St. Joseph's Institute of Technology
St. Joseph's College of Engineering
Department of CSE/IT
COMPILER DESIGN

UNIT I INTRODUCTION TO COMPILERS

Translators-Compilation and Interpretation-Language processors -The Phases of Compiler-Errors Encountered in Different Phases-The Grouping of Phases-Compiler Construction Tools - Programming Language basics.

COMPLIER

Complier is a 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. In this translation process, the complier reports to its user the presence of the errors in the source program.

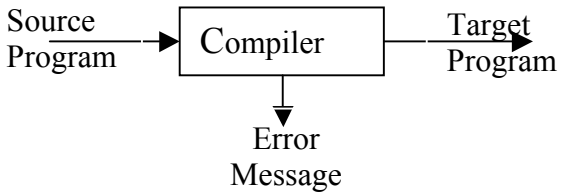
The classifications of compiler are: Single-pass compiler,Multi-pass compiler, Load and go compiler, Debugging compiler, Optimizing compiler.

INTERPRETER

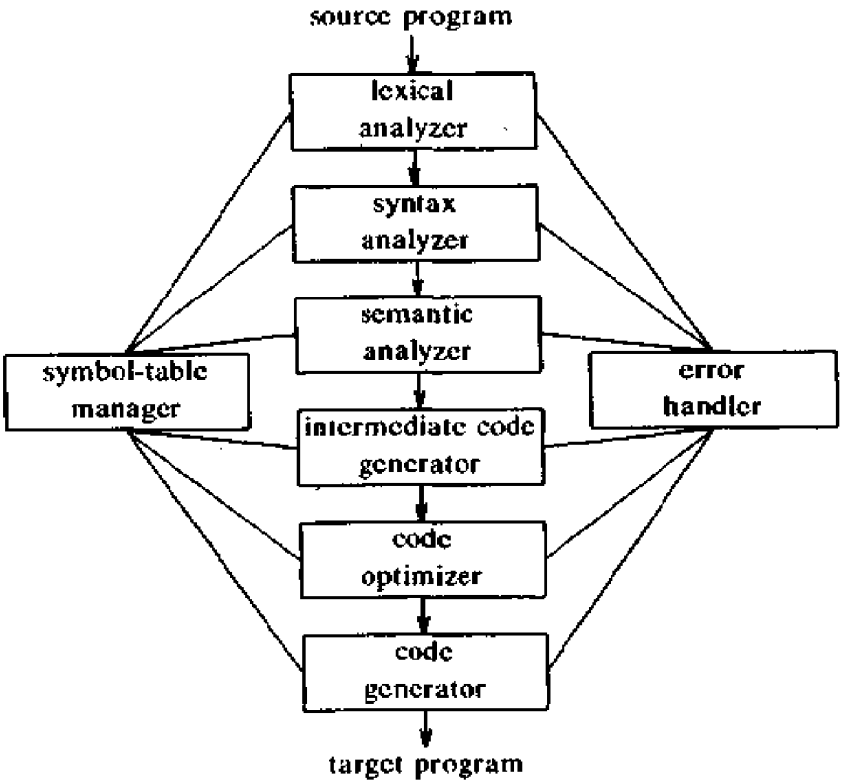
Interpreter is a language processor program that translates and executes source code directly, without compiling it to machine code.

THE PHASE OF COMPILER :

A compiler is a program that reads a program written in one language (source language) and translates it into an equivalent program in another language (target language) .It also reports error in the source program.



- There are two major parts of a compiler: **Analysis** and **Synthesis**
- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.



Each phase transforms the source program from one representation into another representation.

- They communicate with error handlers.
- They communicate with the symbol table.

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimeters and so on)

  Ex
- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.

  Ex
- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Semantic Analyzer

- It checks the source program for semantic errors and collects the type information for the subsequent code-generation phase.
- It uses hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
- An important component of semantic analysis is type checking.

- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars

Intermediate Code Generation

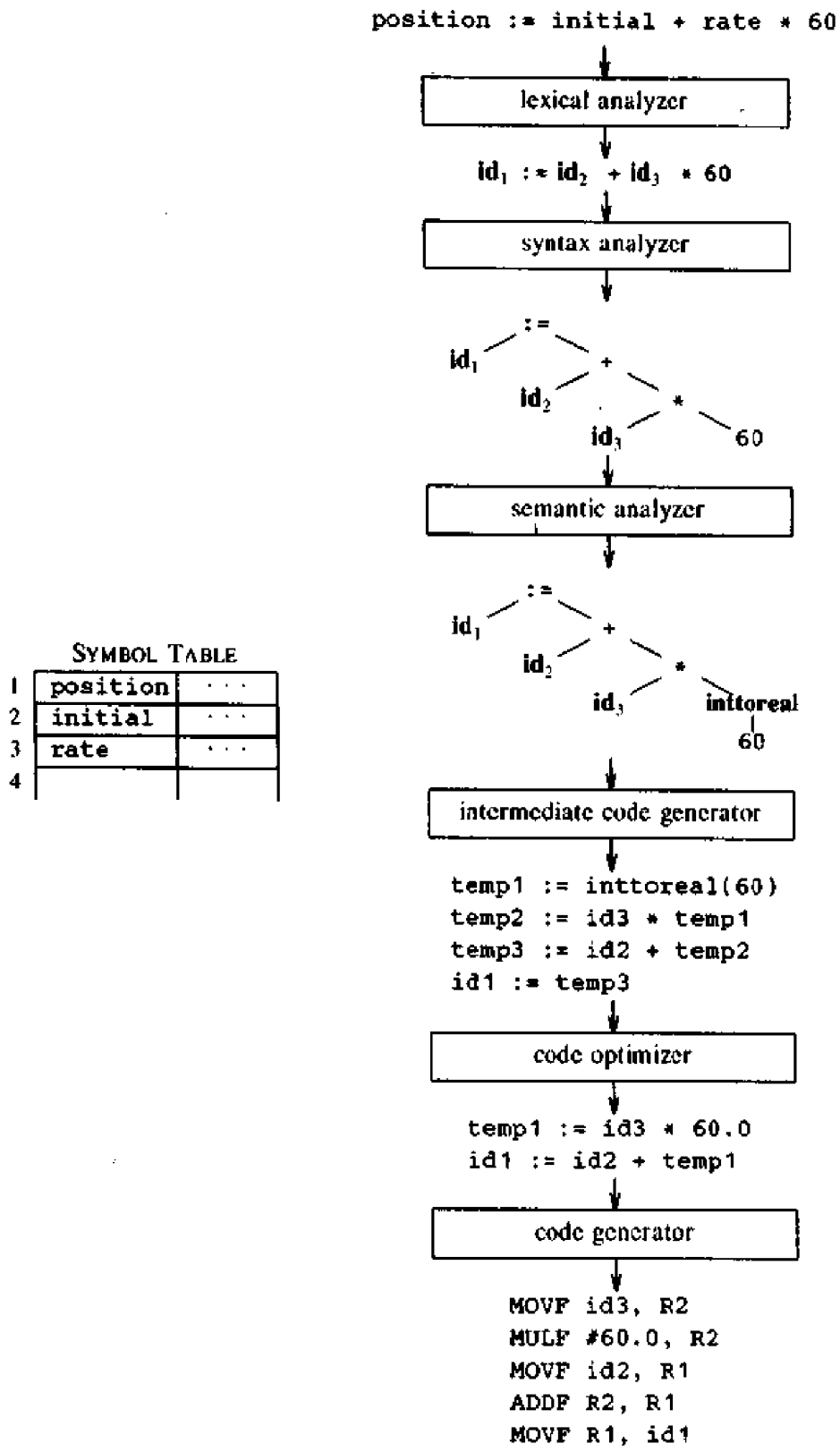
- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture independent). But the level of intermediate codes is close to the level of machine codes.
- An intermediate form called "three-address code" which is 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.

Code Optimizer

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.
- The code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.
- Optimization may involve:
  - Detection and removal of dead(unreachable) code
  - Calculation of constant expressions and terms
  - Collapsing of repeated expressions into temporary storage
  - Loop controlling
  - Moving code outside of loops
  - Removal of unnecessary temporary variables.

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a re-locatable object file containing the machine codes.
- This phase involves:
  - Allocation of registers and memory
  - Generation of correct references
  - Generation of correct types
  - Generation of machine code.


position := initial + rate * 60
lexical analyzer
id1 := id2 + id3 * 60
syntax analyzer
:=
id1
+
id2
*
id3
60
semantic analyzer
:=
id1
+
id2
*
id3
inttoreal
60
intermediate code generator
temp1 := inttoreal(60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
SYMBOL TABLE
1 position ...
2 initial ...
3 rate ...
4

Symbol table

- An essential function of a compiler is to record the identifiers used in the source program and collect information about various attributes of each identifier.
- A symbol table is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- The data structure allows us to find the record for each identifier quickly and to store or retrieve data from that record quickly.
- When an identifier in the source program is detected by the lexical analyzer, the identifier is entered into the symbol table.
- The attributes of an identifier cannot normally be determined during lexical analysis.

  For example, in a Pascal declaration like

  var position initial, rate : real ;*

  The type real is not known when position, initial, and rate are seen by the lexical analyzer.
- The remaining phases enter information about identifiers into the symbol table and then use this information in various ways.

