SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 26(7), 763–779 (JULY 1996)

An Operator Precedence Parser for Standard Prolog Text

KOEN DE BOSSCHERE*
Vakgroep Elektronica en Informatiesystemen, Universiteit Gent, Belgium

SUMMARY

Prolog is a language with a dynamic grammar which is the result of embedded operator declarations. The parsing of such a language cannot be done easily by means of standard tools. Most often, an existing parsing technique for a static grammar is adapted to deal with the dynamic constructs.

This paper uses the syntax definition as defined by the ISO standard for the Prolog language. It starts with a brief discussion of the standard, highlighting some aspects that are important for the parser, such as the restrictions on the use of operators as imposed by the standard in order to make the parsing deterministic. Some possible problem areas are also indicated. As output is closely related to input in Prolog, both are treated in this paper.

Some parsing techniques are compared and an operator precedence parser is chosen to be modified to deal with the dynamic operator declarations. The necessary modifications are discussed and an implementation in C is presented. Performance data are collected and compared with a public domain Prolog parser written in Prolog.

It is the first efficient public domain parser for *Standard Prolog* that actually works and deals with all the details of the syntax.

KEY WORDS: Prolog; ISO standard syntax; operator precedence parser; implementation

INTRODUCTION

To enhance readability, most modern programming languages allow for the use of operators in expressions (e.g., $a + b$) instead of a pure functional notation, such as $+(a, b)$.

The *behavior* of an operator consists of its syntax and semantics. *Overloading* describes the process by which an operator might have more than one behavior while keeping the same appearance. In most languages the binary plus for integer addition is overloaded with the floating point addition. In this case the semantics of the operator is called overloaded; the syntax is the same in both cases. In the case of the unary and binary minus, the syntax is overloaded too because the minus operator can be either a prefix operator or an infix operator. Furthermore, an operator can be *statically overloaded* (fixed in the language definition), or *dynamically overloaded*, (by the user of the language).

Hence, both syntax and semantics can be either (i) non-overloaded (NO), (ii) statically overloaded (SO), or (iii) dynamically overloaded (DO). This gives rise to a taxonomy along two axes, syntax and semantics (see Figure 1). Most languages provide at least static overloading of the semantics for reasons of convenience. Otherwise, the number of operators would become extremely large, especially when the users can add their own types to the language.

* Address for correspondence: Koen De Bosschere, RUG-ELIS, St.-Pietersnieuwstraat 41, B-9000 Gent, Belgium

CCC 0038–0644/96/070763–17 *Received 24 December 1994*
Revised 20 November 1995

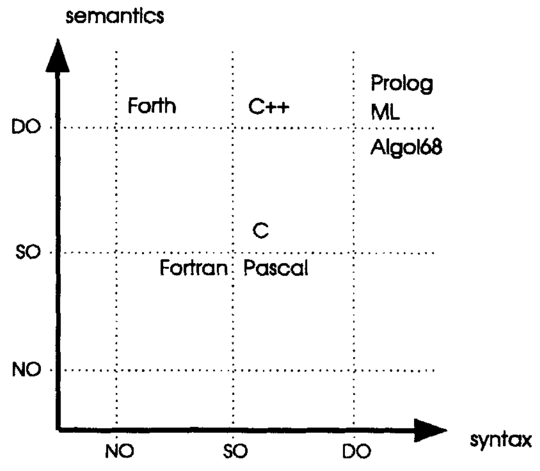

Figure 1. Taxonomy

Some languages provide a fixed set of operators, with a default semantics, but this semantics can be dynamically *overloaded.* This is the case in C++ where one has access to a broad set of operators, and where a new meaning can be given to these operators. However, the *attributes* of the operators (priority, associativity) remain fixed.

Finally there are the languages such as Algol68,[1] Prolog[2] and ML[3] that offer the greatest flexibility to the user. One can change both the syntax and the semantics of the operators.

Since the use of operators is a matter of syntax (operator-based expressions can always be replaced by functional expressions), the only part of a compiler that is affected by it is the parser.

The problem of parsing expressions containing a fixed and predefined set of operators (i.e., NO or SO) is well understood. Program generators such as `yacc`[4] or `bison` can generate parsers for grammars containing operators. Hereby, the operator declarations are given at a high level.

The parsing task is however entirely different when the operator attributes can be changed or when operators can be added (DO). For these, so-called *dynamic grammars,*[5] the parser cannot know in advance which operators will be in effect and how they should be processed. The operators must then be stored in an operator table and they will have to be consulted while processing an expression in order to find a correct parse tree for it.

In Reference 5 a general framework to generate parsers for dynamic grammars is presented and applied to Prolog and standard ML. A table driven LR parser with deferred decision parsing is used to solve the shift reduce conflicts at run time rather than at table construction time. This solution is in a sense too powerful to parse Prolog as defined in the ISO standard because many of the constructs that cause conflicts have been removed. Although it still has a dynamic grammar, a less complex parser is able to do the same job.

This paper describes an operator precedence parser for the Prolog language as defined by the ISO standard. Since a Prolog program and Prolog data have the same syntax (read-terms), both the compiler as the prolog `read` make use of the parser. Furthermore, the Prolog output routine is closely related to the Prolog input routine because the latter should be able to process the output of the former. Therefore, this paper treats input as well as output.

The main contribution of this paper is to show how an operator precedence parser can be adapted for the dynamic grammar of Prolog.

