CSE 309 (Compilers)

Basic Concepts

Dr. Muhammad Masroor Ali

Professor
Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1205, Bangladesh

January 2021

Version: 1.4, Last modified: March 7, 2021

Introduction

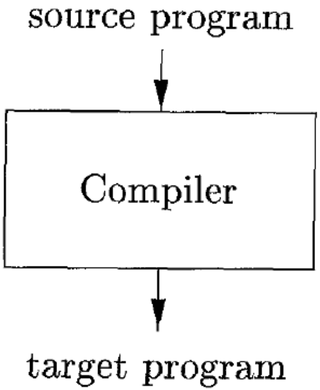
- Programming languages are notations for describing computations to people and to machines.
- The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language.
- But, before a program can be run, it first must be translated into a form in which it can be executed by a computer.
- The software systems that do this translation are called compilers.

Introduction

- Programming languages are notations for describing computations to people and to machines.
- The world as we know it depends on programming languages, because all the software running on all the computers was written in some programming language.
- But, 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

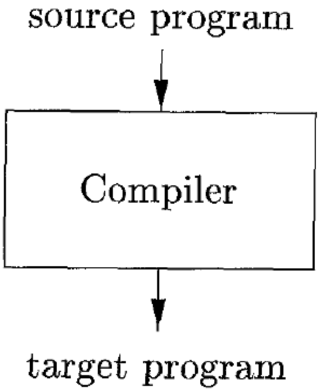
- Simply stated, a compiler is a program,
  - that can read a program in one language — the source language — and
  - translate it into an equivalent program in another language — the target language.



A compiler

Language Processors

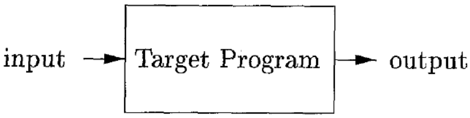
- An important role of the compiler is to report any errors in the source program that it detects during the translation process.



A compiler

Language Processors

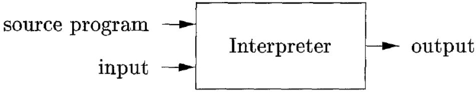
- If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs.



Running the target program

Language Processors — *continued*

- 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.

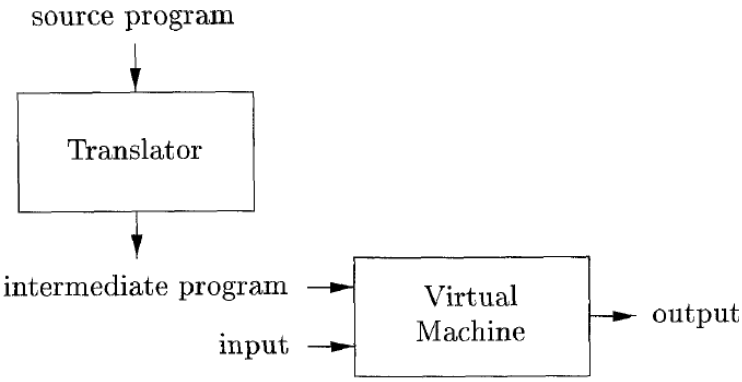


An interpreter

Language Processors — *continued*

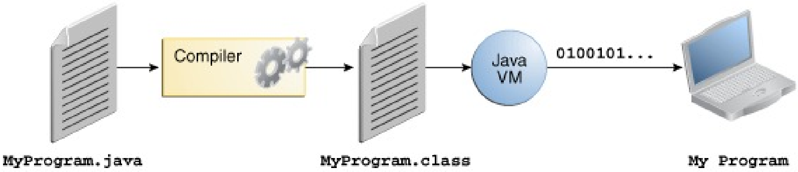
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs.
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

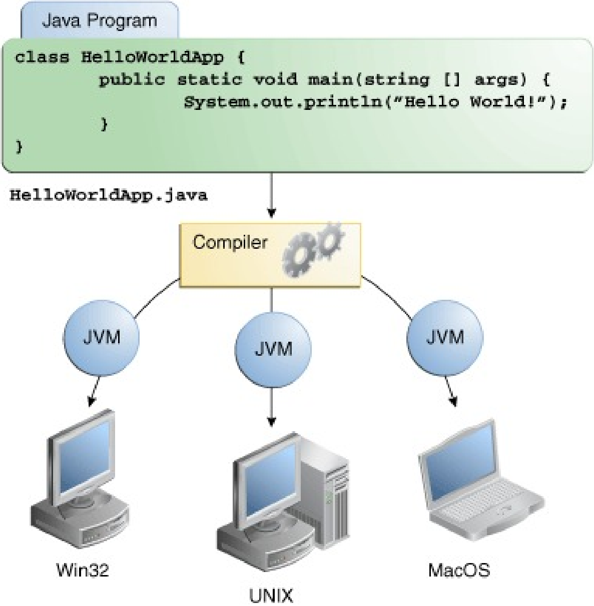
Example



A hybrid compiler

- Java language processors combine compilation and interpretation.


