COMPILER CONSTRUCTION (Phases of Compiler) **Fall 2019**

Lecture 3-4: July 22,24, 2019

Instructor: K.R. Chowdhary *: Professor of CS*

3.1 Phases of compiler

If we look into the analysis and synthesis parts of the compiler, there are further sub-parts of these blocks, each of these transforms one representation into another. These sequence of operations are called phases of the compiler. The symbol table is used by all the phases of the compiler. Usually, the analysis part is machine independent, hence it can be common for many hardware (CPUs) of any given programming language. In such a scenario, the back end can be better optimized to suit the machine architecture. Fig. 3.1 shows all the phases of a compiler.

What varies from compiler to compiler is optimization phase, which may be complex, simple, or may be missing. Some of the compilers generate directly the machine code of target machine, others generate assembly language program. When, it is assembly language program generated, there may optionally be an optimizer after that, which is machine specific optimizer.

3.1.1 Lexical Analysis

The lexical analysis is first part of analysis part, and it produces the tokens (lexemes) from the input program, by meaningful grouping of characters input (scanner). There is a slight difference between lexeme and token. Each *token* has format: $\langle token\text{-}name, attribute\text{-}values$. These lexemes are passed to the next phase, i.e., syntax analyser. The token-name is abstract symbol, that is used during the syntax analysis, and the field attribute value is a pointer to an entry in the symbol table. The contents at this entry is useful for semantic analysis and code generation. Consider a C language statement,

$$dist = initdist + hours * 80;$$

The following analysis is carried out using this statement:

1. *dist* is a lexeme which is mapped to a token $\langle id, 1\rangle$, where *id* is an abstract symbol for *identifier*, and 1 point to the entry number 1 in the symbol table. This entry holds the name of identifier and its type.

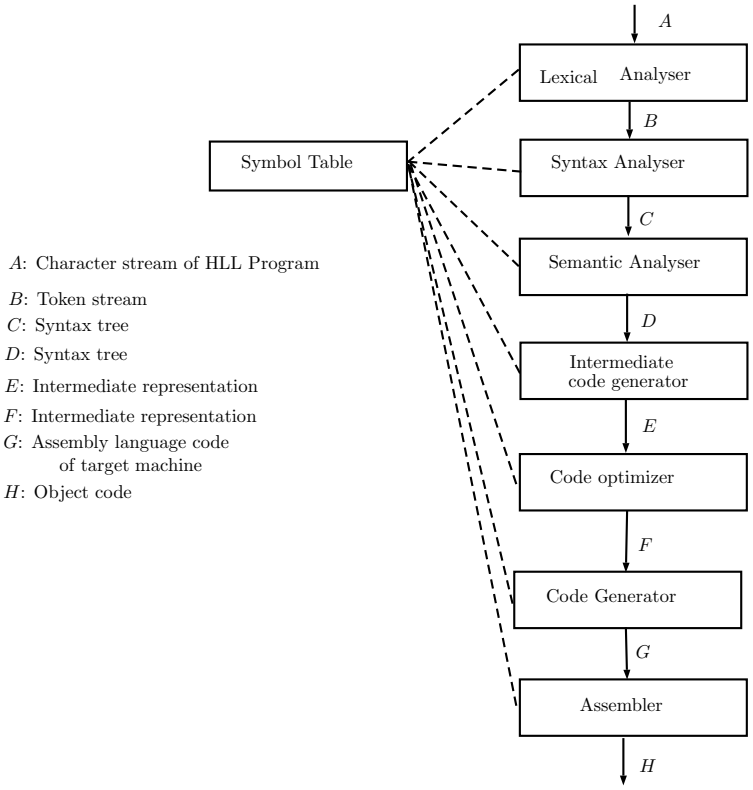

Figure 3.1: Phases of a compiler

2. The "=" is a lexeme that is mapped to the token $\langle = \rangle$, and there is nothing else, as the assignment has no attribute value.
3. The "initdist" lexeme is mapped into the token $\langle id, 2 \rangle$, where 2 indicates a pointer to 2nd entry in the symbol table, for "initdist".
4. The + lexeme is mapped into token $\langle + \rangle$.
5. *hourse* lexeme is mapped into token $\langle id, 3 \rangle$, and 3 points to entry no. 3 in the symbol table.
6. The $*$ lexeme is mapped into token $\langle * \rangle$.
7. 80 is a lexeme that is mapped into token $\langle 80 \rangle$.

