The GAG-System - A Tool for Compiler Construction

U. Kastens
Universität-GH Paderborn

1. Overview

The GAG-System is a compiler generator based on attribute grammars (AGs). From an AG specifying the static properties of a programming language it generates an attribute evaluator implementing the semantic analysis phase of a compiler. A main goal of the development of the GAG-System was its usability as a tool in practical compiler projects.

The system is roughly characterized by the following properties: Attribute evaluation is based on the method of ordered AGs presented in (Kastens 1980): On the one hand it leads to efficient attribute evaluators. On the other hand OAGs are a sufficiently large subset of well-defined AGs such that the designer of an AG usually need not consider restrictions for attribute dependencies, evaluation order, or grouping of attributes to evaluation passes. He can concentrate his attention to the essence of the specification.

The input language of the GAG-System (ALADIN) is a high level applicative specification language especially designed for complete specifications of static language properties. There is no need for writing semantic routines in some host language and "passing them around" the system. High level data types, strong typing, and powerful shorthand denotations allow for comprehensible specifications which are checked for consistency and completeness in numerous aspects.

The attribute evaluators are generated in Standard Pascal (BSI 1982). Space and runtime efficiency is obtained by several optimizing techniques which are automatically applied by the GAG-System.

Embedding of the attribute evaluator in a compiler environment is supported by an interface to a parser (usually generated automatically), by a scanner frame which can easily be adapted to specific requirements, output routines for attributes suitable as interface to the synthesis phase of the compiler, and by a protocol generator.

The GAG-System has been proven to be a valuable tool in several projects: e.g. definition of the German standard for the language PEARL (DIN 1980), development tool for an Ada compiler front-end (Uhl et al.

1982), generation of a translator from Ada to Pascal, and generation of an analyzer for Standard Pascal (Kastens et.al. 1982).

The structure of the GAG-System is comparable with that of a usual compiler for a high level programming language: The analysis phase performs well-known compiler tasks like scanning, parsing, type checking for the specification language ALADIN. Typical for processing attribute grammars is the analysis of attribute dependencies according to the OAG definition and the computation of tables controlling the attribute evaluator (see Sect. 4). Efficiency of the generated compiler is improved in the optimization phase e.g. by space reduction for attributes on the base of lifetime analysis at generation time, by elimination of semantic chain productions, and by transformation of functions with tail recursion. The code generation phase is a translation from the specification language ALADIN to the high level language Pascal. The data types of ALADIN (simple types, sets, records, unions, lists) are mapped to corresponding Pascal types. Since ALADIN is strictly applicative any object of a structured type is implemented by a pointer. That means a drastical space reduction is achieved.

In the following the use of ALADIN for AG development and compiler specification is demonstrated by a small example language (Sect. 2). Its static properties are specified by an AG in the appendix. In Sect. 3 the non pass-oriented strategy for attribute evaluation based on the OAG technique is discussed. Sect. 4 presents the method of life-time analysis for attributes at generation time. A more detailed description of the GAG-System including the definition of ALADIN and a complete AG for Standard Pascal together with a discussion of its design can be found in (Kastens et.al.1982).

2. Development of Compiler Specifications in ALADIN

In this section it shall be demonstrated how the GAG-System with its input language ALADIN is used for compiler specification and generation. For a small, rather artificial example language the steps of AG development and properties of the specification language will be shown.

Before we go into the example some general properties of ALADIN must be mentioned, which strongly influence the character of the resulting specification:

- ALADIN is a complete specification language. It should be used to specify all attribute rules and semantic functions completely, instead of adding functions in some host language.
- ALADIN is a strictly applicative language. One should consider the AG as a specification of language properties, rather than as an algorithm for their computation. The attribute evaluation strategy is powerful enough to disregard evaluation order during the design process.

- ALADIN is strongly typed and provides for high level data types with powerful operations. Hence the attribute types should be chosen such that they describe exactly the language property the attribute stands for.
- ALADIN well suited for the specification of static language properties (e.g. scope rules and type checking). Descriptions of code generation and optimizations are possible but not the typical application (see (Kastens et al. 1982)). Hence we consider the attributed tree as the result of the analysis phase specified by the AG which is transformed by later compiler phases.

