Acta Informatica 4, 183—192 (1975)

Ambiguity and Precedence in Syntax Description*

Jay Earley

Received May 3, 1974

Summary. This paper describes a method of syntax description for programming languages which allows one to factor out that part of the description which deals with the relative precedences of syntactic units. This has been found to produce simpler and more flexible syntax descriptions. It is done by allowing the normal part of the description, which is done in BNF, to be ambiguous; these ambiguities are then resolved by a separate part of the description which gives precedence relations between the conflicting productions from the grammar. The method can be used with any left-to-right parser which is capable of detecting ambiguities and recognizing which productions they come from. We have studied its use with an $LR(1)$ parser, and it requires a small and localized addition to the parser to enable it to deal with the precedence relations.

Introduction

Whether we are designing a language, trying to use a language, or trying to extend a language, when dealing with syntax, we are first interested in the form of a construct—what operations does it involve, what does it look like—as opposed to what its precedence is or whether it introduces an ambiguity into the grammer. Later, when we actually use the construct in a doubtful situation or actually try to construct a parser, we become interested in the latter issues. Reynolds in [4] actually presents a language specification in this way. He first gives a highly ambiguous but simple definition of the syntax, and later gives a separate unambiguous definition which includes all the precedence information. In this paper we improve on that by providing a method whereby a second grammer is not necessary; only additional precedence specifications are needed, to go along with the first grammer. We used this method in the preliminary design of a programming language called VERS 2, and we will refer briefly to our experiences with it in the paper[1]. However, we will use Algol [1] for our examples because of its familiarity.

A similar method has been developed independently at Bell Labs [10].

Precedence Relations

Consider the following BNF for a subset of Algol arithmetic expressions:

$$\langle \text{EXP} \rangle ::= \langle \text{TERM} \rangle \mid \langle \text{EXP} \rangle + \langle \text{TERM} \rangle$$

$$\langle \text{TERM} \rangle ::= \langle \text{IDENT} \rangle \mid \langle \text{TERM} \rangle * \langle \text{IDENT} \rangle$$

* This work was supported by the National Science Foundation under contract GJ 34342X.

1 This design was later modified because we decided to implement VERS 2 as an extension of ECL.

This conventional form gives multiplication higher precedence than addition and makes them both left associative. However, we can write a simpler ambiguous grammer which describes the allowable expressions correctly, but ignores precedence and associativity issues:

$$\langle \text{EXP} \rangle ::= \langle \text{IDENT} \rangle \mid \langle \text{EXP} \rangle + \langle \text{EXP} \rangle \mid \langle \text{EXP} \rangle * \langle \text{EXP} \rangle$$

We can then add a separate precedence description as follows: we name the second and third productions above PLUS and MULT respectively, and then express the relation as

$$\text{MULT} > \text{PLUS}$$
$$\text{PLUS } L \text{ PLUS}$$
$$\text{MULT } L \text{ MULT}$$

This specifies that MULT has greater precedence than PLUS and that MULT and PLUS are both left recursive.

More specifically, the relation "MULT > PLUS" means that there is an ambiguity in the grammar which will be first discovered (in a left-to-right parse) when there is a conflict between MULT and PLUS as to which should be used to construct the next branch of the parse tree. Furthermore, it states that MULT should be used in this case and the PLUS branch ignored. Similarly, the "PLUS *L* PLUS" relation means that an ambiguity involving PLUS and PLUS may also occur and that this is to be resolved by taking the leftmost of the two productions.

Notice that the precedence relations we introduce here are between productions, not between terminals or non-terminals as are those of the precedence parsers.

When dealing with more than two levels of precedence, it is convenient to use a slightly different syntax to express the relations—one in which all the relations don't have to be written out explicitly. With this new syntax, the previous example is written as follows:

$$\text{PLUS}(L)$$
$$\text{MULT}(L)$$

The productions with higher precedence are written last.

We are now in a position to write the syntax for a slight generalization of Algol arithmetic expressions.

⟨AE⟩ ::= ⟨if clause⟩ ⟨AE⟩ *else* ⟨AE⟩ |	CONDAE
⟨AE⟩ ⟨add op⟩ ⟨AE⟩ |	BINADD
⟨add op⟩ ⟨AE⟩ |	UNADD
⟨AE⟩ ⟨mult op⟩ ⟨AE⟩ |	MULT
⟨AE⟩ ↑ ⟨AE⟩ |	EXPON
⟨primary⟩	

We have employed some obvious abbreviations for non-terminals in the Algol Report. Everything not defined here is as in the Report. The precedence relations

are:

CONDAE
UNADD, BINADD(*L*)
MULT(*L*)
EXPON(*L*)

Notice that BINADD and UNADD are marked as being at the same level of precedence, and that CONDAE does not need to be marked (*L*) because it can't be ambiguous with itself.

Similarly, we can give a factored syntax description of Boolean expressions:

⟨BE⟩::=⟨if clause⟩BE⟩ *else* ⟨BE⟩|	CONDBE
⟨BE⟩≡⟨BE⟩|	EQUIV
⟨BE⟩⊃⟨BE⟩|	IMP
⟨BE⟩∨⟨BE⟩|	OR
⟨BE⟩∧⟨BE⟩|	AND
¬⟨BE⟩|	NOT
⟨Boolean primary⟩	

CONDBE
EQUIV(*L*)
IMP(*L*)
OR(*L*)
AND(*L*)
NOT

This still leaves us with some ambiguities (which are also present in the Algol Report). To illustrate one of these, we show the definitions of primaries and general expressions:

⟨primary⟩::=⟨unsigned number⟩\| ⟨variable⟩\|
⟨function designator⟩\| (⟨AE⟩)

⟨Boolean primary⟩::=⟨logical value⟩\| ⟨variable⟩\|
⟨function designator⟩\| (⟨BE⟩)\|
⟨AE⟩.⟨rel op⟩⟨AE⟩

⟨E⟩::=⟨AE⟩\| ⟨BE⟩\| ⟨designational E⟩

Note that an ⟨E⟩ which is simply a ⟨variable⟩ or a ⟨function designator⟩ has two parses—one as an arithmetic and one as a Boolean expression. This ambiguity is different in nature from the ones we have examined so far. The others were introduced onto the original Algol grammar in writing it in the simpler, precedence-free form, and they are resolved by precedence relations. This one was in the original grammar and is resolved by semantic (type) information. There are two ways to handle this problem, and each illustrates another feature of our descriptive method. One involves resolving the ambiguity with type information and is covered in a later section of the paper. The other involves more changes to the grammar, and we discuss that now.

We could have written the grammar so that type information is not reflected at all in the BNF:

⟨E⟩::=⟨if clause⟩⟨E⟩ *else* ⟨E⟩|	COND
⟨E⟩≡⟨E⟩|	EQUIV
⟨E⟩⊃⟨E⟩|	IMP
⟨E⟩∨⟨E⟩|	OR
⟨E⟩∧⟨E⟩|	AND
¬⟨E⟩|	NOT
⟨E⟩⟨rel op⟩⟨E⟩|	REL
⟨E⟩⟨add op⟩⟨E⟩|	BINADD
⟨add op⟩⟨E⟩|	UNADD
⟨E⟩⟨mult op⟩⟨E⟩|	MULT
⟨E⟩↑⟨E⟩|	EXPON
⟨primary⟩	

⟨primary⟩::=⟨logical value⟩\|⟨unsigned number⟩\|⟨variable⟩\|
⟨function designator⟩\|(⟨E⟩)

BE PREC:	AE PREC:
COND	COND
EQUIV(*L*)	REL(*N*)
IMP(*L*)	UNADD, BINADD(*L*)
OR(*L*)	MULT(*L*)
AND(*L*)	EXPON(*L*)
NOT	
REL(*N*)	

Here we have two columns of precedence relations, corresponding to those for Boolean and arithmetic expressions. The REL production (which didn't require a precedence relation in the old grammar) illustrates a new precedence relation *N*. This means that although REL may be ambigous with itself according to the grammar, expressions involving this ambiguity are to be illegal (if these are the only two possibilities). Thus an expression such as "A=B=C" should produce a syntax error. (This is what is desired in Algol since "=" is an arithmetic relation only.) Note that this is different from having no precedence relation between REL and itself. If that were the case, it would mean that REL was not supposed to be ambiguous with itself in the grammar. In this case, if the parser detected such an ambiguity it would produce an ambiguity error for the grammar, not a syntax error for the string being parsed.

Notice that there can be more than two ambigous possibilities for a given string. These are handled two at a time by the precedence relations. In the case that two of them have an *N* relation between them, the string is not immediately marked illegal. Instead, both possibilities are rejected and processing continues since there may be a third valid possibility.

In addition to the *N* relation between REL and itself, we would also like to have this relation between each production in the left column and each in the right (except, of course, for those which are in both). We can signify this using

the column names as follows:

BEPREC N AEPREC

This completes the new description of Algol expressions. We have allowed some semantically meaningless expressions such as

TRUE + FALSE,

but these can be ruled out with type information very easily. We have described Algol expressions this way in order to illustrate the N relation, not necessarily because this is the best way to describe Algol. In fact, with a language like Algol in which it is possible to know all type information at compile time, it may be best to reflect type distinctions directly in the grammar. But with some of the more recent languages in which types may not be known until run time, it may be advisable as we have done here to have just one overall grammar for all types of expressions.

So far we have introduced four precedence relations and in the next section we will introduce a fifth. These can be summarized as follows:

$A > B$	Use A over B.
$A\,L\,B$	Use the leftmost of A and B.
$A\,R\,B$	Use the rightmost of A and B.
$A\,N\,B$	If A and B conflict, reject both possibilities.
$A \sim B$	Type resolved conflict (see next section).
$A - B$	If no relation holds, then A and B may not conflict.

These five relations and their use form the important part of this paper; the abbreviations we have employed to avoid writing out all the combinations of precedence relations are less important.

It may happen that we want to use an operator (such as "$\leftarrow$") which has a different prededence on the left than on the right. This situation can still be handled with the five relations listed above. For example, if "$\leftarrow$" is to have high precedence to its left and low precedence to its right, it would have an R relationship to the normal operators, i.e.

STORE R PLUS.

Type-Resolved Ambiguity

In the description of a programming language we frequently have a syntactic ambiguity which is not an ambiguity at the semantic level—usually because type information resolves the conflict. Instead of requiring the grammar to be rewritten, we can set up the compiler so that it works with the ambiguous grammar, using types to resolve conflicts. There are four ways to do this, each with its drawbacks. We will discuss each, using the first Algol grammar (with separate rules for arithmetic and Boolean expressions) as an example.

(1) The simplest method is to have the lexical analyzer look up type information for variables and function names and return different tokens to the parser depending on the type that it finds. This can actually be done in the Algol example

(see [2], p. 622). We would change the primary definitions as follows:

⟨primary⟩::=⟨unsigned number⟩|⟨arithmetic variable⟩|
⟨arithmetic function designator⟩|(⟨AE⟩)

⟨Boolean primary⟩::=⟨logical value⟩|⟨Boolean variable⟩|
⟨Boolean function designator⟩|(⟨BE⟩)|
⟨AE⟩⟨rel op⟩⟨AE⟩

This removes the ambiguity from the grammar entirely, and it can be handled by an ordinary parser. The method requires that (a) the language be such that the type information is available at compile time, (b) the compiler be such that this type information is available during parsing, and (c) the grammar has been written so that the type information is always reflected in the non-terminals (i.e. ⟨BE⟩ instead of ⟨E⟩). This third restriction may become unwieldy when there are many types.

(2) A less restrictive method is to set up the parser to tolerate the ambiguity until it needs to resolve the conflict and then to request the semantic information it needs at that time. This would involve adding a new precedence relation "$\sim$" to indicate that a type-resolved ambiguity exists between two productions. If we name productions as follows:

⟨primary⟩::=⟨variable⟩|	AVAR
⟨function designator⟩	AFUNC
⟨Boolean primary⟩::=⟨variable⟩|	BVAR
⟨function designator⟩	BFUNC

then we can specifiy these new precedence relations along with the types necessary to distinguish them:

AVAR(real, integer)$\sim$BVAR(Boolean)
AFUNC(real, integer)$\sim$BFUNC(Boolean)

Then, when the compiler discovers such a conflict, it looks to see which type it actually has, and takes that branch. This method suffers from the same first two disadvantages (a and b) as did the previous method, but not from the third (c).

(3) This method can be used when the language or system is such that type information is not available during parsing. (VERS2 is such a language.) We require that when the programmer uses the new relation "$\sim$", he is claiming not only that a type resolved ambiguity exists, but that it is of a very simple kind; that is (a) each of the two paths of the ambiguity has exactly one semantically meaningful production (that is, one which produces code), (b) these productions are the ones marked with "$\sim$", (c) they have the same length, (d) the productions can be distinguished by the types associated with their non-terminals and (e) all other productions in the conflicting paths are of length 1. The parser then follows one path or the other, but compiles code for both productions in conflict, plus code which checks the types of the appropriate objects and executes whichever code is appropriate to the type.

The disadvantage of this method is that the ambiguity must be of the simple kind mentioned above. However, we have found all of our examples in VERS2 so far to fit within this framework, so we favor this third method.

(4) This last method is only mentioned here as a possibility, because we have not worked it out. The most flexible way would be to use a parser which is capable of following all ambiguous paths simultaneously and recognizing when they come back together again. (See [3].) Then the compiler would attempt to generate code for all the paths, plus code to distinguish between them on the basis of type information. It would also have to check to see if the paths could be distinguished on the basis of type information. The disadvantage of this is that it requires a special parser, that it seems difficult to work out, and that it is perhaps not worth the effort for the extra cases it might cover.

Other Uses of Precedence

So far we have used precedence relations solely to resolve ambiguities in grammars for expressions involving binary and unary operators, and the precedence relations have established priorities among the operators. Our precedence relations have broader use than this and we present one such example now. In the unrevised Algol Report, the BNF for **if** statements and conditional statements was roughly as follows:

⟨statement⟩::=⟨if clause⟩⟨statement⟩| IFST
⟨if clause⟩⟨statement⟩ **else** ⟨statement⟩ CONDST

This is ambiguous because the ⟨statement⟩

if B **then if** C **then** $D:=1$ **else** $D:=2$

is ambiguous; the **else** might go with either of the **then**'s. This was corrected in the Revised Report by requiring an ⟨unconditional statement⟩ after a **then**; such statements do not include **if** statements of **for** statements unless they are surrounded by **begin** and **end**. The unfortunate results were to complicate the grammar a certain amount, and to rule out such unambiguous statements as the following:

if B **then if** C **then** $D:=1$ **else** $D:=2$ **else** $D:=3$
if B **then for i**$:=1$ **step** 1 **until** N **do** $D:=1$ **else** $D:=2$

Using our precedence relations we can do better than this. We don't need to change the original grammar at all; we just specify a precedence relation between the two ambiguous productions. If we write

CONDST > IFST

then the ambiguous statement is parsed as follows:

if B **then if** C **then** $D:=1$ **else** $D:=2$

and if we write

IFST > CONDST

then it is given the other interpretation. All the unambiguous statements are, of course, legal, and the grammar remains in the simple form.

This is not an isolated example, even though it may be the only one in Algol. We have found a half dozen cases in VEBS2 where the precedence relations were used for other than operator precedence.

Parsing

The requirements for a parsing method to be usable with our scheme are

(1) that it build the tree from bottom-to-top, left-to-right (because that is how the precedence relations are defined),

(2) that it be able to detect ambiguities, and

(3) that it know when it detects them which productions they came from (so that it can then resolve them). Notice that we do not require that the parser detect **only** ambiguities; it must detect all ambiguities, but if it rejects grammars which are in fact not ambiguous, that is all right. Most fast parsers for compilers do just that. They accept only a certain class of grammars—which is generally a subclass of the unambiguous grammars. So any grammar they accept has no ambiguities. Such a parser will use our method by potentially treating any rejected grammar as if it were ambiguous. If there is, in fact, a precedence relation between the conflicting productions, it assumes that it was an ambiguity and resolves it; if there is not a precedence relation, it rejects the grammar. We will describe in detail here the modifications required in a parser to adapt it to our method, first for a parser that works directly from the grammar and second for one which has a construction phase.

A direct parser cannot reject the grammar a priori since it does no preprocessing, but it must be set up to detect ambiguity (or violation of the set of grammars it accepts) in a particular string it is parsing. When this happens, it refers to the precedence relations. If there is no relation between the two productions, it rejects the grammar. Otherwise, it chooses one of the productions according to the precedence rules and continues parsing as if the possible parse corresponding to the other production had never existed. If the two productions in question have an N relation, it rejects both of them.

A parse constructor has a construction phase in which it produces tables from the grammar which will later be used in the parsing. Generally ambiguity or grammar rejection is discovered during this phase. When this happens, the constructor refers to the precedence relations and produces the table entries which correspond to the correct production indicated by the relations. In the case of an N relation where there are no other possibilities, it must produce a table entry which will allow the conflict to arise and will produce a syntax error during parsing in response to it. In the case that there is no precedence relation between the offending productions, of course, it rejects the grammar.

The $LR(k)$ parsing methods [2, 5, 6] fit our criteria and therefore could be used with our method. The standard top-down and bottom-up methods [7] probably cannot because they do not detect ambiguities as they proceed in the

parsing. The precedence methods [8. 9] probably cannot because they don't know which productions are involved in an ambiguity (precedence violation).

Conclusions

We have used this method in the preliminary design of VERS2 and found it to be quite useful in working out the syntax. About a third of the productions of the grammar are involved in precedence relations[2]. Specifically, the use of our method cuts down on both the number of productions in the grammar and the number of non-terminals. It especially cuts down on the number of meaningless non-terminals (⟨term⟩, ⟨factor⟩) and almost eliminates the need for productions having just a single non-terminal on the right (such as ⟨term⟩::=⟨factor⟩). This serves to decrease the space required for the parser and to speed it up.

More important, however, is the factorization of the syntax description. This has aided us in designing a language and we expect it to aid programmers reading a manual, but we suspect that the biggest advantage is its flexibility. In a language with extensible syntax, extension would be done by modifying the grammar and precedence rules. Consider the problem of adding a new operator "·" to Algol with a precedence between "+" and "=". With our method, we simply add one new production "⟨E⟩::=⟨E⟩ ⟨E⟩" and one new entry in the AEPREC list. But to add such a thing to the BNF in the Algol Report would require modifying a couple of rules, and adding a new non-terminal and two new rules.

In addition, syntax extension can be done without worrying initially about precedence at all. A new rule is added to the grammar and it is run through the parse constructor. If it introduces an ambiguity the constructor will inform the user and then allow him to add a precedence rule in order to resolve it. Extending the syntax should be easier for the naive user also because there will be fewer non-terminals to worry about. For instance, almost everything in Algol would be an ⟨E⟩ or a ⟨statement⟩, and the user would often be able to make extensions in terms of these without referring to the grammar at all.

References

1. Naur, P., *et al.*: Revised report on the algorithmic language ALGOL 60. Comm. ACM **6**, 1–17 (1963)
2. Korenjak, A. J.: A practical method for constructing $LR(k)$ processors. Comm. ACM **12**, 613–623 (1969)
3. Earley, J.: An efficient context-free parsing algorithm. Comm. ACM **13**, 94–102 (1970)
4. Reynolds, J. C.: A set-theoretic approach to the concept of type. Argonne National Laboratory, 1969
5. Knuth, D. E.: On the translation of languages from left to right. Information and Control **8**, 607–639 (1965)
6. De Remer, F. L.: Simple $LR(k)$ grammars. Comm. ACM **14**, 453–460 (1971)
7. Floyd, R. W.: The syntax of programming languages—a survey. IEEE Transactions EC-**13**, 4 (1964)
8. Floyd, R. W.: Syntactic analysis and operator precedence. J. ACM **10**, 316-333 (1963)

2 We are using an augmented BNF notation which reduces the number of productions substantially. Using ordinary BNF, it might be 10–15%.

9. Wirth, N., Weber, H.: Euler—a generalization of ALGOL and its definition. Comm. ACM 9, 13–25 (1966)
10. Aho, A. V., Johnson, S. C.: *LR* Parsing. Bell Laboratories, Murray Hill (N. J.)

Jay Earley
College of Engineering
Computer Science Division
University of California
Berkeley, Cal. 94720
USA