Note that blanks in the above statement have not been considered, as they (including tab character) are discarded by the lexical analyzer in the context of C language.

In C, space/tab/newline separates the lexemes, which may not be the case in other languages. For example, in Fortran, the statement "DO10I" is cleection of three lexemes: "DO", "10", and "I".

Once the tokens are generated, the statement in the form of token stream (B in Fig. 3.1) is:

$$\langle id, 1 \rangle \ \langle = \rangle \ \langle id, 2 \rangle \ \langle + \rangle \ \langle id, 3 \rangle \ \langle * \rangle \ \langle 80 \rangle \tag{3.1}$$

In the above representation, $=, +, *$ are abstract symbols for assignment, addition, and multiplication, respectively.

In Fig. 3.1, we shown various phases of a compiler, where output of each phase is identified by a specific representation $(A - H)$, where A corresponds to $dist = initdist + hours * 80;$, in our example. The output of lexical analyser is set of tokens, represented by equation 3.1. Corresponding to the the value of B (output of lexical analyser), the outputs, $C, ..., H$ are shown in Fig. 3.2.

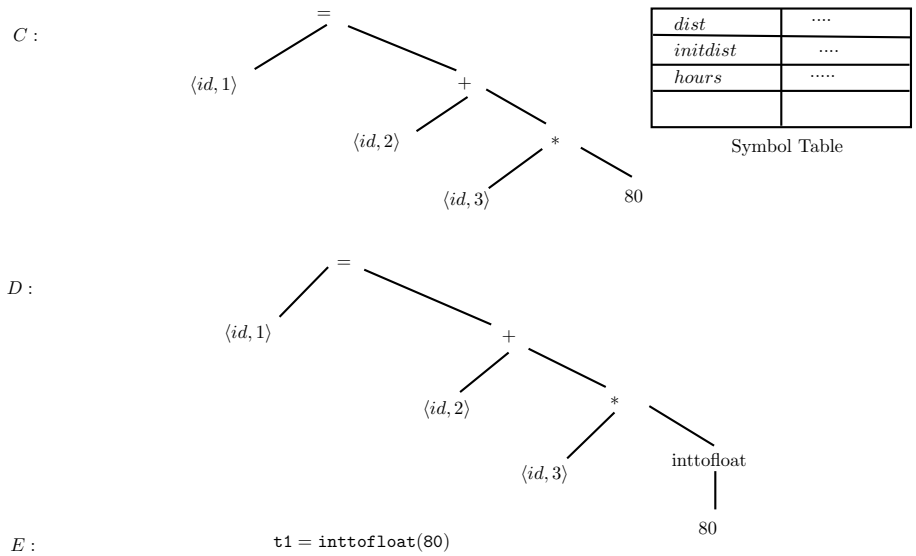

Figure 3.2: Outputs of various phases

3.1.2 Syntax Analysis

The second phase of compiler is syntax analysis (or parsing), which provides a tree like structure to the sequence of tokens produced by the lexical analyser. The syntax tree depicts the grammatical structure of the language sentence, where each interior node is an operation (assignment, arithmetic, or logical) and each child node represents the argument or value of the operation. A syntax tree of the tokens in equation 3.1 is shown in Fig. 3.2(C). The syntax tree shows the order in which operations are performed. Note that, the lower

subtree is always computed before computing the upper subtree.

The subsequent phases of the compiler use the grammatical structure of syntax tree to produce the *object/target code*.

3.1.3 Semantic analysis

The semantic analyser makes use of syntax tree and the information stored in the symbol table to check semantic consistency in the source program with respect to the language definition. This phase also collect the *type* information of the data and stores in symbol table or in the syntax tree, to use during the intermediate code generation.

An important function of semantic analysis is *type checking*, where each of the operator is checked for its matching operands. For example, in C language an array *index* is checked to be an integer. The semantic analyser will check that it is integer. The language specification permits some type conversion, called *coercion*, e.g., in Fig. 3.2(D), *inttofloat* operator converts value 80 to float by adding extra node in the parse-tree.

3.1.4 Intermediate code generation

Before the code is translated into the final object code, a compiler may construct one or more intermediate representations. One of the commonly used format is syntax tree for this representation. The intermediate code requires many properties:

1. it should be easy to produce
2. it should be easy to translate into machine code
3. the machine code translation should be efficient in time as well as space

The intermediate code may be *two-address* code or *three-address* code. We will consider three-address code for our discussions. Each operand in the three address code behaves like a *register*. We reproduce the intermediate code of Fig. 3.2(E), as below.

$$
\begin{aligned}
&\texttt{t1} = \texttt{inttofloat}(\texttt{80})\\
&\texttt{t2} = \texttt{id3} * \texttt{t1}\\
&\texttt{t3} = \texttt{id2} + \texttt{t2}\\
&\texttt{id1} = \texttt{t3}
\end{aligned}
$$

The simplest way to generate the intermediate code is to generate one intermediate code instruction for each of the operator in the syntax tree. The three address instructions have at most one instruction to the right hand side, they fix the order of execution, and the multiplication operator precedes the three instructions.

3.1.5 Code optimization

The optimization may be done either before generating the assembly code or at both places (i.e., after and before). When it is before assembly code generation, it is called machine

Table 3.1: Code optimization results of typical program

Original code	Optimized code
$x = ...$	$x = ...$
$y = ...$	$y = ...$
$w = 1$	$w = 1$
$for\ i = 1\ to\ n$	$t = 2 * w * x * y$
$read\ z$	$for\ i = 1\ to\ n$
$w = 2 * w * x * y * z$	$read\ z$
end	$w = w * t * z$
	end

independent code optimization, at the level of intermediate code. The objective of code optimization is to produce better object code, i.e., the one with better execution efficiency, smaller size of code, so that it consumes lesser memory as well as power. For example, the Fig. 3.2(F) shows the following optimized intermediate code for the four line original code.

$$\begin{aligned} \texttt{t1} &= \texttt{id3} * \texttt{80.0} \\ \texttt{id1} &= \texttt{id2} + \texttt{t1} \end{aligned} \tag{3.2}$$

Note that, optimized code is generated using intermediate code and not through the syntax-tree.

There is lot of variations of optimization from compiler to compiler. The compilers that do this task more are called *optimizing compilers*, and significant amount is spent in this phase.

3.1.6 Assembler

The assembly code generated for the current example is shown in Fig. 3.2 (G). This code, is corresponding to equation 3.2, which shows that register `R1` has been allocated. The assembly code is self explanatory.

```
LD R1, id3
MULT R1, R1, #80.0
LD R2, id2
ADD R1, R1, R2
ST id1, R1
```

3.1.7 Code generations

The code generator phase takes as its input the intermediate code and produce the object/target code of the machine (see Fig. 3.2 (H)), which is assembly language code. If the assembler is not used, the code generator phase directly produces the machine language code. An important aspect of code generation is judicious assignment of CPU registers to hold the variables.

3.1.8 Grouping of Phases into Passes

We have discussed the phases of a compiler, each one as a logical organization of compiler. In the real implementation of the compiler, several phases may be organized as a pass. A pass will read an input file and write an output file. However, in a phase, it may read the input from a file or memory and write the same into a file/memory. For example, the front end phases like lexical analysis, syntax analysis, semantic analysis and intermediate code generation can be grouped into a single a single pass. The back-end may correspond to one pass for code generation for a particular machine. It is possible to make different compilers by combining the front end with back end of different machines.

References

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) by Alfred V. Aho , Monica S. Lam, et al., Sep 10, 2006

[2] Compiler design in C (Prentice-Hall software series) by Allen I Holub, Jan 1, 1990

[3] Engineering a Compiler, by Keith D. Cooper and Linda Torczon, Morgan Kaufmann Publishers, 2004.