The Example Language. In the following a short informal introduction of our example language is given:

The language is block structured and expression oriented. Each block or assignment yields a result and can be part of an expression. For the sake of simplicity control flow statements (except procedure calls) are omitted and the set of operators is restricted. The only types of language objects are int, real, bool, and procedures. The scope rules are taken from Algol: A definition is valid in the whole smallest enclosing block, except all inner blocks with a definition for the same identifier. The type rules allow for widening int values to real values. The operators +, -, * are overloaded for int and real operands and results.

AG Development. A systematic AG development should start with an analysis of the given or intended language and proceed in the following steps:

1) Specification of the context-free grammar
2) Design of attribute types for the description of global language concepts like scope rules and types of objects.
3) Design of attribute rules and context dependent restrictions for each production together with functions for their computation.

In the following we refer to the specification of our example language in the appendix (by identifiers written in *italics*).

The context-free grammar is specified in ALADIN using BNF with extensions for optional parts, iteration (* and +), and iteration with a separator (e.g. see rule *p_2*). Each production is the header of a rule containing attribute rules and context conditions. Alternative productions for the same left-hand side are separated in different rules.

The context-free grammar should speciфy the concrete syntax of the language in order to generate a parser for it automatically by a parser generating system attached to the GAG-System. There is no need for transforming it into an abstract syntax, because the GAG-System automatically eliminates terminals and chain productions which are not needed for attribute evaluation. In some cases it may be advisable to simplify the context-free grammar: Certain language restrictions which can be described context-free are more easily described by context dependent means, especially if properties are involved which are defined by attributes in any way (e.g. restrictions on type denotations).

The second development step is concerned with global language properties like scope rules and type rules which contribute to the attribute rules of many productions. Before all these attribute rules are specified the domains of the corresponding attributes are designed. An important guideline for AG development is: An attribute describes a property of the symbol it is attached to. Hence its domain should be an abstraction of that property. Let us explain that by the concepts of types and scopes for our example language.

Types. In our language we have procedures and objects of simple types *int*, *real*, *bool*. The domain *simple_type* is defined in ALADIN by an enumeration (as in Pascal). It is augmented by an element *no_type* which has no equivalent in our language and is used for error handling avoiding avelanche errors (see function *compatible*). *simple_types* and *proc_types* form the discriminated UNION (in the sense of Algol 68) *mode*. Discrimination of values of the united types is expressed by case expressions as in rule *p*_1. Each *proc_type* is a pair of parameter definitions and a simple result type.

Type attributes are used in two classes of context: Definitions associate a type to an object and expressions and their constituents have a type determined by certain type rules. In our language *type_denoters* are terminals. Their type attribute is determined by the scanner which analyses the key words. The values of this kind of attributes is considered to be predefined for the AG. In the case of procedure declarations the *proc_type* pair is constructed by composition of the *def* attributes of the formal parameters and the type attribute of the result *type_denoter* (see first attribute rule in *p*_5).

A synthesized *type* attribute is associated to expressions and its constituents (*primaries* and *blocks*). Their domain is restricted to *simple_types* because the language neither allows formal procedures nor procedure variables. For each production in expression context an attribute rule determines the type of the left-hand side; e.g. in *p*_2 it is stated that the type of a *block* is the type of the last *expression* (since *expression* is a repeated symbol of the production the term *expression.type* stands for a list of attribute values, where the predefined function *LAST* is applied to). A typical type rule is the operator identification in rule *p*_8: The result type of the formula is determined by the operator symbol and the operands' types. Identification of overloaded operators is specified by the case expression of the function *opr_identify*.

In rules *p*_7 and *p*_9 the shorthand denotation *TRANSFER* specifies that corresponding attributes of the left-hand side and the right-hand side are equal. (Such semantic chain productions are eliminated automatically by the GAG-System.) In the case of procedure calls (rules *p_14*, *p_15*) type checking for actual parameters is specified using an inherited attribute describing the list of the remaining formal parameters. The context conditions require compatibility of actual and formal parameter types and the correct number of actual parameters. For general relations over types like compatibility a function is introduced.

