3 METAMORPHOSIS GRAMMARS: A POWERFUL EXTENSION

3.1. PROLOG REPRESENTATION OF THE PARSING PROBLEM

We shall begin with a very simple formulation of the parsing problem: given a sequence of items, find out whether it has some presupposed structure. The problem appears e.g. in programming languages when we want to make sure that some text is a syntactically valid statement. Admissible structures are usually described by a context-free grammar. As an example we shall consider the following small grammar in Backus–Naur–Form, which describes simple list expressions:

```
        < list > ::= () | ( < items > )
        < items > ::= < item > | < item >, < items >
(3.1)   < item > ::= < atom > | < list >
        < atom > ::= < letter > | < letter > < atom >
        < letter > ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
                       n | o | p | q | r | s | t | u | v | w | x | y | z
```

Terminal symbols of this grammar are small letters, round brackets, and a comma. For example, the list

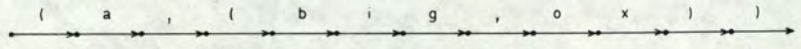
(a,(big,ox))

consists of 12 terminal symbols.

There are several commonly used methods of describing the structure of a list (or, more generally, of a valid sequence of terminal symbols). The

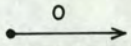
method we adopt here leads to an elegant formulation of the parsing problem in Prolog[1].

We shall depict a sequence of terminal symbols in a graph (3.2):

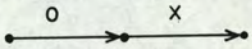
(3.2) 

Every node in this graph corresponds to a boundary between two consecutive terminal symbols; every edge connecting two nodes corresponds to the terminal symbol it is labelled with. Two edges are *contiguous* if they share a node; a sequence $e_1, \ldots, e_m$ of edges is *contiguous* if e_i and e_{i+1} are contiguous for $i = 1, 2, \ldots, m - 1$. For example, the edges labelled *b*, *i*, *g* are contiguous. The labels of contiguous edges are also *contiguous*.

A sequence of contiguous labels may constitute a whole which is meaningful in that it corresponds to the right-hand side of a production. For example, the (only) label of the one-element sequence of edges



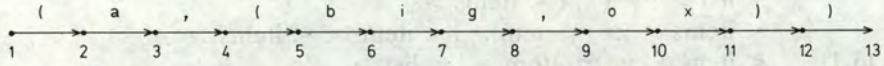
constitutes a letter; the labels of the sequence



constitute an atom.

We shall describe such meaningful combinations by connecting the extreme nodes of a contiguous sequence by an edge. The edge will be labelled with the name of an appropriate non-terminal symbol, as for example in Fig. 3.1.

To be able to represent graphs in a program, we must give each node a unique name. For example, we can name nodes with numbers:



We can represent such a graph as a set of edges, every edge expressed by a unit clause[2] that specifies the label of the edge and the names of the nodes it connects. Perhaps the most compact way is to use the label as the clause name, e.g.

```
atom( 9, 11 ).
letter( 9, 10 ).
o( 9, 10 ).
```

[1] This manner of presentation is due to Colmerauer; it was also used by Kowalski (1979b).

[2] For other ways of representing graphs in Prolog, see Sections 4.2.4 and 4.4.3.

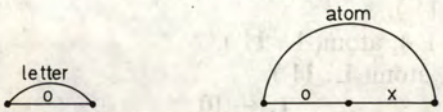


FIG. 3.1 Meaningful combinations of edges.

We now observe that clauses which represent edges labelled with non-terminal symbols might be derived from those corresponding to terminal symbols, by virtue of general structural relationships inherent in the grammar. The reasoning would be roughly as follows:

letter(9, 10) because o(9, 10): o is a letter;
letter(10, 11) because x(10, 11): x is a letter;
atom(10, 11) because letter(10, 11): a letter makes an atom;
atom(9, 11) because letter(9, 10) and atom(10, 11): a letter and an atom make an atom.

Relationships of this kind can be generalized in a straightforward manner, e.g.

```
letter( K, L ) :- o( K, L ).
letter( K, L ) :- x( K, L ).
atom( K, L ) :- letter( K, L ).
atom( K, M ) :- letter( K, L ), atom( L, M ).
```
(3.3)

Contiguity of edges is assured by using the same term (variable name) to denote every intermediate node: once at the end of an edge and once at the beginning of the next one.

Given the clauses that describe edges with terminal symbols, e.g.

```
o( 9, 10 ).
x( 10, 11 ).
```
(3.4)

we might now derive all the remaining relevant edges. Strictly speaking, they would be present only implicitly. For example, to confirm the presence of the edge

atom(9, 11)

we would issue the command

:- atom(9, 11).

from which the following computation might ensue:

```
        atom( 9, 11 ).
        letter( 9, L ), atom( L, 11 ).
        o( 9, L ), atom( L, 11 ).
(3.5)                              L ← 10
        atom( 10, 11 ).
        letter( 10, 11 ).
        x( 10, 11 ).
        success
```

The method of specifying the initial graph is rather awkward, even for this small example. Moreover, it requires that terminal symbols be only identifiers (nullary functors)—the restriction is unnatural but, fortunately, unnecessary. We shall now describe a slightly different and much handier notation.

Names of nodes need not be consecutive integers. On the contrary, it is much better to derive (unique) names from the original sequence of terminal symbols than to introduce another, completely independent nomenclature. We shall exploit the one–one correspondence between a node and the sequence of (contiguous) edges following it. As the name of a node we shall take the *list* of terminal symbols labelling the corresponding sequence. For example, the leftmost node of the graph (3.2) will be named

```
'('.a.','.'('.b.i.g.','.o.x.')'.')'.[]
```

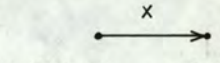
and the name of the rightmost one—corresponding to the empty sequence of nodes—will be

```
[]
```

With this notation, the (implicit) clause describing the atom ox becomes

```
atom( o.x.')'.')'.[], ')'.')'.[] ).
```

Notice how the underlying sequence of terminal symbols can be seen without resorting to separate clauses for o and x: it is simply the "difference" of the first and the second node names, o and x in our case. For a terminal symbol this difference is guaranteed to consist of the symbol itself, as, say, in



In other words, if an edge connects the nodes X, Y and is labelled with the terminal symbol T, then

```
X = T.Y
```

In order to allow arbitrary terms as terminal symbols, we can write, e.g.

```
terminal( o, K, L )
```

instead of o(K, L). Moreover, rather than writing

```
.....
terminal( o, o.x.')'.')'.[], x.')'.')'.[] ).
terminal( x, x.')'.')'.[], ')'.')'.[] ).
.....
```

we shall use the general-purpose, one-clause auxiliary procedure

```
terminal( T, T.Y, Y ).
```

However, now we need some other way of specifying the initial sequence of terminal symbols, which in the previous formulation could be read from the assertions (3.4). Before we explain this, we shall rewrite (3.3):

```
letter( K, L ) :- terminal( o, K, L ).
letter( K, L ) :- terminal( x, K, L ).
atom( K, L ) :- letter( K, L ).
atom( K, M ) :- letter( K, L ), atom( L, M ).
terminal( T, T.Y, Y ).
```

The computation analogous to that shown in (3.5) would now look as follows:

```
        atom( o.x.')'.')'.[], ')'.')'.[] ).
        letter( o.x.')'.')'.[], L ), atom( L, ')'.')'.[] ).
        terminal( o, o.x.')'.')'.[], L ), atom( L, ')'.')'.[] ).
                                        L ← x.')'.')'.[]
(3.6)   atom( x.')'.')'.[], ')'.')'.[] ).
        letter( x.')'.')'.[], ')'.')'.[] ).
        terminal( x, x.')'.')'.[], ')'.')'.[] ).
        success
```

All the necessary information about the initial graph was supplied by the first call. What is more, the graph itself is now implicit: we only get—and manipulate—the two sequences of terminal symbols.

We are now in a good position to restate each instance of the parsing problem in terms of Prolog. A grammar is given in the form of Prolog clauses, each clause corresponding to some structural relationship between a unit and its immediate components (in particular, to a BNF rule). For example,

```
items( K, N ) :-
    item( K, L ), terminal( ',', L, M ), items( M, N ).
```

A call on one of these clauses (or, to be more precise, on the procedure to which it belongs) fully specifies two lists of terminal symbols, the second being the tail of the first. As a matter of convention, the clause name will also be the name of a nonterminal symbol, i.e. it will tell us what structure we want to attribute to the underlying sequence of terminal symbols. For example, the call

```
:- items( b.i.g.',.o.x.')'.')'.[], ')'.')'.[] ).
```

can be interpreted as the question: In the graph determined by the parameters, can an edge labelled with *items* be validly drawn between the extreme nodes? Or briefly: Is *items* the valid structure of a given sequence of terminal symbols, big,ox in our case?

The answer to this question is YES if the call succeeds, and NO otherwise. Examples of unsuccessful attempts to parse are;

```
:- items( ',.o.x.')'.')'.[], ')'.')'.[] ).
    /items cannot begin with a comma/
:- atom( o.x.')'.')'.[], ')'.[] ).
    /atom cannot end with a bracket/
```

Procedural interpretation can be expressed in terms of the successive augmentation of the original graph. Every *successful* call implicitly adds an edge. Parsing succeeds if we can connect the extreme nodes with a single edge. This construction proceeds bottom-up: we can imagine an edge being added only after the successful *termination* of a corresponding call.

We shall illustrate this by a complete program for parsing lists.

(3.7)

```
list( K, M ) :- terminal( '(', K, L ), terminal( ')', L, M ).
list( K, N ) :-
    terminal( '(', K, L ), items( L, M ), terminal( ')', M, N ).
items( K, L ) :- item( K, L ).
items( K, N ) :-
    item( K, L ), terminal( ',', L, M ), items( M, N ).
item( K, L ) :- atom( K, L ).
item( K, L ) :- list( K, L ).
atom( K, L ) :- letter( K, L ).
atom( K, M ) :- letter( K, L ), atom( L, M ).
letter( K, L ) :- terminal( a, K, L ).
.....
letter( K, L ) :- terminal( z, K, L ).
terminal( T, T.Y, Y ).
```

The call on *list* in the command

```
:- list( '('.a.','.'('.b.i.g.','.o.x.')'.')'.[], [] ).
```

results in the implicit construction of the graph shown in Fig. 3.2. Notice the similarity of this graph to a conventional parse tree (Fig. 3.3).

The parameters of a call that initiates the parsing serve as an input and an output parameter. The former contains a given list of terminal symbols. Some initial segment of this list is supposed to constitute the unit under consideration. For example, in

```
atom( o.x.')'.')'.[], ')'.')'.[] )
```

we expect that some initial part of the list

```
o.x.')'.')'.[]
```

constitutes an atom. Should that be the case, the computation succeeds provided the second parameter matches the tail of the list which remains after "chopping off" the initial segment. For example, ')'.')'.[] remains after chopping o and x off the list o.x.')'.')'.[]. In most cases the second parameter is a variable, so that it actually behaves like an output parameter. As an example, the call

(3.8) atom(o.x.')'.')'.[], Tail)

instantiates Tail as ')'.')'.[]—compare this with (3.6).

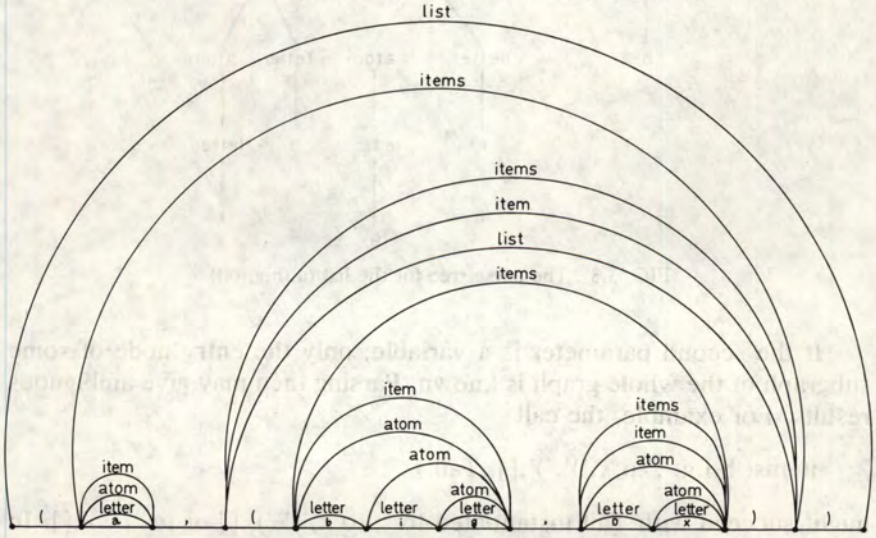


FIG. 3.2 The graph for the list (a,(big,ox)).

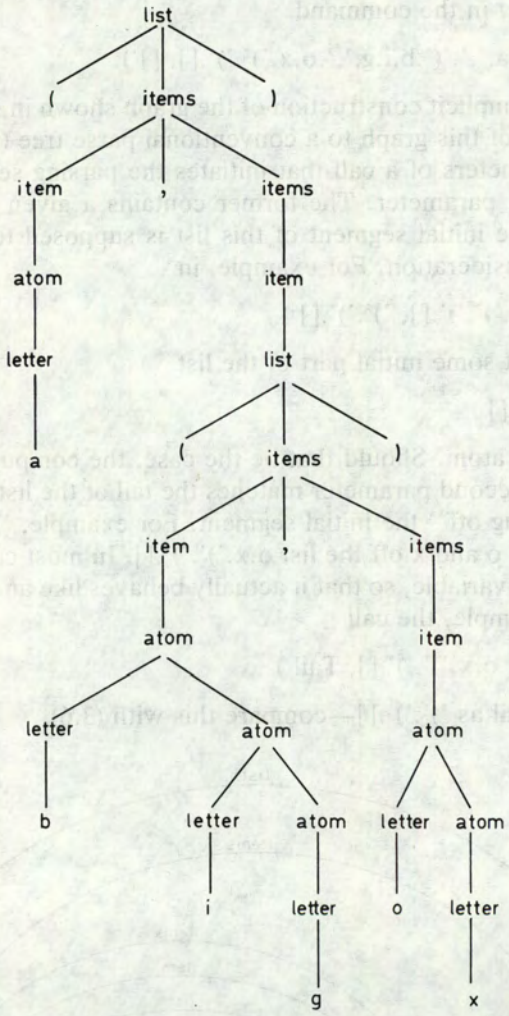


FIG. 3.3 The parse tree for the list (a,(big,ox)).

If the second parameter is a variable, only the entry node of some subgraph of the whole graph is known. Parsing then may give ambiguous results. For example, the call

items(b.i.g.','.o.x.')'.')'.[], Tail)

might succeed with Tail instantiated to ','.o.x.')'.')'.[] or to ')'.')'.[]. In general, the results depend on how the clauses of a parsing program are ordered. In the program above, the recursive clause for *items* would only

be activated because of forced failure coming after a successful parsing of big as *items*.

Recall now that the parameter of a Prolog procedure can, in principle, be bi-directional, the direction—input or output—depending on the form of the corresponding actual parameter. This also applies to calls that initiate parsing. If the first parameter is a variable, what we ask is whether there exists a sequence of terminal symbols that has a particular structure. For example, the call

```
list( AList, [] )
```

should instantiate AList to *any* valid list of terminal symbols; in other words, some list should be constructed, or synthesized. One example of such a list is the empty list.

However, the situation is not fully symmetric. For any given sequence of terminal symbols, a call on *list* either succeeds or fails, i.e. every sequence can be classified as a list or a non-list—can be syntactically analysed. Not so with synthesis. It is easy to see that the two calls

```
list( AList, [] ), fail
```

will act as a generator of one-element lists:

() (a) (b) ... (z) (aa) (ab) ... (az) (aaa) (aab) ...

Moreover, if we reorder the two clauses for *item*, the call on *item* with a variable first parameter would result in infinite recursion.

3.2. THE SIMPLEST FORM OF GRAMMAR RULES

The input and output parameters of the clauses that constitute a parsing program, such as (3.7), are the basis of yet another interpretation of those clauses: in terms of operations on sequences of terminal symbols. Take the clause

```
items( K, N ) :- item( K, L ), terminal( ',', L, M ), items( M, N ).
```

It can be read as follows: (an instance of) *items* can be "chopped off" (recognized at the beginning of) K, leaving N, if (an instance of) *item* can be chopped off K, leaving L, and then a comma can be chopped off L, leaving M, and finally (another instance of) *items* can be chopped off M, leaving N. Now the essence of all this is that *items* consist of an *item*, a comma, and *items*. The other information can be routinely added to this fundamental fact. All we need is four variables to stand for successive remainders of the initial sequence of terminal symbols.

In the notation we shall use henceforth, this routine information is suppressed. The notation resembles BNF productions. The lefthand side of a **Prolog grammar rule** names the construction, and the righthand side enumerates its constituents. For example:

atom → letter, atom.

The symbol → is rendered in Prolog as --> (it must be written without intervening blanks). There is a simple convention to distinguish nonterminal and terminal symbols: the latter are enclosed in square brackets, e.g.

items → item, [','], items.

Contiguous terminal symbols can be enclosed in a single pair of brackets. For example, the rule for empty lists can be written as

list → ['(', ')'].

If all terminal symbols are characters (one-character nullary functors), we can use string notation:

list → "()".

Such grammar rules are merely syntactic sugar for the underlying clauses. The translation is fairly straightforward, the gain in clarity significant. However, some Prolog implementations, especially on small computers, do not support grammar rule notation. Even then it seems worthwhile to write a preprocessor in Prolog (we shall describe such a preprocessor in Section 7.4.4).

The counterpart of a parsing program, written down as a collection of grammar rules, will be called a **metamorphosis grammar**[3], or grammar for short. Here is the grammar of lists, corresponding to the program (3.7).

```
list → [ '(', ')' ].
list → [ '(' ], items, [ ')' ].
items → item.
items → item, [ ',' ], items.
item → atom.
item → list.
atom → letter.
atom → letter, atom.
letter → [ a ].
.....
letter → [ z ].
```

[3] This is the name invented by Colmerauer (1975, 1978). The name "definite clause grammars" was later introduced by Pereira and Warren (1980) for metamorphosis grammars in normal form (as defined by Colmerauer).

The procedure *terminal* need not be explicitly given (it ought to be provided by the implementation).

This grammar deserves its name. It is best understood independently of the Prolog program it has been used to conceal. Every rule reflects the "consist of" relationship between a whole and its constituents, exactly as the original BNF grammar does. However, it should be remembered that the grammar is also a program in disguise, and is executable *immediately*, without any additional effort on the programmer's part!

Parsing can be initiated in two ways. First, we can simply call one of the underlying procedures, e.g.

```
:- list( '('.a.','.'('.b.i.g.','.o.x.')'.')'.[], [] ).
```

Second, we can use the built-in procedure *phrase* with two parameters: the nonterminal symbol and the sequence of terminal symbols (which is supposed to be an instance of the nonterminal). For example:

```
:- phrase( list, '('.a.','.'('.b.i.g.','.o.x.')'.')'.[] ).
```

It should be pointed out that the first way brings out the routine information we just managed to hide. On the other hand, the second way is less flexible, e.g. we cannot use *phrase* to perform calls such as (3.8).

3.3. PARAMETERS OF NON-TERMINAL SYMBOLS

Grammars of the kind described so far are of little practical use. We seldom parse anything just to accept or reject it. More often than not, we need to compute the representation of its structure or to transform it somehow, and we must do this while accepting the input. The representation of the structure will be built step by step, with the terminal symbols taken into account in succession.

We shall give an example. Suppose we want to build a parse tree—a Prolog term—for every valid sequence of terminal symbols that constitute a list; see Fig. 3.3. To this end, we shall give each of the procedures in (3.7) an additional parameter to hold the representation (of a structure) to be constructed upon exit from the procedure. We must not meddle with input and output parameters: their role remains the same as before. Here is the program.

```
list( list( '(', ')' ), K, M ) :-
    terminal( '(', K, L ), terminal( ')', L, M ).
```

```
list( list( '(', ITEMS, ')' ), K, N ) :-
    terminal( '(', K, L ), items( ITEMS, L, M ),
    terminal( ')', M, N ).
items( items( ITEM ), K, L ) :- item( ITEM, K, L ).
items( items( ITEM, ',', ITEMS ), K, N ) :-
    item( ITEM, K, L ), terminal( ',', L, M ),
    items( ITEMS, M, N ).
item( item( ATOM ), K, L ) :- atom( ATOM, K, L ).
item( item( LIST ), K, L ) :- list( LIST, K, L ).
atom( atom( LETTER ), K, L ) :- letter( LETTER, K, L ).
atom( atom( LETTER, ATOM ), K, M ) :-
    letter( LETTER, K, L ), atom( ATOM, L, M ).
letter( letter( a ), K, L ) :- terminal( a, K, L ).
.....
letter( letter( z ), K, L ) :- terminal( z, K, L ).
```

Again, we shall suppress the routine information, i.e. leave out the input and output parameters. The resulting grammar will be as follows:

```
list( list( '(', ')' ) ) → [ '(', ')' ].
list( list( '(', ITEMS, ')' ) ) →
    [ '(' ], items( ITEMS ), [ ')' ].
items( items( ITEM ) ) → item( ITEM ).
items( items( ITEM, ',', ITEMS ) ) →
    item( ITEM ), [ ',' ], items( ITEMS ).
item( item( ATOM ) ) → atom( ATOM ).
item( item( LIST ) ) → list( LIST ).
atom( atom( LETTER ) ) → letter( LETTER ).
atom( atom( LETTER, ATOM ) ) → letter( LETTER ),
    atom( ATOM ).
letter( letter( a ) ) → [ a ].
.....
letter( letter( z ) ) → [ z ].
```

To compute the parse tree of Fig. 3.3, call:

```
:- phrase( list( T ), '('.a.','.'('.b.i.g.','.o.x.')'.')'.[] ).
```

The conciseness and power of metamorphosis grammars can hardly be appreciated in this tiny example. We shall show a grammar that describes (and parses) sequences of statements of a simple programming language. The admissible statements are: assignment, if–then–else–fi,

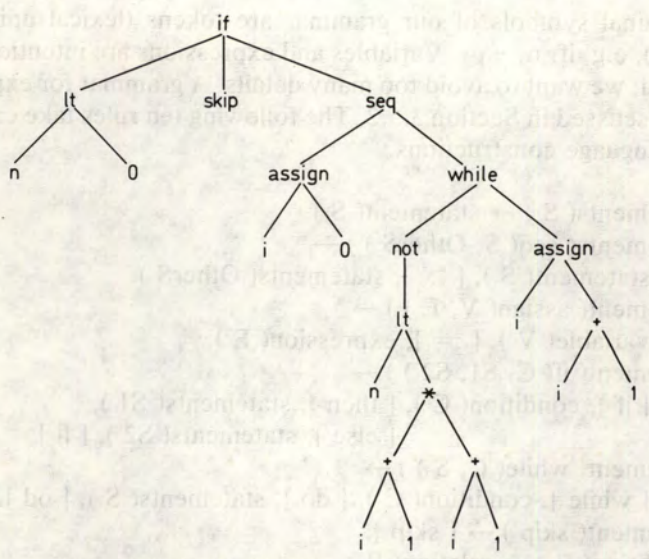


FIG. 3.4 An abstract syntax tree.

while–do–od, and skip. The sequencing operator is the semicolon. The condition is either an arithmetic relation (= or <) or a relation negated[4].

The intended meaning of a sequence of statements is the term that shows its structure. We shall not go into details; instead, we shall give an example which ought to explain the idea. Given the (one-element) sequence of statements:

```
if n < 0 then skip else
    i := 0 ;
    while not n < (i + 1)*(i + 1) do
        i := i + 1
    od
fi
```

we should obtain the abstract syntax tree (a Prolog term):

```
        if( lt( n, 0 ), skip, seq( assign( i, 0 ),
(3.9)       while( not( lt( n, '*'( '+'( i, 1 ), '+'( i, 1 ) ) ) ),
                assign( i, '+'( i, 1 ) ) ) ) )
```

The same tree is shown in Fig. 3.4.

[4] Both parts of this example, here and in Section 3.4.1, are modelled on the illustration in Colmerauer's original paper (1975).

Terminal symbols of our grammar are tokens (lexical units of the language), e.g. if, n, +, (. Variables and expressions are intentionally left undefined: we want to avoid too many details. A grammar for expressions will be discussed in Section 3.5.2. The following ten rules take care of the rest of language constructions.

```
statements( S ) → statement( S ).
statements( seq( S, OtherS ) ) →
    statement( S ), [ ';' ], statements( OtherS ).
statement( assign( V, E ) ) →
    variable( V ), [ := ], expression( E ).
statement( if( C, S1, S2 ) ) →
    [ if ], condition( C ), [ then ], statements( S1 ),
                            [ else ], statements( S2 ), [ fi ].
statement( while( C, S ) ) →
    [ while ], condition( C ), [ do ], statements( S ), [ od ].
statement( skip ) → [ skip ].
condition( R ) → relation( R ).
condition( not( R ) ) → [ 'not' ], relation( R ).
relation( eq( E1, E2 ) ) →
    expression( E1 ), [ '=' ], expression( E2 ).
relation( lt( E1, E2 ) ) →
    expression( E1 ), [ '<' ], expression( E2 ).
```

This grammar would probably be activated by calls such as

```
... read_a_list_of_tokens( LisT ),
    phrase( statements( Structure ), LisT ) ...
```

which analyse LisT and instantiate Structure appropriately, or fail if LisT is not a valid sequence of statements. Another possibility (not always practical, though) is to build—synthesize, if you prefer—a list of Tokens starting from a given structure:

```
... take_a_structure( S ), phrase( statements( S ), Tokens ) ...
```

Here, Tokens will be instantiated if only S is a proper structure. The grammar establishes one–one correspondence between structures and lists of tokens, and provides transformation both ways.

A more realistic example of synthesis based on a metamorphosis grammar will be given in the next section. Here we only observe that in both cases (analysis and synthesis) similar computations ensue. They differ because, on analysis, the sequence of terminal symbols "controls"

the computation (i.e. determines the choice of rules) whereas, on synthesis, it is "controlled" by the initial non-terminal symbol's parameter.

3.4. EXTENSIONS

3.4.1. Conditions

Grammar rules described so far correspond to clauses in which every call manipulates the sequence of terminal symbols, i.e. every call has an input and an output parameter. Other calls could be inserted in between without affecting the transfer of terminal symbols. The question is: Would it be useful, and how could it be interpreted?

As a simple possibility, consider the cut in the first clause of *list*:

```
list( list( '(', ')' ), K, M ) :-
    terminal( '(', K, L ), terminal( ')', L, M ), !.
```

The cut turns the computation based on the *list* procedure into a "deterministic" process: it handles either the empty list or non-empty lists. It does not matter when we want to recognize a list. However, it is now impossible to generate lists. The command

```
:- list( L, T, [] ), write( L ), write( T ), nL, fail.
```

will only write one instance of L and T, namely

```
list( '(', ')' )    and    '('.')'.[]
```

The gain from the cut is small in this case, anyway. Cuts would be of much greater use, say, in the program that parses statements (see the previous section), where long and deep computations may occur.

Another example: suppose we want to change the program for parsing lists so that for an atom it produces a Prolog atom instead of a parse tree, e.g. returns

```
list( '(', items( item( big ), ',', items( item( ox ) ) ), ')' )
```

for the list (big,ox). One way to do so is to make the procedure for atoms return a Prolog list of letters, and apply the built-in procedure *pname* (see Section 5.10) to this list

```
item( item( ATOM ), K, L ) :-
    atom( LETTERS, K, L ), pname( ATOM, LETTERS ).
item( item( LIST ), K, L ) :- list( LIST, K, L ).
atom( LETTER.[], K, L ) :- letter( LETTER, K, L ).
```

```
atom( LETTER.LETTERS, K, M ) :-
    letter( LETTER, K, L ), atom( LETTERS, L, M ).
letter( a, K, L ) :- terminal( a, K, L ).
.....
letter( z, K, L ) :- terminal( z, K, L ).
```

One final example: in the program above we shall replace the 26 clauses that define letters by a single clause:

```
letter( LETTER, K, L ) :-
    terminal( LETTER, K, L ), isletter( LETTER ).
```

with *isletter* defined, say, as

```
isletter( LETT ) :- a @=< LETT, LETT @=< z.
```

This new clause can be used as follows:

```
letter( Lett, x.')'.')'.[], Tail ).
terminal( Lett, x.')'.')'.[], Tail ), isletter( Lett ).
  Lett ← x,      Tail ← ')'.')'.[]
isletter( x ).
etc.
```

The variable in the call on *terminal* matches *every* terminal symbol. If the terminal symbol is *not* a letter, a call on *isletter* will fail and a letter will not be recognized. We call such terminal symbols **variable terminals**: the first (still unprocessed) symbol is selected and is then either accepted or rejected, e.g. according to the result of a test such as *isletter*.

Extra calls that do not comprise input and output parameters have been known as *conditions*, but the name is slightly misleading. Only in the last example *isletter*(*Lett*) can be interpreted as a condition: the clause will only be applied if *isletter* succeeds. The call on *pname* in the second example is rather an action performed on the parameters of non-terminal symbols. Finally, the cut can be reasonably interpreted exactly as in any other clause, as pragmatic information on the future use of the clause.

Conditions in metamorphosis grammars are enclosed in curly brackets, so that they will not be confused with terminal and non-terminal symbols. Examples:

```
list( list( '(', ')' ) ) → [ '(', ')' ], {!}.
item( item( ATOM ) ) → atom( LETTERS ),
    { pname( ATOM, LETTERS ) }.
letter( LETTER ) → [ LETTER ], { isletter( LETTER ) }.
```

As an exception, the cut need not be placed within curly brackets, e.g.

```
list( list( '(', ')' ) ) → [ '(', ')' ], !.
```

Contiguous conditions can be combined in a single pair of brackets, and in general a condition can also contain alternatives conjoined by semicolons, e.g.

```
alphanum( Char ) → [ Char ], { isletter( Char ) ; isdigit( Char ) }.
```

We shall now present a small fragment of a metamorphosis grammar, meant primarily for synthesis (but applicable both ways, although not without reservations). We want to take a structure computed by the grammar for statements (see the previous section) and produce its translation into a machine-oriented symbolic language. We shall only give a hint of the target language by showing schematic translations of while(C, S) and if(C, S1, S2).

Let $\overline{C}$ and $\overline{S}$ be the translations of C and S. The evaluation of $\overline{C}$ sets a flag used implicitly by a conditional jump instruction. Let $\ell 1$, $\ell 2$ be unique labels. The translation of while(C, S) will be

```
label( ℓ1 )
not(C)
jumpiftrue( ℓ2 )
S
jump( ℓ1 )
label( ℓ2 )
```

The translation of if(C, S1, S2) will be

```
C
jumpiftrue( ℓ1 )
S2
jump( ℓ2 )
label( ℓ1 )
S1
label( ℓ2 )
```

The "code generator" can be written as a grammar of the target language. By way of explanation, we shall show three of the rules that belong to the uppermost level of the definition:

```
code( seq( S, OtherS ) ) → code( S ), code( OtherS ).
code( while( C, S ) ) →
    { newlabel( L1 ) }, [ label( L1 ) ], codecond( not( C ) ),
    { newlabel( L2 ) }, [ jumpiftrue( L2 ) ], code( S ),
    [ jump( L1 ), label( L2 ) ].
code( skip ) → [].
```

The action *newlabel* can generate a new, unique label. The definition of *codecond* will be given below. The third rule illustrates a new feature of

grammar rules. If the righthand side contains no terminal and non-terminal symbols, nothing will be produced during synthesis and nothing will be "chopped off" during analysis. The underlying clause is

```
code( skip, K, K ).
```

Try to trace the execution of

```
:- code( seq( skip, skip ), Translation, [] ).
```

Assuming that *coderel* defines the grammar of codes for relations eq and lt, the definition of *codecond* can be as follows:

```
codecond( not( not( C ) ) ) → codecond( C ).
codecond( not( Rel ) ) → coderel( Rel ), [ revert(_) ].
codecond( Rel ) → coderel( Rel ).
```

where "revert" is an instruction of the target language that resets the "condition flag".

The example would be completed after specifying the translation of expressions and of assignments, in particular the handling of variables.

The code generator together with the grammar of statements might constitute the core of a simple compiler. Its overall structure might be:

```
compile :- read_tokens( Token_list ),
           parse( Token_list, Syntax_tree ),
           generate_code( Syntax_tree, Object_code ),
           write_code( Object_code ).
```

with *parse* and *generate_code* defined as

```
parse( T, S ) :- phrase( statements( S ), T ).
generate_code( S, O ) :- phrase( code( S ), O ).
```

The procedure read_tokens, reading the source program in and performing lexical analysis, might also be (partly) written as a metamorphosis grammar—see Colmerauer (1975, 1978).

3.4.2. Context

Another feature of grammar rules in Prolog is a mechanism for modifying the sequence of terminal symbols during the computation. In general, this would require explicit manipulations on input and output parameters, but such general mechanisms seem only necessary in natural language processing (an important application of Prolog). A very restricted mechanism, so-called context grammar rules, is quite sufficient, though, in most of the other applications.

In a context grammar rule, the lefthand side is supplemented by a so-called **context**[5]: terminal symbols, preceding the arrow →. For example:

```
otherst( S, S ), [ Delim ] → [ Delim ], { stsdelim( Delim ) }.
do, [ 'not' ] → dont.
```

The output parameter in the head of an underlying clause is appended to the context. As clauses, the above rules are:

```
otherst( S, S, K, Delim.L ) :-
    terminal( Delim, K, L ), stsdelim( Delim ).
do( K, 'not'.L ) :- dont( K, L ).
```

The first rule can be interpreted without resorting to the corresponding clause; we shall give the interpretation below. The second rule, however, can only be explained in terms of manipulations on sequences of terminal symbols: a *new* terminal symbol appears after recognizing an instance of *dont*, and only then is an instance of *do* recognized as well. We shall elaborate on this example a little, too.

First we come back to the grammar for statements. In its present shape it performs rather poorly on incorrect inputs. It fails without giving any message or diagnostics. We shall try to improve the definition of *statements*, leaving the other rules as an exercise. We observe that a statement (other than the last) may be delimited by a semicolon (it indicates that there are other statements in this sequence), by **else**, **fi**, or **od**. Other delimiters are erroneous. In case of errors, no meaningful structure may be found for the whole sequence of statements, but we elect to continue the analysis, after skipping a portion of input up to the nearest semicolon. Here are some rules of a grammar that implements these ideas.

```
statements( Sts ) → statement( St ), otherst( St, Sts ).
otherst( St1, seq( St1, Sts ) ) →
    [ ';' ], statement( St2 ), otherst( St2, Sts ).
otherst( St, St ), [ Delim ] →
    [ Delim ], { stsdelim( Delim ) }.
otherst( _, _ ) → [ T ], erroneous( T ).
otherst( St, St ) → [].  % this for the last statement
erroneous( T ) → { write( bad( T ) ), nl }, skipped.
skipped, [ ';' ] → [ ';' ].
```

[5] Readers familiar with context-sensitive grammars will notice that neither rule is a proper context-sensitive rule. Even if we disregard parameters and conditions, the rules will only belong to Chomskian type 0.

```
skipped → [ _ ], skipped.
skipped → []. % if we are skipping the last statement
stsdelim( else ).    stsdelim( fi ).    stsdelim( od ).
```

The context rule can be interpreted in the following manner: "the remainder of a sequence of statements is empty if we have encountered a proper delimiter; this delimiter is retained". Notice that we have actually effected one-item lookahead on a list of terminal symbols. In general, we can have lookahead for any *fixed* number of terminal symbols, for example

```
p, [ T1, T2 ] → [ T1, T2 ], { test( T1, T2 ) }.
```

This translates into

```
p( K, T1.T2.M ) :-
    terminal( T1, K, L ), terminal( T2, L, M ), test( T1, T2 ).
```

We can use *p* to make the *test*; e.g. in

```
a → p, b, c.
```

p consumes no input, so that the rule is structurally equivalent to

```
a → b, c.
```

but it will only be applied if two leftmost terminal symbols of the current sequence pass the test.

The second example is a very simplified little grammar that recognizes auxiliary "do not", "don't", does not", "doesn't". This particular problem can easily be solved differently; the way we have chosen is intended as an illustration of context grammar rules:

```
aux → do, [ 'not' ].
do, [ 'not' ] → dont.
do → [ do ].
do → [ does ].
dont → [ 'don''t' ].      %i.e. don't
dont → [ 'doesn''t' ].    %i.e. doesn't
```

The following computation should explain how this grammar is used:

```
aux( 'doesn''t'.like.it.[], Tail ).
do( 'doesn''t'.like.it.[], Tl ), terminal( 'not', Tl, Tail ).
  Tl ← 'not'.L
dont( 'doesn''t'.like.it.[], L ),
  terminal( 'not', 'not'.L, Tail ).
```

```
terminal( 'doesn''t', 'doesn''t'.like.it.[], L ),
    terminal( 'not', 'not'.L, Tail ).
    L ← like.it.[]
terminal( 'not', 'not'.like.it.[], Tail ).
    Tail ← like.it.[]
success
```

Our last example is a small grammar that discards leading zeroes from an integer represented as a list of digits:

```
zeroes, [ D ] → [ 0 ], zeroes, [ D ], { digit( D ) }.
zeroes → [].
```

You may wish to trace the execution of the directives

```
:- zeroes( 0.3.[], Tail ).
:- zeroes( 0.0.[], Tail ).
```

3.4.3. Alternatives

Two or more grammar rules with the same lefthand side (including context and parameters of the non-terminal symbol) can be combined into a single rule with the common lefthand side and with the righthand side taking the form of **alternatives**—a sequence of original righthand sides separated by semicolons. For example:

```
list → [ '(', ')' ] ; [ '(' ], items, [ ')' ].
items → item ; item, [ ',' ], items.
item → atom ; list.
atom → letter ; letter, atom.
letter → [ L ], { isletter( L ) }.
```

Notice how—at last—we managed to come back rather closely to the original BNF grammar (3.1).

The translation of a rule with an alternative into an underlying clause is straightforward. One example should be sufficient:

```
items( K, N ) :- item( K, N ) ;
                 item( K, L ), terminal( ',', L, M ), items( M, N ).
```

The notation with alternatives is, strictly speaking, a "convenience" rather than a real extension, and—like alternatives in ordinary clauses (see Section 1.3.7)—it can sometimes adversely affect the grammar's readability.

3.4.4. Syntax of Grammar Rules: Summary

We shall now give a metamorphosis grammar that describes full syntax of grammar rules supported by Prolog-10. The principles of mapping rules onto underlying clauses have been discussed at length in the previous sections, so we choose not to overburden the grammar with parameters that would take care of the translation. However, we encourage you to try and augment the grammar along these lines. A hint: most of the nonterminal symbols should be given three parameters, two variables (to construct an input and output parameter) and a term (to hold the—partial—translation). For example:

```
grammar_rule( ( Tr_of_left :- Tr_of_right ), In_var, Out_var )
    → lefthand_side( Tr_of_left, In_var, Out_var ), [ '→' ],
      righthand_side( Tr_of_right, In_var, Out_var ), [ '.' ].
rule_items( ( Tr_of_item, Tr_of_items ), Curr_in_var, Out_var )
    → rule_item( Tr_of_item, Curr_in_var, Mid_var ), [ ',' ],
      rule_items( Tr_of_items, Mid_var, Out_var ).
```

In the actual translation we might eliminate the calls on the procedure *terminal*. Since *terminal(T, K, L)* means that K = T.L, we can substitute in advance T.L for K elsewhere in the clause. For example, in the clause

```
list( K, N ) :-
    terminal( '(', K, L ), items( L, M ), terminal( ')', M, N ).
```

we have K = '('.L and M = ')'.N, and after replacing K and M we obtain

```
list( '('.L, N ) :- items( L, ')'.N ).
```

This is, in fact, what is done in many implementations (see, e.g., Section 7.4.9). As we have executed both calls on *terminal* beforehand, every computation started by a call on *list* will be at least two steps shorter. Here are some other examples of such an improved translation of grammar rules:

```
letter( Lett, Lett.L, L ) :- isletter( Lett ).
p( T1.T2.M, T1.T2.M ) :- test( T1, T2 ).
zeroes( 0.L, D.N ) :- zeroes( L, D.N ), digit( D ).
```

We shall now present the grammar *without* parameters (it is, really, equivalent to a BNF definition).

```
grammar_rule → lefthand_side, [ '→' ],
               righthand_side, [ '.' ].
lefthand_side → nonterminal, context.
context → terminals ; [].
```

```
righthand_side → alternatives.
alternatives → alternative ;
               alternative, [ ';' ], alternatives.
alternative → [ [] ] ; rule_items.
rule_items → rule_item ; rule_item, [ ',' ], rule_items.
rule_item → nonterminal ; terminals ; condition ; [ ! ] ;
            [ '(' ] , alternatives , [ ')' ].
nonterminal → name ;
              name, [ '(' ], list_of_terms, [ ')' ].
terminals → [ '[' ], list_of_terms, [ ']' ] ; string.
condition → [ '{' ], procedure_body, [ '}' ].
list_of_terms → term ; term, [ ',' ], list_of_terms.
```

Definitions of name, term, string and procedure_body are left as an exercise.

It should be noted that the original appearance of grammar rules in the Marseilles interpreter of Prolog I (Roussel 1975) was slightly different. In particular, no alternatives were allowed, and terminal symbols and conditions could not be combined. Just to give the flavour of it, we shall rewrite in Marseilles syntax some of the grammar rules for statements (Section 3.4.2).

```
:STATEMENTS( *STS ) == :STATEMENT( *ST )
    :OTHERST( *ST, *STS ).
:OTHERST( *ST1, SEQ( *ST1, *STS ) ) ==
    #; :STATEMENT( *ST2 ) :OTHERST( *ST2, *STS ).
:OTHERST( *ST, *ST ) #*DELIM ==
    #*DELIM -STSDELIM( *DELIM ).
:OTHERST( *DUMMY1, *DUMMY2 ) ==
    #*T :ERRONEOUS( *T ).
:OTHERST( *ST, *ST ) == .
    *THIS FOR THE LAST STATEMENT.
```

3.5. PROGRAMMING HINTS

3.5.1. Efficiency Considerations

Metamorphosis grammars correspond to Prolog programs which implement a very general parsing strategy: nondeterministic top-down parsing with backtracking (Aho and Ullman 1977; Gries 1971). The potential cost of this strategy is exponential. This is the disadvantage of the gener-

ality and ease of programming with metamorphosis grammars. Well-known parsing algorithms for restricted classes of context-free grammars can be quite conveniently programmed in Prolog without metamorphosis grammars. See for example the operator precedence parser described in Section 7.4.3 and Appendix A.3. However, this requires explicit handling of the parsing stack, attributes etc., while metamorphosis grammars by themselves are as powerful as attribute grammars (Knuth 1968) or two-level grammars (van Wijngaarden 1976)—see the discussion in (Pereira and Warren 1980). Parameters and conditions/actions make it possible to construct an intuitively appealing, concise and readable metamorphosis grammar of any existing programming language (and of reasonable subsets of natural languages), capturing semantics as well as syntax—see e.g. (Moss 1979). At the same time, such a grammar can usually be used as a translator of this language, without additional effort on the part of the programmer, but there is often a certain price to be paid in efficiency.

One source of inefficiency is repetition. Consider two rules from the grammar for statements (Section 3.3):

```
relation( eq( E1, E2 ) ) →
    expression( E1 ), [ '=' ], expression( E2 ).
relation( lt( E1, E2 ) ) →
    expression( E1 ), [ '<' ], expression( E2 ).
```

If a given relation is not an equality, we recognize this state of affairs only after parsing the first expression and failing to find an equals sign. We abandon the rule and choose the next but then we must once more parse the first expression (which may be quite large). The problem remains if we change the order of the rules.

To avoid this inefficiency, we may apply **factorization**—the technique already used in Section 3.4.2:

```
relation( R ) → expression( E1 ), op_and_expr( E1, R ).
op_and_expr( E1, eq( E1, E2 ) ) → [ '=' ], expression( E2 ).
op_and_expr( E1, lt( E1, E2 ) ) → [ '<' ], expression( E2 ).
```

Another solution is to combine the original rules into a single rule by replacing the terminal symbols with a variable terminal, and adding a suitable condition:

```
relation( R ) → expression( E1 ), [ Op ],
                { makestruct( Op, E1, E2, R ) },
                expression( E2 ).
makestruct( '=', E1, E2, eq( E1, E2 ) ).
makestruct( '<', E1, E2, lt( E1, E2 ) ).
```

Notice the position of the condition: if we placed it at the end of the rule, we would run the risk of discovering an improper instance of Op only after parsing the whole input, say,

```
( A + b/2 )*c   blah_blah   2*( n - ( x + y )/4 )
```

In its present position the condition fails as soon as it sees an invalid operator.

Both improvements of the original grammar eliminate possible repetitions. Both, though, seem to decrease the readability and elegance of the original solution, and we recommend that they be applied (if at all necessary) only in the late stages of program debugging.

3.5.2. Elimination of Left Recursion

We shall now discuss a problem which frequently arises with inexpert use of metamorphosis grammars. As an example, we shall consider the task of writing a workable grammar of simple arithmetic expressions (see Section 3.3). Here is the definition in BNF (for simplicity, we limit ourselves to two operators only):

```
< expression > ::= < add_expr > |
                   < expression > + < add_expr > |
                   < expression > - < add_expr >
< add_expr > ::= < constant >
```

We now give an obvious transcription of this definition into a metamorphosis grammar. Parameters are used to build the structure of a given expression—see (3.9).

```
expression( E ) → add_expr( E ).
expression( E1 + E2 ) →
    expression( E1 ), [ '+' ], add_expr( E2 ).
expression( E1 - E2 ) →
    expression( E1 ), [ '-' ], add_expr( E2 ).
```

The definition of *add_expr* will be left out (it can be simply an integer constant).

Unfortunately, this grammar—as a program—is not only inefficient but also incorrect. It goes into infinite (left) recursion whenever we give it an expression that contains a minus. Try to analyse the expression 2 - 3 + 5 (represented by 2.'-'.3.'+'.5.[]).

At first sight, it seems we can improve the situation by applying one of the techniques shown in the previous section. For example, the second technique gives the following rules:

```
expression( E ) → add_expr( E ).
expression( E ) → expression( E1 ), [ Op ],
                  { makesum( Op, E1, E2, E ) },
                  add_expr( E2 ).
makesum( '+', E1, E2, E1 + E2 ).
makesum( '-', E1, E2, E1 - E2 ).
```

Now correct expressions will be parsed successfully, although an expression composed of n add-expressions will require $n - 1$ backtracks before reaching the solution. But the grammar will still fall into infinite recursion on any incorrect input (you may wish to check this on 2.+.[]). This means that it is of no practical value. As in all top-down parsing methods, we must eliminate left recursion to avoid trouble.

Suppose we reverse nonterminal symbols in the recursive rules in (3.10):

```
expression( E1 + E2 ) → add_expr( E1 ), [ '+' ], expression( E2 ).
expression( E1 - E2 ) → add_expr( E1 ), [ '-' ], expression( E2 ).
```

Now incorrect input causes the grammar to fail (without any error message, but this can be fixed). However, this grammar interprets operators as right-associative. The instantiation of its parameter for the expression 2 − 3 + 5 will be −(2, +(3, 5)) rather than +(−(2, 3), 5). Here is a possible solution to this new problem:

```
expression( E ) → add_expr( E1 ), rest_of_expression( E1, E ).
rest_of_expression( E1, E ) →
    [ '+' ], add_expr( E2 ), rest_of_expression( E1 + E2, E ).
rest_of_expression( E1, E ) →
    [ '-' ], add_expr( E2 ), rest_of_expression( E1 - E2, E ).
rest_of_expression( E1, E1 ) → [].
```

When we parse an expression, the parameter is initially uninstantiated. It is passed unchanged and instantiated after reaching the end of the expression. (In the terminology of attribute grammars this is a synthesized attribute.) The final structure is accumulated step by step. For example, during the parsing of the expression 2 − 3 + 4 − 5, *rest_of_expression* will be activated four times, with 2, 2 − 3, (2 − 3) + 4 and ((2 − 3) + 4) − 5 as the first parameter. (This parameter is an inherited attribute.) Eventually the third rule will be chosen and E instantiated to ((2 − 3) + 4) − 5.

We shall now present a grammar for expressions, complete with error handling, that fits the grammar for statements (see Sections 3.3 and 3.4.2). The definition of *erroneous* was given in Section 3.4.2.

```
expression( E ) → add_expr( E1 ), rest_of_expression( E1, E ).
rest_of_expression( E1, E ) →
    [ '+' ], add_expr( E2 ), rest_of_expression( E1 + E2, E ).
rest_of_expression( E1, E ) →
    [ '−' ], add_expr( E2 ), rest_of_expression( E1 − E2, E ).
rest_of_expression( E1, E1 ), [ Termin ] →
    [ Termin ], { expr_termin( Termin ) }.
rest_of_expression( _, _ ) → [ T ], erroneous( T ).
rest_of_expression( E1, E1 ) → [].
expr_termin( then ).          expr_termin( else ).
expr_termin( do ).            expr_termin( od ).
expr_termin( ';' ).           expr_termin( fi ).
add_expr( E ) → mult_expr( E1 ), rest_of_add_expr( E1, E ).
rest_of_add_expr( E1, E ) →
    [ '*' ], mult_expr( E2 ), rest_of_add_expr( E1*E2, E ).
rest_of_add_expr( E1, E ) →
    [ '/' ], mult_expr( E2 ), rest_of_add_expr( E1/E2, E ).
rest_of_add_expr( E1, E1 ), [ Termin ] →
    [ Termin ], { add_expr_termin( Termin ) }.
rest_of_add_expr( _, _ ) → [ T ], erroneous( T ).
rest_of_add_expr( E1, E1 ) → [].
add_expr_termin( Termin ) :- expr_termin( Termin ).
add_expr_termin( '+' ).
add_expr_termin( '−' ).
mult_expr( E ) → variable( E ).
mult_expr( E ) → constant( E ).
mult_expr( E ) → [ '(' ], expression( E ), [ ')' ].
```

To make the grammar really complete, we should also define variables and constants. We choose not to do it, because variables require symbol table handling—we shall discuss it in Section 4.2.2.

The techniques described above are only necessary if we want to perform analysis with a metamorphosis grammar. Even more: the transformed grammar is not good for synthesis, i.e. for constructing the sequence of terminal symbols given a (correct!) structure. Specifically, for synthesizing expressions, the only reasonable solution would be the original grammar (3.10).