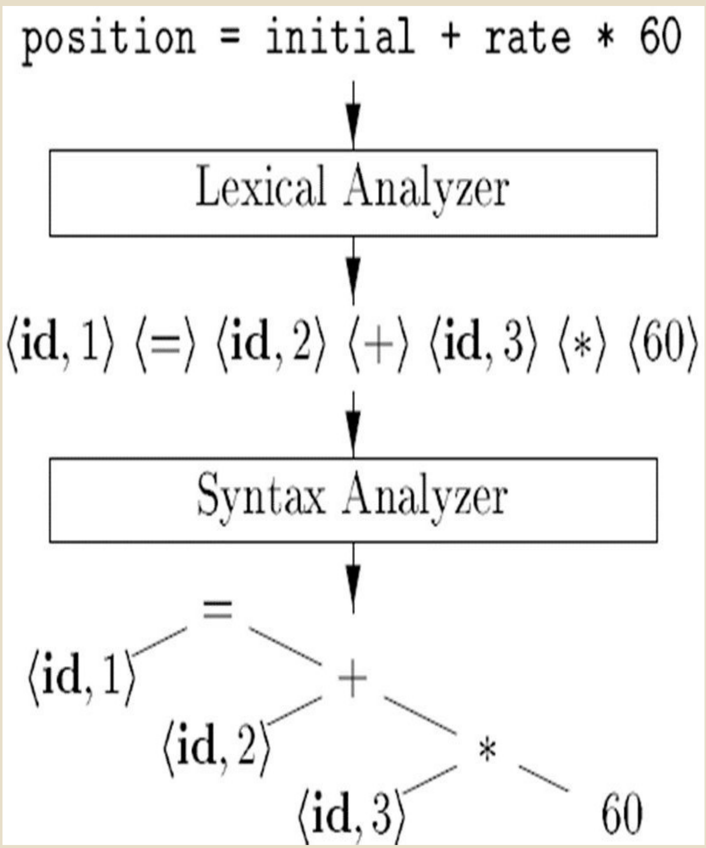
Intermediate Code Generation

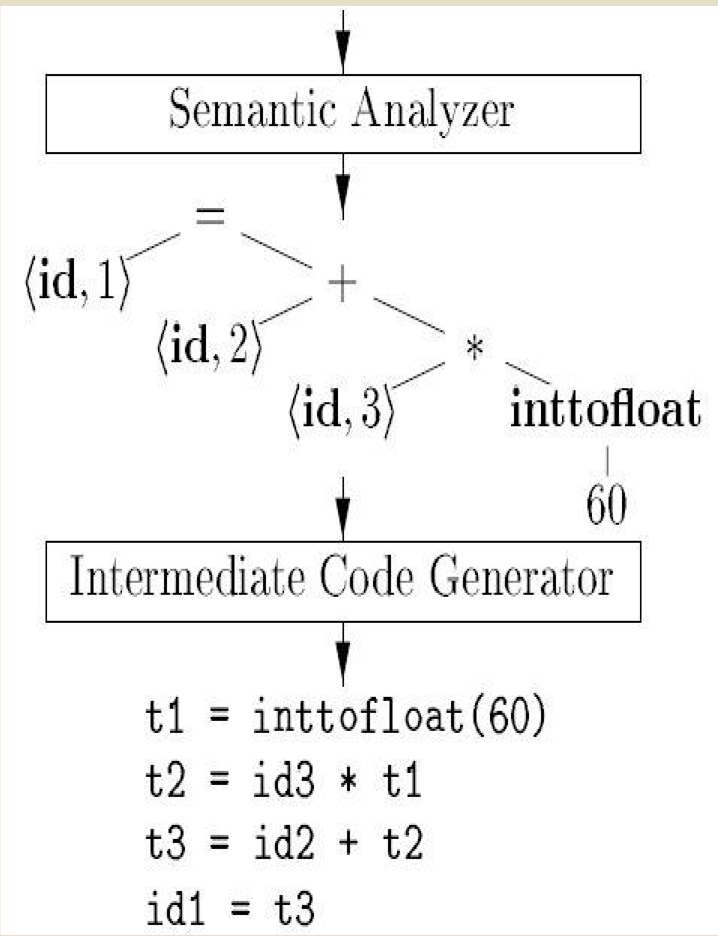
- A common form used for intermediate codes is **Three Address Code (TAC)**
- TAC looks like assembly language but does not represent a particular architecture
- TAC is a sequence of simple instructions, each of which can have at most 3 operands and two operators.

Intermediate Code Generation


```
newval := oldval * fact + 1
        ↓
id1 := id2 * id3 + 1
        ↓
temp1 = int2flot(1)
temp2 = id2 * id3
temp3 = temp1 + temp2
id1   = temp3
```

Intermediate Code Generation





Code Optimization

- The optimizer accepts input in the intermediate representation (TAC) and outputs a streamlined version still in intermediate representation
- In this phase, the compiler attempts to produce the smallest, fastest and most efficient running result by applying various techniques as:
  - Removing unused variables
  - Eliminating multiplication by 1 and addition by 0
  - Loop Optimization
  - Suppressing code generation of unreachable

Code Optimization

- The optimization phase slows down the compiler
- So most compilers allow this feature to be suppressed or turned off by default

- Example:

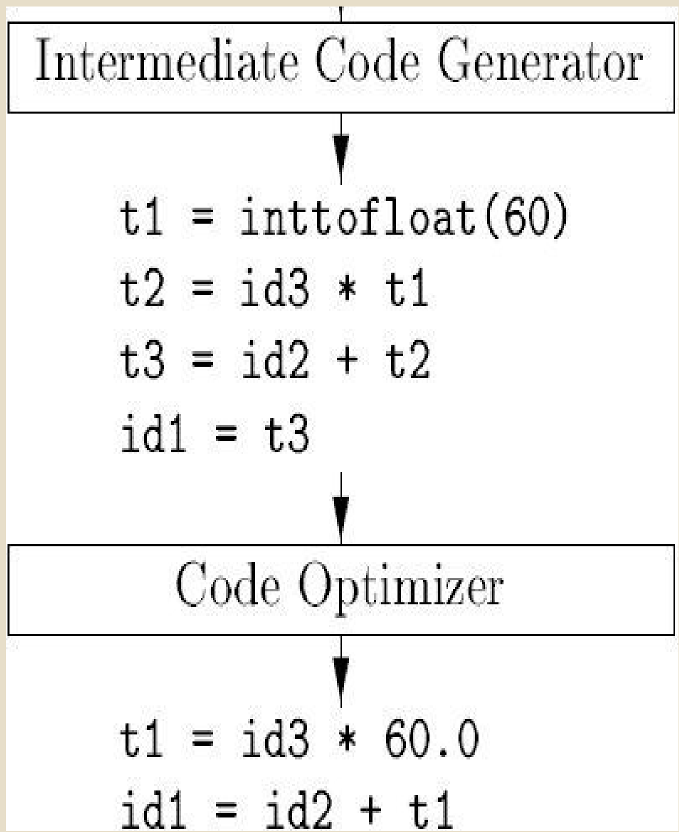
```
t1 = b * c
t2 = t1 + 0
t3 = b * c
t4 = t2 + t3
a = t4
```

Code Optimization

- Optimization: $t1 = b * c$

  $a = t1 + t1$

Code Optimization



Code Generation

- This process takes the intermediate code produced by the optimizer and generates final code in target language
- It is this part of the compilation phase that is machine dependent
- The target code is normally is a relocatable object file containing the machine or assembly codes
- The TAC is translated into a sequence of assembly or machine language instructions that perform the same task