Compiler
Java
VM
0100101...
MyProgram.java
MyProgram.class
My Program

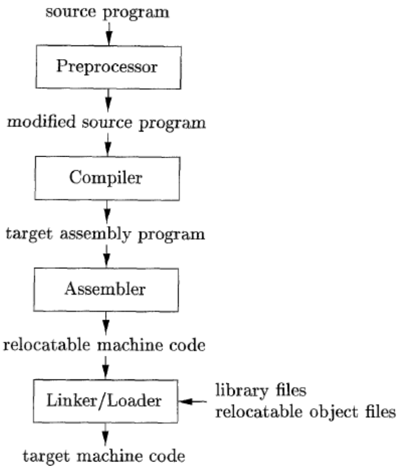

Java Program
class HelloWorldApp {
public static void main(string [] args) {
System.out.println("Hello World!");
}
}
HelloWorldApp.java
Compiler
JVM
JVM
JVM
Win32
UNIX
MacOS

Example — *continued*

- A Java source program may first be compiled into an intermediate form called bytecodes.
- The bytecodes are then interpreted by a virtual machine.
- A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.
- In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

Language Processors — *continued*

- 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.
- The task of collecting the source program is sometimes entrusted to a separate program, called a preprocessor.
- The preprocessor may also expand shorthands, called macros, into source language statements.



A language-processing system

C Preprocessor

- The C preprocessor or `cpp` is the macro preprocessor for the C and C++ computer programming languages.
- The preprocessor provides the ability for:
  - the inclusion of header files,
  - macro expansions,
  - conditional compilation, and
  - line control.

C Preprocessor — *continued*

- In many C implementations, it is a separate program invoked by the compiler as the first part of translation.

C Preprocessor — Including Files

- One of the most common uses of the preprocessor is to include another file:

```
#include <stdio.h>
int main(void)
{
  printf("Hello,␣world!\n");
  return 0;
}
```

C Preprocessor — Including Files

```
#include <stdio.h>
int main(void)
{
  printf("Hello,␣world!\n");
  return 0;
}
```

- The preprocessor replaces the line `#include <stdio.h>` with the text of the file ‘`stdio.h`’.

C Preprocessor — Conditional Compilation

- The `if-else` directives `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif` and `#endif` can be used for conditional compilation.

C Preprocessor — Conditional Compilation — *continued*

- Most compilers targeting Microsoft Windows implicitly define `_WIN32`.
- This allows code, including preprocessor commands, to compile only when targeting Windows systems.

C Preprocessor — Conditional Compilation — *continued*

```
#ifdef _unix_
/* _unix_ is usually defined by compilers
   targeting Unix systems */
# include <unistd.h>
#elif defined _WIN32
/* _WIN32 is usually defined by compilers
   targeting 32 or 64 bit Windows systems */
# include <windows.h>
#endif
```

- The example code tests if a macro `_unix_` is defined.
- If it is, the file `<unistd.h>` is then included.
- Otherwise, it tests if a macro `_WIN32` is defined instead.
- If it is, the file `<windows.h>` is then included.

C Preprocessor — Macro Definition and Expansion

```
#define PI 3.14159
#define RADTODEG(x) ((x) * 57.29578)
```

Line Control Directive

https://www.geeksforgeeks.org/line-control-directive/

- Whenever a program is compiled, there are chances of occurrence of some error in the program.
- Whenever the compiler identifies an error in the program it provides us with the filename in which the error is found along with the list of lines and with the exact line numbers where the error is.
- This makes it easy for us to find and rectify the error.

Line Control Directive — *continued*

- However, what information should the compiler provide during errors in the compilation can be controlled using the `#line` directive.
- It manipulates the compiler to read a line so that programmer can control the line and also the line read by the file compiler.
- It is basically used to reset the line number.

Line Control Directive — *continued*

```
// Demonstrate line control, clinecontrol.cpp
#include <iostream>
using namespace std;

int main()
{
  cout << "CSE␣309␣line␣control" << endl;

  cout << __LINE__ << "␣"
       << __FILE__ << endl;

#line 15 "ourdemo.cpp"

  cout << __LINE__
       << "␣" << __FILE__
       << endl;
  cout << __LINE__ << endl;
}
```

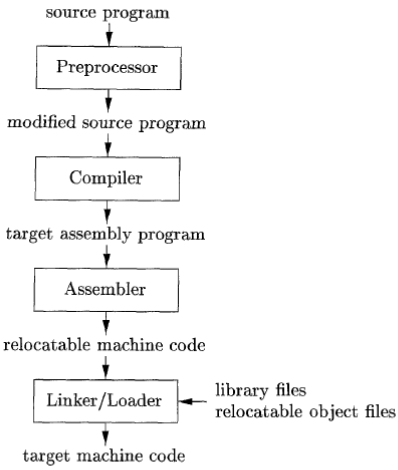
Line Control Directive — *continued*

```
// Demonstrate line control, clinecontrol.cpp
#include <iostream>
using namespace std;

int main()
{
  cout << "CSE␣309␣line␣control" << endl;

  cout << __LINE__ << "␣"
       << __FILE__ << endl;

#line 15 "ourdemo.cpp"

  cout << __LINE__
       << "␣" << __FILE__
       << endl;
  cout << __LINE__ << endl;
}
```

```
CSE 309 line control
9 clinecontrol.cpp
16 ourdemo.cpp
19
```

Language Processors — *continued*

- 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.



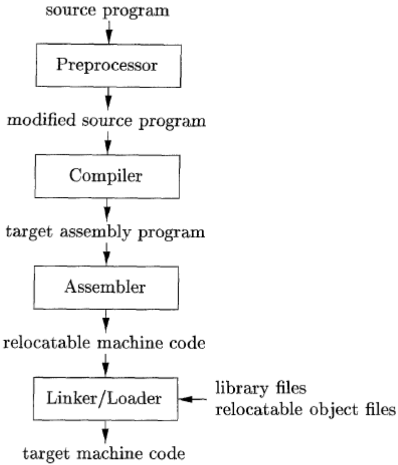
A language-processing system

Assembler

https://cs.stackexchange.com/a/13905/7200

- An assembler translates assembly code to machine code.
- The translation is mechanical, and can be done in only one way.
- Assembly languages are usually designed to be nearly trivial to parse and type check and tend to involve a lot of table-driven generators.

