Attribute Grammars in a Compiler Construction Environment

Uwe Kastens
University of Paderborn

Abstract

Attribute grammar (AG) specifications are implemented by attribute evaluators which perform computations on tree structures as specified. An AG system should concentrate on generating that implementation. In compiler construction the attribute evaluator has to be combined with other modules, like scanner, parser, tree construction, data bases, and translation modules. Those are generated by other compiler tools, taken from a library, or implemented for the specific application. These components have to fit together properly on the specification level as well as on the level of module interfaces. In this paper those relationships are demonstrated using the LIGA system within the compiler construction environment Eli as an example.

1 Introduction

Compiler Construction is a deeply analyzed area of computer science. The compilation task in general is well understood, and there is an overall agreement about its decomposition in subtasks, which can be found in text books like [Kas90,WaG84]. The decomposition of tasks leads to a task oriented modular compiler structure. For the solution of the subtasks specific systematic implementation techniques can be applied. Hence a great part of compiler implementation can be automated by reusing standardized implementations of subtasks and by generating others from specifications.

Attribute grammar (AG) specifications play a central role for that part of the compilation process which requires computations on the source program's structure tree, i.e. the phase of semantic analysis and translation into an intermediate or target language. If that target representation is a tree AGs can be used as well to specify code generation for it. The AG defines the shape of the structure tree, and thus contributes to the specification of the structuring phase.

