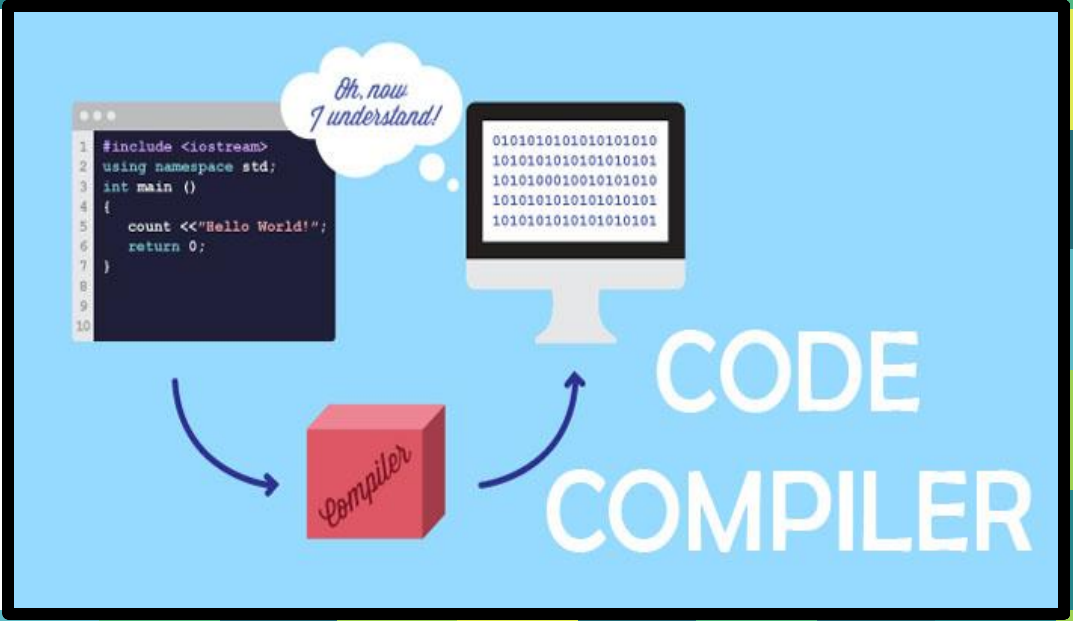
CSE- 303
Compiler
Chapter - 1



```
#include <iostream>
using namespace std;
int main ()
{
   count <<"Hello World!";
   return 0;
}
```

```
0101010101010101010
1010101010101010101
1010100010010101010
1010101010101010101
1010101010101010101
```

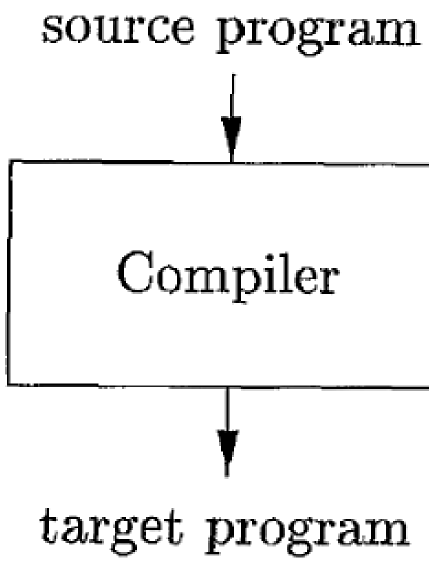
Introduction

- Programming languages are notations for describing computations to people and to machines.
- Before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- The software systems that do this translation are called compilers.

Language Processors

- A compiler is a program that can read a program in one language (source language) and translate it into an equivalent program in another language (target language).

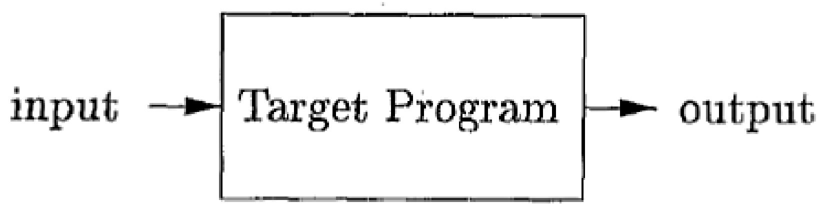
- An important role of the compiler is to report any errors in the source program that it detects during the translation process.



A compiler

Language Processors

- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



Running the target program

Language Processors

- After executing a C program file:

  hello.c → source code

  hello.o → object file; contains function definitions in binary form

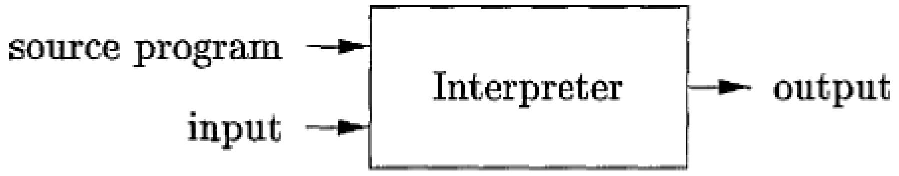
  hello.exe → linker/ executable file; links together a number of object files to produce a binary file which can be directly executed

- C language uses compiler

Language Processors

- 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.



An interpreter

Language Processors

- After executing a python program file:

  hello.py → source code

  hello.pyc → interpreted to bytecode; bytecode is executed by software called a virtual machine (VM)

- Python language does not use compiler; The Python implementation compiles the files as needed; portable

Language Processors

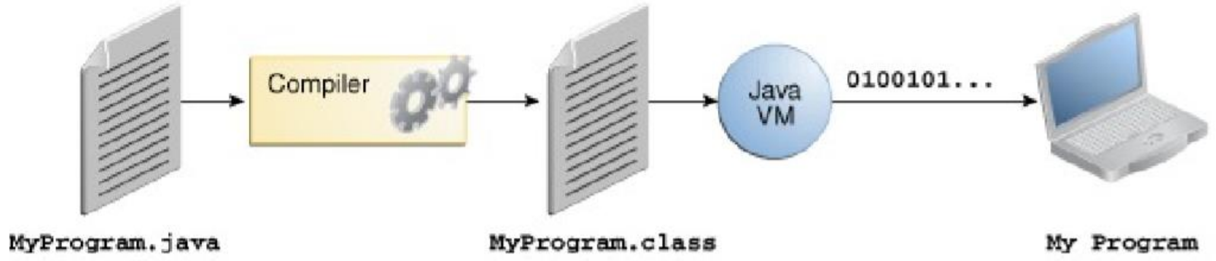
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.

- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

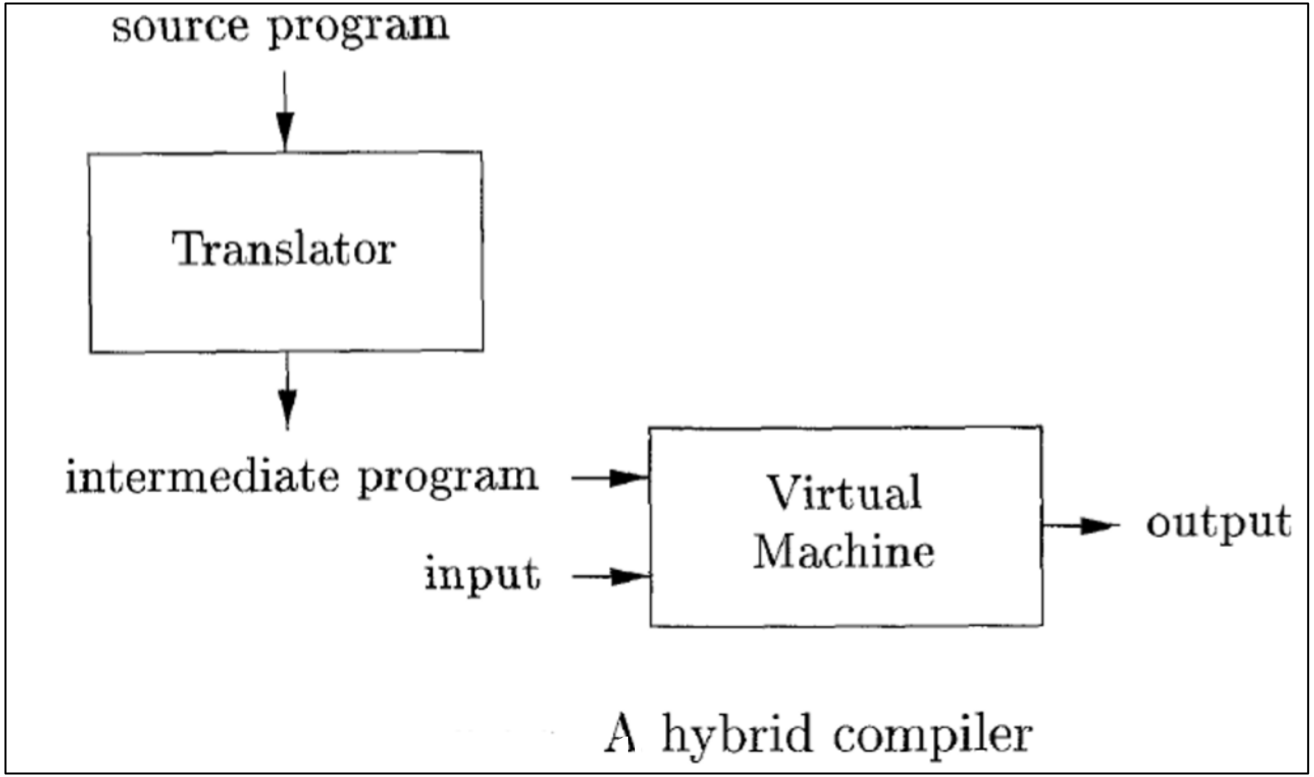
Language Processors

- Example 1.1: Java language processors combine compilation and interpretation (hybrid compiler)

$$\begin{tikzcd}
\text{Java} \arrow[r, "\text{Compiled by JavaC}"'] & \text{Bytecodes} \arrow[r, "\text{input}", "\text{interpreted by JVM}"'] & \text{Output}
\end{tikzcd}$$



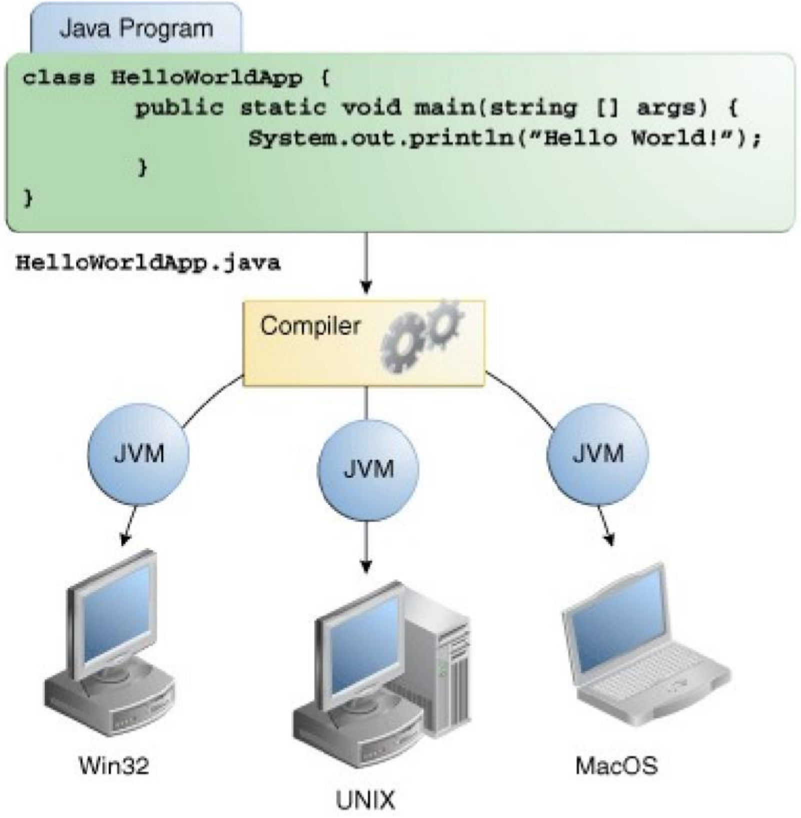
Language Processors



A hybrid compiler

Language Processors

- Benefit: bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.



```
class HelloWorldApp {
        public static void main(string [] args) {
                System.out.println("Hello World!");
        }
}
```

