Attribute Grammars in Prolog

M.J. Walsteijn and M.F. Kuiper.

RUU-CS-86-14
September 1986

Rijksuniversiteit Utrecht

Vakgroep informatica

Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Attribute Grammars in Prolog

M.J. Walsteijn and M.F. Kuiper.

Technical Report RUU-CS-86-14
September 1986

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands

1. Introduction

In this report it is shown how to implement attribute grammars (AG's) in Prolog. Two methods are proposed to systematically convert an AG into Prolog. The reasons for doing so are three-fold.

First Prolog is a descriptive programming language. A Prolog program can be executed directly. Attribute grammars are also descriptive, they do not specify how attributes are to be computed. However, there don't exist interpreters for attribute grammars.

The second reason for implementing AG's in Prolog stems from our interest in attribute evaluation schemes that don't perform a tree walk on a structure tree.

The third reason is the close connection between attribute grammars and logic programs as noted by [Derensant & Maluszynski]. They used this connection to study properties of Prolog programs and to derive efficient strategies for evaluating Prolog programs.

Finally we want to see whether Prolog can be used for the rapid prototyping of compilers.

The reader is assumed to be familiar with Prolog [Clocksin & Mellish]. However, the Grammar Rule Notation of Prolog will be explained.

The rest of this paper is organized as follows: In section 2 the grammar rule notation of Prolog is explained. In section 3 the translation of this notation into ordinary Prolog and the way to use the parser defined by the grammar in this notation is discussed. In section 4 a straightforward translation of AG's into Prolog and the problems that arise are discussed. In section 5 two methods are proposed that handle every Well Defined Attribute Grammar correctly. Section 6 briefly discusses ambiguous grammars. Section 7 compares our results and implementations with other work and lists some conclusions. Section 8 contains a list of literature. Appendix A contains a large AG and Appendix B its translation into Prolog.

2. Prolog and its Grammar Rule Notation

In this section we give a description of Prolog's Grammar Rule Notation that is sufficient to understand the implementation of AG's. Grammar rules, also called Metamorphosis Grammars [Colmerauer], are a way to specify parsers in Prolog. They are not a functional extension to Prolog. A grammar rule can always be written as a normal, but somewhat complicated Prolog clause.

We will use the lexical and syntactical conventions of C-Prolog [Pereira]. The grammar rules obey the following B.N.F. grammar:

```
grammar_rule ::= grammar_head,'-->',grammar_body,'.'
grammar_head ::= non_terminal
grammar_body ::= grammar_body,',',grammar_body |
                 grammar_body_item
grammar_body_item ::= non_terminal |
                      terminal |
                      '{',prolog_goals,'}' |
```

'!'

A non-terminal consists of a name, called functor in Prolog, and a number of arguments (which will be interpreted as attributes later). The arguments are structured like Prolog variables. In C-Prolog variables must begin with a capital or an underscore.

A terminal indicates a number of words that may occupy part of the input sequence. It takes the form of a Prolog list: items, separated by commas and enclosed by square brackets (and [] is a list), that are to be matched against words in the inputsequence as they appear in the order given. A Prolog item can be a Prolog variable or a Prolog atom. A Prolog variable matches against any word in the inputsequence and then becomes instantiated to that inputsymbol. Prolog atoms are used for conventional terminals: a row of characters that should be recognized. It is always correct to use a charactersequence enclosed by single quotes for an atom. They may be left out in many cases.

For instance, write down ['a'] (or [a]) in order to express a terminal 'a'. A terminal 'A' should be written as ['A']. Definitions of atom, functor and variable can be found in [Clocksin & Mellish].

An example of the use of a variable will now be given:

```
identifier(X) --> [X] , { is_alpha_num(X) }.
```

Before this Prolog rule will be explained , the curly brackets ('{' and '}') deserve an explanation.

Curly brackets are used to indicate that the so called Prolog goals between them are not terminals or non-terminals and hence do not "eat" the input sequence. They are not part of the grammar, see section 3.1, but constitute what might be called semantic actions. The Prolog goals between curly brackets can be goals that perform extra tests or actions (think of semantic functions and conditions).

In the example there is one non-terminal, *identifier*, which has one attribute that contains the name of the identifier. The non-terminal has only one alternative: [X],{is_alpha_num(X)}. [X] is a terminal and X is a variable and suppose that *is_alpha_num* is a Prolog procedure, which checks whether X is an alpha-numeric identifier. If not, X is rejected and this alternative fails, i.e. is not the right one. When X (from [X]) is instantiated to a certain inputword, the other occurences of X in the whole rule and only this whole rule are instantiated to that inputword. In this way the name of the identifier is stored in the attribute of the non-terminal *identifier* and *is_alpha_num* does the checking.

