COMPILER LECTURES

COMPUTER SCIENCE

3RD CLASS

M.SC. SAMER AL-YASSIN

2017-2018

LECTURE 2

Compiler structure

A compiler operates in phases, each of which transforms the source program from one representation to another. A typical decomposition of a compiler is shown in fig (7).

1- lexical analysis :

The lexical analyze is the first stage of a compiler. Its main task is to read The input characters and produce as output a sequence of tokens that the Parser uses for syntax analysis.

2- syntax analysis (parsing)

The syntax analysis (or parsing) is the process of determining if a string of tokens can be generated by grammar. Every programming language has syntactic structure of well-formed programs. Syntax Analyzer takes an out of lexical analyzer and produces a large tree.

3- Semantic analysis

The semantic analysis phase checks the source program for semantics errors and gathers type information for the subsequent code-generation phase. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements. Semantic analyzer takes the output of syntax analyzer and produces another tree.

4- Intermediate code generation

Generate an explicit intermediate representation of the source program. This Representation should have two important properties, it should be easy to produce and easy to translate into target program.

5- Code Optimization

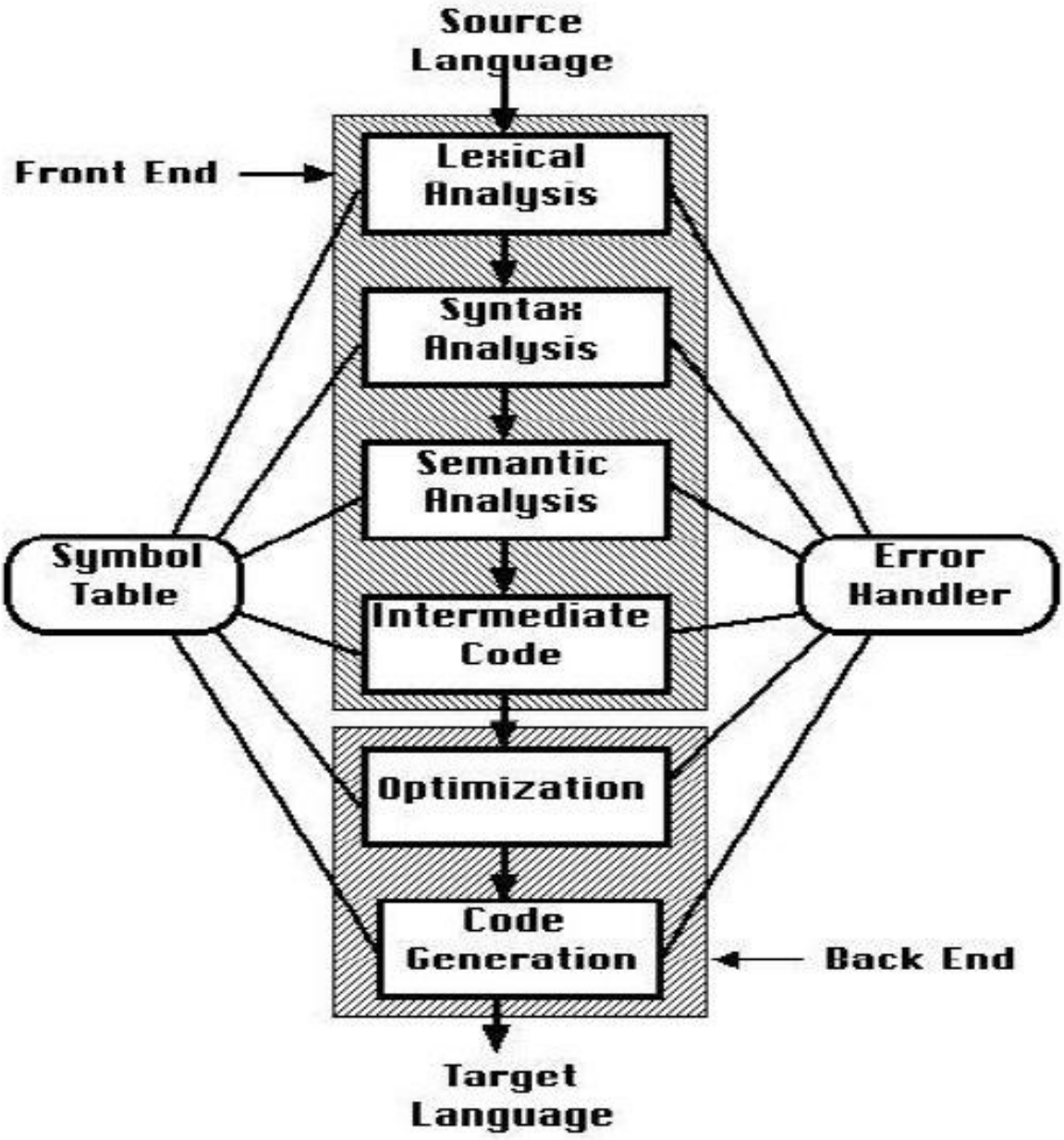
Attempts to improve the intermediate code so that faster running machine code will result.

