Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

141

Uwe Kastens
Brigitte Hutt
Erich Zimmermann

GAG: A Practical Compiler Generator

Springer-Verlag
Berlin Heidelberg New York

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

141

Uwe Kastens
Brigitte Hutt
Erich Zimmermann

GAG: A Practical Compiler Generator

Springer-Verlag
Berlin Heidelberg New York 1982

Authors
Uwe Kastens
Brigitte Hutt
Erich Zimmermann
Institut für Informatik II der Universität Karlsruhe
Postfach 6380, 7500 Karlsruhe 1

CR Subject Classifications (1981): D 2.1, D 3.1, D 3.2, D 3.4, F 4.2

ISBN 3-540-11591-9 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-11591-9 Springer-Verlag New York Heidelberg Berlin

Printing and binding: Beltz Offsetdruck, Hemsbach/Bergstr.
2145/3140-543210

CONTENTS:

Chapter 1: Introduction 1

Chapter 2: The Compiler Generator GAG 5
2.1 Introduction 5
2.2 Attributed Grammars 5
2.3 System Overview 7
2.4 The Generated Compiler 10
2.4.1 Parser Interface 10
2.4.2 The Structure Tree 10
2.4.3 Attribute Evaluation 11
2.4.4 Attribute Types 11
2.4.5 I/O of the Generated Compiler 12
2.5 Dependency Analysis 12
2.6 Attribute Optimization 14
2.7 Compiler Generation 14
2.8 Experience 15
2.8.1 The Processing of Attributed Grammars 15
2.8.2 Generated Compiler Front-ends 16

Chapter 3: ALADIN - A Language for Attributed Definitions 19
3.1 Introduction 19
3.1.1 Notation 20
3.1.2 Basic Symbols 21
3.2 General Structure 22
3.3 Constant Definitions 22
3.4 Type Definitions 22
3.4.1 Enumeration Types 23
3.4.2 Subrange Types 23
3.4.3 Union Types 23
3.4.4 Structure Types 23
3.4.5 Set Types 24
3.4.6 List Types 24
3.5 Symbol and Attribute Definitions 24
3.6 Productions 25
3.7 Semantic Rules 26
3.7.1 Attribute Rules 26
3.7.2 Attribute Names 26
3.7.3 Attribute Transfer 27
3.7.4 Context Conditions 28
3.8 Semantic Expressions 28
3.8.1 Formulas 28
3.8.2 Type and Symbol Tests 29
3.8.3 Remote Attribute Access 30
3.8.4 Let Clauses 31
3.8.5 Operands 31
3.9 Semantic Clauses 32
3.9.1 Type Conversion 32
3.9.2 Calls 33
3.9.3 Case Clauses 33
3.9.4 Conditional Clauses 34
3.9.5 Relational Clauses 34
3.10 Function Definitions 34
3.10.1 Generic Functions 35
3.11 Standard Definitions 35

Chapter 4: Development of an Attributed Grammar for a Pascal-Analyzer 37
4.1 Introduction 37
4.2 Development of the Attributed Grammar 38
4.2.1 Types 39
4.2.2 Scope Rules 43
4.3 Error Handling 46
4.4 Performance and Optimizations 48
4.5 Results of Attribute Evaluation 50

Chapter 5: Generating Efficient Compiler Front-ends 53
5.1 Introduction 53
5.2 Basic Generation Concepts 54
5.2.1 Types in the Generated Compiler 55
5.2.2 The Program Tree Structure 55
5.2.3 Sequence Control of Attribute Evaluators 56
5.3 Improvement of the Tree Representation 58
5.3.1 Tree Compactification 59
5.3.2 Tree Partitioning 60
5.3.3 Attribute Evaluation Without Tree 60
5.4 Attribute Optimization 61
5.5 General Optimization Techniques 64
5.5.1 Common Subexpressions 64
5.5.2 Recursion Elimination 65
5.5.3 Inline Code for Functions 65
5.6 AG Transformations 65
5.7 Application to Pascal 66
5.8 Results 71

Appendix A: Attributed Grammar for Pascal 73

Appendix B: Results of the Usage of GAG 137

References 155

Chapter 1:

Introduction

Modular decomposition of the compilation problem for programming languages leads to the main compiler tasks scanning, parsing, semantic analysis, optimization and code generation. Attributed grammars (AGs) are a well suited method for definition of static context dependent properties of programming languages, thus specifying the semantic analysis phase. For usual high level languages this phase comprises application of scope rules, type checking, overloading resolution, etc. Implementation of semantic analysis can be systematically derived from such a specification: A tree representing the abstract program structure is augmented by attributes specified in the AG. Hence AGs are a widely accepted base for systematic compiler construction. This book shall demonstrate that the AG method is a suitable base for practical compiler generators.

