Attribute Grammar Paradigms — A High-Level Methodology in Language Implementation

JUKKA PAAKKI

Department of Computer Science and Information Systems, University of Jyväskylä, P.O. Box 35, SF-40351 Jyväskylä, Finland

Attribute grammars are a formalism for specifying programming languages. They have been applied to a great number of systems automatically producing language implementations from their specifications. The systems and their specification languages can be evaluated and classified according to their level of application support, linguistic characteristics, and degree of automation.

A survey of attribute grammar-based specification languages is given. The modern advanced specification languages extend the core attribute grammar model with concepts and primitives from established programming paradigms. The main ideas behind the developed attribute grammar paradigms are discussed, and representative specification languages are presented with a common example grammar. The presentation is founded on mapping elements of attribute grammars to their counterparts in programming languages. This methodology of integrating two problem-solving disciplines together is explored with a classification of the paradigms into structured, modular, object-oriented, logic, and functional attribute grammars. The taxonomy is complemented by introducing approaches based on an implicit parallel or incremental attribute evaluation paradigm.

Categories and Subject Descriptors: D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming; D.1.3 [**Programming Techniques**]: Concurrent Programming; D.1.4 [**Programming Techniques**]: Sequential Programming; D.1 5 [**Programming Techniques**]: Object-Oriented Programming; D.1.6 [**Programming Techniques**]: Logic Programming; D.2.2 [**Software Engineering**]: Tools and Techniques—*modules and interfaces*; *programmer workbench*; *structured programming*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features—*abstract data types*; *concurrent programming structures*; *modules*, *packages*; *procedures*, *functions*, *subroutines*, *and recursion*; D.3.4 [**Programming Languages**]: Processors—*compilers*; *interpreters*; *translator writing systems and compiler generators*; F.4.2 [**Mathematical Logic and Formal Languages**]: Grammars and Other Rewriting Systems; K.2 [**History of Computing**]: Software

General Terms: Design, Languages, Theory

Additional Key Words and Phrases: Attribute grammars, blocks, classes, compiler writing systems, functional dependencies, incomplete data, incrementality, inheritance, language processing, language processor generators, lazy evaluation, logical variables, objects, parallelism, processes, programming paradigms, semantic functions, symbol tables, unification

---

CONTENTS

INTRODUCTION

The evolutionary development of the computing discipline has created methodologies and systems to solve problems effectively in many different special areas. The value of such application-specific methods has proved to be greatest in areas that are mature enough for being systematically engineered.

Often the methods designed for a certain application area are useful in other areas (after some adaptation), or different techniques can be combined within the same area. To integrate different techniques into a seamless whole, it is necessary to foresee the effects of their combination. Hence, full understanding of the concepts and mechanisms of the components is the prerequisite for a sensible integration. One example of this approach is the current interest in multiparadigm programming languages. The foundation for designing and implementing such languages is provided by well-established single programming paradigms; for an overview, see ACM Computing Surveys [1989], Sethi [1989], and Watt [1990].

Compiler construction is often mentioned as one of the few really systematically managed disciplines, e.g., Shaw [1990]. This stems from the relatively long tradition of producig compilers, from practical underlying theories, and from a large selection of textbooks on the topic, such as Aho et al. [1986] and Fischer and LeBlanc [1988]. Today, producing compilers (or at least their analysis phase, the "front-end") is routine work with automatic tools.

Context-free grammars are the standard tool used in reference manuals for defining the syntax of programming languages. The accompanying lexical definition of the language is usually given using *regular expressions*, another popular specification formalism. The practical significance of these formalisms is greatly enhanced by the fact that they can be automatically implemented as efficient language processors, context-free grammars as parsers, and regular expressions as lexical analyzers.

Attribute grammars were defined in the late 1960's by Knuth for specifying and implementing the (static) semantic aspects of programming languages [Knuth 1968]. Since then, attribute grammars have been a subject of intensive research, both from a conceptual and from a practical point of view. The conceptual work has produced several subclasses of attribute grammars with advanced implementation algorithms. The closely coupled pragmatic efforts have created a large number of automated systems based on attribute grammars. These systems, usually called *compiler-compilers*, *compiler writing systems*, or *translator writing systems*, generate different kinds of language processors from their high-level specifications. In their early years the systems were used to generate compilers only, but later generators of interpreters, debuggers, editors, etc. have shown that the attribute grammar model provides a basis for a number of alternative language processing schemes as well.

Research on attribute grammars has refined their somewhat primitive original notion into a mature formalism. The maturity can be seen from fairly efficient implementation algorithms of attribute grammars, and also from the large num-

ber of systems produced. Each system has its own specification language, in a sense a special dialect of attribute grammars. Like programming languages, these dialects can be classified according to their fundamental ideas and primitives. In this survey we systematically investigate conceptual attribute grammar classes that can be characterized with the term *attribute grammar paradigm*, giving rise to a taxonomy of language specification styles all based on the attribute grammar model.

Extensive reviews of attribute grammar theory, implementation, systems, and applications are given in, e.g., Deransart et al. [1988], Deransart and Jourdan [1990], and Alblas and Melichar [1991]. This survey does not thoroughly discuss such issues, nor the methodology of applying attribute grammars in the specification and implementation of programming languages. Rather, we focus on the *paradigms*, motivating and presenting the main ideas behind them. The conceptual presentation is complemented with examples of the different paradigms in terms of some specific systems that support these paradigms in their specification language. We hope that the examples also demonstrate the attribute grammar methodology for an uninitiated reader.

An attribute grammar-based specification language is also a language itself, a special-purpose programming language in the application area of language processing. Therefore, an attribute grammar paradigm must involve issues both from the attribute grammar model and from the programming language discipline. Thus, an attribute grammar paradigm is a good example of integrating two different problem-solving methodologies, specification and programming, into a coherent combination. We believe that this explicit case of integration also shows the power of establishing new practical and rigorous methods by assembling two existing mature disciplines together.

We start in Section 1 by presenting attribute grammars in their standard form. The merits and drawbacks of the basic concept are discussed; the drawbacks motivate paradigmatic improvements presented in the subsequent sections.

In our taxonomy, attribute grammar styles are classified into paradigms according to the particular linguistic framework they promote. These frameworks either aid in structuring an attribute grammar into integrated units or improve the semantic expressiveness of attribute grammars by employing some powerful attribute evaluation machinery. Respectively, the survey presents *organizational paradigms* in Section 2 and *evaluation paradigms* in Section 3. The class of organizational paradigms is further refined into *attribution paradigms*, *structured attribute grammars*, *modular attribute grammars*, and *object-oriented attribute grammars* in Sections 2.1 to 2.4, respectively. The evaluation paradigm is elaborated by presenting *logic attribute grammars*, *functional attribute grammars*, and *implicit evaluation* paradigms in Sections 3.1 to 3.3, respectively. We conclude in Section 4 by summarizing and relating the merits of the different paradigms, and by envisaging future directions.

The contributions of the different paradigms vary according to their main emphasis and characteristics. In essence, the organizational paradigms raise the software engineering level of the metalanguages based on attribute grammars. The evaluation paradigms, on the other hand, stress the semantic expressiveness of attribute grammars. Finally, efficiency of the generated language processors is the main concern of the special implicit paradigms.

1. ATTRIBUTE GRAMMARS

Here we present the basic characteristics of attribute grammars. Some historical milestones and central application areas are mentioned in Section 1.1. Section 1.2 gives the definition of attribute grammars. Section 1.3 presents the fundamental concept of characteristic (dependency) graphs that lay the foundation for both analyzing and implementing attribute grammars. An example grammar is given

as well to illustrate the formalism and to serve as a common basis for presenting the paradigmatic variations of the standard model.

1.1 Historical Remarks

Designing a programming language is an intellectual challenge of considerable complexity (for an overview, see Wasserman [1980]). One of the most important considerations is the description of the language. The description must be illustrative enough for a language user, and precise enough for a language implementor. Several language definition formalisms have been developed. The most successful has been the Backus-Naur Form, sometimes also referred to as the Backus-Normal Form (BNF).

BNF was originally presented for defining the programming language Algol 60 [Naur 1960] and is based on the notion of *context-free grammars*, which have been generally accepted as the standard formalism for defining the syntax of programming languages. *Regular expressions* are another standard formalism, used for specifying the lexical structure of programming languages. Context-free grammars and regular expressions were presented already in the 1950's in connection with the seminal linguistic work of Chomsky [1959].

The scope of regular expressions and context-free grammars soon extended from theoretical language definition into practical compiler generation. Subclasses of context-free grammars were defined for automatic generation of efficient syntax analyzers from syntax definitions, the most notable examples being *LR grammars* [Knuth 1965] and *LL grammars* [Lewis and Stearns 1968]. Additionally, mapping regular expressions into finite automata made it feasible to generate lexical analyzers automatically.

While a consensus on the definition and generation formalisms has been reached with respect to the first two phases of a typical language implementation (lexical analysis, syntactic analysis), the subject is more controversial when considering the remaining phases (semantic analysis, code generation, interpretation). Hence, a variety of different formalisms has been proposed for the definition of the semantics of programming languages and for the automatic generation of complete language implementations from such definitions.

The most formal methods have a mathematical foundation providing concepts for writing unambiguous and consistent definitions. *Denotational semantics* and *axiomatic semantics* are two well-known examples of this kind. The former is based on the use of functions mapping the constructs of a language directly to their mathematical meanings, while the latter employs logical axioms and deduction rules for specifying the language. Techniques labeled commonly as *operational semantics* define a language by mapping its constructs into instructions of a simple abstract interpreter whose semantics is well defined. An introductory survey of language definition formalisms is given in, e.g., Watt [1991].

Attribute grammars are probably the most widely applied semantic formalism. This method is founded upon the idea of a "syntax-directed translation scheme" where a context-free grammar is augmented with *attributes* and *semantic rules*. Because attribute grammars are a proper extension of context-free grammars, an efficient implementation is easier to obtain from a language definition given as an attribute grammar than one from one of the more mathematically biased formalisms mentioned above. On the other hand, attribute grammars are less formal and less expressive than the mathematical models because the functions applied in the semantic rules are not precisely defined.

Attribute grammars have gradually developed from a theoretical concept into an advanced computational paradigm. The definition of the formalism was given in Knuth [1968]; the evolution of ideas behind the formalism is summarized in Knuth [1990]. The first decade of research focused on analyzing the character of attribute grammars and on establishing basic implementation algorithms. The first full implementation was

produced by Fang [1972] under the supervision of Knuth. Attribute grammars were soon found to be an attractive tool for modeling passwise compilation strategies. Seminal results in this direction include four attribute grammar subclasses: *S-attributed grammars* and *L-attributed grammars* for one-pass compilation [Lewis et al. 1974], and *absolutely noncircular attribute grammars* [Kennedy and Warren 1976] and *ordered attribute grammars* [Kastens 1980] for multipass compilation.

Several systems were developed with high-level input languages, resulting in powerful language processing tools. Example pioneer systems in this direction are GAG [Kastens et al. 1982], LINGUIST-86 [Farrow 1982], and HLP84 [Koskimies et al. 1988]. In recent years research has shifted from theoretical investigations into improving attribute grammars as a methodology, and into designing systems with major emphasis on pragmatic issues. This tendency is also reflected in this survey, which aims at presenting the main directions of developing attribute grammars into a true engineering discipline.

While implementation of (textual) programming languages is the original and most widely studied area of attribute grammars, they can also be used in many other fields where relations among structured information play a central role. This versatility is demonstrated by an increasing number of new application areas where attribute grammars have been promoted to having significant value. Recent notable examples include general software engineering [Shinoda and Katayama 1988; Frost 1992; Lewi et al. 1992], reactive systems [Ding and Katayama 1993], distributed programming [Kaiser and Kaplan 1993], logic programming [Deransart and Maluszynski 1985; 1993], static analysis of programs [Horwitz and Reps 1992], databases [Ridjanovic and Brodie 1982], natural language interfaces [Alexin et al. 1990], graphical user interfaces and visual programming [Hudson and King 1987; Crimi et al. 1990], pattern recognition [Tsai and Fu 1980; Trahanias and Skordalakis 1990], hardware design [Jones and Simon 1986], computer communication protocols [van de Burgt and Tilanus 1989; Chapman 1990], and combinatorics [Delest and Fedou 1992].

1.2 Definition and Basic Notations [Knuth 1968]

An *attribute grammar* (AG) consists of three components, a *context-free grammar* G, a finite set of *attributes* A, and a finite set of *semantic rules* R: $AG = \langle G, A, R \rangle$. Informally, G specifies the syntax of the target language, and A and R specify the (static) semantics of the language.

A context-free grammar G is a quadruple: $G = \langle N, T, P, D \rangle$, where N is a finite set of *nonterminal symbols*; T is a finite set of *terminal symbols*; P is a finite set of *productions*; and $D \in N$ is the designated *start* symbol of G. An element in $V = N \cup T$ is called a *grammar symbol*. The productions in P are pairs of the form $X \rightarrow \alpha$, where $X \in N$ and $\alpha \in V^*$, i.e., the left-hand side X is a nonterminal, and the right-hand side α is a string of grammar symbols. An empty right-hand side (empty string) is represented by the symbol ε.

A finite set of attributes $A(X)$ is associated with each symbol $X \in V$. The set $A(X)$ is partitioned into two disjoint subsets, the *inherited attributes* $I(X)$ and the *synthesized attributes* $S(X)$. The start symbol and the terminal symbols do not have inherited attributes.[1] Now $A = \cup A(X)$.

[1] The original definition in Knuth [1968] accepts terminal symbols to have inherited attributes but no synthesized attributes. While this is in line with the general discipline of associating attributes with grammar symbols, the rule has been reversed in most implementations. The reason for accepting just synthesized attributes for terminal symbols is modularization of language processors: now the terminal symbols become context-independent, and the lexical analyzers can be generated independently of the other components. As this survey uses existing systems to demonstrate the different paradigms, we also have adopted the nonstandard convention in our definition.

A production $p \in P$, $p: X_0 \rightarrow X_1 \cdots X_n$ ($n \geq 0$), has an *attribute occurrence* $X_i.a$, if $a \in A(X_i)$, $0 \leq i \leq n$. A finite set of *semantic rules* R_p is associated with the production p with exactly one rule for each synthesized attribute occurrence $X_0.a$ and exactly one rule for each inherited attribute occurrence $X_i.a$, $1 \leq i \leq n$. Thus R_p is a collection of rules of the form $X_i.a = f(y_1, \ldots, y_k)$, $k \geq 0$, where

(1) either $i = 0$ and $a \in S(X_i)$, or $1 \leq i \leq n$ and $a \in I(X_i)$;
(2) each y_j, $1 \leq j \leq k$, is an attribute occurrence in p; and
(3) f is a function, called a *semantic function*, that maps the values of $y_1, \ldots, y_k$ to the value of $X_i.a$. In a rule $X_i.a = f(y_1, \ldots, y_k)$, the occurrence $X_i.a$ *depends* on each occurrence y_j, $1 \leq j \leq k$. Now $R = \cup R_p$.

By the definition, synthesized attributes are output to the left-hand-side symbols of productions, and inherited attributes are output to the right-hand-side symbols. (The synthesized attributes of terminal symbols are assumed to be externally defined.)

A *derivation tree* (*parse tree*) for a program $\mathscr{P}$, generated by $G = \langle N, T, P, D \rangle$, is a tree, where

(1) each node is labeled by a symbol $X \in V$ or by ε;
(2) the label of the root is D;
(3) if a node labeled by X has the sons labeled by $X_1, \ldots, X_n$, then $X \rightarrow X_1 \ldots X_n$ is a production in P; and
(4) the labels of the leaves of the tree, concatenated from left to right, form $\mathscr{P}$.

An *attributed tree* for a program $\mathscr{P}$ is a derivation tree for $\mathscr{P}$ where each node n, labeled by X, is attached with *attribute instances* that correspond to the attributes of X. For each attribute $a \in A(X)$ the corresponding instance is denoted with $n.a$.

Attribute evaluation is a process that computes values of attribute instances within an attributed tree T according to the semantic rules R. That is, the value of an attribute instance $n.a$., corresponding to attribute a of symbol Y, is computed by executing the semantic rule $Y.a = f(y_1, \ldots, y_k) \in R_p$ where either (1) $a \in I(Y)$ and p is the production $X \rightarrow \ldots Y \ldots$ applied when generating n into T, or (2) $a \in S(Y)$ and p is the production $Y \rightarrow \ldots$ applied when generating n into T. The semantic rule is executed by calling the function f with the values of those attribute instances as arguments that correspond to $y_1, \ldots, y_k$, and by assigning the returned value to $n.a$. As for attribute occurrences, the instance $n.a$. is defined to *depend* on all the instances for $y_1, \ldots, y_k$. The *meaning* of a program $\mathscr{P}$ consists of the values of the (synthesized) attribute instances associated with the root node of the attributed tree for $\mathscr{P}$.

Prior to the definition of attribute grammars, synthesized information and purely syntax-directed language processing were considered as the "proper" form of programming language semantics [Irons 1961; Steel 1966]. From a purely theoretical point of view, inherited attributes are unnecessary [Knuth 1968]. They are, however, useful in many practical situations, e.g., in representing symbol tables whose value is typically applied outside of the creating context. Inherited attributes also introduce an attractive conceptual balance into the compilation process, and that is why they are widely employed in systems that generate multipass evaluators over the attributed tree.

An attribute grammar $AG = \langle G, A, R \rangle$ is *well defined* if the semantic rules R are such that for each attributed tree T for a program $\mathscr{P}$ (generated by G), the values of the attribute instances within T can be unambiguously computed by an attribute evaluation process. In Knuth [1968] this criterion is stated such that the *attribute dependency graph* for T, induced by R, is acyclic.[2] In other words,

[2] An attribute dependency graph for an attributed tree T has the attribute instances within T as its vertices. The graph contains a directed arc from $n.b$ to $n.a$ if and only if the attribute instance $n.a$ depends on the attribute instance $n.b$.

attribute instances must have a partial evaluation ordering, and no attribute instance may (transitively) depend on itself.

This requirement guarantees that it is always possible to find an execution order for semantic functions such that the value of each attribute instance is computed exactly once, and each function applies previously evaluated arguments only. In other words, the standard attribute evaluation model is *strict*. While this strategy based on noncircular attribute dependencies is the traditional method, it is not the only possible one. In Section 3 we will discuss mechanisms that are based on the intuition that the main objective of an attribute evaluation process is not to compute the value of *all* the attribute instances within T, but instead just the *meaning* of the corresponding program $\mathscr{E}$. This requirement is weaker than that given above and makes it possible to have a broader interpretation of well-defined attribute grammars. Another relaxation of the original meaning of being well defined, also addressed in Section 3, is based on the use of nonstrict functions and incomplete values in attribute evaluation.

1.3 Characteristic Graphs

The general noncircular property of attibute grammars requires an exponential testing algorithm [Jazayeri et al. 1975]. Also, implementing attribute grammars in their most general form is very hard. These aspects call immediately for relaxing the ultimate goal of applying attribute grammars in their full power and thus make it necessary to discover restricted yet powerful subclasses of the formalism.

In this section we introduce the concept of *characteristic (dependency) graphs* that make it possible both to define more practical attribute grammar classes and to design effective implementation strategies for them. The basic idea behind characteristic graphs is to assemble the attribute dependency information statically into a single representation that holds for *all* the possible attributed trees for the grammar. A universal dependency description makes it possible to analyze the characteristic properties of an attribute grammar and the corresponding language processor without considering the concrete source programs. We will only give an introduction to the topic; for a more detailed presentation, refer to Deransart et al. [1988] and Alblas [1991].

To simplify the discussion, we adopt the conventional restriction that the applied attribute occurrences $y_1, \ldots, y_k$ for a semantic rule $X_i.a = f(y_1, \ldots, y_k)$ in R_p must be defined outside of the production p: $X_0 \rightarrow X_1 \cdots X_n$. That is, for each y_j of the form $X_l.b, j = 1, \ldots, k$, either $l = 0$ and $b \in I(X_l)$, or $1 \leq l \leq n$ and $b \in S(X_l)$. If this holds for all the semantic rules of the attribute grammar, it is in *normal form* [Bochmann 1976]. Unless otherwise stated, the attribute grammars in this survey are assumed to be in normal form.[3]

The main building blocks of characteristic graphs are the *local dependency graphs* DG_p. A graph DG_p summarizes the attribute dependencies associated with the production p: $X_0 \rightarrow X_1 \cdots X_n$:

- the vertices of DG_p are the attribute occurrences of p;
- for each pair of attribute occurrences $X_i.a$ and $X_j.b$ of p, there is a directed arc from $X_j.b$ to $X_i.a$ in DG_p if and only if $X_i.a$ depends on $X_j.b$.

The local dependency graphs already lay the foundation of two important subclasses of attribute grammars:

(1) An attribute grammar is *S-attributed* if it has only synthesized attributes (hence for each arc from $X_j.b$ to $X_i.a$ in any DG_p the following conditions hold: $i = 0$ and $1 \leq j \leq n$) [Lewis et al. 1974].

[3]This is not a severe restriction since every (well-defined) attribute grammar can be put in normal form by a simple transformation

(2) An attribute grammar is *L-attributed* if for each arc from $X_j.b$ to $X_i.a$ in any DG_p, where $1 \le i \le n$ (and thus $a \in I(X_i)$), the following condition holds: $j < i$ [Lewis et al. 1974; Bochmann 1976].

While these classes seem very primitive, they have turned out to be surprisingly practical in a large number of language processor generators. For instance YACC [Johnson 1975], probably the most well-known language implementation tool, can be regarded as an implementation of simple S-attributed grammars. A more profound motivation for S-attributed and L-attributed grammars is that they can be considered as models of one-pass language processing, e.g. one-pass compilation, where semantic information is produced in one traversal over an attributed tree and in most cases even interleaved with syntax analysis.

More complex language processing tasks must be solved with attribute grammar classes more general than S-attributed and L-attributed grammars. In essence, in such cases there must be some means to analyze *global* attribute dependencies instead of just the local ones, as above.

Consider the inherited attributes of a symbol as its input and the synthesized attributes as its output. Then the input-output behavior of a (nonterminal) symbol X can be modeled by the graph $IS(X)$ with a directed arc from an inherited attribute $b \in I(X)$ to a synthesized attribute $a \in S(X)$ if the instance for a depends transitively on the instance for b in some attributed subtree with a node labeled by X as the root.

Let us consider a production p: $X_0 \to X_1 \cdots X_n$. (Recall that there is a local dependency graph DG_p for p.) Suppose that for each $i = 1, 2, \ldots, n$ we have a directed input-output graph $IS(X_i)$. Then we denote by $DG_p^* = DG_p[IS(X_1), IS(X_2), \ldots, IS(X_n)]$ the directed graph that is obtained from DG_p by adding an arc from attribute occurrence $X_i.b$ to $X_i.a$ whenever there is an arc from b to a in $IS(X_i)$, $i = 1, 2, \ldots, n$.

The input-output graphs $IS(X)$ can be recursively defined in terms of the graphs DG_p^*:

- the vertices of $IS(X)$ are the attributes $A(X)$;
- there is an arc in $IS(X)$ from an inherited attribute $b \in I(X)$ to a synthesized attribute $a \in S(X)$ if and only if there is a path from $X.b$ to $X.a$ in the graph DG_p^* for some production p: $X \to \ldots$ whose left-hand side is X.