The paper is also a kind of evaluation of the standard. It explains why some of the restrictions are needed to disambiguate the syntax, and gives some suggestions to even further disambiguate it. In the course of this work, a few errors in the syntax description of the ISO standard have been found.

The algorithm was implemented, and is both efficient and well-documented. The implementation is not just another prototype, but really used in BinProlog.[6] The speedup with respect to a previously used public domain parser in BinProlog is about one order of magnitude. Hence, applications that do some input/output such as compilers can be accelerated significantly. This speedup can be considerable (about 50 per cent for the BinProlog compiler). The implementation is in the public domain and is fully operational in BinProlog.

The next section discusses some syntax issues that are important for either the input or the output routine. Subsequently, the algorithms used in both routines and their implementation is described. The paper concludes with some performance data.

The sequel uses some dedicated terminology for reasons of convenience. This paper ofter refers to '.' as *dot*, to ',' as *comma*, and to '(' as *open token*. Furthermore an operator expression can be written in *operator notation* (as in '`a f b`') or in *functional notation* (as in '`f(a,b)`'). The terminology that is borrowed from the ISO standard is printed in a `monospaced` type font.

THE SYNTAX OF PROLOG

The syntax of Prolog used in this paper is the syntax described in the ISO standard for the Prolog language.[7] It is quite remarkable that, in contrast to many claims that syntax of Prolog is simple ('it is only terms'), the Prolog standard needs about ten pages to precisely describe the syntax. About 60 per cent (six pages) has to do with lexical issues (definition of tokens), the other 40 per cent has to do with parsing (definition of terms). About 50 per cent of the parsing part is devoted to operators, indicating that this is a major issue in the syntax of Prolog.

In a programming language, it does not make sense to allow ambiguous constructs, and they should be eliminated from the syntax. In general, when a fixed set of static operators is used, this does not cause problems because the language designer normally takes care of this. When the user is allowed to change the syntax, he might introduce ambiguities in the language.

The standard tries to disambiguate the syntax of Prolog as far as possible by imposing restrictions on the use of operators. It more or less claims that one look ahead character suffices to decide how to process the current token. There is at least one place where this is not true (see below).

The next section discusses aspects of the syntax that are important for the input resp. output routine.

Input syntax

The Prolog input routine is used to read so-called read-terms, i.e., a term followed by a dot and `layout text`. Since Prolog text consists of read-terms, the input routine is also used as frontend for a Prolog compiler. It is a combination of a lexer (constructing tokens) and a parser (verifying syntax and constructing terms).

There are a few remarks to be made about the lexical analyzer.

1. Layout text (space, carriage return, comment,...) can be inserted between any two tokens without changing the meaning of these tokens, except in two cases.
   - (a) Compound terms in functional notation (e.g., `f(a,b,c)`) must not have layout text between the principal functor and the `open token`. Hence `+(a,b)` $\neq$ `+ (a,b)`.
   - (b) Dot (.) is considered a `graphic token` when it is not followed by layout text (e.g., `f(.)`), but it is the `end token` when followed by layout text (as in `f(1). `).
2. A `graphic token` (+,:-,...) cannot start with the characters `/*` because this sequence is reserved to start `bracketed comment`.
3. A `quoted token` that forms a valid atom without quotes denotes that atom, except for quoted tokens containing a `backslash char`. Hence `'abc'` denotes `abc`, but `'\\/'` denotes `\/` and not `\\/`. Furthermore, as shown in Table I, `','` is clearly different from `,`. The idea behind it is that there are two commas: a comma token, and a comma atom. The former one is used to separate arguments in a compound list or elements in a list, the latter one is an atom, and can also be written within quotes. The comma atom is an operator, but the standard defines the comma token as an operator too, and with the same properties as the comma atom. The consequence is that the comma token can replace the comma atom where it is used as operator, but not where it is used as atom, or functor name. Furthermore, the comma atom can never replace the comma token where it is used as separator.

Table I. Possible uses of the comma

`a,b`	(correct)	`a ',' b`	(correct)
`f(,)`	(erroneous)	`f(',')`	(correct)
`,(a,b)`	(erroneous)	`','(a,b)`	(correct)
`f(a,b)`	(correct)	`f(a ',' b)`	(erroneous)
`[a,b]`	(correct)	`[a ',' b]`	(erroneous)

The claim that one look ahead character suffices to have a deterministic parser is not true when e.g., `e` is defined as an infix operator. In the case of `1.2e-7`, as least two look ahead characters are needed after the `e` to find out that it is part of a floating point number, and not the infix operator as in `1.2e-a`.

Operators are just syntactic sugar. This does not however mean that one could globally replace e.g., `op(exp1,exp2)` by `exp1 op exp2` when `op/2` is declared as an infix operator. E.g., it is correct to write `*(a+b,c)` but this is not the same as `a+b * c`. Even worse, one can write down the compound term `/(+,2)`, but `+ / 2` is syntactically incorrect because an operator (+) cannot be the immediate operand of another operator (/). So, one should write `(+)/2`. In general, one could replace `op(exp1,exp2)` by `((exp1) op (exp2))`, but this is hardly more readable than the functional notation.

Output syntax

The basic idea of the display routine is that it should display Prolog text in such a way that the input routine can read it back. There are two options that affect the display format: quoted versus non-quoted atoms, and functional versus operator notation. In general, `quoted tokens` must keep their quotes in order to be read back correctly. So, as a matter of fact, the only two choices left are: quoted atoms with either functional or operator notation.

The *functional notation* does not cause problems, except in one place. The *comma* does not need quotes in general, but should be quoted in expressions like `f(',')` and `','(a,b)` (see also Table I).

The *operator notation* is more complex. There are two major issues: the use of parentheses, and the insertion of layout text.

The use of parentheses. Expressions must be parenthesized when needed. The expression `+(*(a,b),c)` should be displayed like `a*b+c`, but `*(+(a,b),c)` must be displayed like `(a+b)*c`. So, the display routine has to take into account the information about associativity and precedences.

With the operator declarations*

```
op(300,fy,++).     op(300,yf,--).
```

the term `++(--(a))` could be displayed as `++a--`, but `--(++(a))` should be displayed as `(++a)--` (see below).

The operator declarations

```
op(300,fy,&).     op(300,yf,&).
```

give rise to three possible display formats for the compound term `&(&(a))`, namely, `& &a`, `&a&`, and `a& &`.

With operator declarations

```
op(300,fx,&).     op(300,xf,&).
```

possible solutions are: `&(&a)`, `& (&a)`, `(&a)&`, `&(a&)`, `& (&a)`, and `(a&)&`, i.e., mixed operator/functional formats, due to the lack of associativity.

With the operator declarations

```
op(300,fy,&).     op(300,xf,&).
```

the possible solutions are: `&a&`, `& &a`, and the mixed solutions `(&a)&`, `(a&)&`, `&(&a)`, `&(a&)`, `& (&a)`, and `& (a&)`.

Although this is not specified in the standard, the display routine should always choose the expression that has the best readability (whatever that is). It is a pity that nothing is stated about this in the standard. It means that it may be difficult to literally compare the output of a program, run on two different Prolog processors.

Insertion of layout text. The second problem is the problem of insertion of layout text to separate tokens. In order to minimize the amount of layout text inserted, the display routine has to keep track of the last token written†, and the next character to display. There are some simple rules. A `numeric token` shall never be followed by a digit, a variable token or an `identifier` shall not be followed by an alphanumeric character, a `graphic token`

* A Prolog operator definition consists of a priority ranging from 1 to 1200, the associativity parameter with `f` the operator, `x` a non-associative operand, and `y` an associative operand, and the name of the operator.

† The last character is not sufficient. E.g., a number that ends on the digit '2' can be immediately followed by the character 'a', an identifier ending on '2' cannot.

must not be followed by a graphic token character, and a `quoted token` should not be followed by a quote.

However, even when using these rules, there are some special cases that need attention such as the dot used as operator. The declaration `op(300,xfy,.)` makes that the expression `.(1,2)` is hard to display in operator notation. The format `1.2` must be read as a floating point number, the format `1. 2` is parsed with the dot as an `end token`, thus making `1 .2` the only possible display format. Hence, an integer can in this case never be followed immediately by a dot operator.

There is an even more complicated case. Suppose, the operator declaration `op(300, xf, e10).` is in effect. Then the term `e10(3.1)` must not be output as `3.1e10`, but should be output as `3.1 e10`, although, in general, a number may be immediately followed by an `alpha char` without causing ambiguity. In practice, it turns out that numeric constants are better separated by at least one space from the surrounding identifiers (not the graphic tokens) to enhance readability. E.g., `10 is 30 mod 20` is easier to read than `10is 30mod 20`. Both formats are allowed by the standard, though.

Another case is that a prefix operator should be separated from any open token that follows it by at least one layout character. Otherwise, it would be read back as a compound term, with a possibly different structure. Hence, the term `'-'(','(a,b))` should be displayed as `- (a,b)` and not as `-(a,b)`.

The next section explains which parser algorithm has been selected and why. Furthermore, it shows how it can be adapted to parse the Prolog language and indicates how to display Prolog terms in such a way that they can be read back again.

ALGORITHMS

Input routine

A variety of parsers are able to parse operator expressions. However, operator expressions generated by a dynamic grammar are much more difficult to parse with standard techniques.

Top-down parsing by means of a recursive descent parser is hard because this requires the operators to be hard-coded into the parser itself because a recursive descent parser is not table driven. Since the operators are not known at parser generation time, this is difficult. In order to handle the dynamic nature of the language, using a table driven method is a clear advantage.

The remaining parsers (predictive parsers, LR parsers and operator precedence parsers) are table driven. Unfortunately, incrementally updating the table for predictive parsers and for LR parsers is not trivial and might, in some cases, require the reconstruction of the global table. Indeed, adding one operator level requires a modification of the productions that possibly interact with this new operator, and hence the corresponding table entries.

An operator precedence parse table is easier to update, given the priority and associativity of an operator. An operator has exactly one row and one column in the table, and adding an operator means that a row and a column are added, and that the corresponding entries must be filled in. Hereby, the already existing part of the table does not change.

The basic reason why an operator precedence table can be updated incrementally is that this table is operator indexed, whereas the other tables are indexed with respect to parser states or productions. One operator might affect multiple parser states and multiple productions, which makes incremental update rather difficult.

Operator precedence grammar. Now that an operator precedence parser seems to be suited to handle the dynamic grammar of Prolog, it remains to be proved whether Prolog has an operator precedence grammar[4] or not. At first sight, it has. There are no empty productions, and no production has two adjacent nonterminals.* However, a closer look reveals that there are some problems.

1. The above statement is only true if the operators are statically known. In Prolog, this is only the case for the default operators, and they also can be modified.
2. Operator overloading is dynamic. Besides the case of the unary and binary minus, there are other operators (e.g., `:-`) that are overloaded, and the user can add new ones.
3. As a special case of the previous problem, there is the *comma* that is used as (i) infix operator, (ii) argument separator in compound terms in functional notation, (iii) element separator in lists.

