J. Webster (ed.), *Wiley Encyclopedia of Electrical and Electronics Engineering*
Copyright © 1999 John Wiley & Sons, Inc.

PROGRAM COMPILERS

A *compiler* is special software for taking a program (called the *source program*) written in a particular (programming) language as input and producing a program (called the *target program*) in another language as output. For example, a Pascal compiler translates a Pascal program into assembler (or machine) code, and a Java compiler translates a Java program into Java bytecodes. The output of a compiler has to be semantically equivalent to its input, that is, if the source program and the target program are executed on the same data, then they deliver exactly the same results. The compilation process can be executed in one or more successive stages (passes).

Application Fields

The area of compiler construction is one of the best-known disciplines of computer science. Compiler techniques are strongly influenced by results of programming language theory (1) and formal language (and automata) theory (2); see also AUTOMATA THEORY. The classical compiler application field is the translation of programming languages like Fortran, C, C++, and Pascal into machine code of a certain processor. Nowadays we can find further examples:

- Document description languages like TEX, LATEX, HTML, SGML, XML
- Database languages to formulate requests, for example, a sequence of SQL statements implemented as stored procedures
- Design languages for very large scale integration (*VLSI*) to describe a chip layout
- Natural-language processing
- Protocol languages in distributed systems—for example, execution of remote procedure calls requiring the translation (marshalling) of the call into a representation that can be transmitted over a given network

All these areas profit from compiler techniques.

Structure of a Compiler

A compiler is a complex program, which can be divided up into consecutive phases (modules). Each phase transforms semantically equivalent a program representation into another one (see Fig. 1). Fitting together all parts, we obtain the whole compiler, translating a program written in a particular programming language into machine-executable code. In practice, some phases are executed in parallel or pipelined form, but for better understanding we will describe each phase separately as a single unit.

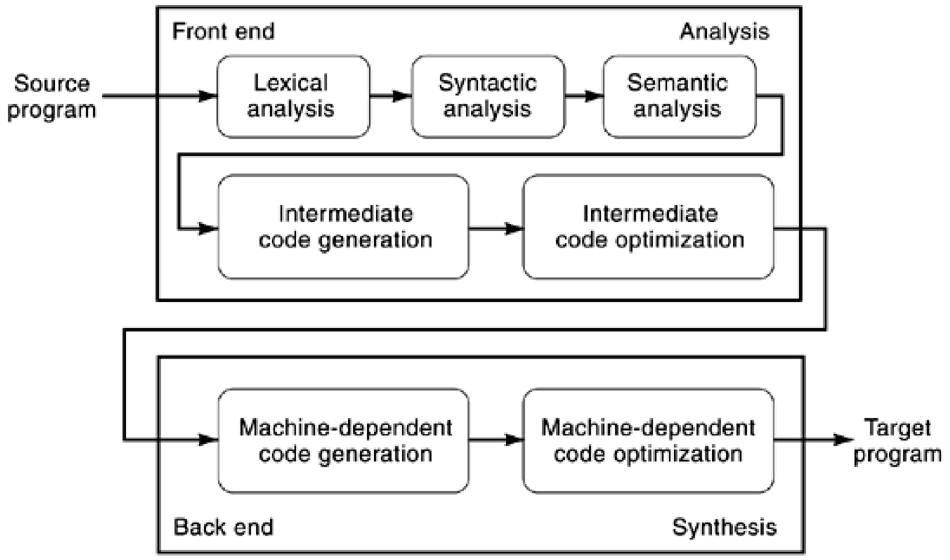
Front end
Analysis
Source program
Lexical analysis
Syntactic analysis
Semantic analysis
Intermediate code generation
Intermediate code optimization
Machine-dependent code generation
Machine-dependent code optimization
Target program
Back end
Synthesis

Fig. 1. The phases of a compiler.

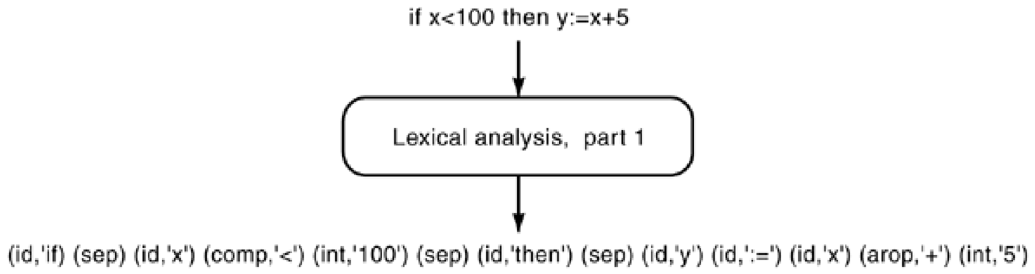
if x<100 then y:=x+5
Lexical analysis, part 1
(id,'if) (sep) (id,'x') (comp,'<') (int,'100') (sep) (id,'then') (sep) (id,'y') (id,':=') (id,'x') (arop,'+') (int,'5')

Fig. 2. Lexical analysis, part 1: Translating a program statement into a token stream.

Lexical Analysis. A program written in a particular language is usually stored in a file as a sequence of characters. The character stream necessarily hides the syntactical structure, which must be reconstructed by the lexical and syntactic analysis.

The task of the *lexical analysis* is to

- Inspect the character stream to see that only well-defined characters are used
- Decompose the character stream into a sequence of lexical units belonging semantically together, called *tokens*
- Delete all irrelevant characters and sequences of characters
- Create a symbol table to handle the identifier names of the token

The module processing the lexical analysis is called the *scanner*. Let us consider the lexical analysis in more detail. First the character stream is transformed into a sequence of symbols (tokens); a simple example is given in Fig. 2.