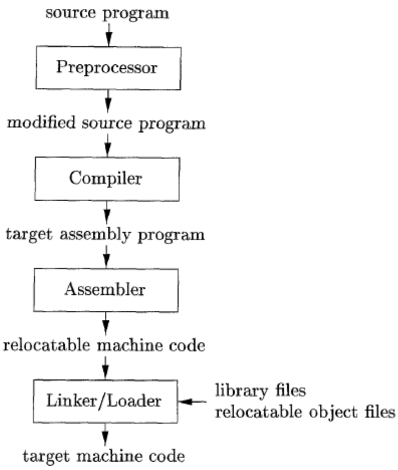
- Large programs are often compiled in pieces.
- So the relocatable machine code may have to be linked together with other relocatable object files and library files into the code that actually runs on the machine.



A language-processing system

Language Processors — *continued*

- 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

Relocatable Machine Code

https://www.wikiwand.com/en/Relocation_(computing), https://stackoverflow.com/a/46101660/1337854 and https://gateoverflow.in/18466/difference-between-relocatable-machine-absolute-machine

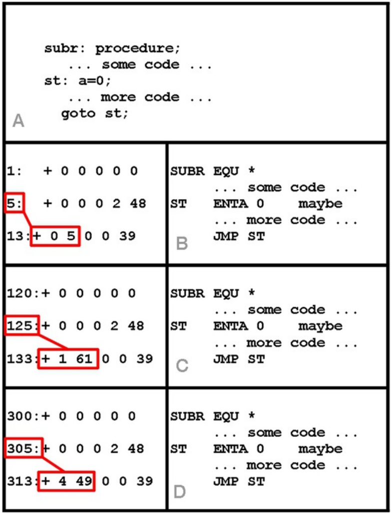
- Relocation is the process of assigning load addresses for position-dependent code and data of a program and adjusting the code and data to reflect the assigned addresses.
- Relocatable-Code, as the name suggests, is relocatable in that code can be loaded anywhere in memory.
- It usually has be relocated or fixed up before it is executable.

Relocatable Machine Code — *continued*

- When the machine code is generated then the computer does not where the code will be placed in RAM till run time.
- So all the lines are numbered from 0-1-2... and so on.
- Now suppose your program is loaded at 230 address then this address on run time will be converted to $230 + 0$, $230 + 1$, $230 + 2$.
- This means every line will be adjusted acordingly.

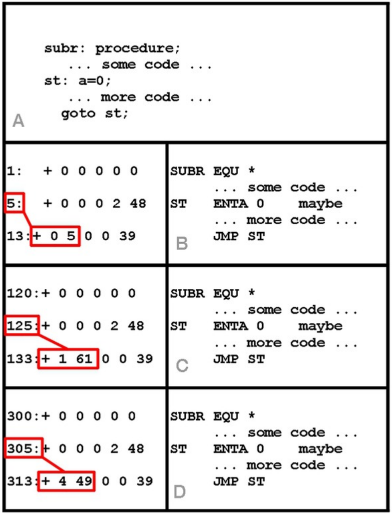
Relocatable Machine Code — *continued*

- (A) Program `SUBR` is compiled to produce object file (B), shown as both machine code and assembler.
- The compiler may start the compiled code at an arbitrary location, often location 1 as shown.
- Location 13 contains the machine code for the jump instruction to statement `ST` in location 5.



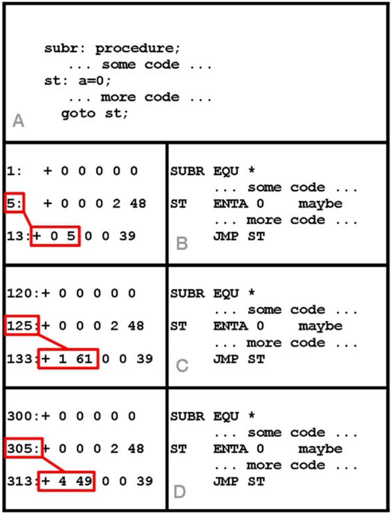
Relocatable Machine Code — *continued*

- (C) If `SUBR` is later linked with other code it may be stored at a location other than 1.
- In this example the linker places it at location 120.

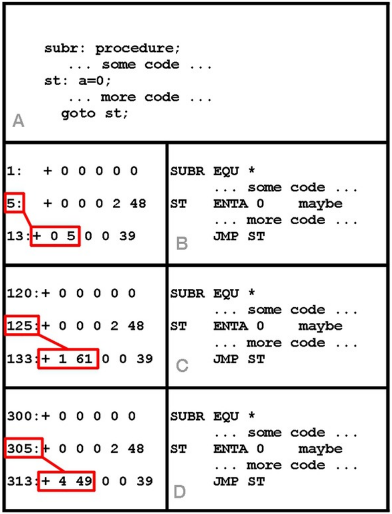


Relocatable Machine Code — *continued*

- The address in the jump instruction, which is now at location 133, must be relocated to point to the new location of the code for statement `ST`, now 125.
- `1 61` shown in the instruction is the MIX machine code representation of 125.

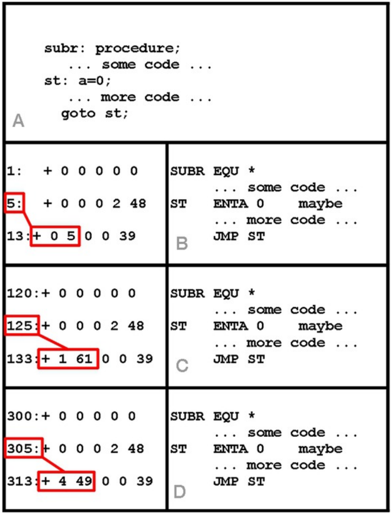


- (D) When the program is loaded into memory to run it may be loaded at some location other than the one assigned by the linker.
- This example shows `SUBR` now at location 300.



Relocatable Machine Code — *continued*

- The address in the jump instruction, now at 313, needs to be relocated again so that it points to the updated location of `ST`, 305.
- `4 49` is the MIX machine representation of 305.



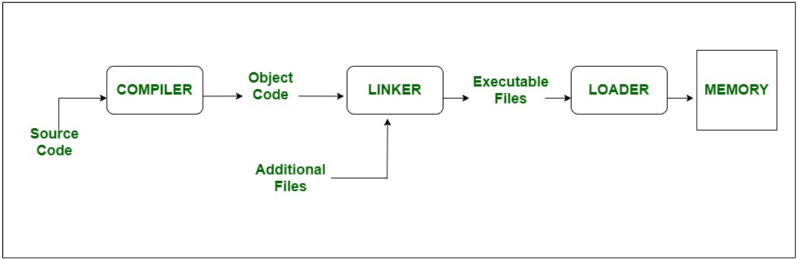
Difference between Linker and Loader

https://www.geeksforgeeks.org/difference-between-linker-and-loader/

Linker

- A linker is special program that combines the object files, generated by compiler/assembler, and other pieces of codes to originate an executable file.
- In the object file, linker searches and append all libraries needed for execution of file.
- It regulates memory space that code from each module will hold.
- It also merges two or more separate object programs and establishes link among them.

Difference between Linker and Loader — *continued*



Difference between Linker and Loader — *continued*

Loader

- The loader is special program that takes input of object code from linker, loads it to main memory, and prepares this code for execution by computer.
- Loader allocates memory space to program.
- Even it settles down symbolic reference between objects.
- It in charge of loading programs and libraries in operating system.

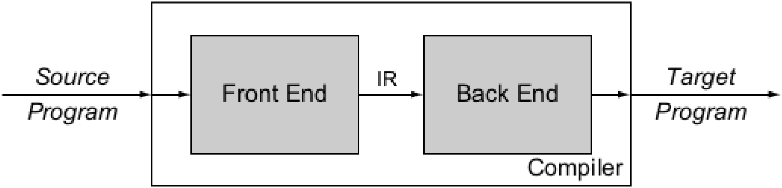
The Structure of a Compiler

- 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.
- It needs to understand the rules that govern syntax and meaning in the output language.
- Finally, it needs a scheme for mapping content from the source language to the target language.

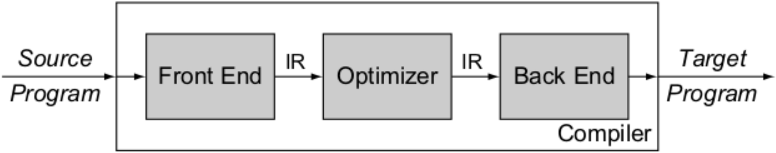
The Structure of a Compiler

- The structure of a typical compiler derives from these simple observations.
- The compiler has a front end to deal with the source language.
- It has a back end to deal with the target language.
- 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.
- To improve the translation, a compiler often includes an optimizer that analyzes and rewrites that intermediate form.

The Structure of a Compiler

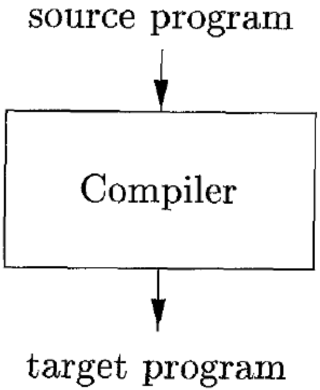


The Structure of a Compiler



The Structure of a Compiler

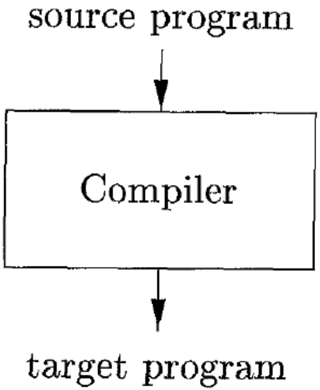
- Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.
- If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.



A compiler

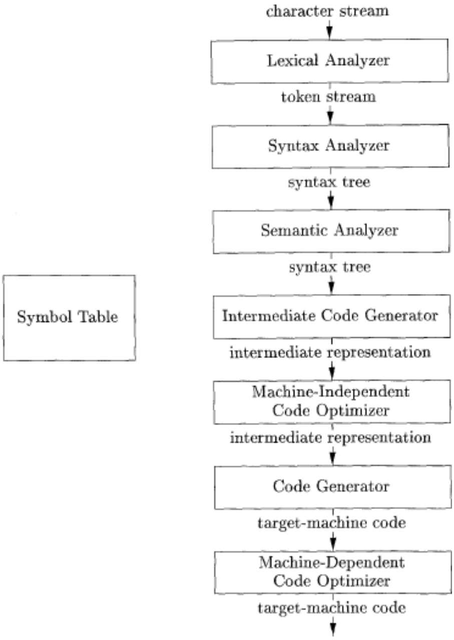
The Structure of a Compiler

- Up to this point we have treated a compiler as a single box that maps a source program into a semantically equivalent target program.
- If we open up this box a little, we see that there are two parts to this mapping: analysis and synthesis.