The GAG-System is a compiler Generator based on AGs. Its development was guided by three dominating aims: The system should be usable in practical compiler projects; it should be able to process AGs which are written as formal language definitions rather than as implementation oriented specifications; and the generated attribute evaluators should be efficient.

The GAG-System uses an attribute evaluation technique (OAG technique, see [Ka80] and Sect. 2.2) which is more powerful than the multi-pass techniques of comparable systems. In practice it turned out that this technique releases the user of the system from consideration of attribute evaluation order during AG development. The evaluation order is computed automatically by the GAG-System - even for complex languages like Ada or PEARL. We developed an input language for the system which is suitable for writing AGs as self-contained definitions of static language properties on a high level of abstraction (see Chapter 3). AG development is supported by elaborated system facilities (see Chapter 2 and Appendix B). The generated attribute evaluators can easily be embedded in a complete compiler environment using suitable interfaces and parameterized modules for compiler phases not specified in the AG (e.g. scanner, protocol generator). Efficiency of the generated attribute evaluator is achieved by many effective optimizations of space and runtime which are automatically applied by the GAG-System (see Chapter 5). Portability and maintainability of both the system and its products are achieved by systematic implementation techniques using Standard Pascal.

Attributed grammars were introduced by Knuth [Kn68] as a method for defining semantics of programming languages based upon the following principles (see Sect. 2.2):

> The syntactic structure of sentences in the defined language is described by a context-free grammar. The derivation of a sentence can be represented by a structure tree in which every node stands for a symbol of the context-free grammar's vocabulary.

The AG associates a set of attributes with each symbol in the context-free grammar's vocabulary, and hence with each node in the structure tree of a sentence. The value of an attribute describes a context dependent property of the language element represented by the symbol, e.g. the type of an expression or the set of definitions valid in the particular context.

The AG associates attribute rules with each production of the context-free grammar specifying the computation of attribute values in terms of attributes of surrounding nodes. Application of a production to derive a symbol selects the attribute rules by which the attributes of the corresponding nodes will be computed. The correspondence between attributes and values is static. An attribute of a node of the structure tree can only take on a single value. The attribute rules are static definitions; they should not be thought of as algorithms over variable attributes.

The AG describes a sublanguage of the language defined by the context-free grammar through context conditions that must be satisfied by the attribute values: A syntactically correct sentence belongs to the defined language if and only if the context conditions are satisfied for all attributes of the structure tree.

There are formal conditions on completeness and consistency for well-defined AGs which ensure that the value of each attribute is uniquely determined and effectively computable in any context.

The input language of the GAG-System (ALADIN, see Chapter 3) is based on these principles for AGs. As a consequence of the static character of AGs ALADIN is a strictly applicative language without any control flow elements. Attribute evaluation order is computed automatically by the GAG-System. A powerful type concept allows definition of the attributes' meaning on a high level of abstraction. The strong typing rules of ALADIN and the well-definedness of the AG - both checked by the GAG-System - guarantee a high degree of consistency and completeness of the specification. The input language encourages the writing of comprehensible AGs by naming all defined entities and by powerful abbreviations.

The development of an AG for a programming language can be compared with software specification. Specification tools like AGs, GAG, and ALADIN must be used reasonably by application of suitable systematic methods and standard techniques. In Chapter 4 general development methods and standard description techniques are presented for common properties of high level programming languages, using an AG for Pascal as an example. The complete Pascal-AG is contained in Appendix A. Furthermore, we show how error recovery and code generation (or interfaces to it) can be specified.

In Chapter 5 it is shown that compilers automatically generated from AGs need not be inefficient. Many effective improvements of storage and runtime are applied automatically by the GAG-System. Measurements on the compiler front-end generated from the Pascal-AG demonstrate that space and runtime requirements necessary for practical usage can be achieved.