The problems described here are not new, and solutions to them have been proposed in literature.[4] However, their applicability to the syntax of Prolog should be verified.

The *first problem* can be solved in the lexer. An operator declaration can dynamically adapt the operator precedence table, and from then on, the lexer can recognize the newly defined operator and return it to the parser with the proper attributes. This solution is not elegant, as there is a feedback loop between the parser and the lexer, but it turns out to work well in practice because operator declarations cannot occur in expressions that make use of the new operator. Having such a feedback loop is not unusual. Parsers for the C-language also have the *typedefs* analyzed by the lexer.

The *second problem* is also known as the *unary minus problem.* The solution proposed in Reference 4 namely delegating the decision about the real nature of an overloaded operator to the lexer and returning a special token for each case turns out to work in many cases. As mentioned in Reference 4 look ahead is not enough to distinguish between overloaded operators; one also needs the top of the parse stack.

Thanks to the following set of restrictions on the use of operators in Prolog, it is possible to prove that the lexer can indeed always select the right attributes for an overloaded operator.

The first restriction is that an infix operator cannot be overloaded as a postfix operator. Otherwise, one look ahead token is not sufficient as shown in Table II. If the sequence of operators is followed by a term, the top line should be selected, resulting in `t1 yfx (fy fy t2)`. If the sequence of operators is followed by a postfix or infix operator it should be parsed as `(t1 yf yf yf) yfx ...` or `(t1 yf yf yf) yf`.

Table II. Ambiguity when infix and postfix operators are allowed.

$$\texttt{t1} \begin{Bmatrix} yfx \\ yf \end{Bmatrix} \begin{Bmatrix} fy \\ yf \end{Bmatrix} \begin{Bmatrix} fy \\ yf \end{Bmatrix} ??$$

A second restriction is that a prefix operator must not be immediately followed by an open token. This eliminates the ambiguity that occurs in Table III by removing the alternative interpretation. Hence, when a prefix operator is immediately followed by an open token, it is always considered as a functor of a compound term, never as a prefix operator.

* The proof is trivial for Prolog text in functional notation. For operator notation, it can be verified by means of case analysis.

Table III. Ambiguity when a prefix operator is immediately followed by an open token.

term	correct interpretation	alternative interpretation
`-(a,b)`	`a - b`	`-(',' (a,b))`
`- (a,b)`	`-(',' (a,b))`	

The third restriction is that an operator cannot be the immediate argument of another operator without being parenthesized, or stated otherwise, that the operators are not overloaded as nullary operators. This is a rather severe deviation from standard practice. It means that expressions like `+/2` are no longer syntactically correct. Instead, one has to write `(+)/2`.

The algorithm. The algorithm that is used to choose the correct operator attributes is as follows:
Input. The current parse stack, and the current input string starting with an operator.
Output. The unique operator attributes.
Method.

1. if the operator is not overloaded, return its attributes;
2. the operator is overloaded (hence: prefix and (infix or postfix)*), then
   (a) if the look ahead character immediately following the operator is a left bracket, the operator cannot be prefix (this would require a layout character); return the attributes of the non-prefix operator;
   (b) if the operator is the first token of a read term (parse stack empty), or if the top element of the parse stack is a `separator token` (see below), the operator cannot be infix or postfix; return the attributes of the prefix operator;
   (c) if the top element of the parse stack is an operator, reduce the parse stack with respect to the priority of the non-prefix operator. If no reductions could be carried out (top of parse stack is unchanged), the operator cannot be infix or postfix (an operator cannot be the immediate operand of another operator); return the attributes of the prefix operator; otherwise (d).
   (d) if the top element of the parse stack is a term (i.e., neither a separator nor an operator), then return the attributes of the non-prefix operator. □

Table IV. Reductions are needed before deciding.

`t1 xfy` $\left\{\begin{matrix} fx \\ yf \end{matrix}\right\}$	
`t1 xfy t2 yf`	$\left\{\begin{matrix} fx \\ yf \end{matrix}\right\}$

The need for an intermediate reduction in 2.c is motivated in Table IV. In both cases, an operator is followed by an overloaded operator. In the first case no reduction is possible, and the attributes of the prefix operator are chosen. In the second case, at least `t2 yf` reduces to one term, and therefore the postfix operator is chosen. Without an intermediate reduction, the (wrong) prefix operator would have been chosen.

* Let us call this non-prefix for convenience.

In Reference 5 this problem is solved by looking at operator combinations. It turns out that `xfy yf` and `yf fx` are combinations that can never occur. The approach here presented is more efficient because the outcome of the reduction, which has to be performed anyway, is used.

So, this proves that the lexer, given the current top of the parse stack, and given one look ahead character, can choose the right attributes for the operators encountered.

Finally, the *third problem* is somewhat harder to solve. In an operator precedence parser, the separator tokens (`(`, `)`, and so on), are also considered to be operators and interact with the other operators in the language in the usual way. In Prolog, separator tokens (`(`, `,`, `)`, `{`, `}`, `[`, `|`, `]`, `.`) cannot be overloaded as operators except for two cases: dot and comma. From previous sections follows that using the dot as operator causes many problems and should better be discouraged, or even be forbidden in the language definition. For comma, the situation is more severe for it already is a default operator, and its attributes cannot be changed. So, one really has to live with it. Unfortunately, this overloading causes a shift-reduce conflict in the parser.

