Compiler Design

Dr. Sahar kamal

Agenda

1. Introduction to Compiler Design
2. **Lexical Analysis (Scanning)**
3. Regular expression
4. Finite automata
5. **Syntax Analysis (Parsing)**
6. Top-Down Parsing
7. Bottom-Up Parsing
8. **Semantic analyzer**
9. **Intermediate Code Generation**
10. **Code Optimization**
11. **Code Generation**

Grading

Grade system	Grade
Attendance	2
Reports / Sheets	2
Quiz 1 / Quiz 2	12
Mid-term Exam	24
Final Exam	60

Introduction to Compiler Design

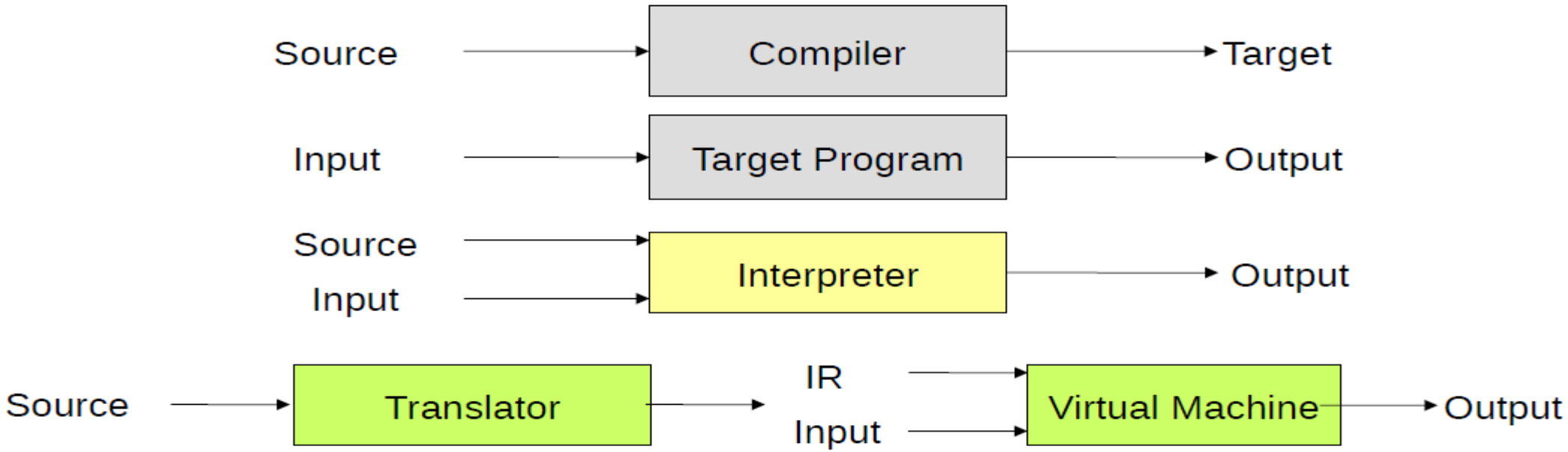
Lecture 1

Compilers *Principles, Techniques, & Tools*

Second Edition

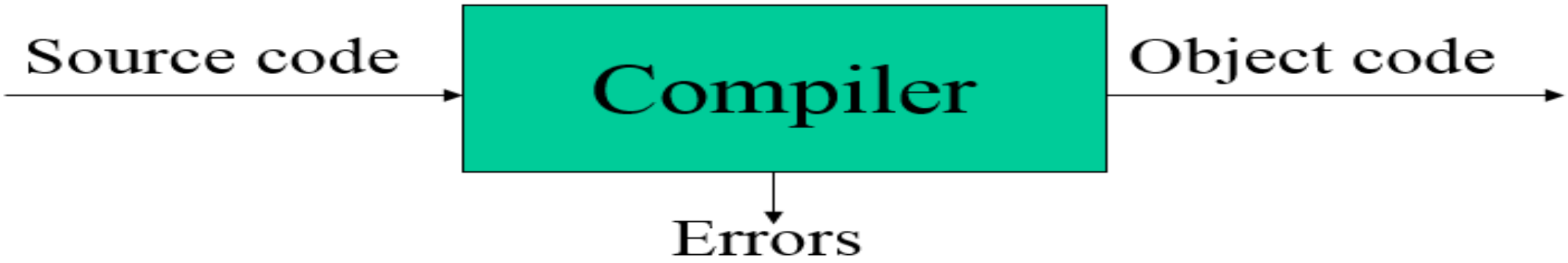
Introduction of Compiler Design

- Language Processors



Introduction of Compiler Design

- **Compiler** is a translator program that reads a program written in one language -the source language- and translates it into an equivalent program in another language-the target language. As an important part of this translation process, the compiler reports to its user the presence of errors in the source program.