In several projects we made very encouraging experience with the GAG-System: In the PEARL project it helped to develop a consistent and complete formal definition of the static properties of the large and complex real-time language PEARL. The AG is part of the German standard document [DIN80]. In compiler projects for LIS and Ada the AG served as a specification for the compiler front-end. The use of

the GAG-System enforced consideration of all consequences of the described language properties before implementation. Open questions could be answered and inconsistencies be removed before they caused costly redesign of programs. Due to certain project requirements, the implementation was produced by systematic manual translation. In spite of that, the front-end generated by the GAG-System was used as a valuable tool for testing the specification. In several smaller scientific projects for application languages the GAG-System reduced the costs for compiler development to the costs for the definition of the language by an AG. Performance measurements on automatically generated Pascal analyzers demonstrate that runtime and space requirements close to those of conventional compilers can be achieved. The efficiency will be sufficient for many practical applications.

The subsequent chapters present different aspects of the GAG-System and its use: Chapter 2 gives a system overview. It contains a short introduction to AGs (Sect. 2.2) intended for readers who are not familiar with this method. In Sect 2.8 our experience with practical applications is summarized. Chapter 3 is the reference manual for the input language ALADIN. The general concepts and the notation are introduced in Sect. 3.1. The rest of that chapter can be skipped in the first reading because most of the later examples are self-explanatory. In Chapter 4 we demonstrate how an AG for a programming language (Pascal) is systematically developed. It contains important know-how for AG development, and it shows how the specification tools are used adequately. AGs for similar languages may be directly derived from the Pascal-AG in Appendix A. Chapter 5 describes performance improvements and their results applied by the GAG-System to the generated compilers. Aspects for comparison of the GAG-System with other compiler generating systems are found most of all in Chapter 2 and 5.

Acknowledgements: We are indebted to Prof. G. Goos who initiated the project, established a fruitful working environment within his group, and who made valuable remarks on early drafts of this book. We thank Dr. J. Schauer, Dr. J. Röhrich, and the Ada Implementation Group Karlsruhe for many helpful discussions and for acting as pilot users. We thank Prof. W. M. Waite for the translation of the ALADIN definition. A significant part of the development of the GAG-System was supported by the Deutsche Forschungsgemeinschaft.

Chapter 2:

The Compiler Generator GAG

2.1 Introduction

The GAG-System (Generator based on Attributed Grammars) generates a compiler for a language defined by an attributed grammar (AG). The underlying concepts were first outlined in [Ka76]. The central part of the system is the analysis of attribute dependencies and the generation of the attribute evaluation phase. Interfaces are defined for scanner and parser.

The system processes attributed grammars of type OAG which is a large subclass of well-defined AGs containing all classes of pass-oriented AGs. For each AG individual visit-sequences are computed which control the attribute evaluation in the generated compiler. Thus the structure tree is traversed in an efficient (non pass-oriented) manner. This method was presented in [Ka8Ø].

The input language ALADIN (cf. Chapter 3) defines a notation for AGs suitable for complete descriptions of the static properties of languages. It is based on typed attributes and includes a powerful and flexible type concept. Expressions and recursive functions over attribute types allow precise and complete definitions of attribute values.

2.2 Attributed Grammars

AGs were introduced by Knuth [Kn68] as a method for defining semantics of programming languages. This section gives a brief formal introduction to AGs.

An AG is a 5-tupel (G, A, Val, R, C) based on a reduced context-free grammar G=(N, T, P, S) - the sets of nonterminals, terminals, productions, and the start symbol. The AG associates a set A(X) of attributes to each symbol X in the vocabulary V=N$\bigcup$T of G. We write X.a to indicate that attribute a is an element of A(X). The sets of attributes are disjoint for different symbols. An attribute a can take on any value of a domain Val(a). It represents a specific (context-sensitive) property of the symbol X.

Each node in the structure tree of a sentence in L(G) represents a symbol X in the vocabulary of G. For each attribute of A(X) an attribute value is associated with such a node. These values are defined by attribute rules R(p) associated with a production p in P: XØ::=X1...Xn. Each attribute rule Xi.a:=f(Xj.b,...,Xk.c) defines an attribute Xi.a in terms of attributes Xj.b,...,Xk.c of symbols in the same production.