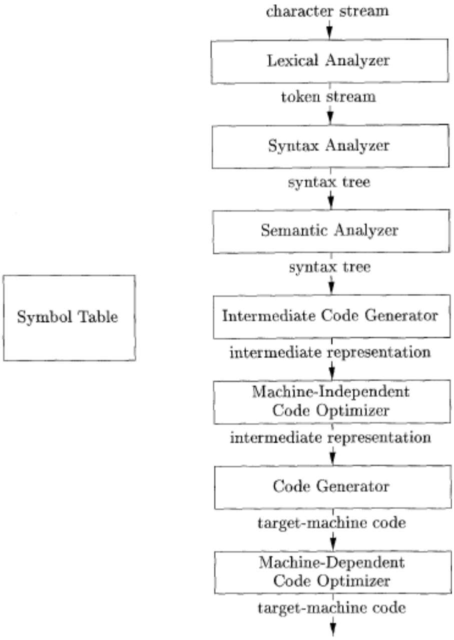
A compiler

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them.
- It then uses this structure to create an intermediate representation of the source program.
- If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action.



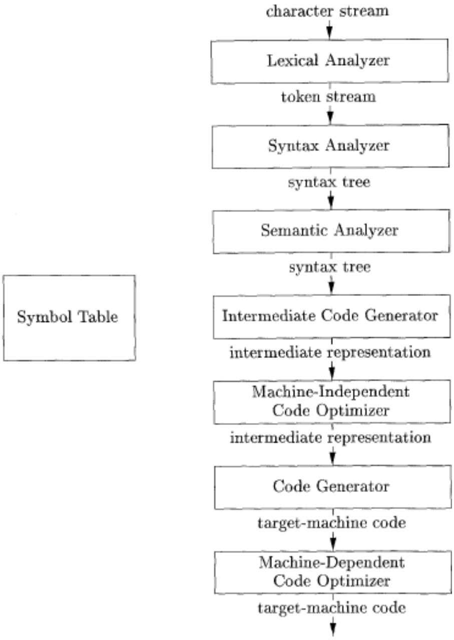
Phases of a compiler

- 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.
- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table.
- The analysis part is often called the front end of the compiler; the synthesis part is the back end.



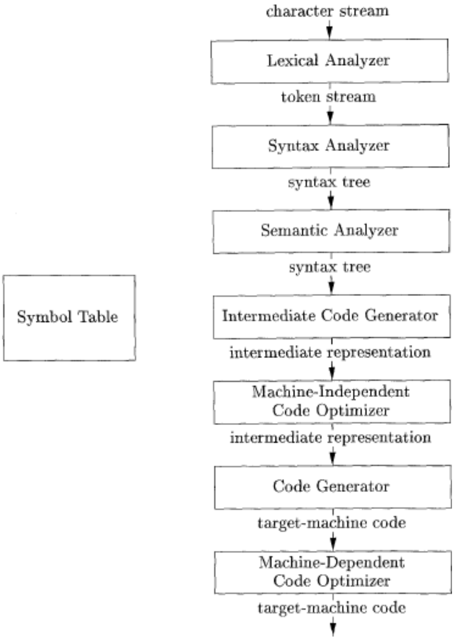
Phases of a compiler

- If we examine the compilation process in more detail, we see that it operates as a sequence of phases, each of which transforms one representation of the source program to another.
- In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly.



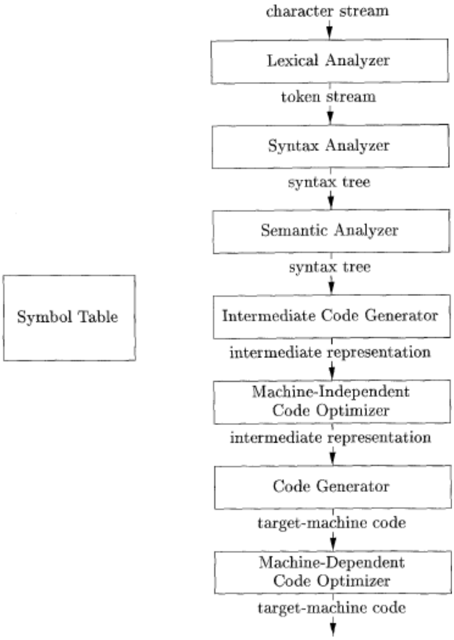
Phases of a compiler

- The symbol table, which stores information about the entire source program, is used by all phases of the compiler.



Phases of a compiler

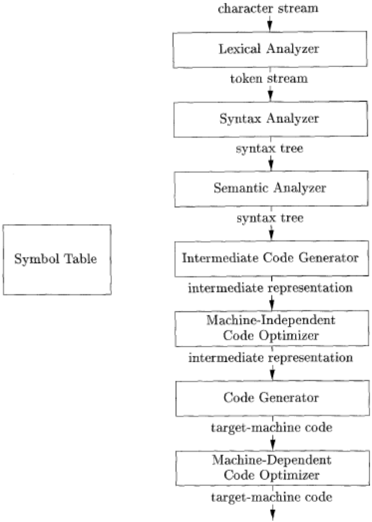
- Some compilers have a machine-independent optimization phase between the front end and the back end.
- The purpose of this optimization phase is to perform transformations on the intermediate representation, so that the back end can produce a better target program than it would have otherwise produced from an unoptimized intermediate representation.
- Since optimization is optional, one or the other of the two optimization phases shown may be missing.



