Lecturer (5)

1.3.4 Semantic Analysis

A semantic analyser takes its input from the syntax analysis phase in the form of a parse tree and a symbol table. Its purpose is to determine if the input has a well-defined meaning; in practice semantic analysers are mainly concerned with type checking and type coercion based on type rules. Typical type rules for expressions and assignments are:

Expression Type Rules. Let exp be an expression.

(a) If exp is a constant then exp is well-typed and its type is the type of the constant.
(b) If exp is a variable then exp is well-typed and its type is the type of the variable.
(c) If exp is an operator applied to further subexpressions such that:

(i) the operator is applied to the correct number of subexpressions,
(ii) each subexpression is well-typed and
(iii) each subexpression is of an appropriate type,
then exp is well-typed and its type is the result type of the operator.

Assignment Type Rules. Let var be a variable of type T1 and let exp be a well-typed expression of type T2. If

(a) T1 = T2 and
(b) T1 is an assignable type

then var assign exp is a well-typed assignment.
For example, consider the following code fragment:

intvar := intvar + realarray

where intvar is stored in the symbol table as being an integer variable, and realarray as an array or reals. In Pascal this assignment is syntactically correct, but semantically incorrect since + is only defined on numbers, whereas its second argument is an array. The semantic analyser checks for such type errors using the parse tree, the symbol table and type rules.

1.3.5 Error Handling

Each of the six phases (but mainly the analysis phases) of a compiler can encounter errors. On detecting an error the compiler must:

• report the error in a helpful way,

• correct the error if possible, and

• continue processing (if possible) after the error to look for further errors.

Types of Error. Errors are either syntactic or semantic:
Syntax errors are errors in the program text; they may be either lexical or grammatical:

(a) A lexical error is a mistake in a lexeme, for examples, typing tehn instead of then, or missing off one of the quotes in a literal.
(b) A grammatical error is a one that violates the (grammatical) rules of the language, for example if x = 7 y := 4 (missing then).

Semantic errors are mistakes concerning the meaning of a program construct; they may be either type errors, logical errors or run-time errors:

(a) Type errors occur when an operator is applied to an argument of the wrong type, or to the wrong number of arguments.

(b) Logical errors occur when a badly conceived program is executed, for example:
while x = y do ... when x and y initially have the same value and the body of loop need not change the value of either x or y.

(c) Run-time errors are errors that can be detected only when the program is executed, for example:

var x : real; readln(x); writeln(1/x)

which would produce a run time error if the user input 0.

Syntax errors must be detected by a compiler and at least reported to the user (in a helpful way). If possible, the compiler should make the appropriate correction(s). Semantic errors are much harder and sometimes impossible for a computer to detect.

1.3.6 Intermediate Code Generation

After the analysis phases of the compiler have been completed, a source program has been decomposed into a symbol table and a parse tree both of which may have been modified by the semantic analyser. From this information we begin the process of generating object code according to either of two approaches:

(1) generate code for a specific machine, or
(2) generate code for a 'general' or abstract machine, then use further translators to turn the abstract code into code for specific machines.

Approach (2) is more modular and efficient provided the abstract machine language is simple enough to:

(a) produce and analyse (in the optimisation phase), and
(b) easily translated into the required language(s).

One of the most widely used intermediate languages is Three-Address Code (TAC).

TAC Programs. A TAC program is a sequence of optionally labelled instructions. Some common TAC instructions include:

(i) var1 := var2 binop var3
(ii) var1 := unop var2
(iii) var1 := num

(iv) goto label

(v) if var1 relop var2 goto label

There are also TAC instructions for addresses and pointers, arrays and procedure calls, but will use only the above for the following discussion.

Syntax-Directed Code Generation. In essence, code is generated by recursively walking through a parse (or syntax) tree, and hence the process is referred to as syntax-directed code generation. For example, consider the code fragment: