Computer Science Journal of Moldova, vol.**10**, no.2(29), 2002

Experimental functional realization of attribute grammar system

I. Attali K. Chebotar N. Meergus

Abstract

In this paper we present an experimental functional realization of attribute grammar(AG) system for personal computers. For AG system functioning only Turbo Prolog compiler is required.

The system functioning is based on a specially elaborated metalanguage for AG description, universal syntactic and semantic constructors. The AG system provides automatic generation of target compiler (syntax–oriented software) using Turbo Prolog as object language.

Keywords: context free grammar, attribute grammar, functional programming, semantic attributes, compiler, metacompiler, syntactic $LALR(SLR)$ constructors, lexical and syntactic analysis, semantic evaluation.

Introduced by D.E. Knuth in 1968 [1] as formalism for specifying semantic of programming languages attribute grammars have proved to be a useful tool for design and implementing compiler, syntax–oriented and graphical editors, symbolic computations, natural language processing and more generally any syntax–oriented computation [5, 6]. The most significant application of AG are the attribute grammars systems [5] used for automatic design and realization of syntax–directed computations. The AG systems are very useful in a fast elaboration of programming systems (software) prototypes.

Inspired by the main results of works [3, 4] we present an experimental functional realization of AG system for personal computers with restricted resources. For AG system functioning only Turbo Prolog compiler is required.

1 Introduction

The system functioning is based on a specially elaborated metalanguage for *AG* description, universal syntactic and semantic constructors. The *AG* system provides automatic generation of target compiler (syntax–oriented software) using Turbo Prolog as object language.

The main definitions and notations related to attribute grammars and systems are specified in section 2.

For attribute grammars description a special nonprocedural language (metalanguage) was elaborated (section 4). All program modules of the system (metacompiler, syntactic constructor and semantic evaluators generator) were written in C^{++} programming language. Metacompiler processes attribute grammar metadescription and generates syntactic and semantic data structures containing the whole information about attribute grammar which will be used by other system components. Sections 5 and 6 describe the general scheme and internal structure of the *AG* system.

The attribute grammar description and semantic functions for binary numbers definition [1] language are presented in Appendixes 1,2.

2 Definitions and notations

Introduced by D.E. Knuth in 1968 [1], attribute grammars have proved to be a useful formalism for specifying and implementing the static (context– sensitive syntax) and dynamic semantics of programming languages and more generally any syntax–oriented computation [2]. Attribute grammars form an extension of context–free grammar. The information ("meaning") is associated with programming language notions by attaching attributes to the grammar symbol representing this notation. Each attribute has a set of possible values. The semantic rules associated with the productions of the context–free grammar specify how to compute the value of certain attribute as a function of other attributes. The attributes associated with a grammar symbol are divided into two disjoint classes, the inherited and synthesized attributes. The semantic functions associated with grammar production define the computation of the synthesized attributes of grammar symbol on the left side and all the inherited attributes of the grammar symbols on the right side of the production.

Generally speaking, a synthesized attribute attached to a grammar symbol contains information concerning the syntactic subtree generated by this symbol. Inherited attributes are used for expressing the dependence of the programming language notion on its left and right contexts.

More formally, an attribute grammar is a tuple

$$AG= (G, T, F, Inh, Syn, \alpha, \tau, Sem),$$

where:

(1) $G = (V_N, V_T, P, Z)$ is *the underlying context free grammar.* V_N and V_T denote finite disjoint sets of nonterminal and terminal symbols, respectively; $V = V_N \cup V_T$. P is a finite set of productions and $Z \in V_N$ is the start symbol, which does not appear on the right side of any production. A production $p \in P$ will be denoted as

$$p : X_0 \to X_1 X_2 \ldots X_{n_p},$$

where $n_p \geq 0$, $X_0 \in V_N$, $X_j \in V$ for $1 \leq j \leq n_p$. The grammar G is assumed to be reduced in the sense that every nonterminal symbol is accessible from the start symbol and can generate a terminal string.

(2) $T = \{t_0, t_1, \ldots, t_{n_T}\}$ is *a finite set of types* (sets).

(3) $F = \{f_0, f_1, \ldots, f_{n_F}\}$ is *a finite set of functions*; $f_i : t_{i_1} \times t_{i_2} \times \ldots \times t_{i_n} \to t_{i_0}$, where $n \leq 0$ and $t_{i_j} \in T$ for $0 \leq j \leq n$.

(4) Inh and Syn – disjoint sets of *inherited* and *synthesized* attributes, respectively. Let $A = Inh \cup Syn$.

(5) τ is *the type distribution function*, $\tau : A \to T$.
$\tau(a) = \{t\}$ means that the attribute a will be attached as a variable of type t.

(6) α is *the attribute distribution function*, $\alpha : V \to 2^A$.
$\alpha(X) = \{a_1, a_2, \ldots, a_k\}$ denotes that symbol $X \in V$ has the attributes $a_1, a_2, \ldots, a_k$. $Inh(X) = Inh \cap \alpha(X)$, $Syn(X) = Syn \cap \alpha(X) . \alpha(X) = Inh(X) \cup Syn(X)$. A set of inherited attributes instances $IO(X_i) = \{a.i \mid a \in Inh(X_i)\}$ and a set of synthesized attributes instances $SO(X_i) = \{a.i \mid a \in Syn(X_i)\}$ is attached to each symbol X_i $(0 \leq i \leq n_p)$ of syntactic rule p. Let $AO(X_i) = IO(X_i) \cup SO(X_i)$, $IO(p) = \bigcup\limits_{i=0}^{n_p} IO(X_i)$. $SO(p) = \bigcup\limits_{i=0}^{n_p} SO(X_i)$. $AO(p) = IO(p) \cup SO(p)$. The instances of attributes $In(p) = IO(X_0) \bigcup\limits_{i=1}^{n_p} SO(X_i)$ and $Out(p) = SO(X_0) \bigcup\limits_{i=1}^{n_p} IO(X_i)$ we define as input and output instances for rule p, respectively.

(7) The $Sem(p)$ for each rule p defines all attributes instances from $Out(p)$ semantic rules for their evaluation.

$$\begin{gathered} Sem(p) = \{a_0.i_0 = f(a_1.i_1, a_2.i_2, \ldots, a_k.i_k) \text{ or} \\ a_0.i_0 = a_1.i_1 \text{ or} \\ a_0.i_0 = constant \mid a_j.i_j \in AO(p), 1 \leq j \leq k, f \in F, \\ f : \tau(a_1) \times \tau(a_2) \times \ldots \times \tau(a_n) \to \tau(a_0)\} \end{gathered}$$

The semantic rule $a_0.i_0 = a_1.i_1$ (copy rule) defines a simple value transfer, and rule $a_0.i_0 = constant$ is used to assign an initial value to attribute instance $a_0.i_0$.

Semantic rules induce on $AO(p)$ an order of computation called local dependency relation $D(p)$ defined as follows: $a_0.i_0 D(p) a_1.i_1$ ($a_0.i_0$ depends on $a_1.i_1$) if and only if there is a semantic rule $a_0.i_0 = f(\ldots, a_1.i_1, \ldots)$ or $a_0.i_0 = a_1.i_1$ in Sem(p). The *local dependency graph* D_p of production p is the graph of relation $D(p)$. Attribute grammar AG is said to be in *normal form* if the condition $(a_j.i_j) \in In(p)$ holds for $1 \leq j \leq k$ for every semantic rule. It is easy to transform every evaluation rule to the normal form. In the sequel, we shell consider only normalized AG.

Given a derivation tree $\mathcal{T}$, attributes instances are attached to the nodes in the following way: if node u is labelled with grammar symbol X, then for each attribute $a \in \alpha(X)$ an instance $a.u$ of a is attached to u. Let u_0 be a node, p the production at u_0, and $u_1, u_2, \ldots u_{n_p}$ the sons of u_0. An attribute evaluation instruction

$$a_0.u_{i_0} = f(a_1.u_{i_1}, a_2.u_{i_2}, \ldots, a_k.u_{i_k})$$

is associated with attribute instance $a_0.u_{i_0}$, if the attribute evaluation rule

$$a_0.i_0 = f(a_1.i_1, a_2.i_2, \ldots, a_k.i_k)$$

is associated with production p.

We denote by $D_{\mathcal{T}}$ the *compound dependency graph* obtained by pasting together the graphs D_p corresponding to each production in the tree and by $W_{\mathcal{T}}$ the set of all *attribute instances* $a_j.u_{i_j}$ in the tree $\mathcal{T}$. An attribute grammar is *well–formed* or *non–circular* if and only if for every tree $\mathcal{T}$ the $D_{\mathcal{T}}$ is cycle free.

The task of an *attribute evaluator* is to compute the values of all attribute instances in $W_{\mathcal{T}}$. In general the order of evaluation is free with the only restriction that an attribute evaluation rule cannot be executed before its arguments are evaluated. Initially the values are assigned only to inherited attribute instances attached to the start symbol and to the terminal leaves (determined by parser).

The *meaning* or *semantic value* of $\mathcal{T}$ are the values of all attribute instances in $W_{\mathcal{T}}$ or of a distinguished subset of them, generally the synthesized attribute (attributes) of the root of $\mathcal{T}$.

3 Example

To illustrate the basic definition and our attribute system functioning we present an example initially appeared in the Knuth's original paper [1]. The purpose of this AG is to give a precise definition of rational numbers written in fixed–point binary notation.

3.1 The underlying grammar

$G = (V_N, V_T, P, Z)$, where $V_N = \{Z, L, B\}$, $V_T = \{"0", "1", "."\}$. B represents a single bit, L represents a list of consecutive bits, and Z, which is the start symbol, represents a whole number.

$$\begin{aligned} P = \{0 : B &::= "0", & 3 : L &::= LB, \\ 1 : B &::= "1", & 4 : Z &::= L, \\ 2 : L &::= B, & 5 : Z &::= L"."L\}. \end{aligned}$$

3.2 Types

$T = \{real, integer\}$.

3.3 Functions

$F = \{power2, sum, inc, neg\}$, where
$power2 : integer \rightarrow integer$, $power2(x) = 2^x$,
$sum : real \times real \rightarrow real$, $sum(x, y) = x + y$,
$inc : integer \rightarrow integer$, $inc(x) = x + 1$,
$neg : integer \rightarrow integer$, $neg(x) = -x$.

3.4 Attributes

$Inh = \{s\}$, $Syn = \{v, l\}$, where inherited attribute s ("scale") is used to compute the value of bit "1" as 2^s, synthesized attribute l ("length") is used to compute the length of binary string L and synthesized attribute v ("value") represents the decimal value of the binary substring generated by nonterminals L and Z.

3.5 Attributes type definition

$\tau(S) = integer$, $\tau(l) = integer$, $\tau(v) = real$

3.6 Attributes distribution

$\alpha(B) = \{v, s\}$, $\alpha(L) = \{v, l, s\}$, $\alpha(Z) = \{v\}$

3.7 Semantic rules

$$\begin{aligned} Sem(0 : B ::= "0") &= \{v.0 = 0\}, \\ Sem(1 : B ::= "1") &= \{v.0 = power2(s.0)\}, \end{aligned}$$

$$\begin{aligned}
Sem(2 : L ::= B) &= \{v.0 = v.1,\\
&\quad\; l.0 = 1,\\
&\quad\; s.1 = s.0\},\\
Sem(3 : L ::= LB) &= \{v.0 = sum(v.1, v.3),\\
&\quad\; l.0 = l.1,\\
&\quad\; s.1 = inc(s.0),\\
&\quad\; s.3 = s.0\},\\
Sem(4 : Z ::= L) &= \{v.0 = v.1,\\
&\quad\; s.1 = 0\},\\
Sem(5 : Z ::= L"."L) &= \{v.0 = sum(v.1, v.3),\\
&\quad\; s.1 = 0,\\
&\quad\; s.3 = neg(l.3)\}.
\end{aligned}$$

3.8 Local dependency graphs

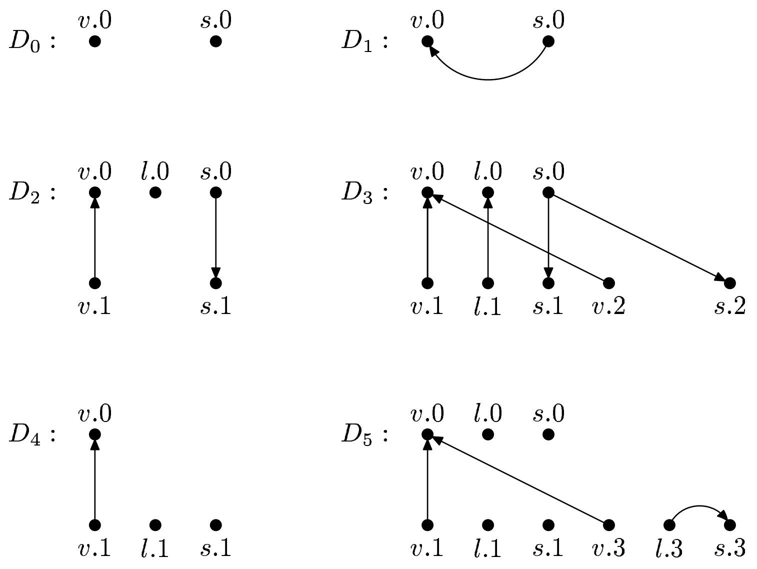

Figure 1. Local dependency graphs

3.9 Compound dependency graph and attributes evaluation

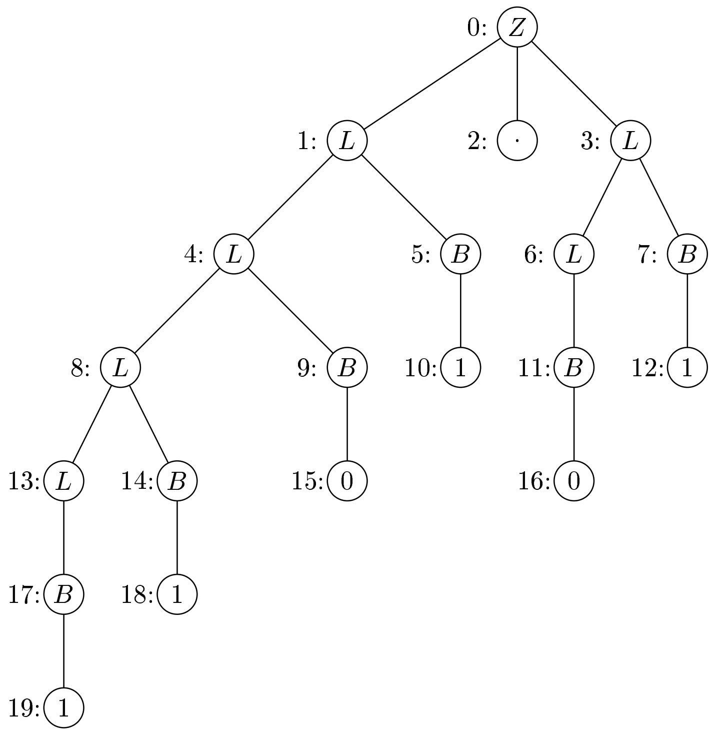

Figure 2. Labelled derivation tree for a binary number 1101.01

The figures 2,3 represent labelled derivation tree, compound dependency graph with evaluated attributes instances for a binary number 1101.01, respectively.

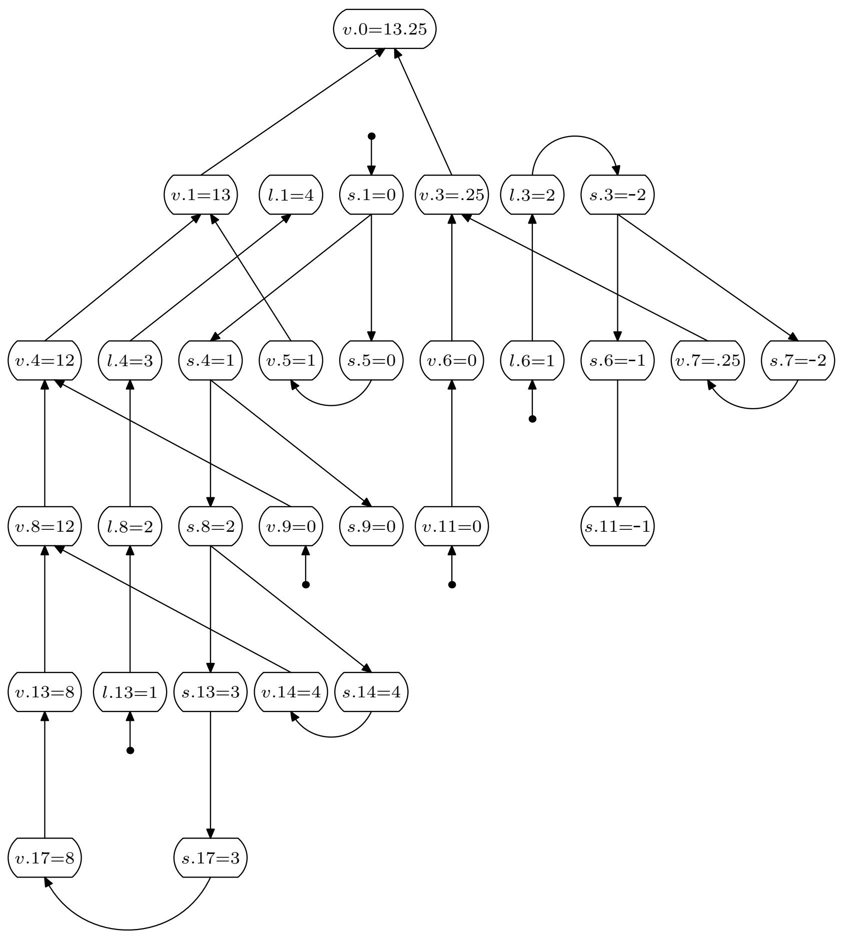

Figure 3.Compound dependency graph and attributes evaluation

4 Metalanguage for attribute grammars description

To describe attribute grammars we proposed a nonprocedural high level language that is an extension of BNF (Backus normal forms) by addition of attributes descriptions and semantic functions. The classic attribute grammar definition was kept as far as it was possible and only the elements which are necessary for tie (binding) with object language were introduced.

The rules of entering metalanguage tokens follow:

- nonterminal symbols must be written as a sequence of symbols placed in corner brackets " ⟨" and " ⟩" . For instance, ⟨*sequence of statements* ⟩.
- terminal symbols which are delimiters or regular expressions consist only of small Latin letters. For instance, *identifier*,*number*.
- terminal symbols which are reserved words of input language must be written in quotation marks. For instance, *"begin"*,*"while"*.
- attributes may consist of Latin letters and underline signs. For instance, *list_of_names*, *value*.
- the semantic functions names and constants are written in accordance with the rules for functions and constants in object language.
- the metalanguage reserved words (INHERITED, SYNTHESIZED) must be written with capital letters and wholly.

In general case the language description contains three sections: the *attributes description section*, the *attributes distribution section* and the *rule section*. Below we shall adduce every section definition and illustrate them using examples from description of attribute grammar of binary numbers [1]. The attributes section consists of subsections of inherited and synthesized attributes description. Any or both (in case of the specification of input language only) subsections may be missing. Subsections begin with keywords "INHERITED ATTRIBUTES" or "SYNTHESIZED ATTRIBUTES". After them follow several descriptions of attribute in the following format:

$$attribute_1, attribute_2, \ldots, attribute_n : type;$$

The attributes must be defined here with the same types as they are defined as arguments or results in semantic functions. All used attributes must be described. In our example the attributes description section may be as follows:

INHERITED ATTRIBUTES
$s : integer;$
SYNTHESIZED ATTRIBUTES
$v : real;$
$l : integer;$

The attributes distribution section begins with "SYMBOLS" keyword. It must contain the definitions of grammar symbols (nonterminals and terminals which are regular expressions) and attach attributes in the following format:

$$symbol_1, symbol_2, \ldots, symbol_n : attribute_1, attribute_2, \ldots, attribute_n;$$

The first symbol in the description section should be the start symbol of the grammar. For example:

SYMBOLS
$< Z >: v;$
$< L >: v, l, s;$
$< B >: v, s;$

The rule's section begins with "RULES" keyword. It contains syntax rules and corresponding rules of attributes evaluation. Syntax rules are written in BNF:

$$< symbol_0 > := < symbol_1 >< symbol_2 > \ldots < symbol_n >$$

After every syntax rule follow semantic rules which define dependencies and methods for attributes evaluation. The semantic rule has the following format:

$$
\begin{gathered}
attribute_0.symbol\_number_0 = \\
function(attribute_1.symbol\_number_1, attribute_2.symbol\_number_2, \\
\ldots, attribute_k.symbol\_number_k) \\
\text{or} \\
attribute_0.symbol\_number_0 = attribute_1.symbol\_number_1 \\
\text{or} \\
attribute_0.symbol\_number_0 = constant
\end{gathered}
$$

The syntax and semantic parts of the rule are not separated by any delimiters; and the whole rule finishes with full stop. In our example this section will contain the following:

```
RULES
< B >:= "0"
      v.0 = 0.
< B >:= "1"
      v.0 = power2(s.0).
```

The default rule:
Some of semantic rules may be missed if they define simple value transfer of the inherited attribute of the goal of the syntax rule to the namesake attribute of leftmost nonterminal in the right part of the rule which has such an attribute. By analogy, one may miss the assignment to the synthesized attribute of the goal of the syntax rule of the value of the namesake attribute of the rightmost nonterminal in the right part of the rule which has such an attribute. This agreement reflects the natural information transfer order which corresponds to transference of the semantic information from up to bottom and from left to right. The adopted agreement makes the attribute grammar description more compact.

For example, in the syntax rule

$$< L >:=< B >$$

may be missed semantic rules

$$v.0 = v.1$$
$$s.1 = s.0$$

Semantic rules for all the rest of synthesized attributes of the goal and inherited attributes of symbols in the right part must be defined explicitly.

5 General scheme and internal structure of the system

For attribute grammar description a specially elaborated nonprocedural language (metalanguage) is used. All program modules of the system (metacompiler, syntactic constructor and semantic evaluators generator) were written in C^{++} programming language. Metacompiler processes attribute grammar metadescription and generates syntactic and semantic data structures containing the whole information about attribute grammar (from attributes description, attributes distribution and rules sections) which will be used by other system components. Using syntactic information generated by metacompiler the syntactic constructor generates SLR(1) or LALR(1) parsing tables. These tables are rewritten as Prolog facts that will be used directly during parsing. The facts format and their meaning will be examined in details in 6. The semantic evaluator generator using syntactic and semantic information of the input language generates universal attributes instances evaluation program for arbitrary parsing tree. The semantic evaluator generator will be examined in details in 7.

5.1 Object language choice

The generation of all program modules needed for target compiler assembling is made using Turbo Prolog as the object language. Its data types and control structures are convenient for translation algorithms realization. The recursive process of attributes evaluation can be naturally defined in Prolog thanks to its recursive ideology. The compound attribute dependency graph may be represented as a structure of recursive calls of Prolog predicates computing the values of attribute instances. Such a representation of dependency graph and attribute's instances as well as the existence of Prolog variables only in the time of proofing predicate where it is defined allows an effective space allocation algorithm.

Really, in every moment of time not the whole dependency graph and not all attribute instances are stored in memory but only the way from root's attribute to currently computed attribute's instance (stack of predicate calls). The generated Prolog program representing compiler's text is assembled from two main parts: common for all generated compilers managing part and concrete part depending on input language. The general attribute system structure is shown in figure 4.

5.2 Common managing part

The common standard compiler's part contains universal lexical and syntax analyzers, procedure for parsing tree building and memory space management, procedure for attributes evaluator at lexical analysis time. The output of analyzers are the parsing tree and list of values of attributes which are regular expressions in order to parsing tree walking from top to bottom and from left to right. These data structures are transformed then into a set of facts, determining the production rules that were applicated to nonterminal nodes and the attribute values at terminal nodes. The constructed tree does not contain keywords and delimiters which do not contain any semantic information.

The node's address in this tree is represented by a list. The first element of this list is ordinal number of this node as direct ancestor

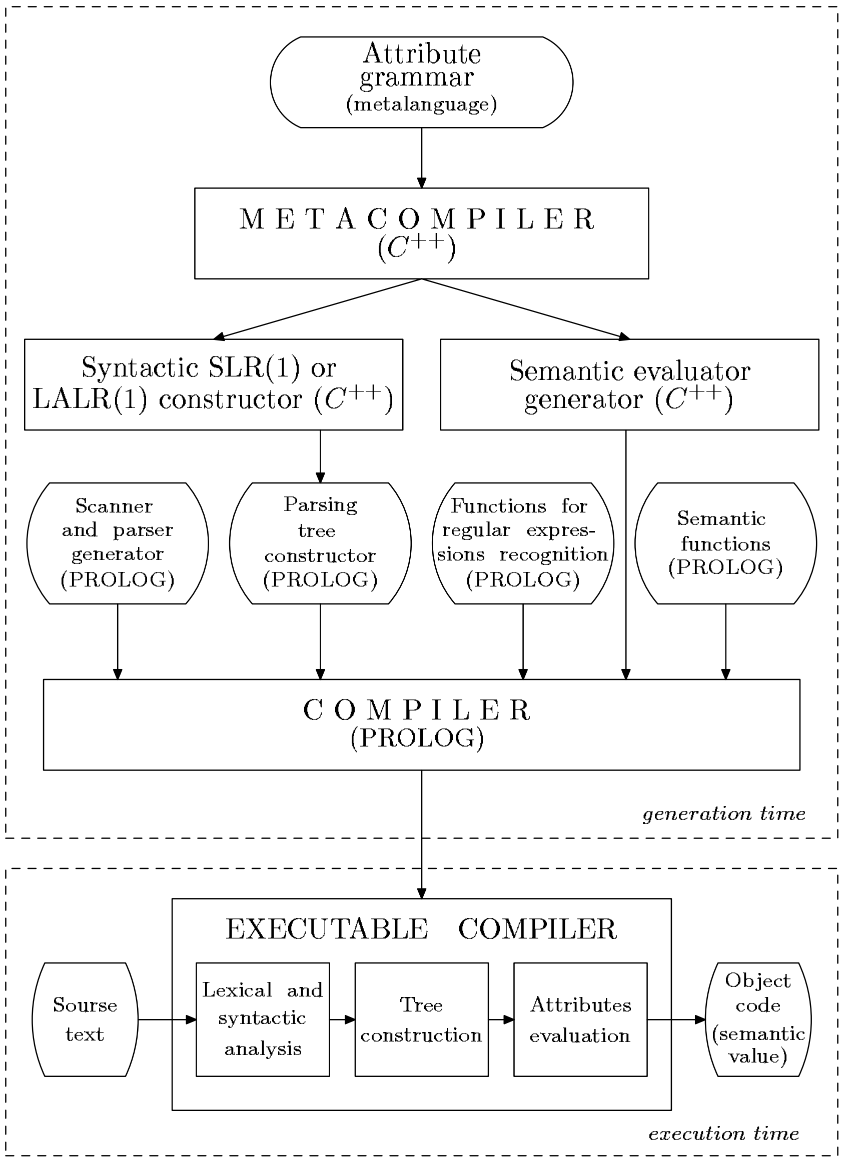
Attribute grammar (metalanguage)
M E T A C O M P I L E R (C^{++})
Syntactic SLR(1) or LALR(1) constructor (C^{++})
Semantic evaluator generator (C^{++})
Scanner and parser generator (PROLOG)
Parsing tree constructor (PROLOG)
Functions for regular expressions recognition (PROLOG)
Semantic functions (PROLOG)
C O M P I L E R (PROLOG)
generation time
EXECUTABLE COMPILER
Sourse text
Lexical and syntactic analysis
Tree construction
Attributes evaluation
Object code (semantic value)
execution time

Figure 4.General attribute system structure of the father–node which is addressed with the tail of the list. Such representation of the tree is very convenient because from any node's address it is very simple to form address of his father (rejecting first element) and addresses of its direct ancestors (adding ancestor ordinal number to the head of list). In figure 5 the tree and its nodes' addresses for parsing the "1101.01" sequence (the decimal point is missed) is shown. The address of tree root is [].

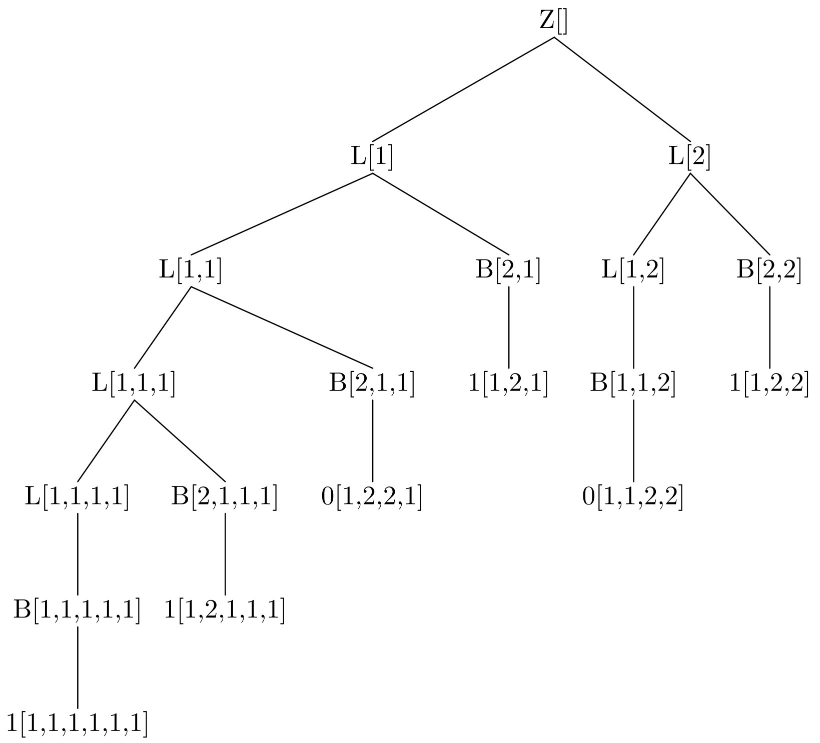

Figure 5.Derivation tree and nodes' addresses for a binary number 1101.01

5.3 Concrete part, depending on input language

The following procedures and data structures are specific for the concrete language:

- written by user procedures for recognizing tokens which are regular expressions and for computing corresponding attributes;
- SLR(1) or LALR(1) syntactic tables;
- written by user semantic functions;
- attributes evaluator;
- formalized record of syntax rules, keywords, terminals and delimiters.

All predicates used for regular expressions parsing (tokens) have a fixed argument structure. The first argument contains the input string starting with parsed token, and the last one contains parsed token length. The rest of arguments are values returned by predicate for the token attributes. The predicate may be proofed if and only if there is an expected token at the beginning of the string. For instance, the procedure for parsing one–letter name of the variable which is terminal identifier and for evaluating its single attribute which is simply the name of this variable will be achieved by proofing the following predicate:

PREDICATES

$\ldots$

$identifier(string, char, integer)$

$\ldots$

CLAUSES

$\ldots$

$identifier(S, Ch, 1) :=$

$\qquad frontchar(S, Ch, \_), Ch >=' A', Ch <=' Z'.$

$\ldots$

The predicate evaluation of the semantic function's result have to be the namesake of this function but to have one more argument. This additional argument must be the first in arguments list and it is used for returning function's value. For example, the corresponding to semantic rule $v.0 = sum(v.1, v.2)$ semantic function sum (addition) must be defined as follows:

```
    PREDICATES
          ...
   sum(real, real, real)
          ...
CLAUSES
          ...
    sum(F, X, Y) := F = X + Y.
          ...
```

The Appendix 1 contains all predicates for regular expressions parsing and for evaluating semantic function for one example the integral definition.

6 Lexical–syntactic construction

The system can process grammars from SLR(1) or LALR(1) classes. The syntactic constructor basing on the information about grammar's syntax generated by metacompiler executes LR–tables for parsing. These LR–tables are decomposed and every meaning sign (shift, reduction and transition) is written as Prolog fact. So, the Prolog program generated by the syntactic constructor contains three types of predicates:

$\_shift(Table, Symbol)$	– shift
$\_reduce(Table, Symbol, RuleNumber)$	– reduction
$\_go(TableOld, Symbol, TableNew)$	– transition

These facts are included into executable compiler text and together with universal LR–analyzer form syntactic ones. The lexical analyzer is assembled from three parts: written by user predicates for parsing the terminals which are regular expressions, generated by metacompiler list of keywords (terminals) and the universal part which generates the sequence of first two calls. The generated by metacompiler keywords list is stored in facts as

$$\_term(Number, Keyword)$$

where $Number$ is the $Keyword$ ordinal number in attribute grammar description.

Using these terms and user written functions for token the parsing lexical analyzer generates for input string a set of facts:

$$tx(Position, Attribute_1, \ldots, Attribute_n)$$

The fact name consists from prefix "t" and the ordinal number "x" of terminals in attributes distribution section. These facts keep the information about the attribute values attached to tree leafs. The syntactic analyzer generates the input string right parse which is used by tree constructor to create the parse tree in a form of Prolog facts stored in Prolog dynamic database. These facts look as follows:

$$use\_rule(Position, Rule).$$

The $Position$ argument contains the address of the node in the tree. The $Rule$ argument is the ordinal number of the grammar rule used by parser in $Position$ node.

7 Semantic evaluator

The attribute evaluator is the main and the most important part of the generated compiler. The evaluator is a Prolog program which releases the recursive procedure which computes the value of all attributes instances attached to the derivation tree.

Any attribute instance in this tree can be located unambiguously by the address of the tree node to which it belongs and by its name. For every attribute an individual predicate is generated. Such a predicate receives as an argument the attribute's position in the tree. Using obtained by syntactic analyzer information about production applicated at every tree node rule it can easy determine which semantic rule must be used for given attribute evaluation. For the synthesized attribute the corresponding semantic rule must be taken from the attribute grammar rule which was used during parsing in the node to which attribute it belongs. For an inherited attribute the corresponding rule must be found in the rule used during parsing in ancestor node. The positions of the neighbor attributes which are necessary for attribute evaluation are determined simply from its position in parsing tree.

In the generated by the system evaluator text there exists one predicate for every attribute defined in attributes description section. The predicates names consist of prefix i (for inherited attributes) or s (for synthesized ones) and an ordinal number of the attribute in the corresponding section of attribute types definition. For example, in the evaluator generated for the binary numbers attribute grammar the following predicate computes the attribute v value:

$$\_s1(Attribute, Position)$$

If at least in one grammar rule the attribute enters in the right part of more than one semantic rule then the predicate responsible for this attribute evaluation stores its value defined by the first call and later uses this stored value. This value is stored as a fact in Prolog dynamic database. For example, the inherited attribute s is used in one of the rules for the evaluation of two neighbor attributes (in the rule $< L >$ $::= < L > < B >$ $s.1$ enters in evaluation of both $s.2$ and $s.3$). After the first computation its result will be stored in the fact
$\_i1s(Attribute, Position)$.
For attribute evaluation the corresponding predicate fulfils the following steps:

- finds the used during parsing grammar rule at this position using facts $use\_rule(Position, Rule)$ stored in dynamic database by syntax analyzer;

- calls the predicates which return values of all attributes instances used as arguments;
- calls the predicates which are a simply Prolog semantic function transcription;
- stores the returned value for later calls if such one is possible.

For our example for evaluating inherited attribute s the following program fragment will be generated:

```
/ * s * /
_i1(A, P)            :- _i1s(A, P).
_i1(A, [N|P])        :- use_rule(P, R), _i1c(R, A, [N|P]).
_i1c(2, A, [1|P])    :- _i1(A, P).
_i1c(3, A, [1|P])    :- _i1(A, P), !, inc(A, A1), assert(_i1s(A, 1, [P])).
_i1c(4, 0, [1|_]).
_i1c(5, 0, [1|_]).
_i1c(5, A, [2|P])    :-_s2(A, [2|P]), !, neg(A, A1), assert(_i1s(A, [2|P])).
```

For the evaluation of the whole input string the semantic values must be evaluated for all root synthesized attributes and the final user procedure is to be executed.

For example if a_1, a_2, a_3 are the synthesized attributes of start symbol the final user predicate may be
_eval(F) :- _s1(A1, []), _s2(A2, []), _s3(A3, []), _finish(F, A1, A2, A3).
where F is the output file. For our example:
_eval(F) :- _s1(A1, []), _finish(F, A1).
$_finish$(_, X):-write(X).

8 Program executing

The integral compiler execution for the input string is presented in this section. The whole executing process may be divided into two parts: parsing execution and attribute evaluation.

8.1 Parsing execution

The right parse and corresponding set of Prolog facts are generated during parsing. For "1101.01" input string the following structures are obtained :
Right parse: $[5,3,1,2,0,3,1,3,0,3,1,2,1]$.
Prolog facts in dynamic database:
$use\_rule([],5)$.
$use\_rule([2],3)$.
$use\_rule([2,2],1)$.
$use\_rule([1,2],2)$.
$use\_rule([1,1,2],0)$.
$use\_rule([1],3)$.
$use\_rule([2,1],1)$.
$use\_rule([1,1],3)$.
$use\_rule([2,1,1],0)$.
$use\_rule([1,1,1],3)$.
$use\_rule([2,1,1,1],1)$.
$use\_rule([1,1,1,1],2)$.
$use\_rule([1,1,1,1,1],1)$.

8.2 Attribute evaluation

Attribute evaluation is initiated by the predicate
$\_eval(F) :- \_s1(A1,[]),_f inish(F,A1)$.
The attribute evaluation process begins as follows:

$$\begin{array}{ll}
\_s1(A,[]) & \text{:-}use\_rule([],5), \_s1c(5,A,[]) \\
\_s1c(5,A,[]) & \text{:-}\_s1(A1,[1]),_s 1(A2,[2]),!,sum(A,A1,A2). \\
\_s1(A,[1]) & \text{:-}use\_rule([1],3),_s 1c(3,A2,[1]). \\
\_s1c(3,A,[1]) & \text{:-}\_s(A1,[1,1]),_s 1(A2,[2,1]),!,sum(A,A1,A2).
\end{array}$$
...

In a more comprehensive form this process is given below. The indents reflect descents and ascents in parsing tree:
v in [] (5 rule) = v in [1] + v in [2]

```
  $v$ in $[1]$ (3 rule ) $= v$ in $[1,1] + v$ in $[2,1]$
    $v$ in $[1,1]$ (3 rule ) $= v$ in $[1,1,1] + v$ in $[2,1,1]$
      $v$ in $[1,1,1]$ (3 rule ) $= v$ in $[1,1,1,1] + v$ in $[2,1,1,1]$
        $v$ in $[1,1,1,1]$ (2 rule ) $= v$ in $[1,1,1,1,1]$
          $v$ in $[1,1,1,1,1]$ (1 rule ) $= 2^{(s\ in\ [1,1,1,1,1])}$
          $s$ in $[1,1,1,1,1]$ (2 rule ) $= s$ in $[1,1,1,1]$
        $s$ in $[1,1,1,1]$ (3 rule ) $= s$ in $[1,1,1] + 1$
      $s$ in $[1,1,1]$ (3 rule ) $= s$ in $[1,1] + 1$
    $s$ in $[1,1]$ (3 rule ) $= s$ in $[1] + 1$
  $s$ in $[1]$ (5 rule ) $= 0$
    $s$ in $[1,1] = 1$
      $s$ in $[1,1,1] = 2$
        $s$ in $[1,1,1,1] = 3$
          $s$ in $[1,1,1,1,1] = 3$
          $v$ in $[1,1,1,1,1] = 8$
        $v$ in $[1,1,1,1] = 8$
        $v$ in $[2,1,1,1]$ (1 rule ) $= 2^{(s\ in\ [2,1,1,1])}$
      $s$ in $[2,1,1,1]$ (3 rule ) $= s$ in $[1,1,1]$
    $s$ in $[1,1,1] = 2$ (saved in database)
      $s$ in $[2,1,1,1] = 2$
      $v$ in $[2,1,1,1] = 4$
    $v$ in $[1,1,1] = 12$
    $v$ in $[2,1,1]$ (0 rule ) $= 0$
  $v$ in $[1,1] = 12$
  $v$ in $[2,1]$ (1 rule ) $= 2^{(s\ in\ [2,1])}$
  $s$ in $[2,1]$ (3 rule ) $= s$ in $[1]$
$s$ in $[1]$ (5 rule ) $= 0$
  $s$ in $[2,1] = 0$
  $v$ in $[2,1] = 1$
$v$ in $[1] = 13$
$v$ in $[2]$ (3 rule ) $= v$ in $[1,2] + v$ in $[2,2]$
  $v$ in $[1,2]$ (2 rule ) $= v$ in $[1,1,2]$
    $v$ in $[1,1,2]$ (0 rule ) $= 0$
    $v$ in $[1,2] = 0$
    $v$ in $[2,2]$ (1 rule ) $= 2^{(s\ in\ [2,2])}$
  $s$ in $[2,2]$ (3 rule ) $= s$ in $[2]$
 $s$ in $[2]$ (5 rule ) $= -(l$ in $[2])$
 $l$ in $[2]$ (3 rule ) $= l$ in $[1,2] + 1$
  $l$ in $[1,2]$ (2 rule ) $= 1$
 $l$ in $[2] = 2$
 $s$ in $[2] = -2$
  $s$ in $[2,2] = -2$
  $v$ in $[2,2] = 0.25$
 $v$ in $[2] = 0.25$
$v$ in $[] = 13.25$
```

References

[1] Knuth D.E. *Semantics of context-free languages.* Math. Systems Theory 2(1968), pp. 127–145. Correction in: Math.Systems Theory 5 (1971), pp. 29–34.

[2] Deransart P., Jourdan M., Lorho B. *ttribute grammars, Definition, Systems and Bibliography.* LNCS 323, Springer Verlag (1988).

[3] Attali I. *Compilation de programmes Typol par Attributs Semantiques*, Doctoral thesis, University of Nice, 1989.

[4] Attali I., Chazarain J. *Functional evaluation of strongly non circular Typol specifications.* In: Attribute Grammars and their Applications. International Conference WAGA, Paris, France, September 1990, Proceedings.P.Deransart&M.Jourdan (eds.). LNCS 461, pp. 157–176, Springer Verlag, 1990.

[5] *Attribute Grammars and their Applications.* International Conference WAGA, Paris, France, September 1990, Proceedings. P.Deransart&M.Jourdan (eds.). LNCS 461, Springer Verlag, 1990.

[6] A*ttribute Grammars, Applications and Systems.* International Summer Scool SAGA, Prague, Chechoslovakia, June, 1991, Proceedings. H.Alblas&B.Melichar (eds.). LNCS 545, Springer Verlag, 1991.

[7] Marcotty M., Ledgard H.F., Bochmann G. V. *A Sampler of Formal Definitions.* Computing Surveys, v. 8, n. 2, 1976, p. 191–276.

Appendix 1. Attribute grammar for binary numbers definition

INHERITED ATTRIBUTES
$s : integer$;
SYNTHESIZED ATTRIBUTES
$v : real$;
$l : integer$;
SYMBOLS
$\langle N \rangle$: v;
$\langle L \rangle$: v, l, s;
$\langle B \rangle$: v, s;
RULES
$\langle B \rangle \longrightarrow$"0"
$v.0 = 0.$
$\langle B \rangle \longrightarrow$"1"
$v.0 = power2(s.0).$
$\langle L \rangle \longrightarrow \langle B \rangle$
$l.0 = 1.$
$\langle L \rangle \longrightarrow \langle L \rangle \langle B \rangle$
$v.0 = sum(v.1, v.2)$
$l.0 = inc(l.1)$
$s.1 = inc(s.0)$
$s.2 = s.0.$
$\langle N \rangle \longrightarrow \langle L \rangle$
$s.1 = 0.$
$\langle N \rangle \longrightarrow \langle L \rangle$"." $\langle L \rangle$
$v.0 = sum(v.1, v.3)$
$s.1 = 0$
$s.3 = neg(l.3).$

Appendix 2. Semantic functions for binary numbers definition

PREDICATES

```
    power2(real, integer)
sum(real, real, real)
inc(integer, integer)
neg(integer, integer)
_start.
_finish(string, real)
```

CLAUSES

```
_start.
 _finish(_, X)    :- write(X).
 power2(1, 0)     :- !.
 power2(Y, X)     :- X > 0, X1 = X - 1, power2(Y1, X1), Y = 2 * Y1.
 power2(Y, X)     :- X1 = X + 1, power2(Y1, X1), Y = Y1/2.
 sum(F, X, Y)     :- F = X + Y.
 inc(F, X)        :- F = X + 1.
 neg(F, X)        :- F = -X.
```

I. Attali, K. Chebotar, N. Meergus Received April 2, 2002

I. Attali
INRIA Sophia Antipolis
2004 Route des Lucioles – BP93 06902
Sophia Antipolis Cedex FRANCE

K. Chebotar, N. Meergus
Institute of Mathematics and Computer Science
5 Academiei, Kishinev
MD 2028 MOLDOVA