Table 1. Example of Distinguished Token Classes

T1	Identifiers
T2	Reserved words (*begin, if, while,* :=, etc.)
T3	Boolean operators (AND, OR)
T4	Arithmetic operators (+,-,∗,/)
T5	Comparison operators (<,>,=,>=,. . .)
T6	Integer numbers
T7	Real numbers
T8	Special characters (blanks, linefeed, etc.)
T9	Comments

The symbol sequence can again be refined so that the class of identifiers is subdivided into separate classes of user-defined identifiers and predefined identifiers (reserved words or keywords) of the language. Furthermore, separator and comment symbols are deleted. Then a symbol table is created where all identifier names are stored. The various occurrences of a token representing a user-defined identifier are numbered and referenced in the output token stream by an index to the symbol table where the concrete name is stored. Usually we have a fixed number of distinguished token classes, e.g., see Table 1.

Figure 3 illustrates the translations process based on the defined token classes. The structure of a token can formally defined by *regular expressions* (3,4,5,6). For example, each user-defined identifier of a programming language has to be a character string in which the first character is a letter and the following ones are letters or digits. Formally that can be defined by

$$\begin{aligned}\text{identifier:} &= \text{identifier } \{ \text{ letter } \mid \text{ digit } \}\\ \text{digit:} &= 0|1|\ldots|9\\ \text{letter:} &= \text{A}|\text{B}|\ldots|\text{Z}|\text{a}|\text{b}|\ldots|\text{z}\end{aligned}$$

where | represents OR and $\{x\}$ the set of finite sequences of symbol x.

In the implementation of the lexical analysis the model of *deterministic finite automata* (2,3,4,7) will be used to process the pattern matching of the token; see also AUTOMATA THEORY.

Syntactic Analysis. The structure of expressions, statements, or declarations cannot be determined by the lexical analysis. Hence a more powerful analysis is required: *syntax analysis* or *syntactic analysis*. The module processing it is called the *parser*. The task of a parser consists of

- Evaluating the syntactic structure (in the form of an *abstract syntax tree*) of a program
- Recognizing and locating syntactical errors
- Sending detailed error messages to the programmer

A program (in token stream representation) can be considered as a word of an appropriately defined *context-free language*. By techniques of formal language theory the correctness of the program structure can be proven, and for output an abstract syntax tree is evaluated. To go into more detail we need the definition of context-free grammars, context-free languages, derivation trees, and abstract syntax trees. We repeat some definitions given in the article AUTOMATA THEORY. The reader familiar with formal language theory or the reader only interested in getting a general idea of compilers can skip the formal definitions.

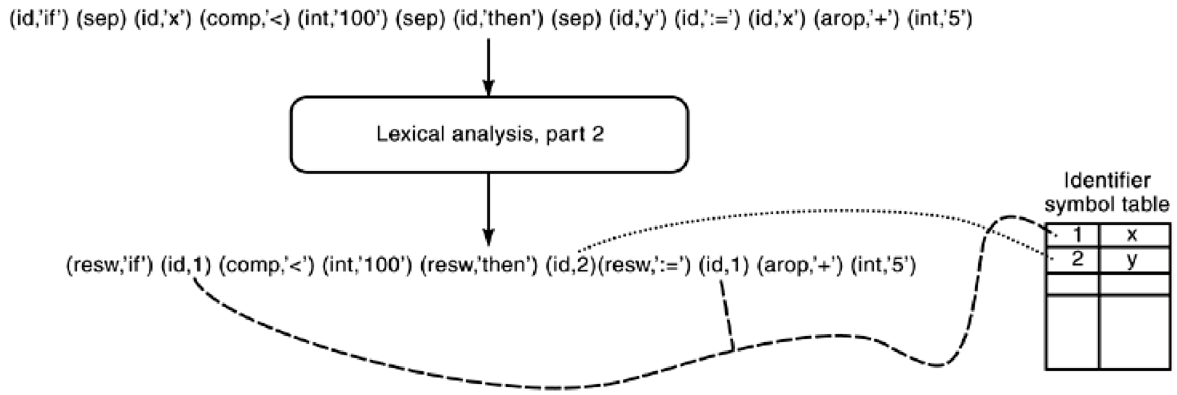
(id,'if') (sep) (id,'x') (comp,'<) (int,'100') (sep) (id,'then') (sep) (id,'y') (id,':=') (id,'x') (arop,'+') (int,'5')
Lexical analysis, part 2
(resw,'if') (id,1) (comp,'<') (int,'100') (resw,'then') (id,2)(resw,':=') (id,1) (arop,'+') (int,'5')
Identifier symbol table
1 x
2 y

Fig. 3. Lexical analysis, part 2: Refining the token stream by using an identifier symbol table.

An *alphabet* is finite set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of *symbols*. A finite sequence $x_1x_2\ldots x_k$ of symbols ($x_i \in \Sigma$, $k \in N$) is called a *word* of *length k*. We include the case $k = 0$ and say that there is a (unique) word of length 0, which will be called the *empty word* and will be denoted by ε. The set of all finite words that can be formed with symbols from Σ, including the empty word ε, will be denoted by $\Sigma*$. Mathematically we may define $\Sigma* = \cup_{k \varepsilon N} \Sigma^k$ with $\Sigma^0 = \{\varepsilon\}$.

A *grammar* $G = (N, T, S, P)$ is a structure where N and T are disjoint finite alphabets, $S \in N$ is the *initial symbol*, and $P \subseteq (N \cup T)* \times (\mathrm{N} \cup \mathrm{T})*$ is a finite set of *rules*. The elements of N are called *nonterminal* and those of T are called *terminal* symbols. The set of all symbols of the grammar G is denoted by V, that is, $V{=}N \cup T$ and $N \cap T = \emptyset$. The initial symbol $S \in N$ is also called the *start symbol* of G.

We define the *one-step derivation* relation that relates pairs of $V*$ as follows: $x \rightarrow y$ iff there is a rule $(u,v) \in P$ such that y is the result of applying the rule (u,v) to x. We extend this relation to its so-called *reflexive* and *transitive closure* $\rightarrow* \subseteq V* \times V*$ by defining $x \rightarrow* y$ iff there is a finite sequence of one-step derivations $x \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \cdots \rightarrow x^{(n)} \rightarrow y$ that transforms x into y or if $x = y$. The sequence $x \rightarrow x^{(1)} \rightarrow x^{(2)} \rightarrow \cdots \rightarrow x^{(n)} \rightarrow y$ is called a *derivation* of y from x. A rule $(u,v) \in P$ is also simply denoted as $u \rightarrow v$.

