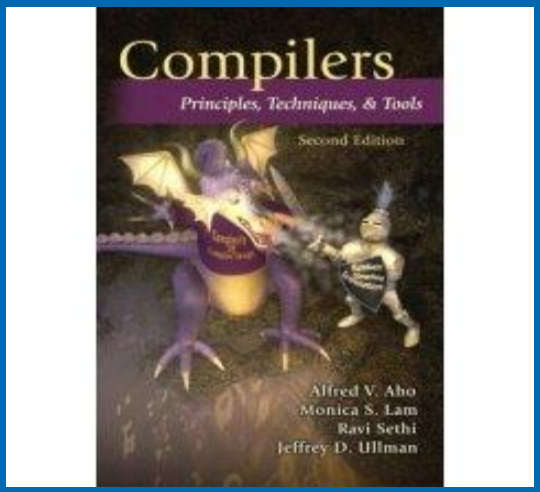
COMPILERS

Bibliography:

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman: Compilers: Princiles, Techniques, and Tools. Addison-Wesley 1986, ISBN 0-201-10088-6

Bibliography:

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compilers: Princiles, Techniques, and Tools. Addison-Wesley 2007, ISBN 0-321-48681-1

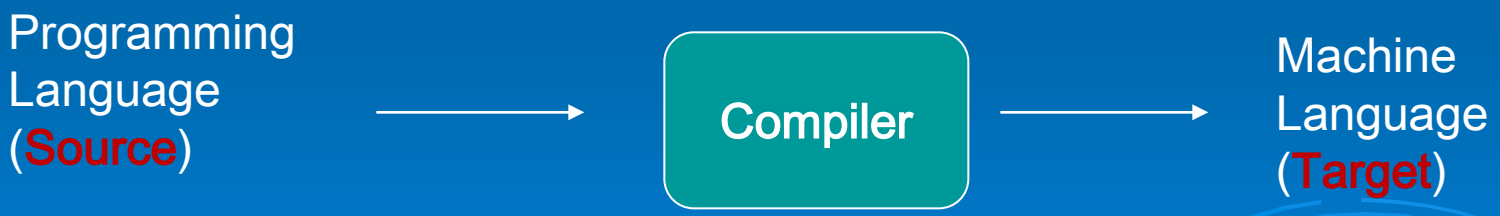

Compilers
Principles, Techniques, & Tools
Second Edition
Alfred V. Aho
Monica S. Lam
Ravi Sethi
Jeffrey D. Ullman

Compilers

- 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.

What Do Compilers Do

- A compiler acts as a translator, transforming human-oriented programming languages into computer-oriented machine languages.
- Ignore machine-dependent details for programmer



What Do Compilers Do

- Compilers may generate three types of code:
  - Pure Machine Code
    - Machine instruction set without assuming the existence of any operating system or library.
    - Mostly being OS or embedded applications.
  - Augmented Machine Code
    - Code with OS routines and runtime support routines.

What Do Compilers Do

- Compilers may generate three types of code:
  - Virtual Machine Code
    - Virtual instructions, can be run on any architecture with a virtual machine interpreter or a just-in-time compiler
    - Ex. Java

Compilers

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

What Do Compilers Do

- Another way that compilers differ from one another is in the format of the target machine code they generate:
  - Assembly or other source format
  - Relocatable binary
    - Relative address
    - A linkage step is required
  - Absolute binary
    - Absolute address
    - Can be executed directly