Scopes. The concept of scope is described by a domain being a list of *definitions* together with an identification function *identify*. The abstraction of a declaration associates the defined properties (here a *mode* descriptor) to the introduced identifier. Each symbol representing a scope (*block* in our case) has an environment attribute *env* for the list of local definitions concatenated with the global definitions of the context (see rule *p_2*). A local definition hides a global definition for the same identifier (according to the Algol scope rule), because the identification function finds the local one first in the list.

For each applied occurence of an identifier (rule *p_16*) the function *identify* yields its definition according to the scope rules. In order to avoid inherited environment attributes with equal values for all symbols of a block-body ALADIN supports a shorthand denotation for "remote attribute access": The term *INCLUDING block_env* refers to the environment attribute of the next ancestor node in the structure tree representing a *block* symbol. This feature usually saves many attributes and attribute rules in the AG and in the attributed tree.

An alternative design of the domain for environment attributes could be a list of sections each containing the list of definitions local to one *block* together with an appropiate identification function. Functions like *identify* operating on lists are fomulated recursively in the applicative language ALADIN. In the function *identify* only tail recursion occurs which is transformed into iteration automatically by the GAG-System - yielding a drastic runtime improvement.

In ALADIN generic functions over different list domains can be defined. The specification of a *KEY* component in the domain *definitions* allows for generic functions (like the predefined *UNIQUE_KEYS*) over lists of records with a special "key" component (*id* for *definitions*).

Error handling. In ALADIN context conditions are expressed by boolean expressions over attributes (e.g. see rule *p_2*). If attribute evaluation yields false for such a condition an error message is given indicating that the input is semantically erroneous (the message text can be specified in the AG). For convenience of notation context conditions can be part of expressions (cf. rule *p_11*).

After an error is detected attribute evaluation proceeds always in a consistent state. Since ALADIN is strongly typed and expression oriented.

the language ensures that in any case the evaluation of each expression yields a proper value (provided that recursive function calls terminate, and that the calls of a few predefined functions like *HEAD* are defined). That means the AG designer is enforced to consider all cases of alternative expressions - including those which occur only for erroneous inputs (e.g. in case expressions incomplete case labels must be completed by an *OUT*-clause, see rule *p_11*). Furthermore the AG can be specified such that avelanche errors are reduced in the generated compiler, e.g. by introduction of error values like *no_type*.

3. Attribute Evaluation for Ordered Attribute Grammars

The GAG-System constructs a visit-oriented attribute evaluator for ordered attribute grammars (OAGs) as presented in (Kastens 1980). A comparison with other evaluation methods can be found in the contribution of Engelfriet ("Attribute Evaluation Methods", in this book). In the following we describe the construction of visit-sequences for an OAG.

A visit-oriented attribute evaluator performs a walk through the structure tree where nodes are visited and values of attribute instances are computed. Let the currently visited node v_0 which is derived by a production $p: X_0 \to X_1 \cdots X_n$. Hence v_0 and its descendent nodes $v_1,\dots,v_n$ represent an application of p. Then the evaluator can perform one of the following classes of operations

eval(a) evaluate an attribute instance according to an attribute rule associated to p

visit(X_i,j) visit the i-th descendent node v_i for the j-th time

leave(j) leave the context p for the j-*th* time reaching the ancestor node of v_0.

For the evaluation of OAGs a *visit-sequence* vs_p is associated to each production p. vs_p is a sequence of the above operations which controls the evaluator actions while a node derived by p is visited. The execution of visit-sequences vs_p and vs_q interact for two adjacent contexts p and q in the tree, as shown in Fig. 1.