On the one hand, the precedence for a comma operator followed by another comma operator is $<\cdot$. On the other hand, the precedence for a comma separator in a compound term or a list (such as `f(a,b,c)` or `[a,b,c]`) is $\doteq$, causing a shift/reduce conflict.

The solution that was used to solve the two first problems (asking the lexer to generate two separate tokens, one for the comma used as operator, and another one for the comma used as separator) is much harder in this case because one look ahead character, and the current top of the parse stack do not suffice any more, as indicated in Table V, where a single layout character makes a difference for all the comma characters in the expression. The lexer is not immediately capable of doing this task without a major extension of its functionality.

Table V. The complete parse stack might be needed to find out whether a comma is a separator or an operator.

```
-(a,b,c,d,e)    = '-'(a,b,c,d,e)
- (a,b,c,d,e)   = '-'(',' (a,','(b,','(c,','(d,e)))))
```

In order to solve the problem, a hybrid solution is proposed by making a distinction between the 'real' operators and the separators. The operators are parsed by means of an operator precedence parser, modified as described above to cope with the dynamic declarations of operators and their overloading. Compound terms, lists, (curly) bracketed expressions are treated separately. First comes the operator part.

Operators. An operator precedence table for all possible operators is not difficult to construct and to update incrementally, but its problem is that it might become large (1200 priorities times 2 overloaded operators gives 2400 separate operators, i.e., $5 \cdot 76 \times 10^6$ entries). A well-known technique to reduce the size of the table is to try to construct operator precedence functions f and g. As there are no precedence loops, the two functions can be constructed (see Table VI for the default operators; the values have been computed with the algorithm in Reference 4).

Although this is a standard technique to reduce the size of the operator precedence table, it has two disadvantages. (i) The precedence functions do not take the error entries into account. For the non associative operator =, they return resp. 7 and 6, suggesting that = would be left associative. Two identical values would have been a better choice. (ii) The

Table VI. Operator table.

Operator	assoc	pri	f	g
`:- -->`	xfx	1200	0	0
`:- ?-`	fx	1200	0	0
`;`	xfy	1100	1	2
`->`	xfy	1050	3	4
`,`	xfy	1000	5	6
`= \=`	xfx	700	7	6
`== \== @< @=< @> @>=`	xfx	700	7	6
`=..`	xfx	700	7	6
`is =:= =\= < =< > >=`	xfx	700	7	6
`+ - /\ \/`	yfx	500	9	8
`* / // rem mod << >>`	yfx	400	11	10
`**`	xfx	200	11	12
`^`	xfy	200	11	12
`- \`	fy	200	11	12
`@`	xfx	100	13	12
`:`	xfx	50	13	14

construction of the precedence functions is not difficult, but is not easily done incrementally. Adding an operator might change many function values.

This means that operator precedence functions are probably not the best choice for the dynamic grammar of Prolog. It turns out that, as the priority and the associativity of the Prolog operators are given, it is straightforward to directly compute the operator precedence table entries, taking into account the error entries.*

While computing the precedence relation, starting from the priorities and the associativity, one hits another ambiguity. The term `fy t yf` can be parsed either as `(fy t) yf` or as `fy (t yf)`. This ambiguity is solved by the restriction that a left associative operator binds slightly more strongly than a right associative operator of the same precedence. This restriction is expressed formally in the standard by defining two types of terms: terms `term` that can appear everywhere and terms `lterm` that can appear only as arguments of left associative operators. It invalidates the alternative interpretation of Figure VII (see also the operator precedence Table VIII).

Table VII. Same precedence operator associativity.

Term	Correct interpretation	Alternative interpretation
`fy t yf`	`fy (t yf)`	`(fy t) yf`
`t1 xfy t2 yfx t3`	`t1 xfy (t2 yfx t3)`	`(t1 xfy t2) yfx t3`
`t1 xfy t2 yf`	`t1 xfy (t2 yf)`	`(t1 xfy t2) yf`
`fy t1 yfx t2`	`fy (t1 yfx t2)`	`(fy t1) yfx t2`

In Table VIII, the operator precedences for three priority levels ($P-1$, P, $P+1$) with respect to P are shown (the full table contains $(7 \times 1200)^2$ entries). This table can be significantly simplified by replacing the error entries with $<\cdot$ and $\cdot>$ entries in such a way

* Remark the important difference between *priority* (Prolog terminology) and *precedence* (parser terminology). Unfortunately, the higher the priority, the lower the precedence.

Table VIII. Operator precedences for operators.

		P	*P*	*P*	*P*	*P*	*P*	*P*
		xfx	xfy	yfx	fx	fy	xf	yf
$P+1$	xfx	<·	<·	<·	<·	<·	<·	<·
$P+1$	xfy	<·	<·	<·	<·	<·	<·	<·
$P+1$	yfx	<·	<·	<·	<·	<·	<·	<·
$P+1$	fx	<·	<·	<·	<·	<·	<·	<·
$P+1$	fy	<·	<·	<·	<·	<·	<·	<·
$P+1$	xf	<·	<·	<·			<·	<·
$P+1$	yf	<·	<·	<·			<·	<·
P	xfx			·>				·>
P	xfy	<·	<·	<·	<·	<·	<·	<·
P	yfx			·>				·>
P	fx			·>				·>
P	fy	<·	<·	<·	<·	<·	<·	<·
P	xf			·>				·>
P	yf			·>				·>
$P-1$	xfx	·>	·>	·>	·>	·>	·>	·>
$P-1$	xfy	·>	·>	·>	·>	·>	·>	·>
$P-1$	yfx	·>	·>	·>	·>	·>	·>	·>
$P-1$	fx	·>	·>	·>	·>	·>	·>	·>
$P-1$	fy	·>	·>	·>	·>	·>	·>	·>
$P-1$	xf	·>	·>	·>			·>	·>
$P-1$	yf	·>	·>	·>			·>	·>

that, for a given priority, the effect of the second operator (horizontal axis) is factored out. This gives a simpler relation (shown in Table IX). The fact that the error entries have been