Language Processors

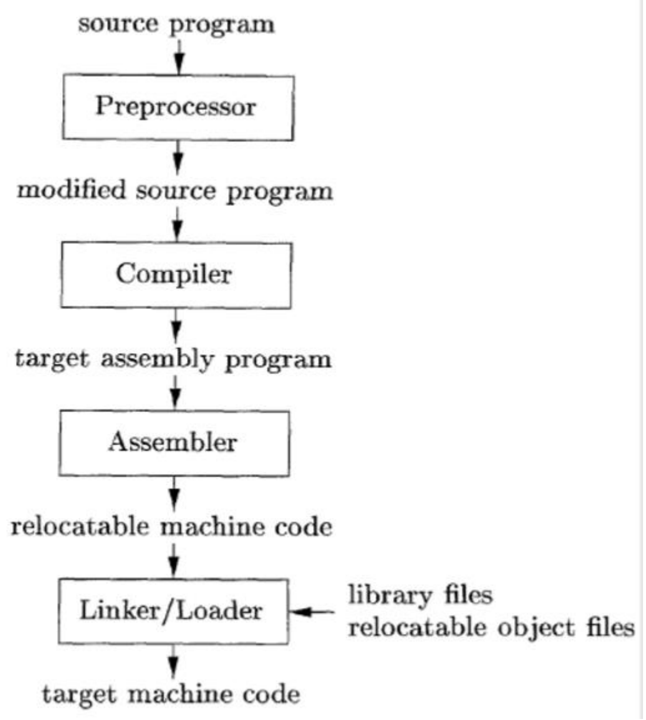
- For faster processing, just-in-time java compiler is used

$$\text{Java} \xrightarrow{\qquad\qquad} \text{Bytecodes} \xrightarrow{\text{JVM +JIT}} \text{Machine Language}$$

- It translates the bytecodes into machine language immediately before they run the intermediate program to process the input.

Language Processors

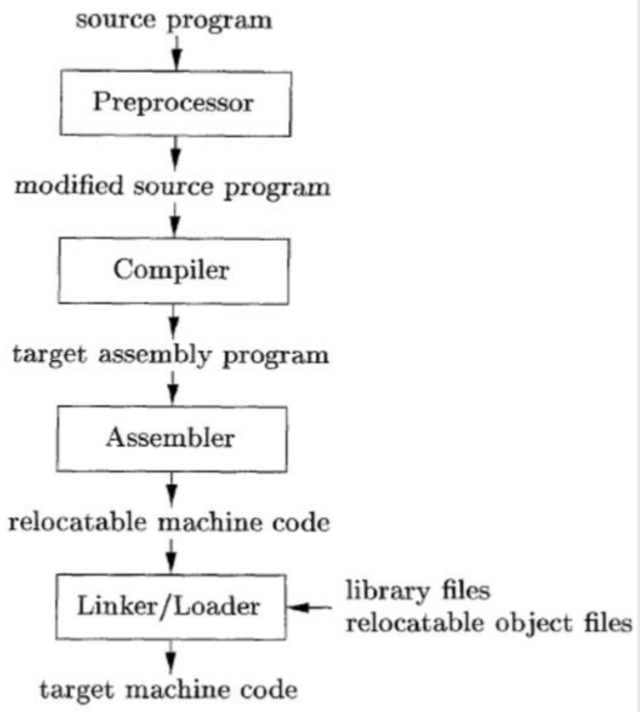
- In addition to a compiler, several other programs may be required to create an executable target program.
- A source program may be divided into modules stored in separate files.
- Preprocessor collects the source program
- The preprocessor may also expand shorthands, called macros, into source language statements.



A language-processing system

Language Processors

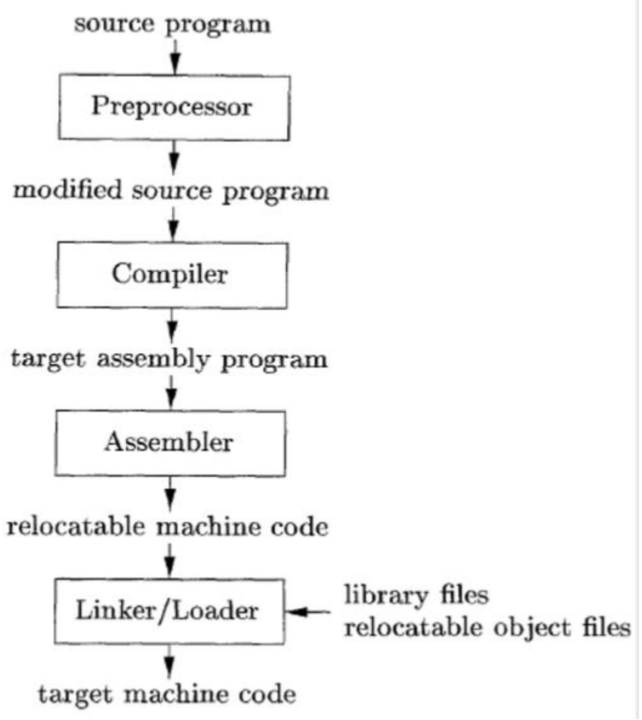
- The modified source program is then fed to a compiler.
- The compiler may produce an assembly-language program as its output, because assembly language is easier to produce as output and is easier to debug.
- The assembly language is then processed by a program called an assembler that produces relocatable machine code as its output.



A language-processing system

Language Processors

- ❑ Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files and library files
- ❑ The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- ❑ The loader then puts together all of the executable object files into memory for execution.