Phases of a compiler

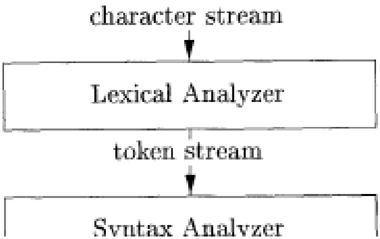
Lexical Analysis

- The first phase of a compiler is called lexical analysis or scanning.



Phases of a compiler

Lexical Analysis — *continued*



- The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes.

Lexical Analysis — *continued*

- For each lexeme, the lexical analyzer produces as output a token of the form

  $<$*token-name, attribute-value*$>$

  that it passes on to the subsequent phase, syntax analysis.
- In the token, 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.

Lexical Analysis — *continued*

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

- We consider a source program that contains the assignment statement
  `position = initial + rate * 60`
- The characters in this assignment could be grouped into the lexemes and mapped into the tokens passed on to the syntax analyzer

Lexical Analysis — *continued*

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

1. `position` is a lexeme that would be mapped into a token $\langle \mathbf{id}, 1 \rangle$.
   - **id** is an abstract symbol standing for identifier,
   - 1 points to the symbol-table entry for `position`,
   - The symbol-table entry for an identifier holds information about the identifier, such as its name and type.

Lexical Analysis — *continued*

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

2. The assignment symbol $=$ is a lexeme that is mapped into the token $\langle = \rangle$.
   - Since this token needs no attribute-value, we have omitted the second component.
   - We could have used any abstract symbol such as assign for the token-name, but for notational convenience we have chosen to use the lexeme itself as the name of the abstract symbol.

Lexical Analysis — *continued*

$\langle$*token-name*, *attribute-value*$\rangle$

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

3. `initial` is a lexeme that is mapped into the token $\langle \mathbf{id}, 2 \rangle$,
   - 2 points to the symbol-table entry for initial.

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

4. $+$ is a lexeme that is mapped into the token $\langle + \rangle$.

Lexical Analysis — *continued*

$\langle$*token-name*, *attribute-value*$\rangle$

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

5. `rate` is a lexeme that is mapped into the token $\langle \textbf{id}, 3 \rangle$, where 3 points to the symbol-table entry for `rate`.

Lexical Analysis — *continued*

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

6. $*$ is a lexeme that is mapped into the token $\langle * \rangle$.

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

7. `60` is a lexeme that is mapped into the token $\langle 60 \rangle$.

Lexical Analysis — *continued*

$$\langle \textit{token-name}, \textit{attribute-value} \rangle$$

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

- Blanks separating the lexemes would be discarded by the lexical analyzer.

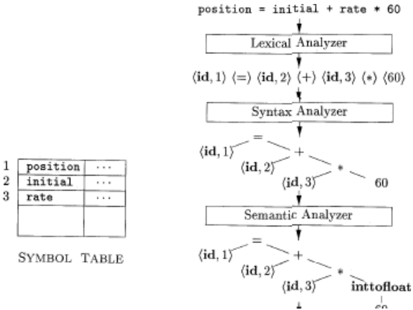
Lexical Analysis — *continued*

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

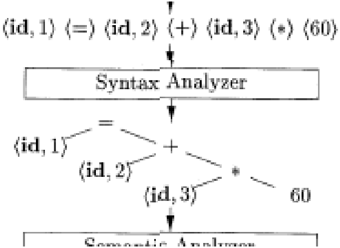
- Figure shows the representation of the assignment statement after lexical analysis as the sequence of tokens,

$$\langle \textbf{id}, 1\rangle \, \langle = \rangle \, \langle \textbf{id}, 2\rangle \, \langle + \rangle \, \langle \textbf{id}, 3\rangle \, \langle * \rangle \langle 60 \rangle$$

- In this representation, the token names $=$, $+$, and $*$ are abstract symbols for the assignment, addition, and multiplication operators, respectively.

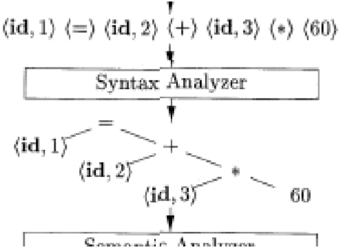


- The second phase of the compiler is syntax analysis or parsing.
- The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream.

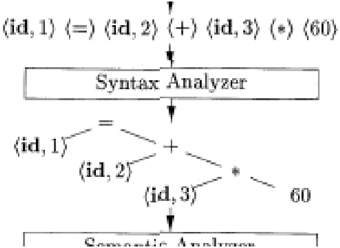


Syntax Analysis

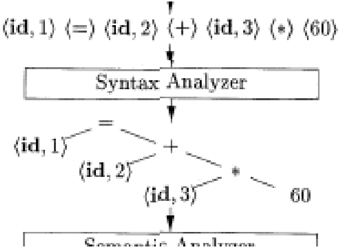
- A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.



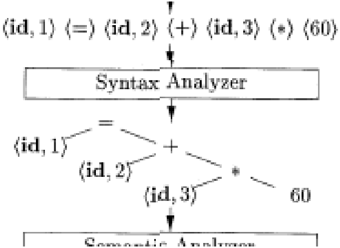
- This tree shows the order in which the operations in the assignment
  `position = initial + rate * 60`
  are to be performed.