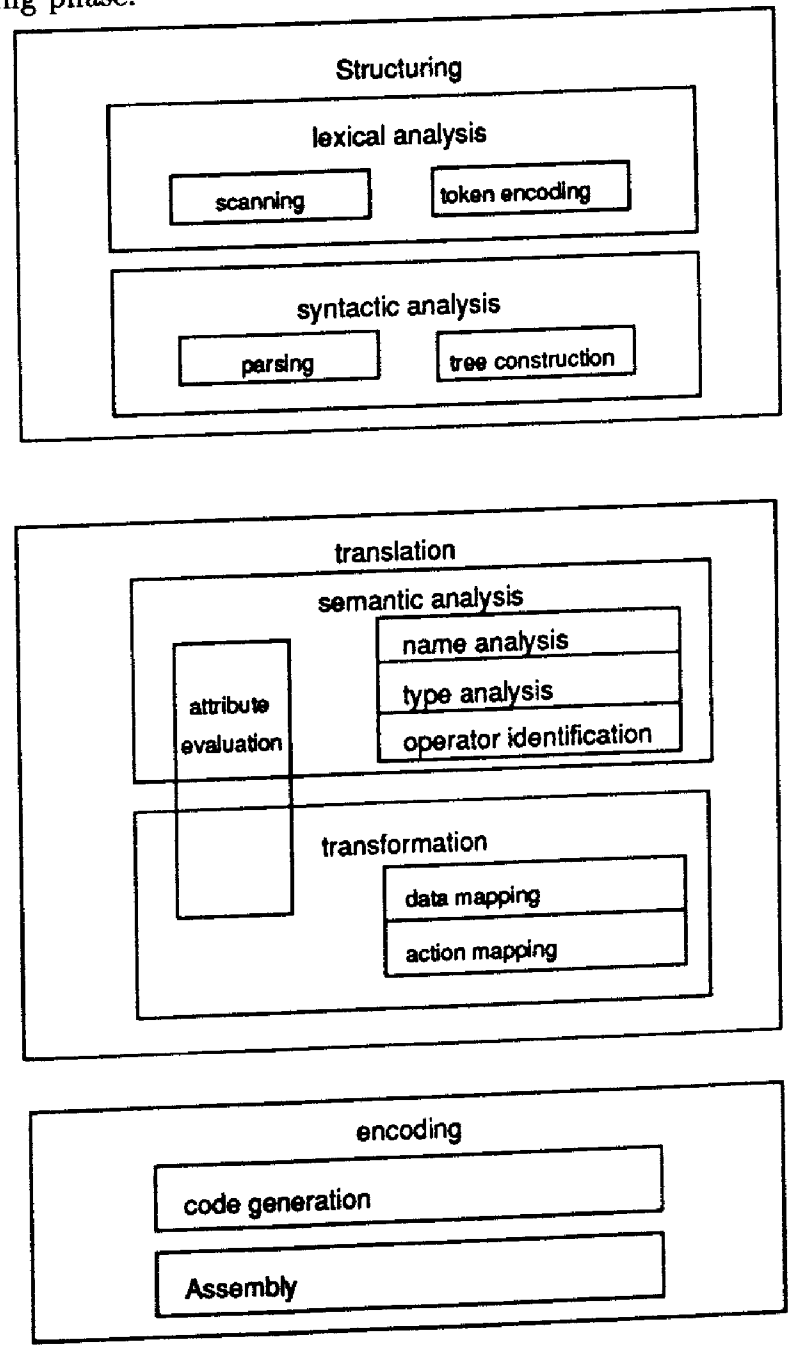

Fig. 1.1 Compiler task decomposition

In Fig. 1.1 a decomposition of the compilation task is shown, taken from [GHK89]. It also describes the overall modular structure of a compiler. The first compiler phase *structuring* transforms the source program text into the *structure tree* representing the program's abstract structure. It is the central data structure for semantic analysis and transformation in the second phase *translation*, which yields an intermediate or target representation of the program. The last phase performs the target machine specific *encoding* and sequencing of operations and data. The task and structure of this phase mainly depends on the class of the target machine for the compiler. Fig. 1.1 does not include an optimization phase which might be inserted before the last phase. In this paper we concentrate on the structuring and the translation phase where AGs play a central role.

The modules scanning, parsing, and attribute evaluation are the active drivers within their phase. They each transform the program representation by using functions of the modules drawn right to them in Fig. 1.1. The latter are implementations of task specific abstract data types (ADTs).

The tasks of scanning, parsing, and attribute evaluation can be specified such that tools can generate the implementations automatically. In fact the generator produces the central algorithms, which is embedded into a suitable interface to the surrounding modules, as shown in the generation scheme of Fig. 1.2. For example the token sequence is the input interface for the parser, which drives the tree construction such that the basic data structure for attribute evaluation is build.

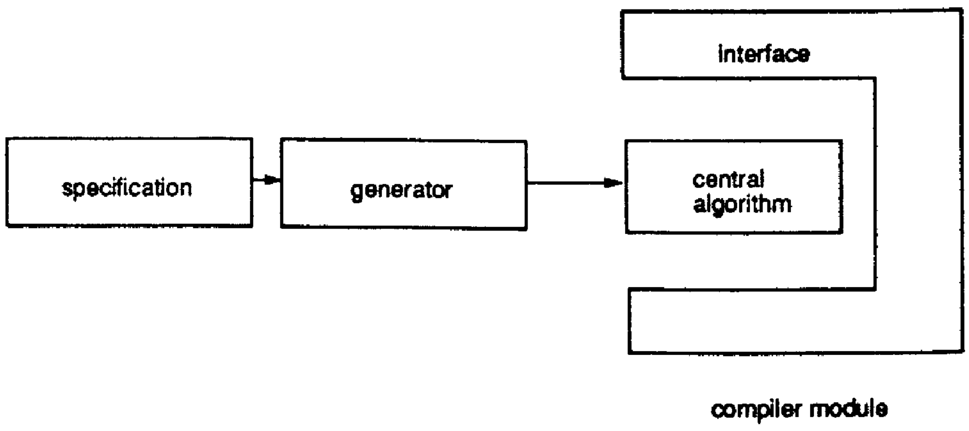

Fig. 1.2 Scheme for generating tools

These interactions between the central modules cause dependencies between their specifications, which will be discussed in Sect. 2. On the implementation side the module interfaces must be designed carefully such that they fit together, and don't introduce unnecessary inefficiency. If the modules are generated using stand-alone tools (scanner, parser, attribute evaluator generator) a considerable amount of technical know how and manual implementation is required to fit the generated products together.