6- code generation

Generates a target code consisting normally of machine code or an assemble Code. Memory locations are selected for each of the variables used by the program. Then intermediate instructions are each translated in to a sequence.

- Symbol table management:

Portion of the compiler keeps tracks of the name used by the program and records essential information about each, such as type (integer, real, etc.). The data structure used to record this information is called symbolic table.



Phases of a Compiler

-Error handler:

Is called when an error in the source program is detected. It must warn the programmer by issuing a diagnostic, and adjust the information being passed from phase to phase so that each phase can produced.

Types of errors:

The syntax and semantic phases usually handle a large fraction of errors detected by compiler.

1- ***Lexical error***: The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. Few errors are discernible at the lexical level alone, because a lexical analyzer has a very localized view of the source program.

Example: If the string **fi** is encountered in a **C** program for the first time in context:

fi (a== f(x))….

A **lexical analyzer** cannot tell whether **fi** is a misspelling of the keyword **if** or An undeclared function name. Since **fi** is a valid identifier, the lexical analyzer must return the token for an identifier and let some other phase of the compiler handle any error.

2- ***Syntax error***: The syntax phase can detect Errors where the token stream violates the structure rules (syntax) of the language.

3- ***Semantic error***: During semantic analysis 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 of which is the name of an array, and the other the name of a procedure.

4- Runtime error.

Passes:

In an implementation of a compiler, portions of one or more phases are combined into a module called a pass. A pass read the source program or the output of the previous pass, makes the transformations specified by its phases and writes output into an intermediate file, which may then be read by subsequent pass.

Lexical Analyzer

The lexical analyzer is the first phase of compiler. The main task of lexical analyzer is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc... for syntax analyzer. This interaction summarized in fig.7, is commonly implemented by making the lexical analyzer be a subroutine of the parser. Up on receiving a "get next token" command from the parser, the lexical analyzer reads input characters until it can identify the next token.

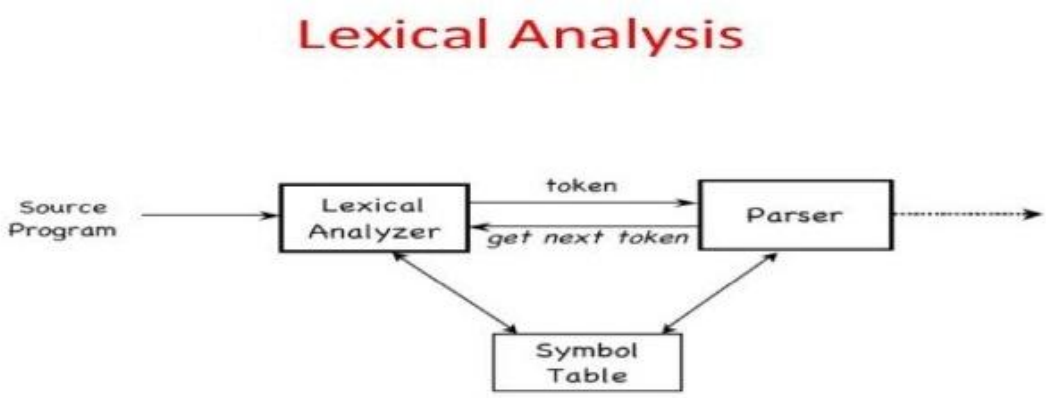


Fig. (7) Interaction of lexical analyzer with parser

Preliminary scanning:

- The source code is converted into stream of tokens.
- Removes white spaces and comments.
- eg: x = a + b * c /* source code */ --> Lexical Analyzer --> id = id + id * id
- This is achieved by using patterns which is known to the lexical analyzer

Tokens, Patterns, Lexemes

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a ***pattern*** associated with the token. The ***pattern*** is said to match each string in the set.
A ***lexeme*** is a sequence of characters in the source program that is matched by the pattern for a token.
Example: Constant pi = 3.1416
The sub string pi is a lexeme for the token “identifier”.

TOKEN	SAMPLE LEXEMES	INFORMAL DESCRIPTION OF PATTERN
const	const	const
if	if	if
relation	<, <=, =, <>, >, >=	< or <= or = or <> or >= or >
id	pi, count, D2	letter followed by letters and digits
num	3.1416, 0, 6.02E23	any numeric constant
literal	"core dumped"	any characters between " and " except "

Fiq (8) example of token

Token: A lexical token is a sequence of characters that can be treated as a unit in the Grammar of the programming languages .in most programming language, the following construct is treated as tokens: keywords, operators, Identifiers, constants, literal strings (any characters between "and "), and punctuations symbols such as parentheses, commas, and semicolons.

The ***lexical analyzer*** returns to parser a representation for the token it has found. This representation is:

- an **integer code** if there is a simple construct such as a left parenthesis, comma or colon .
- or a **pair** consisting of an **integer code** and a **pointer to a table** if the token is more complex element such as an **identifier or constant** .

This integer code gives the token type. The pointer points to the value of the token. For example we may treat "Operator" as a token and let the second component of the pair indicate whenever the operator found is +, *, and so on.

Pattern**:** A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme**:** A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Symbol Table

A symbol table is a table with two fields. A name field and an information field. This table is generally used to store information about various sources language construct.
The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code. The information is collected by the analysis phase of the compiler and used by the synthesis phase to generate the target code.

We required several capabilities of the symbol table we need to be able to:

1- Determine if a given name is in the table, the symbol table routines are concerned with saving and retrieving tokens.
   Insert(s,t) : this function is to add a new name to the table
   Lookup(s) : returns index of the entry for string s, or 0 if s is not found.
2- Access the information associated with a given name, and add new information for a given name.
3- Delete a name or group of names from the table.

For example consider tokens **begin ,** we can initialize the symbol the function**:**
insert("begin",1)

Attributes for Tokens

When more than one pattern matches a lexeme, the lexical analyzer must provide additional information about the particular lexeme that matched to the subsequent phases of the compiler. For example, the pattern **num** matches both the strings 0 and 1, but it is essential for the code generator to know what string was actually matched.

The lexical analyzer collects information about tokens into their associated attributes. The tokens influence parsing decisions; the attributes influence the translation of tokens. As a practical matter, a token has usually only a single attribute — a pointer to the symbol-table entry in which the information about the token is kept; the pointer becomes the attribute for the token. For diagnostic purposes, we may be interested in both the lexeme for an identifier and the line number on which it was first seen. Both these items of information can be stored in the symbol-table entry for the identifier.

Example 3.1. The tokens and associated attribute-values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs:

<id, pointer to symbol-table entry for E>

<assign_op, >

<id, pointer to symbol-table entry for M>

<mult_op, >

<id, pointer to symbol-table entry for C>

<exp_op, >

<num, integer value 2>