- The tree has an interior node labeled $\star$ with $\langle \mathbf{id}, 3 \rangle$ as its left child and the integer 60 as its right child. The node $\langle \mathbf{id}, 3 \rangle$ represents the identifier `rate`.
- The node labeled $\star$ makes it explicit that we must first multiply the value of `rate` by `60`.

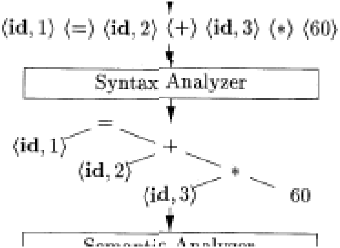


- The node labeled + indicates that we must add the result of this multiplication to the value of `initial`.

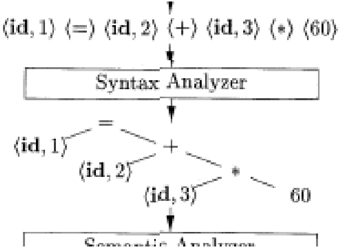


Syntax Analysis

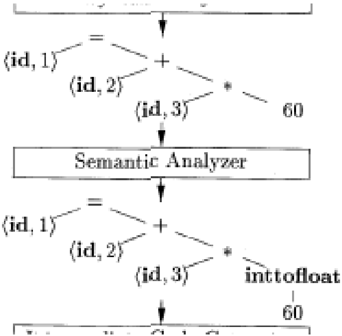
- The root of the tree, labeled =, indicates that we must store the result of this addition into the location for the identifier `position`.



- This ordering of operations is consistent with the usual conventions of arithmetic which tell us that multiplication has higher precedence than addition, and hence that the multiplication is to be performed before the addition.

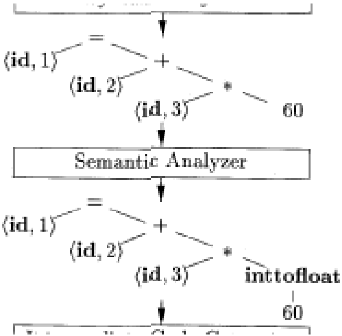


- The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition.
- It also gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.



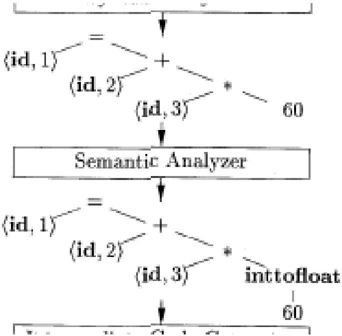
Semantic Analysis

- An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands.
- For example, many programming language definitions require an array index to be an integer.
- The compiler must report an error if a floating-point number is used to index an array.

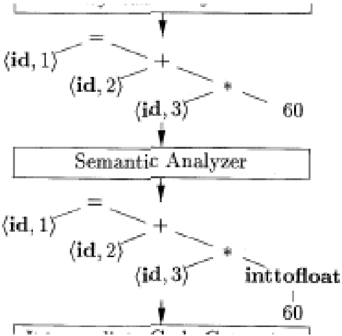


Semantic Analysis

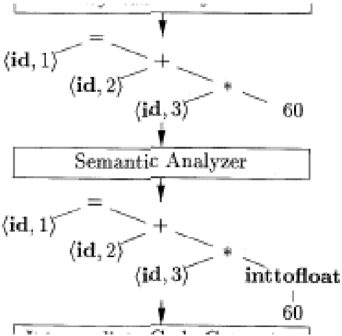
- The language specification may permit some type conversions called coercions.
- For example, a binary arithmetic operator may be applied to either a pair of integers or to a pair of floating-point numbers.
- If the operator is applied to a floating-point number and an integer, the compiler may convert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



- Suppose that `position`, `initial`, and `rate` have been declared to be floating-point numbers, and that the lexeme 60 by itself forms an integer.
- The type checker in the semantic analyzer discovers that the operator $*$ is applied to a floating-point number rate and an integer 60.
- In this case, the integer may be converted into a floating-point number.

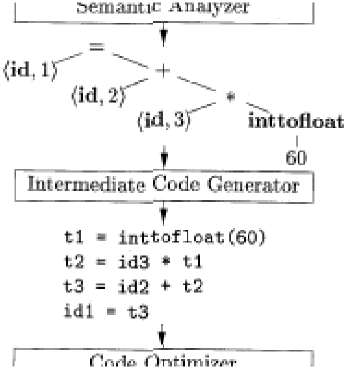


- Notice that the output of the semantic analyzer has an extra node for the operator `inttofloat`, which explicitly converts its integer argument into a floating-point number.



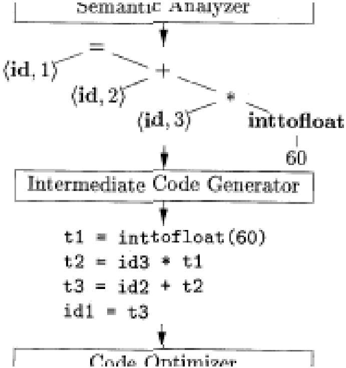
Intermediate Code Generation

- 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.
- Syntax trees are a form of intermediate representation.
- They are commonly used during syntax and semantic analysis.



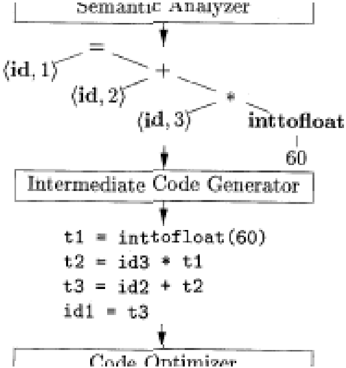
Intermediate Code Generation

- After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation, which we can think of as a program for an abstract machine.
- This intermediate representation should have two important properties:
  - it should be easy to produce and
  - it should be easy to translate into the target machine.



