Unit 1
Introduction to Compiler

……

Topics

1.1 Compiler Structure: Analysis and Synthesis Model of Compilation, different sub-phases within analysis and synthesis phases

1.2 Basic concepts related to Compiler such as interpreter, simple One-Pass Compiler, preprocessor, macros, and Symbol table and error handler.

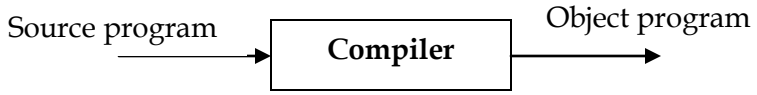
……

What is Compiler?

A compiler is a translator software program that takes its input in the form of program written in one particular programming language (source language) and produce the output in the form of program in another language (object or target language).

The major features of Compiler are listed below:

- A compiler is a translator that converts the high-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When you execute a program which is written in HLL programming language then it executes into two parts.
- In the first part, the source program compiled and translated into the object program (low level language).
- In the second part, object program translated into the target program through the assembler.



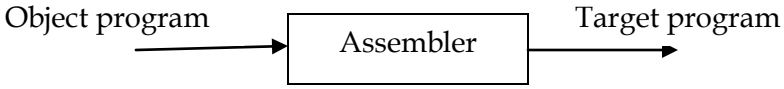


Fig: Execution process of source program in Compiler

List of compilers

- Ada compilers

- ALGOL compilers
- BASIC compilers
- C# compilers
- C compilers
- C++ compilers
- COBOL compilers
- Common Lisp compilers
- ECMAScript interpreters
- Fortran compilers
- Java compilers
- Pascal compilers
- PL/I compilers
- Python compilers
- Smalltalk compilers etc.

Phases of a Compiler

A compiler operates in phases. A phase is a logically interrelated operation that takes source program in one representation and produces output in another representation. The compilation process contains the sequence of various phases. Each phase takes source program in one representation and produces output in another representation. The phases of a compiler are shown in below. There are two phases of compilation.

a. Analysis (Machine Independent/Language Dependent)

b. Synthesis (Machine Dependent/Language independent)

Analysis part

In analysis part, an intermediate representation is created from the given source program.

This part is also called front end of the compiler. This part consists of mainly following four phases:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis and
- Intermediate code generation

Synthesis part

In synthesis part, the equivalent target program is created from intermediate representation of the program created by analysis part. This part is also called back end of the compiler. This part consists of mainly following two phases:

- Code Optimization and
- Final Code Generation

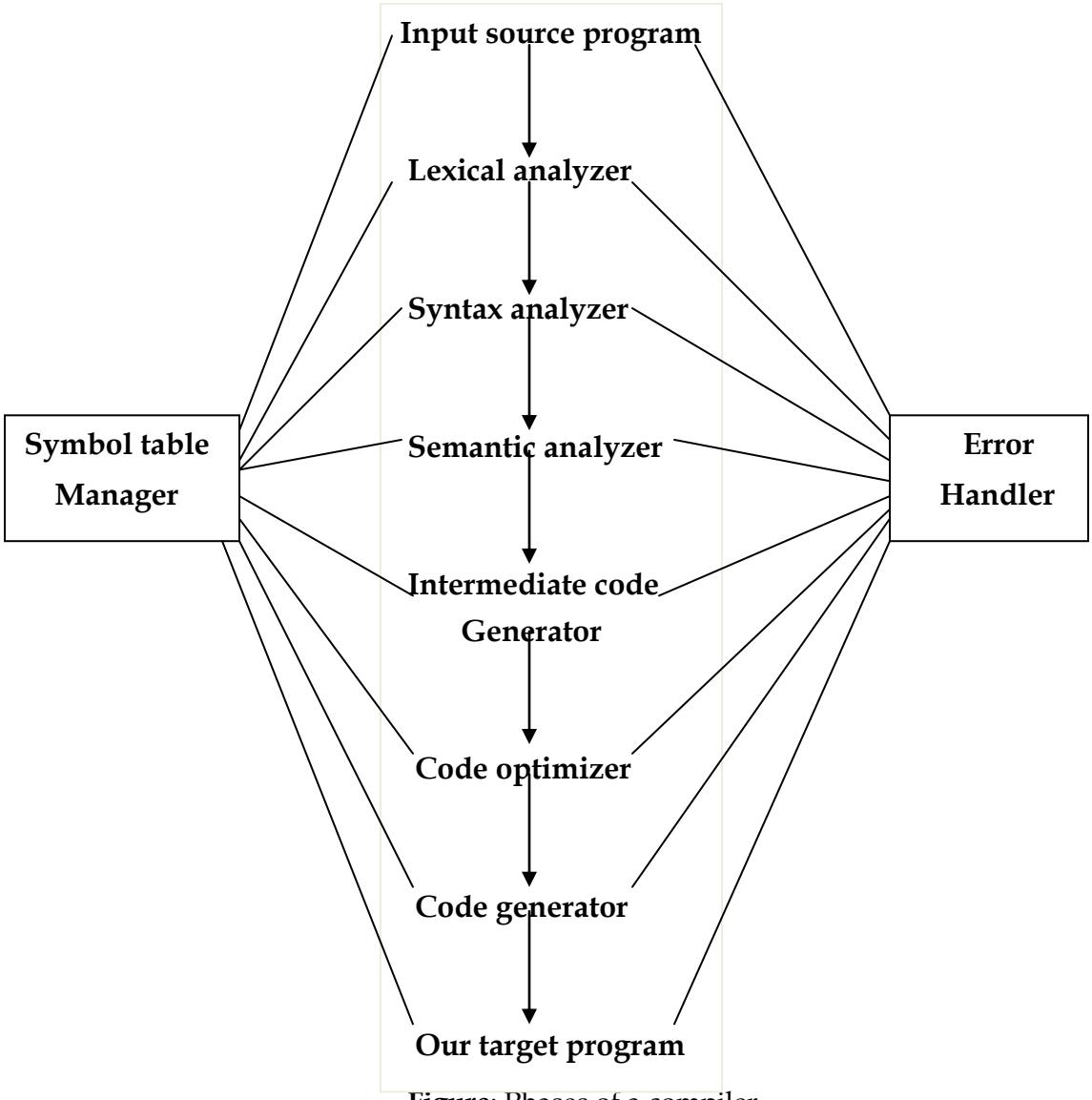
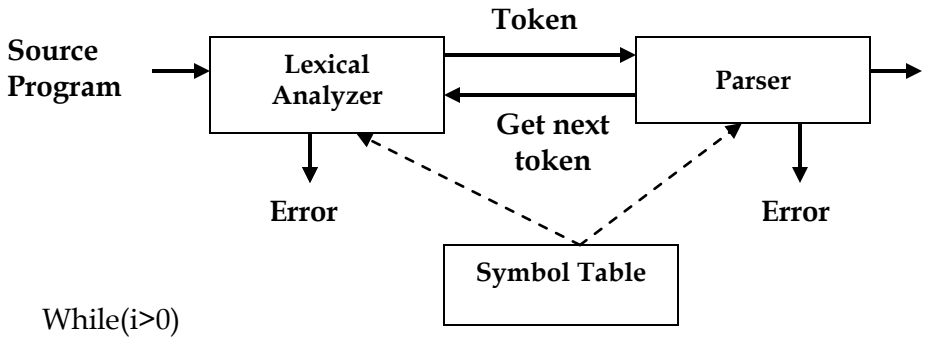


Figure: Phases of a compiler

1. Lexical Analysis (or Scanning)

Lexical analysis or scanning is the process where the source program is read from left-to-right and grouped into tokens. Tokens are sequences of characters with a collective meaning. In any programming language tokens may be constants, operators, reserved words, punctuations etc.

The Lexical Analyzer takes a source program as input, and produces a stream of tokens as output. Normally a lexical analyzer doesn't return a list of tokens; it returns a token only when the parser asks a token from it. Lexical analyzer may also perform other auxiliary operation like removing redundant white space, removing token separator (like semicolon) etc. In this phase only few limited errors can be detected such as illegal characters within a string, unrecognized symbols etc. Other remaining errors can be detected in the next phase called syntax analyzer.



Example:

```
While(i>0)
        i=i-2;
```

Tokens	Description
While	while keyword
(	left parenthesis
I	Identifier
>	greater than symbol
0	integers constant
)	right parenthesis
I	Identifier
=	Equals
I	Identifier
-	Minus
2	integers constant
;	Semicolon

The main purposes of lexical analyzer are:

- It is used to analyze the source code.
- Lexical analyzer is able to remove the comments and the white space present in the expression.
- It is used to format the expression for easy access i.e. creates tokens.
- It begins to fill information in SYMBOL TABLE.

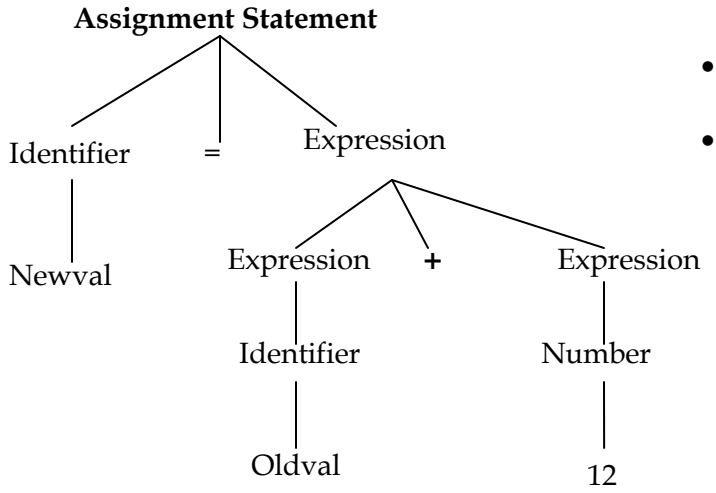
2. Syntax Analyzer (or Parsing)

The second phase of the compiler phases is syntax Analyzer, once lexical analysis is completed the generation of lexemes and mapped to the token, then parser takes over to check whether the sequence of tokens is grammatically correct or not, according to the rules that define the syntax of the source language. The main purposes of Syntax analyzer are:

- Syntax analyzer is capable analyzes the tokenized code for structure.
- This is able to tags groups with type information.

A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given source program. Syntax analyzer is also called the parser. Its job is to analyze the source program based on the definition of its syntax. It is responsible for creating a parse-tree of the source code.

Ex: newval:= oldval + 12



- In a parse tree all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar (CFG).

The syntax of a language is specified by a context free grammar (CFG).
The rules in a CFG are mostly recursive.
A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
– If it satisfies, the syntax analyzer creates a parse tree for the given program.

3. Semantic Analyzer

The next phase of the semantic analyzer is the semantic analyzer and it performs a very important role to check the semantics rules of the expression according to the source language. The previous phase output i.e. syntactically correct expression is the input of the semantic analyzer. Semantic analyzer is required when the compiler may require performing some additional checks such as determining the type of expressions and checking that all statements are correct with respect to the typing rules, that variables have been properly declared before they are used, that functions are called with the proper number of parameters etc. This semantic analyzer phase is carried out using information from the parse tree and the symbol table.
The parsing phase only verifies that the program consists of tokens arranged in a syntactically valid combination. Now semantic analyzer checks whether they form a sensible set of instructions in the programming language or not. Some examples of the things checked in this phase are listed below:

- The type of the right side expression of an assignment statement should match the type of the left side ie. in the expression newval = oldval + 12, The type of the expression (oldval+12) must match with type of the variable newval.
- The parameter of a function should match the arguments of a function call in both number and type.
- The variable name used in the program must be unique etc.

The main purposes of Semantic analyzer are:

- It is used to analyze the parsed code for meaning.
- Semantic analyzer fills in assumed or missing information.
- It tags groups with meaning information.

Important techniques that are used for Semantic analyzer:

- The specific technique used for semantic analyzer is Attribute Grammars.
- Another technique used by the semantic analyzer is Ad hoc analyzer.

4. Intermediate code generator

If the program syntactically and semantically correct then intermediate code generator generates a simple machine independent intermediate language. The intermediate language should have two important properties:

- It should be simple and easy to produce.
- It should be easy to translate to the target program

Some 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.

Example: find intermediate code representation of following expression,

A = b + c * d / f

Intermediate code for above example

T1 = c * d
T2 = T1 / f
T3 = b + T2
A = T3

The main purposes of Intermediate code generation are:

- This phase is used to generate the intermediate code of the source code.

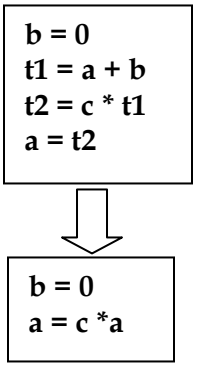
Important techniques that are used for Intermediate code generations:

- Intermediate code generation is done by the use of three address code generation.

5. Code Optimization

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side effects. The process of removing unnecessary part of a code is known as code optimization. Due to code optimization process it decreases the time and space complexity of the program i.e.

- Detection of redundant function calls
- Detection of loop invariants
- Common sub-expression elimination
- Dead code detection and elimination



The main purposes of Code optimization are:

- It examines the object code to determine whether there are more efficient means of execution.