Here's an example *is_alpha_num* (in Prolog):

```
is_alpha_num(X) :- name(X,[H|T]),
                   (65 =< H, H =< 90;    /* uppercase */
                    97 =< H, H =< 122),  /* lowercase */
                   digits_or_letters(T).
digits_or_letters([]).
digits_or_letters([H|T]) :- (65 =< H, H =< 90;    /* uppercase */
                             97 =< H, H =< 122;   /* lowercase */
                             48 =< H, H =< 57),   /* digit */
```

```
digits_or_letters(T).
```

3. The built-in preprocessor

3.1. The translation into ordinary Prolog

The Grammar Rule Notation described in the previous section is in fact a shorthand for an ordinary Prolog program. The program corresponding to a grammar will be explained with an example.

Consider the following grammar rule:

```
items --> item, [','], items.
```

This grammar rule is a shorthand for the following Prolog fragment:

```
items(K,N) :- item(K,L), terminal(',',L,M), items(M,N).
terminal(T,[T|Y],Y).
```

The first procedure can be read as follows: the list K is an instance of *items* followed by the list N, if the list K is an instance of *item* followed by L and L is a comma followed by M and M is an instance of *items* followed by N. The essence of all this is that items consist of an item, a comma and items. The four variables stand for successive remainders of the initial input sequence of terminal symbols represented as a Prolog list of terminal symbols.

These variables can be routinely added to the essence of procedure *items*, i.e. the grammar rule with head *items*. In fact this is what the built-in preprocessor of Prolog does. In general the preprocessor expands the Grammar Rule Notation to an ordinary Prolog program, by adding to each terminal and non-terminal of the grammar two extra parameters in the way described in the Prolog program *items*. Clearly, these two extra parameters are intended for parsing purposes only.

However the preprocessor does not expand Prolog goals between curly brackets, i.e. '{' and '}'. In other words, no extra parameters are added to these goals. Therefore, if expanding is unwanted, e.g. in the case of extra tests or actions in a grammar, such as conditions and semantic functions, curly brackets must be used.

It is important to realize that the Grammar Rule Notation of Prolog is merely syntactic sugar for the underlying clauses. But the gain in clarity by this syntactic transformation is significant. For further reading about the preprocessor and some optimizations, see [Kluzniak & Szpakowicz, chapter 3].

3.2. The parsing problem

From the evaluation strategy of Prolog and the translation described above it follows that a grammar in Prolog implements a very general parsing strategy: nondeterministic top-down parsing with backtracking (see [Aho & Ullman],[Gries]). It is possible to program much more efficient parsing algorithms in Prolog, see the discussion in [Kluzniak & Szpakowicz, 3.5.1], but this requires explicit handling of the parsing

stack, attributes, etc. Furthermore such an approach breaks with the elegant and descriptive nature of Prolog and its Grammar Rule Notation.

It should be clear by now that the grammar in Prolog is an executable program, i.e. a parser (see previous section). Such a program parses an input sentence and checks whether this sentence belongs to the language generated by the given grammar. In the rest of this section we discuss the problem of how to invoke the parser, i.e. grammar in Prolog.

The way to do this is as follows: suppose the root s of a given grammar has n arguments A1,...,An. To check whether "begin 1 end" is a sentence of the language generated by that grammar, ask the following question to Prolog:

s(P1,...,Pn,['begin','1','end'],[]).

where P1,...,Pn denote the actual parameters.

It should be clear that this is a call of one of the underlying procedures. The meaning of this Prolog goal in English is: "Does the sentence "begin 1 end" belong to the language generated by the grammar with root s if the actual parameters of the root are P1,...,Pn ? And if there are actual parameters, which are variables, which values can they have ?". Or more accurate: "Is the parser, defined by the grammar with root s, able to parse the sentence "begin 1 end" until precisely [] remains if the actual parameters are P1,...,Pn ? And if there are actual paramaters, which are variables, which values can they have ?".

An alternative is to use the built-in procedure phrase, which has two parameters: the nonterminal symbol and the input sentence. Used in the above example, one can ask the following question to Prolog (with the same meaning as above):

phrase(s(P1,...,PN), ['begin,'1','end']).

Actual parameters of the root, which are variables, will later be thought of as synthesized attributes of the root, see the next section.

4. Attributed Grammars in Prolog

In this section a mapping from attribute grammars to Prolog is defined. The mapping is only correct for 1-LAG.

4.1. A straightforward approach

As the grammar rule notation of Prolog has been explained and motivated, a straightforward method of systematically converting an AG into Prolog will be introduced.