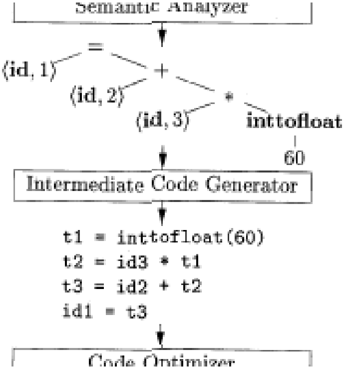
Intermediate Code Generation

- We consider an intermediate form called “three-address code”.
- Like the assembly language for a machine in which every memory location can act like a register.
- Three-address code consists of a sequence of instructions, each of which has at most three operands.

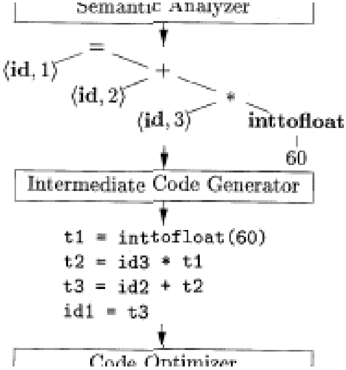


Intermediate Code Generation

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

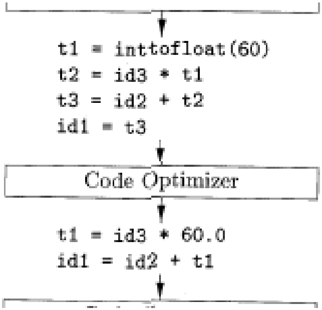


- Each three-address instruction has at most one operator in addition to the assignment.
  - Has to decide on the order in which operations are to be done.
  - Multiplication precedes the addition in the source program.
- Must generate a temporary name to hold the value computed by each instruction.
- Some "three-address" instructions have fewer than three operands.



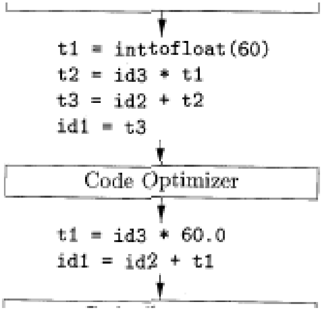
Code Optimization

- The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result.
- Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.



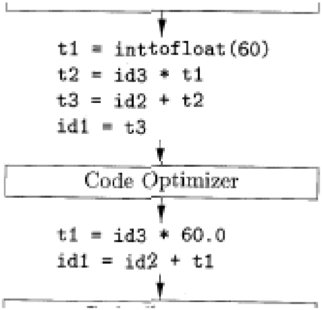
Code Optimization

- A simple intermediate code generation algorithm followed by code optimization is a reasonable way to generate good target code.
- 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 $\langle \textbf{id}, 1 \rangle$.



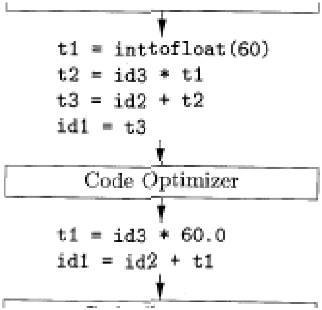
- So the optimizer can transform the code into the shorter sequence,

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



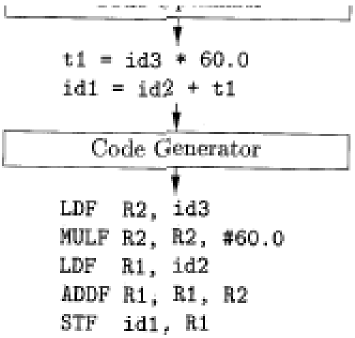
Code Optimization

- There is a great variation in the amount of code optimization different compilers perform.
- In those that do the most, the so-called "optimizing compilers," a significant amount of time is spent on this phase.
- There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much.



Code Generation

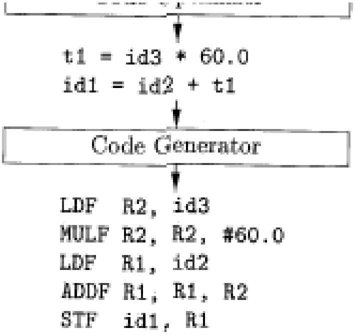
- The code generator takes as input an intermediate representation of the source program and maps it into the target language.
- If the target language is machine code, registers or memory locations are selected for each of the variables used by the program.
- Then, the intermediate instructions are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



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 Rl, id2
ADDF Rl, Rl, R2
STF idl, Rl
```



- 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.

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

SYMBOL TABLE

Symbol-Table Management

- 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

- 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 — *continued*

- 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.

The Grouping of Phases into Passes — *continued*

- 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.

The Grouping of Phases into Passes — *continued*

- 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.
- In addition to these general software-development tools, other more specialized tools have been created to help implement various phases of a compiler.

Compiler-Construction Tools — *continued*

- These tools use specialized languages for specifying and implementing specific components, and many use quite sophisticated algorithms.
- The most successful tools are those that hide the details of the generation algorithm and produce components that can be easily integrated into the remainder of the compiler.

Compiler-Construction Tools — *continued*

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 — *continued*

Some commonly used compiler-construction tools include

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.

End of Slides