Object-Oriented Attribute Grammars

J. Grosch

DR. JOSEF GROSCH

COCOLAB - DATENVERARBEITUNG

GERMANY

Cocktail

Toolbox for Compiler Construction

Object-Oriented Attribute Grammars

Josef Grosch

Aug. 27, 1990

Document No. 23

Dr. Josef Grosch
CoCoLab - Datenverarbeitung
Breslauer Str. 64c
76139 Karlsruhe
Germany

Phone: +49-721-91537544
Fax: +49-721-91537543
Email: grosch@cocolab.com

Object-Oriented Attribute Grammars

Abstract This paper introduces object-oriented attribute grammars. These can be characterized as a notation for all classes of attribute grammars. Based on a subtype relation between grammar rules, inheritance of attributes and attribute computations are defined. With this approach, attributes local to grammar rules and the elimination of chain rules are possible without any special constructs. We present object-oriented attribute grammars by a formal definition and by a few typical examples. They are compared to the concepts of related areas. We conclude by sketching an implementation of object-oriented attribute grammars as specification language of an attribute evaluator generator called *Ag* which processes ordered attribute grammars (OAGs) and higher order attribute grammars (HAGs). A first realistic application showed that the generated attribute evaluators are very efficient and can be used in production quality systems.

1. Introduction

This paper defines a notation for attribute grammars called object-oriented attribute grammars. The notation can be used to describe concrete syntax, abstract syntax, and attribute grammars in a uniform way. Our original research goal was to promote the acceptance of attribute grammars by improving the specification language and by generating evaluators that are as efficient as possible.

Many existing attribute grammar systems [DJL88] are based on the concrete syntax of the source language. It has several advantages to base attribute grammars on the abstract syntax, however. Most of the terminal symbols and chain rules disappear. This makes the specification shorter and clearer. The abstract syntax can be optimized towards the task of semantic analysis. Usually, this leads to fewer grammar rules, fewer attribute computations, and fewer nodes in the structure tree. The result is a simplification of the attribute grammar as well as a reduction of space and run time because less nodes have to be stored and visited during attribute evaluation. Therefore efficiency is increased. In this paper we assume that attribute evaluation is performed on the basis of an attributed tree which is stored in memory.

The roots for our definition of object-oriented attribute grammars can already be found in the implementation of current attribute grammar systems. In attribute grammars, nonterminals can be associated with a set of attributes. There may be several grammar rules having this nonterminal as left-hand side symbol. The implementation of attributed trees uses a separate node type for every grammar rule. Now all node types for one nonterminal have to store the attributes of this nonterminal. This leads to the inheritance of information from a nonterminal to the associated rules. Object-oriented attribute grammars generalize this observation. Besides attributes, right-hand sides and attribute computations can be inherited as well. Inheritance is extended from one level (from nonterminals to rules) to arbitrary many levels. Attributes local to a rule can be introduced without any special construct.

The rest of this paper is organized as follows: The next section defines object-oriented attribute grammars formally. Section 3 presents examples of object-oriented attribute grammars using the specification language of an attribute grammar system. Section 4 compares object-oriented attribute grammars to the concepts of conventional attribute grammars, trees, types, and object-oriented programming. Section 5 shortly sketches an implementation of object-oriented attribute grammars and reports early experiences from a non-trivial application project.

2. Definition

This section defines the principles of object-oriented attribute grammars. As starting point we shortly recall the traditional definition of attribute grammars [Knu68, Knu71].

An attribute grammar is an extension of a context-free grammar. A context-free grammar is denoted by G = (N, T, P, Z) where N is the set of nonterminals, T is the set of terminals, P is the set of productions, and $Z \in N$ is the *start* symbol, which cannot appear on the right-hand side of any production in P. The set $V = N \cup T$ is called the vocabulary. Each production $p \in P$ has the form $p\colon X \rightarrow \alpha$ where $X \in N$ and $\alpha \in V^*$. The relation $\Rightarrow$ (directly derives) is defined over strings in V^* as follows: if $p\colon X \rightarrow \alpha$, $p \in P$, $\nu X \omega \in V^*$, $\nu \alpha \omega \in V^*$ then $\nu X \omega \Rightarrow \nu \alpha \omega$. The relation $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$. The language L(G) is defined as $L(G) = \{ w \mid Z \Rightarrow^* w \}$.