A language-processing system

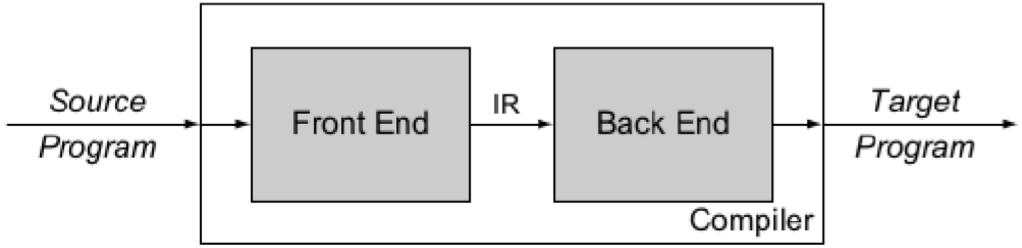
The Structure of a Compiler

- To translate text from one language to another, the tool must understand both the form or syntax, and content or meaning of the input language.

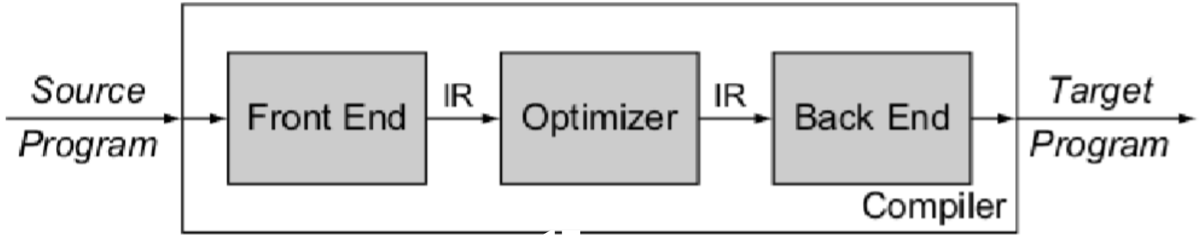
- Two parts of compiler: Analysis & Synthesis
- **Analysis (Front end):** checks for syntactic or semantic error; provide error message; stores information of source code in symbol table.
- **Synthesis (Back end):** constructs target program from intermediate representation and symbol table

- Connecting the front end and the back end, it has a formal structure for representing the program in an intermediate form whose meaning is largely independent of either language.

The Structure of a Compiler



- To improve the translation, a compiler often includes an optimizer that analyzes and rewrites that intermediate form.



The Structure of a Compiler

- Compilation process operates as a sequence of phases
- Several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.
- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

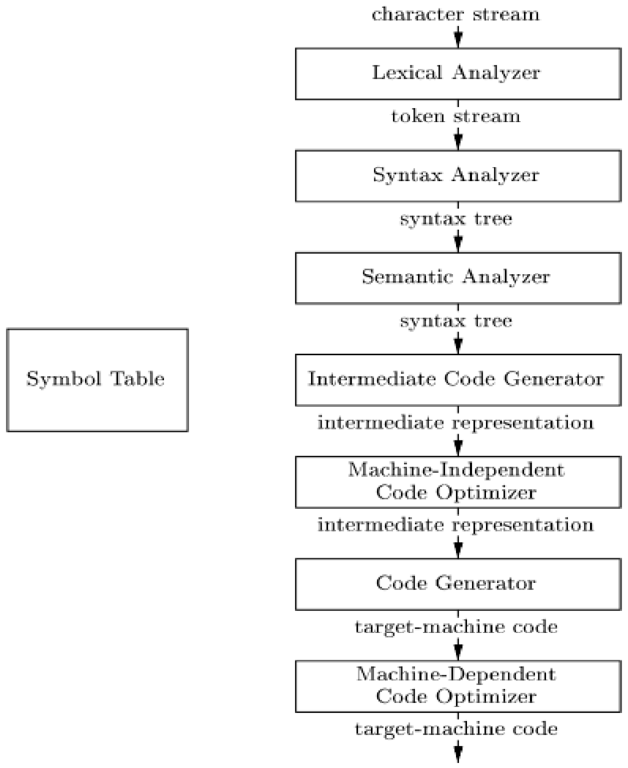
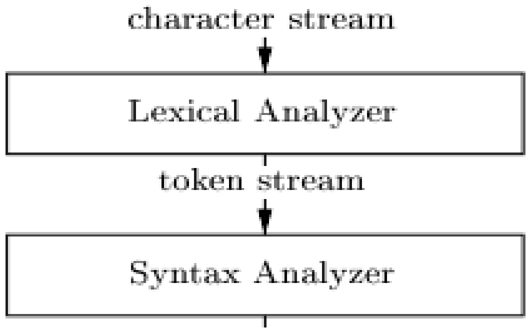


Figure 1.6: Phases of a compiler

Lexical Analysis

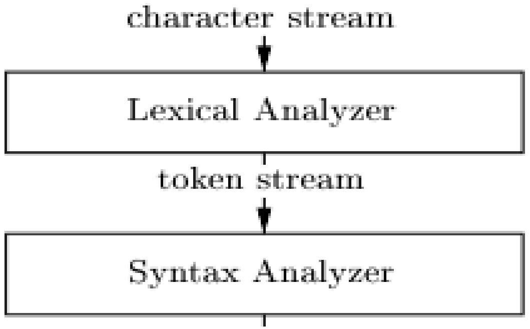
- The first phase of a compiler is called lexical analysis or scanning.
- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.
- For each lexeme, the lexical analyzer produces as output a token of the form

  <token-name, attribute-value>
- It passes token on to the subsequent phase, syntax analysis.



Lexical Analysis

- ❑ The first component token-name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token.
- ❑ Information from the symbol-table entry is needed for semantic analysis and code generation.



Example:

$\texttt{position} = \texttt{initial} + \texttt{rate} * 60$

Lexical Analysis

$$\mathtt{position} = \mathtt{initial} + \mathtt{rate} * 60$$

Tokenization

1. position $\rightarrow$ < id, 1>
2. = $\rightarrow$ < = >
3. initial $\rightarrow$ < id, 2>
4. + $\rightarrow$ < + >
5. rate $\rightarrow$ < id, 3>
6. 60 $\rightarrow$ < 60 >

Symbol Table

Index	Information about token
1	position, identifier
2	initial, identifier
3	rate, identifier
….	….

