Computer Science
Compiler Design
Module 2

Module 2:

Objectives: To understand various phases of a compiler with the help of an example and the tools used for various phases.

Phases of the Compiler: In this module, let us take an example input and understand the various happenings in the phases of the compiler. Consider the following example

position =initial + rate * 60 **(2.1)**

The output of the various phases for the example of statement (2.1) is given in Figure 2.1 and the corresponding symbol table is given in Figure 2.2. The lexical analyser breaks the input into three identifiers and one constant and populates the symbol table. The syntax analyser constructs the parse tree and ensures that the syntax of the statement is correct. The semantic analyser coerce constant 60 into real. The intermediate code generation creates temporary variable and generates three address code. The code optimizer removes redundancy in the three address code and final assembly language code gets generated by the code generator. Let us discuss the individual outputs in the next sub-sections.

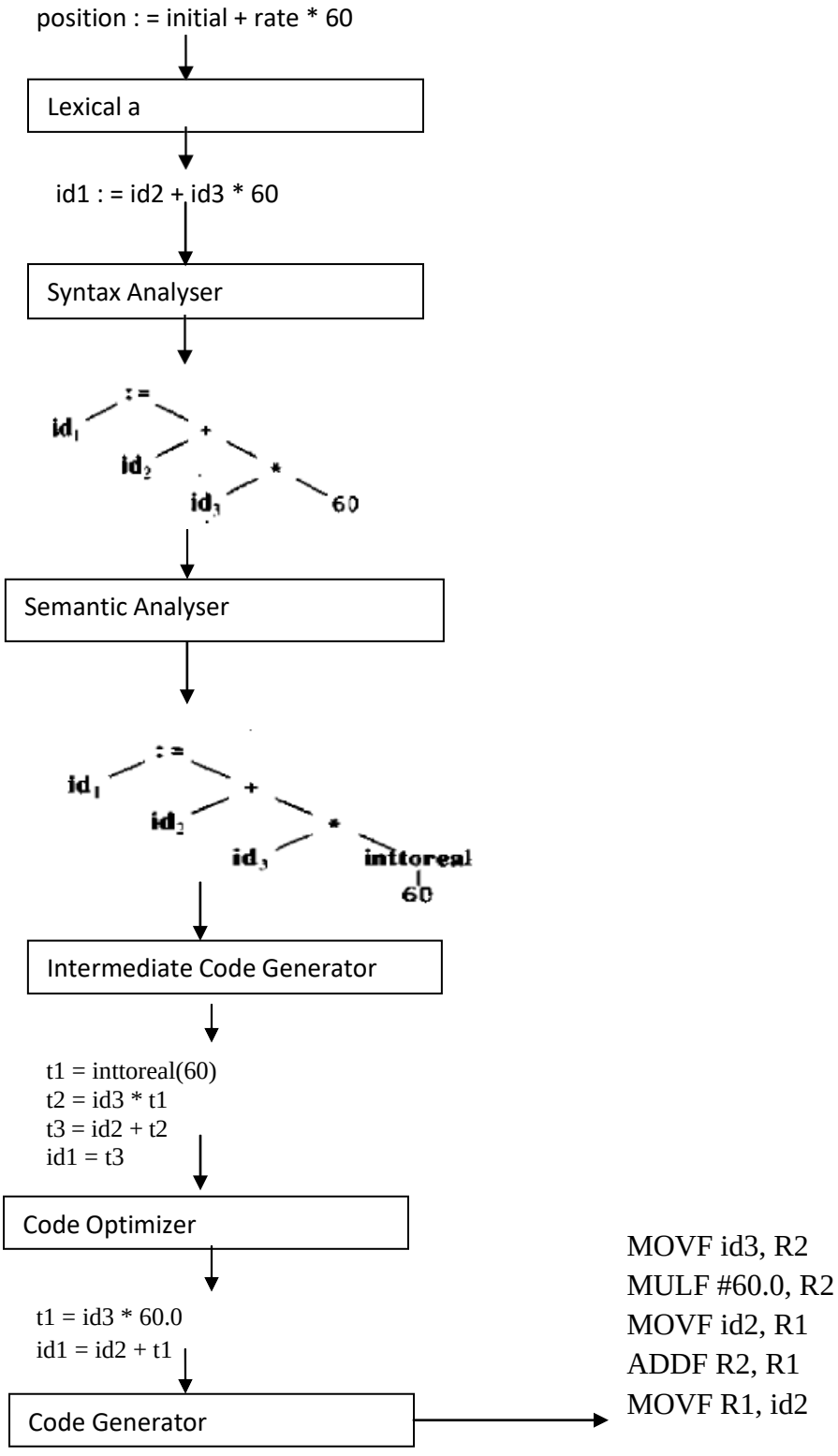

Figure 2.1 Output from the various phases for the example.

Location	Name	Attributes
1	position	…
2	initial	…
3	Rate	…
4	…	…

Figure 2.2 Symbol Table

Lexical Phase: In the example considered, we have the lexemes, “position”, “:=”, “initial”, “+”, “rate”, “*”, “60”. After this the compiler encounters a “;” indicating end of statement. After seeing this, the compiler creates the symbol table for the variables where the variables are referred as identifiers. The symbol-table entry for an identifier holds information about the identifier, such as its name and type. In this example, ”position” is a lexeme 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 as can be seen from Figure 2.2. “=” is a lexeme that is mapped into the token (=). Since this token needs no attribute-value, we have omitted the second component. For notational convenience, the lexeme itself is used as the name of the abstract symbol. “initial” is a lexeme that is mapped into the token (id, 2), where 2 points to the symbol-table entry for” initial” while “rate” is mapped to (id, 3) where 3 points to its symbol table entry. Similarly, “+” and “*” have only the lexemes and no attribute. If in this phase, if the lexer encounters a lexeme like “+++”, then it is logged as error since the compiler cannot identify the attribute for this lexeme. However, the lexer recovers from error by deleting some characters in order to help proceed with compilation.

Syntax Phase: In this phase the tokens are the input and the output will be a syntax tree. Given the sequence of token, the compiler constructs the syntax tree based on the Context Free Grammar that is present with it. For the example of statement (2.1), the input is an expression and hence mapped to the Expression Grammar. The parse (syntax) tree corresponding to this example is given Figure 2.3 and the syntax tree with identifiers is specified in Figure 2.4. The syntax tree has been assigned labels, position, initial, rate, 60 and since the parse tree matches for the input, the syntax analyser gives a go ahead for compilation. If in the syntax phase, there is an error, then it is logged into the error handler and the compiler recovers from error to help proceed with compilation. In this example, if there is a “++” operator in place of the “+” operator, then the parser logs that as an error, however it will delete only “+” and help the compiler move with the compilation process.

Figure 2.3 Parse Tree for the example position := initial + rate * 60

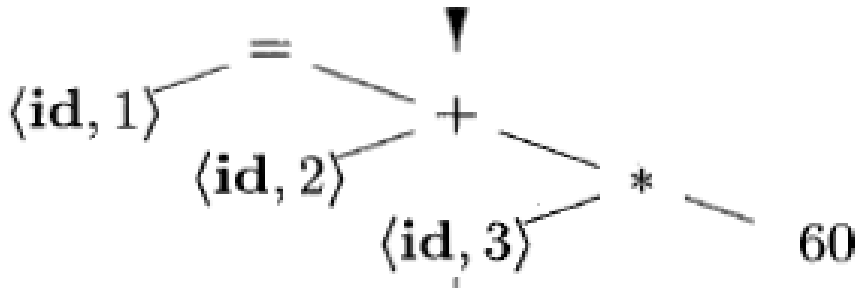

Figure 2.4 Syntax Tree for the same example, with identifiers in place of variable names

Semantic Phase: This phase of the compiler, performs type checking and coercion. In this example, the constant "60" need to be converted to floating point number since it is multiplied by a variable "rate" which is typically a floating point value. The other type checking is performed, and in this example, "position" will have a floating point value since it involves addition of a floating point value with "initial". Hence, it performs this coercion and the modified syntax tree is shown in Figure 2.5

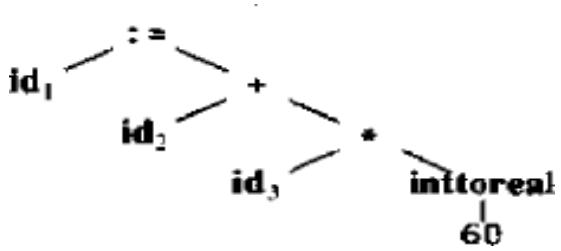

Figure 2.5 Modified Syntax tree implementing coercion

Intermediate Code Generation: The input to this phase is the coerced syntax tree and the output with be three address code. The compiler assigns temporary variables for all the interior nodes. In this example, based on the modified syntax tree of figure 2.5, the interior nodes corresponding to inttoreal, +, *, are assigned temporary variables. The temporary variable corresponding to the interior node will be one of the 3 addresses that comprises of the three address code. This is typically the left hand side variable of the 3-address code. The right hand side expression is derived using the left and right children of the interior node and the operator to join them is based on the interior node. If, there is only one child for the interior node, then the right hand side of the 3-address code will have only one operand. In this example, "inttoreal" is an interior node which is assigned temporary variable "t1". The operator is "inttoreal" and the only right hand side operand is "60". So, the corresponding 3 –address code is given in statement (2.2)

t1 = inttoreal(60) → (2.2)