Compilers

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

Compilers

- 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

Compilers

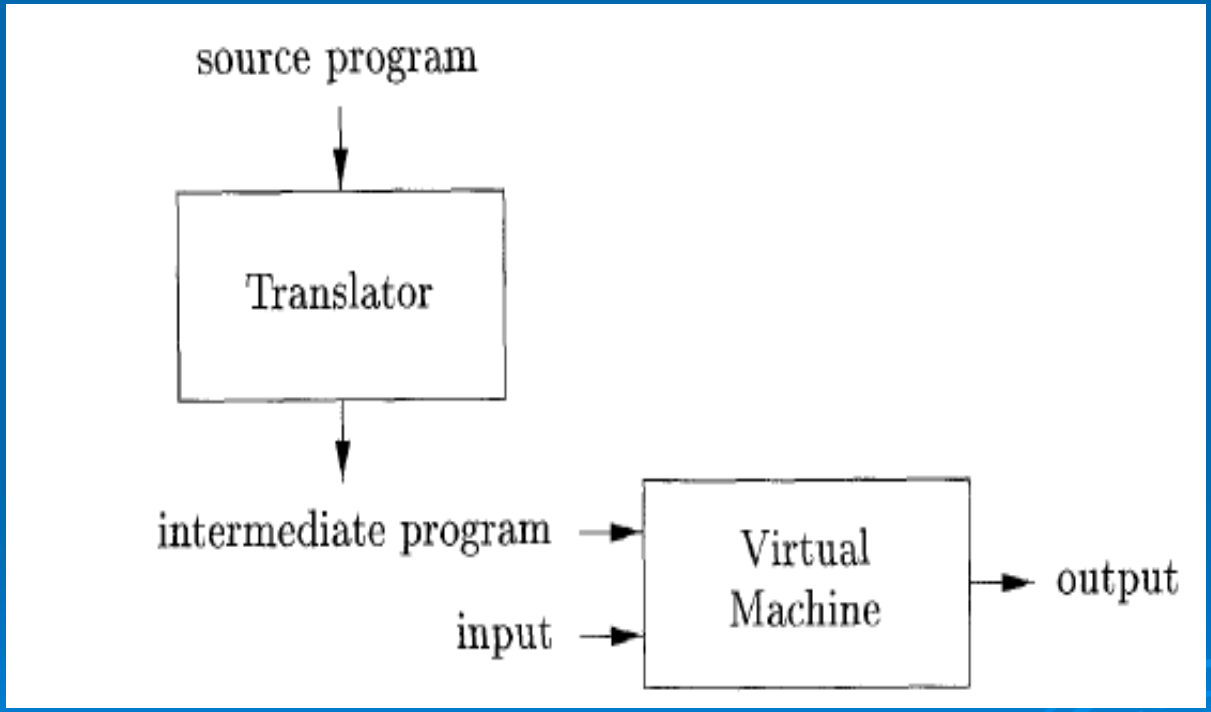
- The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs .
- An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

Compilers

- Java language processors combine compilation and interpretation.
- Java source program may first be compiled into an intermediate form called *bytecodes.* The bytecodes are then interpreted by *a* virtual machine.