Lexical Analysis

position = initial + rate * 60

$\downarrow$

Lexical Analyzer

$\downarrow$

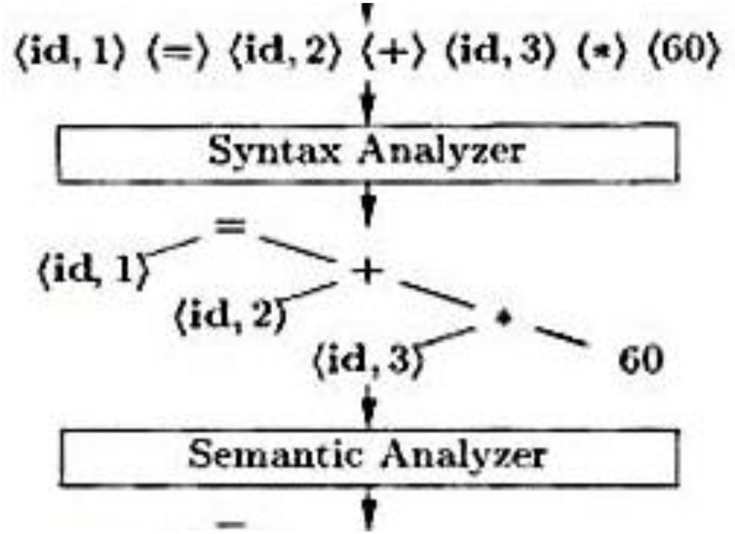
< id,1> <=> <id,2> <+> <id,3> <*> <60>

Symbol Table

Index	Information about token
1	position, identifier
2	initial, identifier
3	rate, identifier
….	….

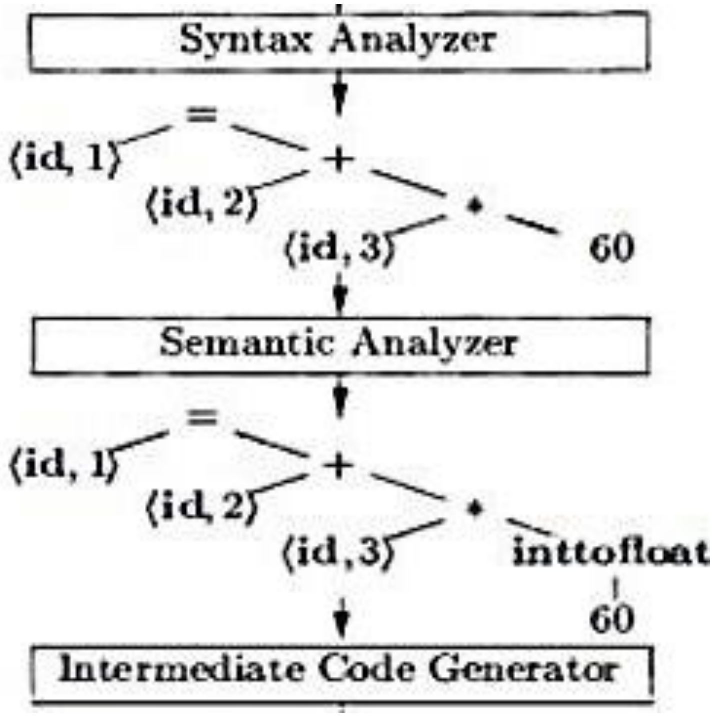
Syntax Analysis

- The second phase of the compiler is syntax analysis or parsing.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation of the token stream (grammatical structure).
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.
- This tree shows the order in which the operations are to be performed (follows conventions of arithmetic)



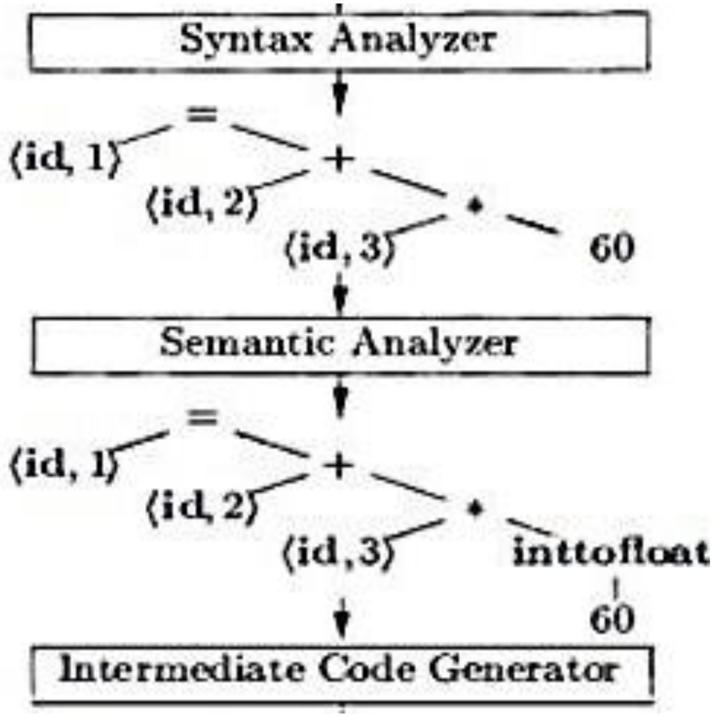
Semantic Analysis

- The semantic analyzer uses the syntax tree and information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.
- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. Example: checks if index array is integer, if not then reports error



