SYNTACTIC DEFINITION MECHANISMS

Teun van Gils

PHILIPS-ELECTROLOGICA N.V.

APELDOORN

ABSTRACT.

The structure of a good syntax is derived from the requirement to understand and therefore to analyse a language easily. Every language possesses an abstract, representation-independent syntax. From this the operator precedence method and other complementary or alternative methods of analysis are deduced. It is possible to construct syntactic definition mechanisms by means of which for any abstract syntax different concrete languages can be defined, which are concise, surveyable, unambiguous and easily analysable. Incorporated in a programming language these syntactic definition mechanisms make that language syntactic extensible.

INTRODUCTION.

At the present day the syntax definition mechanism that is used to define almost every programming language, is Backus Naur Form. It is a simple mechanism but it appears to be difficult to develop an easy syntax directed analysis for it. Many methods have been proposed to improve the efficiency of the analyser, but they remain unsatisfactory unless a number of restrictions is accepted. As a result of these disadvantages, most commercial compilers are written by hand. If in an extensible language the programmer is allowed to declare new BNF rules, an efficient analysis may not be expected. This decreases the value of such an extensible system. However, a language is a means of communication so its sentences should be understood by every reader as well by a human reader as by a computer. The ease of understanding and therefore, the ease of the analysis belongs to the nature of the language itself. The syntax definition mechanism must guarantee the ease of analysis. BNF does not come up to this requirement and therefore, it should be replaced by another mechanism. An examination of the nature of programming languages may provide a top-down deduction of the nature of a good syntax.

ABSTRACT SYNTAX.

A program is only a piece of information, however, a very complex one, Algol 68 describes how complex-information can be built up. A value of a mode of information can be constructed by joining values of several different modes of information, and is called a structure. Joint values of the same kind of information are called array or row. When the value itself determines its mode, it is called a union. It appears to be rather easy to define the mode of an Algol 60 program:

mode program = statement

mode block = struct ([ ] decl, [ ] statement)

mode statement = union (block, assignment, conditional stat, for stat, ...)

mode conditional stat = struct (bool, statement, statement) etc.

A mode describes the kind of its values independent of their representation e.g. a value of the mode integer may have a decimal representation on paper, a punch card representation or a binary representation in a computer memory. Because of this abstract nature delimiters like the semicolon, begin, end, if then else and := are not present in the mode declaration of an Algol 60 program. Except for this, a close relationship exists between a mode and a syntactic unit: A mode can be considered as the set of all its values and a syntactic unit as the set of all its terminal strings. The union operator for modes has its translation in the alternatives of a BNF rule. The fields of a structure correspond with the components of one alternative of a BNF rule. The row operator for modes does not have a direct equivalent in BNF but is implemented as a recursive structure. However, sequences or chains exist in some extended BNF formalisms. The construction union (int, int, real) is forbidden in Algol 68 but it may occur that different cases of a union in an abstract syntax have the same mode accidentally. Then it is necessary to be able to distinguish between the different cases by means of identifiers. An example:

```
mode aexpr = union (arithmvariable a, unsigned number u,
     struct (aexpr a, aexpr b) power, struct (aexpr a, aexpr b) multipli-
                                                                 cation,
     struct (aexpr a, aexpr b) addition, etc .....)
```

A value of a complex mode constructed with the operators of struct, row, and union has the form of a tree, in which the fields of a structure or the elements of a row are parallel branches. With explicit information about the case of the union each time, the tree becomes a labelled tree.

CONCRETE SYNTAX.

A tree value of an abstract syntax may be represented as a tree on a two dimensional sheet of paper but it is handier to have a textual representation. A text means a linear ordered set of symbols. Defining the textual representation of a language is the task of a concrete syntax. The difference between the abstract and the concrete syntax is important to a language designer as because of this difference he does not have to worry about the representation when he is thinking about the essential, abstract structure of his language. After having fixed the structure of his language, he is free to pay attention to a good readibility of the representation. The most natural way to represent a structure by a string is placing the representations of its fields one after another and in the same way the representations of the elements of a row can be placed one after another. This proces is very easy. However, the other way around is more difficult. If one wants to reconstruct the tree value from its textual representation, it is necessary to know where the boundaries of the representations of the subtrees are and further the union information is needed. Information about the boundaries means that between two symbols - or after some reduction: between two syntactic units - it is known whether the first syntactic unit must be reduced first or the two syntactic units must be reduced at the same time or the second syntactic unit must be reduced before the first. These three possibilities of

precedence relation are generally called: greater than, equal, and less than. As the information about the boundaries of the subtrees, and therefore, the knowledge of the precedence relation is of vital importance to an efficient analysis, we must require that it can be deduced from its direct environment. This deduction should not be too difficult and so it seems desirable that it depends on two syntactic units at most. Such dependence can be described as a matrix. Here the first syntactic unit indicates the row, the second one indicates the column and each matrix element is the value of the corresponding precedence relation if the relation exists. However, remembering such a matrix is still too difficult. To reduce the quantity of information that must be remembered it is desirable that the matrix can be deduced from two functions f and g which map the syntactic units into integers. The value of the matrix element of the row x and the column y must, if it exists, be equal to the relation between $g(x)$ and $f(y)$. The actual construction of concrete syntax will show more nuances.

THE FUNCTION OF DELIMITERS.

The two problems of concrete syntax, namely the indication of the boundaries and of the union information, may be solved by placing extra symbols before, between or after the representations of the fields of a structure. The first problem can be solved by surrounding each structure by brackets. The second problem can be solved by preceding the first bracket by a symbol representing the union information. Any abstract syntax can be represented by this phrase marker form, (whose precedence relations can indeed be described by two functions and) which can be analysed very easily. Therefore, we may require that each concrete syntax has at least the same degree of easiness of analysis. The infix notation of dyadic operations demands less symbols and is more readable, as a good readability does not only imply an easy bottom up analysis but also implies that the bottom up analysis leads to a good top down survey. A good top down survey implies that the extent of large operands can be found easily. The parenthesized, the phrase marker, the prefix and the postfix notation need a counter to calculate the extent. The infix notation does not need such a counter and is therefore a more readable notation. Since the operator symbol does not only provide the union information but also the precedence information, the precedence functions of the operator symbols should be compared directly without looking at the operands. So the set of syntactic units must be divided into two subsets: the units that have precedence information and the other ones. Herewith we have deduced the operator precedence method. In order to avoid ambiguities in BNF the syntactic unit aexpr must be divided artificially into term, factor etc. which makes the language more difficult to learn then a language definition containing priority declarations. Moreover, the BNF formulation requires a transformation into priorities anyhow, either by men or by computers to obtain an easy analysis.

PRIORITY DECLARATIONS.

The examples show what the power of an extensible language could be. An operator may have only one left and one right priority; defined e.g.:

priority p = (2, 13).
However usually the left and right priority are the same:
priority += 6 means: priority += (6, 6).
Different from what is stated above: when an operator p1 is succeeded by p2 and when they have equal priorities, they may not be reduced together, but the left one must be reduced first. So the rule is: association to the left. If association to the right is wanted, the following declaration can be used:
priority := = 4 right means: priority := = (4, 3).
In Algol 68 the complement of the priority declaration is the operator declaration:
op (real, real) real += some semantics.
To show the position of the operator in an applied occurrence and to define the formal parameters the operator declaration should be changed to a form declaration.
form real a + real b ⟶ real = some semantics.
The form declaration is more general than the operator declaration:
form if bool a then real b else real c fi ⟶ real = some semantics.
This declaration implies the priority declarations:
priority if = (maximum, minimum), priority then = (min, min)
priority else = (min, min), priority fi = (min, max)
If a form is open to the left (which means: it begins with an operand) the first operator is called infix and needs an explicit priority declaration, similarly when a form is open to the right. So the else and the do of Algol 60 require explicit priority declarations.
monadic - = 6 means: priority - = (max, 6)

MONADIC OPERATORS. TWO DEVIATIONS OF AN OPERATOR GRAMMAR.

When the same operator symbol is used to represent dyadic as well as a monadic operator (e.g. the minus symbol), the strong rule that a symbol may have only one left and right priority is violated. In this special case however, the violation can be allowed with the following analysis: If an operator that must be reduced, requires an operand at its right side and finds an operator, it examines whether that operator is allowed to be monadic. If it is not, then a syntactic error has been discovered, otherwise that second operator must be reduced first, as a monadic operator.
Every form declaration has to contain one operator symbol at least. There are no restrictions on the number of operands that follow each other immediately. The priorities of the operators determine which operator must be reduced and then this operator asks for that number of operands on the left and on the right it wants itself. So prefix and postfix dyadic operators are allowed. The main function of an operator symbol is not to have priorities but to give the union information. The union information is necessary if two or more production rules have the same kinds of operands (e.g. the addition and the multiplication). If two kinds of operands have only one corresponding production rule or if one corresponding production rule is selected, the union information may be absent, e.g. the procedure call: ln sin x. This asks for a Wirth & Weber precedence. Above, operands have been defined as those syntactic units that have no precedence information. Now operands have precedence functions but these functions are neglected if an operand is enclosed between operators. The harmonic cooperation between the monadic operators and the first and the second deviation of an operator grammar needs further examination before examples of explicit formal syntactic definitions can be given.

CHARACTERIZING A RULE.

When more abstract syntax rules exist with the same fields, in the concrete syntax an extra (operator) symbol is needed to obtain the union information. Now it would be wasteful to reserve this symbol for this purpose only. It is sufficient when the combination of the operator and its operands indicates the rule. So, more operator declarations having the same operator symbol are allowed, e.g.: int +, real +, complex +. However, large syntactic forms need the characteristic information at the beginning in order to give the human reader an easy top down survey. Following operators are allowed to be not characteristic and may occur in several form declarations as separators. In a large syntactic form parts following the characterizing information should be allowed to be defined optional. The language designer is strongly advised to define all large syntactic forms such that they are opened by a prefix symbol which is characterizing on its own and such that they are closed by a postfix symbol which is characterizing too. The postfix symbol is important to facilitate the recognition of the extent of the large form. By this a number of different types of brackets is introduced: ( ), if fi, case esac, for do od. This decreases strongly the need for counting the brackets. Brackets remain necessary when infix operators are used for raising the priority. So infix operators, normal brackets and special brackets together with prefix information are important to get an easy top down survey. The readability can also be improved by underlining exactly those symbols that have priority. So the booleans: true and false, the primitive modes: real, int, char and bool and all mode indications must loose their underlining and the difference between indications and identifiers disappears.

ABSTRACT METASYNTAX.

The abstract syntax has previously been defined by mode declarers, which generate new modes (syntactic units) from old ones, and by mode declarations, which connect new indications (identifiers) with generated modes. Such an indication on an applied occurrence can be substituted by the right side of its defining occurrence. To obtain a recursive mode a declaration is necessary because the indication cannot be removed by substitution. So it becomes an entity in itself and the right side of its declaration becomes a set of rules by which the indication can be generated. The definition of aexpr presented above was not satisfactory since new operator declarations change the mode of aexpr. Now a new operator declaration gives aexpr a new rule, but aexpr remains itself. The (concrete) metasyntax of ALGOL 68 implies an infinite number of syntactic units. The abstract syntax defined so far, only allows a finite number. So it should be made more powerful. The metasyntax of ALGOL 68 appears to be really a metasyntax. A metalevel contains schemes by which objects and rules of the lower level are generated. Indeed, the metasyntax is a means to generate syntactic units and syntactic rules. The most simple form of such a scheme that generates syntactic units, is a function that maps the values of a certain set into syntactic units. The last ones are created by this mapping. This is not compatible with the definition of syntactic units by mode declarations, but it is when they are entities in themselves. The function may be considered as

a syntactic unit with one or more parameters in it. A set of values that is mapped into syntactic units is called a metanotion in ALGOL 68. There, a metanotion is defined by a BNF grammar. The set is a set of terminal strings then. However, it is better to take the lesson of ALGOL 68 that every set should be defined by a mode declaration. The BNF rule that defines the metanotion SORT:
SORT ::= soft | weak | strong | firm;
may be read as: the set SORT has the values soft, weak, strong and firm. The mode of an abstract finite set can be defined as integer.
So:
mode sort = int;
sort soft = 1, sort weak = 2, sort strong = 3, sort firm = 4;
At the same way the meta notion: PRIMITIVE MODE can be defined as an integer with the values real, int, char and bool. The metanotion MODE needs a more complex declaration:
mode mod = union (primitive mode p, array mod union, array struct (mod, identifier) struct, mod ref, mod row, struct (array mod, mod) proc)
Here again a union is wanted that may have the same mode more than once.
Examples of schemes that generate abstract syntactic rules:
mode mod condexpr = struct (boolexpr, modexpr, modexpr)
mode ref mod assign = struct (ref mod expr, mod expr)
These examples show the power of the metasyntax: as mod is an infinite set, an infinite number of abstract rules is generated by these two examples.
When a metanotion occurs more than once in a scheme, it should be replaced by the same value at all occurrences.
"Ref" should be considered as a monadic operator that maps a metavalue on a new metavalue.

CONCRETE METASYNTAX.

For a good cooperation between meta- and usual syntax rules it seems necessary to require that the usual syntax is totally independent of the metasyntax. This means that during the usual analysis the metavalues within the syntactic units stay undefined. So for a moment the functions that may metavalues on syntactic units become syntactic units themselves. After the labelled tree has been generated by the usual rules the metavalues get relevant. Some labels cause the creation of a metavalue in their corresponding mode. Then such a metavalue may pass some modes up or downwards until it is used. A consistent use of metavalues requires them to walk in one direction only, either up or down. This direction could be fixed in a declaration:
meta sort down
Indeed after being generated in some node, the values of SORT always walk downwards and get used there. The metanotion MODE does walk in both directions. The direction appears to depend on the value of SORT:

meta soft, mod up	meta strong, mod down
meta weak, mod up	meta firm, mod down

The best example is the assignation:
form soft ref mod expr := strong mod expr ⟶
    ref mod coercend = ...
Soft descends along the tree until a coercend is reached. There a value of mod is generated by the soft coercion that strips all possible proc's of the mode contained within the coercend. After this the value climbs the tree until the assignation is reached. As a ref mod expression is

expected, the value which is a union value, is examined on being the ref case of the union. If it is not then an error has been found, because no other form declaration with ":=" exists. If it is then the value of that case is taken and is considered as the value of mod in the assignation. So the first occurrence of mod denotes a defining occurrence, and the other denote applied ones. Indeed the second occurrence must be an applied one as the presence of strong indicates that the mod value must be passed downwards.

THE COERCION.

The coercion is that part of the grammar that consists of rules which reduce only one syntactic unit into another one. Most of them have disappeared by means of the precedence analysis: e.g. term and factor get identical. Only the production rules with a really semantic meaning are left over, e.g.
identifier ⟶ variable identifier
variable identifier ⟶ primary.
These rules represent monadic operations and could be preceded by a monadic operator symbol. However, if it is possible, a short notation is important, as well for the size itself as for a good top down survey over the program. The information which rules must be applied, has to be provided by the context. It is called the syntactic position. The context of a syntactic unit is the form in which it is an operand. So it should be possible to provide the position of an operand in a form declaration with the information which coercions must be applied. This information should descend to the operand. The metanotion sort contains this information exactly and indeed it walks top down. At the moment that the coercion should be executed, two or three kinds of information are available. Firstly the syntactic position e.g. a value of sort, secondly the syntactic unit a priori, which has already been generated by a bottom up process and thirdly the syntactic unit a posteriori that should be reached by the coercion. The last information is only available if provided by the form of which this unit is an operand. This information should descend the tree together with the syntactic position. An example is the right side of an assignation. The syntactic position is strong. The mode is known from the left side and descends the tree. The definition of syntactic position implies that a set of rules is fixed, or better that an algorithm is fixed. An algorithm is a procedure. So a syntactic position should be considered as a procedure, which is called at the moment that the descent of the syntactic position is stopped because a coercend has been reached.
coercion soft up = (proc mod coercend ⟶ mod coercend; soft mod coercend ⟶ mod expr)
'up' means that the a posteriori mode or unit is not available so that only bottom up rules may be applied. The result should climb the tree. The meaning of the right side is: If the meta value within the coercend is the proc case of the union, and if its parameter part is void, then a new node is built with the result part of the proc as its meta value, after which the coercion soft is called again, else the mod coercend becomes a mod expr and the coercion has been finished. In order to describe the body of such a procedure a usual, powerful algorithmic language (e.g. Algol 68) is needed which has to be extended with operations that generate new syntactic nodes bottom up or top down and with the possibility of backtracking. The algorithmic nature of the body prevents ambiguities because the produced syntactic (sub)tree is

uniquely determined by the input; among other things by a serial treatment of syntactic alternatives. A same approach appears advantageous to treat competitive operator declarations e.g. int +, real +, complex +. A form: 'a+b' is equivalent to a procedure call: plus (a,b). The parameters of a call may have a strong position as the mode of the procedure contains the mode of its parameters. So a strong position is wanted for the operands of a form too. This increases the danger of ambiguities. Algol 68 solves the problem as well by restrictions on operator declarations as by weakening the coercion to the firm coercion. The compromise makes both things dirty. Both give troubles when learning the language. Both decrease the syntactic power of the language: The stronger the coercion is, the more restrictions to operator declarations are necessary. The syntactic order solves the problem. When the alternatives should be treated in a predetermined sequence ambiguities are excluded, the coercion may be so strong as possible and operator restrictions become superfluous.
The preference in the treatment of the alternatives could be defined by:
form int a + int b pref 1 ⟶ int = ...
form real a + real b pref 2 ⟶ real = ...
form complex a + complex b pref 3 ⟶ complex = ...
By means of the strong coercion together with the preference the addition of an integer and a complex is considered as an addition of two complex numbers. This appears to be the usual mathematical convention. Each coercion could be replaced by an equivalent monadic operator and each quasi ambiguous operator symbol could be replaced by a number of characterizing procedure identifiers. So coercion and quasi ambiguous operators are only a matter of notation and are therefore absent in the abstract syntax. Without coercion metavalues need not descend the tree. The strictly bottom up causality of metavalues gives the abstract syntax its desired simplicity, although we do not yet understand the meaning of causality in an abstract syntax.

CONCLUSION.

We have seen that every (one level) abstract syntax rule can be represented by an operator precedence form. Although already rather good, this representation may be improved by deviations of an operator grammar, coercion and by quasi ambiguous operators. Further it seems possible to construct a definition mechanism by means of which an easily analysable metasyntax can be declared.

Reference.

Wijngaarden A. van, et al. Report on the algorithmic language Algol 68. Mathematisch Centrum Amsterdam, MR 101, February 1969.