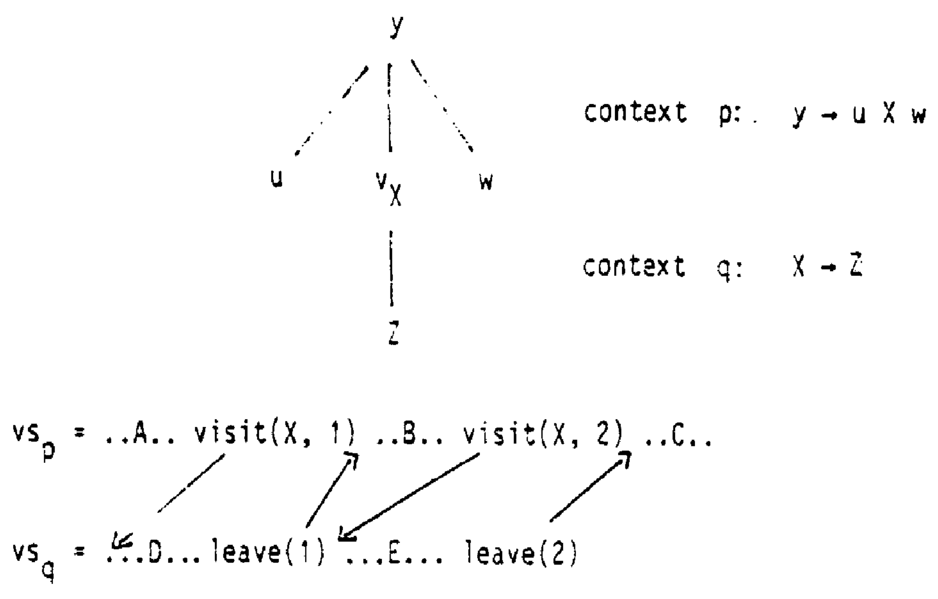
y
u
v_X
w
Z
context p: y → u X w
context q: X → Z
vs_p = ..A.. visit(X, 1) ..B.. visit(X, 2) ..C..
vs_q = ...D... leave(1) ...E... leave(2)

Fig. 1 Interaction of visit-sequences for adjacent contexts

A visit(X, j) in vs_p leads to the beginning of vs_q if $j=1$ or to the operation following a *leave* $(j-1)$ in vs_q otherwise. A *leave* (j) in vs_q leads to the operation following *visit*(X,j) in vs_p. (We assume that the parts A, B and C of vs_p does not contain a *visit*(X,j) and the D, E in vs_q does not contain a *leave*(j).) Obviously the visit-sequences for each pair of productions of the form p and q must "fit together", i.e. the number of corresponding *visit*- and *leave*-operations must be equal; the last element of a visit-sequence must be a *leave*-operation; and a vs_p must contain an *eval*(a)-operation for each attribute rule associated to p.

A correct evaluation order for the attribute instances of any structure tree is obtained by the following principle - the basic idea of OAGs: Consider the attribute instances of v_X in Fig. 1. The inherited (synthesized) attributes of X are evaluated by *eval*(a)-operations in $vs_p(vs_q)$ and used in $vs_q(vs_p)$. Hence the attribute set A_X is partitioned by the context switches (*visit* and *leave*) into disjoint subsets $A_{X,i}$, $i=0,\ldots,m_X$ which alternately contain inherited or synthesized attributes only (Fig. 2).

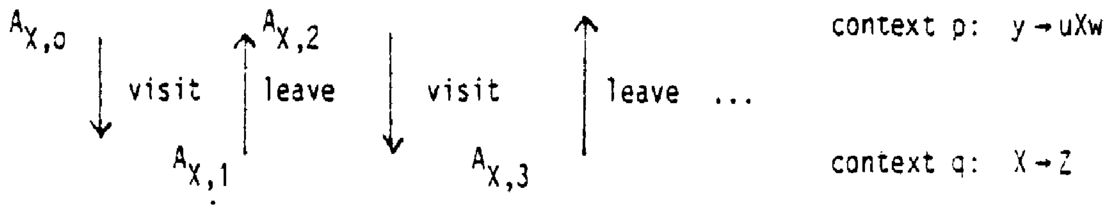

Fig. 2 Attribute partitions

The subsets $A_{X,i}$ are ordered such that $i<j$ implies that the attributes in $A_{X,i}$ are evaluated before those in $A_{X,j}$ for any X-node independent of its context. The partitions can be considered as an interface between any pair of visit-sequences vs_p, vs_q for rules of the form $p: y \rightarrow uXw$, and $q: X \rightarrow Z$. Assuming that a suitable partition is given for each symbol V then visit-sequences can easily be computed: Let DP_p be the dependency graph for ***direct dependencies*** between attribute occurences for a production $p: X_0 \rightarrow X_1 \cdots X_n$ such that

$DP_p = \{(X_i.a, X_j.b) \mid$ there is an attribute rule $X_j.b := f(..X_i.a..)$ associated to $p\}$

This graph is overlaid with the evaluation order given by the partitions:

$AP_p = DP_p \cup \{(X_i.a, X_i.b) \mid X_i = X,\ X.a \in A_{X,r},\ X.b \in A_{X,s},\ r>s\}$

In AP_p the nodes for attributes in $A_{X_0,r}$, for $r=0,2,4,\cdots$, are mapped into a single node for each i representing a ***leave***-operation. Correspondingly the $A_{X_i,r}$, for $i=1..n$, $r=1,3,5\cdots$ are mapped into nodes representing ***visit***-operations. The remaining nodes represent ***eval***-operations. A topological sorting of the graph yields a visit sequence vs_p. (The first ***leave***-operation must be deleted and a ***leave*** -operation is appended at the end.)

Now let us consider the probelm how to find a suitable partition for each symbol X. Obviously it must be consistent with any direct or indirect dependency between two attribute instances of any X-node in any attributed tree. Hence we define recursively graphs for *induced dependencies* for productions and symbols:

$$IDP_p = DP_p \cup \{(X_i.a, X_i.b) \mid X_i = X,\ (X.a, X.b) \in IDS_X\}$$

$$IDS_X = \{(X.a, X.b) \mid \text{there is a production } p: X_0 \to X_1 \cdots X_n \text{ and } 0 \le i \le n: X = X_i,\ (X_i.a, X_i.b) \in IDP_p^+\}$$

(IDP_p^+ is the non-reflexive, transitive closure of IDP_p.)

Acyclic IDS_X are a necessary but not sufficient condition for the existence of a suitable partion for all X yielding acyclic AP_p graphs. Even worse, the decision problem whether such a partition exists for an AG i.e. it is a simple multi-visit AG has been proven to be NP-complete (Engelfriet & File 1980).

For OAGs a canonical partition is defined such that each attribute is included in an $A_{X,i}$ with a maximal index i consistent with IDS: A function $ind_X : A_X \to \{0, \dots, m_X\}$ maps the attributes of A_X into the subsets $A_{X,i}$; i.e. $a \in A_{X,ind(a)}$. ind_X is defined by the following conditions

(1) m_X is odd and minimal

(2) $ind_X(a)$ is even $\Leftrightarrow a \in AI_X$

(3) $(a,b) \in IDS_X \Rightarrow ind_X(a) \le ind_X(b)$

(4) Let ind'_X be a function solving (1)-(3) then for all $a \in A_X : ind_X(a) \ge ind'_X(a)$

The AG is an OAG if all AP_p are acyclic with the partition defined by ind_X . This choice of the partition causes attributes to be evaluated as late as possible (lazy evaluation) - a desired effect for attribute storage optimization.

The reason for a simple multi-visit AG not to be an OAG is roughly characterized: For several symbols X, Y there are attributes independent in IDS and some DP_p contain dependencies between attribute occurences of X and Y which cause cycles in the AP_p. Such a situation is rare in practical cases; it may occur e.g. during AG development when not all attribute rules are completely specified. The GAG-System supports two techniques to cope with such situations:

a) One can enforce a certain partition by adding dependencies to the AG (the AG is arranged orderly (Kastens 1980)). They can be specified separately without changing the AG.

b) According to the OAG definition the partitions are computed independently for each A_X. If the OAG - condition fails. The GAG-System can optionally use a more "careful" algorithm: After the computation of the partition for one symbol the consequences for all AP_p and the "feedback" to the *IDS* graphs of symbols are computed. If it succeeds, the AG is "automatically arranged orderly".

4. Storage Reduction for Attributes

A naive implementation of an AG requires a huge amount of storage for the values of attribute instances: Many attributes describing typical language properties represent structured information like environments containing a set of valid definitions or structured type information. Such attributes are usually associated to many tree nodes, e.g. an environment attribute for all nodes containing applied occurences of identifiers. It is not tolerable to associate an attribute value representing the contents of a definition table to many nodes. The GAG-System reduces these storage requirements drastically by an efficient implementation of the ALADIN types and by attribute life-time analysis at generation time.

ALADIN is a strict applicative language without variables, loops, pointers, etc.. Furthermore there are no semantic functions outside the AG specification which are not under control of the GAG-System. Hence the semantics of an ALADIN specification does not change if all (structured) attribute values are implemented by pointers to objects; and copies of such objects are implemented by copies of the pointers. This technique is consequently applied in the GAG-System: A substructure of an object is represented by a pointer to the value it is constructed from. In our example language for each definition exactly one structured value is allocated. The environment attributes are lists of pointers to these objects. Even in the case of concatenation of such lists (when the environment is augmented on block level) none of the pointer lists need to be copied - only a new list header is allocated. The same holds for structured type attributes (e.g. describing procedure types).

Life-time Analysis. Attributed structure trees are implemented straight forward by tree nodes representing a symbol instance of X with a component for each of its attributes of A_X. A significant improvement of storage requirements is achieved if for some attributes all instances can be implemented by a global object outside the tree instead of many node components. From the visit-sequences one can derive assertions on the life-time of attribute instances at compiler generation time: If the life-times for all instances of an attribute $X.a$ are

- pairwise disjoint it can be implemented by a global variable,
- properly included it can be implemented by a global stack,
- overlapping it must be implemented by a node component.

These properties are defined by a mapping of the visit-sequences into context-free grammars. The description specifies an analysis algorithm based on well-known techniques for grammar analysis.

Each attribute a is separately considered for globalization. The set of visit-sequences is mapped to a context-free grammar $G_a=(N_a,T_a,P_a,S_a)$. G_a is defined such that for any structure tree the sequence of operations performed by the attribute evaluator which involve an instance of a (i.e. definition and last application of one instance) is a sentence of $L(G_a)$. (Since G_a is context-free $L(G_a)$ will in general contain additional sequences which cannot occur in attribute evaluation. Hence the conditions derived below are sufficient but not necessary.)

Let $G=(N,T,P,S)$ be the context-free grammar the AG is based on. For ease of description we assume that the AG is in Bochmann-Normal-Form.

N_a is the set of nonterminals with

$N_a=\{X^i \mid X\in N, i=1,..,k$, and k is the number of inherited subsets $A_{X,j}\}$.

$T_a=\{D,A\}$ is the set of terminals where D represents a defining rule for a, and A represents the last application of a in a visit-sequence.

$S_a=S^1$, where S is the start symbol of G

The productions P_a are constructed by the following rules. Consider a visit-sequence

$$vs_p=u_1\ \textit{leave}_1 u_2\ \textit{leave}_2 \dots u_k\ \textit{leave}_k$$

for a production $p\in P$ $X_0\to X_1\cdots X_n$. Then k productions of P_a are constructed:

$$X_0^1\to v_1,\dots,X_0^k\to v_k.$$

where the u_i are mapped into v_i. The *leave*-operations are omitted. Each **visit**(X_j,r) in an u_i is mapped into X_j^r in v_i. For each **eval**(b) in an u_i we destinguish the following cases:

a) If $b\neq a$ and the corresponding attribute rule does not depend on a it is omitted in v_i.

b) If $b\neq a$ and the corresponding attribute rule depends on a and there is no further application of a to the right in vs_p it is mapped to A.

c) If $b=a$ it is mapped to D.

d) If both (b) and (c) hold (i.e. different occurrences of a are involved) it is mapped to AD.

For a pair of rules in P of the form $p: Y \to y_1 X y_2$, and $q: X \to x$ synthesized attributes of X are defined in vs_p and applied in vs_q (for inherited attributes vice versa). For some p the attribute a may be not applied in vs_p at all. In that case in the production $Y^i \to uX^r v$ an A is inserted: $Y^i \to uX^r Av$, where r is the maximum number such that a production $X^r \to wDz$ exists. (The case of inherited attributes is treated accordingly.)

Now we can state sufficient life-time conditions in terms of G_a:

1) All instances of an attribute $a \in A_X$ can be implemented by a simple **global variable** if $L(G_a) = (DA)^*$, i.e. an attribute instance is used for the last time before the next instance is defined. This property can easily be checked by computation of the *FIRST* and *FOLLOW* functions for G_a:

   FIRST $(S_a) = \{D\}$,

   FOLLOW $(D) = \{A\}$, and

   FOLLOW $(A) = \{D, \varepsilon\}$.