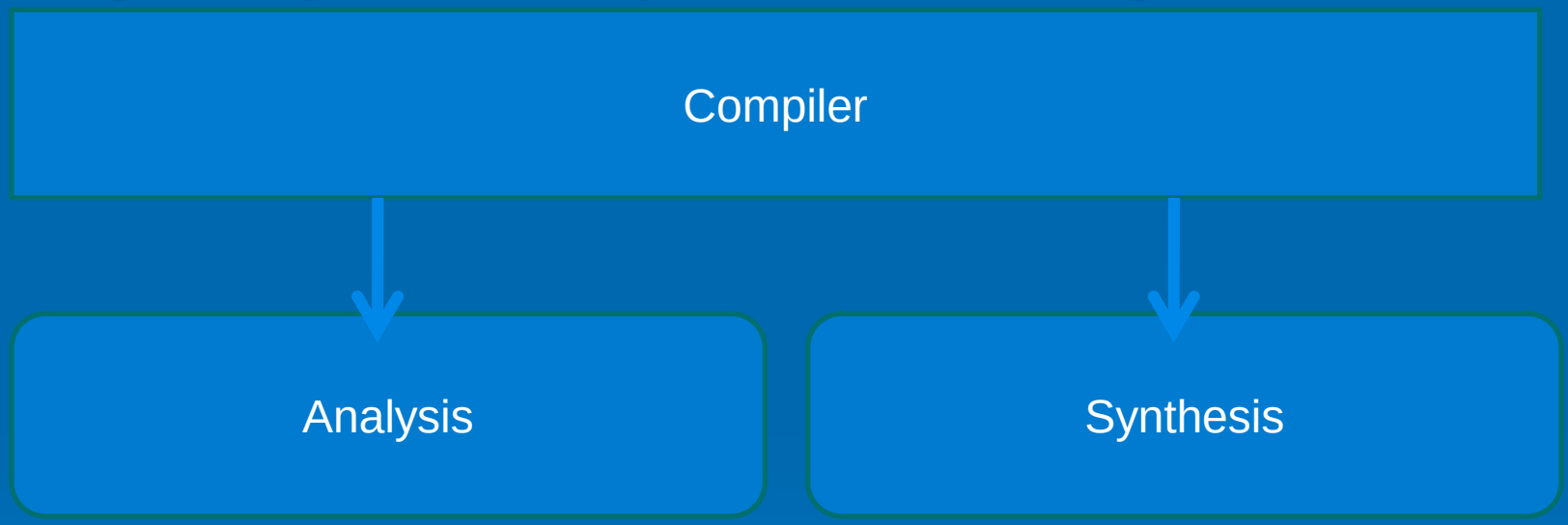
Compilers

- A hybrid compiler



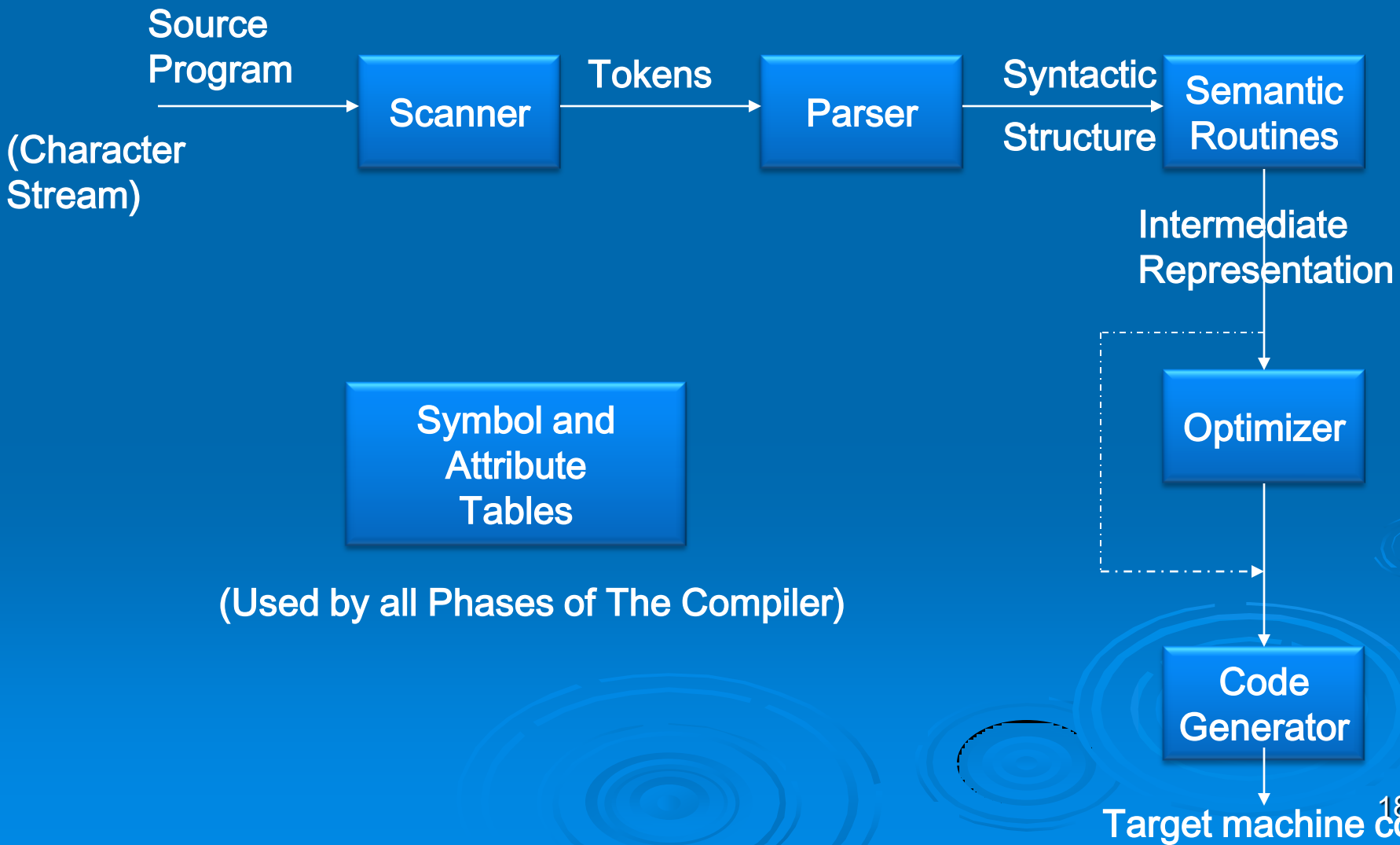
The Structure of a Compiler

- Any compiler must perform two major tasks



- ***Analysis*** of the source program
- ***Synthesis*** of a machine-language program