Consider one more interior node, "*". It is assigned temporary variable "t2" and its left and right children are "id3" and temporary variable "t1". Hence, the corresponding 3-address code is given in statement (2.3)

t2 = id3 * t1 → (2.3)

Similarly, for the other two interior nodes, the corresponding three-address codes are given in statement (2.4)

t3 = id2 + t2

id1 = t3 → (2.4)

Code Optimizer: This is the fifth phase of the compiler, whose input is the three address code and output is an optimized three address code. In this example, the statements represented by (2.2) and (2.3) could be combined together in one statement, by literally defining the floating point value of 60 as 60.0. Similarly, the statements represented in (2.4) could be represented as one statement thus getting rid of one more three address code. The modified code is given in statement (2.5)

t1 = id3 * 60.0

id1 = id2 + t1 → (2.5)

Code Generation: This phase generates code using the input received from the Code optimizer. The statements represented in (2.5), is a multiplication expression which involves one variable and one constant. So, we need to get the variable into register and multiply this register content with 60.0. The following two statements does that. The MOVF command indicates it is a

floating point operation and moves id3 into R2. The second statement is an immediate operand involving the same register and the constant.

```
MOVF id3, R2
MULF #60.0, R2          → (2.6)
```

The result of this is necessary in the next computation, the computed value is retained in register itself. The first statement of (2.7) moves variable id2 to R1 and the ADDF command adds the registers R2, R1 compute the final result. The result is then put back into the variable id1.

```
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1            → (2.7)
```

Cousins of the Compiler

The Compiler requires the help of other system software to execute a code. They are discussed in the following sub-sections:

Preprocessors: Typically the macros which are defined as #include, # define is handled by the pre-processor. #include ensures that the compilation can proceed by copying the prototypes of the function definition of the library in the current module. #define, replaces the variable with its corresponding value before compilation.

Assemblers: Compiler may produce assembly code instead of generating relocatable machine code directly. Since it is difficult, assemblers help the compilers to generate relocatable machine code from the input assembly code.

Loaders and Linkers: Loader copies code and data into memory, allocates storage, setting protection bits, mapping virtual addresses, etc. Linker handles relocation and resolves symbol references.

Debugger: Helps in locating errors in the code by the use of break-points

Compiler Construction Tools

A compiler is very difficult to build from scratch for a particular programming language and a target language. Hence, we have some tools to aid the designing of compiler relatively easier. The following sub-sections discuss few of them.

Scanner generators: This helps in the lexical phase of the compiler. The input is the source program and the output is the tokens. The task of reading characters from source program and

recognizing tokens or basic syntactic components is done by this tool. It does so by maintaining a list of reserved words. Examples are FLEX and LEX.

Flex (fast lexical analyzer generator) or LEX are lexical phase tools which are rule based programming languages. Let us look at a simple example.

LEX Example - specifies a scanner which replaces the string “username” with the user’s login name. The following is the code, where the pattern “username” is replaced with the name got from getlogin().

```
%%

username printf(“%s”, getlogin());
```

FLEX and LEX will be discussed in detail in the forthcoming modules.

Parser generators: The input to this tool is the tokens and the context-free grammar. The output will be a typical yes or no indicating the correctness or incorrectness of the statement. The task of the syntax analyzer is to produce a representation of the source program in a form directly representing its syntax structure. Example: Bison (YACC-compatible parser generator). It is general purpose parser generator that converts grammar description for an LALR(1) CFG into a C program.

Syntax-directed translators: This tool acts as an intermediate code generatorby getting the parse tree as input and generates intermediate code as output.

Intermediate code generators: The output of the Semantic analyser gives the tree as output and the code generator gets intermediate code rules as input and uses this tree to produce functionally equivalent three address code. This could also produce a crude machine program.

Data Flow Engines: Gets intermediate code as input and produces transformed code as output. The transformed code may be optimized but is not guaranteed to produce the most efficient code.

Automatic code generators: Takes optimized intermediate code as input and produces assembly language code using the instruction set of the machine.

Example: (8 * x) / 2

```
Load a, x
Mult a, 8
Div a, 2
```

3. Summary: This module discussed the phases of the compiler for a given input and briefed on the compiler construction tools.