In addition to attribute rules the AG associates context conditions

C(p) to each production p. A context condition is a relation g(Xi.a,...,Xj.b) over attributes of symbols occurring in p. A context condition of C(p) is fulfilled for a tree node derived by the production p if the relation holds for the corresponding attribute values. A sentence of L(G) is a sentence of L(AG) if and only if no context condition is violated. We subsume attribute rules and context conditions under the general term semantic rules.

The following requirements ensure that each attribute value of any tree node is defined by exactly one attribute rule: An attribute X.a is called synthesized if there is an attribute rule X.a:=f(...) associated with a production of the form X::=w. If such an attribute rule is associated to a production Y::=uXv the attribute is called inherited. Let AS(X) and AI(X) be the sets of synthesized and inherited attributes of X. Then AS(X) and AI(X) must be disjoint and $AS(X) \cup AI(X) = A(X)$. For a production p: X0::=X1...Xn the set of defining occurrences of attributes is

$$AS(X0) \cup AI(X1) \cup AI(X2) \cup \ldots \cup AI(Xn).$$

The set of applied occurrences is

$$AI(X0) \cup AS(X1) \cup AS(X2) \cup \ldots \cup AS(Xn).$$

There is exactly one attribute rule for each defining occurrence in the set R(p) associated with p. R(p) contains no attribute rule for an applied occurrence.

An AG is well-defined if all attribute values of any structure tree are effectively computable. As a consequence the dependencies between attributes of a tree which are established by the attribute rules must be acyclic for the structure tree of any sentence in L(G). Unfortunately, exponential time is required to verify that an AG is well-defined [Ja75], and attribute evaluators which are applicable for any well-defined AG are rather inefficient. Hence subclasses of the class of well-defined AGs are considered for compiler construction.

The partitionable AGs (in [Ka80]: "AGs which can be arranged orderly") form a large subclass of well-defined AGs including all classes based on pass-oriented attribute evaluation (e.g. n left-to-right depth-first passes [Bo76], or n alternating passes [JaW75]). The definition of partitionable AGs is based on a partition of disjoint subsets Ak(X) of each A(X) such that the attribute dependencies allow to evaluate an attribute X.a of a certain tree node before X.b of the same node if X.a is in Ai(X), X.b is in Aj(X), and $i<j$. A partitionable AG is called ordered (OAG) if the partitions are canonical in a certain sense. A complete formal definition and a decision algorithm for OAGs which requires polynomial time is given in [Ka80]. Such an algorithm is implemented in the GAG-System. It analyses attribute dependencies completely at compiler generation time and computes a non pass-oriented attribute evaluation order if the input AG is ordered. A set of "visit-sequences" controls the attribute evaluation in the generated compiler for any structure tree of the defined language:

The attribute evaluation phase in the generated compiler operates on a structure tree built by parsing the input program. Each inner node with its direct descendants represents an application of a production. The associated semantic rules are evaluated during a visit of such a node. A visit-sequence associated with a production defines the execution order for semantic rules and visits of surrounding nodes. Thus a visit-sequence consists of the following operations:

- evaluate an attribute rule
- check a context condition
- visit the i-th descendant
- leave to the ancestor node

Attribute evaluation at any node derived by a production p in any structure tree is controlled by the visit-sequence associated with p.

The computation of the visit-sequences is based on the partitioned attribute sets. The sets Ak(X) contain alternatingly inherited (AIk(X)) or synthesized (ASk(X)) attributes only: AI1(X), AS1(X), AI2(X), AS2(X), ... The precondition for the n-th visit to any node labelled X is the evaluation of all attributes in AIn(X). The postcondition of the visit is the evaluation of all attributes in ASn(X). It holds after the corresponding leave-operation. Thus the attribute partitions are to be considered as context independent interfaces between visit-sequences applied to adjacent tree nodes. The partitions are computed such that no direct or indirect attribute dependency violates that partial evaluation order.