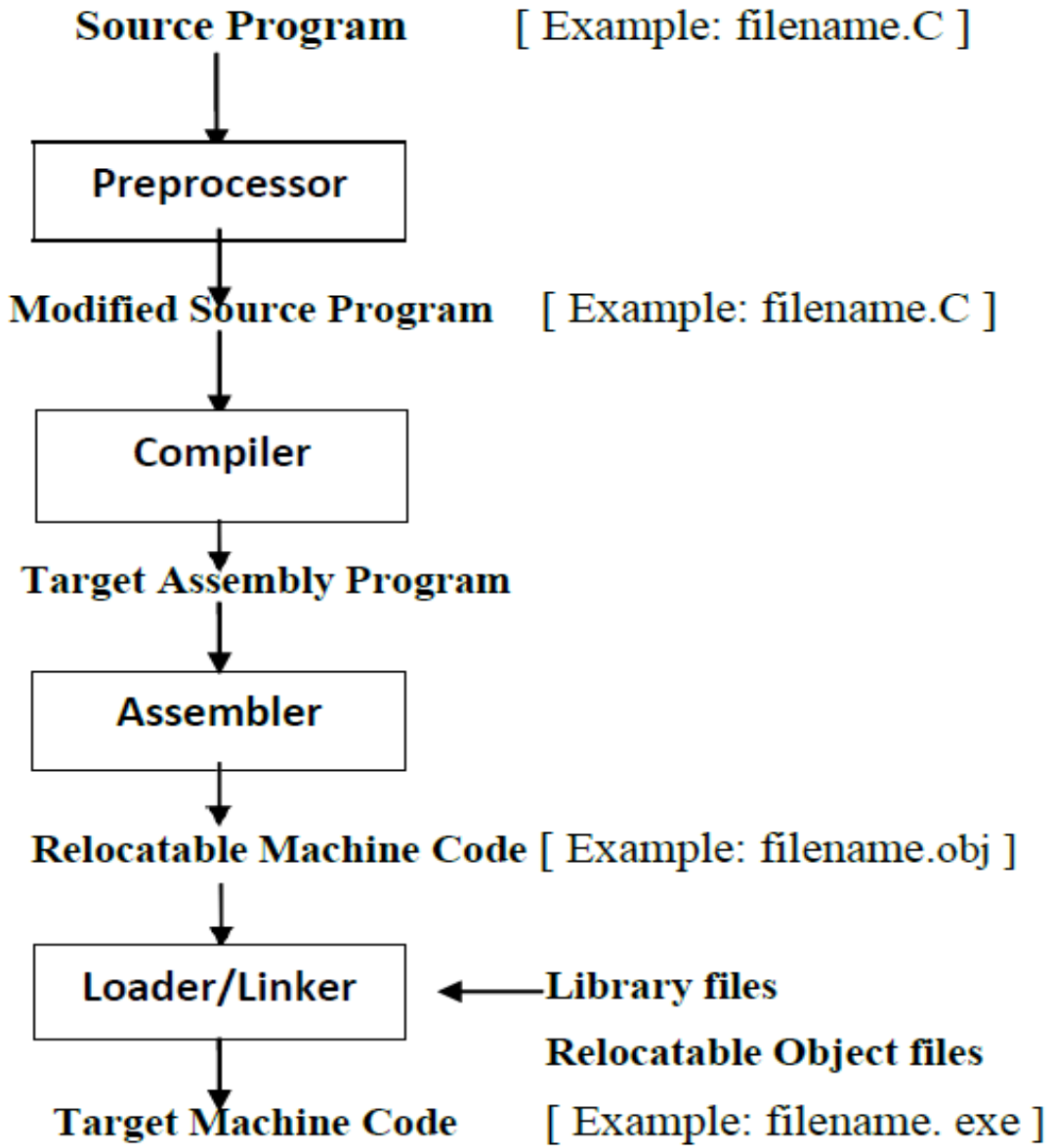


Introduction of Compiler Design

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

Interpreter: An interpreter is another commonly used language processor. Instead of producing a target program as a single translation unit, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user.

Language Processing System



Language Processing System

- **A LANGUAGE-PROCESSING SYSTEM:**
- **Preprocessors:** Preprocessors produce input to compilers. They may perform the following functions:

1. 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.
2. The preprocessor may also expand short hands, called macros, into source language statements.