An attribute grammar is *absolutely noncircular* if no *augmented dependency graph* DG_p^* contains a directed cycle [Kennedy and Warren 1976]. The graphs DG_p^* and $IS(X)$ can be computed iteratively by starting from the local dependency graphs DG_p and the graphs $IS(X)$ without arcs, and by adding a nonexisting arc to an $IS(X)$ if there is a production p with left-hand side X such that the current DG_p^* contains a path from $b \in I(X)$ to $a \in S(X)$ (simultaneously extending the graph DG_q^* for each production q having X on its right-hand side), until no more arcs can be added to any $IS(X)$.

The arcs in $IS(X)$ reflect all the input-output dependencies that could exist in an attributed subtree with X as its root. The approximation is, however, overpessimistic and may reject an attribute grammar which is actually well defined. Still, absolutely noncircular attribute grammars have been accepted as the largest tractable class of attribute grammars because the decision algorithm for the class is polynomial instead of the intrinsically exponential test for general well-defined grammars.

Another powerful and widely applied multipass grammar class of polynomial complexity is ordered attribute grammars [Kastens 1980]. While being a proper subclass of absolutely noncircular attribute grammars, ordered attribute grammars are attractive because they can be implemented in a simpler and more efficient way. Another aspect is that even the ordered attribute grammars seem to include all the practical cases

when considering the processing of programming languages. This class is only sketched below; for technical details, see Kastens [1980] or Alblas [1991].

In addition to the input-output dependencies $IS(X)$, the output-input dependencies $SI(X)$ are approximated in ordered attribute grammars. That is, the graph $SI(X)$ includes an arc from a synthesized attribute $b \in S(X)$ to an inherited attribute $a \in I(X)$ if a may (transitively) depend on b in some attributed derivation tree. Notice that such information is necessarily computed in terms of the upper contexts of X-nodes in the attributed derivation trees.

From the compound graph $ISSI(X) = IS(X) \cup SI(X)$, a linear *partial order* $TO(X)$ for the attributes of each symbol X is computed. $TO(X)$ comprises all the direct and indirect attribute dependencies which may be derived from any context of X. An attribute grammar is *ordered* if and only if for each production p: $X_0 \rightarrow X_1 \ldots X_n$ the graph $DG_p[TO(X_0), TO(X_1), \ldots, TO(X_n)]$ is acyclic. A central property of ordered attribute grammars is that $ISSI(X) \subseteq TO(X)$ for each symbol X. For techniques of computing the applicable linear orders $TO(X)$ as extensions of $ISSI(X)$ and forcing a circular dependency pattern into an *orderly arranged* form with augmented attribute dependencies, see Kastens [1980].

A context-independent evaluation procedure for a symbol X arranges the attributes $A(X)$ into a linear sequence $I_1(X)$, $S_1(X)$, $I_2(X)$, $S_2(X), \ldots, I_m(X)$, $S_m(X)$. The disjoint subsets of $A(X)$, $I_j(X)$ and $S_j(X)$, form a partition of $A(X)$. For ordered attribute grammars, $TO(X)$ defines a linear *total order* over the subsets $I_j(X)$ and $S_j(X)$, $j = 1, 2, \ldots, m$, even though the evaluation order within each subset is irrelevant and thus undefined.

The evaluation algorithm for an ordered attribute grammar exhibits a strategy of evaluating the attributes of X in m visits, such that each set $I_j(X)$ includes the inherited attribute instances to be evaluated when entering a node labeled by X for the jth time, and each set $S_j(X)$ includes the synthesized attribute instances to be evaluated when exiting a node labeled by X for the jth time. Between these evaluation actions, the algorithm visits other nodes in the context and evaluates some of the attribute instances therein in the same manner.

Example 1.3.1

For demonstrating attribute grammars, let us study a simple desk calculation language, DESK. In DESK, one can write programs of the form

PRINT ⟨expression⟩ WHERE ⟨definitions⟩

where ⟨expression⟩ is an arithmetic expression over numbers and defined constants, and ⟨definitions⟩ is a sequence of constant definitions of the form

⟨constant name⟩ = ⟨number⟩

Each named constant applied in ⟨expression⟩ must be defined in ⟨definitions⟩, and ⟨definitions⟩ may not give multiple values for a constant. For instance, the following is a DESK program:

PRINT x + y + 1 WHERE x = 1, y = 2 (1.1)

We specify the DESK language with an attribute grammar. To illustrate the various paradigms presented, the DESK language (with minor variations) will be used as a case example throughout this survey. Because of repeated elaboration, the example is concise. For instance, only additions are allowed in expressions; a richer selection would just make the grammar longer without bringing in any new aspects. However, DESK still has many central characteristics of a real programming language:

- declaration of named entities (here: constants)
- use of declared entities
- conditions on the declaration and use of an entity:
  - —no constant can be multiply declared

—only declared constants can be referenced by name
- executable entities (here: expressions).

Without loss of focus of the presentation, DESK may also be considered as one part of a complete, more advanced programming language.

On the implementation side DESK introduces the following typical tasks:

- lexical and syntactic analysis
- name analysis (i.e., symbol table management)
- checking of static conditions
- multipass, or more precisely, right-to-left processing (because named entities can be used before declaration) and
- interpretation, or code generation for the executable entities.

Let us start by specifying the DESK language in the original "standard" form of an attribute grammar. Since attribute grammars are more suitable for describing compilation than interpretation, the (dynamic) meaning of a DESK program is defined implicitly as a mapping into a lower-level target code. For simplicity, we make use of an assembly code for a simple one-register machine as the target. The relevant assembly instructions are the following:

LOAD n,	loads the value n into the register;
ADD n,	increments the contents of the register with value n;
PRINT 0,	prints the contents of the register;
HALT 0,	halts the machine.

The execution of a legal DESK program evaluates the expression and prints its value, whereas an illegal program halts the machine without printing. For instance, the code generated for the DESK program (1.1) is

```
LOAD 1 (x)
ADD  2 (y)
ADD  1
PRINT 0
HALT  0
```

The context-free grammar G for DESK consists of the following elements:

```
N = {Program, Expression, Factor,
     ConstName, ConstPart,
     ConstDefList, ConstDef},

T = {'PRINT', 'WHERE', ',', '=',
     '+', Id, Number},

P = {Program →
       'PRINT' Expression ConstPart,
     Expression →
       Expression '+' Factor,
     Expression →
       Factor,
     Factor →
       ConstName,
     Factor →
       Number,
     ConstName →
       Id,
     ConstPart →
       ε,
     ConstPart →
       'WHERE' ConstDefList,
     ConstDefList →
       ConstDefList ',' ConstDef,
     ConstDefList →
       ConstDef,
     ConstDef →
       ConstName '=' Number},

D = Program.
```

We make use of the following attributes A:

code,	synthesized target code;
name,	synthesized name of a constant;
value,	synthesized value of a constant or a number;
envs,	synthesized environment (symbol table);
envi,	inherited environment (symbol table);
ok,	synthesized indicator of correctness.

The target code is a list of instructions of the form (operation code, argument). An environment is a list of bindings of the form (name, value).

The semantic rules R are associated with the productions as follows, using the notation "production {semantic rules}". Within a production, different occurrences of the same grammar symbol are denoted by distinct subscripts.

(1.2)

```
(p1)  Program → 'PRINT' Expression ConstPart
      {Program.code = if ConstPart.ok
          then Expression.code + (PRINT, 0) + (HALT, 0)
          else (HALT, 0),
      Expression.envi = ConstPart envs}

(p2)  Expression1 → Expression2 '+' Factor
      {Expression1.code = if Factor.ok
          then Expression2.code + (ADD, Factor.value)
          else (HALT, 0),
      Expression2.envi = Expression1.envi,
      Factor.envi = Expression1.envi}

(p3)  Expression → Factor
      {Expression.code = if Factor.ok then (LOAD, Factor.value)
          else (HALT, 0),
      Factor.envi = Expression.envi}

(p4)  Factor → ConstName
      {Factor.ok = isin (ConstName.name, Factor.envi),
      Factor.value = getvalue (ConstName.name, Factor.envi)}

(p5)  Factor → Number
      {Factor.ok = true, Factor value = Number value}

(p6)  ConstName → Id
      {ConstName.name = Id.name}

(p7)  ConstPart → ε
      {ConstPart.ok = true, ConstPart.envs = ()}

(p8)  ConstPart → 'WHERE' ConstDefList
      {ConstPart.ok = ConstDefList.ok,
      ConstPart envs = ConstDefList.envs}

(p9)  ConstDefList1 → ConstDefList2 ',' ConstDef
      {ConstDefList1.ok = ConstDefList2.ok and not
          isin(ConstDef name, ConstDefList2.envs),
      ConstDefList1.envs = ConstDefList2.envs +
          (ConstDef.name, ConstDef.value)}

(p10) ConstDefList → ConstDef
      {ConstDefList.ok = true,
      ConstDefList.envs = (ConstDef.name, ConstDef.value)}

(p11) ConstDef → ConstName '=' Number
      {ConstDef.name = ConstName.name,
      ConstDef.value = Number.value}
```

As usual in attribute grammars, we assume that the values of the intrinsic attributes Id.name and Number.value of the terminal symbols are externally provided, e.g., by a lexical analyzer.

We have applied the following semantic functions (some of them written in a more convenient infix notation):

if(c, v1, v2),	if c yields *true*, returns v1 else returns v2;
+(1, 2),	returns the catenation of lists 1 and 2;
and (c1, c2),	returns the conjunction of c1 and c2;
not (c),	returns the negation of c;
isin (n, e),	if the environment e contains a binding (n, x), returns *true* else returns *false*;
getvalue (n, e),	if the environment e contains a binding (n, x), returns x else returns **0**.

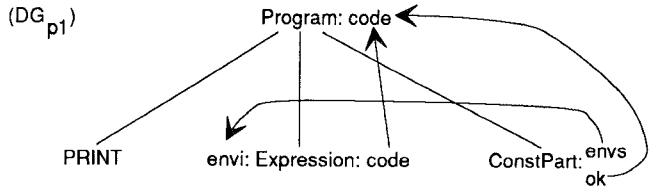


Figure 1. Local dependency graph for $p1$.

We have also applied some conventional constant functions, such as the integer 0, the Boolean *true*, and the empty list ().

For clarity, all the applications of the identity function I ($I(x) = x$ for all x) have been directly replaced by the result value. (Recall that the attribute grammar definition only permits semantic rules of the form $X.a = f(\ldots)$ where f is a function.)

The meaning of a DESK program is the value of Program.code, i.e., the target code associated with the root of the corresponding attributed derivation tree. Note that in this case a semantically incorrect DESK program also has a meaning; for instance the meaning of

```
PRINT z WHERE x = 1, y = 2
```

is the target code

```
HALT   0
PRINT  0
HALT   0
```

(*End of Example* 1.3.1)

Let us analyze the DESK attribute grammar (1.2). The grammar is not L-attributed (nor S-attributed), due to a right-to-left arc in the local dependency graph DG_{p1} for production $p1$, as shown in Figure 1. The inherited attributes are associated to the left and the synthesized attributes to the right of a grammar symbol.

The grammar is, however, absolutely noncircular. This is shown in Figure 2 by the acyclic augmented dependency graph DG_p^* for each production p that has a symbol in its right-hand side with both inherited and synthesized attributes. The transitive input-output dependencies in $IS(X)$ are denoted by dashed arrows.

The inclusion of the input-output dependency arcs *envi*—→ *ok* and *envi*—→ *value* in $IS(Factor)$ (as well as in DG_{p2}^* and DG_{p3}^*) is revealed by the local dependency graph for production $p4$, shown in Figure 3. Notice that $IS(Factor)$ implicitly introduces the arc *envi*—→ *code* into $IS(Expression)$ (and into DG_{p1}^* and DG_{p2}^*) via the graph DG_{p3}^* in Figure 2.

Since the attribute grammar is absolutely noncircular, it is also well defined.[4] Hence, the attribute dependency graph is acyclic for each attributed tree spanned by the grammar. To illustrate the relation between the characteristic graphs and the attributed trees for concrete source programs, Figure 4 sketches the attributed tree for the DESK program (1.1). For clarity, only the upper level of the tree is completely given. Semantic functions are denoted by circles.

This attribute dependency graph shows explicitly the data flow within the derivation tree. The graph shows, most notably, that in order to evaluate Program.code, the synthesized instance of ConstPart.envs, the inherited instance of Expression.envi, and the synthesized instance of Expresson.code must first be evaluated, in that order. (Recall the same information in the augmented characteristic graph DG_{p1}^* of Figure 2.) In other words, for generating the target code for the program, the symbol table must be constructed in the subtree for ConstPart and consulted in the subtree for Expression (since the values of named constants

[4]The grammar also belongs to a number of other well-defined supersets of L-attributed grammars, such as the class of ordered attribute grammars. A further elaboration is omitted.

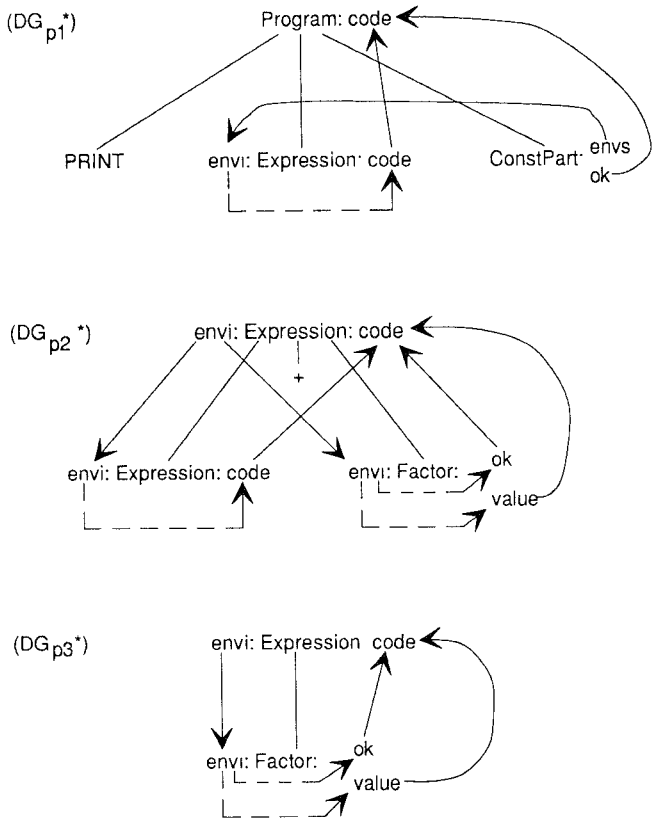


Figure 2. Augmented dependency graphs.

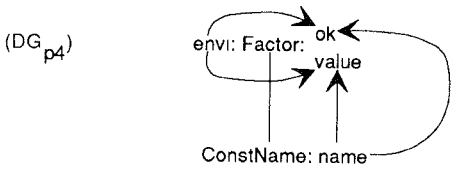


Figure 3. Local dependency graph for $p4$.

are needed for generating the code). This also applies to name analysis for the expression part of the program.

When considering the attribute grammar (1.2) as specification of a DESK compiler, we immediately notice that compilation cannot be totally done during parsing (and embedded lexical analysis). This is due to the fact that when parsing the expression part of a program (e.g., x + y + 1 in program (1.1)), the values of the named constants applied in it (e.g., x and y) are not yet available. Therefore, name analysis and code generation for the expression part have to be postponed until the constant definition part of the program has been processed, leading necessarily to a *multipass compilation* scheme.

The ability to describe multipass processing is an inherent and powerful characteristic of attribute grammars. This computational expressiveness is provided by inherited attributes that can model data flow into a construct from its context. This aspect is demonstrated in our DESK grammar by the dependency from the synthesized attribute occurrence ConstPart.envs to the inherited occurrence Expression.envi in production $p1$.

The notation used in the attribute grammar (1.2) is quite common in exist-

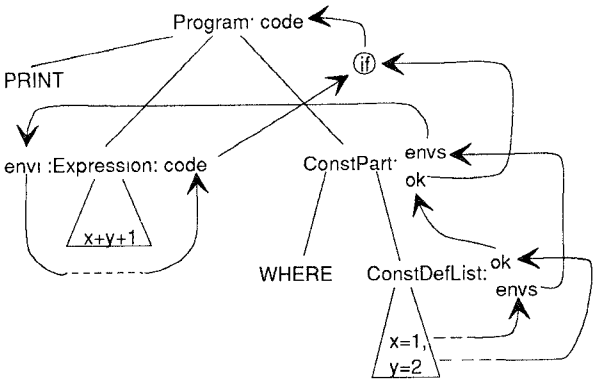


Figure 4. Attribute dependency graph over an attributed tree.

ing language processor generators. A specification of the DESK language and its compiler, written in the input language (the *metalanguage*) of such a system, contains typically the following components:

(1) a list of grammar symbol declarations (N, T) with associated attributes (A),
(2) a sequence of productions (P) with associated semantic rules (R), and
(3) the semantic functions (f) in an external file, usually written in some conventional programming language.

This simplistic style reflects the definition of attribute grammars in a straightforward way and is used in all the older systems. Multipass systems based on this "pure" attribute grammar paradigm are for instance GAG [Kastens et al. 1982], LINGUIST-86 [Farrow 1982], and HLP78 [Räihä et al. 1983]. From the grammar (1.2) such a system would generate a multipass DESK compiler that (1) scans and parses the source program, (2) constructs an explicit attributed derivation tree as the intermediate representation for the program, and (3) evaluates the attribute instances in the tree during a traversal over it. Phase (3) would involve both (static) semantic analysis and target code generation. Examples of one-pass systems whose metalanguage follows the standard paradigm are MIRA [Expert Software Systems 1984] (originally developed as LILA [Lewi et al. 1979]), and YACC [Johnson 1975; Mason and Brown 1990].

The standard attribute grammar notation has, however, severe practical shortcomings. This stems from the fact that attribute grammars, as a concept, are just a theoretical model of language processing without any advanced engineering facilities. The grammars written in this manner tend to become most unstructured and lengthy. As an example, an attribute grammar for the programming language Ada written in GAG's specification language comprises an entire book with over 500 pages and 20,000 lines, more than 260 homogeneous productions, and about 600 global semantic functions [Uhl et al. 1982]. The problems with such specifications are parallel with the known problems encountered when programming in a low-level unstructured programming language: the specifications are hard to understand, modify, and maintain. Some attribute grammars are even larger than their procedural counterparts (e.g., compilers), an aspect that makes it rather questionable to consider such grammars as *specifications*.

Another problem with standard attribute grammars is their unduly strict notion of dependencies and well-definedness. Recall that, by definition, an

attribute instance cannot be evaluated until all the instances on which it depends have been evaluated, and that the semantic rules shall not introduce any potential circularities into the attribute dependency graphs. In some cases these restrictions force an attribute grammar to be written in a form which is not the most natural or intuitive one. Note that to an extent these restrictions are merely technical and due to the assumption of applying a strict computational mechanism in attribute evaluation.

In the following chapters we will present techniques that have been developed for removing the above-mentioned drawbacks of standard attribute grammars. The approaches can be roughly classified into two categories: *organizational paradigms* and *evaluation paradigms*. The organizational paradigms provide facilities to compose an attribute grammar of autonomous, yet integrated units. By this, the grammars become more manageable and easier to understand. The evaluation paradigms, on the other hand, employ powerful computational mechanisms in attribute evaluation, thus raising the expressive power beyond standard attribute grammars.

The organizational paradigms will be discussed in Section 2. We start in Section 2.1 by presenting *attribution paradigms* where abstract notations are provided for commonly applied patterns of semantic rules. More general solutions adopt their model from established programming paradigms that are founded on some central structural concept. These are presented under the notions of *structured attribute grammars* (based on the concept of a block), *modular attribute grammars* (based on the concept of a module), and *object-oriented attribute grammars* (based on the concept of a class) in Sections 2.2, 2.3, and 2.4, respectively.

In Section 3 we will present evaluation paradigms. Section 3.1 discusses *logic attribute grammars* where the special features of logic programming are utilized for increasing the semantic power of attribute grammars. A similar effect, using functional programming, is demonstrated in Section 3.2 under the notion of *functional attribute grammars*. In order to give a complete picture of the state of the art, we briefly discuss in Section 3.3 two special techniques, *parallel* and *incremental* evaluation, that are currently subjects of intensive research.

The main principles of the different paradigms are illustrated by examples showing how some representative systems tackle the problems discussed above. For a quick overview, example systems promoting the paradigms are given in Table 1. Some of the systems in the table are given merely for reference and will not be further discussed in Sections 2 and 3.

2. ORGANIZATIONAL PARADIGMS

In this section we present techniques for managing the complexity of an attribute grammar by organizing some of its parts as linguistic units. The methods are similar to those developed for structuring ordinary programs in terms of concepts like compound statements, blocks, procedures, modules, packages, and classes. Accordingly, the presented paradigms affect the *structure* of the attribute grammar rather than its semantics. As a consequence, a specification written in any of these styles could be translated into a semantically equivalent standard attribute grammar by reducing its abstraction level (as a program written in a high-level programming language can be compiled into machine code). In fact, this is the implementation method applied in some systems, although the normal practice is to translate an organized attribute grammar directly into a programming language with similar structuring facilities.

2.1 Attribution Paradigms

Often a significant number of semantic rules in a typical attribute grammar are much alike. Since these attribute definition patterns occur very frequently, using short and comprehensible special

Table 1. Classification of Attribute Grammar-Based Systems

STANDARD PARADIGM		
Pure attribute grammars	YACC [Johnson 1975]	LILA [Lewi et al. 1979]

ORGANIZATIONAL PARADIGMS		
Attribution paradigms	GAG [Kastens et al. 1982]	HLP78 [Räihä et al. 1983]
Structured attribute grammars	HLP84 [Koskimies et al. 1988]	FNC-2 [Jourdan et al. 1990]
Modular attribute grammars	MAGGIE [Dueck and Cormack 1990]	Linguist [Declarative Systems 1992]
Object-oriented attribute grammars	Mjølner/Orm [Hedin 1989]	TOOLS [Koskimies and Paakki 1990]

EVALUATION PARADIGMS		
Logic attribute grammars	Pan [Ballance et al. 1990]	PROFIT [Paakki 1991]
Functional attribute grammars	FNC/ERN [Jourdan 1984]	YACC/CAML[Cousineau and Huet 1990]
Parallel attribute evaluation	FOLDS [Fang 1972]	[Boehm and Zwaenepoel 1987]
Incremental attribute evaluation	OPTRAN [Lipps et al. 1988]	The Synthesizer Generator [Reps and Teitelbaum 1989]

notations for them may drastically reduce the size of a language (processor) specification. These compact abstractions can be systematically transformed into the basic notations without changing any properties of the attribute grammar. A number of systems provide such *attribution paradigms* in their specification languages; examples include HLP78 [Räihä et al. 1983], GAG [Kastens et al. 1982; 1987], HLP84 [Koskimies et al. 1988], LINGUIST-86 [Farrow 1982], Metauncle [Tarhio 1989], and FNC-2 [Jourdan et al. 1990]. An introduction to the attribution paradigms is included in Kastens [1991].

Analyses of attribute grammars have shown that many of their semantic rules are simply *copy rules* (or "transfer rules") of the form

$$X.a = Y.a.$$

Copy rules are frequently needed especially in describing the propagation of context information, such as a compiler's symbol table. Some systems either provide a special notation for such attribution schemes, or simply include them as implicit default rules.

For instance, GAG provides the attribute transfer rule TRANSFER which, when associated with a production $X_0 \rightarrow X_1 \cdots X_n$, denotes (1) a copy rule $X_0.a = X_i.a$ for each attribute $a \in S(X_0)$ that is associated with exactly one symbol X_i in $X_1 \cdots X_n$, and (2) a copy rule $X_j.b = X_0.b$ for each attribute $b \in I(X_0)$ and for each symbol X_j in $X_1 \cdots X_n$ that owns the attribute b. This mechanism is based on the natural convention that attributes describing the same property of different symbols should have the same name.

Using this facility, the first production for Expression and the production for ConstDef in the grammar (1.2) can be written as follows:

```
(p2)   Expression1 → Expression2 '+' Factor
       {Expression1.code = if Factor ok
            then Expression2.code + (ADD,
              Factor.value)
            else (HALT, 0),
       TRANSFER}

(p11)  ConstDef → ConstName ' = ' Number
         {TRANSFER}
```

Another commonly applied attribution pattern is to use an attribute whose instance is located in a context other than the current local one. In terms of the dynamic attributed tree, the value of an attribute instance for a node n depends in such a case on an attribute instance a that is associated with an ancestor or a descendant node of n. In the standard framework the value of a would have to be propagated to n with a chain of copy rules. There exist, however, more convenient techniques to handle such a situation.

HLP78 provides a form of *global attributes* that were originally proposed in Knuth [1971]. A global attribute instance can be directly accessed and updated within the subtree with whose root it is associated, without passing it explicitly through the nodes in the subtree. The drawback of the approach is that it is not always easy to foresee the exact execution order of the semantic rules on a global attribute. This problem is removed, e.g., in HLP84 where global attributes are tuned solely for one-pass language processing (with an unambiguous execution order).

GAG provides the remote access instructions INCLUDING (for accessing global attribute instances within a subtree) and CONSTITUENTS (for composing the value of a global attribute instance of distinct attribute values within the subtree). The expression

INCLUDING ($Y_1.a_1, \ldots, Y_m.a_m$)

associated with the production $X_0 \rightarrow X_1 \cdots X_n$ yields the value of either $Y_1.a_1$, or ..., or $Y_m.a_m$, depending on which of the symbols $Y_1, \ldots, Y_m$ is the label of the nearest ancestor of the node for X_0 in the attributed tree. The expression

CONSTITUENTS Y.a

associated with the production $X_0 \rightarrow X_1 \cdots X_n$ yields a list of attribute values Y.a from left to right in the subtree whose root is the curret node for X_0. In both cases the attribute grammar must guarantee that the expression always produces a proper value.

As an example, the attribute grammar (1.2) can be revised such that the symbol table is expressed as a global attribute for the DESK expression, and the target code is a list of individual instructions. Then a DESK expression can be specified as follows:

```
(p1)   Program → 'PRINT' MainExpression ConstPart
       {Program.code = if ConstPart.ok
            then MainExpression.code + (PRINT, 0) + (HALT, 0)
            else (HALT, 0),
       MainExpression.envi = ConstPart.envs}

(p1')  MainExpression → Expression
       {MainExpression.code = CONSTITUENTS Expression.code}

(p2)   Expression1 → Expression2 '+' Factor
       {Expression1.code = if Factor.ok
            then (ADD, Factor.value)
            else (HALT, 0)}

(p3)   Expression → Factor
       {Expression.code = if Factor.ok
            then (LOAD, Factor.value)
            else (HALT, 0)}
```

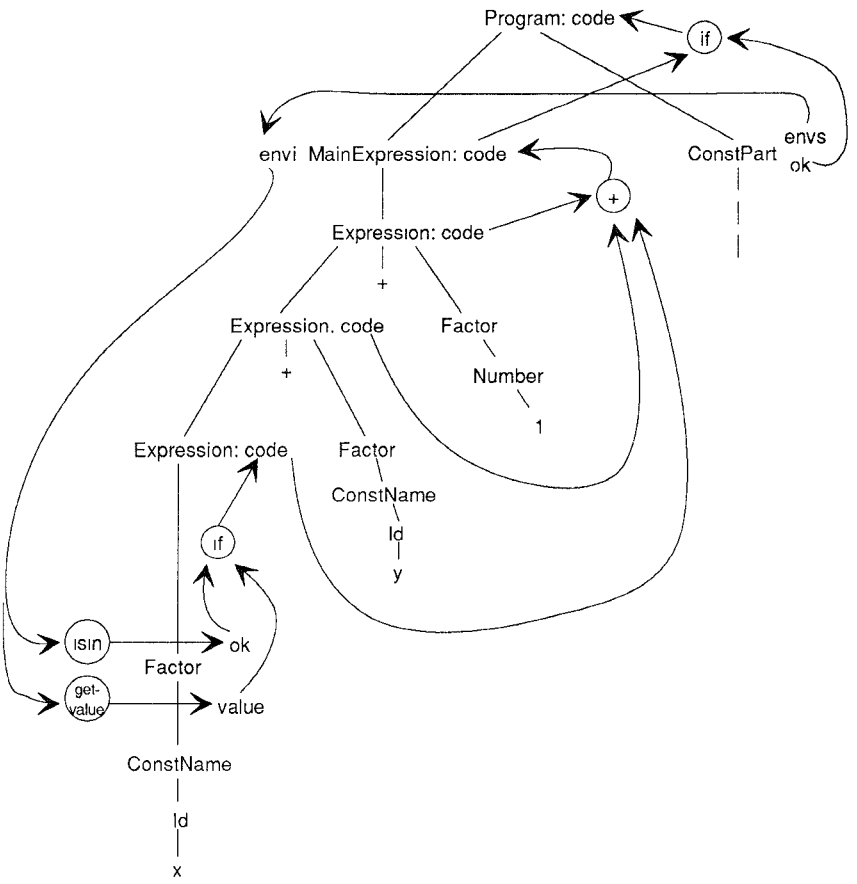


Figure 5. Globalized attributed tree.

(*p*4) Factor → ConstName
{Factor.ok = isin (ConstName.name,
INCLUDING (MainExpression.envi)),
Factor.value = getvalue (ConstName.name,
INCLUDING (MainExpression.envi))}

(*p*5) Factor → Number
{Factor.ok = *true*, Factor.value = Number.value}

(*p*6) ConstName → Id
{ConstName.name = Id.name}

The productions are shorter and simpler than in the original grammar (1.2), thanks to replacing the local attribute occurrences envi with the global attribute occurrence MainExpression.envi. Also the definition of Expression.code is simpler because it now represents one target instruction only instead of the complete code sequence synthesized from a subtree. Note that in the case of an invalid expression the final target code might now be different from that specified in grammar (1.2). The dynamic semantics of DESK, however, is the same: computation of an invalid expression will always be aborted by a HALT instruction.

We clarify the globalizing method by sketching in Figure 5 the attributed tree

for the DESK program (1.1). The attribute evaluation pattern in each Factor context is the same as shown for the leftmost one.

Note that now the Expression and Factor nodes do not have an instance of the envi attribute at all, whereas conceptually the instances remain both in the grammar and in the tree when using the copy rule elimination technique discussed above (although a space-optimizing system, such as HLP78 or GAG, would even then probably remove them automatically from the tree).

Semantic error handling is cumbersome in standard attribute grammars. For instance, the grammar (1.2) makes use of the ok attribute for expressing static semantic restrictions. GAG and HLP84, among others, extend attribute grammars with *consistency conditions* that are boolean-valued expressions over attribute occurrences. The conditions can be associated with nonterminals (HLP84) or productions (GAG). As an example, the following production in GAG's convention expresses the constraint that each named constant applied in a DESK expression must be defined. The consistency condition involves an error message to be emitted if the condition is violated.

```
(p4)  Factor → ConstName
      {Factor.value = getvalue
      (ConstName name,
        INCLUDING
          (MainExpression envi));
      CONDITION isin (ConstName name,
      INCLUDING (MainExpression.envi))
      MESSAGE "Constant not defined"}
```

Notice that by this feature the ok attribute and its semantic rules (some of which are redundant in grammar (1.2)) become useless. The incorrectness of a DESK program is now manifested by error messages that are given by the generated compiler at run-time.

2.2 Structured Attribute Grammars

The key issue in managing large systems is *decomposition*, or *structuring*. A system with small integrated components is easier to build and maintain than a monolithic system. The fundamental concepts for structuring software have been developed in parallel with the evolution of programming languages. The first structuring primitives introduced into programming languages were subprograms in Fortran, compound statements in Algol 58, and blocks in Algol 60. These have been the basic models for a number of later improvements.

In this section we address the question of how to apply conventional structuring mechanisms of programming languages to attribute grammars. The term "structured programming" has in this context a specific meaning. Two traditional narrow interpretations might consider the term as a synonym to "programming without goto statements" or "programming with structured data types." However, since statements or data types conceptually (albeit in practice, of course) are not central issues in attribute grammars, structured programming is here considered as a programming style based on *blocks* and *procedures*.

Blocks

A block is a language structure with an arbitrary number of local entities that may be blocks themselves. Hence, blocks can be characterized with the following general propeties:

- static partition of programs into hierarchical elements and
- local name environments.

A dominant characteristic of programming languages is *visibility (scope) rules* that specify how a name applied in a program is associated with its declaration. This principle, which strongly affects both language design, definition, and implementation, originates in Algol 60's notion of nested blocks. In Algol 60 an applied occurrence of a name is associated with its defining occurrence such that if the program contains several defining occurrences for the name, then

the one belonging to the innermost block enclosing the applied occurrence is selected. This principle of statically determining the name association has been adopted in most structured programming languages.

Attribute grammars in their standard form, as defined in Section 1.2, provide only limited structuring facilities. They can be characterized with the correspondence

Production = Block

since a standard attribute grammar is composed of a list of productions with their local semantic rules. This approach was demonstrated in Example 1.3.1. The local declaration property of blocks is provided in some metalanguages as attributes that are local to a production (e.g., CWS2 [Bochmann and Ward 1978] and FNC-2 [Jourdan et al. 1990]). This conceptual coupling of productions and blocks is, however, rather primitive because typically only a few productions can sensibly make use of local attributes and because the nesting obtained this way is merely flat (productions cannot be given within productions). Hence, more natural and general means have to be established in order to introduce useful structuring facilities into attribute grammars.

In a declarative attribute grammar, the semantic emphasis should be laid on describing the nonterminals in terms of attributes. Thus, a natural design decision is to choose a nonterminal as the structuring primitive of attribute grammars, and to reflect conventional structuring methodologies on nonterminals. This view gives rise immediately to the following characterization of *structured attribute grammars* and, from an operational point of view, for a *structured language processing methodology*:

Nonterminal = Block

In other words, nonterminals of attribute grammars correspond to blocks of programming languages, in that

(1) nonterminal definitions can be arbitrarily nested,
(2) nonterminals can have local entities, and
(3) visibility of entities is controlled by conventional block-structured scope rules.

Nonterminal-based structuring makes a language specification more comprehensible by dividing it into logical components: each nonterminal definition specifies a sublanguage, and the nesting describes the hierarchy of those sublanguages. As is normal in block-oriented structuring, a sublanguage that is included in several contexts of the whole language must be expressed by a nonterminal which is defined on a global level.

It must be emphasized that this "Nonterminal = Block" correspondence is only employed for structuring language *specifications*, not for mapping nonterminals into blocks in the *implementation* of attribute grammars. In general, the structure of a specification and the structure of the corresponding program are completely different topics. For instance, HLP84 whose metalanguage is block structured generates language processors where this structuring is totally effaced [Koskimies et al. 1988].

Procedures

In addition to blocks, nonterminals can be mapped to other structuring units as well, giving rise to alternative forms of structured language processing. The conventional style of programming language processors is to apply the *recursive-descent* method where the nonterminals of the underlying grammar are represented by recursive procedures.[5] When considering parameters of these proce-

[5] In this case nonterminals are mapped into the implementation. However, recursive-descent compilers are factually equivalent with LL-attributed grammars [Wilhelm 1982], and thus represent a special case where implementation follows specification closely.

dures as "semantic attributes," we can characterize the recursive-descent method as being based on the following correspondence:

Nonterminal = Procedure

Most implementations of *L*-attributed grammars generate (one-pass) recursive-descent compilers that apply this correspondence operationally. Nonterminals are considered as procedures even conceptually in some language processor generators. For instance, the metalanguage of DEPOT2a, ML2a, is stuctured according to the correspondence given above [Grossmann et al. 1984]. A language (processor) specification written in ML2a is composed of procedures such that each procedure expresses the processing of a sublanguage in terms of other procedures. Inherited attributes are represented as input parameters, and synthesized attributes as output parameters. The block structure in ML2a is flat: all the nonterminal procedures must be defined at the same level of the specification. In DEPOT2a the "Nonterminal = Procedure" correspondence is carried over from the metalanguage into the implementation, and the generated language processors are backtracking recursive descent.

2.2.1 TOOLS

TOOLS is a language processor generator whose metalanguage is structured following the style outlined above: nonterminals are expressed as blocks [Koskimies and Paakki 1990; 1991]. The system is a descendant of HLP84 [Koskimies et al. 1988] that was designed to support one-pass language implementation with a high-level extended attribute grammar as the metalanguage. The extensions, available both in HLP84 and TOOLS, include (besides the structuring facilities) a restricted form of global attributes, semantic consistency conditions, and an abstraction of "ordinary" symbol table operations.

TOOLS is much more powerful than HLP84, being able to generate multipass implementations as well. This is achieved by specifying in the attribute grammar both a mapping of the source program into its intermediate abstract syntax tree and a traversal scheme over the tree. The traversal may correspond to the code generation phase of a compiler, or to immediate interpretation.

TOOLS is a versatile system with an expressive specification language. The custom of describing the abstract syntax tree is to map instances of central nonterminals with nodes of the tree. Those nonterminals that correspond to syntax tree nodes are specified as *classes* in the attribute grammar. This principle brings an object-oriented flavor into TOOLS. We do not address this aspect here but refer to Section 2.4 where object-oriented attribute grammars are discussed.

Let the attribute grammar for a language include the following two productions for the nonterminal N, where A and B denote other nonterminals:

N → ··· A ··· {⟨semantic rules (1)⟩}
N → ··· B ··· {⟨semantic rules (2)⟩}

In TOOLS' metalanguage, the grammar fragment is written in the following block-structured form:

```
structure N: ⟨attributes of N⟩;
  structure A: ⟨attributes of A⟩; ... end A,
  structure B: ⟨attributes of B⟩, ... end B;
  form ... A ... do ⟨semantic rules (1)⟩;
  form ... B ... do ⟨semantic rules (2)⟩;
end N;
```

Hence, the nonterminal N is a block (expressed by "**structure** N ... **end** N") with local declarations. The nonterminals used in the productions of N (here A and B) are declared as local nonterminals of N. As usual in block-structuring, A (or B) has to be declared on a global level if it is also referenced outside of the block for N. A **form** clause stands for a production (its right-hand side) and its associated semantic rules for the enclosing nonterminal. Note the centralizing effect of block-structuring: all the aspects of a nonterminal can be specified as one unit,

without spreading them over individual productions throughout the specification, as is the case with standard attribute grammars.

As an example of a block-structured attribute grammar, we describe our DESK language (see Example 1.3.1) using the metalanguage of TOOLS. Since the preferable application area of TOOLS is interpretation, the grammar specifies now an interpreter rather than a compiler. Notice that this solution is actually quite natural in this case since the primitive DESK language does not have any recursive or iterative control structures.

The structure of the specification can be systematically extracted from the context-free grammar of DESK. When following the guidelines given above, the following nesting of grammar symbols is obtained:

```
Program
  Id
  Number
  Expression
    Factor
      ConstName
  ConstPart
    ConstDefList
      ConstDef
        ConstName
```

Notice that Id and Number must be declared at the same level as Expression and ConstPart to make them globally visible for the local nonterminals Factor, ConstName, and ConstDef. Notice also that the hierarchy above introduces two different nonterminals of the same name ConstName. This reflects the fact that even though the occurrences have the same name, they have different semantics: the ConstName occurrence for Factor stands for an application of a defined constant, whereas the ConstName occurrence for ConstDef represents the definition of a constant. This distinction matches with the TOOLS methodology better than the unified solution used in the attribute grammar (1.2), and it also demonstrates the central principle of local naming in block-structured attribute grammars. The specification of the DESK interpreter is given below.

```
external
  procedure Print (i:Integer);
end;

structure Program;

  type Constant = class   —symbol table
    key name: String;
    value: Integer;
  end;

  —intermediate tree and its interpretation
      (dynamic semantics):

  type Expr = class
    value: Integer;
    sort Literal: ( );
    sort ConstRef: (name: String);
    begin
      value:= [Constant]name.value;
    end;
    sort Add: (left: Ref (Expr); right: Ref (Expr));
    begin
      execute left;
      execute right;
      value:= left.value + right.value;
    end;
  end;

—attribute grammar (static semantics):

  token Id: String = Letter + ;
  token Number: Integer = Digit + ;

  structure Expression: Ref (Expr);
    structure Factor: Ref (Expr);
      object ConstName: Ref (ConstRef);
        form Id do name:= Id end;
      end ConstName;
      object LiteralConstant: Ref (Literal);
        form Number do value:= Number;
      end LiteralConstant;
      form ConstName do
      Factor:= ConstName;
      form LiteralConstant do
      Factor:= LiteralConstant;
    end Factor;
    object Addition: Ref (Add);
      form Expression '+' Factor do begin
        left:= Expression;
        right:= Factor;
      end;
    end Addition;
    form Addition do Expression:= Addition;
    form Factor do Expression:= Factor;
  end Expression;

  structure ConstPart;
    structure ConstDefList;
      object ConstDef: Ref (Constant);
        structure ConstName: String;
```

```
        form Id do ConstName:= Id,
      end ConstName,
      form ConstName '=' Number
        do begin
        name:= ConstName;
        value:= Number;
      end,
    end ConstDef;
      form ConstDefList ',' ConstDef,
      form ConstDef,
    end ConstDefList;
    form Empty;
    form 'WHERE' Const'DefList,
  end ConstPart,
  form 'PRINT' Expression ConstPart
  do begin
    execute Expression;
    Print (Expression.value);
  end;
end Program
```

TOOLS provides high-level abstractions for managing hierarchical and linked data structures. In this application we have used these abstractions for the definition of the symbol table and the intermediate syntax tree. They are represented in the specification as the class types Constant and Expr, respectively, with associated attributes.

The symbol table collects all the constants defined in a DESK program. Each of them has name and value as its attributes. The name attribute is used for identifying the constant (**key** indication). A symbol table lookup is expressed by the notation [Constant]X where **X** is the key value used for identification. A symbol table insertion is specified by denoting a nonterminal as being an **object** (of type Ref (Constant)) instead of being an ordinary **structure**. The created symbol table entry receives its attribute values from the associated nonterminal, in this case from an instance of ConstDef. Both operations, lookup and insertion, automatically yield an error message if a constant with the same name is already in the symbol table, or if the referenced constant is not found there, respectively.

The intermediate tree has several kinds of nodes, one for each different executable DESK entity. In this example an expression may consist of literal constants (**sort** Literal), named constants (**sort** ConstRef), and additions (**sort** Add). The creation of a node is specified in the same manner as for symbol table entries, with **object** nonterminals in the attribute grammar. The dynamic semantics (that is, interpretation) of the intermediate tree is described by associating sort-specific operations with the class Expr. The **execute** clause denotes execution of the code associated with the indicated node. Note that dynamic semantics is specified operationally rather than declaratively.

The actual attribute grammar is organized following the block-structured principles discussed above. The attributes of a symbol are defined by a semantic type that follows the symbol's name. Ref (C) is a predefined type for (symbol table and intermediate tree) references of class C. If the semantic type is simple (e.g., Ref (Expr)), the associated symbol has only one attribute of the same name as the symbol itself. The attributes of terminal symbol instances (Id and Number) are automatically evaluated from the lexical representation of the instance in the source program. We have included some technical **object** nonterminals in the grammar for specifying the creation of the symbol table and the intermediate tree. Notice how the decision to use conventional blocks as the model of structuring the attribute grammar separates the productions of a nonterminal from its attributes, a feature which may make it hard for a novice to read the grammar.

From this specification, TOOLS generates a "partially two-pass" interpreter for DESK. The first pass does lexical, syntactic, and static semantic analysis of the constant definitions, as well as creates an intermediate tree for the expression. For example, after executing the first pass over the program

```
PRINT x + y + 1 WHERE x = 1, y = 2
```

the symbol table (Constant) and the intermediate tree (Expr) have the contents shown in Figure 6. Dotted lines denote symbol table bindings to be realized in the second pass of interpretation.

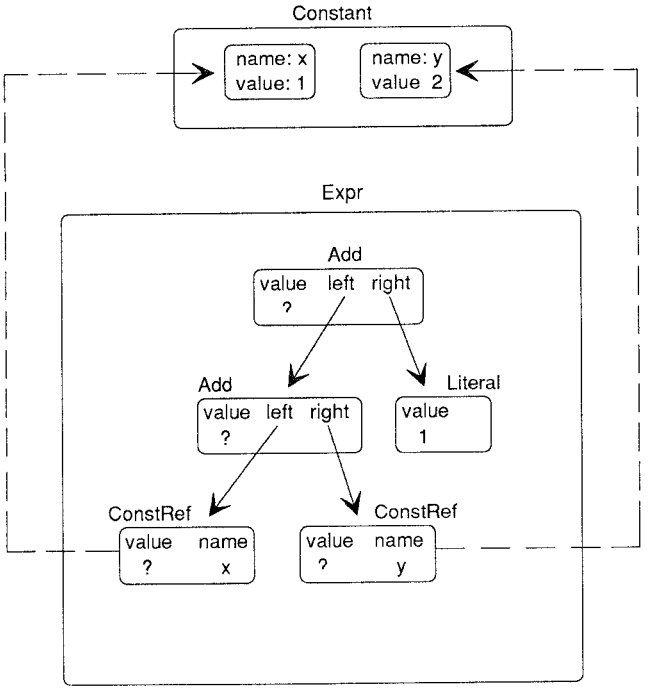


Figure 6. Symbol table and intermediate tree.

In the second phase the tree is traversed, and the intermediate code associated with its nodes is executed. Due to the use-before-declaration feature of DESK, this phase also involves resolving and checking referenced constants. Interpretation is finished by printing the value of the expression using the external (Pascal) procedure Print. In the situation depicted in Figure 6, the value 4 will be produced as a result.

Note that TOOLS makes it necessary to specify the structure of the intermediate syntax tree explicitly, whereas the tree would be implicit in more declarative multipass systems, such as GAG [Kastens et al. 1982]. On the other hand, the TOOLS approach allows for quite general and complex language processing strategies (as interpretation in this example) instead of supporting a fixed compilation scheme only.