The Structure of a Compiler



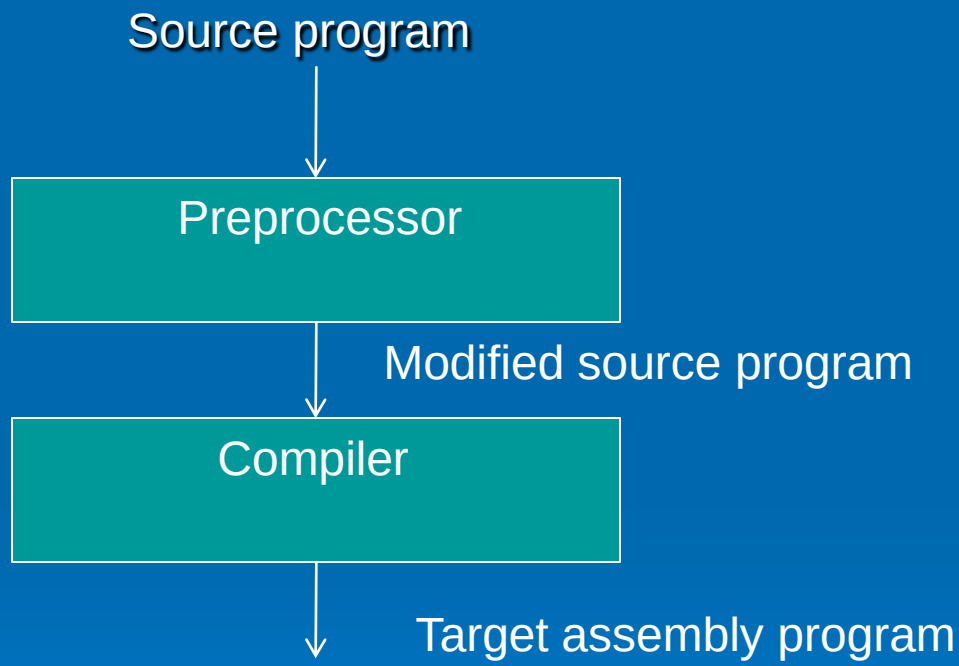
Compilers

- In addition to a compiler, several other programs may be required to create an executable target program.

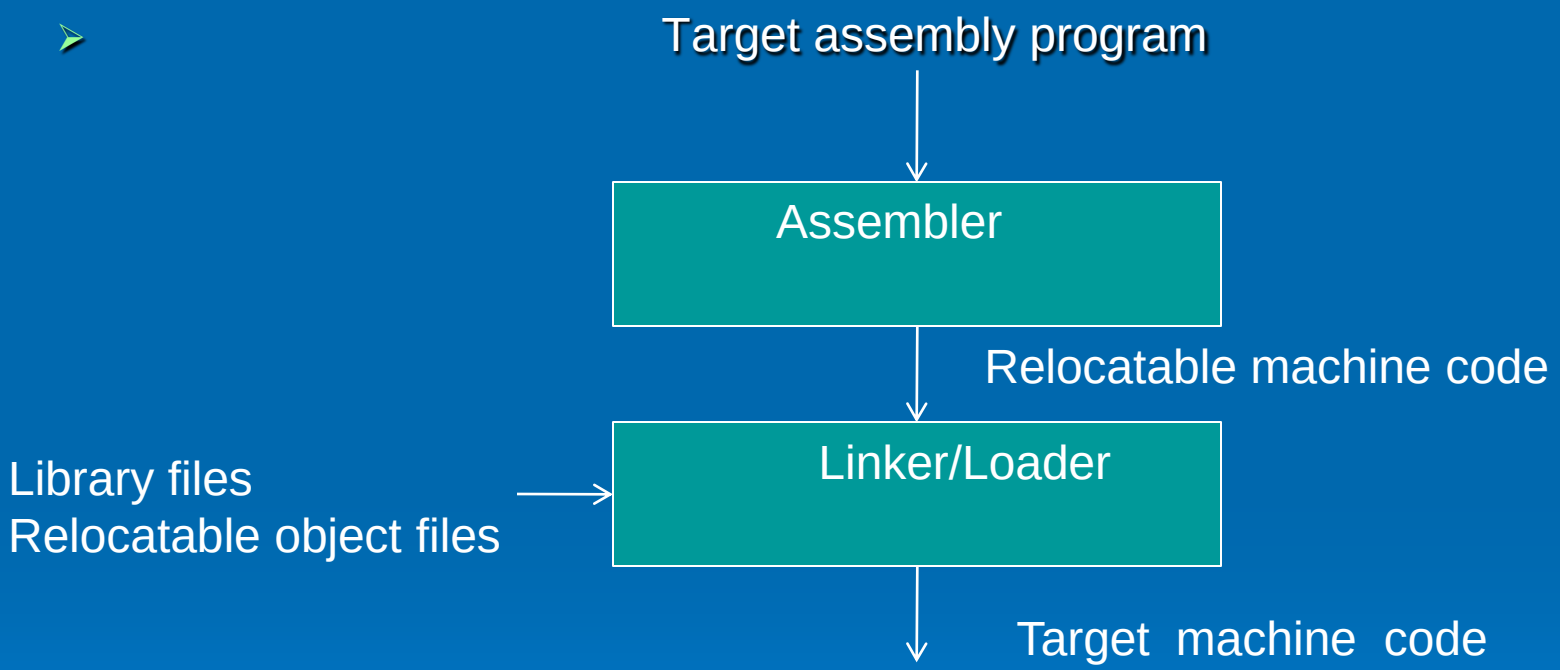
Compilers

- 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.