The method will be introduced by an example. We use (pseudo-) Aladin [Kastens,Hutt & Zimmermann] to write down attribute grammars. We omit all the types from the definitions. Suppose we have the following fragment of an AG:

```
NONTERM x0 : A : INH,
           B : INH,
```

```
        C : SYNT;
RULE rule :
  x0 ::= x1,x2
STATIC
  x1.D := f(x0.A,x2.A);                    (1)
  x2.B := x0.B;                            (2)
  x0.C := g(x2.E)                          (3)
END;

NONTERM x1 : D : INH;
RULE ...
STATIC
  ...
END;

NONTERM x2 : A : SYNT,
        B : INH,
        E : SYNT;
RULE ...
STATIC
  ...
END;
```

The context-free part is easy in Prolog:

```
x0 --> x1,x2.
```

The attributes of the AG will be represented as extra arguments in Prolog. Now we first write down the framework of the grammar, leaving space for the attributes of the non-terminals.

```
x0( , , ) --> x1( ),x2( , , ).
```

In x0(, ,) the first place is reserved for x0.A, the second for x0.B, etc. Attributes with the same name (i.e. A in x0.A,x2.A) will be denoted with the number of the position in the rule, appended to the name of the attribute (thus A0,A2).

Consider (1). Because x1.D depends on x0.A and x2.A these latter have to be added to the frame:

```
x0(A0, , ) --> x1( ),x2(A2, , ).
```

Note that in C-Prolog arguments (=attributes) have to begin with a capital letter. After x0.A and x2.A have been added to the grammar then f can be handled as follows:

```
x0(A0, , ) --> x1(D1), x2(A2, , ), { f(A0,A2,D1) }          (4)
```

where f is the translated Aladin function f with output parameter D1 (In Prolog a procedure never has a result, so an extra parameter is necessary).

Continuing this way, the resulting grammar in Prolog would be:

```
x0(A0,B0,C0) --> x1(D1), x2(A2,B0,E2), {f(A0,A2,D1), g(E2,C0)}.      (5)
```

where in g C0 is an output parameter.

4.2. Problems with this approach

Even the context free part gives rise to some problems. [Kluzniak & Szpakowicz] noted that in some cases left recurson causes Prolog to recur infinitely. This problem can be solved by the well known techniques of factorization and substitution, see [Kluzniak & Szpakowicz, 3.5.2]. A more drastic, but eventually more convenient way to prevent left recursion is to keep the grammar LL(1).

Another problem occurs with Well Defined Attribute Grammars [Knuth] written this way: consider e.g. the example in the previous section. It is possible that at the time the semantic function f is encountered in (5) x0.A and/or x2.A are still uninstantiated. The semantic function cannot always be evaluated at this moment. In general, if a function possibly looks at the value of such an uninstantiated attribute, i.e. dereferences it, then the semantic function cannot be evaluated at this moment. Otherwise, if it is certain that a function never dereferences an uninstantiated variable, there's no problem and it can always be evaluated when encountered.

Examples of Prolog operators which dereference their arguments are the relational operators and the 'is' operator. The latter dereferences all variables in the arithmetical expression on the right hand side of 'is'. Such operators fail when one of their arguments is still uninstantiated.

Examples of functions that never dereference their arguments are list-constructor functions or record constructor functions. The variables in the list or record will become instantiated as soon as the argument with which it is shared becomes so.

If the reader is familiar with the evaluation strategy of Prolog, he will recognize that the situation of still uninstantiated arguments of a semantic function can never occur if the grammar is evaluable in one pass from left to right [Bochmann]. The class of grammars evaluable in one pass from left to right is too restricted in practice. More practical classes of grammars are OAG [Kastens] and ANCAG [Kennedy & Warren]. Even example (5) implies that it will not pass the left to right test, because x1.D depends on x2.A, therefore on something on the right.

In the next section we will see how to solve the problem in the case of the largest class of non circular attribute grammars, the Well Defined Attribute Grammars.

5. Two methods for Well Defined Attribute Grammars

5.1.1. First solution

In this method the attributes yield a term, which represents a value, instead of the value itself. The term is flattened by a procedure *eval* when it is certain that all attributes are instantiated. Another way to interpret this method is that there are two phases: in phase 1 the term is computed. This computation involves postponing the evaluation of semantic functions by storing the successive calls to these functions in terms (=records). Naturally the arguments are included in the calls. The postponing is

necessary to avoid dereferencing of uninstantiated variables. The uninstantiated arguments are shared with the corresponding attributes and will become instantiated automatically.

In phase 2 the term is rewritten to a value. A procedure *eval* simulates the semantic functions whose names were remembered in the term, in order to yield the intended value.

Now reconsider the previous example.