Table IX. Improved precedence relation.

```
reduce(P1,xfx,P2) :- P1 =< P2.
reduce(P1,xfy,P2) :- P1 < P2.
reduce(P1,yfx,P2) :- P1 =< P2.
reduce(P1, fx,P2) :- P1 =< P2.
reduce(P1, fy,P2) :- P1 < P2.
reduce(P1,xf ,P2) :- P1 =< P2.
reduce(P1,yf ,P2) :- P1 =< P2.
```

removed is not that severe. For a syntactically correct program, they never occur, and for an erroneous program, the reductions will eventually detect the error.

First of all, the error entries that were replaced by $<\cdot$ cannot do any harm because nothing is reduced. Eventually the accept state is never reached because there is at least one production application missing.

The error entries that were replaced by $\cdot>$ need more attention because in this case a reduction might be caused. This reduction must in turn verify whether the other conditions are fulfilled (e.g., whether the argument priorities do not exceed the operator priority). The mistake is also detected quite early in this case.

Separators. The treatment of the separator tokens is completely different. First of all, the opening brackets (`(`, `[`, `{`) are given a precedence that is lower than any other operator. This ensures that this bracket cannot be reduced by any operator in the language. As a matter of fact, it behaves like the *begin token* \$ and as such it creates a new kind of stack frame on top of the current stack. Now, the processing of the input symbols simply continues. Hereby the comma is treated as the right associative operator. Thanks to a restriction on the kind of terms that can appear as arguments of compound terms and of lists, an operator with a priority higher than (i.e., precedence lower than) that of a comma cannot occur, unless parenthesized. Hence, the comma can never be reduced when used as a separator of arguments or list elements. All the other operators reduce as usual.

The head tail separator token `|` is given the same precedence as the comma operator.

When shifting the final bracket, the last argument is first reduced properly by giving the closing bracket the same precedence as the comma. Then the stack is scanned for a series of expressions alternated with comma operators until the opening bracket is found. If the token below the opening token is an atom (without layout text between the atom and the opening token) that is not an infix operator (unless preceded by a separator token or another operator), then the current top of stack elements is reduced to a compound term.

If it is not possible to reduce to a compound term, pop the closing bracket off the stack, and shift it again on the stack, now with the same precedence as the opening bracket. Now, the comma operators reduce up to one single term. This constitutes the bracketed expression.

The shift/reduce conflict just explained is thus solved dynamically by this parser thanks to: (i) the restriction that arguments and list elements cannot have a priority higher than that of the comma so that the comma stays unreduced on the stack; (ii) to the restriction that there can be no layout text between the functor of a compound term and its opening bracket; and (iii) to the use of the opening bracket at the lowest precedence, creating a new stack frame.

If one of the arguments would have a precedence lower than comma, this operator stays unreduced on the stack. Since this operator prevents the reduction to a compound term, the stack is further reduced to a bracketed expression. Finally, this gives rise to a functor atom followed by a bracketed expression which is a syntax violation in Prolog, raising a syntax error.

For the curly bracketed expressions, the closing bracket is given the same precedence as the opening bracket. Hence, the enclosed expression reduces to one term that constitutes the argument of the curly bracketed expression.

Output routine

The method used to display Prolog text is as follows: a recursive procedure `writeterm` is used. It has three arguments: the term to be displayed, and the maximum priority it is allowed to have as term, and as atom.* If the term or atom to be displayed has a priority that is higher than or equal to the maximum allowable priority, it must be displayed between parentheses. Terms and atoms must be treated separately because operators used as atoms behave differently when used as operator arguments than terms do (see the restriction on operators used are arguments of other operators). Moreover, an exception is made for the comma that is always displayed as a quoted token except when used as an infix operator.

* Actually, it is the maximum priority plus one.

Furthermore, when operator notation is in effect, there are two more exceptions.

1. If the arity of the compound term is equal to one, and if the functor is {}, then the term should be displayed as a curly bracketed expression. There is no limitation on the priority of the terms or atoms occurring in a curly bracketed expression.
2. If the arity of the compound term is equal to two, and if the functor is dot, then the term should be displayed in list notation. Atoms can have any priority, the term priority should be lower than that of the comma.

IMPLEMENTATION

The algorithms described in this paper have been implemented and tested. The implementation has been done in C mainly for two reasons.

1. *Efficiency.* Lexical analysis, and the creation of Prolog terms starting from bits and bytes is not something which Prolog is particularly good at. There is a lot of overhead and unneeded memory consumption involved with it. A lower level imperative language is better suited to do this task, and hence more efficient. On the other hand, as Prolog implementations are becoming better and better, the performance gap will get smaller in the future.
2. *Easy integration.* Writing a parser in Prolog itself requires the system be bootstrapped which is not an easy task. The parser is written in a modular way such that it can be integrated in most systems (at least those written in C). It further integrates very well with Prolog compilers that use C as the target language.[8] All system dependent parts have been merged into one module. The other parts are quite universal and strongly reflect the structure of the ISO document.