Important techniques that are used for lexical analyzer:

- Loop unrolling.
- Common-sub expression elimination
- Operator reduction etc.

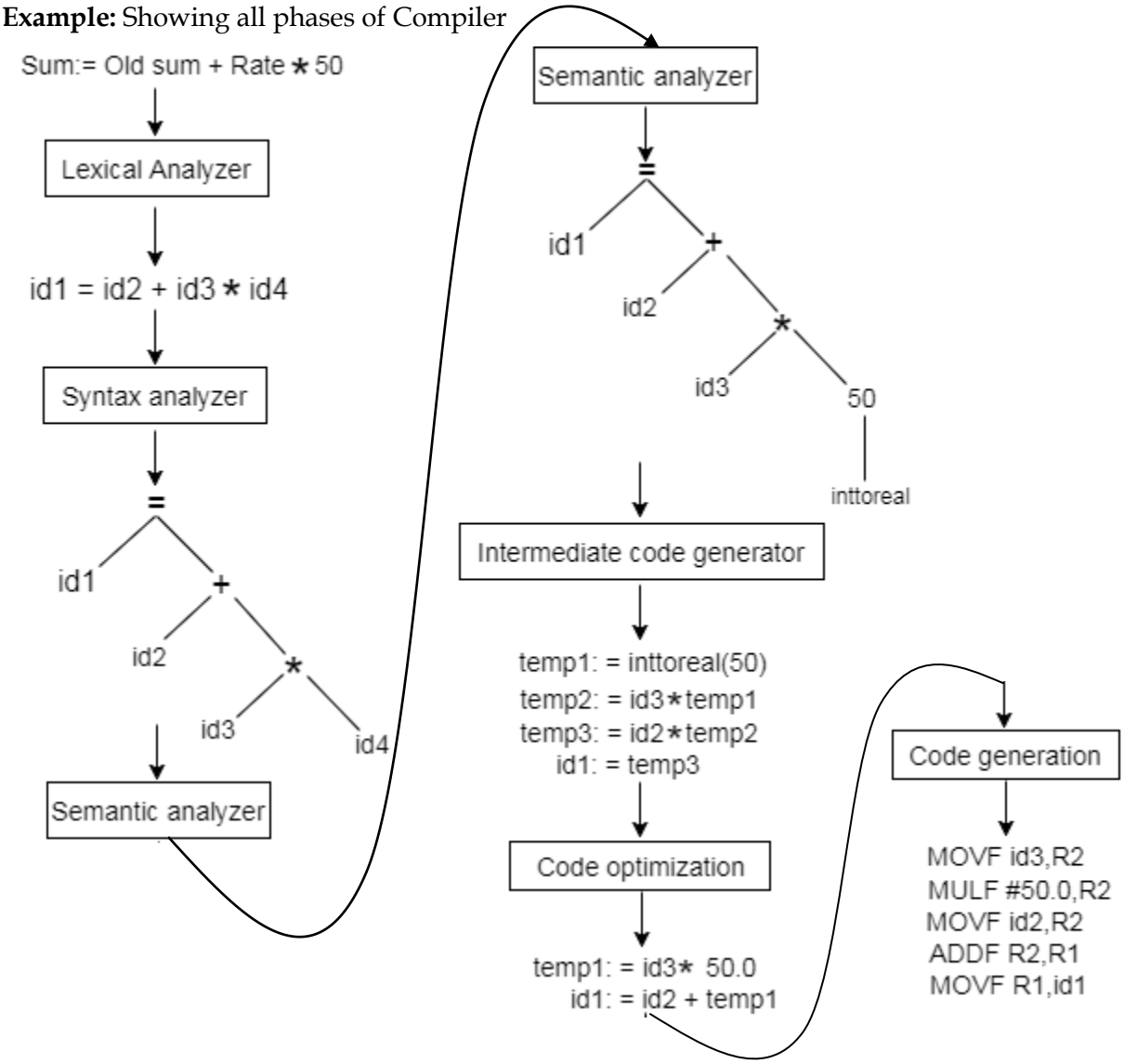
6. Code Generation

It generates the assembly code for the target CPU from an optimized intermediate representation of the program. Ex: Assume that we have an architecture with instructions whose at least one of its operands is a machine register.

A = b + c * d / f

```
MOVE    c, R1
MULT    d, R1
DIV     f, R1
ADD     b, R1
MOVE    R1, A
```

Example: Showing all phases of Compiler



One pass VS Multi-pass compiler

Each individual unique step in compilation process is called a phase such as lexical analysis, syntax analysis, semantic analysis and so on. Different phases can be combined into one or more than one group. These each group is called passes. If all the phases are combined into a single group then this is called as one pass compiler otherwise more than one pass constitute the multi-pass compiler.