2.3 Modular Attribute Grammars

Blocks and procedures are sufficient for organizing programs on a small scale. These concepts of structured programming, however, fall short in sizable applications. That is why more advanced facilities have been developed for programming-in-the-large. Such concepts, generally called *modules*, support in particular the construction of programs from independent units that provide services to each other through an explicit narrow interface. Modules are superior to blocks and procedures in their reuse and modification capabilities: a well-designed module can be incorporated in many programs, and it may be revised without forcing major (if any) changes to other modules.

Therefore, even though modules are in many respects similar to blocks and procedures, this particular programming style is usually characterized by the special term *modular programming*. The paradigm involves, besides the general properties of structured programming outlined in Section 2.2, also the following fundamental characteristics:

- data abstraction
- information hiding

• separate compilation.

These modular programming features make it possible to design *abstract data types*. In an abstract data type specification the values of the type are not given explicitly but are instead defined indirectly by the operations. An abstract data type specification (as well as a module in general) is usually divided into two parts: an *interface part* introduces the public components that are available to other modules, and a separate *implementation part* gives the exact realization of the components. The data type is "abstract" in the sense that a user of the type has only access to the interface of the public components, not to their implementations nor to the private components of the module. Thus the implementation part can freely be modified without inducing any changes into other parts of the program, as long as the interface remains the same.

Another pragmatic merit of modules is that they are autonomous to the extent that they can be compiled separately from other modules. This facilitates incremental software development, makes it feasible to assemble program libraries, and reduces recompilation efforts on software updates.

In the spirit of the structured attribute grammar paradigm, one form of a *modular attribute grammar* methodology is founded on the following correspondence:

Nonterminal = Module

Now the nonterminals of an attribute grammar are represented as modules that make it possible to specify and store linguistic elements completely that are common in several programming languages. Such sublanguage specifications can then be used in a new language specification by importing the nonterminal modules into the new specification. The interface of a nonterminal module consists of the attributes of the nonterminal, whereas the semantic rules are hidden within the implementation part. This makes it possible to change the semantics of a nonterminal without modifying its clients. The same also holds for the syntax: the productions for the nonterminal and for the grammar symbols appearing in them are given only in the implementation part.

Modular attribute grammars introduce an interesting implementation problem: how can a lexical and syntactic analyzer be produced for a nonterminal without any global syntactic information (such as the *First* and *Follow* sets; e.g. Aho et al. [1986])? One approach recently suggested is to employ a lazy/incremental method where the analyzers are constructed dynamically at parse-time, simultaneously with parsing the input (see Heering et al. [1990] or [Koskimies 1990]).

Attribute grammars based on modularizing the nonterminals have been discussed, e.g., in Koskimies [1989] and in Toczki et al. [1989]. Other, principally different methods of modularizing attribute grammars have also been developed. The correspondence

Attribute = Module

is the basis in MAGGIE [Dueck and Cormack 1990]. This approach suggests that attributes, rather than nonterminals, are the central semantic elements of an attribute grammar. In this case a module contains all the semantic rules for the corresponding attribute, which implies that the module also must list all the productions that define an occurrence of the attribute. The modular properties of MAGGIE are rather primitive since they do not support true information hiding nor separate implementation. (Before compiler generation, the modular structure of the attribute grammar is effaced by a transformation into a standard monolithic grammar.)

A more general and flexible form of modularity is suggested in, e.g., Watt [1985], Reps and Teitelbaum [1989], Jourdan et al. [1990], and Farrow et al. [1992], that pursue modularization in terms of different semantic subproblems. The following characterization summa-

rizes this direction for modular attribute grammars:

Semantic aspect = Module

Now a module contains all those parts of an attribute grammar that assist with the specification of a semantic aspect, typical examples being the scope and type rules of the language, data-flow analysis, and code generation. Typically the method assigns several nonterminals, attributes, and productions to the same module. This general form of modularization can also be applied in a more operational style by mapping the modules with the different phases of a compilation process, as suggested in Ganzinger and Giegerich [1984] and Ganzinger et al. [1982].

One way of introducing data abstraction into attribute grammars is to consider the semantic functions as abstract operations on attribute domains. Then the operations applied on the same domain constitute an abstact data type that can be externally implemented. The attribute grammar can be pasted together with the abstract data types in the usual manner, through the interfaces of the types. This method has been suggested, e.g., in Waite [1986], as an enhancement of the GAG system [Kastens et al. 1982].

An application-oriented utilization of data abstraction is to focus on the central data structure in language implementation: the symbol table. Considering the symbol table as an abstract data type makes it possible both to specify symbol processing on an abstract level in the attribute grammar and let the underlying system generate a suitable implementation.

The obvious problem with this idealistic view is that the visibility and type rules of two languages can be totally different, and that even related languages may have subtle variations in their symbol processing policy. That is why it is quite hard to design high-level abstractions that universally apply to a large class of programming languages. One general model of visibility control is given in Garrison [1987], including an extensive study of the variations in different languages. However, the model has not been implemented. Another approach is to concentrate on features that are common in a family of related languages and to design high-level abstactions for those languages only. Such a resticted symbol table abstraction for one-pass languages has been presented in Koskimies [1984] and implemented in HLP84 [Koskimies et al. 1988] and TOOLS [Koskimies and Paakki 1990]. Reiss [1983] presents a more general model that is influenced most notably by the features of Ada. The model has been integrated as part of the ACORN compiler writing system [Reiss 1987]. Other approaches to high-level description of common symbol table mechanisms include those of Vorthmann and LeBlanc [1988] and Kastens and Waite [1991].

2.3.1 Linguist

Modular attribute grammars are discussed in Farrow et al. [1992] under the notion of *composable attribute grammars*. The module concept of this formalism has been designed to support reuse and separate implementation of attribute grammar units. Composable attribute grammars are directed at separate specification of different subproblems in language processing and thus belong to the category "Semantic aspect = Module" of the modular attribute grammar methodology. A restricted class, *separable composable attribute grammars*, has been implemented in the Linguist translator writing system [Declarative Systems 1992].

A composable attribute grammar consists of a set of *component* (module) attribute grammars and one *glue* grammar. Each component specifies the phrase structure of a particular subproblem. Since the components should be highly reusable, they are expressed in terms of abstract, language-independent context-free grammars. The syntactic structure of the language under implementation is specified as the glue grammar, where the syntax of the language is given as a con-

crete context-free grammar, and the semantics is defined in terms of relevant component grammars. The ideal principle is to pick the components from a specification library fit for the particular language and for the specific implementation strategy.

The interface of a component consists of its (abstract) context-free grammar and its association of data flow with the (abstract) terminal symbols. The novel design decision has been to base the semantic interfacing on the *input* and *output* attributes of the terminals of a component attribute grammar. The input attributes represent contextual data to the component, while the output attributes represent resulting semantic information. This principle is quite simple and makes it easy to verify the correctness of the final interfaces. On the other hand, the method may in some cases force the introduction of unintuitive and artificial semantic handles into the components.

The explicit interactions between component attribute grammars are not fixed until the glue grammar is constructed. Hence, modularity is preserved, and a component can be freely modified without affecting the other components. A modification of a component even leaves the glue grammar unchanged, as long as the interface remains unaffected. This property makes it possible to design interchangeable components having the same phrase structure and interface (i.e., they address the same problem) but expressing different solutions for different classes of languages.

As an example of the approach, we specify the DESK compiler as a composable attribute grammar. Our implementation must take care of two semantic subproblems: name analysis and code generation. Accordingly, it is reasonable to base the modular specification on two component attribute grammars, one for name analysis and the other for code generation.

Since the notation for composable attribute grammars has not been absolutely fixed in Farrow et al. [1992], we use the same basic notations as earlier. An abstract terminal symbol t with input attributes $i_1, \ldots, i_n$ and output attributes $o_1, \ldots, o_m$ is defined as follows:

terminal t: {in $i_1, \ldots, i_n$; out $o_1, \ldots, o_m$}

The productions are preceded by their symbolic name, to be used in the glue grammar. A grammar module M is expressed in the form

module M
...
end M.

The component grammar Env for name analysis is given below. For emphasizing the reuse property of grammar modules, the component specifies symbol table management in a slightly more general manner than is actually needed for implementing the DESK language. The component specifies the pattern of the name analysis phase for a typical unstructured programming language with declared symbols. The grammar could thus be adopted as a fragment in the specification and implementation of another language in the unstructured family. We make use of the same attributes as in the attribute grammar (1.2) of Example 1.3.1.

```
Module Env
  terminal symbolDEF  {in  name, value, out ok}
  terminal symbolREF: {in  name, out value, ok}
  init  symbols1 → symbols2
  —creation of an empty symbol table
    {symbols2.envi = ( ),
      symbols1 envs = symbols2.envs}
  empty. symbols → ε
      —empty structure
    {symbols envs = symbols.envi}
  chain· symbols1 → symbols2
      —symbol table propagation
    {symbols2.envi = symbols1.envi,
    symbols1.envs = symbols2.envs}
  list· symbols1 → symbols2 symbols3
      —recursive structure
    {symbols2.envi = symbols1.envi,
    symbols3.envi = symbols2.envs,
    symbols1.envs = symbols3 envs}
  def· symbols → symbolDEF
      —symbol declaration
```

```
    {symbols.envs = symbols.envi +
      (symbolDEF.name, symbolDEF.value),
    symbolDEF.ok = not
      isin (symbolDEF.name, symbols.envi)}
  ref: symbols → symbolREF
    —symbol reference
    {symbols.envs = symbols.envi,
    symbolREF.value
      = getvalue (symbolREF.name, symbols.
        envi),
    symbolREF.ok
      = isin (symbolREF.name, symbols.envi)}
end Env.
```

The component grammar Code for assembly code generation is given below. The module is constructed with the observation that in an assembly problem of this kind we must be able to handle (1) generation of single instructions, (2) generation of instruction sequences, and (3) provision of final code.

```
module Code
  terminal loadable:   {in: value, ok}
  terminal operand:    {in: instr, ok}
  terminal end:        {in. ok, out: code}
  single: assembly → loadable
    —single load instruction
    {assembly.code = if loadable.ok
    then (LOAD, loadable.value)
        else (HALT, 0)}
  seq: assembly1 → assembly2 operand
    —sequence
    {assembly1.code = if operand.ok
        then assembly2.code
        +operand.instr
        else (HALT, 0)}
  final: assembly1 → assembly2 end
    —total code
    {assembly1.code = assembly2.code,
    end.code = if end.ok
    then assembly2.code + (PRINT, 0)
    +(HALT, 0)
    else (HALT, 0)}
end Code.
```

Finally, the glue attribute grammar for the DESK language is constructed. The complete specification imports the component grammars Env and Code. The method of integrating component grammars into a total attribute grammar is to use *syntactic attributes* and *production constructors* in the glue grammar. A nonterminal N of the glue grammar may have a syntactic attribute with the same name *a* as a nonterminal defined in one of the component grammars. Suppose that a component grammar includes the following definition for *a*:

$$r: a \rightarrow b_1 \ldots b_n \{\ldots\}$$

Then the semantic rule for N.a must be of the following form:

$$N.a = r(b'_1, \ldots, b'_n)$$

where r is a production constructor (i.e., the name of a production in the component grammar), and each b'_i is an occurrence of a syntactic attribute or an application of another production constructor. Hence, the rule for N.a integrates the glue grammar structurally with the component grammar, and the actual semantic rules applied on N.a are those defined in production r of the component grammar. If some element b_i in production r is a terminal symbol, then the corresponding element b'_i in the glue production must be a local terminal constant (see below). The glue production must in that case define the input attributes of the terminal constant, serving as the input interface to the corresponding terminal symbol in the component production r. Accordingly, the output attributes of the terminal symbol, representing its output interface, can be accessed in the glue grammar via the corresponding terminal constant.

Recall that the *only* form of propagation of data between the modules is expressed using the terminal symbols of the components and the corresponding terminal constants of the glue. (Syntactic) nonterminals cannot be used for this purpose, but they instead serve for defining the phrase structure of attribute evaluation in the component grammars. We use the following notation for defining a terminal constant c that is mapped with a terminal symbol t:

```
c: t = {⟨definition of input attributes⟩}
```

The concrete glue grammar for DESK is given in Algorithm 1. The concrete attribute evaluation scheme for the source program "PRINT x WHERE x = 2", as defined by the glue grammar DESK, is

Algorithm 1.

```
module DESK (Env, Code)      --glue grammar, given in terms of
                             --the components Env and Code
  Program → 'PRINT' Expression ConstPart
    {Program.symbols =                    --syntactic attribute
        init (list(ConstPart symbols,     --production constructor
              Expression.symbols)),
    Program.assembly = final(Expression assembly, STOP),
    Program.code = STOP.code,             --ordinary attribute
    STOP. end = {ok = ConstPart.ok}}      --terminal constant

Expression1 → Expression2 '+' Factor
  {Expression1 symbols = list(Expression2.symbols, Factor.symbols),
  Expression1 assembly = seq(Expression2 assembly, OP),
  OP: operand = {instr = (ADD, Factor.value), ok = Factor.ok}}

Expression → Factor
  {Expression.symbols = chain(Factor.symbols),
  Expression.assembly = single(OP),
  OP: loadable = {value = Factor value, ok = Factor ok}}

Factor → ConstName
  {Factor.symbols = ref(CREF),
  Factor value = CREF value,
  Factor ok = CREF ok,
  CREF: symbolREF = {name = ConstName.name}}

  Factor → Number
    {Factor.symbols = empty,
    Factor value = Number value,
    Factor.ok = true}

  ConstName → Id {ConstName name = Id name}

  ConstPart → ε {ConstPart.symbols = empty, ConstPart.ok = true}

  ConstPart → 'WHERE' ConstDefList
    {ConstPart.symbols = chain(ConstDefList symbols),
    ConstPart.ok = ConstDefList.ok}

  ConstDefList1 → ConstDefList2 ',' ConstDef
    {ConstDefList1.symbols = list (ConstDefList2.symbols, def(CDEF)),
     ConstDefList1.ok = ConstDefList2 ok and CDEF.ok,
     CDEF: symbolDEF = {name = ConstDef name,
                        value = ConstDef.value}}

  ConstDefList → ConstDef
    {ConstDefList.symbols = def(CDEF),
     ConstDefList.ok = true,
     CDEF. symbolDEF = {name = ConstDef name,
                        value = ConstDef value}}

  ConstDef → ConstName '=' Number
    {ConstDef.name = ConstName name,
     ConstDef value = Number value}
end DESK.
```

illustrated in Figure 7. The figure shows how the glue grammar gives rise conceptually to two abstract attributed syntax trees, one for name analysis and the other for code generation. The nodes in these trees stand for grammar symbols of the corresponding component grammars Env and Code. The association of a node to the symbol in the glue grammar is indicated by giving in parenthesis the name

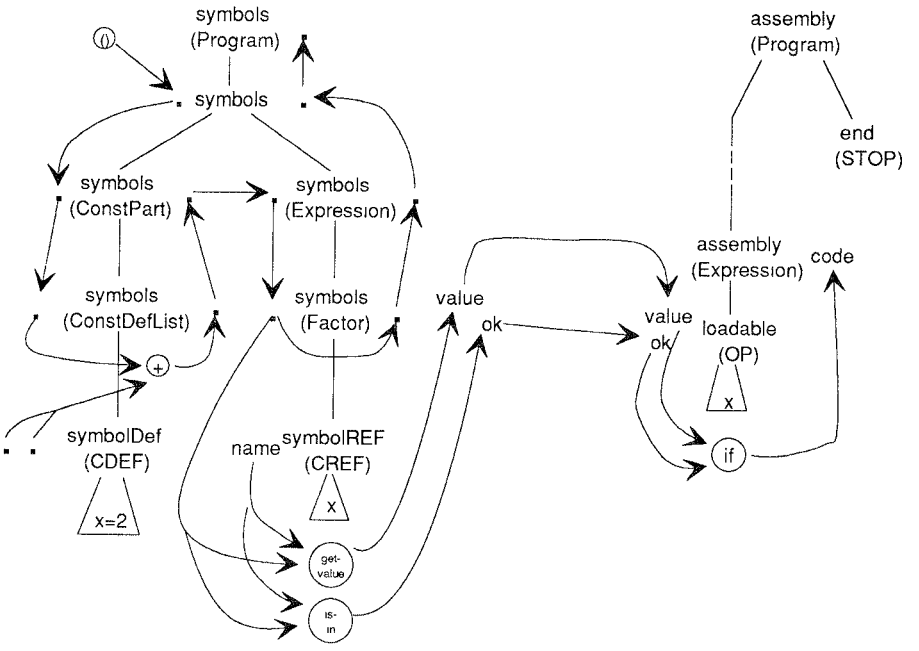


Figure 7. Tree association by modular attribution.

of the glue symbol. The tree rooted by the node with label symbols (Program) represents name analysis, and the tree rooted by the node with label assembly (Program) is for code generation. Note that a node is associated with attribute instances for both its component symbol (e.g., symbols) and its glue symbol (e.g., Program).

For simplicity, Figure 7 includes only the environment attribute instances of the nodes for symbols (envi to the left and envs to the right of a node), except for one node that illustrates the semantic coupling between the attributed trees via the instances of value and ok. The attribution of symbolDEF has been simplified. Moreover, the attributed tree for code generation is given in a reduced form that includes only the interface between the two trees. Notice that the order of the subtrees for the constant definition part and for the expression part of the source program (nodes for ConstPart and Expression, respectively) is different from the order in the glue grammar and in the source program. This facility, provided by the conceptual separation of the (abstract) component grammars and the (concrete) glue grammar, makes it possible to elegantly coerce the concrete right-to-left flow of symbol table information (see Figure 4) into the abstract left-to-right form as implemented in the component grammar Env.

2.4 Object-Oriented Attribute Grammars

In the previous section we characterized modular programming as a paradigm based on static decomposition of programs, data abstraction, information hiding, and separate compilation. *Object-oriented programming* is a refined form of the structured and modular styles and combines the concepts of a module (a purely syntactic unit) and a type (a construct that can give rise to an arbitrary number of dynamic instances) into a single concept, a *class*. Thus, a class can be employed on one hand as a program structuring tool and on the other hand as an abstract data type. The entities that are created at run-time as instances of a

class are called *objects*. The third fundamental concept is *inheritance*: a class can be defined as a generalization (superclass) or a specialization (subclass) of another class. Accordingly, the objects can be grouped into hierarchies that are induced by their superclass and subclass relations. Objects of a subclass inherit implicitly all the properties defined for objects of the superclass. Additionally, objects may have specific properties defined in the subclass. The reference mechanism is *polymorphic*: a reference of class C can denote, besides an object of class C, also an object of a subclass of C. This feature implies that in general the actual object referenced must be resolved at run-time by *dynamic binding*.

The basic concepts of object-oriented programming, in addition to those of modular programming, are thus

- classes
- inheritance
- objects, having an inner state and a set of applicable operations ("methods")
- polymorphism and dynamic binding.

These special properties support system reusability and extendibility in a more flexible way than the non-object-oriented structuring facilities discussed in Sections 2.2 and 2.3: a new system can be built from an existing one by specializing the existing classes and objects through inheritance to fit the problem domain better.

When introducing object-oriented concepts into attribute grammars, the following correspondence is one relevant framework for *object-oriented attribute grammars*:

Nonterminal = Class

With respect to static interpretation of classes, this amounts to the structured and modular language processing paradigms discussed in Sections 2.2 and 2.3: nonterminals serve as structuring tools of the attribute grammar. The essential feature, however, that makes the object-oriented style most powerful in language processing is the dynamic nature of nonterminals it promotes: a nonterminal can be considered as an abstract type (class) that specifies the properties of the dynamically created objects of that type. This view is consistent with the principle used in syntax-directed language processing where nonterminals, terminals, and productions represent a static language specification from which an underlying parser creates the corresponding dynamic objects as nodes of the parse tree.

Object-oriented language processing is appropriate for both compilation and interpretation, the two main implementation schemes of programming languages. The operations of an object can specify either compilation of the corresponding language structure into some target code or dynamic execution of the object in terms of its state. Hence, objects can be regarded as abstract operations of an underlying intermediate language that is executed by an abstract interpreter.

In the object-oriented language processing paradigm a source program is viewed as a set of interrelated and hierarchical objects that behave according to the semantics of the source language. The objects are created as instances of the nonterminal classes that specify both the syntax and the semantics of the objects. Hence, for each object there is (1) a nonterminal (class) that defines the syntactic structure of the object in terms of productions and (2) a class (nonterminal) that specifies the static and dynamic semantics of the object in terms of its attributes ("state") and operations. This pertains to an application-oriented view that is often mentioned as a profound factor in the object-oriented approach toward software engineering.

The correspondence "Nonterminal = Class" suggests that the *syntactic specification* of the source language derives implicitly the superclass/subclass hierarchy in the attribute grammar and, from a dynamic point of view, the relations between the objects generated by a source program (i.e., the "attributed derivation

tree"). In order to gurantee that the class and object hierarchies are regular enough both from a general object-oriented and from a language processing point of view, the syntactic specification should be given in a disciplined form. The idea of mapping a context-free grammar into a class hierarchy has implicitly been applied in language definitions as a rather natural tool. For instance, the production

```
compound_statement ::= if_statement |
case_statement |
loop_statement | block_statement |
accept_statement | select_statement
```

in the Ada reference manual specifies that: "if_statement **is-a** compound_statement", "case_statement **is-a** compound_statement", ..., "select_statement **is-a** compound_statement" [Department of Defense 1983]. Different forms of object-oriented context-free grammars are defined in Koskimies and Paakki [1990] and Koskimies [1991].

A conceptual correspondence between nonterminals and classes is the basis in TOOLS [Koskimies and Paakki 1990] and in AG [Grosch 1990]. The TOOLS system has been presented in Section 2.2, where structured attribute grammars were discussed. Besides a block-structured basis, the specification language of TOOLS has a number of primitives that support the object-oriented language processing methodology. For instance, nonterminals can be organized as classes into (restricted) inheritance hierarchies over attributes. The classes may also specify operations for their objects that are polymorphic within their own class hierarchy. Some of these features were applied in the example of Section 2.2.1.

A slightly broader perspective is followed in Mjølner/Orm [Hedin 1989] and in OOAG [Shinoda and Katayama 1990] that suggest the correspondence

Production = Class

A completely different view is taken in SmallYACC where a context-free grammar as a whole is subject to inheritance of productions from "superclass" grammars [Aksit et al. 1990].

2.4.1 *Mjølner/Orm*

Mjølner/Orm is a metaprogramming system based on a general object-oriented view of programs. The system maintains programs consistently as attributed abstract syntax trees, and supports the development of integrated programming environments including such program processing tools as syntax-directed editors and interpreters. An overview of the system is included in Knudsen et al. [1993].

Mjølner/Orm includes as a central subcomponent an object-oriented attribute grammar-based metalanguage for specifying and developing different kinds of language processors of a programming environment. Besides treating nonterminals as (abstract) classes, the metalanguage of Mjølner/Orm also considers each production as a class that specifies the syntactic structure of the production, the attributes of its left-hand-side nonterminal, and the associated (default) semantic rules. All these elements can be inherited, specialized, and overridden in subclasses, thus introducing *virtual properties* and *dynamic binding* into the metalanguage. The original object-oriented notation of Mjølner/Orm is presented in Hedin [1989] and its further elaboration in Hedin [1992].

Because Mjølner/Orm is designed to produce integrated environments, the context-free grammar for a language is given in terms of its abstract syntax. In order to process concrete programs, a special interface must be separately built between the concrete and the abstract representation. Each production is expressed as a class that can be one of the following kinds:

(1) An *abstract class* models an entity whose syntactic structure is not fixed and that usually represents general semantic properties.

(2) A *structured class* represents an entity with a particular syntactic outlook.

(3) A *case class* represents an entity whose syntactic structure is inherited as is but whose semantics can be refined.

A structured class specifies a production's right-hand side either as a sequence of named symbols, as a symbol list, or as a lexical element ("lexeme"). In addition to the conventional inherited (*ancestral* in the Mjølner/Orm terminology) and synthesized attributes local attributes can be associated with a production. They cannot be accessed from outside of the production class.

The production classes are defined using the following notation. P denotes the name of the class, and S its optional superclass. A sequence of attribute declarations and semantic rules is denoted briefly by *Sem*.

⟨P⟩ · ⟨S⟩ · . = **Abstract** *Sem*	(abstract class)
⟨P⟩ : ⟨S⟩ . . = ⟨C⟩* *Sem*	(structured list of symbols C)
⟨P⟩ : ⟨S⟩ : : = {⟨t1 · C1⟩ & . . . & ⟨tn : Cn⟩} *Sem*	(structured sequence of symbols t1 (of class C1), . . . , tn (of class Cn))
⟨P⟩ : ⟨S⟩ : : = **Lexeme** *Sem*	(structured lexeme class)
⟨P⟩ : ⟨S⟩ · · = **Case** *Sem*	(case class)

In the conventional notation of context-free grammars, these correspond to the following productions, respectively:

P → P_1 | P_2 | . . .
P → C*
P → C1 . . . Cn
P → *some lexical definition*
P → *same syntax as defined for* S

Hence, an abstract class models a group of related language entities $P_1, P_2, \ldots$ (such as a compound_statement of Ada; see above), and lexeme classes stand for terminal symbols. Case classes are not given any syntactic structure because they are implicitly defined by the superclass S.

As an example of the approach, we give an object-oriented attribute grammar for DESK in the Mjølner/Orm metalanguage in Algorithm 2. Inherited attributes are denoted with **Anc** and synthesized attributes with **Syn**. Note that the underlying context-free grammar is abstract, and therefore the delimiters and operators of DESK are not specified.

The abstract classes in Algorithm 2, Node, Root, Descendant, Expression, and ConstPart, are semantic superclasses that do not give rise to explicit objects in the syntax tree. They are instead used for specifying general properties of tree nodes, most notably the common attributes and their default semantic rules. These general properties are refined, when needed, in the structured and case subclasses Program, IdUse, Number, BinOp, Addition, EmptyConstPart, ConstDefList, and ConstDef specifying the actual objects as tree nodes.

The class hierarchy of nonterminals as specified by the grammar is depicted in Figure 8 with subclasses drawn below their superclass. The classes are associated with their final set of attributes, obtained after applying the inheritance mechanism. The figure also shows the border between the abstract classes and their structured and case subclasses. Notice that an abstract class must have at least one structured subclass in order to be useful for the implementation. The flexibility granted by inheritance is the introduction of attributes and semantic rules at any level of the class hierarchy. Default behavior is typically specified in the abstract classes and redefined in their subclasses, if necessary. For instance, the multiple semantic rule "**for all sons** (X) **in** Descendant **son** (X).envi:= ( )" for the class Root states that all the Descendant sons of a Root object node in an abstract syntax tree will get an empty tables as the default value of their envi attribute. This default rule is partially overridden in the class Program that specifies that the envi attribute of an Expression object node (with Descendant as superclass)

Algorithm 2.

```
⟨Node⟩ : := Abstract,      - -general node of an abstract syntax tree

⟨Root⟩ . ⟨Node⟩ : := Abstract      - -abstract root node
  for all sons(X) in Descendant son(X).envi : = ( ),

⟨Descendant⟩ : ⟨Node⟩ : := Abstract      - -abstract interior node
  Anc envi: SymbolTable; Syn ok· Boolean;
  for all sons(X) in Descendant son(X).envi . = envi;
  ok : = true,

⟨Program⟩  ⟨Root⟩ : := {⟨e. Expression⟩ & ⟨cp  ConstPart⟩}
                          - -structured class for production
                          - -Program → Expression ConstPart
  Syn targetCode· InstrList,
  e.envi · = cp.envs;
  targetCode : = if cp.ok then e.code + (PRINT, 0) + (HALT, 0)
                 else (HALT, 0);

⟨Expression⟩ : ⟨Descendant⟩ : := Abstract
  Syn code: InstrList; Syn value: Integer;
  code : = (HALT, 0);
  value : = 0;

⟨IdUse⟩ : ⟨Expression⟩ : := Lexeme
  ok  = isin(string, envi),
  value : = getvalue(string, envi),
  code : = if ok then (LOAD, value) else (HALT, 0);

⟨Number⟩ : ⟨Expression⟩ : := Lexeme
  value : = StrToInt(string);
  code : = (LOAD, value),

⟨BinOp⟩ : ⟨Expression⟩ : :=
          {⟨left: Expression⟩ & ⟨right: Expression⟩};

⟨Addition⟩ : ⟨BinOp⟩ : := Case
  code : = if right.ok then left.code + (ADD, right.value)
           else (HALT, 0);

⟨ConstPart⟩ : ⟨Descendant⟩ : := Abstract
  Syn envs: SymbolTable,
  envs : = ( ),

⟨EmptyConstPart⟩ : ⟨ConstPart⟩ : := { };

⟨ConstDefList⟩ : ⟨ConstPart⟩ . . = {⟨cp  ConstPart⟩ & ⟨cd: ConstDef⟩}
  ok : = cp.ok and not isin(cd.name, cp.envs);
  envs : = cp.envs + (cd.name, cd.value);

⟨ConstDef⟩ : ⟨Node⟩ : := {⟨i: Id⟩ & ⟨n: Number⟩}
  Syn name: String; Syn value: Integer;
  name : = i.string,
  value : = n.value;

⟨Id⟩ . : = Lexeme,
```

shall not have the empty table as its value but instead a copy of the synthesized envs instance of the sibling ConstPart node.

In the object-oriented attribute grammar above we have used the intrinsic attribute string that gives the lexical contents of a terminal symbol, that is, of an object of a lexeme class. StrToInt is a predefined function that returns the integer value of a character string consisting of numerals. Note that since productions

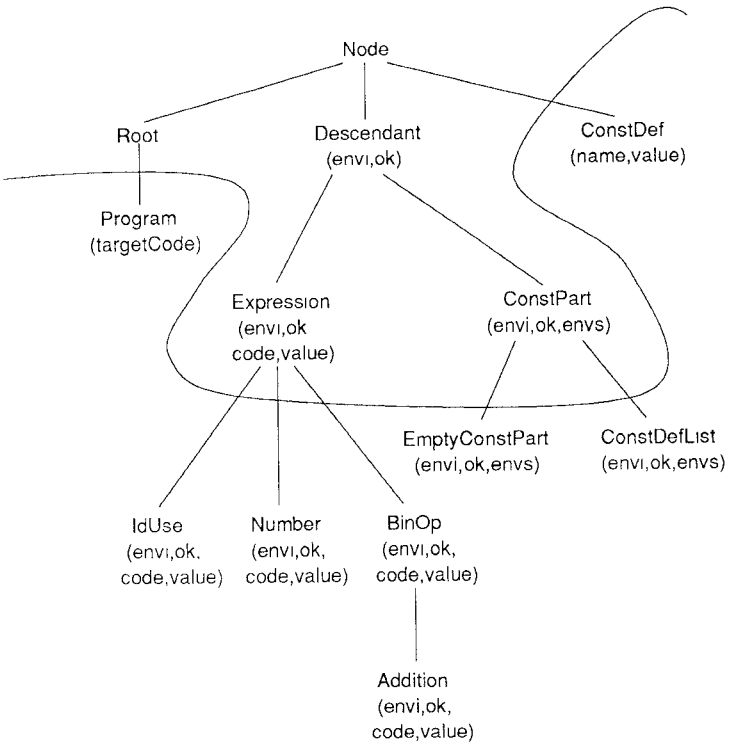


Figure 8. Class hierarchy of productions.

are represented as (named) classes in Mjølner/Orm, the alternative productions must be uniquely named (e.g., ⟨EmptyConstPart⟩). This is a central principle of object-oriented context-free grammars in general.

The example illustrates some important aspects of object-oriented attribute grammars. Inheritance of attributes and semantic rules makes object-oriented attribute grammars shorter than their standard counterparts. Having general semantic properties centralized in one place (in an abstract class) makes it easier to understand and maintain the specifications. Polymorphism complements the classification and inheritance framework by making it possible for an object of class C to apply properties implemented in the subclasses of C. This is shown in our example, e.g., by the semantic rule

code . = **if** right.ok **then** *left.code*
+(ADD, right.value) **else** (HALT, 0)

specified for the class Addition. Since left is of class Expression, left.code is allowed to stand for a code attribute defined in any of the subclasses of Expression: IdUse, Number, BinOp (applies the default of class Expression), or Addition. The selection of the applied rule for *left.code* is made by dynamic binding, on the basis of the actual class of the object that *left* represents at run-time of the DESK compiler. Thus, the dynamic binding property introduces special semantic power into the attribute evaluation process similar to the primitives to be discussed in Section 3. We have, however, classified the formalism as an organizational paradigm rather than an evaluation paradigm because most of the central features of object-oriented attribute grammars have their merits on the organizational side.

The fusion of attribute inheritance and class inheritance in Mjølner/Orm causes some unexpected problems. Since a

structured class C with structure {⟨a: A⟩ & ⟨b: B⟩} only knows the attributes of the (static) classes A and B and *not* the attributes of their (dynamic) subclasses (since these depend on the shape of the syntax tree), one cannot ensure that all the inherited attributes of the dynamic subclasses have a defining rule in the class C. To prevent such problems with well-definedness, dynamic subclasses are not allowed to have inherited attributes at all. The same reason rules out multiple inheritance as well: a class (production) may have only one immediate superclass (production) [Hedin 1989].

Another object-oriented enhancement in Mjølner/Orm is to allow references to mutable objects as a domain of attributes. This mechanism has been developed especially for flexible attribute reevaluation in incremental semantic analysis. An account of the developed class of *door attribute grammars* is given in Hedin [1992; 1994].

3. EVALUATION PARADIGMS

In Section 2 we have presented techniques for organizing attribute grammars into independent and coherent units. The organizational paradigms contribute mostly to managing the complexity of large specifications, whereas the objective of this section is to concentrate on the attribute evaluation process and discuss how its power can be raised with special mechanisms. The presented formalisms are more expressive than the standard notion of attribute grammars. Unlike with the organizational paradigms, a specification written in any of these styles cannot be trivially transformed into an equivalent standard attribute grammar.

The semantic power of the paradigms is due to the use of special computational primitives of logic and functional programming in attribute evaluation. These mechanisms make it possible to accept a restricted form of temporarily *undefined attribute values*, a feature which is considered formidable in standard attribute grammars based on strict semantic functions.

The evaluation paradigms can be related to standard attribute grammars by using characteristic dependency graphs as the common denominator. Most notably, extensions of (one-pass) *L*-attributed grammars and well-defined attribute grammars are addressed in Section 3.1 on logic attribute grammars and in Section 3.2 on functional attribute grammars, respectively.

3.1 Logic Attribute Grammars

In declarative programming, the emphasis is laid on expressing the *problem* abstractly rather than on expressing an explicit *solution* to that problem, as is the case with imperative languages. *Logic programming* is a declarative paradigm founded on the mathematical system of predicate calculus. A logic program is given as a set of *clauses* (also called *rules*) over *relations* between objects of the problem domain. The clauses correspond to formulas of first-order logic with implication as the major logical connective and with at most one atom as the conclusion. Hence, logic programming is not based on full mathematical logic but instead on the computable Horn subset of first-order logic.

Some features making logic programming an established paradigm are

- relations
- nondeterminism
- logical variables
- unification.

Relations (in logic programming usually called *predicates*) treat arguments and results uniformly; that is, there is no explicit division of arguments into input and output ones. *Unification* is an equation-solving algorithm which aims at making its two arguments syntactically equivalent by generating suitable bindings on variables in them. Bindings can be made both on the arguments of f and on the arguments of g when resolving an equation $f(\ldots) = g(\ldots)$. Hence, unification is more general that its directed counterpart in the functional program-

ming side: pattern matching (see Section 3.2).

Closely coupled with unification is the concept of a *logical variable*, which makes it possible to process provisionally incomplete values that will be completed later during program execution, and to build equivalence classes over uninstantiated variables. When one variable in the equivalence class becomes instantiated, all the other variables in the class will be simultaneously instantiated with the same value.

During the execution of a logic program the choices of the next goal to be solved and the next clause to be applied are made nondeterministically among the potential alternatives. Ths mechanism is usually simulated in the implementations by *backtracking*.

While the logic programming paradigm and the attribute grammar formalism may at first glance seem to be quite different, they both share a common structural representation: the *proof tree* in logic programming and the *attributed tree* in attribute grammars, which provides for a rigorous framework for relating these two. Indeed, the relationship between logic programs and attribute grammars has been analyzed to an extent which makes it possible to transform logic programs into semantically equivalent attribute grammars, and vice versa [Maluszynski 1991; Deransart and Maluszynski 1985; 1993]. Building such a close relationship is based on the correspondence

Nonterminal = Predicate

which is the obvious result from unifying proof trees and attributed derivation trees. This characterization also directly promotes the correspondences "Production = Clause", "Attribute value = Term", and "Semantic function = Predicate", which together provide the methodological basis for *logic attribute grammars*.

In addition to attribute grammars, the logic programming paradigm has been related with other semantic formalisms, such as two-level grammars [Maluszynski 1982] and denotational and axiomatic semantics [Bryant and Pan 1989]. This flexibility is based on the "two-level" facilities of logic programming: resolution over predicates can be applied in specifying a language's syntax and unification over terms in specifying its semantics.

The integration of attribute grammars and logic programming can also be founded on some principle other than the "Nonterminal = Predicate" correspondence. In Sataluri [1988] a concept of "relational attribute grammars" is presented that is based on the guideline implicit already in the preceding discussion:

Semantic rule = Horn clause

In this approach an attribute can flexibly be considered as inherited or synthesized depending on the shape of the derivation tree.[6] Another generalization is that circular attribute grammars can be evaluated to some extent since the semantic rules written as Horn clauses accept nonstrict arguments. As a third enhancement, relational attribute grammars may in principle express, besides compilation from language A into language B, also the reversed compilation from B into A, thanks to the flexibility of relations and their arguments [Arbab 1986]. This principal possibility of implicitly inverting attribute grammars is, however, limited in practice because of the impure features that are usually included in logic programs. Hence, an explicit attribute grammar inversion may still be necessary [Yellin 1988].

Pan is an incremental editing system where logic attribute grammars ("logical constraint grammars") can be applied for specifying the static semantics of a programming language [Ballance et al. 1990; Ballance and Graham 1991]. While the relational attribute grammar style makes

[6] Note, however, that such generality is in contrast with the consistent inherited/synthesized principle of attributes and makes it hard to verify statically the validity of the grammar.

full use of the special features of logic programming, it is interesting to note that the pragmatic development of Pan has made it necessary both to restrict the usage of the features and to modify their operational model. For instance, "code" and "data" are separated, and only ground (completely instantiated) attribute values are accepted.

3.1.1 Definite Clause Grammars

The "Nonterminal = Predicate" correspondence has been adopted in a number of syntactic tools built on top of logic programming. The most well-known logic grammar formalism is the *definite clause grammars* (DCGs) that were originally introduced for specifying and parsing natural languages [Pereira and Warren 1980]. Other formalisms of similar kind are, for example, metamorphosis grammars [Colmerauer 1978], gapping grammars [Dahl and Abramson 1984], and definite clause translation grammars [Abramson 1984]. A survey of logic grammars is given in Dahl and Abramson [1989].

DCGs are a logic counterpart to context-free grammars such that nonterminals can be augmented with arguments, and productions can be augmented with arbitrary code in Prolog. When considering the arguments as "attribute values" and the embedded Prolog code as "semantic rules," DCGs match closely with attribute grammars. DCGs are useful because they are provided in many Prolog implementations. These systems include as a requisite a DCG preprocessor that translates the grammar part of the program into ordinary Prolog. The generated program acts as a parser-driven processor for the language specified in the DCG and can be directly executed by the Prolog system. Note that the underlying linear execution model of Prolog makes DCGs operationally close to *L*-attributed grammars (see Section 1.3).

The DCG productions (rules) are written in the form

$$n(a_1, \ldots, a_i) \rightarrow n_1(b_1, \ldots, b_j), \ldots, n_m(c_1, \ldots, c_k)$$

where $n, n_1, \ldots, n_m$ are predicates, and $a_1, \ldots, a_i, b_1, \ldots, b_j, c_1, \ldots, c_k$ $(i \geq 0, j \geq 0, k \geq 0)$ are terms. This corresponds to the attribute grammar production

$$n \rightarrow n_1 \ldots n_m\{\ldots\}$$

such that the argument terms represent attribute values of the associated nonterminals. A terminal symbol t with attribute value a is represented by the unary list $[t(a)]$, and the empty string is represented by an empty list. Arbitrary Prolog code can be included in the rule's right-hand side by enclosing it in curly braces $\{\ldots\}$.

We give an example of the DCG style in the next section where the PROFIT system is presented.

3.1.2 PROFIT

PROFIT is a language and system designed for compiler writing in a logical style. PROFIT integrates logic programming and attribute grammars in a simple manner, by extending Prolog with some facilities that are necessary in the application area. The most notable facilities are deterministic error-recovering DCGs and functions [Paakki and Toppola 1990; Paakki 1991].

Conceptually, PROFIT is based on *logical one-pass attribute grammars*, a proper superset of *L*-attributed grammars [Paakki 1990]. The semantic power of logical one-pass attribute grammars is granted by the correspondence

Attribute = Logical variable

that makes it possible to delay certain attribute evaluations. The idea is to allow the value of an attribute instance to be undefined as long as it is not needed in making *control decisions* in the attribute evaluation process. The formalism restricts the binding of undefined values to nonstrict argument positions only and guarantees that the conventional *L*-attributed evaluation strategy [Lewis et al. 1974; Bochmann 1976] is

always able to compute the meaning of a source program.[7]

PROFIT is implemented by translation into ordinary Prolog. The DCG facility is based on a conventional error-recovering recursive-descent execution of LL(1) context-free grammars, and the functions are transformed into predicates. The functions of PROFIT are written as conditional equations of the form

$$f(f_1, \ldots, f_n) = t \quad -g_1, \ldots, g_m.$$

where f is the function's name; the f_i are its formal parameters; t is the result value (Prolog term), and the g_i are guards (Prolog predicates). When executed, this function will return the value t, provided that the formal parameter list unifies with the actual parameter list and that all the guards succeed. Several declarations can be given for f in which case that succeeding declaration which is given first defines the result. A function must never fail to produce a value; in such a case execution is aborted.

Function applications are expressed as *functional terms* of the form $@f(a_1, \ldots, a_n)$ where f is the function's name, and the a_i are its actual parameters. Functional terms given in the right-hand side of a rule (DCG production) model inherited attributes and are called *inherited functional terms*, whereas those given in the left-hand side model synthesized attributes and are accordingly called *synthesized functional terms*.

The operational behavior of inherited and synthesized attributes with an *L*-attributed evaluation scheme is reflected in the functional terms as follows:

(1) Let a predicate (i.e., a nonterminal) p be associated with an inherited functional term:

$$r(\ldots) \rightarrow \ldots p(\ldots, @f(a_1, \ldots, a_n), \ldots) \ldots$$

When execution reaches p, the term $@f(a_1, \ldots, a_n)$ is first evaluated by calling the function f with arguments $a_1, \ldots, a_n$. Then p is entered with the value returned by f replacing the functional term.

(2) Let a predicate q be associated with a synthesized functional term:

$$q(\ldots, @g(b_1, \ldots, b_m), \ldots) \rightarrow \ldots$$

When entering q (and unifying its head with the current goal), the term $@g(b_1, \ldots, b_m)$ is replaced by an augmented variable V. The evaluation of $@g(b_1, \ldots, b_m)$ is suspended until the body of q has been successfully executed. Then g is called with arguments $b_1, \ldots, b_m$ (having their current values), and the returned value is unified with V.

Successive and nested functional terms are allowed and evaluated in a left-to-right depth-first manner.

As an example of the semantic power of logical one-pass attribute grammars, we give the DESK attribute grammar in PROFIT. To simplify the discussion, the ok attribute is omitted. Now the context-free grammar that corresponds to the DCG is right-recursive and not left-recursive because PROFIT accepts only attribute grammars whose underlying context-free grammar is LL(1) (see Aho et al. [1986]). The constructor op / 2 represents the assembly operations, and the constructor sym / 2 represents symbol table entries. Notice that the attributes are not explicitly named.

```
program ([Code, op(print, 0), op(halt, 0)] →
    ['PRINT'], expression(Env, Code), constpart
    (Env, Env).

expression (Env, [op(load, Value) | Code]) →
    factor (Env, Value), expressionrest
    (Env, Code).

expressionrest (Env, [op(add, Value) | Code]) →
    ['+'], factor (Env, Value), expressionrest
    (Env, Code).

expressionrest (Env, [ ]) → [ ].

factor (Env, @getvalue (Name, Env)) →
    constname(Name).
```

[7] Note that the logical variable mechanism may leave some attribute values temporarily undefined during evaluation. The formalism, however, guarantees that when finishing the one-pass evaluation process, all the synthesized attribute instances associated with the root of the parse tree have a defined (completely ground) value

```
factor (Env, Value) → [number (Value)].
constname (Name) → [id(Name)].
constpart (Envi, Envs) → ['WHERE'],
constdeflist (Envi, Envs).
constpart (Env, Env) → [ ].
constdeflist (Envi, Envs) →
  constdef (Name, Value),
  constdefrest
(@define(Name, Value, Envi), Envs).
constdefrest (Envi, Envs) →
  [', '], constdef (Name, Value),
  constdefrest(@define(Name, Value, Envi),
  Envs).
constdefrest (Env, Env) → [ ].
constdef (Name, Value) →
  constname(Name), ['='], [number(Value)].
—semantic functions and predicates:
getvalue(Name, Env) =
Value :-lookup(Name, Env, Value).
define(Name, Value, Env) =
Env :-lookup(Name, Env, Value).
lookup(Key, [sym(Key, Value) | Env], Value).
lookup(Key, [sym(Key1, Value1) | Env], Value) :-
  Key ≠ Key1, lookup(Key, Env, Value).
```

The attributes are associated with the nonterminals of this PROFIT-DCG as follows, where i denotes an inherited attribute and s a synthesized attribute:

program(s)
expression(i, s)
expressionrest(i, s)
factor(i, s)
constname(s)
constpart(i, s)
constdeflist(i, s)
constdefrest(i, s)
constdef(s, s)

The semantic functions getvalue and define are strict in their first argument that gives the name of the referenced constant. The functions are nonstrict in all the other arguments; that is, those arguments can be uninstantiated (undefined) when calling the function. In addition to these explicitly defined functions, we have made use of the implicit identity function whose argument is nonstrict, the list construction function [ ] whose arguments are nonstrict, a few strict constant functions, the lookup predicate, and the built-in predicate $\neq$.