2S) For a synthesized attribute $a \in A_X$ consider all productions in P_a containing D. They have the form $X^i \to wDz$, $s \le i \le r$. If all productions with X^s on the right-hand side have the form $Y^j \to uX^s vAy$ then a can be implemented by a **global stack.**

2I) For an inherited attribute $a \in A_X$ consider all productions in P_a containing A. They have the form $X^i \to zAw$, $s \le i \le r$. If all productions with X^r on the right-hand side have the form $Y^j \to uDvX^r y$, then a can be implemented by **global stack.**

The conditions 2S and 2I ensure that the life-times of all instances of a are disjoint or properly included. 2S and 2I are sufficient but not necessary because if a life-time is interrupted by an operation leaving the considered subtree the worst case - overlapping life-times - is assumed. If an attribute is decided to be implemented by a stack the above conditions describe where to insert push- and pop-operations into the visit-sequene (immediately before the D-respectively the A-operation).

Furthermore the GAG-System compares the life-time of different global attributes in order to implement groups of attributes by a single global variable or stack. (The result of this algorithm is of course not optimal.) The main effect of this grouping is not the reduction of attribute storage rather than the saving of code space and runtime for deleted copy operations within the variables of one group.

The effect of the improvement based on this life-time analysis are demonstrated by figures from an Ada-AG: The 517 attributes are implemented by 59 global variables, 24 global stacks, and 85 node components. 21% of all attribute rules are eliminated (copy rules within a group). (For more figures see (Kastens et.al. 1982)).

References

British Standards Institution (1982). Specification for Computer programming language Pascal.

DIN Deutsches Insitut für Normung e.V. (1980). Programmiersprache PEARL, Normentwurf, Beuth-Verlag.

Engelfriet, J., File, G. (1980). Simple Multi-Visit Attribute Grammars TH Twente, Memorandum Nr. 314.

Kastens, U. (1980). Ordered Attributed Grammars, in Acta Informatica 13, 229-256.

Kastens, U., Hutt, B., Zimmermann, E. (1982). GAG - A Practical Compiler Generator, LNCS 141, Springer-Verlag.

Kennedy, K. Warren, S.K. (1976). Automatic generation of efficient evaluators for attribute grammars; Conference record of the 3rd ACM Symp. on Principles of programming language, 32-49.