A sequence $x \in V*$ that can be derived from S is called a *sentential form* of G, and if the sentential form only consists of terminal symbols ($x \in T*$), then x belongs to the language defined by G. So G defines (generates) the *language* $L_G = \{x \in T* \mid S \rightarrow* x\}$.

A *grammar* $G = (N, T, S, P)$ is called *context-free* iff $P \subseteq N \times V*$. This means that the rules have just one nonterminal symbol on the left hand side. A *language* $L \subseteq T*$ is called *context-free* iff there exists a context-free grammar G that generates L.

For a context-free grammar a derivation may also be represented by a *tree* where the nodes are labeled with the symbols of the grammar. The root of the tree is labeled with the initial symbol, and if a node is labeled with a nonterminal symbol $X \in N$ and in one step X is replaced by the right-hand side of a rule $X \rightarrow v_1v_2 \ldots v_k$, then the node has exactly k successor nodes labeled with $v_1, v_2, \ldots, v_k$. A node labeled with a terminal symbol has no successor. Such a tree is called a *derivation tree* or, in the case of programming languages, an *abstract syntax tree*.

For our example we represent the translation of the token stream into its abstract syntax tree in Fig. 4. If the evaluation proceeds correctly, then the program is syntactically correct; otherwise the evaluation process breaks down with an error, that is, the derivation is not possible. This implies also that a more or less precise incorrect program point is discovered.

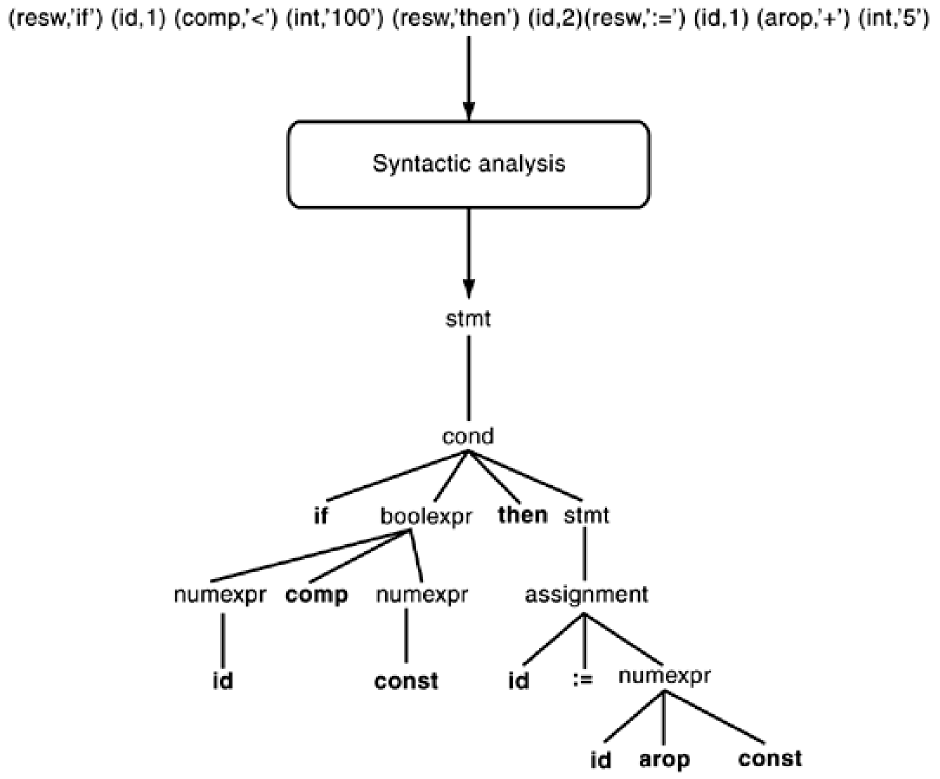
(resw,'if') (id,1) (comp,'<') (int,'100') (resw,'then') (id,2)(resw,':=') (id,1) (arop,'+') (int,'5')
Syntactic analysis
stmt
cond
if
boolexpr
then
stmt
numexpr
comp
numexpr
assignment
id
const
id
:=
numexpr
id
arop
const

Fig. 4. Syntactic analysis: Translating the token stream into an abstract syntax tree.

In practice, the above-defined model of context-free grammars is too general to provide efficient syntax analysis. More restricted forms of context-free grammars are used in real compilers. Properties like unambiguity (for each member of the generated context-free language there exists exactly one derivation tree) and run-time efficiency of the analysis are very important.

Usually grammars generating deterministic context-free languages are employed, because the corresponding parser can be generated automatically and is easier to implement. In addition, the run-time efficiency of the generated parser is pretty good. Linear run-time complexity of such a parser can be obtained (3,7,8). Syntax analysis can be divided into two classes: *top-down* and *bottom-up*. Both of them can analyze the program from left to right and construct an abstract syntax tree.

Top-down Syntax Analysis. The parser begins its work with the start symbol of the grammar (see Fig. 5). Let the program be of the form $t_1t_2 \ldots t_n$, the first (leftmost) symbol (t_1) of the program (in token-stream form) be the so-called actual token *acttok*, and the start symbol be the so-called actual grammar symbol *actgra*.

(1) The parser predicts a grammar rule to be applied with *actgra* as left-side nonterminal. Let $actgra \rightarrow v_1v_2 \ldots v_k$ be the selected rule. The actual grammar symbol *actgra* is now v_1.

(2) The parser compares the actual token *acttok* and the actual grammar symbol *actgra*.

   a. If they are equal, then the selected rule is deemed to be the right one and the actual token will be accepted. The token following the actual one will become the actual token, and the following grammar symbol will become the actual grammar symbol.