2.3 System Overview

The requirements a compiler generator must meet are manifold: Of course the main task of such system is the generation of an attribute evaluator specified by an AG. As the system is intended for application in realistic compiler projects, it must support AG development by providing helpful information about the AG such as attributes dependencies, errors and other characteristics. The generated attribute evaluator must be embedded in a compiler environment: As the structure tree is the central data structure for the attribute evaluation the system generates of a module for program tree construction; it provides interfaces to compiler modules not generated by the system (scanner, parser, and synthesis phase if not specified by the AG); it offers facilities for testing and for measurements of space and runtime to be integrated into the attribute evaluator. The user can easily control these facilities by specifications added to the AG. The interface to the parser generator used here [De77] can be adapted to the requirements of comparable systems. Applications for usual high level programming languages are supported by standard solutions for scanner and protocol generator. An example for a protocol of a generated compiler is included in Appendix B, Sect. B.4. The error messages inserted in the source text are defined within the underlying AG (as can be seen in Appendix A).

The functions of the GAG-System are shown in Fig. 2.1.

The first phase of the GAG-System is a usual compiler task : The ALADIN-text is scanned and checked for syntactical correctness (by a generated LALR(1)-parser). The output of the parser, a sequence of symbols and reductions, is used to build up the structure tree. The semantic analysis is done by using the same techniques as applied in the generated compiler: the structure tree is traversed for attribute evaluation controlled by visit-sequences (VS). Since the scope rules of ALADIN are very simple the main task of semantic analysis is type checking.

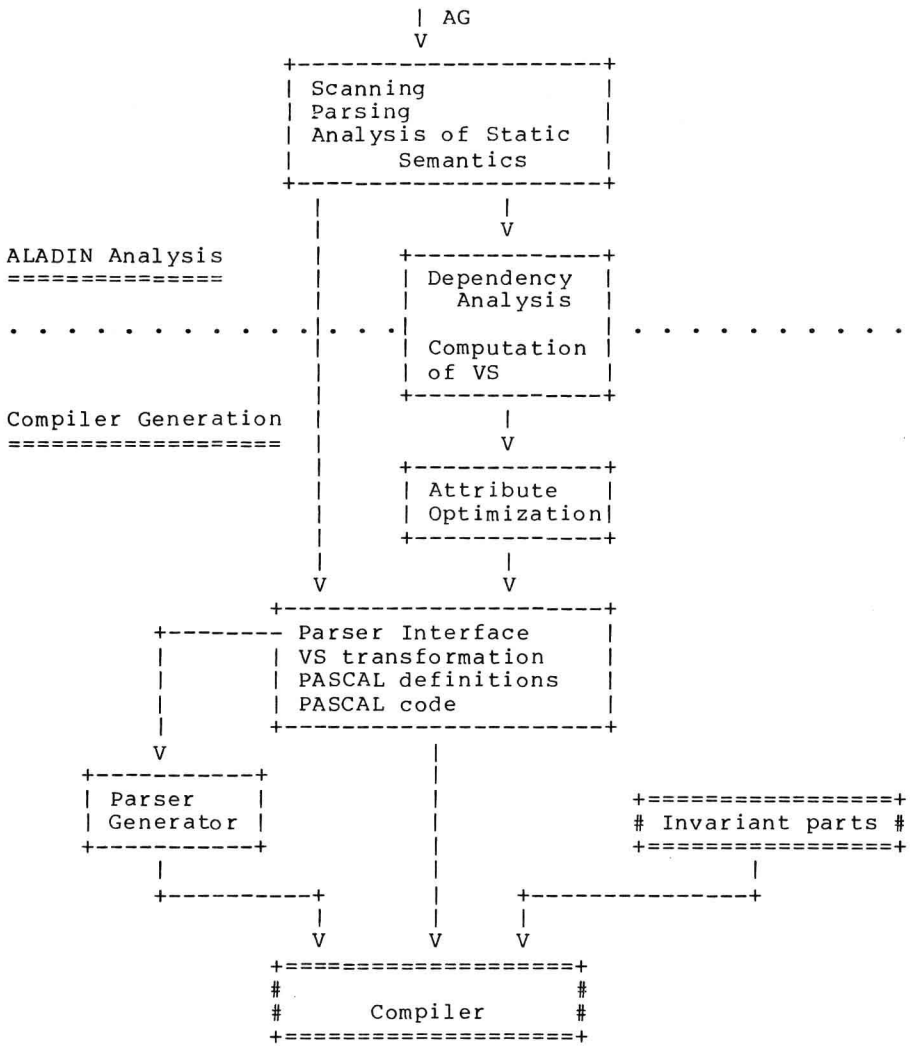


Fig. 2.1: Functions of the GAG-System

The last function of the analysis part has no direct correspondence to general compiler tasks. First the shorthand notations of ALADIN describing attribute transport are expanded. For each shorthand notation a set of equivalent attribute rules is generated. New attributes are introduced for "long range" attribute transport. This expansion is a precondition for the following analysis of attribute dependencies. The dependency analysis checks whether the AG is ordered (or arranged orderly by additional dependencies) and computes visit-sequences using the method described in [Ka80]. These phases

are discussed in Sect. 2.5.