As already concluded in Example 1.3.1, the DESK attribute grammar is not L-attributed, due to right-to-left dependencies for attribute occurrences Env in the first production. This implies that no strict one-pass left-to-right evaluation method is capable of generating the target code for a DESK program. The grammar is, however, *logically one-pass* since the non-L-attributed attributes (the first argument of expression and constpart) are always applied nonstrictly in the semantic functions. The logic-based strategy of PROFIT succeeds therefore in evaluating all the attributes in a top-down left-to-right manner over the (implicit) attributed tree for a DESK program. In conclusion, the attribute grammar can be evaluated using a much simpler one-pass strategy than when employing a multi-pass system, such as those presented in Sections 1 and 2.

One central solution is to present the symbol table (the attribute values Env, Envi, Envs) as an incomplete data structure. Now it is possible to insert constants with a defined name and an undefined value into the symbol table while processing the expression part of the source program, and to complete the entries with value information in the subsequent constant definition part. Note that such a facility, based on the delayed binding property of logical variables, introduces a synthesized flavor to the formally inherited first attribute of the expression part. Both symbol table insertions and lookups can be implemented using the same lookup predicate over the incomplete data structure. This elegant solution has originally been suggested in Warren [1980].

The target code for a DESK expression may also be incomplete during attribute evaluation. The PROFIT evaluator, however, binds the undefined operands with the (incomplete) entries in the symbol table. These operands will be implicitly instantiated while completing the symbol table during the processing of the con-

stant definition part of the source program. Thus, the final target code will be completely defined.[8] A similar delayed one-pass evaluation effect as provided here implicitly by the logical variables has been proposed in Noll and Vogler [1994] where unevaluated attributes are maintained and resolved in an explicit graph.

As an example of the logical one-pass attribute evaluation process, consider the following DESK program:

```
PRINT x + y + x WHERE x = 1, y = 2
```

When the compiler has reached the end of the expression x + y + x, the symbol table is in the form

```
[sym(x, V1), sym(y, V2) | Env]
```

Here V1, V2, and Env are uninstantiated variables (that is, undefined attribute values). The target code has the value

```
[op(load, V1), op(add, V2), op(add, V1)]
```

Here the second operands of the instructions are bound with the corresponding uninstantiated variables in the symbol table. After processing the whole program, the symbol table has obtained its final value:

```
[sym(x, 1), sym(y, 2) | Env]
```

Thanks to the logical variable facility, the target code will be simultaneously completed into the following form, without having to employ any explicit backpatching operations:

```
[[op(load, 1), op(add, 2), op(add, 1)],
 op(print, 0), op(halt, 0)]
```

3.2 Functional Attribute Grammars

In the previous section we discussed the integration of logic programming and attribute grammars. Besides logic programming, *functional programming* is a paradigm dominating the declarative scene. Functional programming is based on the mathematical model of expressing computation with functions. A function f: $D \rightarrow R$ expresses a rule that maps the elements of the *domain* D to the elements of the *range* R. The evolution of the functional programming paradigm has produced a variety of concepts and languages for expressing such functions. The functional style of programming can be summarized with the folloing characteristics:

- referential transparency
- higher-order functions
- lazy evaluation
- pattern matching

Referential transparency is the term used traditionally to express that the value of an expression depends only on the values of its subexpressions, and that the value of each occurrence of an object is the same, irrespective of the context. In other words, *side effects* are ruled out. Functions being *higher-order* means that they are first-class values and can, for instance, be stored in data structures and returned as result of function calls. *Lazy evaluation* is tantamount to "nonstrictness," meaning that the value of an object is not computed until it is actually needed, and thus the value of an argument may be undefined when calling and executing the function. *Pattern matching* is a general term used to express the mechanism applied when trying to make X and Y identical in an equation $X = Y$ where X and Y are expressions. Pattern matching is restricted in functional programming to introduce bindings only to the left-hand side X when solving such an equation. (Recall from Section 3.1 that the more general unification primitive of logic programming may instantiate Y as well.

The referential-transparency property is characteristic of a number of metalanguages based on attribute grammars. For instance the metalanguage of GAG is, in its original form, free of side effects and

[8] Note that this holds for valid source programs only since undefined constants leave their corresponding operands in the target code uninstantiated This aspect could be improved simply by reintroducing the ok attribute

in that respect of a functional nature [Kastens et al. 1982]. Usually the referentially transparent metalanguages, however, do not provide the other central features of functional programming.

Actually the special mechanisms of functional programming have not been exploited very extensively in attribute grammars (which is a bit surprising since declarative functions are a main concept in attribute grammars), but rather in denotational semantics. Since denotational semantics-directed formal language definition and implementation is an approach based on *valuation functions* (mapping the syntactic constructs of a language to their semantic meanings), the central concepts of the functional programming paradigm fit well with the approach. A more detailed discussion of functional programming exclusively within denotational semantics is beyond the scope of this survey. Such analysis is included in, e.g., Watt [1984] and Lee [1989].

Defining the semantics of a source language using functions makes it possible to relate attribute grammars with denotational semantics. Chirica and Martin [1979], Ganzinger [1980], Mayoh [1981], and Courcelle and Franchi-Zannettacci [1982], among others, discuss methods for relating an attribute grammar with an equivalent denotational system of recursive functional equations. These formulations are based on the principle of modeling each synthesized attribute of an attribute grammar by a function which maps the underlying derivation tree and the values of the inherited attributes of the associated nonterminal onto the semantic value of the tree. Hence, the correspondence

Synthesized attribute = Function

is followed in these mathematical approaches. The equation system may require a least-fixpoint computation to be applied in attribute evaluation and may thus be rather laborious. On the other hand, this general method makes it possible to evaluate even circular attribute grammars in case the derived recursive formulas have a computable least-fixpoint solution.

Because relating attribute grammars and denotational semantics is based on functional attributes, the theoretical guidelines mentioned above can be applied in practice by including higher-order functions into an attribute grammar-based metalanguage. Then the semantic functions of an attribute grammar can be written in a denotational style, mapping functional attribute values into functional attribute values. Such an amalgamation of these two language definition formalisms provides means to express the "total" semantics of a programming language in a unified manner. The attribute grammar concepts of the metalanguage can be exploited in defining the static semantics of the language, while the denotational concepts are better suited for defining the dynamic semantics. Such mixed formalisms with an automatic implementation are proposed in, e.g., Paulson [1982], Ganzinger et al. [1982], and Johnsson [1987] that also discuss in more detail the merits of higher-order functions in attribute grammars. The concept of higher-order functions is loosely applied in *higher-order attribute grammars* where parts of the attributed tree can be defined in terms of attributes and vice versa [Vogt et al. 1989].

When sticking solely to the attribute grammar formalism, the most obvious way to make use of the functional programming paradigm is to express functionally the semantic functions of an attribute grammar. Hence the proposition of

writing the semantic functions in a functional language

results in a formalism that can be characterized as *functional attribute grammars*. As a concrete example of the advantages of such grammars, we will show in Section 3.2.1 how lazy evaluation

can be exploited to extend the concept of well-defined attribute grammars from their original meaning to accept a restricted form of circular attribute dependencies. This enhancement will be further elaborated in Section 3.2.2 where a more powerful fixpoint evaluation technique is presented.

While the power of functional attribute grammars has not been widely applied in general systems, several functional versions of the one-pass system YACC have been developed. In these tools the underlying implementation and programming language *C* of standard YACC has been replaced by a functional language. Such functional parser generators are presented in, e.g., Peyton Jones [1985] (with Sasl as the base language), in Jones [1986] and Longley [1987] (Miranda), in Cousineau and Huet [1990] (a dialect of ML), and in Appel and MacQueen [1991] (standard ML). A more general view is to consider attribute grammars as a special class of functional programming languages, as proposed in Frost [1992; 1993].

3.2.1 Lazy Evaluation (FNC / ERN)

The standard definition considers circular attribute grammars to be ill defined [Knuth 1968], and therefore such grammars are rejected by a vast majority of language processor generators. Recall that an attribute grammar is circular, if it generates an attributed tree where an attribute instance depends transitively on itself.

Forbidding circular attribute dependencies exclusively is inconvenient because many common language processing tasks could quite naturally be expressed as a simple circular attribute grammar. As a demonstrating example, let us revise the DESK language such that the value of a constant can be defined symbolically as well. Moreover, the constant definitions can be given in any order. Thus, the following is a program in the extended language, in the sequel referred to as DESK + :

```
PRINT x + y + 1 WHERE x = y, y = z, z = 2(3.1)
```

The symbolic value of a constant may be defined in terms of a single constant only. This restriction is made for simplifying the discussion. Of course, a realistic language would accept general expressions, but such an extension would not bring any additional nuances into our discussion of circularity. Recursively or mutually defined constants are not accepted in DESK + . For instance, the following program is illegal:

```
PRINT x + y + 1 WHERE x = y, y = x
```

An attribute grammar for DESK + is given in Algorithm 3. Since a particular metalanguage is not essential for our theme, we use the same pure notation as in Example 1.3.1. Much of the original DESK grammar remains unaffected by the generalization of allowing symbolically defined constants. The only notable reformulation is that the symbol table is also needed for processing the constant definition part of a DESK + program, since now the constant values have to be available even there. This feature is expressed using the inherited attribute envi that is propagated to the constant definition part. To select a literal value or a symbolically defined value for a constant, we make use of an additional synthesized attribute isliteral to denote whether or not a value is literal. The semantic function *undefined* represents an undefined (and useless) attribute value. For simplicity of discussion, the ok attribute is omitted for the constant definition part. Consequently, invalidity of the program is manifested by abortion of the executed code with a premature HALT instruction. The crucial semantic rules of the grammar are emphasized.

By constructing the characteristic dependency graphs, the attribute grammar can be shown to be circular and thus ill defined in the original sense; for instance, the grammar is not absolutely noncircular.[9] To verify this, Figure 9

[9] The grammar could be transformed into a noncircular form (e.g., by introducing additional symbol table attributes), but then the specification would become far less declarative.

Algorithm 3.

```
Program → 'PRINT' Expression ConstPart
  {Program.code = Expression.code + (PRINT, 0) + (HALT, 0),
   Expression.envi = ConstPart.envs,
   ConstPart.envi = ConstPart.envs}

Expression1 → Expression2 '+' Factor      {- -as in Example 1.3.1 - -}

Expression → Factor                       {- -as in Example 1.3.1 - -}

Factor → ConstName                        {- -as in Example 1.3.1 - -}

Factor → Number                           {- -as in Example 1.3.1 - -}

ConstName → Id                            {- -as in Example 1.3.1 - -}

ConstPart → ε      {ConstPart.envs = ( )}

ConstPart → 'WHERE' ConstDefList
  {ConstPart.envs = ConstDefList.envs,
   ConstDefList.envi = ConstPart.envi}

ConstDefList1 → ConstDefList2 ',' ConstDef
  {ConstDefList1.envs = ConstDefList2.envs +
                        (ConstDef.name, ConstDef.value),
   ConstDefList2.envi = ConstDefList1.envi,
   ConstDef.envi = ConstDefList1.envi}

ConstDefList → ConstDef
  {ConstDefList.envs = (ConstDef.name, ConstDef.value),
   ConstDef.envi = ConstDefList.envi)

ConstDef → ConstName '=' Value
  {ConstDef.name = ConstName.name,
   ConstDef.value = if Value.isliteral then Value.value
                    else getvalue(Value.name, ConstDef.envi)}

Value → Number
  {Value.isliteral = true, Value.value = Number.value,
  Value.name = undefined}

Value → ConstName
  {Value.isliteral = false, Value.value = undefined,
   Value name = ConstName.name}
```

sketches the attribute dependency graph for the following DESK + program:

PRINT x + y + 1 WHERE x = y, y = 2

The cyclic dependency paths are indicated by arrows with a solid head.

Let us consider the DESK + grammar as a functional attribute grammar; that is, the semantic functions (including the identity function) are assumed to be implemented in a functional language with a lazy "call-by-need" evaluation strategy. As a consequence, only those attribute instances will be evaluated that are actually needed by the compiler to generate the target code for the source program. Now the termination problem with recursive evaluation of circular attribute grammars becomes less severe.

Assume that the DESK + program does not contain any symbolically defined constants. In that case all the isliteral attribute instances associated with the Value nodes will have the value *true*, and the compiler does not execute any *getvalue* function call associated with the production "ConstDef → ConstName '=' Value" in Algorithm 3. Therefore, none of the useless envi attribute instances for the ConstPart subtree have to be evaluated. Hence, the spurious *static* cycle in the attribute dependency graph will be *dynamically* removed by the lazy evaluation mechanism (cf., Figure 9).

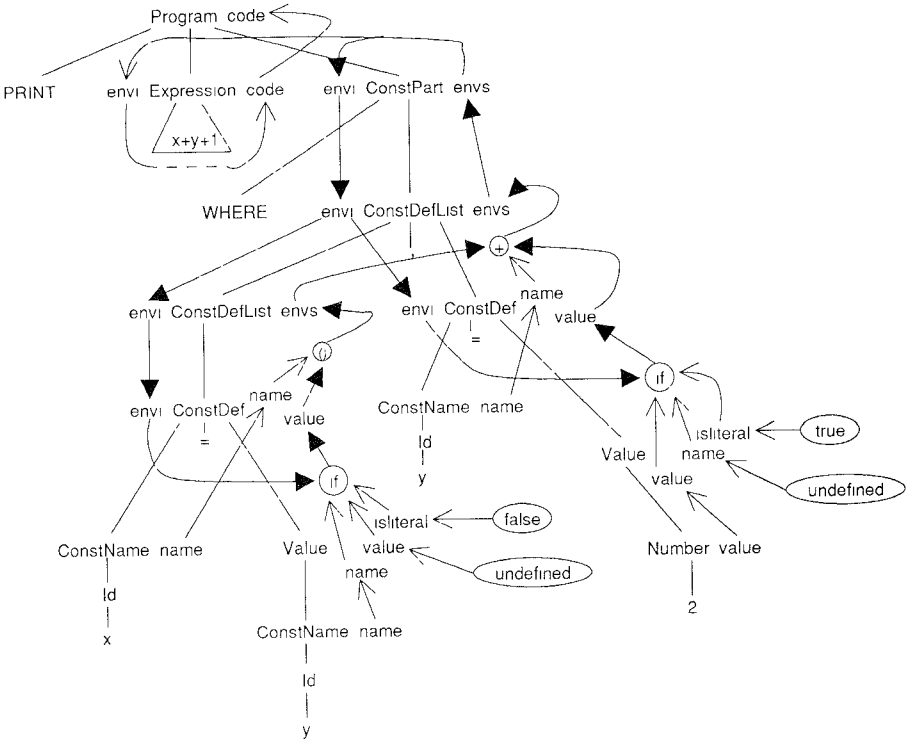


Figure 9. Circular attribute dependency graph

The solution is only partial, though. While the lazy strategy is able to compile the following DESK + program correctly

PRINT x + y + 1 WHERE x = 1, y = 2 (3.2)

it does not terminate, e.g., with the program (3.1), because symbolically defined constants induce a true cycle on the attributed tree (cf., Figure 9). Notice, however, that a conventional strict attribute evaluation strategy would loop even for the DESK + program (3.2) since in that case *all* the attribute instances (including the circularly defined envi instances) would have to be evaluated, whether actually needed or not.

Functional attribute grammars are implemented in some systems. The lazy attribute evaluation method described above is provided, for example, in FNC/ERN where the language for the semantic functions is Lisp [Jourdan 1984]. The system implements a functional attribute grammar by translating each synthesized attribute into a function that takes a syntactic subtree and the values of some inherited attributes as arguments and that returns the value of the synthesized attribute as result. Hence, the implementation follows the "Synthesized attribute = Function" principle as well. The FNC component of the system implements the class of absolutely noncircular attribute grammars that excludes statically circular attribute grammars, such as the DESK + grammar above. However, the ERN component provides for a dynamically noncircular evaluation scheme that behaves in the manner discussed above for DESK + programs with literal constants only.

A different method of lazy attribute evaluation is implemented in Elegant [Augusteijn 1990]. The system accepts "pseudocircular" attribute grammars

where the circular dependency chains must be explicitly broken by the user with an additional semantic rule for one of the attributes within each cycle. This mechanism, however, has not been formalized nor completely automated.

3.2.2 Fixed-Point Evaluation (Linguist)

In the previous section we discussed the problem of circular attribute grammars and presented a partial solution based on a lazy attribute evaluation method. In this section we continue with a special technique that is more powerful than the lazy strategy, in that a larger class of cyclic dependencies can be solved. Note, however, that even this method based on an iterative computation scheme is able to evaluate only a subclass of circular attribute grammars. In order to evaluate grammars falling beyond this technique, it would be necessary to transform the grammar into a tractable form, e.g., by introducing more powerful semantic functions, which usuallly would make the attribute grammar more complex than the original circular one.

Formally, a circular attribute grammar induces a recursive definition $x = f(x)$ on all the attribute instances x within a cycle in the attribute dependency graph. Extensive investigations, based on mathematical fixed-point theory [Kleene 1952], have been made on establishing computational means to solve equations of such form. The general idea is to compute the *least fixed point* of x by starting from an initial special *bottom* value $\perp$, and by successively approximating the value of x until it is not changed anymore; that is, the least fixed point has been reached:

$$x := \perp\ ; x := f(x); x := f(x); \ldots$$

To guarantee termination of the fixed-point computation, the domain of x must be a complete partial order in which equality can be tested (having $\perp$ as the smallest value), and the function f must be (Scott-)continuous [Chirica and Martin 1979]. If that is the case then some computation of $f(x)$ in the sequence above will finally return the current value of its parameter x, and iteration can be finished with x having its final value $f(f(\ldots f(\perp)\ldots))$.

We can apply an iterative fixed-point attribute evaluation method on the circular DESK + attribute grammar of the previous section. Intuitively, the method first assigns $\perp$ to each circularly defined attribute instance in the subtree for the constant definition part (ConstPart) of the source program; see Figure 9. In the first iteration, the value attribute instances of all the constants (ConstDef) with a literal value can be evaluated because they are not actually circularly defined. Also, the environment attribute instances envi and envs can be updated to include the literal constants. In the next iteration, those constants can be processed and inserted in the symbol table that are symbolically defined in terms of the constants evaluated in the previous iteration. This gradual evaluation scheme is repeated until no more constants become defined during some iteration.

For instance, let us apply the iterative strategy for evaluating the constant definition part of the DESK + program (3.1):

$$\ldots \text{WHERE x = y, y = z, z = 2}$$

Initially, all the defined constants have *bottom* as their value, resulting in the following symbol table:

$$[(x, \perp), (y, \perp), (z, \perp)]$$

In the first iteration, the actual value of z is computed since it is literal:

$$[(x, \perp), (y, \perp), (z, 2)]$$

In the second iteration, the constant y defined in terms of z can be evaluated:

$$[(x, \perp), (y, 2), (z, 2)]$$

In the third iteration, the constant x can be evaluated, resulting in the following final symbol table as the fixed point:

$$[(x, 2), (y, 2), (z, 2)]$$

Notice that there are two reasons why a constant c might still have *bottom* as its value in the final symbol table:

(1) c is defined in terms of an undefined constant d: e.g.,

```
PRINT...WHERE c = d; or
```

(2) c is recursively defined: e.g.,

```
PRINT...WHERE c = d, d = c
```

In both cases the DESK + program is illegal. To be able to detect such situations, the final value of the symbol table could be checked. Another, more liberal solution is to forbid only the *use* of such invalid constants in the expression part of the program. Then a symbolic operand of an expression could be specified, e.g., with the following production which, for brevity, is in this case not in normal form:

```
Factor → ConstName
  {Factor.value = getvalue
    (ConstName.name, Factor.envi),
  Factor ok = isin
    (ConstName.name, Factor envi)
      and Factor.value ≠ ⊥ }
```

To be safe with the iterative evaluation sketched above, the domain of both the constant values and the symbol table must be a complete partial order, and the involved semantic functions must be continuous with a stationary upper bound. Such a rigorous interpretation of domains and semantic functions within a circular attribute grammar similar to our DESK + grammar is given in Farrow [1986], where also the fixed-point method is precisely described in connection of the Linguist system [Declarative Systems 1992]. Other strategies of effectively evaluating circular attribute grammars are presented in, e.g., Jones and Simon [1986] and Jones [1990]. All these approaches define general attribute grammar classes that are circular in their characteristic graphs but still well defined with respect to a fixed-point computation.

3.3 Implicit Paradigms

In the previous sections we have discussed different ways to raise the expressiveness of attribute grammars by applying logic or functional programming in the attribute evaluation process. The semantic power of these attribute grammar paradigms is provided by the semantic functions that are written in a logic or a functional language. Hence, the evaluation paradigm is *explicitly* included in the attribute grammar itself.

In addition to such explicit mechanisms, some specialized techniques have been developed where an unconventional attribute evaluation behavior is automatically generated from an attribute grammar, thus being *implicit* in nature. Ideally, exploiting such an evaluation paradigm should in no way affect the form of the attribute grammar. In practice, however, it may be necessary to include some instructing commands or semantic rules in the specification, typically for optimizing the performance of the generated language processor. In this section we give a brief account of two popular implicit and specialized attribute evaluation techniques, *parallel evaluation* (Section 3.3.1) and *incremental evaluation* (Section 3.3.2).

3.3.1 Parallel Attribute Evaluation

A programming language whose operations have a total order at execution time can be referred to as a sequential one. In contrast, *concurrent programming* is a paradigm where operations can be executed *in parallel*, that is, simultaneously under a special multiprocessor hardware.

With the advent of commercial parallel architectures, concurrent programming has in recent years gained considerable theoretical and practical importance. A *process* is the basic self-standing element within a concurrent program, having local data which is manipulated by local code. The exchange of data between processes makes them *communicate*, either using explicitly sent data structures (messages), or through a shared memory that can be accessed by several processes. *Synchronization* is the mechanism making the execution of the entire program proceed in a controlled fashion by imposing that a set of processes must be in a certain inner state before they

can continue their execution (e.g., because one process needs data from another process). If processes can be executed independent of each other, they are called *asynchronous*.

Although concurrent concepts are most natural ones for modeling some inherently parallel systems (such as operating systems), the primary merit of concurrent programming is the exploitation of parallel hardware for achieving maximal performance. Thus the benefits of the approach should be the largest when applied on frequently used software, such as compilers. Parallel compilation techniques are, however, still a rather unexplored area, and experimental results have not been available until recently (e.g., Katseff [1988], Gross et al. [1989], and Seshadri and Wortman [1991]). Typical results from these parallelizing experiments indicate a compilation speedup of 3 using 5 processors and a speedup of 5 using 9 processors.