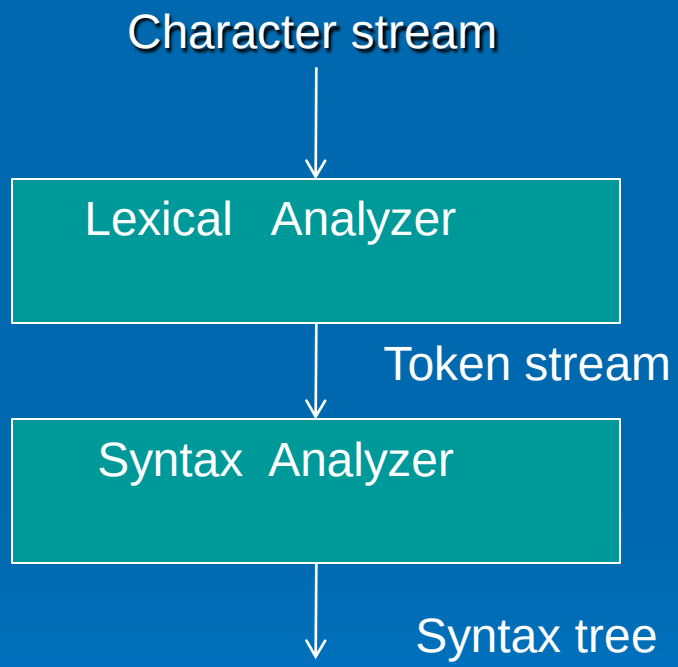
A language-processing system



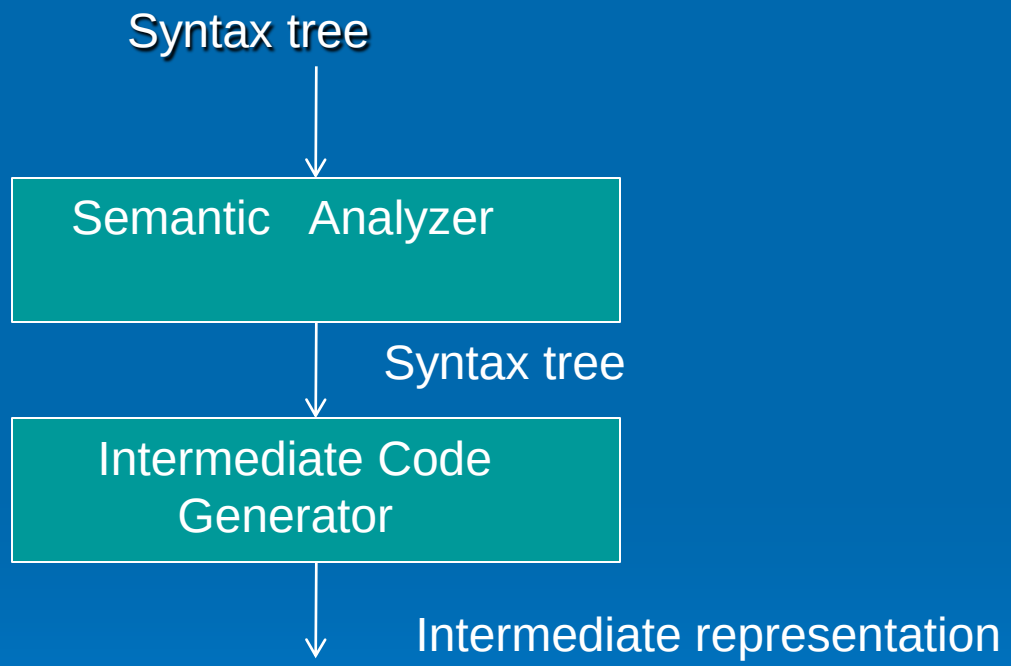
A language-processing system



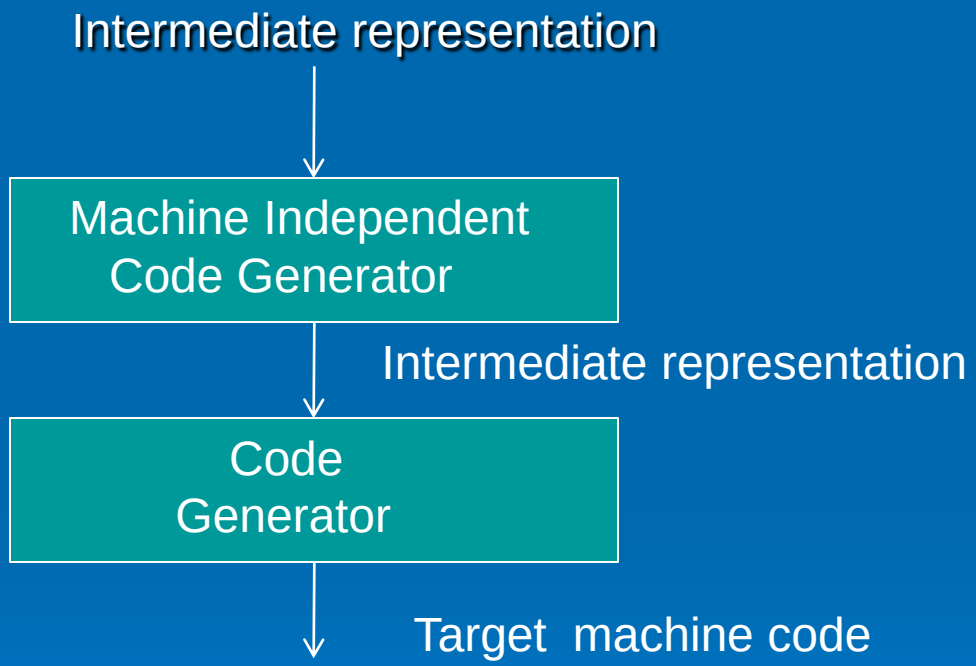
Phases of a compiler



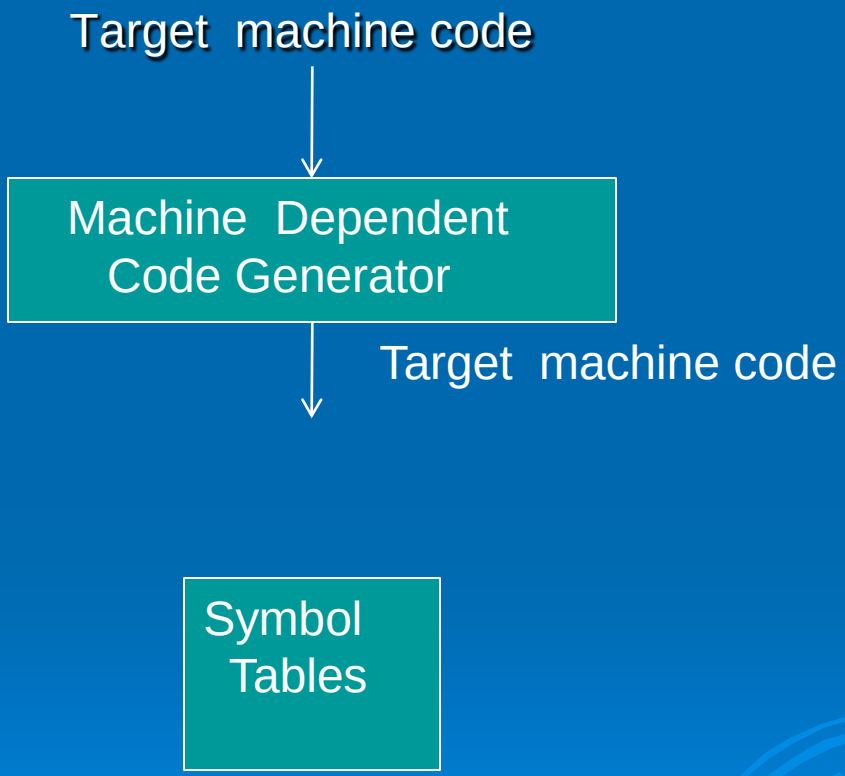
Phases of a compiler



Phases of a compiler



Phases of a compiler



Lexical Analysis

- 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>

Lexical Analysis

- For example, suppose a source program contains the assignment statement

  position = i n i t i a l + r a t e * 60

- The characters in this assignment could be grouped into the following lexemes and mapped into the following tokens passed on to the syntax analyzer:

Lexical Analysis