An attribute grammar augments a context-free grammar by attributes and attribute computations. A set of attributes is associated with each symbol in V. Attribute computations are added to each production describing how to compute attribute values. This simple view of attribute grammars shall suffice for the scope of this paper.

In general there can be several productions having the same nonterminal on the left-hand side. This allows for different derivations starting from one nonterminal. In object-oriented attribute grammars, one production is permitted for one left-hand side symbol, only. This way the notions production and nonterminal (vocabulary respectively) become the same and are called *node type* in the following. Several different derivations are made possible through the newly introduced subtype relation.

An object-oriented attribute grammar is formally denoted by G = (N, T, A, C, Z) where N is the set of nonterminals, T is the set of terminals, A is the set of attributes, C is the set of attribute computations, and Z is the start symbol ($Z \in N$). The set $NT = N \cup T$ is called the set of *node types*. Each element $n \in NT$ is associated with a tuple n: (R, B, D, S) where $R \in NT^*$ is the right-hand side, $B \in A^*$ is the set of attributes, $D \in C^*$ is the set of attribute computations, and $S \in NT$ is the base type.

The elements of NT induce a relation $\subseteq$ (subtype) over NT as follows: if $n\colon (\alpha, \beta, \delta, m) \in NT$ then $n \subseteq m$. m is called *base* or *super* type, n is called *derived* type or *subtype*. The relation $\subseteq$ is transitive: if $n \subseteq m$ and $m \subseteq o$ then $n \subseteq o$.

The relation $\Rightarrow$ (directly derives) is defined here only for the context-free part of an object-oriented attribute grammar. There are two possibilities for derivations which are defined over strings in NT^* as follows:

$$\begin{aligned}
\text{if } \nu n_i \omega \in NT^* \text{ and } & n_1\colon (\alpha_1, \beta_1, \delta_1, n_0) \in NT,\\
& n_2\colon (\alpha_2, \beta_2, \delta_2, n_1) \in NT,\\
& \quad \cdots\\
& n_i\colon (\alpha_i, \beta_i, \delta_i, n_{i-1}) \in NT \text{ then } \nu n_i \omega \Rightarrow \nu \alpha_1 \alpha_2 \cdots \alpha_i \omega.
\end{aligned}$$

$$\text{if } \nu n \omega \in NT^* \text{ and } m \subseteq n \text{ then } \nu n \omega \Rightarrow \nu m \omega.$$

We assume the existence of a predefined node type $n_0\colon (\emptyset, \emptyset, \emptyset, -)$ with empty components. In a direct derivation step, a node type can be replaced by its right-hand side ($\alpha_1 \cdots \alpha_i$) or by one of its subtypes (m). All replacing right-hand sides are the union of right-hand sides according to the subtype hierarchy. The relation $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$. The language L(G) is defined as $L(G) = \{ w \mid Z \Rightarrow^* w \}$.

The subtype relation has the following properties: a derived node type inherits the right-hand side, the attributes, and the attribute computations from its base type. As consequence of the transitive nature of this relation, a derived type inherits all the components from all base types according to the subtype hierarchy. It may extend the set of inherited items by defining additional right-hand side elements, attributes, or attribute computations. All accumulated right-hand side elements and attributes must be distinct because they are united. An attribute computation for an attribute may overwrite an inherited one. The definition of the subtype relation allows exactly single inheritance.

3. Examples

We implemented an attribute grammar system called *Ag* based on object-oriented attribute grammars [GrE90]. The following examples of object-oriented attribute grammars are given in the specification language of *Ag*. The language tries to adhere to the conventional style of grammars as far as possible. It offers far more features for practical usage than can be explained here. The interested reader is referred to the user's manual [Groa].

Example 1:

```
Expr        = <
   Add      = Lop: Expr '+' Rop: Expr .
   Sub      = Lop: Expr '-' Rop: Expr .
   Const    = Integer .
> .
Integer     : .
'+'         : .
'-'         : .
```

Example 1 describes the concrete syntax of primitive expressions. It defines four nonterminal node types (Expr, Add, Sub, and Const) and 3 terminal node types (Integer, '+', and '-'). The distinction between nonterminals and terminals is based on the characters '=' and ':'. Terminal node types without attributes do not have to be defined explicitly because undefined node types are defined implicitly as terminals. Node types can be named by identifiers or strings. In this example only the right-hand side components and the subtype relation are used - attributes and attribute computations do not appear. Expr has zero, Add and Sub have three, and Const has one right-hand side element(s) or children. A child consists of a selector name and node type. Selector names are added to allow unambiguous access to children. Missing selector names are implicitly defined to be equal to the name of the node type. The children of the node type Add have the selector names Lop, '+', and Rop. Their node types are Expr, '+', and Expr. The node type Const has one child with the selector name Integer of type Integer. The subtype relation is expressed by enclosing all subtypes of a base type in < > brackets. In Example 1 the subtype relation is: Add $\subseteq$ Expr, Sub $\subseteq$ Expr, Const $\subseteq$ Expr.