Error Detection and Reporting

- Each compiler phase can encounter errors. However, after detecting an error, a phase must somehow deal with that error, so that compilation can proceed, allowing further errors in the source program to be detected.
- The lexical phase can detect errors where the characters remaining in the input do not form any token of the language.
- Errors where the token stream violates the structure rules (syntax) of the language are determined by the syntax analysis phase.
- During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operation involved.

THREE COMPILER CONSTRUCTION TOOLS

In addition to these software-development tools, other more specialized tools have been developed for helping implement various phases of a compiler.
Some general tools have been created for the automatic design of specific compiler components. These took use specialized languages for specifying and implementing the component, and many use algorithms that are quite sophisticated. 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 a compiler.

The following is a list of some useful compiler-construction tools:

1. Parser generators: These produce syntax analyzers, normally from input that is based on a context-free grammar. In early compilers, syntax analysis consumed not only a large fraction of the running time of a com- compiler, but a large fraction of the intellectual effort of writing a compiler. This phase is now considered one of the easiest to implement. Many parser generators utilize powerful parsing algorithms that are too complex to be carried out by hand.

2. Scanner generators: These automatically generate lexical analyzers, normally from a specification based on regular expressions. The basic organization of the resulting lexical analyzer is in effect a finite automaton.

3. Syntax-directed translation engines: These produce collections of routines that walk the parse tree, generating intermediate code. The basic idea is that one or more "translations" are associated with each node of the parse tree, and each translation is defined in terms of translations at its neighbor nodes in the tree.

4. Automatic code generators: Such a tool takes a collection of rules that define the translation of each operation of the intermediate language into the machine language for the target machine. The rules must include sufficient detail that we can handle the different possible access methods for data; e.g.variables may be in registers, in a fixed (static) location in memory, or may be allocated a position on a stack. The basic technique is "template matching". The intermediate code statements are replaced by "templates" that represent sequences of machine instructions, in such a way that the assumptions about storage of variables match from template to template. Since there are usually many options regarding where variables are to be placed (e.g., in one of several registers or in memory), there are many possible ways to "tile" intermediate code with a given set of templates, and it is necessary to select a good tiling without a combinatorial explosion in running time of the compiler.

5. Data-flow engines: Much of the information needed to perform good code optimization involves "data-flow analysis," the gathering of information about how values are transmitted from one part of a program to each other part. Different tasks of this nature can be performed by essentially the same routine, with the user supplying details of the relationship between intermediate code statements and the information being gathered.

GROUPING OF PHASES.

Logically each phase is viewed as a separate program that reads input and produced output for the next phase. In practice some phases are combined.

Front and Back Ends

- Modern compilers contain two parts, each which is often subdivided. These two parts are the ***front end*** and back ***end.***
- The front end consists of those phases, or parts of phases, that depends primarily on the source language and is largely independent of the target machine. These normally include lexical and syntactic analysis, the creation of the symbol table, semantic analysis, and the generation of intermediate code. A certain amount of code optimization can be done by the front end as well. The front end also includes the error handling that goes along with each of these phases.
- The back end includes those portions of the compiler that depend on the target machine, and generally, these portions do not depend on the source language, just the intermediate language. In the back end, we find aspects of the code optimization phase, and we find code generation, along with the necessary error handling and symbol-table operations.

Passes

- Several phases of compilation are usually implemented in a single pass consisting of reading an input file and writing an output file. In practice, there is great variation in the way the phases of a compiler are grouped into passes, so we prefer to organize our discussion of compiling around phases rather than passes.
- It is common for several phases to be grouped into one pass, and for the activity of these phases to be interleaved during the pass. For example, lexical analysis, syntax analysis, semantic analysis, and intermediate code generation might be grouped into one pass.
- If so, the token stream after lexical analysis may be translated directly into intermediate code. In more detail, we may think of the syntax analyzer as being "in charge." It attempts to discover the grammatical structure on the tokens it sees; it obtains tokens as it needs them, by calling the lexical analyzer to find the next token. As the grammatical structure is discovered, the parser calls the intermediate code generator to perform semantic analysis and generate a portion of the code.

Reducing the Number of Passes

It is desirable to have relatively few passes, since it takes time to read and write intermediate files. If we group several phases into one pass, we may be forced to keep the entire program in memory, because one phase may need information in a different order than a previous phase produces it. The internal form of the program may be considerably larger than either the source program or the target program, so this space may not be a trivial matter.

For some phases, grouping into one pass presents few problems. For example, as we mentioned above, the interface between the lexical and syntactic analyzers can often be limited to a single token.