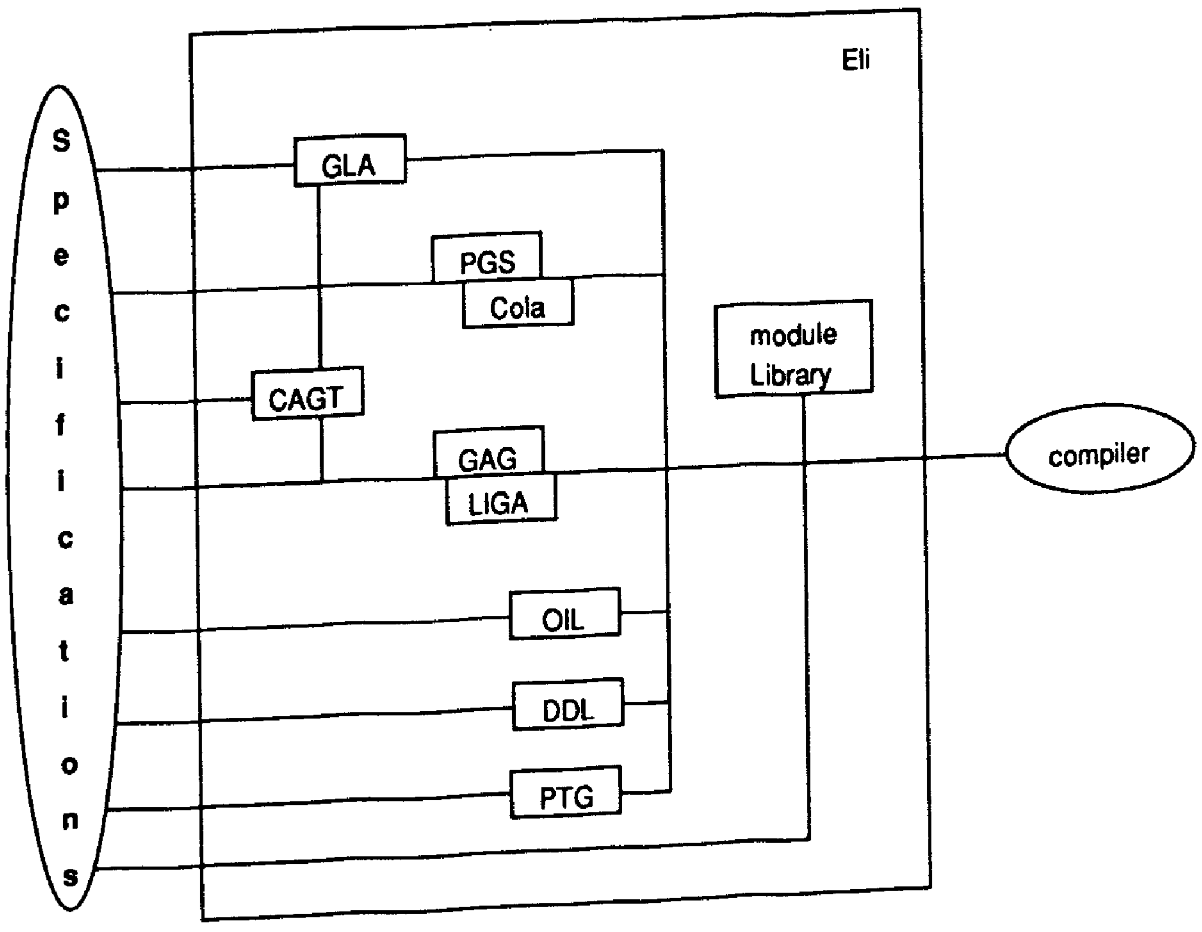

Fig. 1.3 Compiler Construction Environment Eli

A compiler construction environment like Eli [Dep91,GHK89,WHK88] solves the problems of keeping specifications consistent, interfacing the generated products, and configuring the whole compiler. Hence the user can concentrate on the development of the specifications and of the problem specific modules. Eli combines a set of compiler generating tools the scanner generator GLA [Heu86], the parser generators PGS [DDH84] and Cola [Pro89], the attribute evaluator generators GAG [KHZ82] and LIGA [Kas89], further generators for compilation tasks described in Sect 3 and 4 and implementations of several modules for standard compilation tasks. Fig. 1.3 shows a rough structure of Eli. Its main general properties are:

- The know-how for tools usage and tools cooperation is embodied in the system. The user delivers specifications instead of activating stand-alone tools.
- The system solves the configuration problem by composing the compiler (in source and executable) from many intermediate products.
- Eli helps the users in tracing errors back to the specifications. This is a non-trivial problem due to the interdependence between specifications and interaction between tools.
- The system is implemented on top of the general tool management system ODIN [Cle88]. Hence it is flexible and extendable with respect of integration of tools and modules.

AG specifications for the generators GAG or LIGA play a central role in Eli. In Sect. 2 we show how AGs are related to the specification of the structuring phase, and give some guidelines for the design of the structure tree specification. Standard solutions for subtasks of the translation phase are presented in Sect. 3.

2 Attribute Grammars and the Structuring Phase

The first compilation phase, *structuring* (cf. Fig. 1.1), transforms the source program into a structure tree. It is the basic data structure for attribute evaluation in the second phase, *translation*. The context-free grammar of the AG is the abstract syntax which specifies those trees. Hence the target of the structuring transformations is given by the AG. The mapping is achieved in two steps: The lexical analysis transforms the character sequence of the program into a token sequence, specified by descriptions of the tokens for the scanner generator. In the second step the parser computes a derivation for the token sequence according to the concrete syntax. It drives the tree construction module which builds the structure tree.

The three specifications, tokens, concrete syntax, and abstract syntax in the AG, are related. In the following we first discuss the scanner specification, then describe the relationship between concrete and abstract syntax, and give some guidance for abstract syntax design with respect to attribution.