One pass compiler	Multi-pass compiler
1. In a one pass compiler all the phases are combined into one pass. 2. Here intermediate representation of source program is not created.	1. In multi-pass compiler different phases of compiler are grouped into multiple phases. 2. Here intermediate representation of source program is created.

3. It is faster than multi-pass compiler. 4. It is also called narrow compiler. 5. Pascal's compiler is an example of one pass compiler. 6. A single-pass compiler takes more space than the multi-pass compiler	3. It is slightly slower than one pass compiler. 4. It is also called wide compiler. 5. C++ compiler is an example of multi-pass compiler. 6. A multi-pass compiler takes less space than the multi-pass compiler because in multi-pass compiler the space used by the compiler during one pass can be reused by the subsequent pass.

Multi-pass Compiler

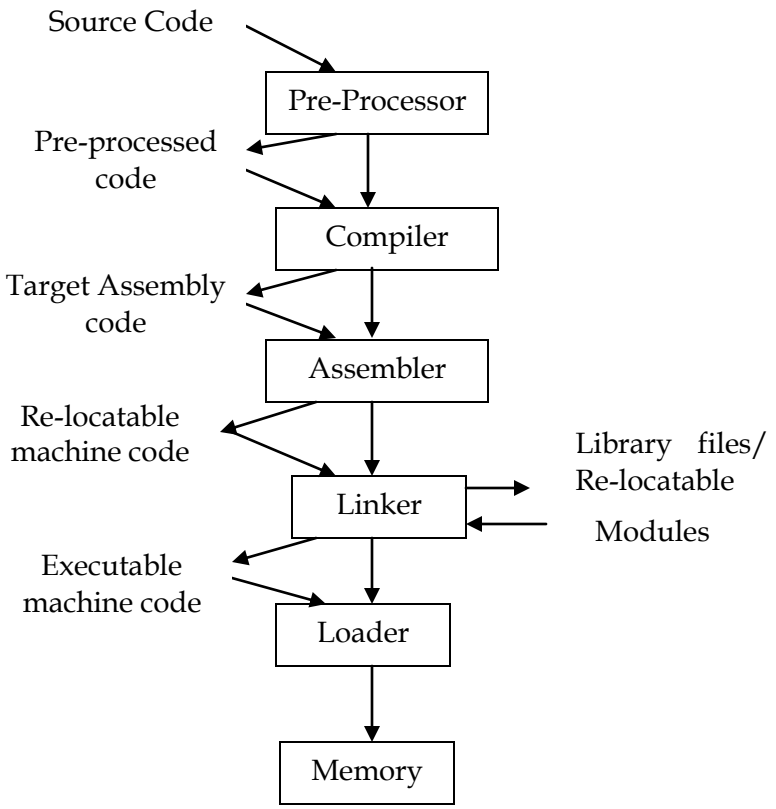
Multi pass compiler is used to process the source code of a program several times. In the first pass, compiler can read the source program, scan it, extract the tokens and store the result in an output file. In the second pass, compiler can read the output file produced by first pass, build the syntactic tree and perform the syntactical analysis. The output of this phase is a file that contains the syntactical tree. In the third pass, compiler can read the output file produced by second pass and check that the tree follows the rules of language or not. The output of semantic analysis phase is the annotated tree syntax. This pass is going on, until the target output is produced.

One-pass Compiler

One-pass compiler is used to traverse the program only once. The one-pass compiler passes only once through the parts of each compilation unit. It translates each part into its final machine code. In the one pass compiler, when the line source is processed, it is scanned and the token is extracted. Then the syntax of each line is analyzed and the tree structure is build. After the semantic part, the code is generated. The same process is repeated for each line of code until the entire program is compiled.

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember. These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



Preprocessor

A preprocessor, generally considered as a part of compiler, is a tool that produces input for compilers. It deals with macro-processing, augmentation; file inclusion, language extension, etc. A preprocessor produce input to compilers. They may perform the following functions.

- **Macro processing:** A preprocessor may allow a user to define macros that are short hands for longer constructs.
- **File inclusion:** A preprocessor may include header files into the program text.
- **Rational preprocessor:** these preprocessors augment older languages with more modern flow-of-control and data structuring facilities.
- **Language Extensions:** these preprocessor attempts to add capabilities to the language by certain amounts to build-in macro.