In this section, the performance of the lexer + parser and display routine are compared with the public domain parser and write routine written by O'Keefe, used in BinProlog and several other Prolog systems.

The benchmark environment is BinProlog, a public domain Prolog developed by Paul Tarau.[6] Although BinProlog is certainly not the fastest Prolog available at this moment, we have used it to compare both parsers because BinProlog is currently the only system where the parser has been completely integrated, and was therefore readily available to run the benchmarks. At the end of this section, we compare the performance of BinProlog with that of a state of the art native code compiler for Prolog. All benchmarks have been run on a SUN work station Sparc center 1000, running `nrev` at 670 KLIPS and on a HP-PA RISC running `nrev` at 400 KLIPS.

The Prolog text used to read and write in this benchmark is the concatenation of all the Prolog text in the system files of the BinProlog 2.26 system (compiler, O'Keefe's read and write routine, DCGs, etc). It contains a wide variety of constructs. O'Keefe's parser contains many special cases described in the standard. The total file contains 1100 clauses and has a size of 97512 bytes.

Tables X and XI contain some remarkable data. The input routine (lexer and parser) is about a factor 10 to 20 faster than O'Keefe's parser in BinProlog. The output routine is about a factor 5 to 10 faster than O'Keefe's write in BinProlog. It is quite remarkable that the speedups are about twice as high on the HP-PA than on the Sparc. BinProlog appears to be slower on HP than on SUN where it was originally developed.

Table X. Performance comparison on SUN Sparc Center 1000

Benchmark	C (gcc -O2) ms	BinProlog ms	BinProlog/C
input (free format)	290	3040	10·48
input (operator notation)	267	3827	14·33
input (functional notation)	350	4205	12·01
output (operator notation)	300	1650	5·50
output (functional notation)	294	1522	5·17
compiler	5440	7820	1·44

Another point worth mentioning is that it is quite surprising that C-based output takes about the same time as C-based input. Although both routines perform similar tasks (scanning terms and atoms, accessing the symbol table, fetching operator attributes, etc), displaying a term is still less complex than parsing a term.

Besides the facts that the output routine needs dereferencing, and is fully recursive, about 50 per cent of the execution time on the HP-PA, and about 33 per cent for execution time on the Sparc is spent in the processing of the list of write options (read does not have options in this benchmark). Taking all this into account, the write is about 33 to 50 per cent faster than the read. In the Prolog-based implementation, this is roughly a factor of two to three.

The output in functional notation is only slightly more efficient than in operator notation. Hence, retrieving the operator precedences, and inserting layout text when needed are not major costs when displaying terms in operator notation.

One last remark about these performance data is that it is rather counterintuitive to observe that it is harder to parse Prolog text in functional notation than in operator notation. There is however a simple explanation for this effect. Since the parser does not know beforehand that a read term is in functional notation, it will treat it as a read term in operator notation. In `p :- a,b.` it has to classify two possible operators and to perform two reductions. In `:-(p,',')(a,b)).` it has to classify *four* possible operators, and it has to resolve twice the ambiguity between a compound term and bracketed term. Hence, the functional notation requires more processing from the parser. If the parser could know in advance the kind of notation of a term (functional or operator-based), parsing could be accelerated a lot in the functional case.

Table XI. Performance comparison on HP PA-RISC

Benchmark	C (cc -O) ms	BinProlog ms	BinProlog/C
input (free format)	650	11780	18·12
input (operator notation)	700	14940	21·12
input (functional notation)	820	16250	19·81
output (operator notation)	660	6940	10·51
output (functional notation)	640	5760	9·00
compiler	20410	31010	1·52

Input/output is only a part of the total execution time of a program. In order to have an idea of the global impact of the C-based IO on an application, the BinProlog compiler was run on the Prolog text used in this benchmark set. The BinProlog compiler is about 50 per cent faster when using the C-based IO. This indicates that the use of it really pays off. The time spent in doing input drops from 35 per cent down to less than 5 per cent.

In order to obtain a precise idea about where most of the execution time is spent while reading free format Prolog text, the execution times for the global benchmarks are detailed in Table XII.

Table XII. Lexer versus Parser

	C (gcc -O2)	BinProlog	BinProlog/C
	ms	ms	
Lexer	148 (51%)	2027 (67%)	13·7
Parser	142 (49%)	1013 (33%)	7·1
Total	290	3040	10·5

It turns out that the lexer (basic IO included) is slightly more efficient in the C-based parser than it is in BinProlog. The basic IO in the C-based lexer is only 26 ms (i.e., 17 per cent of the time needed to do the full lexical analysis).

As BinProlog is not the fastest possible Prolog implementation, it should also be compared with a state of the art implementation. Therefore, the same benchmark was run under Aquarius Prolog (apc) and on Sicstus Prolog (fastcode). The results are shown in Table XIII. It turns out that O'Keefe's parser is again about 10 times slower, but that the Aquarius read is only 50 per cent slower. BinProlog, as well as Aquarius and Sicstus Prolog use a version of O'Keefe's public domain parser, where the IO routines were changed for better performance. As a reference, the original version of this parser was compiled with apc and the execution time is also added.

Table XIII. Performance comparison on SUN

Benchmark	C (gcc -O2)	BinProlog	Aquarius	Sicstus	O'Keefe/apc
	ms	ms	ms	ms	ms
input (free format)	290	3040	545	2055	3600

Hence, by improving the code for the basic IO and tokenizers, the execution speed of the public domain parser can be improved considerably. Even then, a C-based parser stays about 100 per cent ahead.

