Chapter

21 *Basic Compiling Theory*

This chapter gives the ground basis for basic compiling theory. It will give an introduction to compiling theory as it is the corner-stone on which TTCN Access is built upon. See chapter 22, *TTCN Access*.

This chapter is intended to be read by developers of executable test suites, translator developers and test result analyzers.

> **Note: UNIX only**
>
> **TTCN Access is only available on UNIX.**

Basic Compiling Theory

A programming language can be defined by describing what its programs look like (the *syntax* of the language) and what its programs mean (the *semantics* of the language).

For specifying the syntax of a language, such as TTCN, the widely used notation called *context-free grammar*, or BNF (Backus-Naur Form) is used. (A more precise definition of context-free grammar will be defined in "Syntax Definition" on page 938).

To describe the semantics of a language is more difficult, as no appropriate notation for semantic description is available. Consequently, when specifying the semantics of a language an informal description technique has to be used. TTCN uses *Semantic actions* that, in natural language, describes the actual semantic given a specific context.

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of a programs. A grammar oriented compiling technique, known as syntax-directed translation, is very helpful for organizing a compiler front end.

The Compiler

Conceptually, a compiler operates in phases, each of which transforms the source program from one representation to another. Often, some of the phases may be grouped together and the intermediate representation between the grouped phases need not be explicitly constructed.

The first three phases of a compiler, forming the bulk of the analysis portion is often called *front end*. The front end consist of those phases that depend primarily on the source language and are largely independent of the target language. This front end normally includes lexical and syntactic analysis, the creation of the symbol table, semantic analysis and the generation of intermediate code.

The last phases of a compiler, the *back end*, include those portions of a compiler that depend on the target language. In the back end, one will find code optimization, code generation along with the necessary error handling and symbol-table operation.

This chapter will discuss and explain the concept of the front end thoroughly. The back end will be mentioned just for completeness. The reason for this is quite obvious:

TTCN Access is the front end to a TTCN compiler.

Lexical Analyzer

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

Since the lexical analyzer is the part of a compiler that reads the source text, it may also perform certain secondary tasks at the user interface. One such task is stripping out comments and white space in the form of blank, tab, and newline character. Another is correlating error messages from the compiler with the source program. For example, the lexical analyzer may keep track of the number of newline characters seen, so that a line number can be associated with an error message.

In lexical analysis the stream of characters making up the program is read from left-to-right and grouped into tokens that are sequences of characters having a collective meaning.

Example 160 ––

For example, the characters in the assignment statement below:

```
position := initial + rate * 60
```

would be grouped into the following tokens:

1. The identifier `position`
2. The assignment symbol `:=`
3. The identifier `initial`
4. The plus sign
5. The identifier `rate`
6. The multiplicand sign
7. The number `60`

––

The blanks separating the characters of these tokens would normally be eliminated during lexical analysis.

Syntax Definition

In this section, a notation called a *context-free* grammar (grammar for short) is introduced. This notation is used for specifying the syntax of a language.

A grammar naturally describes the hierarchical structure of many programming language constructs. For example, an if-else statement in C has the form as below.

```
if ( expression ) statement else statement
```

That is, the statement is the concatenation of the keyword *if*, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword *else*, and another statement. Using the variables *expr* to denote an expression and the variables `stmt`, `stmt1` and `stmt2` to denote three (possible different) statements, this structuring rules can be expressed as

```
stmt -> if ( expr ) stmt1 else stmt2
```

in which the arrow may be read as “can have the form”. Such a rule is called a *production*. In a production lexical elements like the keyword **if** and the parentheses are called *tokens*. Variables like `expr` and `stmt2` represent sequence of tokens and are called *nonterminals*.

A *context-free grammar* has four components:

- A set of tokens, known as *terminal* symbols.
- A set of nonterminals.
- A set of productions where each production consists of a nonterminal, called the *left side* of the production, an arrow, and a sequence of tokens and/or nonterminals, called the *right side* of the production.
- A destination of one of the nonterminals as the *start* symbol.

Through this chapter, examples and pictures will be present in order to clarify a topic or explain a specific concept. As TTCN, at this stage, has a too large grammar definition to be included in this document, a language called SMALL is defined that in this chapter will be used for examples and discussions.

SMALL will be defined as a language only consisting of digits and the plus and minus signs, e.g., 9-5+2, 3-1 and 7. In the language SMALL, a plus or minus sign must appear between two digits, and expressions

as above will be referred to as “lists of digits separated by plus or minus sign”.