On the other hand, it is often very hard to perform code generation until the intermediate representation has been completely generated. For example, languages like PUI and ***Algol 68*** permit variables to be used before they are declared. We cannot generate the target code for a construct if we do not know the types of variables involved in that construct. Similarly, most languages allow goto's that jump forward in the code. We cannot determine the target address of such a jump until we have seen the intervening source code and generated target code for it.

In some cases, it is possible to leave a blank slot for missing information, and fill in the slot when the information becomes available. In particular, intermediate and target code generation can often be merged into one pass using a technique called ***"back patching."***

PROGRAMMING LANGUAGE BASICS

- Static / Dynamic distinction
- Environments and states
- Static scope and block structure
- Explicit access control
- Dynamic scope
- Parameter passing mechanism
- Aliasing

Static / Dynamic Distinction

- A language uses static scope or lexical scope if it is possible to determine the scope of the declaration by looking at the program.
- Otherwise the language uses dynamic scope.
- Languages like C, Java uses static scope.
- Example: "static" is applied to data in a java class declaration.

public static int a; (static - Determine the location in memory where the declared variable can be found.)

- "static" been omitted from this declaration, then each object of the class would have its own location where x would be held.
- So the compiler could not determine all these places in advance of running the program.

Environments and States

- Environments, which map names to locations in store, and states, which map locations to their values.
- Names are not only associated with variables
  - Also associated with labels, subprograms, formal parameters, and other program constructs
  - As execution proceeds, the same name can denote different data objects.
  - Example:

Consider the two declarations of the name 'x'.

Environments and States

Consider a storage location 100 is associated with a variable x.

currently x=0

Environment

x is bound to the storage location 100

State

Value held is 0

Consider the assignment statement x:=10

Environment

x is still bound to the storage location 100

State

Value held is 10

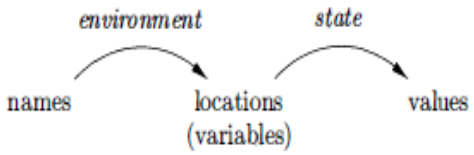


Figure 1.8: Two-stage mapping from names to values

- The environment and state mappings in the above fig are dynamic, but there are a few exceptions:
    1. Static versus dynamic binding of names to locations.
    2. Static versus dynamic binding of locations to values.
- The C definition

    #define ARRAYSIZE 1000

binds the name ARRAYSIZE to the value 1000 statically.

Static Scope and Block Structure

- Early programming languages including c uses static scope.The scope of a declaration is determined implicitly.
- Later languages like C++,Java , C# uses dynamic scope. They have explicit control over scopes by using keywords like **public, private** and **protected.**
- Block is a grouping of declarations and statements.
- Ex: c uses { and }
- Consider the following example

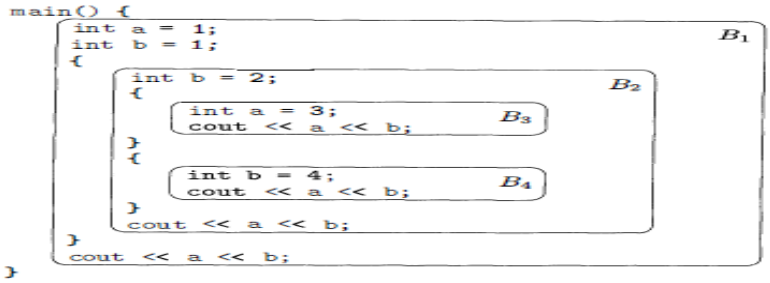


Figure 1.10: Blocks in a C++ program

DECLARATION	SCOPE
`int a = 1;`	$B_1 - B_3$
`int b = 1;`	$B_1 - B_2$
`int b = 2;`	$B_2 - B_4$
`int a = 3;`	B_3
`int b = 4;`	B_4

Figure 1.11: Scopes of declarations in Example 1.6

Explicit Access Control

- use of keywords like public, private, and protected, object oriented languages such as C++ or Java provide explicit control over access to member names in a superclass.
  - Protected names are accessible to subclasses.
  - Public names are accessible from outside the class.
  - private names are accessible to that class and any "friend" classes (the C++ term) .

Dynamic Scope

- Scoping policy is dynamic if it is based on factor(s) that can be known only when the program executes.
- Two examples of dynamic policies:
  1. Macro expansion in the C preprocessor
  2. Method resolution in object-oriented programming.

Parameter Passing Mechanism

- call by value a copy of actual arguments is passed to respective formal arguments.
- call by reference the location (address) of actual arguments is passed to formal arguments, hence any change made to formal arguments will also reflect in actual arguments.

Aliasing

- When parameters are (effectively) passed by reference, two formal parameters can refer to the same location; such variables are said to be aliases of one another.
- This possibility allows a change in one variable to change another.