Example 2:

```
Expr        = [Value: INTEGER]        { Value := 0; } <
   Add      = Lop: Expr '+' Rop: Expr { Value := Lop:Value + Rop:Value; } .
   Sub      = Lop: Expr '+' Rop: Expr { Value := Lop:Value - Rop:Value; } .
   Const    = Integer                 { Value := Integer:Value; } .
   Zero     = .
> .
Integer     : [Value: INTEGER] .
```

Example 2 adds attributes and attribute computations to Example 1 and describes the evaluation of expressions. Attribute definitions are in enclosed in [ ] brackets. Similarly to children, attributes are characterized by a selector name and a certain type. The attribute types are given by names taken from the target language (here Modula-2).

The node types Expr and Integer define one attribute named Value of type INTEGER. The subtypes Add, Sub, Const, and Zero inherit the attribute Value from Expr. Attribute computations are mainly written as assignments of target language expressions to attributes and are enclosed in { } brackets. The attribute computations are expressed with respect to a current tree node and may contain so-called *attribute denotations*. At a tree node, the attributes of this node and the attributes of the children are accessible. Attributes of the current node or of the left-hand side of a rule are denoted just by their name. Attributes of a child or of the right-hand side of a rule are denoted by the child's selector name, a colon, and the attribute name. The subtype Zero inherits the computation of the attribute Value from the base type Expr, whereas the node types Add, Sub, and Const overwrite it with node type specific computations. The value for the attribute of the node type Integer has to be provided at the creation time of the syntax tree e. g. from a scanner and parser.

A node of a base type like *Expr* usually does not occur in an abstract syntax tree for a complete program. However, this node type can be used as placeholder for unexpanded nonterminals in incomplete programs which occur in applications like syntax directed editors.

Example 3:

```
Stats       = <
   NoStat   = .
   OneStat  = Stats Stat .
> .
Stat        = [Pos: tPosition] <
   If       = Expr Then: Stats Else: Stats .
   While    = Expr Stats .
   Call     = Actuals [Ident: tIdent] .
> .
```

Example 4:

```
Stats       = <
   NoStat   = .
   Stat     = Next: Stats [Pos: tPosition] <
      If    = Expr Then: Stats Else: Stats .
      While = Expr Stats .
      Call  = Actuals [Ident: tIdent] .
   > .
> .
```

Examples 3 and 4 describe two possibilities for the specification of the abstract syntax of statement sequences. Both examples use two nonterminals to describe a sequence (Stats) and various statements (Stat). Whereas these nonterminals are independent in Example 3 they are related as subtypes in Example 4. Therefore Example 4 shows a non-trivial subtype relation of nesting depth two. The subtype relation is: NoStat $\subseteq$ Stats, Stat $\subseteq$ Stats, If $\subseteq$ Stat, While $\subseteq$ Stat, Call $\subseteq$ Stat. In Example 4 the node types If, While, and Call inherit the child Next of type Stats and the attribute Pos from the base type Stat. They add their own children and attributes (Call only). The big difference between the two solutions arises with regard to efficiency. Example 3 allocates in the structure tree two nodes per statement in a sequence. These nodes need space and have to be traversed (visited) during attribute evaluation. Example 4 allocates only one node per statement thus saving both, space and attribute evaluation time. The price is the additional child called Next in every node type for a statement. Nevertheless, it has to be defined only once because of the inheritance mechanism.

The node type Call uses a local attribute called Ident. There is no need for the description of a procedure call statement to have two children: the name of the procedure and the list of actual parameters. The name of the procedure can become a local attribute of the node type and therefore one child for the parameters suffices.