Uhl, J., Drossopoulou, S., Persch, G., Goos, G., Dausmann, M., Winterstein,G., Kirchgässner, W. (1982). An Attribute Grammar for the Semantic Analysis of ADA, LNCS 139, Springer Verlag.

```
%                          Appendix
%
%           Static semantics of the example language
%            specified in ALADIN for the GAG-System

(* Attribute types: *)

TYPE mode        : UNION (simple_type, proc_type);
TYPE proc_type   : STRUCT (params : definitions, result : simple_type);
TYPE simple_type : (int, real, bool, no_type);
TYPE definition  : STRUCT (id : SYMB, descr : mode);
TYPE definitions : LISTOF definition KEY id;

(* Attributes associated to (non)terminals : *)

NONTERM program : ;

NONTERM block :
  globals   : definitions,
  env       : definitions,
  type      : simple_type;

NONTERM expression, formula, primary :
  type      : simple_type;

NONTERM act_params :
  formals   : definitions;

NONTERM declaration, simple_decl :
  def       : definition;

NONTERM identifier :
  def       : definition;

TERM    type_denoter :        (* attribute values of terminals *)
  type      : simple_type;    (* are supplied by the scanner   *)

TERM    ident :
  id        : SYMB;

TERM    literal :
  type      : simple_type;

TERM    opr :
  symbol    : SYMB;


RULE  p_1:   program ::= block
STATIC
  block.globals := definitions();
END;

RULE  p_2:   block ::= '(' declaration *  (expression // ';') ')'
STATIC
  block.env := declaration.def + block.globals;
  block.type := LAST (expression.type);
  CONDITION UNIQUE_KEYS (definitions (declaration.def))
    MESSAGE 'multiple defined identifiers';
END;
```

```
RULE  p_3:   declaration ::= simple_decl
STATIC TRANSFER
END;

RULE  p_4:   simple_decl ::= type_denoter ident ';'
STATIC
  simple_decl.def := definition (ident.id, type_denoter.type);
END;

RULE  p_5:   declaration ::=
                 'proc' ident '(' simple_decl * ')'
                 type_denoter ':' block ';'
STATIC
  declaration.def :=
    definition
      (ident.id, proc_type (simple_decl.def, type_denoter.type));
  block.globals := simple_decl.def + (INCLUDING block.env);
  CONDITION UNIQUE_KEYS (definitions (simple_decl.def))
    MESSAGE 'multiple defined parameters';
  CONDITION compatible (block.type, type_denoter.type)
    MESSAGE 'incompatible result type';
END;

RULE  p_6:   expression ::= identifier ':=' formula
STATIC
  expression.type :=
    CASE identifier.def.descr OF
    IS simple_type : THIS
    OUT (no_type CONDITION FALSE MESSAGE 'wrong variable')
    ESAC;
  CONDITION compatible (formula.type, expression.type)
    MESSAGE 'incompatible type in assignment';
END;

RULE  p_7:   expression ::= formula
STATIC TRANSFER
END;

RULE  p_8:   formula ::= formula opr primary
STATIC
  formula[1].type :=
    opr_identify (opr.symbol, formula[2].type, primary.type);
END;

RULE  p_9:   formula ::= primary
STATIC TRANSFER
END;

RULE  p_10:  primary ::= block
STATIC
  primary.type := block.type;
  block.globals := INCLUDING block.env;
END;
```

```
RULE  p_11:  primary ::= identifier '(' [ act_params ] ')'
STATIC
  primary.type :=
    CASE identifier.def.descr OF
    IS proc_type: THIS.result
    OUT (no_type CONDITION FALSE MESSAGE 'wrong call')
    ESAC;

  act_params.formals :=
    CASE identifier.def.descr OF
    IS proc_type:  (THIS.params
                     CONDITION EMPTY (IT) OR (act_params IS THERE)
                     MESSAGE 'missing parameter')
    OUT             definitions ()
    ESAC;
END;

RULE  p_12:  primary ::= identifier
STATIC
  primary.type :=
    CASE identifier.def.descr OF
    IS simple_type : THIS
    OUT (no_type CONDITION FALSE MESSAGE 'wrong variable')
    ESAC;
END;

RULE  p_13:  primary ::= literal
STATIC TRANSFER
END;

RULE  p_14:  act_params ::= expression ',' act_params
STATIC
  act_params[2].formals := TAIL (act_params[1].formals);
  CONDITION
    IF EMPTY (act_params[1].formals)
    THEN FALSE
    ELSE compatible (expression.type,
                     HEAD (act_params[1].formals).descr QUA simple_type)
    FI
    MESSAGE 'wrong parameter';
END;

RULE  p_15:  act_params ::= expression
STATIC
  CONDITION
    IF EMPTY (act_params.formals)
    THEN FALSE
    ELSE compatible (expression.type,
                     HEAD (act_params.formals).descr QUA simple_type)
    FI
    MESSAGE 'wrong parameter'
END;

RULE  p_16: identifier ::= ident
STATIC
  identifier.def := identify (ident.id, INCLUDING block.env);
END;
```

```
CONST no_def : definition ('', no_type);

FUNCTION identify (id : SYMB, defs : definitions) definition :
  IF EMPTY (defs)
  THEN (no_def CONDITION FALSE MESSAGE 'identifier not defined')
  ELSE IF HEAD (defs).id = id
       THEN HEAD (defs)
       ELSE identify (id, TAIL (defs))
       FI
  FI;

FUNCTION opr_identify
         (opr : SYMB, left , right : simple_type) simple_type :
  (CASE opr OF
   '+': '-': '*':
        IF compatible (left, int) AND compatible (right, int)
        THEN int ELSE
        IF compatible (left, real) AND compatible (right, real)
        THEN real ELSE no_type FI FI;
   '/': IF compatible (left, real) AND compatible (right, real)
        THEN real ELSE no_type FI;
   'AND': 'OR':
        IF compatible (left, bool) AND compatible (right, bool)
        THEN bool ELSE no_type FI
   OUT  no_type
   ESAC CONDITION IT =/ no_type MESSAGE 'wrong operation');

FUNCTION compatible (t1, t2 : simple_type) BOOL :
  (t1 = t2) OR (t1 = no_type) OR (t2 = no_type) OR
  ((t1 = int) AND (t2 = real));
```