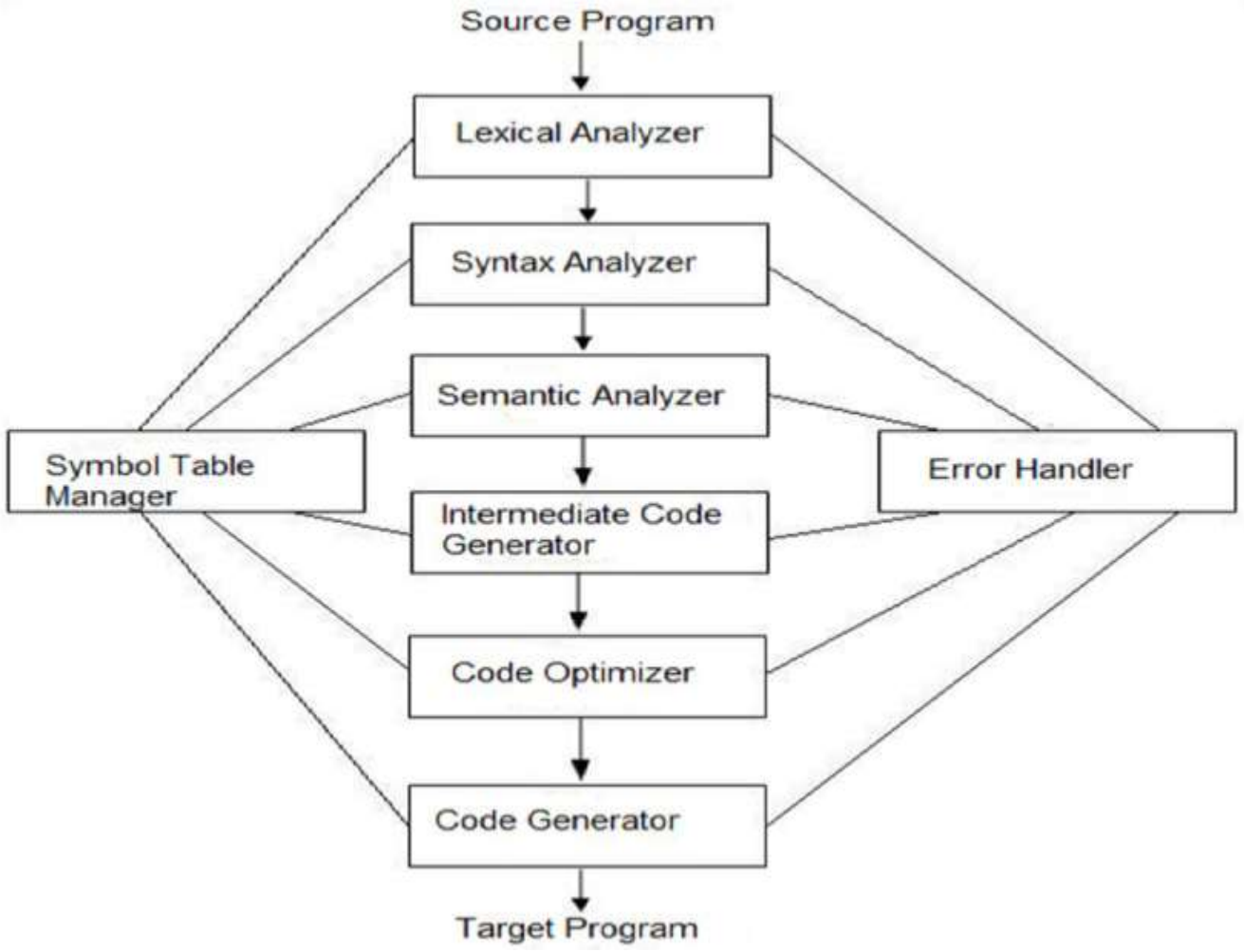
Language Processing System

- **Assemblers:**
- Some compilers produce assembly code, which is passed to an assembler for producing a relocatable machine code that is passed directly to the loader/linker editor. The assembly code is the mnemonic version of machine code. A typical sequence of assembly code is:
- MOV a, R1
- ADD #2, R1
- MOV R1, b

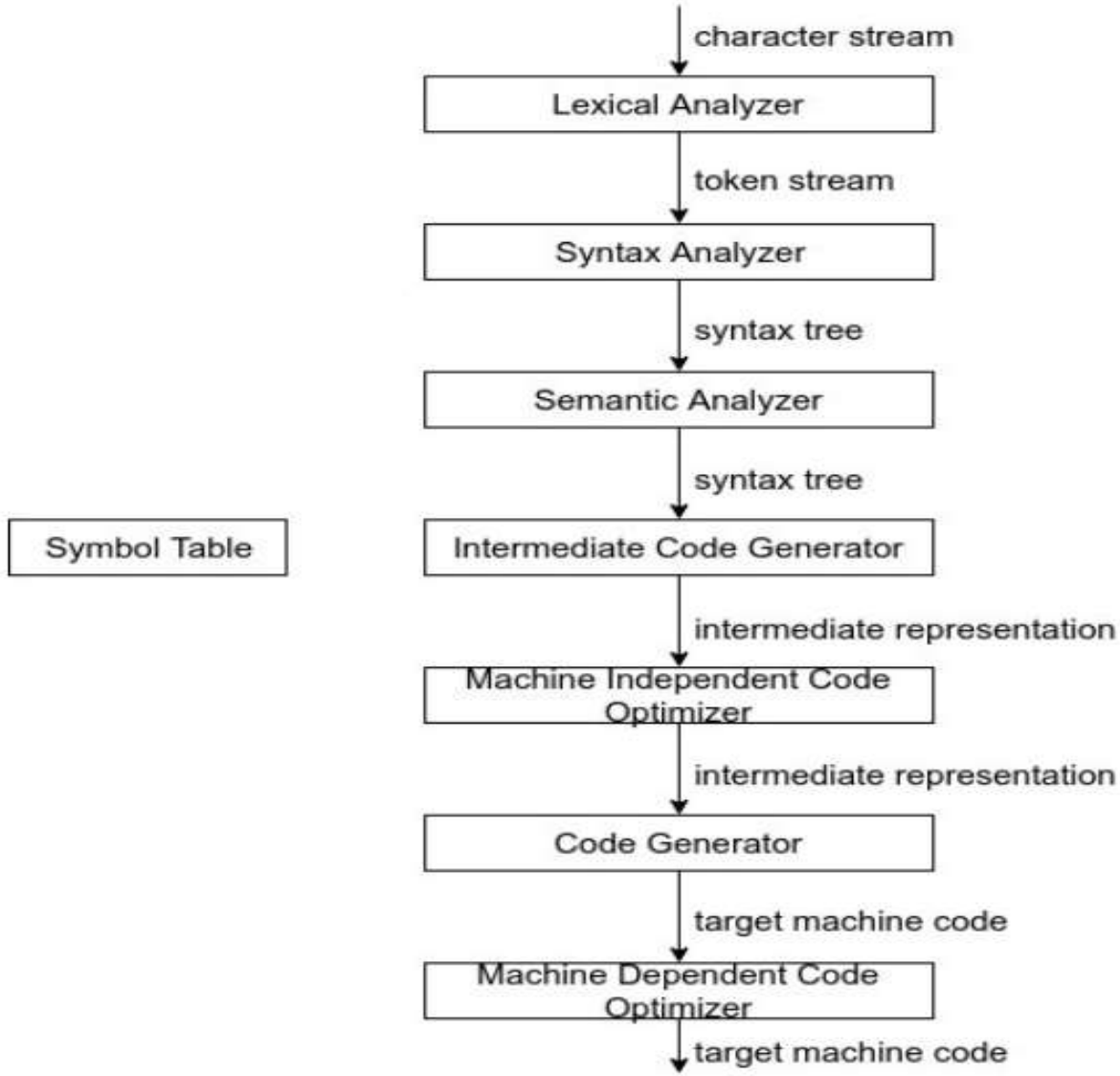
- **Loaders and Linker-Editors:**
- Large programs are often compiled in pieces, so the relocatable machine code may have to be linked together with other relocatable object files that actually runs on the machine
- **The link-editor** allows making a single program from several files of relocatable machine code
- **A loader** program performs two functions namely, loading and link-editing. The process of loading consists of taking relocatable machine code, altering the relocatable addresses and placing the alters instructions and data in memory at the proper locations.