The specification of node types can be grouped into modules. This feature can be used to structure a specification or to extend an existing one. If a node type has already been declared the given children, attributes, attribute computations, and subtypes are added to the existing declaration. Otherwise a new node type is introduced. This way of modularization offers several possibilities:

- Context-free grammar and attribute declarations (= node types) as well as attribute computations can be combined in one module as in conventional, monolithic attribute grammars.
- The context-free grammar, the attribute declarations, and the attribute computations can be placed in three separate modules.
- The attribute computations can be subdivided into several modules according to the tasks of semantic analysis. For example, there would be modules for scope handling, type determination, and context conditions.
- The information can be grouped according to language concepts or nonterminals. For example, there would be modules containing the grammar rules, the attribute declarations, and the attribute computations for declarations, statements, and expressions.

Example:

```
MODULE my_version

Stats        = [Env: tEnv] <                    /* add attribute   */
   While     = Init: Stats Terminate: Stats .   /* add children    */
   Repeat    = Stats Expr .                     /* add node type   */
> .

Zero         = { Value := 1; }                  /* add computation */

END my_version
```

4. Comparison

This section compares object-oriented attribute grammars as introduced in this paper with the well-known concepts of (attribute) grammars, tree and record types, type extensions, and object-oriented programming. These areas are related because of the following reasons: attribute grammars are usually based on context-free grammars. An attribute grammar specifies an evaluation of attributes of a tree defined by such a context-free grammar. Trees can be implemented using a set of record type declarations. Therefore context-free grammars, trees, and record types deal more or less with the same concept. Table 1 compares the most important notions from these areas. Additionally we included the notions from the area of object-oriented (oo) programming as described e. g. in [Bla89].

(attribute) grammars	trees	types	oo-programming
rule	node type	record type	class
attribute	field in a node type	record field	instance variable
nonterminal	set of node types	union of record types	-
terminal	distinct node type	record type without pointer fields	-
rule application	tree node	record variable	object, instance
attribute computation	-	procedure declaration	method
-	-	procedure call	message
-	-	base type	superclass
-	-	derived type	subclass
-	-	extension	inheritance

Table 1: Comparison of notions from the areas of grammars, trees, types, and oo-programming

Object-oriented attribute grammars are missing in Table 1. For them we used the notions from attribute grammars and added the notions node type, base type, subtype or derived type, and inheritance from the other areas.

4.1. Attribute Grammars

Conventional grammars in BNF allow several productions with the same nonterminal symbol on the left-hand side. A node type in object-oriented attribute grammars, which corresponds to a nonterminal as well as to a rule name, has exactly one right-hand side. The selector names can be regarded as syntactic sugar. To allow for several different derivations, a subtype relation between node types is added. During a derivation, a node type may be replaced by its right-hand side or by a subtype. Additionally, the subtype feature allows to express chain rules and single inheritance. Inheritance is a notation to factor out parts that are common to several node types such as right-hand sides, attributes, and attribute computations. Fortunately, attributes local to a rule (node type) are possible without any special construct.

Object-oriented attribute grammars are a notation to write BNF grammars in a short and concise way and where the underlying tree structure can be exactly described. With respect to attribute grammars the same notational advantages hold. Attribute grammars are a special case of object-oriented attribute grammars. They are characterized by a one level subtype hierarchy, right-hand sides and attribute computations are defined for subtypes only, and attributes are associated only with base types. In terms of attribute grammar classes or attribute grammar semantics object-oriented attribute grammars are equivalent to attribute grammars.

4.2. Trees and Records

When trees are stored in memory, they can be represented by linked records. Every node type corresponds to a record type. Object-oriented attribute grammars directly describe the structure of attributed syntax trees. The node types can be seen as record types. The right-hand side elements resemble pointer valued fields describing the tree structure and the attributes are additional fields for arbitrary information stored at tree nodes. The field name and field type needed for record types are also present in the node types of object-oriented attribute grammars.

4.3. Type Extensions

Type extensions have been introduced with the language Oberon by Wirth [Wir88a, Wir88b, Wir88c]. They allow the definition of a record type based on an existing record type by adding record fields. This extension mechanism induces a subtype relation between record types. The subtype and inheritance features are equivalent in object-oriented attribute grammars and type extensions with the difference that Wirth uses the word extension in place of inheritance.