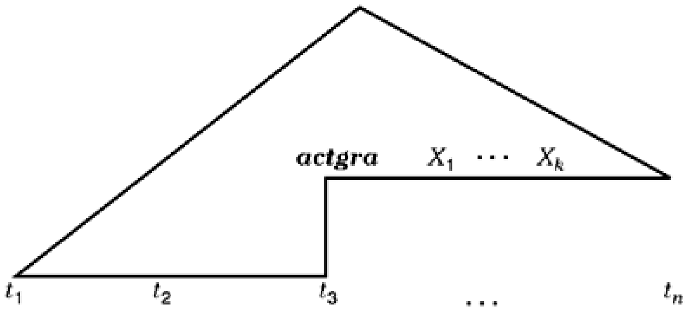

Fig. 5. Top-down syntax analysis: Constructing a derivation in top-down manner.

b. If the actual grammar symbol *actgra* is a nonterminal and a corresponding rule with *actgra* as left-side nonterminal exists, then again a rule will be selected. Let *actgra* $\rightarrow w_1 w_2 \ldots w_m$ be the selected rule. The actual grammar symbol *actgra* changes now to w_1. Continue with the comparison.
c. If the actual grammar symbol *actgra* is a nonterminal and no corresponding rule exists, then the previous selection of a grammar rule was incorrect. Select another one, and continue with the comparison. If all alternatives of the previous rule selection are exhausted, then one of the earlier rule predictions was wrong. The parser then performs the process of rule prediction backwards (also called *backtracking*) until it finds an alternative grammar rule still to be investigated, and goes on.
d. If no possible rule prediction can lead to a correct derivation tree, then the parser reports an error.

To illustrate the algorithm consider the following example. Let G=($\{E, T, F\}$, {(,+,∗,),id}, P, E} with P={

(1) $E \rightarrow E + T$
(2) $E \rightarrow T$
(3) $T \rightarrow T * F$
(4) $T \rightarrow F$
(5) $F \rightarrow (E)$
(6) $F \rightarrow$ id }

describing arithmetic expressions. Deriving the abstract syntax tree for id+id∗id leads to the steps given in Fig. 6, resulting in a wrong derivation tree. The second application of rule 1 is the obstacle to generating a correct derivation. Hence the derivation must be reset to the situation before the second application of rule 1 has taken place. Figure 7 depicts the correct derivation tree.

A parser working like the above-described model is quite easy to implement, but has unacceptable run-time efficiency. But more sophisticated analysis algorithms (3,4,8) hav been developed to overcome the inefficient reset (backtracking) situation. *LL*(k) *grammars* (a special form of deterministic *context-free grammars*) allow us to construct a parser that determines the grammar rules in a unique (deterministic) manner. By a so-called lookahead (examining the following k symbols of the actual grammar symbol), the reset situation can be avoided.

Bottom-up Syntax Analysis. Similarly to top-down syntax analysis, a bottom-up parser analyzes the program from left to right, but the construction of the derivation tree happens in another way: by using an

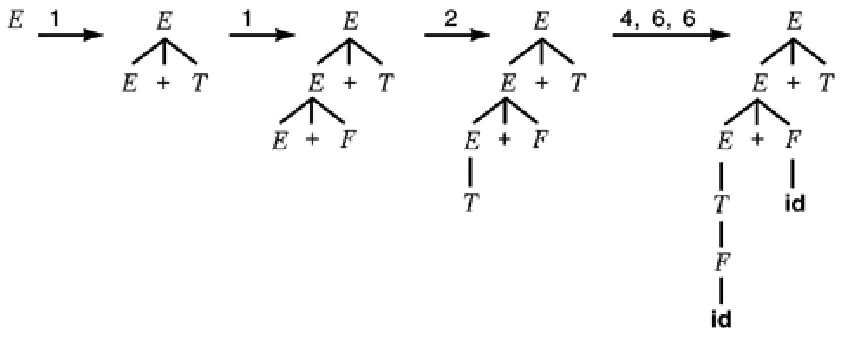

Fig. 6. A wrong derivation for the expression id+id∗id.

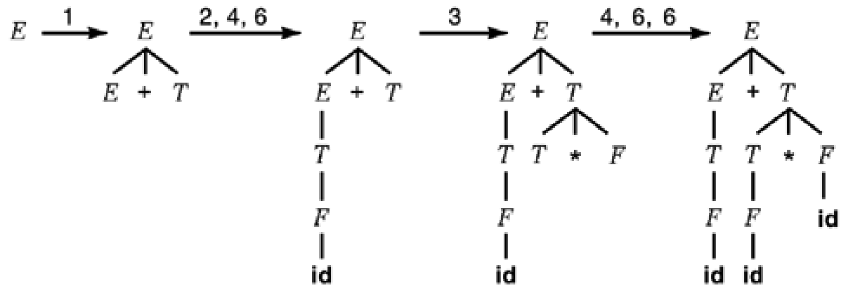

Fig. 7. A correct derivation for the expression id+id∗id.

additional (pushdown) store in which symbols (nonterminals and terminals) are stored until the right side of a grammar rule is being generated. The parser essentially processes two operations:

- *Shifting* the next program symbol to the store
- Finding a grammar rule with right side corresponding to the stored symbols or to the right part of them, and then *reducing* (i.e., replacing) the corresponding symbols by the nonterminal of the left side of the grammar rule

The parsing process starts with a shift of the first program token to the store. Next, a further shift or a reduce operation takes place.

Parsers working in this manner are also called *shift–reduce parsers*. The name *bottom-up parser* comes from the direction of the derivation-tree construction. The crucial point in the shift–reduce parsing process is again the selection of an appropriate grammar rule, i.e., the decision whether a shift or a reduce operation should be processed next when both are possible. If a decision turns out to be wrong, then backtracking (similar to the reset situation of the top-down parser described above) takes place to investigate an alternative derivation tree. In Table 2 the parsing of the arithmetic expression id∗id, taken from Ref. 3, exemplifies crucial situations.