Because concurrent and parallel compilation techniques are still immature, the design of methods and tools for specifying and automatically implementing such compilers is a relatively young research topic. It is surprising, though, that the first historical implementation of attribute grammars, FOLDS [Fang 1972], was already based on considering productions as processes:

Production = Process

In FOLDS, processes could also be defined attributewise, giving rise to the following correspondence:

Attribute = Process

In the current approaches, the most popular principle can be expressed with the correspondence

Subtree = Process

emphasizing that the parallelization is especially directed to the dynamic instance (attributed tree) of an attribute grammar. Two subtrees should be evaluated in parallel only if the semantic coupling between them is loose enough. The ideal case is that the subtrees are totally independent such that no attribute instance in one subtree depends on an attribute instance in the other subtree. While total independence is rare in practical attribute grammars, an analysis of some frequently occurring attribution patterns has shown that often the symbols within one production can be arranged into disjoint sets such that there is only one attribute dependency between the sets [Klein and Koskimies 1989]. If each set is realized as a process, the communication between the processes is quite inexpensive. Within each process, the nodes for the symbols of the corresponding set are evaluated sequentially in the order determined by the attribute dependencies in the grammar.

While the tendency in the parallel attribute evaluation paradigm lies on the dynamic side, it is desirable to partition the attributed tree into processes by a static analysis of the attribute grammar. General static multipass strategies for parallel evaluation are proposed in, e.g., Kuiper and Zwierstra [1990], Zaring [1990], and Klein [1992]. One-pass parallel evaluation is discussed in, e.g., Klein and Koskimies [1989; 1990]. A less automated method, based on explicit user-given parallelization instructions in the attribute grammar, is presented in Boehm and Zwaenepoel [1987]. An extensive survey of parallel attribute evaluation methods is given in Jourdan [1991].

One attribute evaluation approach reaching for a maximal amount of parallelism would be to consider an attribute grammar as a data-flow program, as suggested in Farrow [1983]. Then attribute instances would map into data-flow nodes that would fire when all their input instance nodes have been evaluated. Rather surprisingly, though, no practical experiments on this natural and appealing idea have been reported.

We could apply a parallel evaluation strategy on the DESK(+) attribute

grammar for raising the speed of the generated DESK(+) compiler. Without going into details, the most effective way would be to parallelize the code generation phase of the compiler by allocating expression subtrees into processors, thus employing the "Subtree = Process" principle. A static analysis of the attribute grammar (see Example 1.3.4) shows that the different operands of a DESK(+) expression, represented by the nonterminal Factor, are loosely coupled in the sense that their attribute instances do not depend on each other. Thus, the Factor subtrees could be effectively evaluated in parallel, as soon as the symbol table has been constructed and propagated to the Factor.envi attribute instances. The analysis also shows that the constant definitions (instances of ConstDef) are similarly independent and could be evaluated in parallel as well.

3.3.2 Incremental Attribute Evaluation

The development of high-performance workstations has given rise to advanced interactive applications. The fundamental principle behind such systems is *incremental computation*, where existing information is constantly updated in response to the user's actions. To be effective, the update should exclude the execution of those parts of the application that are not affected by the particular interaction.

The key element in interactive program development is a language-based programming environment that provides integrated tools for the whole software engineering process. In such a framework the tools must be based on incremental techniques so as to reduce the workload caused by program modifications. Usually the main component of an integrated programming environment is a syntax-directed editor that maintains an internal representation of the program and keeps track of the modifications induced on it by the program-editing process. The other components of the environment (such as an interpreter, a compiler, and a debugger) effectively share the same internal representation. Smalltalk [Goldberg and Robson 1983] and Mjølner/Orm [Knudsen et al. 1993] are two representative examples of an integrated programming environment.

An attractive characteristic of integrated language-based programming environments is that they can be automatically generated from a high-level description. The generation of syntax-directed editors has been an especially popular topic of research, usually based on attribute grammars as the specification formalism. The Synthesizer Generator is the most well-known editor generator based on attribute grammars [Reps and Teitelbaum 1989]. GANDALF is a more general system that supports the generation of complete environments, also with attribute grammars as one of the specification formalisms [Notkin 1985].

The central problem in incremental language processing based on attribute grammars is the efficient evaluation of attributes. For source programs of realistic size, an attributed tree may be quite large for an internal representation. Also, the attribute dependency graph that models the information needed for implementing an editing update of the source program covers typically the whole tree. For these reasons, it is not reasonable to reevaluate the whole attributed tree completely in reaction to a small syntactic modification.

The need for efficient and immediate updates has resulted in the development of *incremental attribute evaluation techniques*. Intuitively, these methods try to minimize the semantic cost of a syntactic program modification by reevaluating only those attribute instances whose values are affected by the modification. Usually the methods rely on the characteristic dependency graphs for the underlying attribute grammar to compute incrementally the affected region of the attributed tree. Recall from Section 1.3 that the characteristic graphs approximate the functional dependencies *statically* between the attribute instances in any derivation tree for the grammar, thus

providing for an effective basis to deduce immediately which attribute instances depend (indirectly) on the modified nodes in the tree and shall therefore be reevaluated.

Example systems with an optimized incremental attribute evaluation support include, e.g., OPTRAN [Lipps et al. 1988], the Synthesizer Generator [Reps and Teitelbaum 1989], Pan [Ballance and Graham 1991], and Mjølner/Orm [Hedin 1992]. Combining the specialized techniques of parallel and incremental evaluation is discussed in Alblas [1990].

The attribute grammar for DESK(+) could be used as such for generating a syntax-directed DESK(+) editor with an implicit incremental attribute evaluator embeded. In order to make the editor pleasant to use, the attribute grammar should, however, be extended with rules pertaining solely to the interactive editing process without touching the actual semantics of DESK(+) and its compiler. Such customizing instructions should define how the program is displayed on the screen, which error messages the editor shall give, and what kind of source-level transformations are allowed during editing. For instance, the grammar could enforce restrictions that the main expression and the constant definitions of the edited DESK(+) program are each displayed on their own line, and that the editor shall, in response to a request, replace the main expression with its literal value.

Regarding the compilation process, an incremental attribute evaluator for DESK(+) would reevaluate only those attribute instances that are affected by an editing action. For instance, changing the main expression of the program would not induce any reevaluation on the constant definition part because there is no flow of attribute values from the expression subtree into the constant definition subtree in the characteristic graphs (see Figure 1) nor in the underlying attributed tree (see Figure 4). The target code, however, depends on the contents of the main expression, and that is why code attribute instances would have to be reevaluated after such a modification, resulting in an updated sequence of generated assembly instructions.

4. CONCLUSIONS

In this survey we have presented alternative methods of expressing attribute grammars by integrating into them various concepts of well-founded programming paradigms. Starting from the standard notion of attribute grammars, different styles of expressing the formalism have been created. The analogy between attribute grammar styles and programming paradigms is reflected in this survey by the characterization of the formalisms as *structured*, *modular*, *object-oriented*, *logic*, and *functional attribute grammars*. To complement these self-standing paradigms, we have also presented two implicit paradigms whose nature is manifested by a special *parallel* or *incremental* attribute evaluation strategy which is automatically generated, rather than being explicit in the attribute grammar.

The motivation to develop these attribute grammar paradigms has been the same as the trend in the evolution of programming languages: to provide more high-level, flexible, expressive, or efficient concepts for specifying and implementing a particular application. The following list summarizes the aspects that are often emphasized when analyzing the merits of programming languages (e.g., Wasserman [1980]). The same criteria can also be used for evaluating and relating the different attribute grammar paradigms against each other:

- simplicity
- readability
- expressiveness
- regularity (orthogonality)
- security
- reuse
- application support
- formal definition
- efficiency

Structured, modular, and object-oriented attribute grammars most notably sup-

port the organizational aspect of attribute grammars. We can illustrate the differences between these approaches by sketching in Figure 10 the structure of attribute grammars obtained when applying these paradigms. The views correspond to the following standard attribute grammar with the nonterminal symbols S (the start symbol), X, Y, and Z, the terminal symbols y and z, and the attributes a and b:

```
S → XY   {S.a = X.a}
X → Z    {X.a = Z b, X b = Z.a}
Y → y    { }
Z → z    {Z.a = 0, Z.b = 1}
```

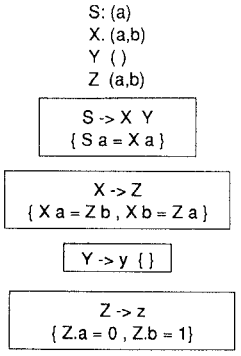
Treating productions as blocks (Figure 10a) results in an attribute grammar whose structure resembles the flat block structure of the programming language C, for example. Having nonterminals as blocks or procedures (Figure 10b) results in a more general form that corresponds to the nested block structure of Pascal, for example. Modeling either nonterminals, attributes, or productions as modules or classes (Figures 10c and d) has its counterpart, e.g., in Ada packages or Eiffel classes. (The interface part of a module/class is expressed using "windows" on the left border of the module/class. An arrow from a module/class A to a module/class B indicates that the public components of B are used by A.)

The merits of conventional structuring methods have been thoroughly analyzed in the literature of programming languages. A special account of blocks, procedures, and modules is given in Hanson [1981]. The general discussion is relevant in this application area as well. Blocks and procedures are valuable in structuring programs (here: attribute grammars) of modest size, whereas modules and classes are superior when building larger systems with a necessity for maintaining and reusing components (here: attribute grammar fragments). The two forms of modular attribute grammars differ in that having attributes as modules (Figure 10d) typically produces less modules, whereas the modules for nonterminals (Figure 10c) tend to be smaller and thus easier to maintain. The smallest number of modules would result when composing them as different aspects of an attribute grammar (not shown in the figure). Object-oriented attribute grammars provide additionally the inheritance concept which makes the paradigm superior to both structured and modular attribute grammars in abstraction capabilities.

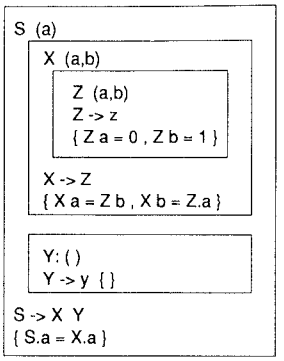
Functional and logic attribute grammars provide for powerful semantic facilities. On the functional side, we have discussed especially the use of lazy and iterative evaluation, and on the logic side the use of predicates and logical variables. These mechanisms have their merits in evaluating attribute grammars that are problematic under strict evaluation methods. Two concrete examples presented in this survey are statically circular attribute grammars that dynamically turn out to be noncircular when evaluated in a lazy or fixed-point-finding fashion, and strictly counter-one-pass grammars [Giegerich and Wilhelm 1978] that can be evaluated during parsing when employing logical variables as attribute instances.

The special attribute evaluation tools of functional and logic attribute grammars can be demonstrated by showing (Figure 11) what kind of attribute dependencies they accept. In the attributed subtree, X, Y, and Z denote nonterminal nodes, and inherited attribute instances are attached to the left and synthesized attribute instances to the right of the nodes. A strict dependency chain (i.e., a chain of attribute instances that are evaluated using strict semantic functions) is represented by the attributes ai, a lazy functional dependency chain by the attributes bi, and a logical one-pass dependency chain by the attributes ci.

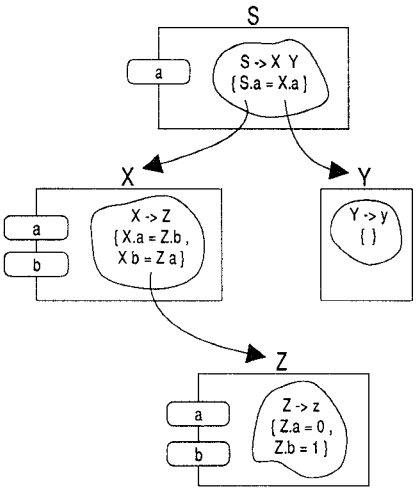
Let us assume that the underlying implementation applies a one-pass left-to-right attribute evaluation strategy. The grammar fragment is in each case well defined with respect to the attribute instances ai since they can always be evaluated using the standard L-attributed strategy. The functional and logical



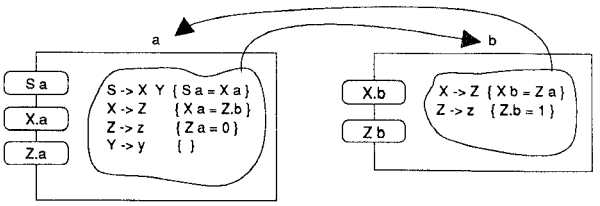
(a) Production = Block



(b) Nonterminal = Block
Nonterminal = Procedure



(c) Nonterminal = Module
Nonterminal = Class
Production = Class



(d) Attribute = Module

Figure 10. Organizational attribute grammars.

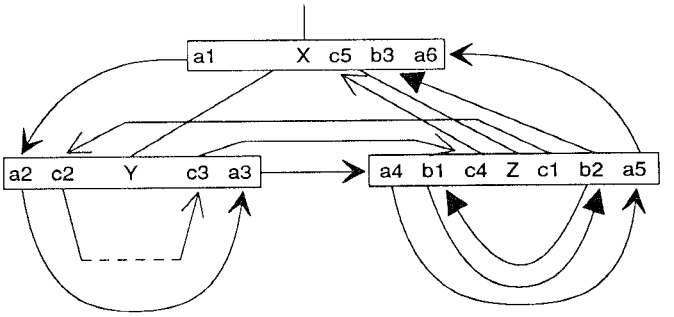


Figure 11. Strict, lazy, and logical dependencies.

chains are less restricting in their attribute dependencies, but on the other hand they induce some conditions on the semantic functions. The grammar fragment is well defined with respect to the attribute instances *bi* only if the value of $b3$ is not needed in computing the meaning of the program (or if the static dependency chain between $b1$ and $b2$ disappears dynamically when executing their semantic rules). Finally, the fragment is well defined with respect to the instances *ci* if the values of $c2$, $c3$, $c4$, and $c5$ (and all the instances depending on them) are not used as a strict argument of a semantic function, since then they can be evaluated using the logical one-pass strategy. A strict one-pass method can under no circumstances evaluate the *bi* and *ci* attribute instances.

In contrast to the paradigms discussed above, the facilities of concurrent and incremental programming do not primarily raise the conceptual level of attribute grammars. Instead, they focus on the performance of the generated language processors. Therefore, as a notation these implicit paradigms do not differ from ordinary attribute grammars, except that they in practice may contain some explicit user-specified instructions to the system. The significance of concurrency and incrementality should by no means be underestimated when selecting the applied attribute grammar paradigm because efficiency is a major concern in frequently used software, such as compilers.

To summarize the discussion, the presented approaches can be roughly divided into three categories. The main contribution of the organizational (structured, modular, object-oriented) paradigms is raising the *software engineering* support of the metalanguages. Hence, these paradigms emphasize in particular such aspects as readability, security, and reuse of specifications. The evaluation (logic, functional) paradigms provide for powerful *semantic facilities*, thus facilitating simplicity, expressiveness, and regularity. Finally, the *efficiency* of language processors is stressed by the implicit (parallel, incremental) paradigms. Note that since the paradigms are based on the same model of standard attribute grammars, they all have a formal foundation, although the notion of well-definedness may slightly vary.

Choosing a particular attribute grammar paradigm for specifying or implementing a language depends on how the various aspects are emphasized. With a small and simple language that presumably is not going to change, even the standard pure style is sufficient. However, a larger or more complex language that may exist in many generations involves a significant degree of software engineering. In that case a modular or object-oriented paradigm is certainly a better choice. The powerful logic and functional facilities are valuable when one wants to focus on the specification of the language and on keeping the attribute grammar as declarative as possi-

ble, without forcing any explicit attribute evaluation decisions. Finally, parallelism should be considered when both performance of the language processor is the primary goal and an appropriate architecture is available.

The application area (when some other than mere language processing) may also affect the decision. Interactive applications require obviously the use of incremental evaluation techniques. Modeling of general sequential software with attribute grammars [Shinoda ad Katayama 1988; Frost 1992] is an area where an organizational paradigm with an underlying one-pass implementation is most feasible. The object-oriented paradigm is the most appropriate one when modeling communication protocols [Chapman 1990], with interfaces, abstract data, and state machines as a major concern. Distributed applications [Kaiser and Kaplan 1993] require a parallel evaluation strategy in order to be effective. As the last example, VLSI design [Jones and Simon 1986] and data-flow analysis [Babich and Jazayeri 1978] are two areas whose natural description leads to a circular attribute grammar, thus making it feasible to apply the functional fixed-point evaluation paradigm.

The choices by the designers of a language processor generator may also be strongly influenced by the intended application area of the tool. For instance, both the Synthesizer Generator [Reps and Teitelbaum 1989] and Pan [Ballance et al. 1990] generate incremental attribute evaluators. Since the main objective of the Synthesizer Generator is just syntax-directed editing, the metalanguage of the system is a rather direct adaptation of the standard form of attribute grammars. In contrast, the Pan system has a broader scope of supporting the development and maintenance of general (un)structured documents, not jut programs. Such a versatile environment is usable only if it maintains a sharable database of documents. To map a specification more directly into the database with a set of integrated access operations, the metalanguage of the system has been based on the logic attribute grammar paradigm.

In this survey we have demonstrated how attribute grammars have been integrated with different programming paradigms. Most existing systems support a single paradigm. The next natural phase in the evolution of attribute grammars may well be the development of language processor generators whose metalanguage integrates several paradigms. Using the taxonomy of this survey, such a *multiparadigm* system should (1) have advanced software engineering capabilities, (2) provide for powerful semantic abstractions, and (3) generate efficient language processors. One recent example of such a versatile system is FNC-2 whose metalanguage is based on blocks and modules (category 1), pattern matching and polymorphism (category 2), and an underlying parallel and incremental attribute evaluation strategy (category 3) [Jourdan et al. 1990; Jourdan and Parigot 1991].

While the original and most widely recognized application area of attribute grammars is compiler generation, only a few commercial compilers have actually been developed using attribute grammars as a design or implementation tool. Indeed, it has been argued that attribute grammars fall short in the production of high-speed compilers for conventional general-purpose programming languages [Waite 1990]. Two common explanations for such a conjecture are that attribute grammars are just a *model* of compilation and thus too primitive for a real engineering discipline, and that they do not directly support the generation and optimization of low-level machine code. On the other hand, the success of YACC shows the practical usability of attribute grammars at least in simple front-end applications [Johnson 1975].

In this survey we have shown that attribute grammars have developed into an advanced language processing methodology by a paradigmatic evolution, and that the software engineering requirements are taken into account in modern metalanguages and systems. The second main

problem, lack of proper support for code generation and optimization, originates in the model itself and is harder to attack.

From this viewpoint, we can argue that attribute grammar paradigms should actually not be applied in the development of conventional optimizing compilers. Instead, a much more suitable area is the design and implementation of *special-purpose application languages*, typical examples being database query languages (e.g., SQL), robot control languages, hardware design languages (e.g., VHDL), protocol specification languages (e.g., ASN.1, SDL), and narrow "little languages" [Bentley 1986]. Such a language is usually developed with emphasis on high-level application abstractions rather than on optimal performance. Other typical aspects are rapid implementation, exploratory and flexible design, modifiability, reuse of existing language concepts, and mapping into a general-purpose programming language ("source-to-source translation"). These characteristics conform to the methodology of language processor generators which indeed have widely and successfully been used in the implementation of a variety of application languages.

Due to the rapid growth of software intensive applications, it is most likely that the main trend of language design will more and more focus on application languages. To be productive, the development of these languages must be based on high-level automated tools with support for both design and implementation. A realistic methodological foundation for application-oriented language processing in the future are attribute grammar paradigms, such as promoted in this survey.

ACKNOWLEDGMENTS

This survey has been inspired by my various joint research projects with Kai Koskimies. The comments from Görel Hedin, Reino Kurki-Suonio, Seppo Sippu, and Esko Ukkonen have been useful when producing earlier versions of the survey. The suggestions of the anonymous referees have been very helpful for focusing and improving the presentation.

REFERENCES

ABRAMSON, H. 1984. Definite clause translation grammars. In *Proceedings of the 1984 IEEE Symposium on Logic Programming* (Atlantic City, N.J.). IEEE, New York, 233–240

ACM COMPUTING SURVEYS *21*, 3, 1989. Special Issue on Programming Language Paradigms.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986 *Compilers—Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

AKSIT, M., MOSTERT, R., AND HAVERKORT, B. 1990. Compiler generation based on grammar inheritance. Rep. 90-07, Dept. of Computer Science, Univ. of Twente.

ALBLAS, H. 1991. Attribute evaluation methods. In *Attribute Grammars, Applications, and Systems*. Lecture Notes in Computer Science, vol 545, Springer-Verlag, New York, 48–113.

ALBLAS, H. 1990. Concurrent incremental attribute evaluation. In *Proceedings of the International Conference on Attribute Grammars and their Applications*, Lecture Notes in Computer Science, vol 461 Springer-Verlag, New York, 343–358.

ALBLAS, H. AND MELICHAR, B. (EDS) 1991. *Attribute Grammars, Applications, and Systems*. Lecture Notes in Computer Science, vol 545. Springer-Verlag, New York.

ALEXIN, Z , GYIMÓTHY, T., HORVÁTH, T , AND FÁBRICZ, K. 1990. Attribute grammar specification for a natural language understanding interface In *Proceedings of the International Conference on Attribute Grammars and their Applications*, Lecture Notes in Computer Science, vol 461, Springer-Verlag, New York, 313–326.

APPEL, A. W. AND MACQUEEN, D. B. 1991. Standard ML of New Jersey In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*. Lecture Notes in Computer Science, vol. 528, J. Maluszynski and M. Wirsing, Eds. Springer-Verlag, New York, 1–13.

ARBAB, B. 1986. Compiling circular attribute grammars into Prolog. *IBM J. Res. Devel. 30*, 3, 294–309.

AUGUSTEIJN, L. 1990. The Elegant Compiler Generator System. In *Proceedings of the International Conference on Attribute Grammars and their Applications*, Lecture Notes in Computer Science, vol 461. Springer-Verlag, New York, 238–254.

BABICH, W. A AND JAZAYERI, M. 1978 The method of attributes for data flow analysis, Part I. Exhaustive analysis; Part II: Demand analysis *Acta Informatica 10*, 245–272.

BALLANCE, R. A. AND GRAHAM, S. L. 1991. Incremental consistency maintenance for interactive applications In *Proceedings of the 8th International Conference on Logic Programming*, K. Furukawa, Ed The MIT Press, Cambridge, Mass., 895–909.

BALLANCE, R. A., GRAHAM, S. L., AND VAN DE VANTER, M. L. 1990. The Pan language-based editing system for integrated development environments. In *Proceedings of the 4th ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '90). ACM SIGSOFT Softw. Eng. Notes 15*, 6, 77–93.

BENTLEY, J. 1986. Little languages. *Commun. ACM 29*, 8, 711–722.

BOCHMANN, G. V. 1976. Semantic evaluation from left to right. *Commun. ACM 19*, 2, 55–62.

BOCHMANN, G. V. AND WARD, P. 1978. Compiler writing system for attribute grammars. *Comput. J. 21*, 2, 144–148.

BOEHM, H.-J. AND ZWAENEPOEL, W. 1987. Parallel attribute grammar evaluation. In *Proceedings of the 7th International IEEE Conference on Distributed Computing Systems* (Berlin). IEEE, New York, 347–354.

BRYANT, B. R. AND PAN, A. 1989. Rapid prototyping of programming language semantics using Prolog. In *Proceedings of IEEE COMPSAC '89* (Orlando, Fl.). IEEE, New York, 439–446.

CHAPMAN, N. P. 1990. Defining, analysing and implementing communication protocols using attribute grammars. *Formal Aspects Comput. 2*, 359–392.