Structure Of A Compiler

- There are two parts to compilation: analysis and synthesis. The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program. The synthesis part constructs the desired target program from the intermediate representation.
- The structure of compiler is shown in Fig. The first three phases form the analysis portion of the compiler and rest of the phases form the synthesis phase.



The structure of a Compiler



Phases Of A Compiler

1. Lexical Analysis:

- In a compiler, linear analysis is called lexical analysis or scanning.
- It is a first phase of a compiler
- Lexical Analyzer is also known as **scanner**
- Reads the characters in the source program from left to right and groups the characters into stream of **Tokens**.
- Such as Identifier, Keyword, Punctuation character, Operator.
- **Pattern:** Rule for a set of string in the input for which a token is produced as output.
- A **Lexeme** is a sequence of characters in the source code that is matched by the Pattern for a Token.

Lexical Analysis

- Called scanning
- in character stream, out: lexemes, tokens
  <token-name, attribute-value>
  attribute-value: points to an entry in symbol table
- Ex: position = initial + rate * 60
  <id, 1> <=> <id, 2> <+> <id, 3> <*> <60>

position	...
initial	...
rate	...

Syntax Analysis

- Syntax Analysis
- Hierarchical analysis is called parsing or syntax analysis. It involves grouping the tokens of the source program into grammatical phrases that are used by the compiler to synthesize the output.

- This phase is called the syntax analysis or **Parsing**.
- It takes the token produced by lexical analysis as input and generates a parse tree.
- In this phase, token arrangements are checked against the source code grammar.
- i.e. the parser checks if the expression made by the tokens is syntactically correct.

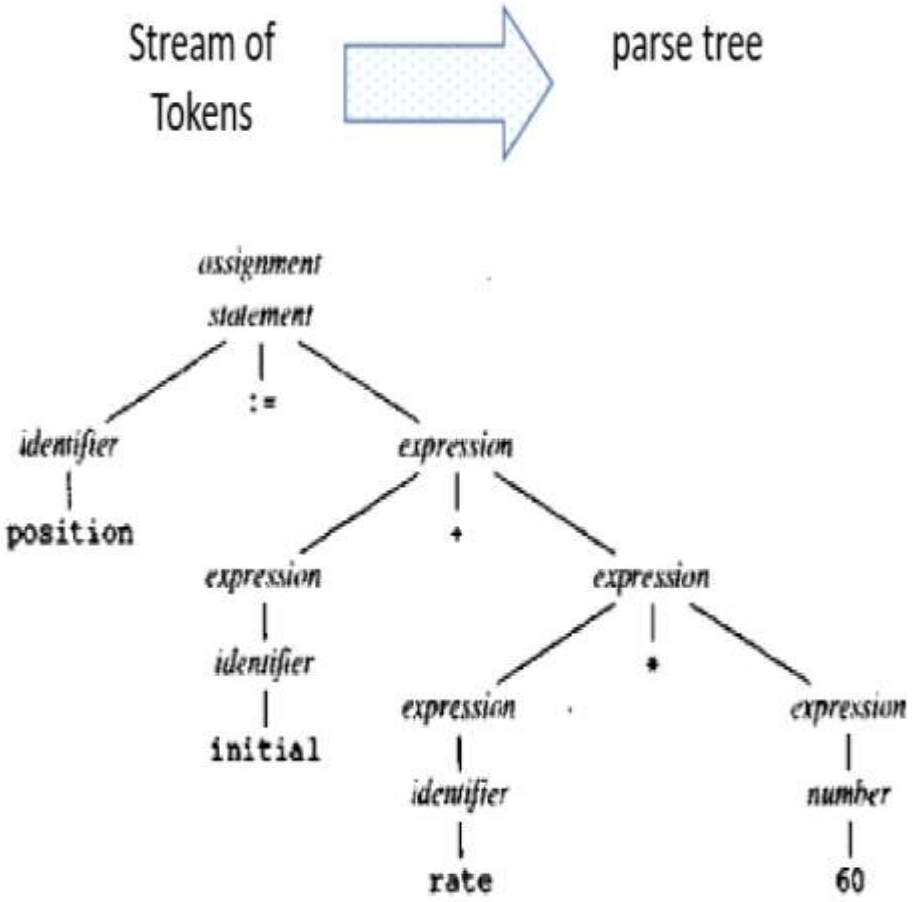


Fig. Parse tree for position = initial + rate * 60

Semantic Analysis

2. Semantic Analysis

• The semantic analysis phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.

- An important component of semantic analysis is type checking. i.e .whether the operands are type compatible. For example, a real number used to index an array.

Fig. Semantic analysis inserts a conversion from integer to real

Intermediate Code Generation

- **Intermediate Code Generation:**
- After semantic analysis, some compilers generate an explicit intermediate representation of the source program. This representation should be easy to produce and easy to translate into the target program. There are varieties of forms.
- Three address code
- Postfix notation
- Syntax Tree

- The commonly used representation is three address formats. The format consists of a sequence of instructions, each of which has at most three operands. The IR code for the given input is as follows:
- temp1 = inttofloat (60)
- temp2 = id3 * temp1
- temp3 = id2 + temp2
- id1 = temp3

Code Optimization

- This phase attempts to improve the intermediate code, so that faster running machine code will result.

- There is a better way to perform the same calculation for the above three address code ,which is given as follows:
- temp1 = id3 * 60.0
- id1 = id2 + temp1

- There are various techniques used by most of the optimizing compilers, such as:

1. Common sub-expression elimination
2. Dead Code elimination
3. Constant folding
4. Copy propagation
5. Induction variable elimination
6. Code motion
7. Reduction in strength. ..... etc..

Code Generation

- **Code Generation:**
- The final phase of the compiler is the generation of target code, consisting of relocatable machine code or assembly code.
- The intermediate instructions are each translated into sequence of machine instructions that perform the same task. A crucial aspect is the assignment of variables to registers.