LR(k) *grammars* (3,4,8) are certain forms of context-free grammars. The corresponding parser can decide in a unique (deterministic) manner if a shift or a reduce operation must take place. The decision is based on the next k symbols to be analyzed. In contrast to LL(k)-grammar-based analysis, the LR(k) parsing algorithm analyzes all possible derivations in parallel so long as both shift and reduce are possible. The construction

Table 2: Parsing of the Arithmetic Expression id∗id

Store	Input	Comments
	id∗id	
Id	∗id	
F	∗id	Shifting ∗ would deliver a wrong derivation
T	∗id	
T∗	Id	
T∗id		
$T * F$		Shifting ∗ would deliver a wrong derivation
T		
E		Ready

of parsers corresponding to LR(k) grammars is complicated and expensive, but fortunately it is supported by compiler generator tools like Yacc (9). The input of Yacc is an LALR(1) grammar [a simpler form of LR(1) grammars (3,4,8)]. As output an appropriate parser will be evaluated.

Semantic Analysis. Context-free grammars are not powerful enough to extract certain (static) syntactic properties of a program—for example, does a variable identifier occurring in a statement have a defining statement? or are the variables a and b in the assignment a:=b be of the same type? The former property is called *identification of the identifier*, and the latter is called *type checking*. The task of *semantic analysis* (also called *static semantics*) is to investigate and to inspect static program properties like the above.

One technique used is to decorate the nodes of the abstract syntax tree with additional attributes. During the analysis, the attribute values are evaluated by means of previously evaluated attribute values. The semantic analysis can be implemented by using either symbol tables or *attribute grammars*. The second technique frequently used is based on the context-freegrammar definition of the previous phase and defines additionally functional dependences between the attributes of the grammar rules. To each grammar rule an equation is associated showing how to evaluate the attribute value of the left side from attribute values on the right side. A formal exposition is laborious; for details we refer to (3,10).

Intermediate-Code Generation. The attributed abstract syntax tree can be used to generate machine executable code. But first a so-called *intermediate code* will be generated that is more abstract than machine code and independent of the underlying real machine. The advantage of using intermediate code is that machine-independent optimization can be applied to optimize the code, and portability to other hardware platforms can be gained.

One possible form of intermediate code is the *three-address code*, where operations with three arguments (addresses, registers) are definable, for example,

- x:=op(a,b)

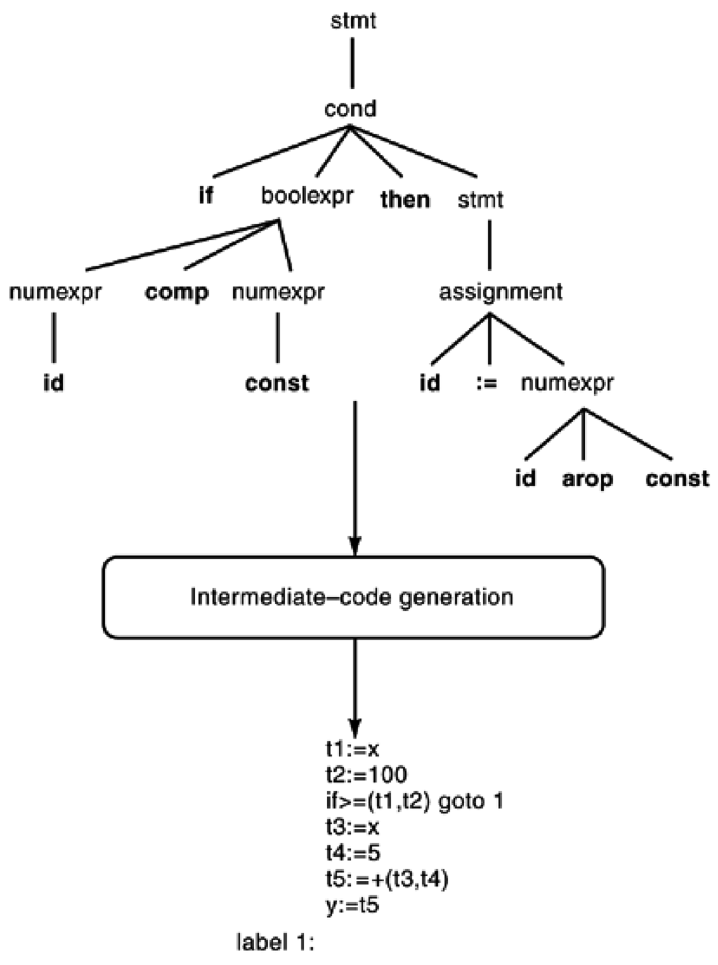

Fig. 8. Intermediate-code generation: Translating an abstract syntax tree into machine-independent code (three-address code).

- x:=a
- if comp(a,b) goto L

with x,a,b,L are addresses in the store, op is an (arithmetical) operation $\{+,-,*,\ldots\}$, comp is a compare operation $\{<,>,=,\ldots\}$, and L is a jump address. The three-address code applied to our previous example is illustrated in Fig. 8.

Machine-Independent Code Optimization. Examining Fig. 8 in detail, we find that the three-address code sequence has room for improvement. Since t1, t2, t3 are integer values and not compound expressions, their occurrences in the third and fifth statements can be replaced with their values (see Fig. 9). Code optimizations denote program transformation to improve the storage or run-time efficiency of programs. By means of *data-flow analysis or abstract interpretation* (3,11,12), program properties like the following can be computed and used to transform programs into semantically equivalent ones:

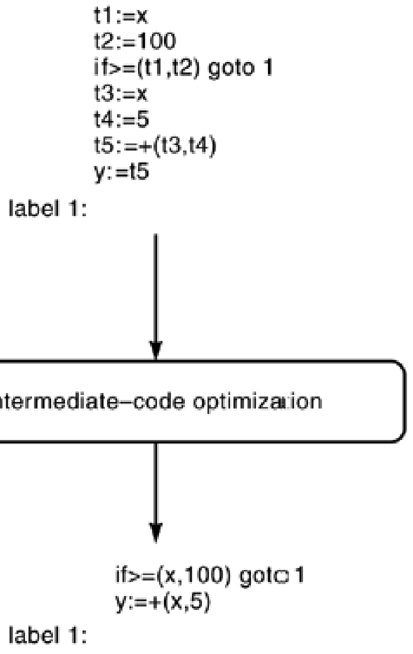

Fig. 9. Intermediate-code optimization: Refining the three-address code sequence by optimizations.