Semantic Analysis

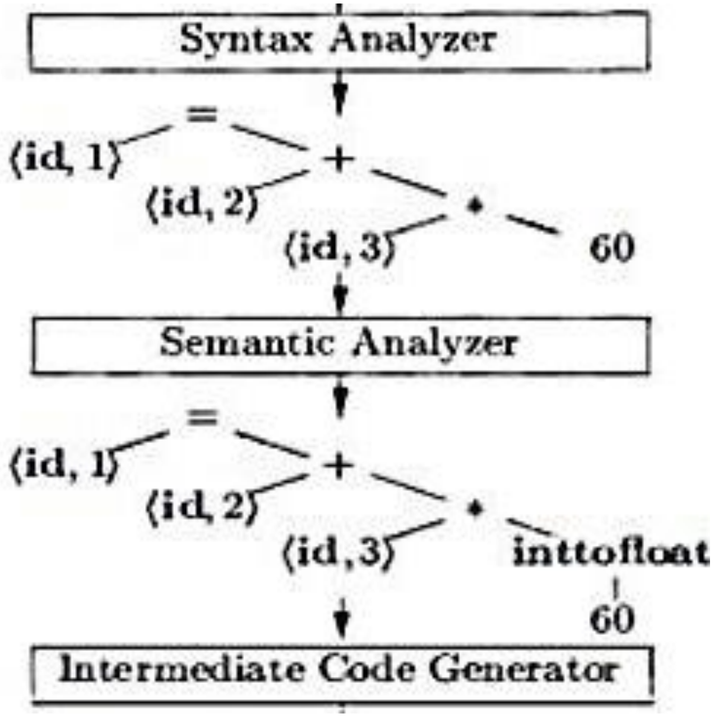
- The language specification may permit some type conversions called coercions.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.
- If the operator is applied to a floating-point number and an integer, the compiler may convert or coerce the integer into a floating-point number.



Semantic Analysis

- Example: if rate is declared as floating point number and 60 is an integer, the type checker in the semantic analyzer discovers that and converts integer into a floating-point number.

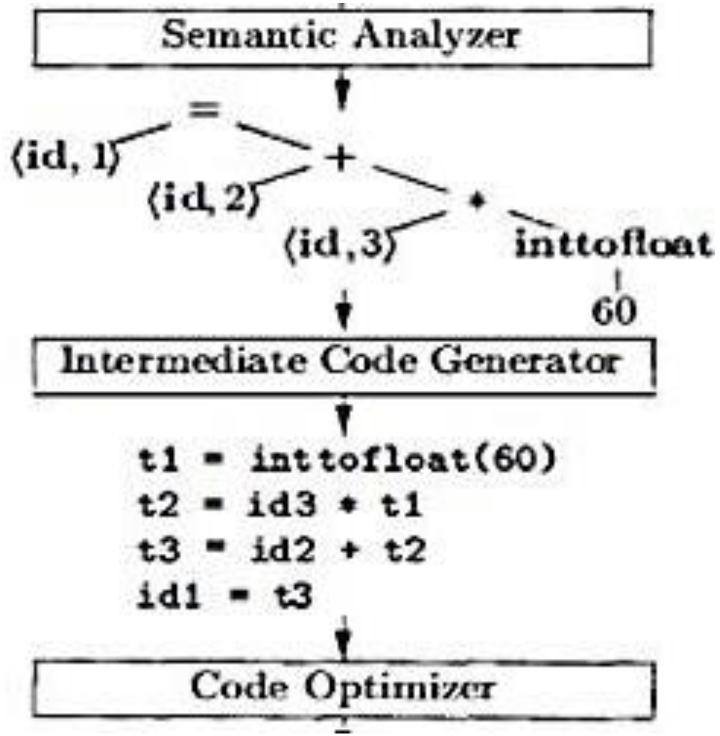
- Notice that the output of the semantic analyzer has an extra node for the operator inttofloat, which explicitly converts its integer argument into a floating-point number.



Intermediate Code Generation

- In the process of translating a source program into target code, a compiler may construct one or more intermediate representations, which can have a variety of forms.

- Example: syntax tree, machine-like intermediate representation



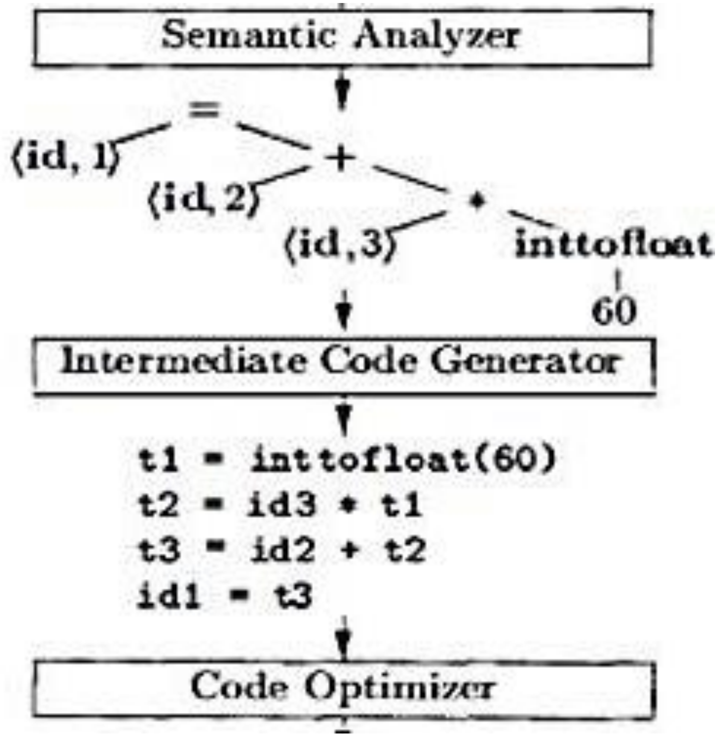
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate Code Generation

- ❑ After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation.

- ❑ This intermediate representation should have two important properties:
  - ✓ It should be easy to produce
  - ✓ It should be easy to translate into the target machine.