The C-implementation described in this paper is publicly available by anonymous ftp from `ftp.elis.rug.ac.be`, directory `pub/prolog/prolog.parser.tar.Z`.

RELATED WORK

This is clearly not the first attempt to write a parser for a grammar with dynamic operator overloading. Besides more general attempts to parse ambiguous grammars,[9] dynamic grammars,[10,11] and general context free grammars,[12,13] there are two attempts for Prolog that are worth discussing.

O'Keefe's public domain parser is a left corner parser. It starts by tokenizing a complete read term, and then parses the token list in a predictive way. Since this parser is written in Prolog, ambiguities are easily solved by backtracking. The only cost incurred is some extra execution time. This parser works fine, but is less efficient as shown in the previous section. Furthermore, as follows from Table XIV, the extra heap consumption for reading the file used for benchmarking is quite impressive. The C-based parser consumes about 2.5 times the size of the file, where the Prolog-based parser consumes about 45 times the size of the file.

Table XIV. Memory consumption

C (gcc -O2)	BinProlog	BinProlog/C
243 Kb	4510 Kb	18·53

C Prolog has had a fast parser for Prolog for more than ten years. However, it parses a language that is more general than the proposed standard for Prolog. As the language is more regular, with less restrictions, there are fewer cases to distinguish, and as a consequence, the resulting parser is about twice as fast.

A recent and important attempt is the so-called *deferred decision parsing*.[5] This is an extension of LR parsing where the final decision of how to parse an expression (i.e., the shift reduce conflicts) is not made at parser construction time, but at parse time. All possible actions (shift/reduce) for an overloaded operator are collected in a set. If the set is empty, an error is raised, if it contains both shift and reduce, the input is ambiguous. In all other cases, there is a unique action to be performed.

This approach works fine for operator expressions but is far more general than that needed in order to parse Prolog, given the restrictions of the ISO standard (there should never be an ambiguous construct).

However, the major criticism is that fact that the real hard problems like the overloading of the comma (operator and separator), and the possible overloading of the dot is not treated at all in Reference 5. Furthermore, the restrictions that are proposed to produce a deterministic induced grammar (e.g., all declarations for the same symbol have the same precedence) are not applicable for Prolog (e.g., the unary and binary minus).

CONCLUSION

This paper presents a modified operator precedence parser for the grammar described in the ISO standard for the Prolog language. The paper also shows why most of the restrictions that are imposed by the standard are needed. It also contains some suggestions that could make the syntax description of the standard even more precise.

The resulting parser in C is as easy to understand as any comparable parser in Prolog, but it is about one order of magnitude faster. So, any application that does some input/output benefits from it. It even outperforms Aquarius Prolog by 50 per cent.

It is the first efficient public domain parser for *Standard Prolog* that actually works and deals with all the details of the language.

ACKNOWLEDGEMENTS

Koen De Bosschere is senior research assistant with the Belgian National Fund for Scientific Research. The author would like to thank J. Van Campenhout for support and advice, S. Debray and P. Tarau for remarks on previous versions of this paper, and D. Bowen, B. Demoen, P. Deransart, R. O'Keefe and R. Scowen for the inspiring discussions on the syntax of Prolog. He is also indebted to the referees for pointing out some weaknesses in the benchmarking methodology of the original manuscript.

REFERENCES

1. A. D. McGettrick, *Algol68: a First and Second Course, Volume 8, Cambridge Science Texts*, Cambridge University Press, Cambridge, London.
2. W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer Verlag, Berlin, 1981.
3. M. Tofte, R. Milner and R. Harper, *The Definition of Standard ML*, MIT Press, 1990.
4. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, MA, 1977.
5. K. Post, A. Van Gelder and J. Kerr, 'Deterministic parsing of languages with dynamic operators', In *Proceedings of the 1993 ILPS Conference*, Vancouver, Canada, 1993. MIT Press, pp. 456–472.
6. P. Tarau, 'BinProlog 3.30 user guide', *Technical Report 95-1*, Département d'Informatique, Université de Moncton, February 1995. Available by ftp from *clement.info.umoncton.ca.*
7. R. S. Scowen, 'Draft prolog standard', *Technical Report ISO/IEC JTC1 SC22 WG17 N110*, International Organization for Standardization, 1993.
8. K. De Bosschere and P. Tarau, 'High performance continuation passing style Prolog-to-C mapping', In E. Deaton, D. Oppenheim, J. Urban and H. Berghel, eds, *Proceedings of the 1994 ACM Symposium on Applied Computing*, Phoenix/AZ, March 1994. ACM Press, pp. 383–387.
9. A. V. Aho and S. C. Johnson, 'Deterministic parsing of ambiguous grammars', *Communications of the ACM*, **18** (8), 441–452 (1975).
10. J. Heering, P. Klint and J. Rekers, 'Incremental generation of parsers', *IEEE Transactions of Software Engineering*, **16** (12), 1344–1351 (1990).
11. S. L. Peyton Jones, 'Parsing distfix operators', *Communications of the ACM*, **29** (2), 118–122 (1986).
12. J. Earley, 'An efficient context-free parsing algorithm', *Communications of the ACM*, **13** (2), (1970).
13. M. Tomita, *Efficient Parsing for Natural Language*, Kluwer Academic Publishers, Boston, 1986.