- Elimination of redundant evaluations. For example, in

  a:=1; b:=2; a:=1; c:=a+b;

  the second a:=1; is superfluous.
- Elimination of dead code. For example, in

  a:=1; if a=1 then b else c;

  the statement c will never be computed and is therefore superfluous.
- Moving loop invariants from the loop body to outside the loop, implying that the invariant is only evaluated once.

A lot of program transformation are well-known (3,11,12) but all are of heuristic nature. Optimization (i.e., the best possible code) cannot be obtained and formally proven, that is, it is not possible to prove that the generated code is optimal.

Generation of Machine-Dependent Code. The code generation of the last phase does not generate real-machine executable code. Now two alternatives are provided:

- A *mapping* from the machinelike code to a code sequence of a certain (real) machine
- A so-called *abstract* (or *virtual*) *machine* implemented on a concrete machine that interprets the machinelike code

Code Mapping. The machinelike code can again be improved by machine dependent optimizations (3):

- A real machine has a number of registers dependent on a concrete processor, enabling very fast access. Since only a restricted number of registers are available, skillful *register allocation* can enormously shorten the overall runtime.
- Each real machine offers a set of instructions. The quality of the code mapping has much to do with a good *selection* of the best (fastest) instruction sequence. The selection depends strongly on the concrete processor architecture.
- If the real machine enables *parallel processing* at instruction level, then the mapping generates certain instructions that can runin parallel. Additionally the mapping must guarantee the correctness of the parallel instructions.

Today, programs written in high-level languages frequently use additional program libraries provided as precompiled units (or machine executable code). Hence the generation of real-machine executable code can still include *linker* and *loader* processes. The address management of the given program and of the used parts of libraries must be linked together, and the occurring relative addresses must be translated into absolute addresses. Finally the code has to be loaded into the main memory for execution under the control of the operating system.

Abstract Machine. Another programming-language implementation technique is to translate a program into intermediate code, which will be considered as executable code of a certain machine that is more abstract than a concrete machine. That machine can be written in another programming language (e.g. in C or C++) or in an *assembler language*, and is called an *abstract machine*. The instruction and operation set of an abstract machine is defined independently of the underlying processor. Usually the abstract machine model is based on the model of *stack machines* equipped with a stack acting as a store of arbitrary length. Two operations on the stack are allowed. First, a new word can be pushed on top of the store, whereby the top element will be deleted. Second, just the top element of a nonempty stack can be erased. In contrast to concrete machines, no differentiation is made between the various store variants (such as register, main memory, cache, background store).

A compiled program executed by an abstract machine running on hardware platform A can also be executed on another hardware platform B provided an implementation of the abstract machine on B exists; see Fig. 10.

The advantage of using abstract machines is that they are easier to implement and to improve on a concrete machine than when one must modify the back end of a compiler. The abstract-machine technique was used in the UCSD P-System (13), one of the first commercial Pascal implementations. Nowadays the Java portability concept (14) is also based on abstract machines. The *Java compiler* generates *bytecodes* (i.e., intermediate code), which are executed on the *Java virtual machine* (i.e., an abstract machine) and can be transmitted over the Internet. For each well-known hardware platform an appropriate implementation of the Java virtual machine exists.

Interpreter

Having described the structure of a compiler, we briefly consider an alternative realization of program processing. The separation of program translation and program execution can be abolished, so that both occur simultaneously. That means the statements and expressions of a program will be evaluated (interpreted) as they are parsed. The drawback of that procedure is that code optimization is nearly impossible. Another drawback is inefficiency, because the source program must be parsed whenever it is executed. On the other hand, the target code generated by a compiler need not always be parsed at execution time. An advantage of using interpreters is their support of rapid prototyping in that an *interpreter* is easier to implement than a corresponding compiler.

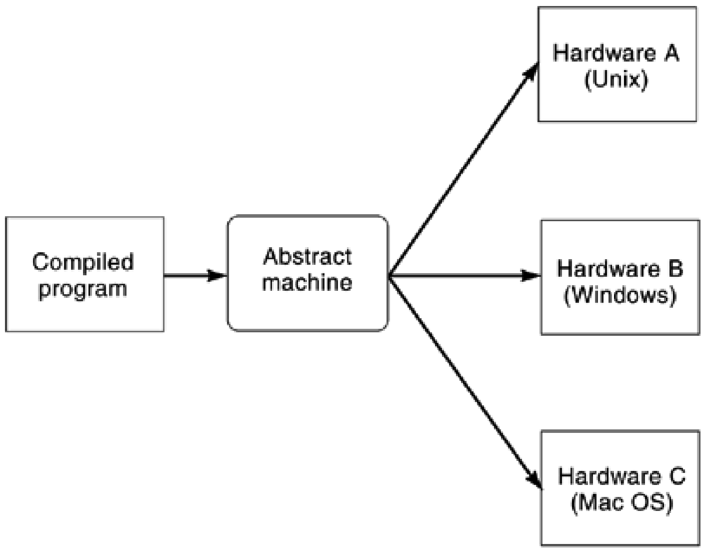

Fig. 10. A compiled program is executed by an abstract machine, which can be implemented on various platforms.

A typical interpreted programming language is the functional language Lisp or the logic language Prolog. To speed up the execution of Lisp and Prolog programs there exist compilers to generate more efficient target code, which again will be interpreted by the Lisp or Prolog interpreter. Abstract machines can also be considered as low-level machine-language interpreters.

Front End and Back End

The entire compiler structure can be divided into two components: *analysis* or *front end*, and *synthesis* or *back end*. The analysis part of a compiler consists of lexical analysis, syntactic analysis, semantic analysis, intermediate-code generation, and optimization. The synthesis part includes the machine-dependent code generation and optimization. The obvious advantage of this classification is that if a language L has compilers for k platforms, then only one front end and k back ends are needed. And vice versa, for one back end a set of various programming languages may exist, all translated into the same intermediate-representation form. Altogether, for m programming languages and k target languages, only m front ends and k back ends are necessary instead of $m * k$ different compilers (see Fig. 11).