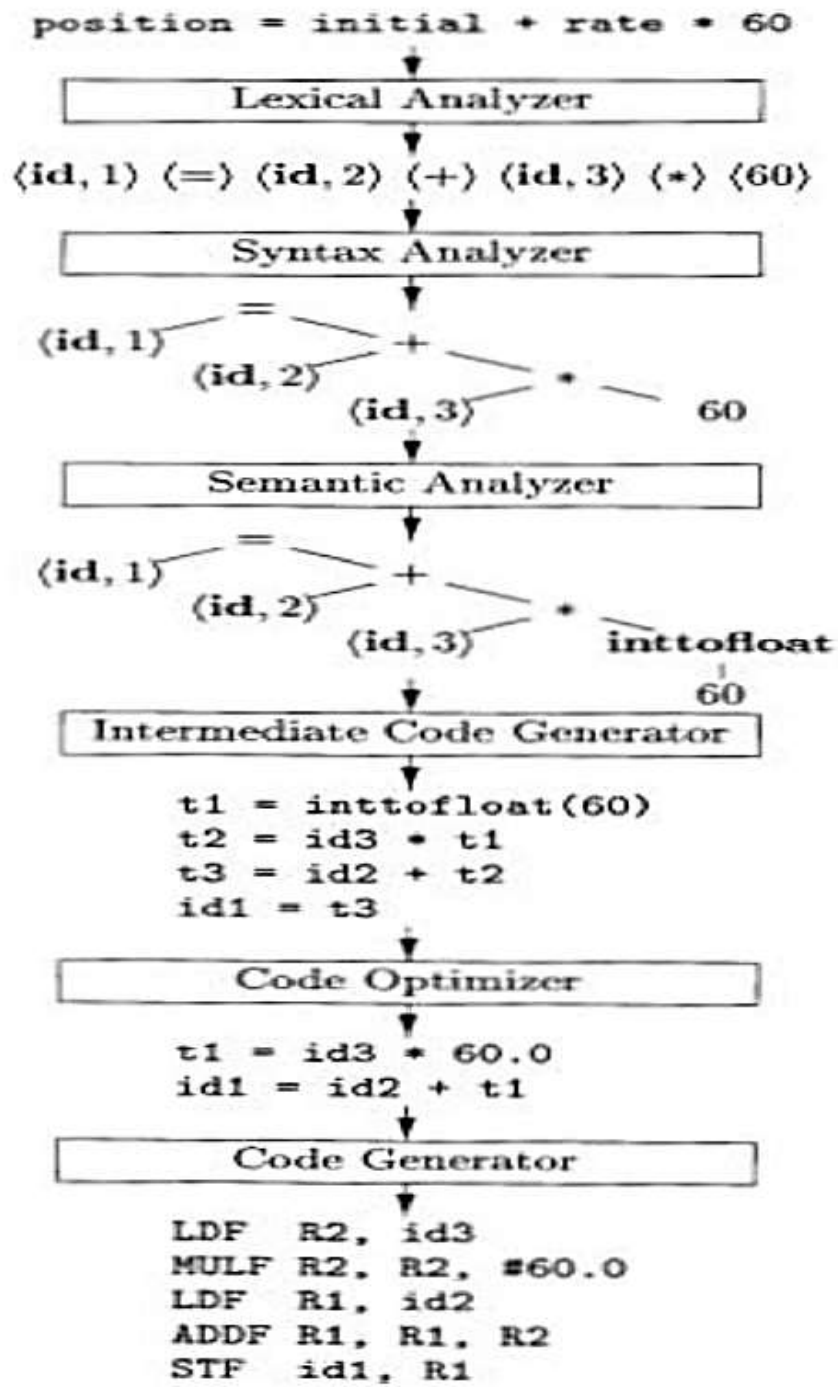
- Using registers R1 and R2,the translation of the given example is:

- MOV id3 ,R2
- MUL #60.0 , R2
- MOV id2 , R1
- ADD R2 , R1
- MOV R1 , id1

Symbol-Table Management & Error Handling and Reporting

- **Symbol-Table Management:**
- An essential function of a compiler is to record the identifiers used in the source program and collect its information.
- A symbol table is a data structure containing a record for each identifier with fields for attributes.(such as, its type, its scope, if procedure names then the number and type of arguments etc.,)
- The data structure allows finding the record for each identifier and store or retrieving
- data from that record quickly.

- **Error Handling and Reporting:**
- Each phase can encounter errors. After detecting an error, a phase must deal that error, so that compilation can proceed, allowing further errors to be detected.
- Lexical phase can detect error where the characters remaining in the input do not form any token.
- The syntax and semantic phases handle large fraction of errors. The stream of tokens violates the syntax rules are determined by syntax phase.
- During semantic, the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved. e.g., if we try to add two identifiers ,one is an array name and the other a procedure name.


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 Generator
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
Code Optimizer
t1 = id3 * 60.0
id1 = id2 + t1
Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1

Programming Languages Evolution

- 1st generation :machine language
- 2nd generation: assembly
- 3rd generation: high level (Fortran, Cobol, Lisp, C, C++, C#, Java)
- 4th generation: SQL
- 5th generation: Prolog
- OOP languages
- scripting languages: interpreted, gluing