The following grammar specifies the syntax of this language with *list* as its start symbol:

```
1:list -> list + digit
2:list -> list - digit
3:list -> digit
4:digit -> 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

From production (3), a single digit by itself is a *list*. Productions (1) and (2) express the fact that if one take any *list* and follow it by a plus or a minus sign and then another digit one has a new *list*. For example, it is possible to deduce that 9-5+2 is a list as follows:

a) 9 is a list by production (3), since 9 is a digit.

b) 9-5 is a list by production (2), since 9 is a list and 5 is a digit.

c) 9-5+2 is a list by production (1), since 9-5 is a list and 2 is a digit.

This reasoning is illustrated by the tree in the next figure. Each node in the tree is labelled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the left side of the production, the children to the right side. Such trees are called *parse trees* and are discussed in the next chapter.

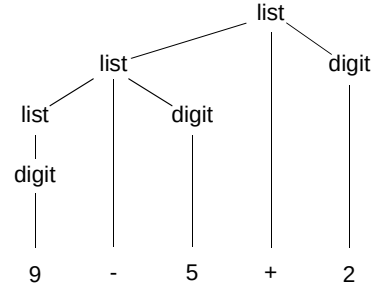

Figure 177: A parse tree for 9-5+2

Syntax Analyzer

Every programming language has rules that prescribe the syntactic structure of well-formed programs. In Pascal, for example, a program is made out of blocks, a block out of statement, a statement out of expressions, an expression out of tokens, and so on. The syntax of programming language constructs can be described by context-free grammars or BNF notation. Grammars offer significant advantages to both language designers and compiler writes:

- A grammar gives a precise, yet easy-to-understand, syntactic specification of a programming language.
- From certain classes of grammars it is possible to automatically construct an efficient parser that determines if a source program is syntactically well formed.
- A properly designed grammar imparts a structure to a programming language that is useful for translation of source programs into correct object code and for the detection of errors.
- Languages evolve over a period of time, acquiring new constructs and performing additional task. These new constructs can be added to a language more easily when there is an existing implementation based on a grammatical description of the language.

During syntax analysis the tokens of the source program are grouped into grammatical phrases that are used by a compiler to synthesis the output. Usually, the grammatical phrase of the source program is represented by a *parse tree*.

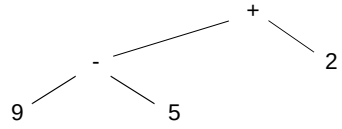

Figure 178: A syntax tree for the assign statement 9-5+2

Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production as:

```
A -> X Y Z
```

then a parse tree may have an interior node labelled A with three children labelled X, Y and Z, from left to right:

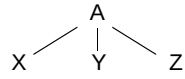

Figure 179: Parse tree

Formally, given a context-free grammar, a *parse tree* is a tree with the following properties:

1. The root is labelled by the start symbol.
2. Each leaf is labelled by a token or empty.
3. Each interior node is labelled by a nonterminal.
4. If A is the nonterminal labelling some interior node, and X1, X2, ..., Xn (where Xi are terminals or nonterminals) are the labels of the children of that node, then
   `A -> X1 X2 ..., Xn`
   is a production in the grammar.

The leaves of a parse tree read from left to right form the yield of the parse tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse tree. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if *a* and *b* are two children with the same parent, and *a* is to the left of *b*, then all descendants of *a* are to the left of descendants of *b*.

Another definition of a language generated by a grammar is as the set of strings that can be generated by some parse tree. The process of finding a parse tree for a given string of tokens is called *parsing* that string.

Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar. In discussing this topic, it is helpful to think of a parse tree being constructed, even though a compiler may not actually construct such a tree. However, a parser must be capable of constructing the tree, or else the translation cannot be guaranteed correct.

Intermediate Code

After syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program. This intermediate representation can be seen as a program for an abstract machine. In the context of the TTCN suite and TTCN Access, the intermediate representation is the actual parse tree generated by the Analyzer. For more information see chapter 27, *Analyzing TTCN Documents (on UNIX)*.

Code Generator

The final phase of a compiler is the generation of target code, that is transforming the intermediate code into something useful such as an executable. As TTCN Access is an application programmers interface towards the intermediate representation, the parse tree, it is the TTCN Access application that will define the actual transformation of the intermediate code. It can be an executable as well as a reporter, interpreter, etc.

The Phases of a Compiler

Conceptually, 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 the next picture. In practice, some of the phases may be grouped together and the intermediate representation between the grouped phases need not be explicitly constructed.

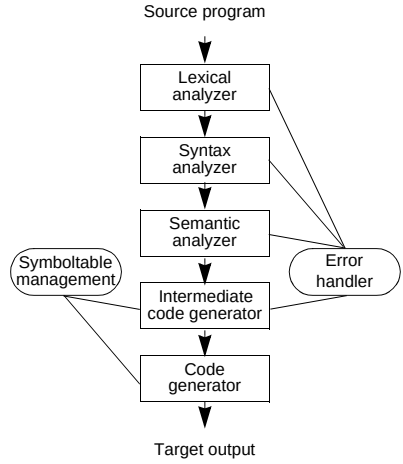

Figure 180: The phases of a compiler

The first three phases, forming the bulk of the analysis portion of a compiler has previously been discussed. Two other activities, symbol-table management and error handling, are shown interacting with the five phases of lexical analysis, syntax analysis, semantic analysis, intermediate code generation and code generation. Informally, the symbol-table management and the error handler are also called *phases*.

Symbol Table Management

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. These attributes may provide information about the identifier as its type, its scope (where in the source program it is valid) and, in the case of procedure names, such things as the number and type of its arguments, the method for passing each argument (e.g., by reference), and the type returned, if any.

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 a specific identifier quickly and to store or retrieve data from that record.

When an identifier in the source program is detected by the lexical analysis, the identifier is entered into the symbol table. However, the attributes of an identifier cannot normally be determined during lexical analysis. The remaining phases enter information about identifiers into the symbol table and then use this information in various ways. For example, when doing semantic analysis, the compiler needs to know what types the identifiers have, so it can be checked that the source program uses them in a valid way.

Error Detection and Reporting

Each 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 syntax and semantic analysis phase usually handle a large fraction of the errors detectable by the compiler. The lexical phase can detect errors where the characters remaining in the input do not form any token of the language. During semantic analysis the compiler tries to detect constructs that have the right syntactic structure but no meaning to the operations involved, e.g., when adding two identifiers, one of which is the name of an array and the other is a name of a procedure.