4.4. Object-Oriented Programming

The concepts of subtype and inheritance in object-oriented attribute grammars and object-oriented programming have many similarities and this explains the name object-oriented attribute grammars. The notions class, instance variable, object, superclass, and subclass have direct counter parts (see Table 1). There are also some differences. Object-oriented programming allows an arbitrary number of named methods which are activated by explicitly sending messages. In object-oriented attribute grammars there is exactly one attribute computation (unnamed method) for an attribute. There is nothing like messages: the attribute computation for an attribute is activated implicitly and exactly once.

5. Experiences

As already mentioned, we implemented an attribute evaluator generator called *Ag* which processes object-oriented attribute grammars. It accepts *ordered attribute grammars* (OAGs) [Kas80] and *higher order attribute grammars* (HAGs) [VSK89, Vog93] which are a reinvention of *generative attribute grammars* [Den84]. *Ag* is written in Modula-2 and runs under UNIX. The sources for *Ag* also exist in C. Attribute evaluators in the target languages Modula-2 and C are supported. We tried to avoid the disadvantages of closed and complex systems. Instead, we had the goal of keeping everything as small, simple, open, powerful, efficient, practical usable, and language independent as possible. The specification language has a concise and clear concrete syntax to support compact attribute grammars which are easy to write and to read. The attribute grammars are oriented towards abstract syntax what is a great simplification in comparison to the use of concrete syntax. Readability of attribute grammars is furthermore supported by modules where the context-free grammar (abstract syntax) is specified only once in order to assure consistency. The attributes are typed using the types of the target language - tree-valued attributes are possible. The attribute computations are programmed in the target language and should be written in a functional style. External functions can be called and non-functional statements as well as side-effects are possible. These features make attribute grammars open and allow the attributed trees as well as the attribute evaluators to be implemented efficiently.

Using the target language for attribute computations makes the generator largely target language independent because there is no need to analyze a special language for attribute computations and to generate code for it. This job is already performed by usual compilers. Therefore *Ag* is a relatively small and simple program which concentrates on the analysis of object-oriented attribute grammars and attribute dependencies. It generates evaluators for (OAGs) and (HAGs). The generated attribute evaluators are very efficient because they are directly coded using recursive procedures.

A large application of object-oriented attribute grammars and *Ag* was the generation of the semantic analysis phase of a Modula-2 to C translator [Mar90]. The program called *Mtc* translates Modula-2 programs into readable C code without any restrictions (even nested procedures and modules). This program was largely generated using our compiler construction tool box [GrE90]. Table 2 gives the sizes of the specifications and the generated source modules.

part	specification			source module			tool	
	formal	code	total	def.	impl.	total	name	references
scanner	392	133	525	56	1320	1376	Rex	[Gro88, Grob]
parser	951	88	1039	81	3007	3088	Ell	[Gro88, GrV]
tree	189	51	240	579	2992	3571	Ast	[Groa]
symbol table	115	938	1053	413	1475	1888	Ast	[Groa]
semantics	1886	151	2037	9	3288	3297	Ag	[Groa]
code generator	2793	969	3762	47	7309	7356	Estra	[Vie89]
reusable parts	-	-	-	819	2722	3541	Reuse	[Grob, Groc]
miscellaneous	-	-	-	698	3153	3851		
total	6326	2330	8656	2702	25266	27968		

Table 2: Sizes of the specifications and source modules of *Mtc*

The binary program comprises 301,240 bytes. It runs at a speed of 810 tokens per second or 167 lines per second on a SUN workstation (MC 68020 processor). These figures are computed by taking only the size of the translated modules into account. If we include the definition modules which are imported transitively and which are scanned, parsed, and analyzed as well, we arrive at 1320 tokens per second or 418 lines per second. In comparison, the compilers supplied by the manufacturer run at a speed of 97 lines per second (excluding header files) or 163 lines per second (including header files) in the case of C and 48 lines per second in the case of Modula-2. The run time of *Mtc* is distributed as follows:

scanning + parsing + tree construction	42 %
semantic analysis	33 %
code generation	25 %

The semantic analysis spends 95 % in attribute computations using user supplied code and 5 % in tree traversal or visit actions respectively. By the way, there are up to five visits to 11 node types.

Mtc uses approximately 360 K Bytes dynamic memory per 1000 source lines to store the abstract syntax tree, the attributes, and the symbol table without optimization of attribute storage. Another 600 K Bytes are used by the transformer generated with *Estra*. This is bearable with today’s memory capacities. Contrary to the literature this shows that it is possible to store all attributes in the tree. We even do this for the environment attribute. This becomes possible by implementing the symbol table as an abstract data type in the target language. The implementation is time and space efficient by taking advantage of pointer semantics and structure sharing.

