5 Semantic and Type Analysis

In Chapter 3 and 4 different analyzing methods have been introduced. All these analyzers (lexical as well as syntax analyzers) are not concerned with the semantics or the meaning of given programs. A compiler has to check not only the syntactical correctness of a given source code, but also whether the semantics correspond to that of the programming language. This means, that semantic analysis has to guarantee that all context-sensitive rules of the programming language are considered.

An example of such a context-sensitive rule of a programming language is that in languages like PASCAL or MODULA-2 identifiers have to be declared before they are used. Symbol tables are used to check, if an identifier already has been declared, as explained in Chapter 3. Thus, symbol tables and their usage can be seen as a part of the semantic analysis process and, therefore, it is obvious that semantic analysis can be carried out in parallel to syntax analysis (or must be done in parallel at least in certain situations).

In general, the semantic analysis of a compiler uses the information of the syntax analysis in combination with the semantic rules of the programming language to generate an internal representation of the source code which is to be compiled. This generation of an internal representation means that the compiler has to interpret the meaning of the source code and, therefore, that the compiler semantically analyzes the syntactical structures of the source code as they are recognized by the scanner and parser. The internal representation is an intermediate code which will be passed on to the code generator.

An introduction to intermediate codes is given below before the so-called syntax-directed translation will be introduced.
5.1 Intermediate Codes

The term intermediate code indicates a code structure which lies in complexity between high-level language source code and machine code and therefore, intermediate codes can be understood as an interface between the code generator and all the previous phases of the compiler.

Intermediate codes can be regarded as the machine code of an (ideal) hypothetical computer. The usage of such an abstract machine level has the advantage that

- the compiler itself is independent of the target machine and by this is more portable, because all the characteristics of the target hardware are considered in the code generator;

- some optimization strategies (e.g. optimizations depending on register allocations) are easier to perform on intermediate code than on source code.

However, the code which can be generated from an intermediate code will be in general less efficient than a directly generated machine code because of the additional translation level.

In the following, three kinds of intermediate code are introduced which are often used in compilers.

Postfix Notation

Postfix notation is a very simple notation which places an operator on the right end of an expression, i.e. directly behind the operands instead of between the operands. For example, the expression

\[ x + y - x^*y \]

is in postfix notation

\[ xy + xy^* - \]

since \( xy^+ \) represents the infix expression \( x + y \) and \( xy^+ \) represents \( x^*y \), etc.

Postfix notation is normally used for stack machines since it can be handled very easily using a stack: When scanning a postfix notation from left to right, each time an operand occurs it will be pushed onto the stack. The occurrence of an operator with \( m \) operands means that the \( n \)-th operand will be found in position \( m - n \) below