The **lexical analysis phase** deals with two kinds of tokens: those which are only relevant for the structuring phase, and those which carry information to the translation phase. Keywords, like **begin** and **then**, delimiters like ; and parenthesis, and operators, like + and :=, belong to the first kind. They are specified by a fixed character sequence each. Their specification can easily be extracted from the concrete syntax, such that the user need not supply a separate specification for them. A syntactic code is assigned automatically and is communicated to the parser generator for identification of the terminal symbols.

The second kind of tokens represent both syntactic and semantic information. Identifiers and literals typically belong to this kind. Their notation is usually specified by a regular expression, e.g. in a specification for the scanner generator GLA:

```
Ident: [a-zA-Z] ([a-zA-Z] | [0-9]) [mkidn]
```

The token name, e.g. Ident, is used as a terminal symbol in the concrete syntax and in the AG. There is usually one symbol attribute associated to such a token which represents its identity, e.g. a unique encoding of identifiers, specified in the AG

```
TERM Ident: id: Symb;
```

The value of the symbol attribute is computed by one of the token encoding modules in the lexical phase. Eli provides a library of standard modules for identifiers and literals. Their functions have a standardized interface used in calls of the GLA generated scanners. Hence it is sufficient just to add the name of that function, e.g. *mkidn*, to the token specifications. For each of these terminals a leaf node is made in the structure tree. Its symbol attribute is computed by the encoding function prior to attribute evaluation. The function and the attribute evaluation have to agree upon its representation.

The **syntactic analysis phase** performs a transformation from the token sequence to the structure tree. It is specified by the concrete syntax, the abstract syntax, and their relationship. The concrete syntax is derived from the CFG given in the source language definition. Usually some transformations are applied to disambiguate the grammar and to achieve the grammar class required by the parser generator, e.g. LALR(1) in case of Eli. Development of the abstract syntax is an early step of AG design. Its objective is to specify a structure tree, which describes the semantic structure without unnecessary redundancy. As a general rule those productions of the concrete syntax, that have the same computation pattern in the AG should be unified in the abstract syntax, in order to avoid repetition of attributions. Those chain productions which don't have specific attributions except transfer rules can be omitted in the abstract syntax. In practice abstract syntax design needs already a good plan of the computations to be specified in the AG, if revisions of the decisions should be avoided. In the following we first explain formal relationships between concrete and abstract syntax, and then give some guidance for typical cases in compiler applications.

In Fig. 2.1 the concrete syntax for simple expressions is given. The grammar is unambiguous and LALR(1). It describes certain structural properties of the operators: The left-recursive productions *p1* and *p2* define both *AddOpr* and *MulOpr* being left-associative, i.e. the structure of $a+b+c$ is equivalent to that of $(a+b)+c$. Furthermore the *MulOpr* have higher precedence than the *AddOpr*, i.e. the structure of $a+b*c$ is equivalent to that of $a+(b*c)$. This property results from the hierarchical order of the nonterminals *Expr* $\rightarrow$ *Fact* $\rightarrow$ *Term*. Other bindings of operands to operators can be achieved by parenthesized subexpressions according to *p5*, i.e. $a+(b+c)$ or $(a+b)*c$. These structural properties of expressions are the determined by a parser which constructs a derivation according to the concrete grammar.

The design of AGs for expressions leads to the observation that the computation pattern is the same for any binary operator context. It comprises rules for overloading resolution, type determination, and translation coercion. Hence there is no need to distinguish contexts for *AddOpr* and *MulOpr*. There is only one other context *p6* which requires non-trivial attribution for identification of the applied identifier, computation of its type and translation. Hence attribution can be associated to an abstract syntax as given in Fig. 2.2.

```
p1:   Expr ::= Expr AddOpr Fact
p2:   Expr    ::= Fact
p3:   Fact    ::= Fact MulOpr Term
p4:   Fact    ::= Term
p5:   Term    ::= '(' Expr ')'
p6:   Term    ::= Ident
p7:   AddOpr ::= '+'
p8:   AddOpr ::= '-'
p9:   AddOpr ::= '*'
p10:  MulOpr  ::= '/'
```

Fig. 2.1 Concrete expression syntax

```
RULE ap1:   Expr ::= Expr Opr Expr END;
RULE ap2:   Expr ::= Ident         END;
RULE ap3:   Opr  ::= '+'           END;
RULE ap4:   Opr  ::= '-'           END;
RULE ap5:   Opr  ::= '*'           END;
RULE ap6:   Opr  ::= '/'           END;
```

Fig. 2.2 Abstract expression syntax

The relationship between concrete and abstract syntax can be described by a straightforward formalism: The nonterminals of the concrete syntax form a set of equivalence classes, {*Expr, Fact, Term*} and {*AddOpr, MulOpr*} in our example. Their members are mapped to the same nonterminal of the abstract syntax, *Expr* and *Opr*. Hence both productions *p1, p2* are mapped to *ap1, p6* to *ap2*, and *p7*, ..., *p10* to *ap3*, ..., *ap6*. The chain productions *p2, p4* for symbols within one equivalence class do not occur in the abstract syntax. *p5* is considered as a chain production, too, because it has only structural but no semantical relevance.

This mapping of productions also defines the construction of the structure tree. Ac-

```
p1:   Stmts  ::=  Stmts  Stmt
p2:   Stmts  ::=
p3:   Stmts  ::=  Stmt
p4:   Stmt   ::= 'begin' Stmts 'end'
p5:   Stmt   ::= 'while'  Expr 'do' Stmt
p6:   Stmt   ::= 'if' Expr 'then' Stmt 'else' Stmt
p7:   Stmt   ::=  Var ':=' Expr ';
```

Fig. 2.3 a) Concrete statement syntax

tions for node construction are associated to those concrete productions which have a corresponding production in the abstract syntax. In the Eli system the design of the abstract syntax is supported by a tool, CAGT [Dep91], which automatically relates the productions and attaches the actions for tree construction to the parser grammar.

In Fig. 2.2 the operator symbols are retained for better readability only, they do not occur in the structure tree. It has nodes for nonterminals and attributed terminals, like *Ident*, only. A closer look at the attribution for *Opr* shows that the grammar can be further simplified. All four contexts would have a computation pattern like

```
RULE  ap3:   Opr  ::= '+'
STATIC   Opr.op = Addsym
END;
```

We could drop the four productions and let *Opr* be a terminal with the symbol attribute *op*. Then the attribute is computed by an encoding module of the lexical phase, as described in the symbol specification.

The technique of abstract syntax design as described for expressions can be applied to other parts of the grammar correspondingly, as shown for statements in Fig. 2.3.

In contrast to the techniques described above, where chain productions are eliminated, the AG can also be simplified by introducing specific chain productions. Such a situation is indicated if a common computation pattern occurs in several contexts. Typical examples are occurrences of identifiers. There are two different computation patterns,

```
RULE  ap1:  Stmt  ::=  Stmt Stmt                           END;
RULE  ap2:  Stmt  ::=                                      END;
RULE  ap3:  Stmt  ::= 'while' Expr 'do' Stmt               END;
RULE  ap4:  Stmt  ::=  'if' Expr 'then' Stmt 'else' Stmt   END;
RULE  ap5:  Stmt  ::=   Var ':=' Expr ';'                  END;
```

Fig. 2.3 b) Abstract statement syntax

one for defining occurrences (generating a key for the defined properties, introducing the (identifier, key)-pair into the environment of the surrounding block, and checking for multiple definitions), and one for applied occurrences (identifying the key in the surrounding environment and checking whether a definition exists). These attributions can be bound to context like

```
RULE  apDef: DefIdent ::= Ident END;

RULE  apApp: AppIdent ::= Ident END;
```

Each occurrence of *Ident* in the concrete syntax is then classified to be a *DefIdent* or an *AppIdent*. So it is avoided to repeat those attributions in several contexts. This principle of factorizing computations by chain productions can usually be applied to several contexts in the grammar. Another typical example is given by those contexts which introduce a new scope, e.g. block, procedure body, with-statement, etc. A new nonterminal, e.g. *Range*, would introduce suitable contexts for those attribution patterns.

Application of such design rules for the abstract syntax contributes to the quality of the AG. Since the abstract syntax is designed before the computations are specified, a good plan for the attribution is required in advance. Often situations for improvements by modifying the abstract syntax arise during AG development. Such design iterations do not require much effort, if the mechanical work is supported by tools in a compiler construction environment like Eli.

3 Modules and Tools for Semantic Analysis

Several tasks of semantic analysis have general standard solutions which can be systematically adapted to the specific requirements of the particular compiler. The Eli system provides modular implementations of such solutions for common tasks, like name analysis, property association, and operator identification. The functions of these modules are used in the AG specification according to the principles described in [Kas91]. In this section we give two examples for such standard solutions in Eli, a name analysis module and a generator for operator identification. Another module for property association is described in [Kas91].

3.1 Name analysis according to scope rules

One of the central semantic analysis tasks in any compiler is name analysis according to scope rules: For each applied identifier occurrence the definition is found which is valid at that program point. Language specific scope rules describe where a definition is valid. The following rule is basic for many block structured languages:

> A definition for an identifier *a* is valid in its smallest enclosing range, but not in inner ranges which also have a definition for *a*.

This rule describes scopes in Algol 60; the rules for Pascal, C and Ada are variants of it.

The general identification problem can be reduced to few concepts and operations: A definition is represented by a pair of an identifier and a unique key, which allows to associate and access properties of the defined object. An environment is the set of definitions (identifier, key)-pairs which are valid at a certain program point. Environments are constructed hierarchically by adding a set of definitions, called a scope, to an environment.

In [KaW90] an abstract data type is defined for name analysis based on these concepts, and an efficient implementation based on a state model is given. Its operations can be used to specify different variants of scope rules in common programming languages; examples for C, Pascal, and Modula-2 are given in that article. Such a module is available in the Eli system. Its main operations are

e = *NewEnv ()*

Generates a new environment, e. The function is used once for each disjoint name space.

e_2 = *NewScope* (e_1)

Generates a new environment e_2. Definitions may be added to the scope of e_2. e_2 inherits all definitions of e_1, unless they are redefined in e_2.

k = *DefineIdn (e, i)*

Adds a pair (i, k) to the scope of e with a new key k, if there is no definition for i in the scope of e; otherwise no definition is entered, and k is the key of the existing definition. (The latter case indicates multiple definition of an identifier in usual applications.)

k = *KeyInEnv (e, i)*

Yields the key of a pair (i, k) in e, or k = *NoKey* if e does not contain a definition for i.

A typical attribution for scope rule specification is shown in Fig. 3.1, comprising the context of the program root, of ranges where possibly nested scopes are introduced, and of defined and applied identifiers. It should be noted that two environment attributes are used: *initial_env* describes just the hierarchy of environments, *env* describes the environments with all their definitions completed. The message computation in the *DefIdent* context is made dependent on *Range.env*, in order to yield a message at each of the multiple definitions.

Association and access of defined properties should be described independently of name analysis, because it is a separate task to be solved by different means. There is a general applicable module in Eli for that purpose, defining *Set* and *Get* operations for properties of different representations. In the attribution of Fig. 3.1 the property *IsDef* is used to check for multiple definitions. Further application patterns of such a module are discussed in [Kas91].

```
RULE  Prog  ::= ...
STATIC
   Prog.initial_env = New Env();
   Prog.env = EnterPredefs (Prog.initial_env);
END;
RULE   Range  ::= ...
STATIC
   Range.initial_env =
      NewScope (INCLUDING (Range.initial_env, Prog.initial_env));
   Range.env =
      Range.initial_env DEPENDS_ON CONSTITUENTS DefIdent.def,
                                   INCLUDING (Range.env, Prog.env);
END;
RULE   DefIdent ::= Ident
STATIC
   DefIdent.key = DefineIdn (INCLUDING (Range.initial_env), Ident.id);
   DefIdent.def = SetIsDef (DefIdent.key, defined, multiple)
   message_if (EQ (GetIsDef (DefIdent.key, undefined), multiple),
                   "multiple defined identifier")
   DEPENDS_ON INCLUDING (Range.env);
END;
RULE  AppIdent ::= Ident
STATIC
   AppIdent.key = KeyInvEnv (INCLUDING Range.env, Ident.id);
   message_if (EQ (AppIdent.key, NoKey), "undefined identifier");
END;
```

Fig. 3.1 Scope attribution with name analysis module

3.2 Operator identification

In many programming languages operator symbols are overloaded with several different meanings, e.g. in Pascal the + symbol is used for integer addition, real addition, and set union. Both tasks of type checking and operator translation require overloading resolution, i.e. for each operator symbol occurrence its meaning has to be identified. The specific language rules for operator identification are described in terms of type signatures for operators and coercion relations between types.

```
RULE  Expr  ::=  Expr  Opr  Expr
STATIC
   Opr.target =
      OilIdOp2 (Opr.op, Expr[2].type, Expr[3].type);
      message_if (EQ (Opr.target, NoOpr),
                  "incompatible operator and operand types");
   Expr[1].type     = OILOprType (Opr.target, 0);
   Expr[2].posttype = OILOprType (Opr.target, 1);
   Expr[3].posttype = OILOprType (Opr.target, 2);
   Expr[1].coerce   = OILOprCoerce (Expr[1].type, Expr[1].posttype);
END;
```