Bootstrapping

Another way to implement a compiler is to implement first an unoptimized prototype version. Then, the prototype compiler, frequently implemented as an interpreter, can be used to translate a compiler written in the programming language itself into a compiler written in machine language. This process is called *bootstrapping*. We now explain the procedure in more detail. Let S be the source language, T be the target language, and I be the implementation language of the compiler, depicted as a so-called T diagram as in Fig. 12.

In a first step the compiler from S to T is written in the programming language S itself (see Fig. 13). To get an real implementation it is necessary to write “by hand” a compiler implementation in an appropriate

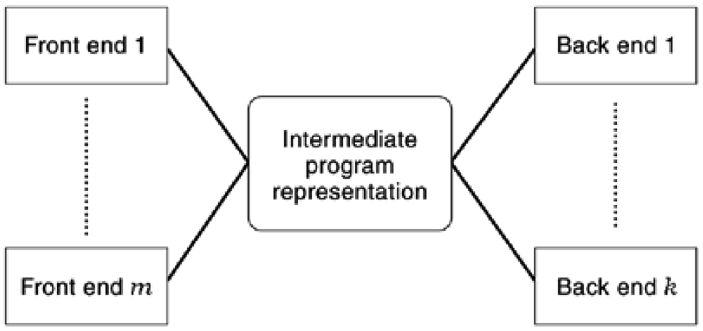

Fig. 11. Front end and back end.

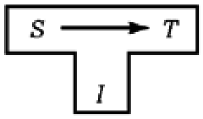

Fig. 12. T diagram: A compiler translating source programs written in language S into target programs written in language T, where the compiler is written in language I.

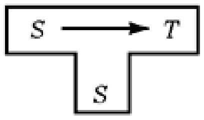

Fig. 13. Bootstrapping, part 1: A compiler from S to T written in S.

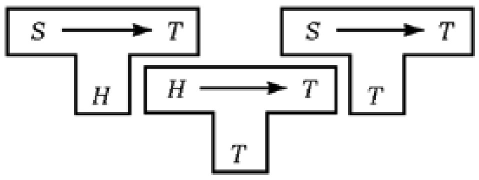

Fig. 14. Bootstrapping, part 2: Writing "by hand" a compiler in an appropriate language H.

language H, such as the programming language C (see Fig. 14). On most platforms a C compiler exists, which translates C into machine code. Then we obtain a compiler implemented in the target language T. But of course the generated implementation is highly inefficient and is unoptimized, because the "by hand" implementation is a rapid prototype implementation. Fortunately, the inefficiency of the first bootstrapping step does not propagate. Suppose the language S is extended by some new features or constructs. Let S' be the extended language based on S. Then a new compiler version can be generated in the following steps (see Fig. 15):

(1) Implementing the new features in S
(2) Using the new language features to reformulate (improve) the compiler implementation in S′
(3) Showing the correctness of the generated compiler by proving that the generated compiler translates the original one into the generated one (i.e. itself).

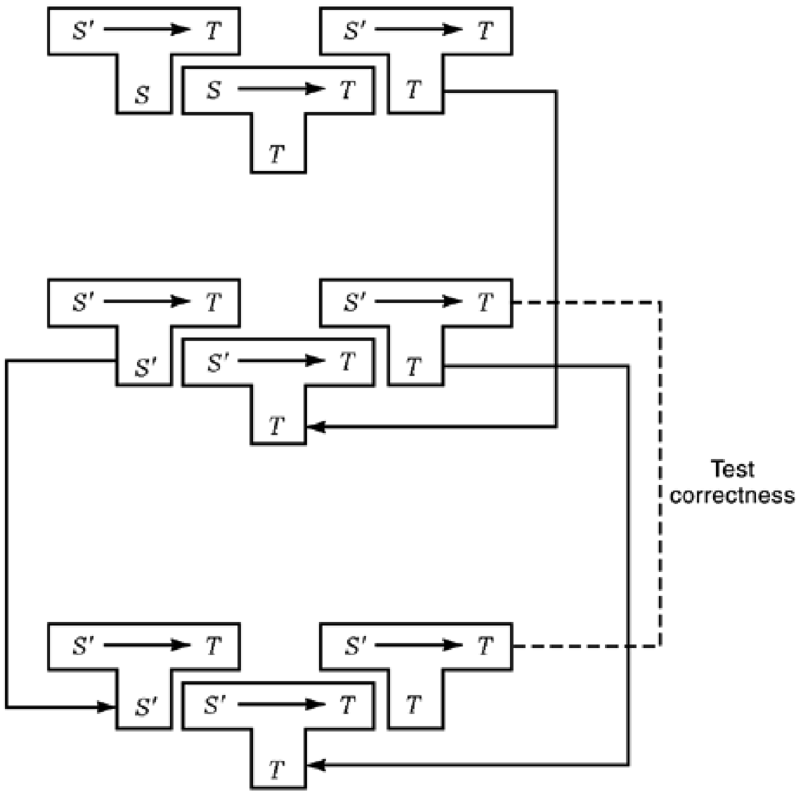

Fig. 15. Bootstrapping, part 3: Improving the compiler implementation, for example, by extending the features of language *S* or by optimizing the compilation process.

The bootstrapping process can also be applied to improve implementations of the compiler, for example by using a more sophisticated compilation algorithm.

Compiler Tools

Since compiler construction is a mature discipline, there exist tools for each compiler phase:

- Scanner generator
- Parser generators for LALR(1) or LL(1) grammars
- Abstract syntax tree generator
- Attribute grammar generator
- Code generator

The most famous compiler tools are Lex and Yacc (9), both originally implemented for the operating system Unix. Lex is a scanner generator that evaluates a corresponding scanner to a specification based on regular expressions. Yacc (Yet Another Compiler Compiler) is a powerful parser generator for LALR(1) grammars. Lex and Yacc work together: see Fig. 16. For corresponding Java tools (Jlex, CUP) we refer to Ref. (15).