These results demonstrate the time efficiency of the generated attribute evaluators in the semantic analysis of non-trivial languages like Modula-2. The performance makes (object-oriented) attribute grammars and *Ag* usable for production quality systems.

6. Summary

We introduced object-oriented attribute grammars as a common notation for concrete syntax, abstract syntax, and attribute grammars. The subtype and inheritance features represent a shorthand notation for attribute grammars. The advantages of object-oriented attribute grammars become primarily apparent when used in practice as specification language for an attribute grammar system.

They allow the exact description of the underlying tree structure in order to create compact and storage efficient trees. Right-hand side elements, attributes, and attribute computations common to several rules can be factored out. They allow the expression of chain rules and the definition of attributes local to rules. We compared object-oriented attribute grammars to similar concepts such as conventional attribute grammars, trees, types, and object-oriented programming. An attribute evaluator generator system for object-oriented attribute grammars has been implemented. The generated attribute evaluators are very efficient especially in terms of run time. We reported early experiences from a non-trivial application which are quite satisfying.

References

[Bla89] G. Blaschek, Implementation of Objects in Modula-2, *Structured Programming 10,* 3 (1989), 147-155.

[Den84] P. Dencker, Generative attributierte Grammatiken, Dissertation, Universität Karlsruhe, 1984.

[DJL88] P. Deransart, M. Jourdan and B. Lorho, Attribute Grammars - Definitions, Systems and Bibliography, *LNCS 323,* (1988), , Springer Verlag.

[Gro88] J. Grosch, Generators for High-Speed Front-Ends, *LNCS 371*, (Oct. 1988), 81-92, Springer Verlag.

[GrE90] J. Grosch and H. Emmelmann, A Tool Box for Compiler Construction, *LNCS 477*, (Oct. 1990), 106-116, Springer Verlag.

[Groa] J. Grosch, Ag - An Attribute Evaluator Generator, Cocktail Document No. 16, CoCoLab Germany.

[Grob] J. Grosch, Rex - A Scanner Generator, Cocktail Document No. 5, CoCoLab Germany.

[GrV] J. Grosch and B. Vielsack, The Parser Generators Lalr and Ell, Cocktail Document No. 8, CoCoLab Germany.

[Groa] J. Grosch, Ast - A Generator for Abstract Syntax Trees (Revised Version), Cocktail Document No. 15, CoCoLab Germany.

[Grob] J. Grosch, Reusable Software - A Collection of Modula-Modules, Cocktail Document No. 4, CoCoLab Germany.

[Groc] J. Grosch, Reusable Software - A Collection of C-Modules, Cocktail Document No. 30, CoCoLab Germany.

[Kas80] U. Kastens, Ordered Attribute Grammars, *Acta Inf. 13,* 3 (1980), 229-256.

[Knu68] D. E. Knuth, Semantics of Context-Free Languages, *Mathematical Systems Theory 2,* 2 (June 1968), 127-146.

[Knu71] D. E. Knuth, Semantics of Context-free Languages: Correction, *Mathematical Systems Theory 5*, (Mar. 1971), 95-96.

[Mar90] M. Martin, Entwurf und Implementierung eines Übersetzers von Modula-2 nach C, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, Feb. 1990.

[Vie89] B. Vielsack, Spezifikation und Implementierung der Transformation attributierter Bäume, Diplomarbeit, GMD Forschungsstelle an der Universität Karlsruhe, June 1989.

[VSK89] H. H. Vogt, S. D. Swierstra and M. F. Kuiper, Higher Order Attribute Grammars, *SIGPLAN Notices 24*, 7 (July 1989), 131-145.

[Vog93] H. H. Vogt, *Higher Order Attribute Grammars*, PhD Thesis, University of Utrecht, Feb. 1993.

[Wir88a] N. Wirth, Type Extensions, *ACM Trans. Prog. Lang. and Systems 10*, 2 (Apr. 1988), 204-214.

[Wir88b] N. Wirth, From Modula to Oberon, *Software—Practice & Experience 18,* 7 (July 1988), 661-670.

[Wir88c] N. Wirth, The Programming Language Oberon, *Software—Practice & Experience 18*, 7 (July 1988), 671-690.

Contents