Macros

A macro (which stands for macroinstruction) is a programmable pattern which translates a certain sequence of input into a preset sequence of output. Macros can make tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.

A Macro instruction is the notational convenience for the programmer. For every occurrence of macro the whole macro body or macro block of statements gets expanded in the main source code. Thus Macro instructions make writing code more convenient.

Features of Macro Processor:

- Macro represents a group of commonly used statements in the source programming language.
- Macro Processor replaces each macro instruction with the corresponding group of source language statements. This is known as expansion of macros.
- Using Macro instructions programmer can leave the mechanical details to be handled by the macro processor.
- Macro Processor designs is not directly related to the computer architecture on which it runs.
- Macro Processor involves definition, invocation and expansion.

Compiler Construction Tools

For the construction of a compiler, the compiler writer uses different types of software tools that are known as compiler construction tools. These tools make use of specialized languages for specifying and implementing specific components, and most of them use sophisticated algorithms. The tools should hide the details of the algorithm used and produce component in such a way that they can be easily integrated into the rest of the compiler. Some of the most commonly used compiler construction tools are:

- **Scanner generators:** They automatically produce lexical analyzers or scanners. Example: flex, lex, etc
- **Parser generators:** They produce syntax analyzers or parsers. Example: bison, yacc etc.
- **Syntax-directed translation engines:** They produce a collection of routines, which traverses the parse tree and generates the intermediate code.
- **Code generators:** They produce a code generator from a set of rules that translates the intermediate language instructions into the equivalent machine language instructions for the target machine.
- **Data-flow analysis engines:** They gather the information about how the data is transmitted from one part of the program to another. For code optimization, data-flow analysis is a key part.
- **Compiler-construction toolkits:** They provide an integrated set of routines for construction of the different phases of a compiler.

Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phase of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the

same identifier within a program. The lexical analyzer can create a symbol table entry and can return token to the parser, say id, along with a pointer to the lexeme. Then the parser can decide whether to use a previously created symbol table or create new one for the identifier.

The basic operations defined on a symbol table include

- **allocate –** to allocate a new empty symbol table
- **free –** to remove all entries and free the storage of a symbol table
- **insert –** to insert a name in a symbol table and return a pointer to its entry
- **lookup –** to search for a name and return a pointer to its entry
- **set_attribute –** to associate an attribute with a given entry
- **get_attribute –** to get an attribute associated with a given entry

Other operations can be added depending on requirement

- For example, a **delete** operation removes a name previously inserted

Possible entries in a symbol table:

- Name: a string.
- Attribute:
  - ✓ Reserved word
  - ✓ Variable name
  - ✓ Type name
  - ✓ Procedure name
  - ✓ Constant name
  - ✓ _ _ _ _ _ _ _
- Data type
- Scope information: where it can be used.
- Storage allocation, size…
- ……

Example: Let's take a portion of a program as below:

```
void fun ( int A, float B)
{
        int D, E;
        D = 0;
        E = A / round (B);
        if (E > 5)
        {
                Print D
        }
}
```

Its symbol table is created as below:

Symbol	**Token**	**Data type**	**Initialization?**
Fun	Id	Function name	No
A	Id	Int	Yes

B	Id	Float	Yes
D	Id	Int	No
E	Id	Int	No

Symbol	**Token**	**Data type**	**Initialization?**
Fun	Id	Function name	No
A	Id	Int	Yes
B	Id	Float	Yes
D	Id	Int	Yes
E	Id	Int	Yes

Error handling in compiler

Error detection and reporting of errors are important functions of the compiler. Whenever an error is encountered during the compilation of the source program, an error handler is invoked. Error handler generates a suitable error reporting message regarding the error encountered. The error reporting message allows the programmer to find out the exact location of the error. Errors can be encountered at any phase of the compiler during compilation of the source program for several reasons such as:

- In lexical analysis phase, errors can occur due to misspelled tokens, unrecognized characters, etc. These errors are mostly the typing errors.
- In syntax analysis phase, errors can occur due to the syntactic violation of the language.
- In intermediate code generation phase, errors can occur due to incompatibility of operands type for an operator.
- In code optimization phase, errors can occur during the control flow analysis due to some unreachable statements.
- In code generation phase, errors can occurs due to the incompatibility with the computer architecture during the generation of machine code. For example, a constant created by compiler may be too large to fit in the word of the target machine.
- In symbol table, errors can occur during the bookkeeping routine, due to the multiple declaration of an identifier with ambiguous attributes.