CHIRICA, L. M. AND MARTIN, D. F. 1979. An order-algebraic definition of Knuthian semantics. *Math. Syst. Theory 13*, 1, 1–27.

CHOMSKY, N. 1959. On certain formal properties of grammars. *Inf. Contr. 2*, 137–167.

COLMERAUER, A. 1978. Metamorphosis grammars. In *Natural Language Communication with Computers*, L. Bolc, Ed. Springer-Verlag, New York, 133–189.

COURCELLE, B. AND FRANCHI-ZANNETTACCI, P. 1982. Attribute grammars and recursive program schemes. *Theor. Comput. Sci. 17*, 2, 163–191 (Part I) and *17*, 3, 235–257 (Part II).

COUSINEAU, G. AND HUET, G. 1990. The CAML Primer. Tech. Rep. 122, INRIA.

CRIMI, C., GUERCIO, A., PACINI, G., TORTORA, G., AND TUCCI, M. 1990. Automating visual language generation. *IEEE Trans. Softw. Eng. 16*, 10, 1122–1135.

DAHL, V. AND ABRAMSON, H. 1989. *Logic Grammars*. Springer-Verlag, New York.

DAHL, V. AND ABRAMSON, H. 1984. Gapping grammars. In *Proceedings of the 2nd International Logic Programming Conference* (Uppsala).

DECLARATIVE SYSTEMS 1992. *User Manual for the Linguist Translator-Writing System*. Declarative Systems, Inc., Palo Alto, Calif.

DELEST, M. P. AND FEDOU, J. M. 1992. Attribute grammars are useful for combinatorics. *Theor. Comput. Sci. 98*, 1, 65–76.

DEPARTMENT OF DEFENSE. 1983. *The Programming Language Ada, Reference Manual*. ANSI/MIL-STD-1815A-1983. Springer-Verlag, New York.

DERANSART, P. AND JOURDAN, M. (ED.). 1990. *Proceedings of the International Conference on Attribute Grammars and their Applications*. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York.

DERANSART, P. AND MALUSZYNSKI, J. 1993. *A Grammatical View of Logic Programming*. The MIT Press, Cambridge, Mass.

DERANSART, P. AND MALUSZYNSKI, J. 1985. Relating logic programs and attribute grammars. *J. Logic Program. 2*, 2, 119–155.

DERANSART, P., JOURDAN, M., AND LORHO, B. 1988. *Attribute Grammars—Definitions, Systems and Bibliography*. Lecture Notes in Computer Science, vol. 323, Springer-Verlag, New York.

DING, S. AND KATAYAMA, T. 1993. Attributed state machines for behavior specification of reactive systems, In *Proceedings of the 5th International Conference on Software Engineering and Knowledge Engineering (SEKE'93)* (San Francisco). Knowledge Systems Institute, 695–702.

DUECK, G. D. P. AND CORMACK, G. V. 1990. Modular attribute grammars. *Comput. J. 33*, 2, 164–172.

EXPERT SOFTWARE SYSTEMS. 1984. *MIRA, User's Manual*. Expert Software Systems.

FANG, I. 1972. FOLDS—A declarative semantic formal language definition system. Rep. STAN-CS-72-239, Computer Science Department, Stanford University, Stanford, Calif.

FARROW, R. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined attribute grammars. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction. ACM SIGPLAN Not. 21*, 7, 85–98.

FARROW, R. 1983. Attribute grammars and data-flow languages In *Proceedings of the SIGPLAN'83 Symposium on Programming Language Issues in Software Systems. ACM SIGPLAN Not. 18*, 6, 28–40.

FARROW, R. 1982. LINGUIST-86, yet another translator writing system based on attribute grammars. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction. ACM SIGPLAN Not. 17*, 6, 160–171.

FARROW, R., MARLOWE, T. J., AND YELLIN, D. M. 1992. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the 19th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mex.). ACM, New York, 223–234.

FISCHER, C. N. AND LEBLANC, R. J. 1988. *Crafting a Compiler*. Benjamin-Cummings, Menlo Park, Calif.

FROST, R. A. 1993. Guarded attribute grammars. *Softw. Prac. Exp. 23*, 10, 1139–1156.

FROST, R. A. 1992. Constructing programs as executable attribute grammars. *Comput. J 35*, 4, 376–389.

GANZINGER, H. 1980. Transforming denotational semantics into practical attribute grammars. In *Proceedings of the Workshop on Semantics-*

Directed Compiler Generation Lecture Notes in Computer Science, vol. 94, N. D. Jones, Ed. Springer-Verlag, New York, 1–69.

GANZINGER, H AND GIEGERICH, R. 1984. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction. ACM SIGPLAN Not. 19*, 6, 157–170.

GANZINGER, H., GIEGERICH, R., MONCKE, U., AND WILHELM, R. 1982. A truly generative semantics-directed compiler generator. In *Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction. ACM SIGPLAN Not. 17*, 6, 172–184.

GARRISON, P. 1987 Modeling and implementation of visibility in programming languages. Rep. UCB/CSC 88/400, Computer Science Division, Univ. of California, Berkeley, Calif.

GIEGERICH, R. AND WILHELM, R. 1978. Counter-one-pass features in one-pass compilation: A formalisation using attribute grammars. *Inf. Process. Lett. 7*, 6, 279–284.

GOLDBERG, A. AND ROBSON, D. 1983. *Smalltalk*-80: *The Language and Its Implementation* Addison-Wesley, Reading, Mass.

GROSCH, J. 1990. Object-oriented attribute grammars. In *Proceedings of the 5th International Symposium on Computer and Information Sciences (ISCIS V)*, A. E. Harmancı and E. Gelenke, Eds. 807–816.

GROSS, T., ZOBEL, A., AND ZOLG, M. 1989. Parallel compilation for a parallel machine. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 24*, 7, 91–100.

GROSSMANN, R., HUTSCHENREITER, J., LAMPE, J., LÖTZSCH, J., AND MAGER, K. 1984. DEPOT2a—Metasystem fur die Analyse und Verarbeitung verbundener Fachsprachen. Anwenderhandbuch, Sektion Mathematik, Technische Universität Dresden.

HANSON, D. R. 1981 Is block structure necessary? *Soft. Prac. Exp. 11*, 8, 853–866.

HEDIN, G. 1989. An object-oriented notation for attribute grammars. In *Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP '89)*, S. Cook, Ed. British Informatics Society Ltd., Nottingham, 329–345.

HEDIN, G. 1994. An overview of door attribute grammars In *Proceedings of the International Conference on Compiler Construction (CC'94)* Lecture Notes in Computer Science, vol 786, P. Fritzson, Ed. Springer-Verlag, New York.

HEDIN, G. 1992. Incremental semantic analysis. Doctoral dissertation, Lund University.

HEERING, J., KLINT, P., AND REKERS, J. 1990. Incremental generation of parsers. *IEEE Trans. Softw Eng. SE-16*, 12, 1344–1351.

HORWITZ, S. AND REPS, T. 1992. The use of program dependence graphs in software engineering. In *Proceedings of the 14th International Conference on Software Engineering* IEEE Computer Society Press, Los Alamitos, Calif., 392–411.

HUDSON, S. E. AND KING, R. 1987. Implementing a user interface as a system of attributes In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. ACM SIGPLAN Not. 22*, 1, 143–149.

IRONS, E. T 1961. A syntax directed compiler for Algol 60. *Commun. ACM 4*, 1, 51–55

JAZAYERI, M., OGDEN, W F., AND ROUNDS, W C. 1975. The intrinsically exponential complexity of the circularity problem for attribute grammars. *Commun. ACM 18*, 12, 679–706.

JOHNSON, S. C. 1975. YACC, yet another compiler compiler. Rep. CS-TR-32, Bell Laboratories, Murray Hill, N.J.

JOHNSSON, T. 1987. Attribute grammars as a functional programming paradigm In *Proceedings of the Symposium on Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 274, G. Kahn, Ed. Springer-Verlag, New York, 154–173

JONES, L. G. 1990. Efficient evaluation of circular attribute grammars. *ACM Trans. Program. Lang. Syst. 12*, 3, 429–462.

JONES, R. 1986. Flex—An experience of Miranda. UKC Computing Laboratory Rep. 38, Univ. of Kent at Canterbury.

JONES, L. G AND SIMON, J. 1986. Hierarchical VLSI design systems based on attribute grammars. In *Conference Record of the 13th ACM Symposium on Principles of Programming Languages* ACM, New York, 58–69.

JOURDAN, M. 1991. A survey of parallel attribute evaluation methods. In *Attribute Grammars, Applications, and Systems*. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, New York, 234–255.

JOURDAN, M. 1984. Strongly non-circular attribute grammars and their recursive evaluation. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction ACM SIGPLAN Not. 19*, 6, 81–93.

JOURDAN, M. AND PARIGOT, D. 1991. Internals and externals of the FNC-2 attribute grammar system. In *Attribute Grammars, Applications and Systems* Lecture Notes in Computer Science, vol 545. Springer-Verlag, New York, 485–506.

JOURDAN, M , LE BELLEC, C , AND PARIGOT, D. 1990 The OLGA attribute grammar description language: Design, implementation and evaluation In *Proceedings of the International Conference on Attribute Grammars and their Applications*. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 222–237.

KAISER, G AND KAPLAN, S. M. 1993. Parallel and distributed incremental attribute evaluation algorithms for multiuser software development environments. *ACM Trans. Softw Eng. Method. 2*, 1, 47–92.

KASTENS, U. 1991. Attribute grammars as a specification method. In *Attribute Grammars, Applications and Systems* Lecture Notes in Com-

puter Science, vol. 545. Springer-Verlag, New York, 16–47.

KASTENS, U. 1980. Ordered attributed grammars. *Acta Informatica 13*, 229–256.

KASTENS, U. AND WAITE, W. M. 1991. An abstract data type for name analysis. *Acta Informatica 28*, 6, 539–558.

KASTENS, U., HUTT, B., AND ZIMMERMANN, E. 1987. *User Manual for the GAG-System, Version 7.* GAG Documentation, GMD Forschungsstelle an der Universität Karlsruhe.

KASTENS, U., HUTT, B., AND ZIMMERMANN, E. 1982. *GAG: A Practical Compiler Generator.* Lecture Notes in Computer Science, vol. 141. Springer-Verlag, New York.

KATSEFF, H. P. 1988. Data partitioning to implement a parallel assembler. In *Proceedings of the ACM Conference on Parallel Programming: Experiences with Applications, Languages and Systems (PPEALS 1988). ACM SIGPLAN Not. 23*, 9, 66–76.

KENNEDY, K. AND WARREN, S. 1976. Automatic generation of efficient evaluators for attribute grammars. In *Conference Record of the 3rd ACM Symposium on Principles of Programming Languages.* ACM, New York, 32–49.

KLEENE, S. C. 1952. *Introduction to Metamathematics.* North-Holland, Amsterdam.

KLEIN, E. 1992. Parallel ordered attribute grammars. In *Proceedings of the IEEE Conference on Computer Languages.* IEEE, New York, 106–116.

KLEIN, E. AND KOSKIMIES, K. 1989. The parallelization of one-pass compilers. Arbeitspapiere der GMD 416, Gesellschaft für Mathematik und Datenverabeitung mbH.

KLEIN, E. AND KOSKIMIES, K. 1990. Parallel one-pass compilation. In *Proceedings of the International Conference on Attribute Grammars and their Applications.* Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 76–90.

KNUTH, D. E. 1990. The genesis of attribute grammars. In *Proceedings of the International Conference on Attribute Grammars and their Applications.* Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 1–12.

KNUTH, D. E. 1971. Examples of formal semantics. In *Proceedings of the Symposium on Semantics of Algorithmic Languages.* Lecture Notes in Mathematics, vol. 188, E. Engeler, Ed. Springer-Verlag, New York, 212–235.

KNUTH, D. E. 1968. Semantics of context-free languages. *Math. Syst. Theory 2*, 2, 127–145. (Corrigenda: *Math. Syst. Theory 5*, 1, 1971, 95–96.)

KNUTH, D. E. 1965. On the translation of languages from left to right. *Inf. Contr. 8*, 6, 607–639.

KOSKIMIES, K. 1991. Object-orientation in attribute grammars. In *Attribute Grammars, Applications and Systems.* Lecture Notes in Computer Science, vol. 545. Springer-Verlag, New York, 297–329.

KOSKIMIES, K. 1990. Lazy recursive descent parsing for modular language implementation. *Softw. Pract. Exp. 20*, 8, 749–772.

KOSKIMIES, K. 1989. Software engineering aspects in language implementation. In *Proceedings of the 2nd Workshop on Compiler Compilers and High Speed Compilation.* Lecture Notes in Computer Science, vol. 371, D. Hammer, Ed. Springer-Verlag, New York, 39–51.

KOSKIMIES, K. 1984. A specification language for one-pass semantic analysis. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction. ACM SIGPLAN Not. 19*, 6, 179–189.

KOSKIMIES, K. AND PAAKKI, J. 1991. High-level tools for language implementation. *J. Syst. Softw. 15*, 2, 115–131.

KOSKIMIES, K. AND PAAKKI, J. 1990. *Automating Language Implementation—A Pragmatic Approach.* Ellis Horwood, Chichester, England.

KOSKIMIES, K., NURMI, O., PAAKKI, J., AND SIPPU, S. 1988. The design of a language processor generator. *Softw. Pract. Exper. 18*, 2, 107–135.

KUIPER, M. F. AND SWIERSTRA, S. D. 1990. Parallel attribute evaluation: Structure of evaluators and detection of parallelism. In *Proceedings of the International Conference on Attribute Grammars and their Applications.* Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 61–75.

LEE, P. 1989. *Realistic Compiler Generation.* The MIT Press, Cambridge, Mass.

LEWI, J., DE VLAMINCK, K., HUENS, J., AND HUYBRECHTS, M. 1979. *A Programming Methodology in Compiler Construction.* Vol. 1 and 2. North-Holland, Amsterdam.

LEWI, J., DE VLAMINCK, K., STEEGMANS, E., AND VAN HOREBEEK, J. 1992. *Software Development by LL(1) Syntax Description.* John Wiley & Sons, New York.

LEWIS, P. M. AND STEARNS, R. E. 1968. Syntax-directed transduction. *J. ACM 15*, 3, 465–488.

LEWIS, P. M., ROSENKRANTZ, D. J., AND STEARNS, R. E. 1974. Attributed translations. *J. Comput. Syst. Sci. 9*, 279–307.

KNUDSEN, J. L., LÖFGREN, M., LEHRMANN MADSEN, O., AND MAGNUSSON, B. (EDS.). 1993. *Object-Oriented Environments: The Mjølner Approach.* Prentice-Hall, Englewood Cliffs, N.J.

LIPPS, P., MÖNCKE, U., OLK, M., AND WILHELM, R. 1988. Attribute (re)evaluation in OPTRAN. *Acta Informatica 26*, 213–239.

LONGLEY, M. 1987. Generating parsers in Miranda. UKC Computing Laboratory Rep. 49, Univ. of Kent at Canterbury.

MALUSZYNSKI, J. 1991. Attribute grammars and logic programming: A comparison of concepts. In *Attribute Grammars, Applications, and Systems.*

Lecture Notes in Computer Science, vol. 545. Springer-Verlag, New York, 330–357.

MALUSZYNSKI, J. 1982. A comparison of the logic programming language Prolog with two-level grammars. In *Proceedings of the 1st International Logic Programming Conference*, M. van Caneghem, Ed. Faculte des Sciences de Luminy, Marseilles, 193–199

MASON, T AND BROWN, D. 1990. *Lex & Yacc*. O'Reilly & Associates, Inc., Sebastopol, Calif.

MAYOH, B. 1981. Attribute grammars and mathematical semantics. *SIAM J. Comput. 10*, 3, 503–518.

NAUR, P. (ED.). 1960. Report on the algorithmic language ALGOL 60. *Commun. ACM 3*, 5, 299–314.

NOLL, T. AND VOGLER, H. 1994. Top-down parsing with simultaneous evaluation of noncircular attribute grammars *Fundamenta Inf. 20*, 4, 285–331.

NOTKIN, D. 1985. The Gandalf Project. *J. Syst Softw. 5*, 91–106.

PAAKKI, J. 1991. PROFIT: A system integrating logic programming and attribute grammars. In *Proceedings of the 3rd International Symposium on Programming Language Implementation and Logic Programming (PLILP '91)*. Lecture Notes in Computer Science, vol. 528, J. Maluszynski and M. Wirsing, Eds. Springer-Verlag, New York, 243–254.

PAAKKI, J. 1990. A logic-based modification of attribute grammars for practical compiler writing. In *Proceedings of the 7th International Conference on Logic Programming*, D. H. D Warren and P. Szeredi, Eds. The MIT Press, Cambridge, Mass., 203–217.

PAAKKI, J. AND TOPPOLA, K. 1990. An error-recovering form of DCGs. *Acta Cybernetica 9*, 3, 211–221.

PAULSON, L. 1982. A semantics-directed compiler generator In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages* (Albuquerque, New Mex.) ACM, New York, 224–233.

PEREIRA, F. C. N. AND WARREN, D H. D. 1980. Definite clause grammars for language analysis—A survey of the formalism and a comparison with augmented transition networks. *Artif. Intell. 13*, 231–278.

PEYTON JONES, S. L. 1985. Yacc in Sasl—An exercise in functional programming. *Softw. Pract. Exp. 15*, 8, 807–820

REISS, S. P. 1987 Automatic compiler production: The front end. *IEEE Trans. Softw. Eng. SE-13*, 6, 609–627.

REISS, S. P. 1983. Generation of compiler symbol processing mechanisms from specifications. *ACM Trans. Program. Lang. Syst 5*, 2, 127–163.

REPS, T. W. AND TEITELBAUM, T. 1989. *The Synthesizer Generator—A System for Constructing Language-Based Editors*. Springer-Verlag, New York.

RÄIHÄ, K.-J., SAARINEN, M., SARJAKOSKI, M , SIPPU, S., SOISALON-SOININEN, E., AND TIENARI, M. 1983. Revised report on the compiler writing system HLP78. (Helsinki Language Processor). Rep A-1983-1, Dept of Computer Science, Univ of Helsinki.

RIDJANOVIC, D. AND BRODIE, M. L. 1982. Defining database dynamics with attribute grammars. *Inf. Process. Lett. 14*, 3, 132–138.

SATALURI, S. R. 1988 Generalizing semantic rules of attribute grammars using logic programs. Ph.D. thesis, Univ. of Iowa, Ames, Iowa.

SESHADRI, V. AND WORTMAN, D. B. 1991. An investigation into concurrent semantic analysis. *Softw. Pract. Exp. 21*, 12, 1323–1348

SETHI, R. 1989. *Programming Languages—Concepts and Constructs*. Addison-Wesley, Reading, Mass.

SHAW, M. 1990. Prospects for an engineering discipline of software. *IEEE Softw. 7*, 6, 15–24.

SHINODA, Y. AND KATAYAMA, T. 1990. Object-oriented extension of attribute grammars and its implementation. In *Proceedings of the International Conference on Attribute Grammars and their Applications*. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 177–191.

SHINODA, Y. AND KATAYAMA, T 1988. Attribute grammar based programming and its environment. In *Proceedings of the 21st Hawaii International Conference on System Sciences*. IEEE Computer Society Press, 612–620.

STEEL, T B., JR. (ED). 1966. Formal language description languages for computer programming. In *Proceedings of the IFIP Working Conference on Formal Language Description Languages*. North-Holland, Amsterdam.

TARHIO, J. 1989. A compiler generator for attribute evaluation during LR parsing. In *Proceedings of the 2nd International Workshop on Compiler Compilers and High-Speed Compilation* Lecture Notes in Computer Science, vol. 371, D. Hammer, Ed. Springer-Verlag, New York, 146–159.

TOCZKI, J., GYIMÓTHY, T., HORVÁTH, T., AND KOCSIS, F. 1989. Generating modular compilers in PROF-LP In *Proceedings of the Workshop on Compiler Compiler and High Speed Compilation. Berlin (GDR)* Rep. 3/89, Akademie der Wissenschaften der DDR, 156–166.

TRAHANIAS, P. AND SKORDALAKIS, E. 1990 Syntactic pattern recognition of the ECG *IEEE Trans Patt. Anal. Machine Intell. 12*, 7, 648–657.

TSAI, W. AND FU, K. S. 1980 Attributed grammar—A tool for combining syntactic and statistical approaches to pattern recognition. *IEEE Trans Syst Man Cybernet. 10*, 873–885.

UHL, J., DROSSOPOULOU, S., PERSCH, G., GOOS, G., DAUSMANN, M., WINTERSTEIN, G , AND

KIRCHGÄSSNER, W. 1982. *An Attribute Grammar for the Semantic Analysis of Ada*. Lecture Notes in Computer Science, vol. 139. Springer-Verlag, New York.

WAITE, W. M. 1990. Use of attribute grammars in compiler construction. In *Proceedings of the International Conference on Attribute Grammars and their Applications*. Lecture Notes in Computer Science, vol. 461. Springer-Verlag, New York, 255–266.

WAITE, W. M. 1986. Generator for attributed grammars—Abstract data type. Arbeitspapiere der GMD 219, Gesellschaft für Mathematik und Datenverarbeitung mbH.

VAN DE BURGT, S. P. AND TILANUS, P. A. J. 1989. Attributed ASN.1. In *Proceedings of the 2nd International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*. North-Holland, Amsterdam, 298–310.

WARREN, D. H. D. 1980. Logic programming and compiler writing. *Softw. Pract. Exp. 10*, 2, 97–125.

WASSERMAN, A. I. (ED.). 1980. Tutorial on programming language design. In *Proceedings of IEEE COMPSAC '80*. IEEE Computer Society Press, Los Alamitos, Calif.

WATT, D. A. 1991. *Programming Language Syntax and Semantics*. Prentice-Hall, Englewood Cliffs, N.J.

WATT, D. A. 1990. *Programming Language Concepts and Paradigms*. Prentice-Hall, Englewood Cliffs, N.J.

WATT, D. A. 1985. Modular description of programming languages. Rep. A-81-734, Computer Science Division-EECS, Univ. of California, Berkeley, Calif.

WATT, D. A. 1984. Executable semantic descriptions. *Softw. Pract. Exp. 14*, 1, 13–43.

WILHELM, R. 1982. LL- and LR-attributed grammars. In *Programmiersprachen und Programmentwicklung, 7*. Fachtagung, H. Wössner, Ed. Informatik-Fachberichte. Springer-Verlag, Berlin, 151–164.

VOGT, H. H., SWIERSTRA, S. D., AND KUIPER, M. F. 1989. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 24*, 7, 131–145.

VORTHMANN, S. A. AND LEBLANC, R. J. 1988. A naming specification language for syntax-directed editors. In *Proceedings of the IEEE Conference on Computer Languages* (Miami Beach, Fla.). IEEE, New York, 250–257

YELLIN, D. M. 1988. *Attribute Grammar Inversion and Source-to-Source Translation*. Lecture Notes in Computer Science, vol. 302. Springer-Verlag, New York.

ZARING, A. K. 1990. Parallel evaluation in attribute grammar-based systems. Ph.D. thesis, Tech. Rep. 90-1149, Dept. of Computer Science, Cornell University, Ithaca, N.Y.

Received June 1992; final revision accepted September 1994.