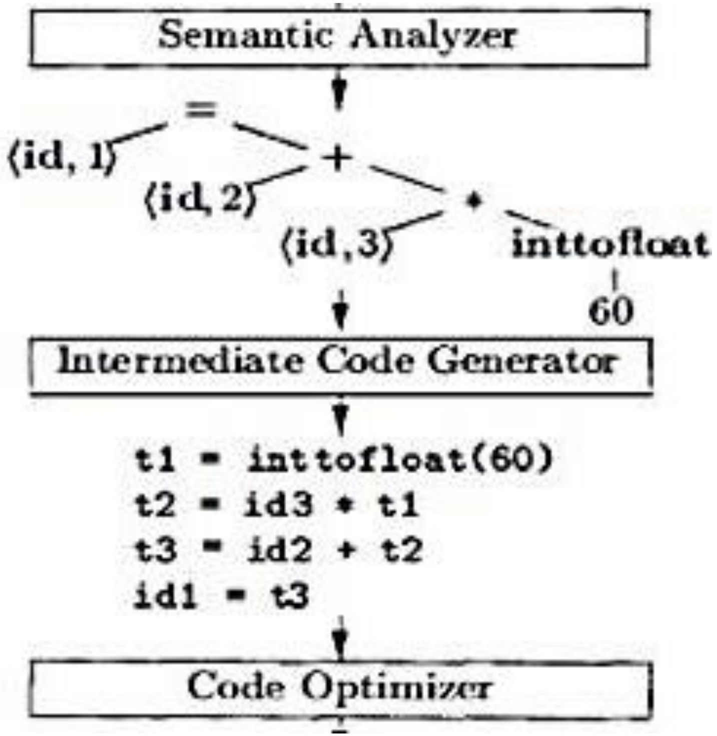
- ❑ We consider an intermediate form called “three-address code”.



```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate Code Generation

- In assembly language, every memory location can act like a register with three operands per instruction.
- Similarly, Three-address code consists of a sequence of instructions, each of which has at most three operands (three-address code sequence).



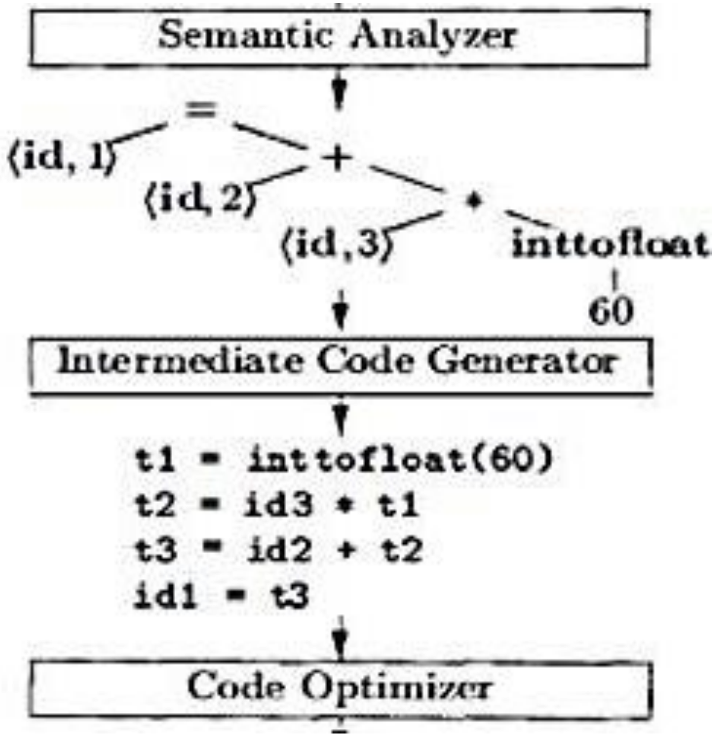
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Intermediate Code Generation

- ❑ Points about Three-address instructions:

- ➢ At most one operator at the right side (decides the order in which operations are to be done.)
- ➢ Must generate a temporary name to hold the value computed by each instruction.
- ➢ Some "three-address" instructions have fewer than three operands.

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



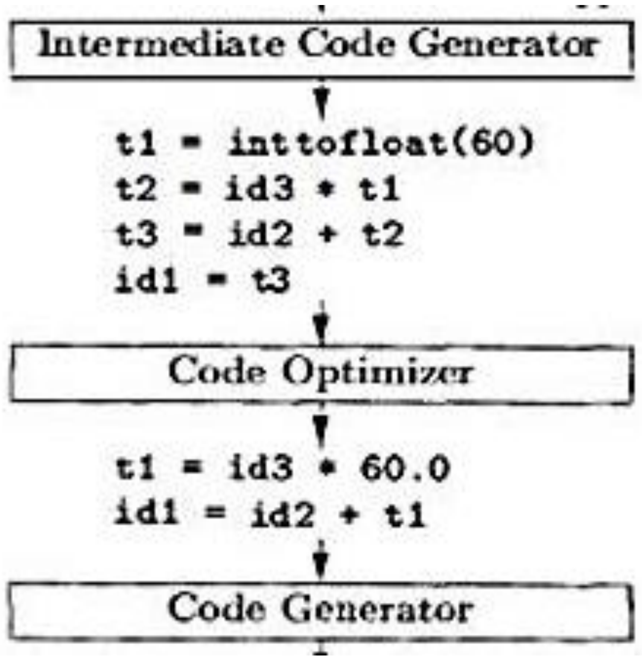
```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

Code Optimization

- The machine independent code optimization phase attempts to improve the intermediate code so that better target code will result.

- Better means faster or shorter code or target code that consumes less power.

- A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.

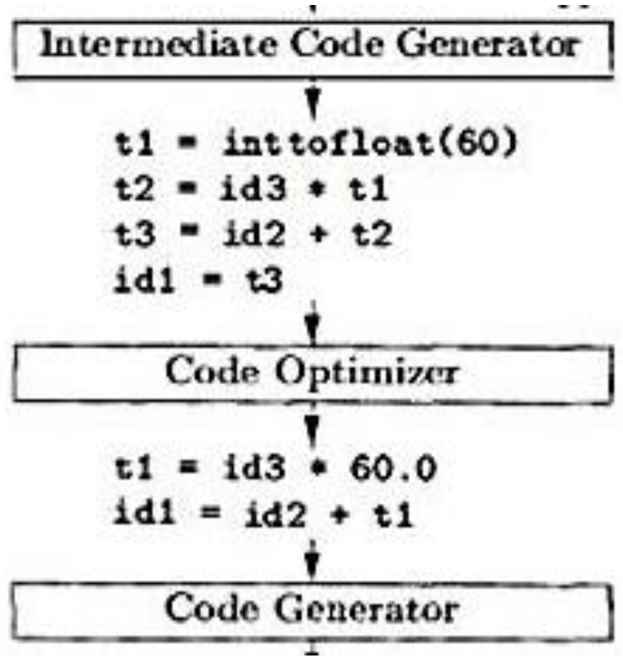


Code Optimization

- The optimizer can deduce that the conversion of 60 from integer to floating point can be done once and for all at compile time. So the inttofloat operation can be eliminated by replacing the integer 60 by the floating-point number 60.0

- Moreover, t3 is used only once to transmit its value to <id,1>

- There is a great variation in the amount of code optimization different compilers perform.