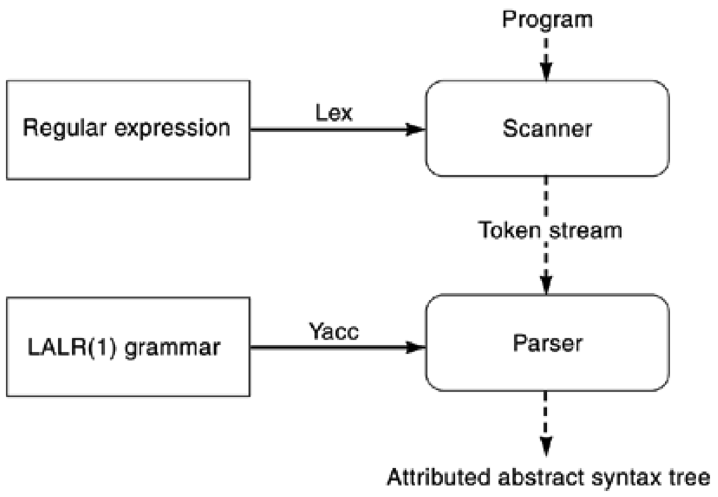

Fig. 16. Compiler tools: Applying Lex and Yacc.

Parallelizing Compilers

To decompose a program into parallel-executable components is a great challenge, since the demands of the resources (run time, store requirements, communication costs, etc.) are hard to approximate. Based on data dependency analyses, it is sometimes possible to evaluate separable code sequences. The High Performance Fortran system (12) is a programming language based on Fortran extended by some parallel language constructs and includes an appropriate compiler.

Implementing Imperative, Functional, Logical, or Object-Oriented Languages

The implementation of the different programming paradigms (imperative, functional, logical, and object-oriented) requires compiler techniques that vary in detail. Most of the differences concern the optimization phase and the construction of the back end, since the optimizations and the design of an abstract machine are very specific to the underlying programming paradigm. For reasons of space we refer to Refs. (3,4,16,17) where many further references can be found.

Just-In-Time Compilation

In conclusion, we take a look at a compilation technique used in Java (14) to speed up the run time. A Java program is translated by the Java compiler into bytecodes that are intermediate machine code and platform-independent. The Java virtual machine interprets the bytecodes. Bytecodes can be sent to and run on any hardware platform on which an implementation of the Java virtual machine exists. During the execution of a program, profiling of method calls has shown that only a few methods are frequently called. A compilation of these methods into machine (native) code can speed up the program run time. Hence a second compiler, called a *just-in-time* (*JIT*) compiler, processes this task. The JIT compiler is an integral part of the Java virtual machine and therefore invisible to the user. The JIT compilation process takes place in parallel with the execution (interpretation) of the bytecodes, and from then on, whenever a call of the JIT compiled method occurs, the machine code version will be executed. The JIT code does not always run faster than the interpreted code, however. If the Java virtual machine does not spend its time interpreting bytecode, the JIT compilation is superfluous. But in most cases these techniques also called *on-the-fly* or *on-demand* compilation), if applied to the methods most frequently called, are very helpful.

BIBLIOGRAPHY

1. C. Ghezzi M. Jazayeri *Programming Language Concepts*, New York: Wiley, 1997.
2. J. E. Hopcroft J. D. Ullman *Intoduction to Automata Theory, Languages and Computation*, Reading, MA: Addison-Wesley, 1979.
3. R. Wilhelm D. Maurer *Compiler Design*, Reading, MA: Addison-Wesley, 1995.
4. A. V. Aho R. Sethi J. D. Ullmann *Principles of Compiler Design*, Reading, MA: Addison-Wesley, 1986.
5. A. V. Aho J. D. Ullmann *The Theory of Parsing Translation and Compiling*, Vol. **1**: Parsing, Upper Saddle River, NJ: Prentice-Hall 1972.
6. A. V. Aho J. D. Ullmann *The Theory of Parsing Translation and Compiling*, Vol. **2**: Compiling, Upper Saddle River, NJ: Prentice-Hall 1973.
7. S. Sippu E. Soisalon-Soininen *Parsing Theory*, Vol. **1**: Languages and Parsing, Monographs in Theoretical Computer Science (EATCS Series), Vol. **15**, Springer-Verlag, 1988.
8. S. Sippu E. Soisalon-Soininen *Parsing Theory*, Vol. **2**: LR(k) and LL(k) Parsing, Monographs in Theoretical Computer Science (EATCS Series), Vol. **20**, Springer-Verlag, 1990.
9. J. R. Levine T. Mason D. Brown *lex & yacc, 2nd ed.*, Sebastopol, CA: O'Reilly & Associates, 1992.
10. P. Deransart M. Jourdan B. Lorho *Attribute Grammars—Definitions, Systems, and Bibliography*, Lecture Notes of Computer Science 323, New York: Springer-Verlag, 1988.
11. S. S. Muchnick N. D. Jones *Program Flow Analysis, Theory and Applications, Prentice-Hall*, Upper Saddle River, NJ: 1981.
12. H. Zima B. Chapman *Supercompilers for Parallel and Vector Computers*, ACM Press Frontier Series, Reading, MA: Addison-Wesley, 1990.
13. N. Wirth Recollections about the development of Pascal, in T. J. Bergin and R. G. Gibson (eds.); *History of Programming Languages—II*, New York, ACM Press, 1996.
14. K. Arnold J. Gosling *The Java Programming Language*, Java Series, Reading, MA: Addison-Wesley, 1997.
15. A. W. Appel *Modern Compiler Implementation in Java*, Cambridge, UK: Cambridge University Press, 1998.
16. S. L. Peyton-Jones *The Implementation of Functional Programming Languages*, Upper Saddle River, NJ: Prentice-Hall, 1987.
17. H. Ait-Kaci *Warren's Abstract Machine—A Tutorial Reconstruction*, Cambridge, MA: MIT Press, 1991.

WOLFGANG GOLUBSKI
University of Siegen