- 1. position is a lexeme that would be mapped into a token <id, 1> where id

  is an abstract symbol standing for identifier and 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

- 2. The assignment symbol = is a lexeme that is mapped into the token < = >.
- Since this token needs no attribute-value, we have omitted the second component.

Lexical Analysis

- 3. initial is a lexeme that is mapped into the token <id, 2>, where 2 points to the symbol-table entry for initial

Lexical Analysis

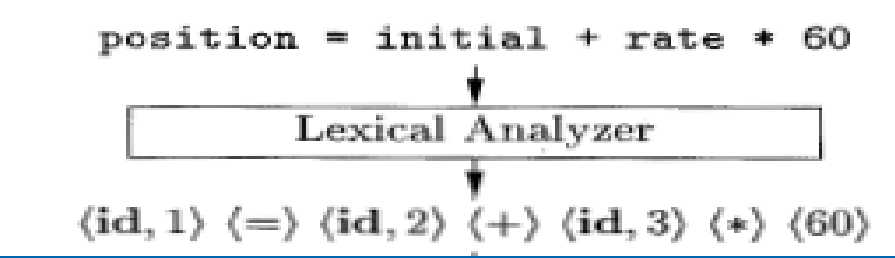
- 4. + is a lexeme that is mapped into the token <+>.
- 5. rate is a lexeme that is mapped into the token <id, 3>, where 3 points to the symbol-table entry for rate .