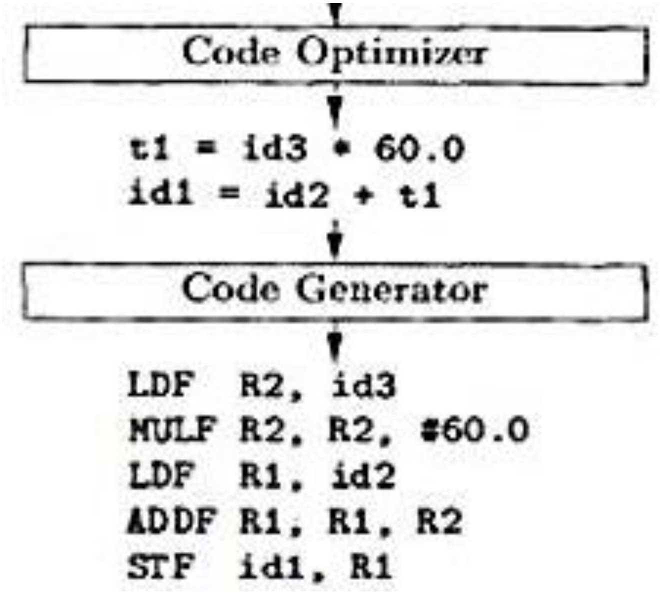


```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```

```
t1 = id3 * 60.0
id1 = id2 + t1
```

Code Generation

- ❑ The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- ❑ If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- ❑ Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task.



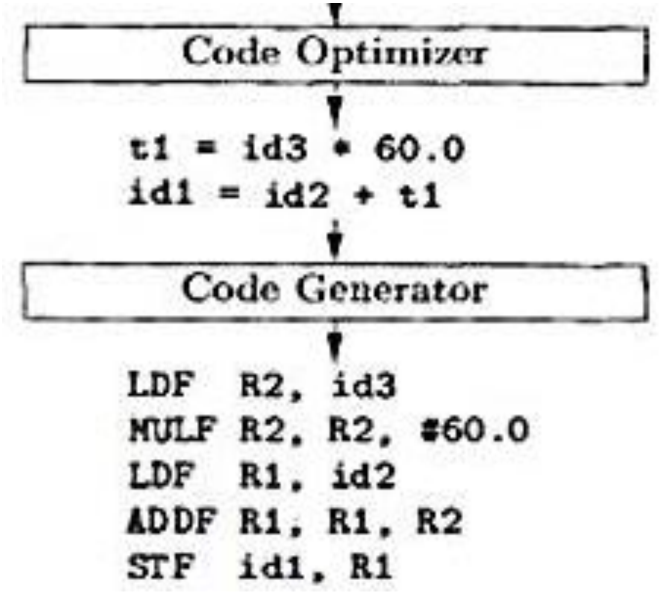
```
t1 = id3 * 60.0
id1 = id2 + t1
```

```
LDF  R2, id3
MULF R2, R2, #60.0
LDF  R1, id2
ADDF R1, R1, R2
STF  id1, R1
```

Code Generation

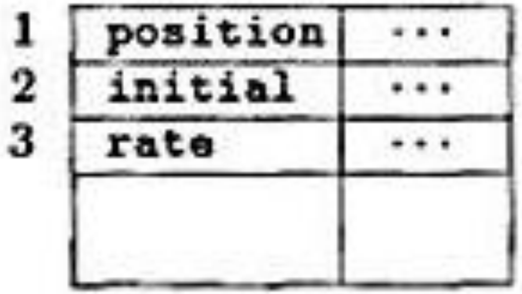
- For example, using registers R1 and R2, the intermediate code might get translated into the machine code.

```
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```



Symbol-Table Management

- ❑ An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name.
- ❑ These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used)
- ❑ In the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.



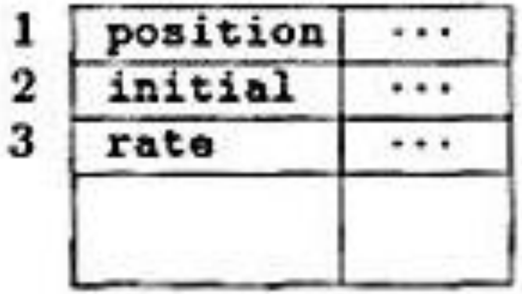
1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

Symbol-Table Management

- The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name.

- The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.



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

SYMBOL TABLE

The Grouping of Phases into Passes

- ❑ The discussion of phases deals with the logical organization of a compiler.
- ❑ In an implementation, activities from several phases may be grouped together into a pass that reads an input file and writes an output file.
- ❑ For example, the front-end phases of lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped together into one pass.
- ❑ Code optimization might be an optional pass.
- ❑ Then there could be a back-end pass consisting of code generation for a particular target machine.

The Grouping of Phases into Passes

- Some compiler collections have been created around carefully designed intermediate representations that allow the front end for a particular language to interface with the back end for a certain target machine.
- With these collections, we can produce compilers for different source languages for one target machine by combining different front ends with the back end for that target machine.
- Similarly, we can produce compilers for different target machines, by combining a front end with back ends for different target machines.

Compiler-Construction Tools

- The compiler writer, like any software developer, can profitably use modern software development environments containing tools such as language editors, debuggers, version managers, profilers, test harnesses, and so on.
- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.
- Some commonly used compiler-construction tools include:

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators that produce lexical analyzers from a regular-expression description of the tokens of a language
3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.

Compiler-Construction Tools

4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data-flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each other part. Data-flow analysis is a key part of code optimization.
6. Compiler-construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Thank You