Fig. 3.2 Operator identification computation pattern

There are two different general schemes for overloading resolution: In the Pascal-like scheme [Ame83b] the operator is identified on the base of its operand types; the Ada-like scheme [Ame83a,PWD79] additionally considers the result type required by the context of the expression. The OIL tool [Dep91,Kas88] in the Eli system supports both schemes by a module with two sets of functions. They are parameterized with operator signatures generated from specifications for OIL. The following example demonstrates the application of OIL in case of Pascal-like operator identification. Fig. 3.2 shows a computation pattern for identification of binary operators. The call of *OilIdOp2* yields the identified target operator for the given operator symbol and the operand types. Its result type is the type of the whole expression. The operands have to be coerced to the *posttype* taken from the operator's signature.

```
% Signatures of target operators:
OPER iAdd, iSub, iMul           ( int, int ):   int;
OPER rAdd, rSub, rMul           ( real, real ): real;
OPER sUnion, sDiff, sIntersect ( set, set ):   set;

% Source operators overloaded by target opertors:
INDICATION Add: iAdd, rAdd, sUnion;
INDICATION Sub: iSub, rSub, sDiff;
INDICATION Mul: iMul, rMul, sIntersect;

 % Coercion operator:
 COERCION Float ( int ): real;
```

Fig. 3.3 Operator identification specification

This computation pattern is completely independent of the operator signatures involved. Those are specified by rules in the OIL specification language. Fig. 3.3 shows an example specification for the source operators *Add, Sub, Mul* which are overloaded by *int*, *real*, and *set* operators each. A widening coercion is defined from *int* to *real*. OIL translates such a specification into data used by the functions of the attribution. Hence the AG is not influenced by the usually great number of operators and their signature.

4 Modules and Tools for Transformation

On the base of the tree structure and the information computed by the semantic analysis the transformation phase transforms operations and data of the program into an intermediate or target representation, cf. Fig. 1.1. Structure and contents of that representation depends on the interface of the encoding phase. For certain compilation tasks that phase is not needed, and the target representation is produced directly. In any case the transformation is specified by computation patterns of the AG, which should be separated into specific AG modules (see [Kas91].)

It is a good design practice to implement the target representation by a module, which provides constructor functions for components of the target structure. Such a target modules depends very much on the design decisions for the specific compiler. Hence the Eli system provides some mechanisms for certain classes of target structures, rather than precoined solutions. In the following we briefly describe the support for producing plain text files, tree structures, and tree structured target text.

```
Frame:          "#include <stdio.h>/n"
                "main () {/n"
                $/* body */ "/n"
                "/n}/n"


Block:          "/n{"
                $/* decls and stmts */
                "/n}/n"


Decl:           $/* type */  $/* identifier */  ";/n"


Assign:         $  "=" $ ";/n"


Seq:            $  $


Empty:


Numb:           int [printnumb]


Ident:          int int [printident]
```

Fig 4.1 Translation specification for PTG tool

Unstructured textual output is easily produced using print functions which append strings to an output file. They may be given directly as arguments, taken from the string memory if source symbols are written, or converted from numbers. The attribution has to specify the sequencing of output components by dependencies between the function calls, as described in [Kas91].

If the target structure is an abstract intermediate language, it is usually represented by a tree composed of nodes of different types. A constructor module is easily developed, such that it provides a constructor function for each node type, like *PTGBlock* and *PTGDecl* in Fig. 4.2.

The computations of the AG in Fig. 4.2 then specify bottom-up tree construction by calls of those functions, which use attributes representing subtrees. The same principle can be used to produce directed acyclic graphs. If cyclic graphs should be produced, the module needs additional functions for definition and use of labeled subgraphs. Such modules may also be generated by other tools like IDL [Lam87].

A special case arises if the target tree is interpreted as the structure tree of a subsequent attribution phase, which specifies register and label allocation for example. In that case the second AG specifies the tree structure, i.e. the node types with descendants and attributes. The AG systems integrated in Eli (GAG [KHZ82] and LIGA [Kas89]) generate the constructor functions which are used in the first AG to produce the tree. Eli also supports a mechanism to combine both attribution phases into one, yielding a mechanism for AG specified tree transformation.

There are many compiler applications which translate a source program into a program of a different source language, e.g. Pascal to C translation, or in general produce text with an underlying hierarchical structure, e.g. transform a data manipulation language into a sequence of nested calls of a data base system. This task can be easily solved if it is decomposed into its two subtasks: The hierarchical structure is computed as a target tree using the technique applied in Fig. 4.2. The tree is then transformed into a textual representation by recursively applying a transformation rule for each node type. This kind of source to source transformation is supported in Eli by a specific tool, PTG (Program Text Generator, [Dep91]). The form of the output is described by a specification for PTG. It has one rule for each node type, describing its components and its textual representation as shown in Fig. 4.1.

The rule for ***Assign*** describes that the node is composed of two subtrees. It should be textually represented by the text of those subtrees separated by a = symbol and followed

```
NONTERM Block:   ctext: PTGNode;

CHAIN cseq: PTGNode;


RULE p1:    Prog ::= Block
STATIC
  Prog.ctext =
     PTGOut (PTGFrame (Block.ctext));
END;
RULE p2:    Block ::= 'begin' Decls Stmts 'end'
STATIC
  CHAINSTART Decls.cseq = PTGEmpty ();
  Block.ctext = PTGBlock (Stmts.cseq);
END;
RULE p3:    Decl ::= Ident ':' Ident ';'
STATIC
  Decl.cseq = PTGSeq (Decl.cseq,
                  PTGDecl (PTGIdent (Ident[2].id),
                           PTGIdent (Ident[1].id)));
END;
RULE p4:    Stmt ::= Ident ':=' Numb ';'
STATIC
  Stmt.cseq = PTGSeq (Stmt.cseq,
                  PTGAssign (PTGIdent (Ident.id),
                             PTGNumb (numb.val)));
END;
RULE p5:    Stmt ::= Block
STATIC
  Stmt.cseq = PTGSeq (Stmt.cseq, Block.ctext);
END;
```

Fig 4.2 Attribution for target tree construction

by a ;. From such specifications PTG generates a tree construction module to be used in the AG specification like that of Fig. 4.2. In this case it would contain a function

```
PTGNode PTGAssign (/* PTGNode a, b */) ...
```

Furthermore there is a predefined function *PTGOut* which produces the specified textual representation for its tree argument. Other PTG specification rules allow to attach user defined functions for generating symbols like identifiers or numbers. Hence for this kind of application both the tree construction module and the output module are generated automatically and are applied in an attribution like that of Fig. 4.2.

5 References

[Ame83a] American National Standards Institute, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815, 1983.

[Ame83b] American National Standards Institute, *Programming Language PASCAL*, ANSI/IEEE 770 X3.97-1983, 1983.

[Cle88] Clemm, G. M., *The Odin Specification Language*, in International Workshop on Software Version and Configuration Control '88, Teubner, Stuttgart, 1988.

[DDH84] Dencker, P., Dürre, K. and Heuft, J., *Optimization of Parser Tables for Portable Compilers*, ACM Transactions on Programming Languages and Systems 6 (October 1984), 546–572.

[Dep91] Department of Electrical and Computer Engineering, University of Colorado, *Eli Documentation*, Technical Report, Boulder, CO, 1991.

[GHK89] Gray, R. W., Heuring, V. P., Krane, S. P., Sloane, A. M. and Waite, W. M., *Eli: A Complete, Flexible Compiler Construction System*, Department of Electrical and Computer Engineering, University of Colorado, SEG 89-1-1, Boulder, CO, June 1989.

[Heu86] Heuring, V. P., *The Automatic Generation of Fast Lexical Analyzers*, Software - Practice & Experience 16 (September 1986), 801–808.

[Kas89] Kastens, U., *LIGA: A Language Independent Generator for Attribute Evaluators*, Universität-GH Paderborn, Bericht der Reihe Informatik Nr. 63, 1989.

[Kas90] Kastens, U., *Übersetzerbau*, Handbuch der Informatik, Oldenbourg Verlag, München, 1990.

[Kas91] Kastens, U., *Attribute Grammars as a Specification Method*, Proceedings of the International Summer School on Attribute Grammars, Application and Systems, Prague (1991).

[Kas88] Kastens, U., *Code Generation Based on Operator Identification*, Universität-GH Paderborn, Reihe Informatik, Bericht Nr. 49, Januar 1988.

[KHZ82] Kastens, U., Hutt, B. and Zimmermann, E., *GAG: A Practical Compiler Generator*, Lecture Notes in Computer Science, vol. 141, Springer Verlag, Heidelberg, 1982.

[KaW90] Kastens, U. and Waite, W. M., *An Abstract Data Type for Name Analysis*, accepted for publication in Acta Informatica, 1990.

[Lam87] Lamb, D. A., *IDL: Sharing Intermediate Representations*, ACM Transactions on Programming Languages and Systems 9 (1987), 297–318.

[PWD79] Persch, G., Winterstein, G., Dausmann, M. and Drossopoulou, S., *Overloading in Ada*, Fakultät für Informatik, Universität Karlsruhe, Bericht 23/79, Karlsruhe, BRD, 1979.

[Pro89] Prott, K-J., *Effiziente LALR(1)-Analyse mit Bestimmung sicherer Anknüpfungspositionen in einem Parsergenerator*, Universität-GH Paderborn, Diplomarbeit, 1989.

[WaG84] Waite, W. M. and Goos, G., *Compiler Construction*, Springer Verlag, New York, NY, 1984.

[WHK88] Waite, W. M., Heuring, V. P. and Kastens, U., *Configuration Control in Compiler Construction*, in International Workshop on Software Version and Configuration Control '88, Teubner, Stuttgart, 1988.