Phase 1:

Instead of making a Prolog procedure f, a Prolog term f is made (with the original number of arguments). In this case the strategy of this approach is that f will not be evaluated until all the attributes f depends on are instantiated. In the example f(x0.A,x2.A) can be written down on the place of x1.D :

```
x0(A0, , ) --> x1( f(A0,A2) ),x2(A2, , ).
```

In the same way rule (2) is added :

```
x0(A0,B0, ) --> x1( f(A0,A2) ),x2(A2,B0, ).
```

and rule (3) :

```
x0(A0,B0, g(E2) ) --> x1( f(A0,A2) ),x2(A2,B0,E2).
```

Now if one of the relevant attributes, say A0, is not yet defined when f is encountered Prolog shares this occurence of attribute x0.A with the first field of the term, because they are the same variable in one rule. This means that if x0.A becomes instantiated at some moment the first field of the record will also becomes instantiated to the same value at the same moment. This mechanism also works for x0.B and x2.B : if x0.B becomes instantiated, x2.B will too (note that Prolog is more general : it works both ways). The same applies for g.

This is precisely what we want! We don't need complex algorithms to calculate the evaluation order. The grammar does not have to be ordered or absolutely non circular, because things that are not yet defined will become so automatically while Prolog is busy parsing the rest of the sentence (if the grammar is well defined).

Phase 2:

The only thing left to do is to evaluate the attributes, where we need them, particularly the synthesized attributes of the root. Remember the attributes now contain terms instead of values. It should be pointed out that if it is known (for example in the case of inherited attributes always passed down through the parse tree) that the attributes, on which a function depends are always instantiated at a certain point, a construction like (5) may be used. If this it not the case, the term-method has to be used and therefore we have to be able to evaluate a postponed semantic function. The idea is to write an evaluation function for a term with functor f instead of writing f itself. This evaluation function simulates f. The information this function needs is stored in the term with functor f and the place where the evaluation function is called is usually near the root, because all attributes in the term will then be instantiated.

Consider the following fragment as an extension of the previous example:

```
NONTERM root : R : SYNT;
RULE rt :
  root ::= x0
STATIC
  root.R := x0.C;
  x0.A := 1;
  x0.B := 10
END;
```

The following Prolog fragment is proposed:

```
root(R0) --> x0(1,10,C1),{ eval(C1,R0) }.
```

where *eval* is a function, which acts as the semantic function, here *g*. Hence in this example *eval* has the struct *g* as an argument. Here's an example *eval* if *g* is a semantic function which adds one to its argument (assume this argument only can contain g's) :

```
eval(g(X),OUT) :- eval(X,TEMP), OUT is TEMP + 1,!.
eval(X,X).
```

In this way it is possible to define a number of operations, which can be used within semantic functions. These are the normal arithmetical operations. The names of the operations are chosen in such a way the operations don't need further explanation. Now a procedure *eval* follows, in which a number of arithmetical operations are incorporated. These operations can be used in phase 1, while *eval* itself can be used in phase 2. Naturally, this list can easily be extended with those extra arithmetical operations, which are supplied by a particular Prolog implementation or user defined operations. As one would expect *eval* has to be extended with the semantic functions, which are specific for this grammar.

```
eval(X+Y,O) :- eval(X,T1), eval(Y,T2), O is T1+T2, !.
eval(X-Y,O) :- eval(X,T1), eval(Y,T2), O is T1-T2, !.
eval(X*Y,O) :- eval(X,T1), eval(Y,T2), O is T1*T2, !.
eval(X/Y,O) :- eval(X,T1), eval(Y,T2), O is T1/T2, !.
eval(X mod Y,O) :- eval(X,T1), eval(Y,T2), O is T1 mod T2, !.
eval(eq(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1==T2, O=ok;
                    T1==T2, O=nok), !.
eval(ne(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1==T2, O=ok;
                    T1==T2, O=nok), !.
eval(gt(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1>T2, O=ok;
                    T1=<T2, O=nok), !.
eval(ge(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1>=T2, O=ok;
                    T1<T2, O=nok), !.
```

```
eval(lt(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1<T2, O=ok;
                    T1>=T2, O=nok), !.
eval(le(X,Y),O) :- eval(X,T1), eval(Y,T2),
                   (T1=<T2, O=ok;
                    T1>T2, O=nok), !.
eval(if(B,T,E),O) :- eval(B,TB),
                     (TB=ok, eval(T,O);
                      TB=nok, eval(E,O)), !.
eval(A,A).
```

An example of the use of this standard procedure *eval* is now given. Consider the following semantic function in Aladin:

```
FUNCTION semfun (attr1,attr2:attrtype)attrtype:
  IF attr1 > 0 THEN attr1 + 1
  ELSE attr2 + 1
  FI
```

and this is used in a grammar with nonterminals nt1, nt2, nt3 with attributes A,B,C respectively in the following way:

```
RULE rule:
  nt1 ::= nt2,nt3
STATIC
  A := semfun(B,C)
END;
```

Note that B and C are dereferenced in this example.

Then the translation into Prolog with the standard procedure *eval* becomes:

```
nt1( if( gt(B,0) , B+1 , C+1 ) ) --> nt2(B),nt3(C).
```

Eval has to be invoked when the actual value of attribute A is needed (normally at the root).

5.1.2. A large example

The example in this section is based on [Bochmann]. It demonstrates the handling of Algol 60 scope rules (declare after use). In Algol 60 a variable must be declared on the same level it occurs on or on a previous level. An example of a recognized sentence is:

```
( dec A , ( b , A , dec B ,) , A , dec b , b ,).
```

where dec stands for declaration and a single identifier for an applied identifier occurence. The grammar describes the generation of code. The code of a program is a list of pairs (level,displ), one pair for each applied occurence of an identifier. For the above sentence the list becomes:

```
( (0,1) (0,0) (0,0) (0,1) ).
```

The complete grammar is given in Aladin and can be found in Appendix A.

Now one rule from the Aladin grammar is taken and the translation into Prolog is shown. The rule that is chosen is *exec_stat.* For the translation of the rest of the grammar see Appendix B. In Appendix B a lexical scanner is included. It's an adapted one from [Clocksin & Mellish]. For explanation see section 5.3 of this book. This scanner can be used for almost every programming language (with some minor modifications ofcourse).

Now consider *exec_stat.*

```
NONTERM executable_statement:  used : tp_symbol_table INH,
                               code : tp_code_list SYNT;

RULE exec_stat:
  executable_statement ::= T_identifier
STATIC
  executable_statement.code := tp_code_list(allocation_of(T_identifier.name,
                                            executable_statement.used))
END;
```

Note that *executable_statement* has two attributes: an environment (*used*), which is a list, and a code list (*code*). The code of *executable_statement* is defined as the list of one code-tuple, which is generated by *allocation_of. Allocation_of* searches for the identifier in the environment. For the incorporation of *allocation_of* in *eval* see Appendix B (this is another *eval* really, because it has three arguments). If the identifier is not declared an error message is generated.

Now the translation into Prolog becomes:

```
executable_statement(USED0,[allocation_of(NAME1,USED0)]) -->
t_identifier(NAME1).
```

5.1.3. Remarks on the method

An advantage of this way of writing down the grammar is that the notation looks very natural. The grammar is very concise compared to Aladin. A (little) syntactic difference in the grammar (or the only notational overhead) is the call to *eval* near the root. Furthermore it is necessary to extend *eval* instead of writing the semantic function itself (but that is equally difficult).

If you transform the grammar into a functional program, e.g. SASL, the dependencies of the attributes can also be written down very concisely (without an *eval* call, lazy evaluation takes care of the problem of undefined attributes in the dependency set), but then a parser has to be programmed, which is a considerable effort. In Prolog the grammar, which is already very concise, is a parser itself! However this gives rise to another problem: AG's are more efficient and practical by using abstract syntax. In Prolog one uses concrete syntax.

5.2. Second solution

There is an alternative solution to the problem of uninstantiated variables at the moment of semantic function evaluation. Instead of inventing another method of evaluation and passing attributes (however totally obscured by a concise notation), a much simpler and more elegant solution exists.

Many Prolog interpreters have a built-in procedure freeze. The procedure freeze(X,P) tests whether the variable X has been bound. If so, P is executed, otherwise the pair [X,P] is placed in a freezer (P is "frozen"). As soon as X becomes bound, P becomes the next goal that will be executed. After that normal operation continues. (For further study about freeze and its implementation see [Cohen]).

It is not difficult to see how this predicate can be used in the AG's written down in Prolog: If it is possible that an argument of a semantic function is dereferenced and uninstantiated at the same time, write a freeze in front of that argument of the semantic function. Freeze is needed in the same cases as described in section 4.2. However if one would rather not think about whether or not it is necessary to put a freeze in front of an argument, just write a freeze in front of every argument of a semantic function; superfluous freezes don't harm. Use a construction like (4) of section 4.1:

x0(A0, ,) --> x1(RESULT), x2(A2, ,), {freeze(A0,freeze(A2,f(A0,A2,RESULT)))}.

In this way it is possible to write a real semantic function f instead of a function which evaluates a struct f. When an attribute in the dependency set of a semantic function is not instantiated at the time the function is encountered, the evaluation of the function will automatically be postponed (the function is frozen) and will be evaluated instantaneously when that attribute becomes instantiated. While the semantic function is postponed, the parsing of the rest of the input sentence goes on, and recontinues after the moment the attribute becomes bound and the corresponding semantic function is evaluated.

Clearly, this is a coroutine mechanism. In this way the advantages of lazy evaluation (and therefore of SASL) in the context of attribute evaluation of AG's are incorporated in Prolog.

The method of translating an AG into Prolog is simple, straightforward and most elegant this way. The reason why we didn't incorporate this method in the large example was purely pragmatic: The procedure freeze isn't incorporated in our Prolog interpreter and that's why we had to search for an alternative in order to use AG's on our interpreter (of C-Prolog).

There's still an improvement possible in this scheme, one of a syntactic nature: In [Shapiro] an alternative notation for freeze is used: the special mark '?' of concurrent Prolog. The question mark is a shorthand notation for freezes. For example, the goal P(X?,Y) can be viewed as a form of freeze(X,P(X,Y)).

In this notation scheme the construction becomes like:

x0(A0, ,) --> x1(RESULT), x2(A2, ,), {f(A0?,A2?,RESULT)}.

This completes the second solution. Despite being the most elegant solution it cannot be used in C-Prolog.

6. Ambiguous grammars

Another feature of Prolog combined with (Attribute) grammars is it's handling of ambiguous grammars. Despite the fact that ambiguity is an unwanted property in the compiler building area, applications exist in the natural language processing area (see [de Moor]). Prolog takes the leftmost alternative defined by the grammar and if the user lets Prolog know he wants another solution, Prolog takes the next possibility defined by the ambiguous grammar. An example is now given (';' means 'or'):

```
root(IN,OUT) --> (x(IN,TEMP);y(IN,TEMP)),{eval(TEMP,OUT)}.
x(A,A+1) --> [t].
y(B,B-1) --> [t].
```

If you now ask the following question to Prolog:

```
root(10,OUT,[t],[]).
```

then Prolog answers with OUT = 11. If you then ask for another solution (this is done by typing a ';') then Prolog answers with OUT = 9. This is precisely what you want if your grammar is intended to be ambiguous.

7. Conclusions and related work

It is possible to combine the advantages of AG's, which are widely accepted in the compiler building area, and those of Logic Programming which were already noted by [Warren]. Warren experienced less programming effort, less likelihood of error and maintainability of the implementation of a compiler while programming in Prolog. Moreover in Prolog the specification is the implementation, in other words Prolog is highly descriptive.

For a discussion on the practibility and efficiency of Prolog for compiler writing, see [Warren]. For a discussion of higher efficiency through parallelism see for example [Cohen].

Because a grammar in Prolog is a program at the same time, and because a program in Prolog is directly executable, it seems that Prolog (including AG's) is convenient for fast prototyping of compilers.

The main conclusion of this report must be that all Well Defined Attribute Grammars can be systematically written down in Prolog. The resulting grammar is very concise and natural, even if the predicate freeze can't be used. Furthermore, we showed that in Prolog no tree walk on a structure tree is performed, while evaluating the attributes of an attribute grammar. Nor is it necessary in Prolog to determine dependencies between attributes.

8. Literature

[Aho & Ullman]: A.V. Aho and J.D. Ullman - Principles of Compiler Design. Addison-Wesley. 1977.

[Bochmann]: G.V. Bochmann - Semantic Evaluation from Left to Right. Communications of the ACM, Volume 19, Number 2. February 1976.

[Clocksin & Mellish]: W.F. Clocksin and C.S. Mellish - Programming in Prolog. Springer Verlag. 1981.

[Cohen]: J. Cohen - Describing Prolog by its Interpretation and Compilation. Communications of the ACM, Volume 28, Number 12. December 1985.

[Colmerauer]: A. Colmerauer - Metamorphosis Grammars. In Natural Language Communication with Computer (L. Bolc, ed.), pp. 133-189. 1978.

[Deransart & Maluszynski]: P. Deransart and J. Maluszynski - Relating Logic Programs and Attribute Grammars. Technical Report 393,INRIA. April 1985.

[Gries]: D. Gries - Compiler Construction for Digital Computers. Wiley and sons. 1971.

[Kastens]: U. Kastens - Ordered Attributed Grammars. Acta Informatica 13, pp. 229-256. 1980.

[Kastens, Hutt & Zimmermann]: U. Kastens, B. Hutt and E. Zimmermann - GAG: A Practical Compiler Generator. LNCS 141. Springer Verlag. 1982.

[Kennedy & Warren]: K. Kennedy and D. Warren - Automatic generation of efficient evaluators for attribute grammars. In Proceedings third conference on POPL. pp. 32-49. 1976.

[Kluzniak & Szpakowicz]: F. Kluzniak and S. Szpakowicz - Prolog for Programmers. Apic Studies in Data Processing no. 24. Academic Press. 1985.

[Knuth]: D.E. Knuth - Semantics of Context Free Languages. Math. Syst. Theory 2, pp. 127-145. 1968.

[de Moor]: O. de Moor - Computational aspects of the Eurotra Framework (unpublished paper). Internal paper of dpt. Language Science

of the University of Utrecht. 1986.

[Pereira]: C-Prolog User's Manual Version 1.5.

[Shapiro]: E. Shapiro - Systems Programming in Concurrent Prolog.
ICOT Tech. Rep. November 1983.

[Warren]: D.H.D. Warren - Logic Programming and Compiler Writing.
Software-Practice and Experience, Vol. 10, pp. 97-125. 1980.

APPENDIX A

The example grammar in Aladin

```
% This grammar is written by Oege de Moor. (1986).

TYPE tp_definition   : STRUCT( name  : SYMB, level : INT, displ : INT);

TYPE tp_symbol_table : LISTOF tp_definition;

TYPE tp_alloc        : STRUCT( level : INT, displ : INT );

TYPE tp_code_list    : LISTOF tp_alloc;

CONST c_error_alloc  : tp_alloc( -1, -1);

TERM T_identifier: name : SYMB SYNT;

NONTERM program: code : tp_code_list SYNT;

RULE root:
  program ::= block
STATIC
  block.used := tp_symbol_table();
  block.level := 0;
  program.code := block.code
END;

NONTERM block: used  : tp_symbol_table INH,
               level : INT INH,
               code  : tp_code_list SYNT;

RULE blck:
  block ::= '(' statement_list ')'
STATIC
  statement_list.used := statement_list.updated;
  statement_list.original := block.used;
  statement_list.displ_in := 0;
  block.code := statement_list.code
END;

NONTERM statement_list: used     : tp_symbol_table INH,
                        original : tp_symbol_table INH,
                        displ_in : INT INH,
                        updated  : tp_symbol_table SYNT,
                        code     : tp_code_list SYNT,
                        displ_out: INT SYNT;

RULE stat_list1:
```

```
  statement_list ::= statement ',' statement_list
STATIC
  statement.used := statement_list[1].used;
  statement.original := statement_list[1].original;
  statement_list[2].used := statement_list[1].used;
  statement_list[2].original := statement.updated;
  statement.displ_in := statement_list[1].displ_in;
  statement_list[2].displ_in := statement.displ_out;
  statement_list[1].updated := statement_list[2].updated;
  statement_list[1].displ_out := statement_list[2].displ_out;
  statement_list[1].code := statement.code + statement_list[2].code
END;

RULE stat_list2:
  statement_list ::=
STATIC
  statement_list.updated := statement_list.original;
  statement_list.code := tp_code_list();
  statement_list.displ_out := statement_list.displ_in
END;


NONTERM statement: used     : tp_symbol_table INH,
                   original : tp_symbol_table INH,
                   displ_in : INT INH,
                   updated  : tp_symbol_table SYNT,
                   code     : tp_code_list SYNT,
                   displ_out: INT SYNT;

RULE stat1:
  statement ::= block
STATIC
  block.used := statement.used;
  block.level := (INCLUDING(block.level)) + 1;
  statement.code := block.code;
  statement.displ_out := statement.displ_in;
  statement.updated := statement.original
END;

RULE stat2:
  statement ::= id_declaration
STATIC
  statement.updated := statement.original +
                       tp_symbol_table(id_declaration.declaration);
  statement.displ_out := id_declaration.displ_out;
```

```
  id_declaration.displ_in := statement.displ_in;
  statement.code := tp_code_list()
END;

RULE stat3:
  statement ::= executable_statement
STATIC
  statement.updated := statement.original;
  executable_statement.used := statement.used;
  statement.displ_out := statement.displ_in;
  statement.code := executable_statement.code
END;


NONTERM id_declaration: displ_in    : INT INH,
                        declaration : tp_definition SYNT,
                        displ_out   : INT SYNT;

RULE decl:
  id_declaration ::= 'dec' T_identifier
STATIC
  id_declaration.declaration := tp_definition( T_identifier.name,
                                               id_declaration.displ_out,
                                               INCLUDING(block.level) );
  id_declaration.displ_out := id_declaration.displ_in + 1
END;


NONTERM executable_statement:  used : tp_symbol_table INH,
                               code : tp_code_list SYNT;

RULE exec_stat:
  executable_statement ::= T_identifier
STATIC
  executable_statement.code := tp_code_list( allocation_of(T_identifier.name,
                                             executable_statement.used))
END;


FUNCTION allocation_of(name : SYMB,
                       used : tp_symbol_table) tp_alloc :
   IF NOT EMPTY(used)
   THEN LET h : HEAD(used) IN
          IF h.name = name
          THEN tp_alloc( h.level,
```

```
                    h.displ )
      ELSE allocation_of(name, TAIL(used))
      FI
ELSE (c_error_alloc
      CONDITION FALSE
      MESSAGE "Identifier NOT declared"
     )
FI;
```

APPENDIX B

The example grammar translated into Prolog

```
/* The grammar */

program(CODE0) --> block([],0,CODE1),['.'],{eval(CODE1,[],CODE0)}.


block(USED0,LEVEL0,CODE1) --> ['('],statement_list(UPDATED1,USED0,0,UPDATED1,
                              CODE1,DISPL_OUT1,LEVEL0),[')'].


statement_list(USED0,ORIGINAL0,DISPL_IN0,UPDATED2,CODE0,DISPL_OUT2,LEVEL0) -->
  statement(USED0,ORIGINAL0,DISPL_IN0,UPDATED1,CODE1,DISPL_OUT1,LEVEL0),
  [','],
  statement_list(USED0,UPDATED1,DISPL_OUT1,UPDATED2,CODE2,DISPL_OUT2,LEVEL0),
  {append(CODE1,CODE2,CODE0)}.

/* remark 1: this construction is chosen because append doesn't dereference */

statement_list(USED0,ORIGINAL0,DISPL_IN0,ORIGINAL0,[],DISPL_IN0,LEVEL0) --> [].


statement(USED0,ORIGINAL0,DISPL_IN0,ORIGINAL0,CODE1,DISPL_IN0,LEVEL0) -->
            block(USED0,LEVEL1,CODE1),
            {LEVEL1 is LEVEL0 + 1}.

/* remark 2: this construction is chosen because it is known that LEVEL0 is */
/* instantiated here */

statement(USED0,ORIGINAL0,DISPL_IN0,UPDATED0,[],DISPL_OUT1,LEVEL0) -->
              id_declaration(DISPL_IN0,DECLARATION1,DISPL_OUT1,LEVEL0),
              {append(DECLARATION1,ORIGINAL0,UPDATED0)}.

/* idem remark 1 */

statement(USED0,ORIGINAL0,DISPL_IN0,ORIGINAL0,CODE1,DISPL_IN0,LEVEL0) -->
            executable_statement(USED0,CODE1).


id_declaration(DISPL_IN0,[struct(NAME1,LEVEL0,DISPL_IN0)],DISPL_OUT0,LEVEL0) -->
            [dec],t_identifier(NAME1),
            {DISPL_OUT0 is DISPL_IN0 + 1}.
```

```
/* idem remark 2 */


executable_statement(USED0,[allocation_of(NAME1,USED0)]) -->
                t_identifier(NAME1).



t_identifier(NAME) --> [NAME].


/* semantic functions */


printstring([]).
printstring([H|T]) :- put(H),printstring(T).

append([],L,L).
append([X|L1],L2,[X|L3]) :- append(L1,L2,L3).

eval([],C_IN,C_IN).
eval([allocation_of(NAME,[])|T2],C_IN,C_OUT) :-
                printstring("Identifier not declared: "),
                write(NAME),nl,!,eval(T2,C_IN,C_OUT).

eval([allocation_of(NAME,[struct(NAME,LEVEL,DISPL)|T])|T2],C_IN,C_OUT) :-
                !,append(C_IN,[[LEVEL,DISPL]],C_TEMP),
                eval(T2,C_TEMP,C_OUT).

eval([allocation_of(NAME1,[struct(NAME2,LEVEL,DISPL)|T])|T2],C_IN,C_OUT) :-
                eval([allocation_of(NAME1,T)|T2],C_IN,C_OUT).


/* call program */


go(CODE) :- printstring("Give program: "),nl,
          read_in(LIST),
          program(CODE,LIST,[]).


/* scanner */


read_in([W|Ws]) :- get0(C),readword(C,W,C1),restsent(W,C1,Ws).
```

```
restsent(W,_,[]) :- lastword(W),!.
restsent(W,C,[W1|Ws]) :- readword(C,W1,C1),restsent(W1,C1,Ws).

readword(C,W,C1) :- single_character(C),!,name(W,[C]),get0(C1).
readword(C,W,C2) :- in_word(C,NewC),!,
                    get0(C1),
                    restword(C1,Cs,C2),
                    name(W,[NewC|Cs]).
readword(C,W,C2) :- get0(C1),readword(C1,W,C2).

restword(C,[NewC|Cs],C2) :- in_word(C,NewC),!,
                            get0(C1),
                            restword(C1,Cs,C2).
restword(C,[],C).

single_character(44). /* , */
single_character(46). /* . */
single_character(40). /* ( */
single_character(41). /* ) */

in_word(C,C) :- C>96,C<123.  /* a b .. */
in_word(C,C) :- C>64,C<91.   /* A B .. */
in_word(C,C) :- C>47,C<58.   /* 0 1 .. */

lastword('.').
```

A test whether NAME (from t_identifier) is alphanumeric could easily be incorporated in the grammar (see section 2).