Lexical Analysis

- 6. * is a lexeme that is mapped into the token <*> .
- 7. 60 is a lexeme that is mapped into the token <60>

Lexical Analysis

- After lexical analysis, we have



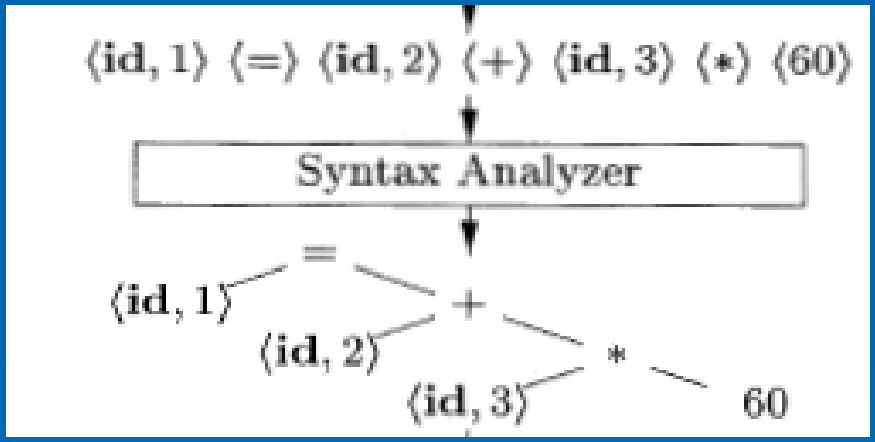
Syntax Analysis

- The parser uses the first components of the tokens produced by the lexical analyzer to create an 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.

Syntax Analysis



Semantic Analysis

- 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.

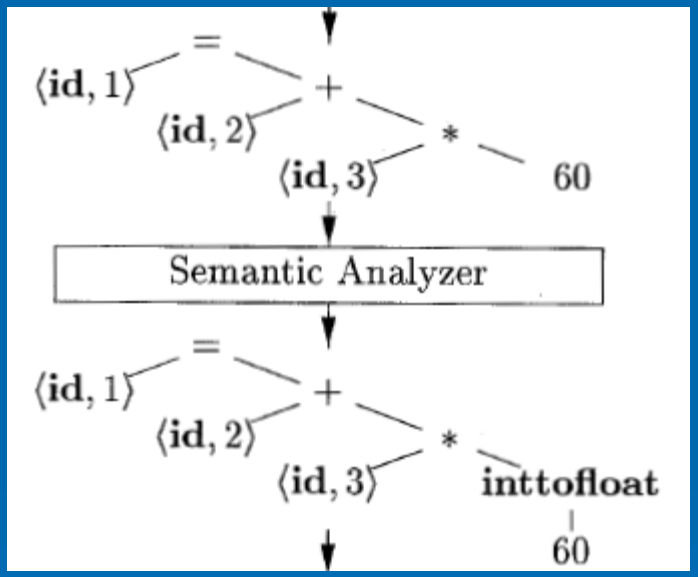
Semantic Analysis

- 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.

Semantic Analysis

- In the Figure on the next slide, the operator *inttofloat*, which explicitly converts its integer argument into a floating-point number.

Semantic Analysis



Intermediate Code Genaration

- 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 Genaration

- An intermediate form, called three-address code, consists of a sequence of assembly-like instructions with three operands

  per instruction:

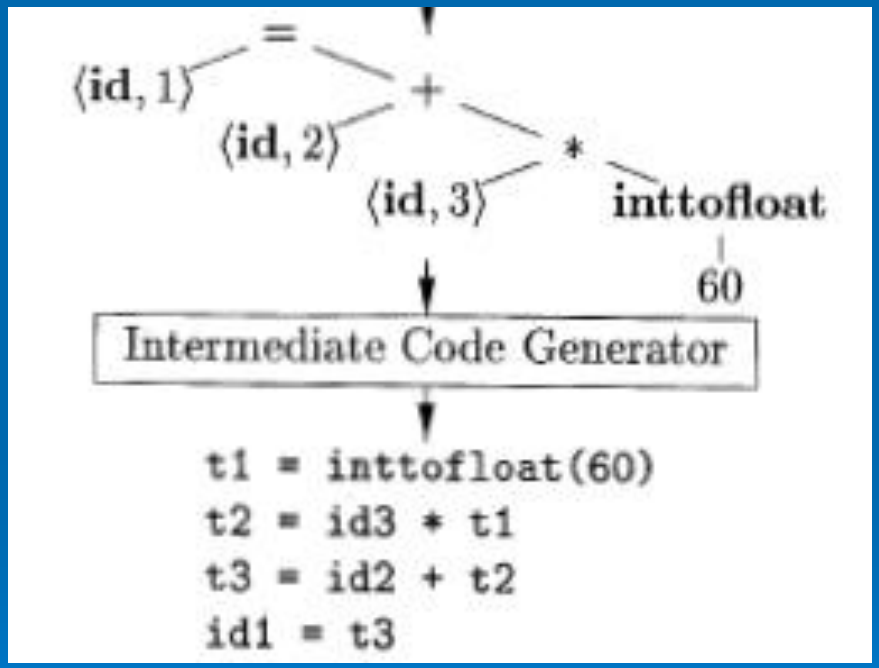
  t1 = i n t t o f l o a t (60)

  t 2 = id3 * t1

  t 3 = id2 + t 2

  id1 = t 3

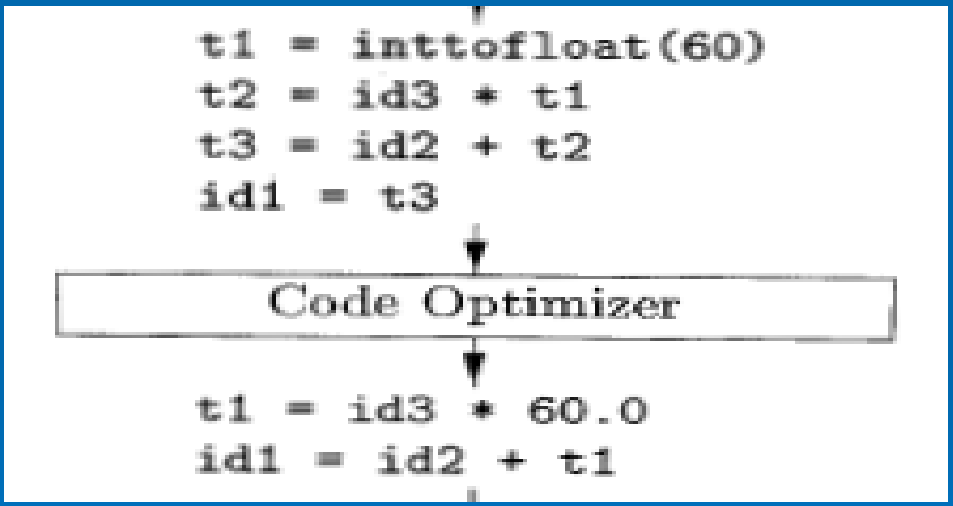
Intermediate Code Genaration



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



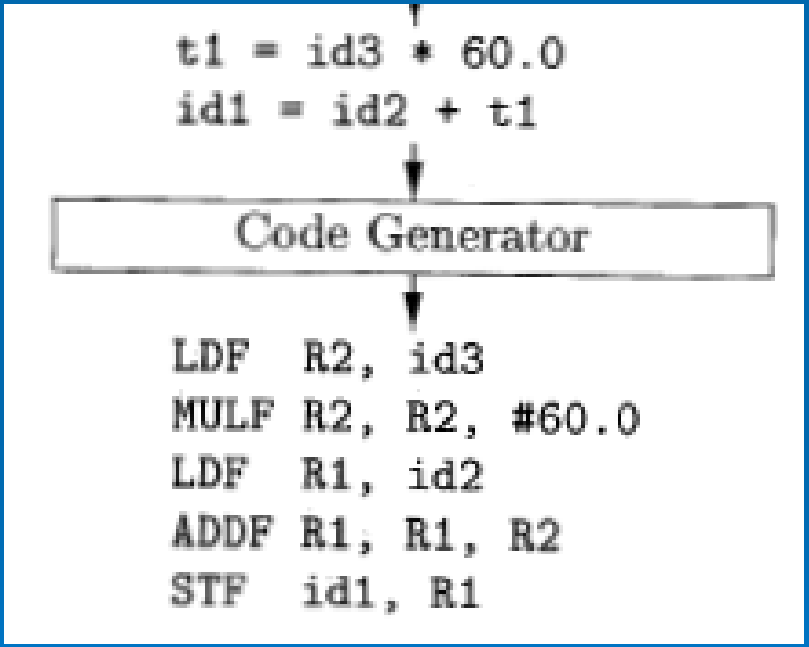
Code Generation

- 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.

Code Generation

- 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



Symbol Table

- 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.

Grouping 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.

Grouping Phases into Passes

- 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.

Thank you