Code Generation

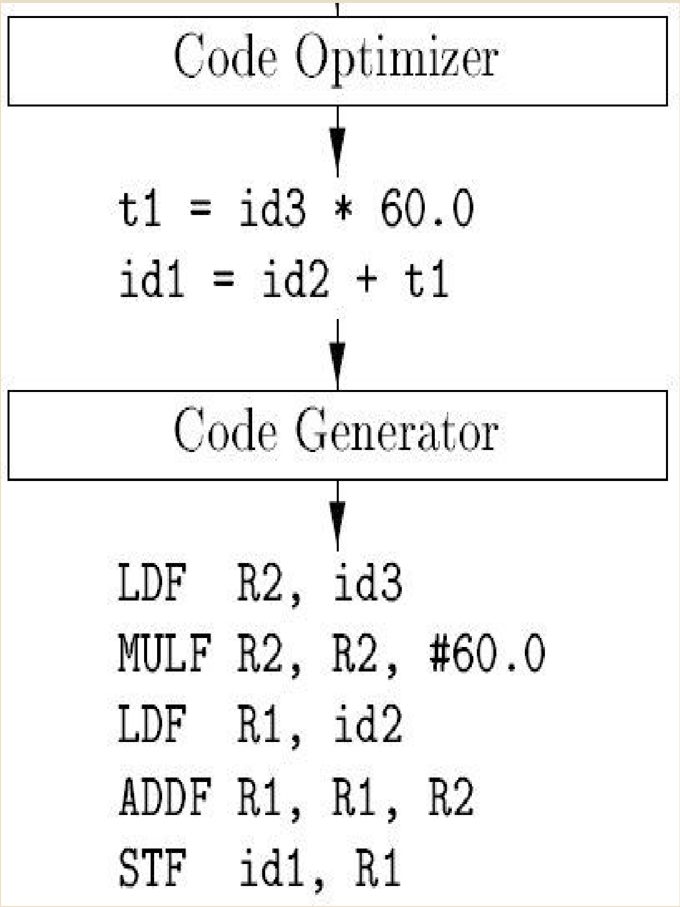
- Example (TAC):

```
t1 = b * c
a = t1 + t1
```

- Corresponding assembly code (target program):

```
LDA R1, b
LDA R2, c
MUL R1, R2
STA t1, R1
MOV R3, t1
ADD R3, t1
MOV a, R3
```

Code Generation



Object Code Optimization

- In this phase, the object code is transferred into more efficient code by making more efficient use of processor and registers
- The compiler can take advantage of machine specific idioms, specialized instructions, pipelining, branch prediction and other optimization techniques
- As with intermediate code optimization, this phase of compiler is either configurable or skipped entirely

Object Code Optimization

- Optimized version of above example:

```
LDA R1, b
MUL R1, c
STA t1, R1
ADD R1, t1
MOV a, R1
```

Symbol Table

- A symbol table stores information about keywords and tokens found during lexical analysis
- The symbol table is consulted in almost all phases of the compiler
- Example:

Insert("dist", id)   //insert a symbol table entry associating the string "dist" with token type "id"

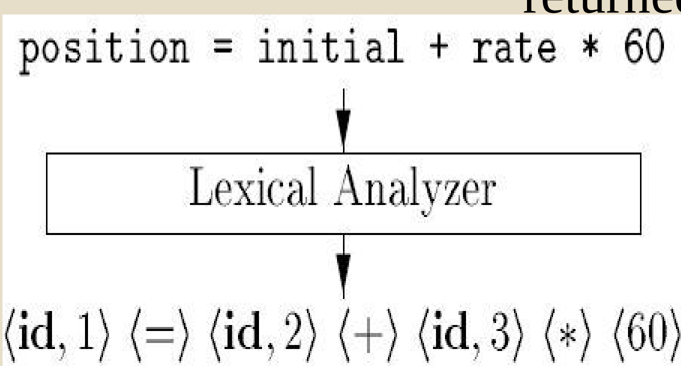
Symbol Table

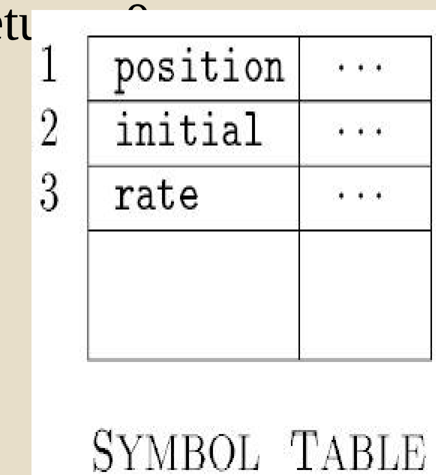
- Example:

Lookup(“dist”) //An occurrence of string “dist” can be looked up in the

symbol table. If found, the reference to the “id” is

returned else lookup retu





Error Handling

- Errors may be encountered in different phases of compiler
- Objective of error handling is to go as far as possible in compilation whenever an error is encountered
- Examples:
  - Handling missing symbols during lexical analysis by inserting symbol
  - Automatic type conversion during semantic analysis

Self Explore:

- A simple one-pass compiler
- One-pass/Multi-pass compiler