The attribute optimization phase reduces the amount of storage needed for attribute instances in the generated compiler. Lifetime analysis based on the visit-sequences determines which attributes can be implemented by variables or stacks instead of tree node components. Optimization is performed completely at generation time. Its results are reflected by the generation phase.

The main task of the generation phase is the generation of a Pascal program : the compiler specified by the AG. On the one hand the types and objects (attribute values) of ALADIN are mapped to types and declarations in Pascal, on the other hand the attribute computations, conditions, and functions are translated into Pascal statements, functions, and procedures. A further task of the generation phase is the transformation of the visit-sequences into an optimized table which controls the tree-traversal in the generated compiler. In order to use an externally generated parser the generation phase extracts a description of the syntax which serves as input to a parser generating system.

As mentioned above the development of an AG is supported by different available system output: A protocol is generated showing which GAG-passes have been executed and merging the messages (informations, warnings, errors) with the input-text. A cross-reference-listing and a listing of the dependency paths between selected attributes are available. An example of the detailed information produced by the system is shown in Appendix B. Sect. B.1 shows the header of the protocol resulting from a complete execution of GAG.

The implementation of the GAG-System reflects the following program qualities:

a) Modularity. The main phases of the GAG-System form separate programs communicating by files. This principle yields programs of manageable sizes for development and maintenance. Each phase is decomposed into modules according to its logical structure.

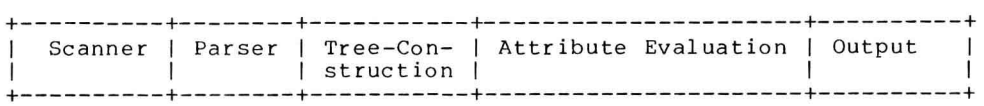
b) Portability. The system is implemented in Pascal as described in [BSI82]. Parts using implementation dependent constructs, e. g. for time measurements, are enclosed within special commands and may be omitted using the preprocessor PROPP [Jan82].

c) Fault tolerance. The system is implemented in a defensive programming style. Assumptions for interfaces between modules and phases are verified on both sides. Sophisticated error recovery is implemented in all phases. Hence the system can handle numerous input and system errors.

The decision for Pascal as implementation language was determined by its high portability. Other languages which support the hierarchical development of large systems better were dropped due to their lack of portability.

2.4 The Generated Compiler

According to the task of a compiler the generated attribute evaluator has the following structure:



In standard applications the structure tree is built up by scanning and parsing an input stream representing a program of the defined language.

Besides the messages arising from violation of context conditions the result of the attribute evaluation is the attributed structure tree. The tree or only some "final" attributes can be written onto a file.

2.4.1 Parser Interface

The attribute evaluation is performed on the structure tree which must be built up according to the context-free syntax contained in any AG. Thus a description of the syntax is produced augmented by actions which control the construction of the abstract program tree. With each rule a call of a tree construction procedure is associated. Its parameters specify the type of the tree node to be generated and the number of subtrees. Additionally a list of those terminal symbols is produced for which a tree node has to be generated (the symbols which are introduced by a terminal definition in the AG).

Any top-down or bottom-up parser performing the specified actions can be integrated into the generated compiler. We use a parser generating system based on the LALR(1)-method [De77].

2.4.2 The Structure Tree

The internal representation of a program is the structure tree. It consists of nodes for nonterminals and terminals and of special nodes for repetition and optional clauses. The attribute evaluation needs certain fields in the node records for nonterminals:
- a pointer to the subtrees,
- an indicator for the applied context-free rule,
- an indicator for the symbol on the lefthand side,
- attribute values of that symbol,
- a position referring to the source text.

The nodes for terminals only contain the symbol indicator and fields for attribute values.

The tree is built up by a set of predefined procedures for allocating and linking the nodes. These procedures are called by the (generated) parser.

2.4.3 Attribute Evaluation

Attribute evaluation is implemented by a stack automaton (as described in Sect. 6.3 of [Ka8Ø]). The transition table contains the encoded visit-sequences. The state of the automaton is a pair made up of the actually visited node in the structure tree and an element of the transition table. A visit of a descendant node is executed by pushing the actual state and computing the new state. For an ancestor visit the new state is popped from the stack. Sequences of semantic rules to be executed successively (not containing a visit) are considered as a basic operation of the algorithm. By that means space for the transition-table and runtime for control operations are reduced to a negligible size.

Context conditions check attribute values. If a condition fails, an error message is written and, in general, attribute evaluation continues. In Chapter 4 we show how semantic error recovery can be integrated in an AG. The violation of certain implicit conditions - such as the call of "HEAD" with an empty list - may result in an undefined value. Only in that case does attribute evaluation stop.

2.4.4 Attribute Types

At first glance, the type concepts of ALADIN and Pascal are rather similar: basic types, subrange, set, and structured types. The main differences are: Instead of the Pascal variant concept ALADIN has UNION types (as in ALGOL68), which are better suited for an applicative language, and ALADIN contains a LIST concept for sequences of equally typed values with powerful list operations. On the one hand there are no pointer types in ALADIN; on the other hand (almost) any recursively defined type is allowed in ALADIN.

The basis of the type mapping of ALADIN into Pascal is the fact that attributes never change their values. So a mapping is chosen which avoids copies of large data structures. Simple types are mapped to their corresponding Pascal type (scalars, subranges, etc.). All types of complex data structures (STRUCT, UNION, LISTOF) are mapped to Pascal types using pointers:

STRUCT, UNION : pointer to a record
LISTOF : the anchor of the list (a record of two pointers to first and last element) and
the elements (a record with the element value and a pointer to the next element).

Thus identical copies of complex objects are implemented by pointer copies: the object itself is allocated only once. Several complex objects which share some components logically, even share them physically as well.

2.4.5 I/O of the Generated Compiler

In standard applications the input to the compiler is the textual form of the program to be translated. The output of the generated attribute evaluator may be the attributed structure tree or (only) some final attributes. Thus the system provides types and procedures for binary I/O which can be used for several purposes:

- The AG specifies a translation into a target (or intermediate) language and the interface is a binary coded file.

- The specification of a language consists of several AGs, each defining a pass of the compilation: The partially attributed program tree is written onto a file and read by the next pass to continue attribute evaluation.

- The specified language comprises facilities for separate compilation: The compiler needs a connection to a library file containing information of units which have already been compiled in the form of attribute values or program trees.

To fulfil all these requirements, the generated program is provided with definitions of intermediate representation for all attribute types and for the program tree, and with generated procedures to read and write these data.

In addition procedures for textual output are generated which are helpful to produce a readable form of attribute-values for testing the AG i.e. for validation of the specification by appropriate input.

2.5 Dependency Analysis

The GAG-System analyses attribute dependencies completely at compiler generation time. If the input AG is ordered "visit-sequences" are constructed which control attribute evaluation in the generated compiler (see Sect. 2.2).

The central data structure for the analysis of attribute dependencies is a collection of dependency graphs: For each symbol a graph over the associated attributes, and for each production a graph over the attribute occurrences of the symbols in the rule. The graphs are iteratively updated: The direct dependencies of the attribute rules are entered in the rule graphs. Any path between two attributes of one symbol occurrence is entered in the corresponding symbol graph and induced to all occurrences of that symbol in the rule graphs. Iteration terminates when all graphs remain invariant. The symbol graphs represent a superset of direct and indirect dependencies between two attributes of any symbol occurrence in any attributed structure tree (cf. [Ka80]).

For each symbol graph the partitions (as described above) are computed and represented by additional ordering dependencies. If an attribute can be assigned to more than one partition, the later evaluated partition is chosen. By that means lifetime of attribute values is shortened ("lazy attribute evaluator" in [Po79], cf. Sect. 2.5). Appendix B, Sect. B.2 contains examples for dependency graphs.

The partially ordered rule graphs augmented by the ordering dependencies are linearized and converted to visit-sequences. Any freedom in the partial order is used for optimizing strategies: