GP4 - A Generic Prolog Parsing and Prototyping Package

Graham G. Thomason

Report Relating to the Thesis "The Design and Construction of a State Machine System that Handles Nondeterminism"

Department of Computing
School of Electronics and Physical Sciences
University of Surrey
Guildford, Surrey GU2 7XH, UK

July 2004

GP4 - A Generic Prolog Parsing and Prototyping Package

GP4 is a package that facilitates the translation between, and prototype implementation of, domain-specific languages for **automatic test case generation**. It is equally applicable to domains other than testing. Although the main reason for developing it was for STATECRUNCHER, a program that provides a test oracle for state based testing, the package is independent of STATECRUNCHER, and is described mainly without further reference to it.

Although Unix tools such as *Yacc* and *Lex* could have been used for parsing, GP4 uses PROLOG only for its implementation. This is because PROLOG is extremely well-suited to implementing the run-time part of the languages to be developed, and we wish to avoid tool diversity.

GP4 is not a testing tool in itself. But it is an underlying part of a testing tool which itself is only part of a tool chain. The tool chain will typically contain commercial tools as well.

Contents

1. Introduction

GP4 provides a generic framework in which language elements such as operators and statements can be defined. It supports parsing from source code in that language to a Prolog-readable nested list structure. Certain generic aspects of run-time support for program execution (e.g. expression evaluation) are also supported. The following figure illustrates the framework:

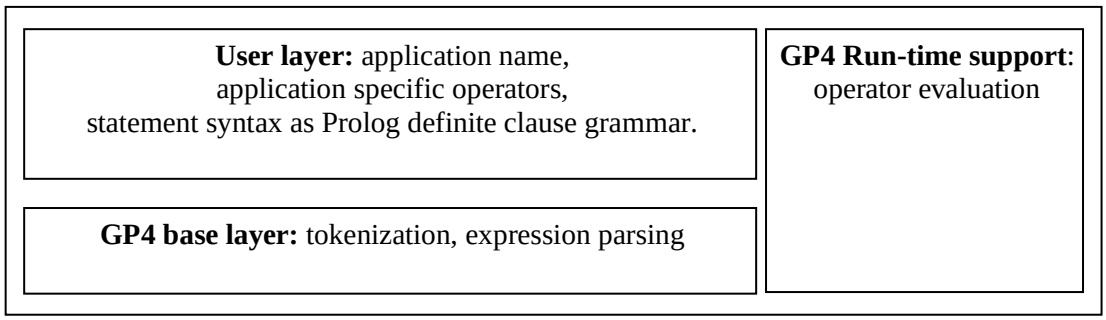


Figure 1. GP4 framework

GP4 does *not* support the semantic side to compilation. It does not distinguish between declarations and executable statements and it does not construct symbol tables. This must be done separately. The emphasis is on syntax-driven parsing.

One unusual (but not unique) feature of GP4 adds to its power as a prototype compiler tool. Operator sequences are not tokenized during lexical analysis. Instead, they are parsed as part of the operator/expression grammar. This makes it possible to use several *different* operator sets in different contexts in the same source language. This is applicable where portions of C-style syntax may be embedded in a non-C-conformant domain-specific language.

GP4 is implemented in PROLOG. Since PROLOG is a good choice of language for the end *application execution* such as test case generation, and as it supports Definite Clause Grammars (DCG's), it is convenient to use it for *parsing* as well. A large part of GP4 is simply a collection of statements in DCG notation. The use of one language for parsing and execution helps ensure consistency between these phases, and reduces overall tool diversity.

GP4 is intended to be a practical means to a practical end: tools in a tool-set for automated test generation.

The techniques used in GP4 are not claimed to be original; they are a practical means to help implement other tools that may contain original material. However, the techniques for expression parsing were developed from first principles, after the basics of the PROLOG

Definite Clause Grammar had been learned, mainly from [Clocksin]. The feed-forward expression grammar result may be of interest to those working in the field of compiler technology.

The techniques described here were first used by the author in a simpler form for an expert system shell called DEXIOS, reference [Dexios].

2. GP4 Architecture

The GP4 system provides a generic parsing layer to enable a domain-specific language to be parsed with a minimum of overhead. The emphasis is on generality. In principle, all the user need do is define

- the higher level part of the syntax of the language (comparable with the statement level in an imperative language)
- a set of operators to be used wherever an *expression* occurs.

The output of a parse is a set of Prolog-readable nested list structures. However, the output could easily serve as input to other languages such as Perl and TCL. The kind of engine that is envisaged could be

- A state-machine engine or translator
- A cause-effect graph test case generator
- A probabilistic network and inference engine

An application may require the use of a validator and data generator to supplement this, e.g. to generate a symbol table and cross-reference, and to generate predicates representing data variables. A validator / data generator is rather application specific, and is outside the scope of GP4.

Figure 2 shows data flow of an application with a division between the compile-time modules, run-time modules, and general support routines.

Figure 3 shows the compile-time modules in a layered architecture. Note that all the user supplies are operator definitions, syntax rules, and application-specific data and texts. The non-generic example names use the affix “sm” (state machine application). Even the operator definitions are likely to be highly reusable, since basic arithmetic and relational operators are common to many application areas.

Figure 4 shows the additional modules for run-time support. The “highly reusable” operators supported by the compiler are supported by “highly reusable” evaluation routines for these operators. In addition a number of useful functions (such as maximum, minimum) are supported.

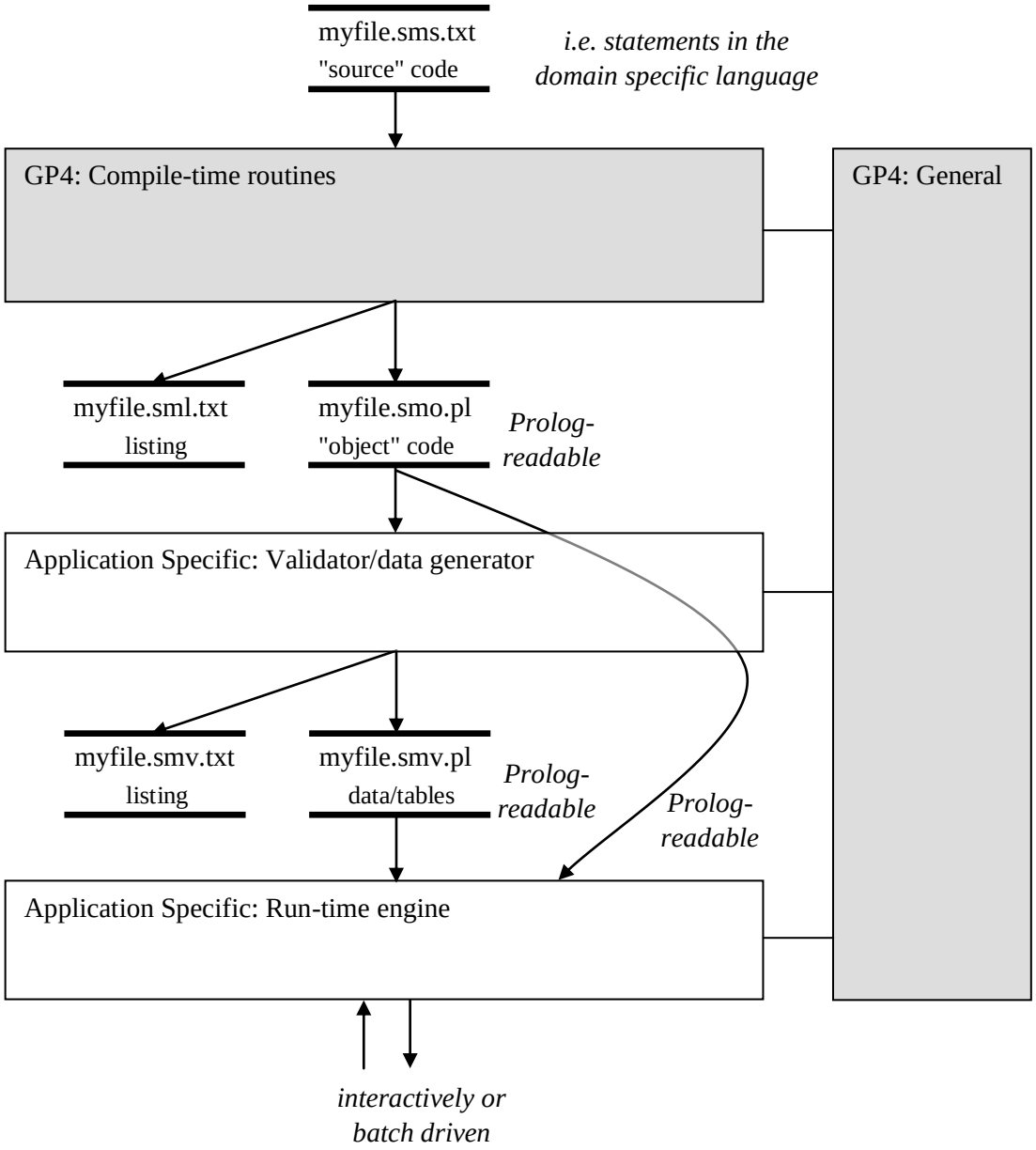


Figure 2. Data Flow of a GP4-based Application in Use

GP4 components shown shaded.

For examples of *source*, *object* and *listing* files, refer to section 8.

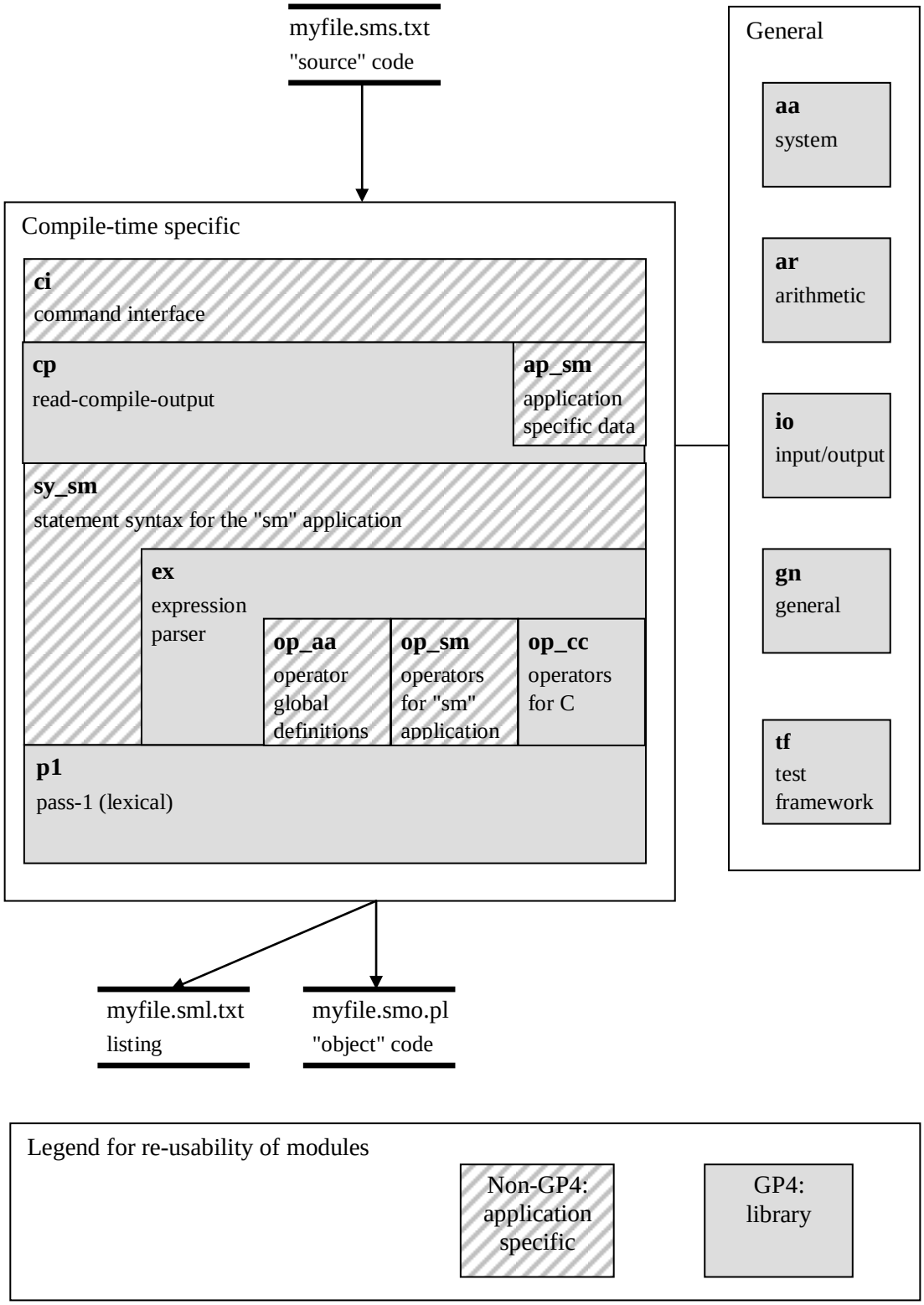


Figure 3. Compile Time Compositional Architecture

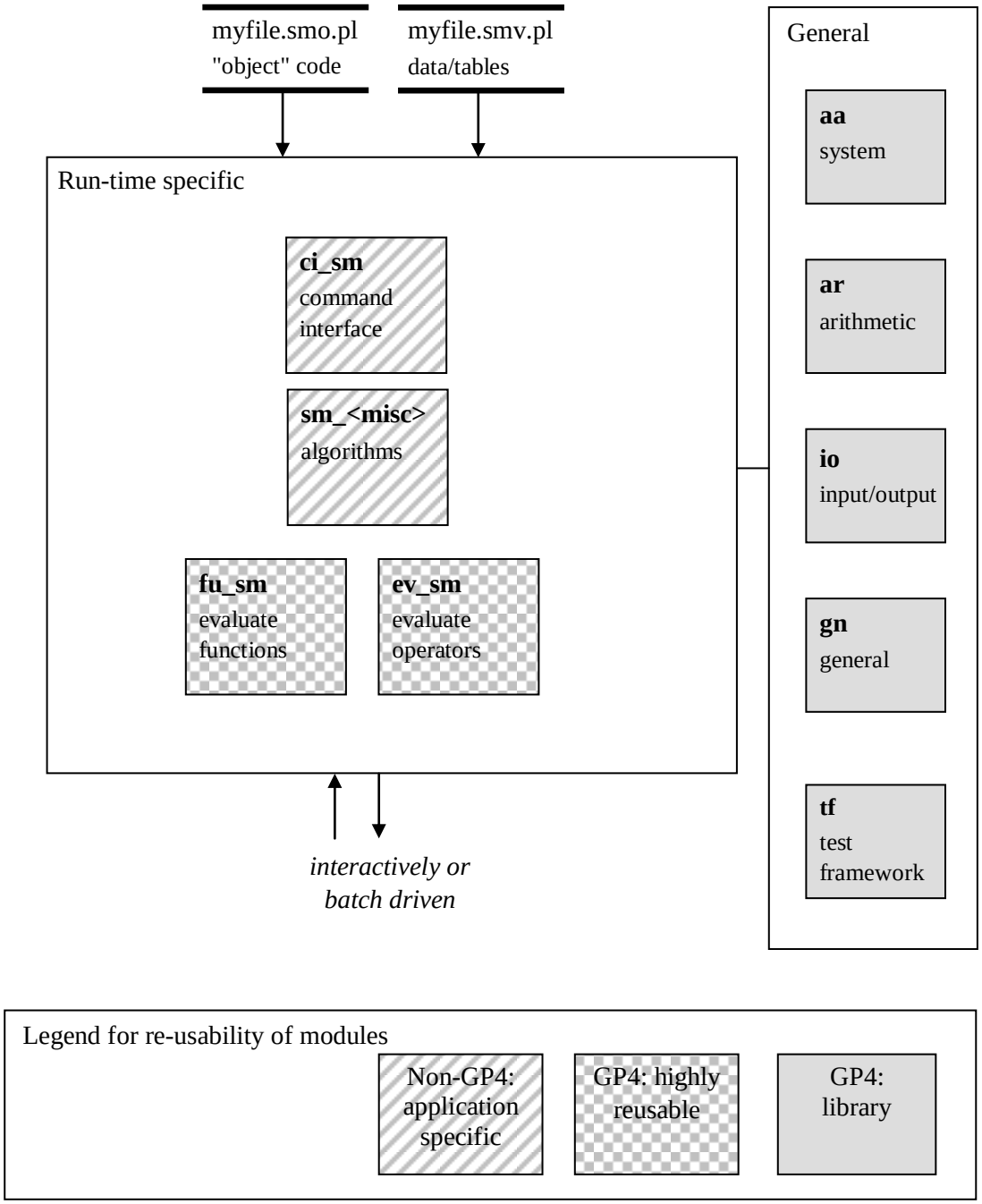


Figure 4. Run-time specific

3. Pass 1 parsing

3.1 Pass 1 overview

The term "pass 1" in GP4 is the first pass and transformation of the input string to be parsed; it rather similar to the lexical analysis phase of a conventional compiler. As such, it is at a lower level than the first pass of many complete two-pass compilers where the term *pass 1* is used of much more than lexically analysing input, where it includes the construction of a symbol table. The second pass of such compilers generates object code using known values of forwardly-referenced symbols.

The task of GP4 pass 1 parsing is to identify certain lexical items in the input source code. Source code is read in line by line, and may be offered to the parser in chunks (provided suitable delimiters between the chunks can be identified), or as the entire contents of a file.

The input to pass 1 is a list of ASCII codes, as obtained when reading an ASCII file. A non-list input item will be returned as the output. Non-ASCII list elements will be returned in the output list. This being the case, no error messages are produced in pass 1.

The output of pass 1 is a list containing certain lexical items such as identifiers and constants, and unaffected ASCII codes.

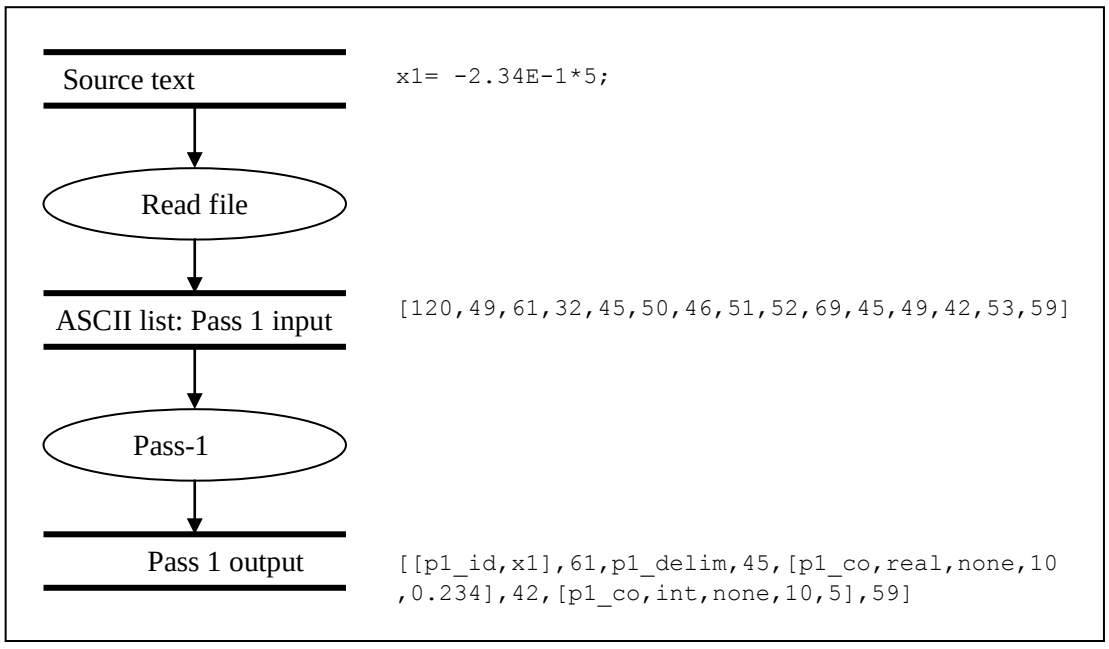


Figure 5. Pass-1 processing overview

3.2 Rationale concerning pass-1 processing

Pass-1 processing here is a lexical analysis phase. Although it would be possible to parse a language in one pass using grammar rules that span symbols from statements and expressions to alphanumeric terminals, there are advantages to separating this process into two phases:

- Performance: it guarantees that processing that is only required once, e.g. examining a substring to see if it contains an identifier, only takes place once (or at worst, as many times as pass-1 does this processing).
- Modularity: there is a natural layered relationship between pass-1 processing and pass-2 processing. Pass-1 reduces the lexical space in which pass-2 works, so that each process is considerably simpler than the combination.
- Testability: pass-1 and pass-2 can be tested separately with fewer tests than would be the case without the separation. This is due to the reduced lexical space in which pass-2 works.

A major issue is: what should pass-1 do, and what should it leave untouched? We address some questions here and show what choices have been made in the answers.

Q 1: What should Pass-1 definitely do?

A:

- Identify and package as output tokens **identifiers** (but whether to distinguish language keywords is an issue). We do, however, need to assume that identifiers are of the C/Java kind - see the question relating to this below.
- Identify and package **numerical constants** (but whether to evaluate them is an issue).
- Identify and package **strings.**
- Identify and simplify **white space and comments** (but whether to remove them completely or replace sequences of them by a single token is an issue).

Q 2: Should Pass-1 assume C conventions in the source language?

A: Yes in many respects, (but not for operators). Otherwise, pass-1 is too weak and has very little to do. C conventions cover C++, Java, and Unix tools generally. The prototype languages that are envisaged have much commonality with C. If non-C conventions are required, (e.g. not case sensitive and different case mixings will be used in identifiers), we should use a variant pass-1 module. In our standard pass-1 routine, we observe

- C identifier conventions
- C constant representations (chars, octals, hexadecimals, numerical suffixes)
- C escape sequences (\n \r etc in chars and strings)
- C and C++ style comments (/*...*/, //-----)

Q 3: Should language keywords, function names and keyword operators be distinguished?

A: No. It would spoil the low degree of coupling of pass-1 to other modules. It is acceptable to package all identifiers in the same way, even if they are really operators. Pass-2 should be responsible for distinguishing.

Q 4: Should an attempt be made to identify operators in pass 1?

A: No. The rule may be that from an operator sequence, the longest match will yield the operator (e.g. in C, −> would always be a single operator), but we should not assume this in pass-1. We should give pass-2 the freedom to pick-and-choose operator sets per expression context, so pass-1 would need more information than we want to give if it is it to single out operators. An identifier-like-item might be a keyword operator in one place, but an ordinary identifier in another place. This feature is not unique, but it is not common in conventional parsers.

A disadvantage of this choice is that it introduces a little backtracking in pass 2 in determining what actual operators are present.

An even more flexible way to achieve "dynamic syntax" would have been for pass 2 to request pass 1 tokens in a parameterised way one at a time as needed. This approach would have been an interesting alternative, and may have been a better choice, but it is not known whether it would be easy to accommodate it to Prolog Definite Clause grammars. The current approach is adequate for the envisaged applications. A hybrid approach is also conceivable where an additional pass is made to convert from very primitive tokens to 'operatorised' tokens. In either case, there is the advantage that if a longest-sequence-match strategy can be employed, then pass 2 parsing can be done without backtracking.

Q 5: Should numerical constants be evaluated in pass 1?

A: Yes - at least for a prototype system. For a Prolog-based implementation, we are restricted to Prolog's own representation of numerical values anyway (unless we are prepared to write our own floating-point package and mathematical library, or link to an external one, such as the Gnu library *gmp*). There is nothing to be gained by postponing the conversion to the numerical value until after pass 1. This may mean that long integers or long doubles cannot be represented. But if the Prolog implementation does not support them, then a difficult work-around will be needed anyway. If possible, our language should avoid them. The only compensation that we do offer is that suffix information (long, unsigned etc.) is retained in the output token.

Q 6: Should we identify negative constants?

A: No. We do not wish to state with certainty that a minus sign before a constant really does apply to the constant. The user may define other operators ending in a minus sign. We leave it to Pass-2 to supply a monadic operator. Similarly the monadic plus operator is left to pass-2.

Q 7: Should white space be removed entirely in pass-1?

A: No. White-space[1] sequences, including comment sequences, should be reduced to a single white-space output token. Although this means that pass 2 will have to absorb white-space in various places, the white-space information is needed to distinguish between e.g. in C:

`c++1` the expression c++ followed by the integer 1 (never legal C).

and

`c+ +1` the sum of c and +1 (legal C expression).

We note that a C-only parser need not retain white space in the tokenization because it has already committed to the tokens at the lexical analysis stage.

Note that if the method of requesting parameterised pass 1 tokens had been used (see Q 4:) then no white space would need to be kept after pass 1.

If GP4 had been designed just imposing the following restriction on the sets of operators that make up any one application:

> *the sets of operators in use must admit of a unique tokenization scheme (not one for one set and another for another set)*

then non-parameterised tokenization of operators could have take place in pass 1 and white space could have been removed entirely in pass 1. The tokenization scheme would typically be*: longest match wins*. This restriction would not imply that the different operator sets in use would have to share precedences, associativities etc. for the various operators; it only affects the longest-first tokenization rule across all operators.

The above restriction would have simplified life in pass 2 considerably. But the choice was made to retain maximum flexibility in operator definitions (within the scope of PROLOG Definite Clause Grammars); hence the need to retain white space in pass 1.

[1] As a legacy issue, in the GP4 code of version 1.0, and in the related figures, but not in the main text of this paper, *white space* is rather inaccurately referred to as a *delimiter*. The term *delimiter* would have been better reserved for delimitation by parentheses, and *white space* for spaces and tabs, and can include comments. For the purposes of this report, *delimiters* and *white space* mean the same thing.

3.3 The pass-1 call

The call is

```
p1_p(OUTPUT,INPUT,[]).
```

`INPUT`: A list of ASCII values.

`OUTPUT`: A list of output tokens.

Note: this call is of the standard format to invoke goals involving definite clause grammar rules - see [Clocksin, p.225]. The empty list as a third parameter specifies that we require no rest-string of unparsable items.

Example:

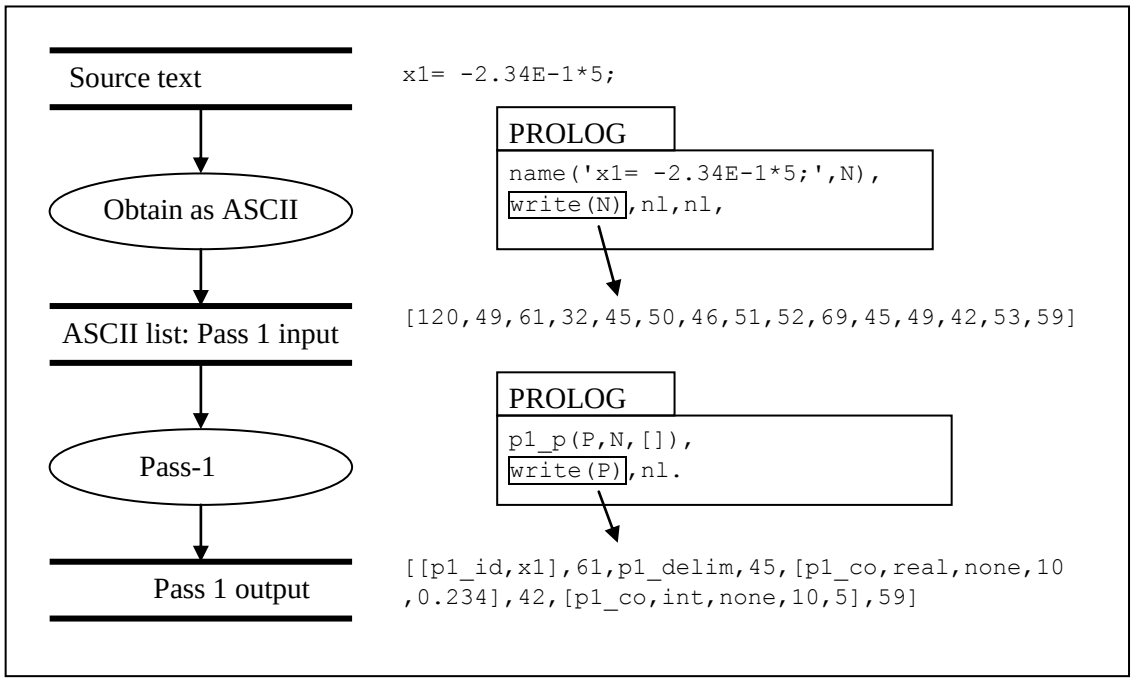


Figure 6. Pass-1 call example

3.4 Pass-1 output tokens

The following tables show the output tokens produced.

Token	Produced from	Explanation of attributes
[p1_id,*IDENTIFIER*]	an identifier	
[p1_co,char,n,n,*VALUE*]	a char (as in 'C')	*VALUE* is the ASCII value of the char
[p1_co,int,*QUALIFIER*,*BASE*, *VALUE*]	an integer	*QUALIFIER*=integer suffix representation as in C: none, u, ul or l (irrespective of case/order in the source definition) *BASE*=original base used *VALUE*=integer value
[p1_co,real,*QUALIFIER*,10, *VALUE*]	a real	*QUALIFIER*=real suffix representation as in C: none, l or f (irrespective of case in the source definition) *VALUE*=real value
[p1_str,*LIST*]	a string, enclosed in double quotes	*LIST* contains ASCII values.
p1_delim	white space and/or comments	only one token produced per sequence of white space/comment combinations
Unaffected-ASCII-Code	all other input	

Table 1. Pass-1 outout tokens

Examples:

INPUT item (as ASCII string)	OUTPUT token(s) - element(s) of output list
tweedledum	[p1_id,tweedledum]
'B'	[p1_co,char,n,n,66]
23LU	[p1_co,int,ul,10,23]
2.34E-1	[p1_co,real,none,10,0.234]
"ABC"	[p1_str,[65,66,67]]
+	43
a 6	[p1_id,a],p1_delim,[p1_co,int,none,10,6]
/* comment */	p1_delim

Table 2. Examples of pass-1 tokens

3.5 Pass-1 grammar

Pass-1 is performed by applying grammar rules. The following diagrams show these rules in a form that is close to the actual implementation.

The syntax diagrams are backtrackable and imply *sequence of attempt* at parsing.

Consider the following figure:

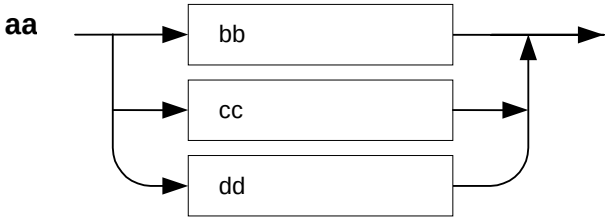
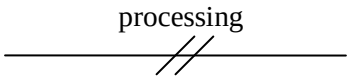


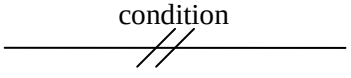
Figure 7. Grammar rule example

It can be read as follows: A syntactic item "aa" is preferably a syntactic item "bb" – if this can be satisfied by the input string, accept it. If "bb" cannot be satisfied, attempt to satisfy "cc". A third choice is "dd".

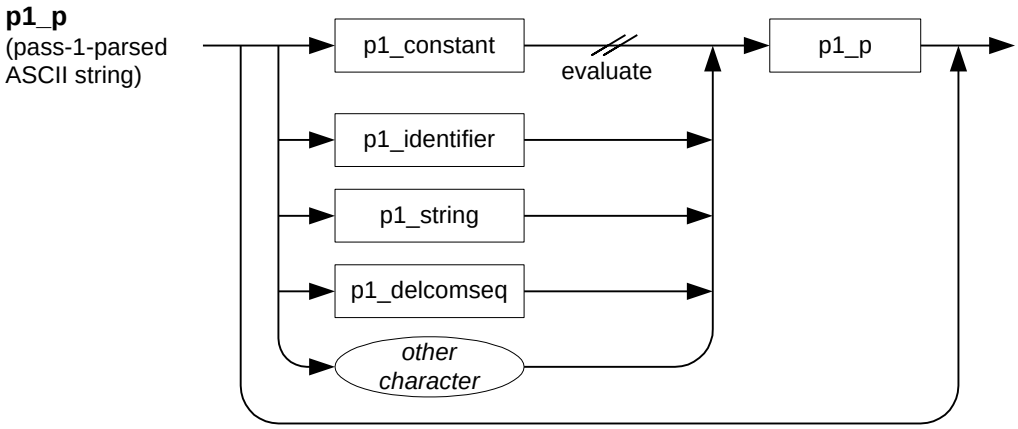
The diagrams occasionally deviate from a pure syntax definition. The symbol in the figure below indicates that some semantic information is set, or that a condition must be satisfied for the parse to succeed.

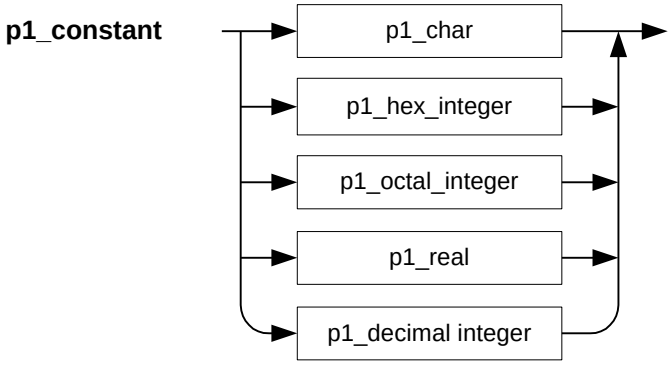


e.g. evaluate a constant

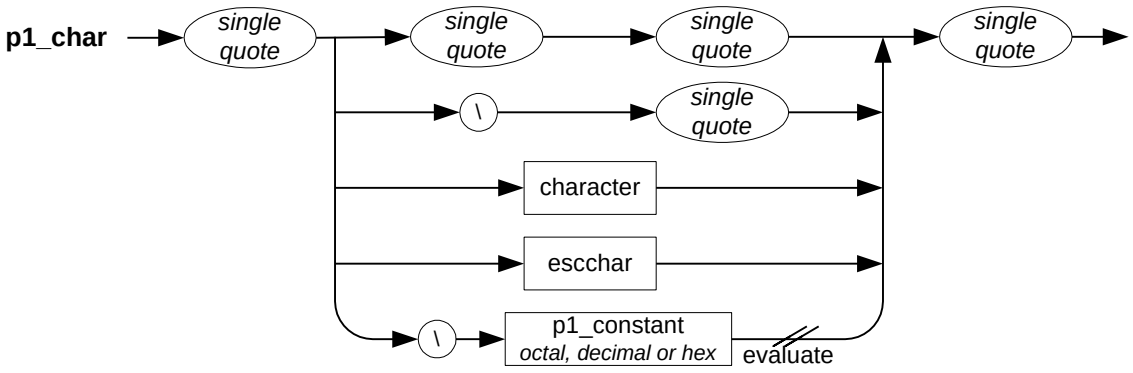


e.g. apply constraint on preceding parsed items

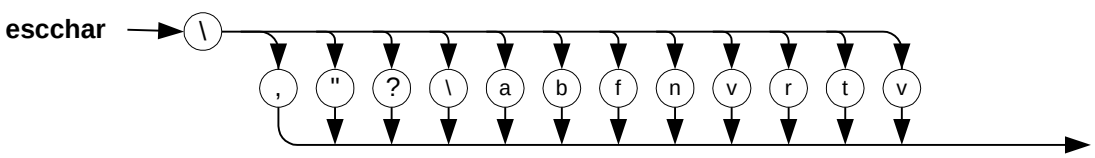




Implementation note: these items (p1_char etc.) are not separately named predicates in the PROLOG code



Note: ANSI C also allows for multibyte characters



Note: single quote is not an escchar. \' in a string is not an escape sequence

Figure 8. Pass-1 grammar rules (1)

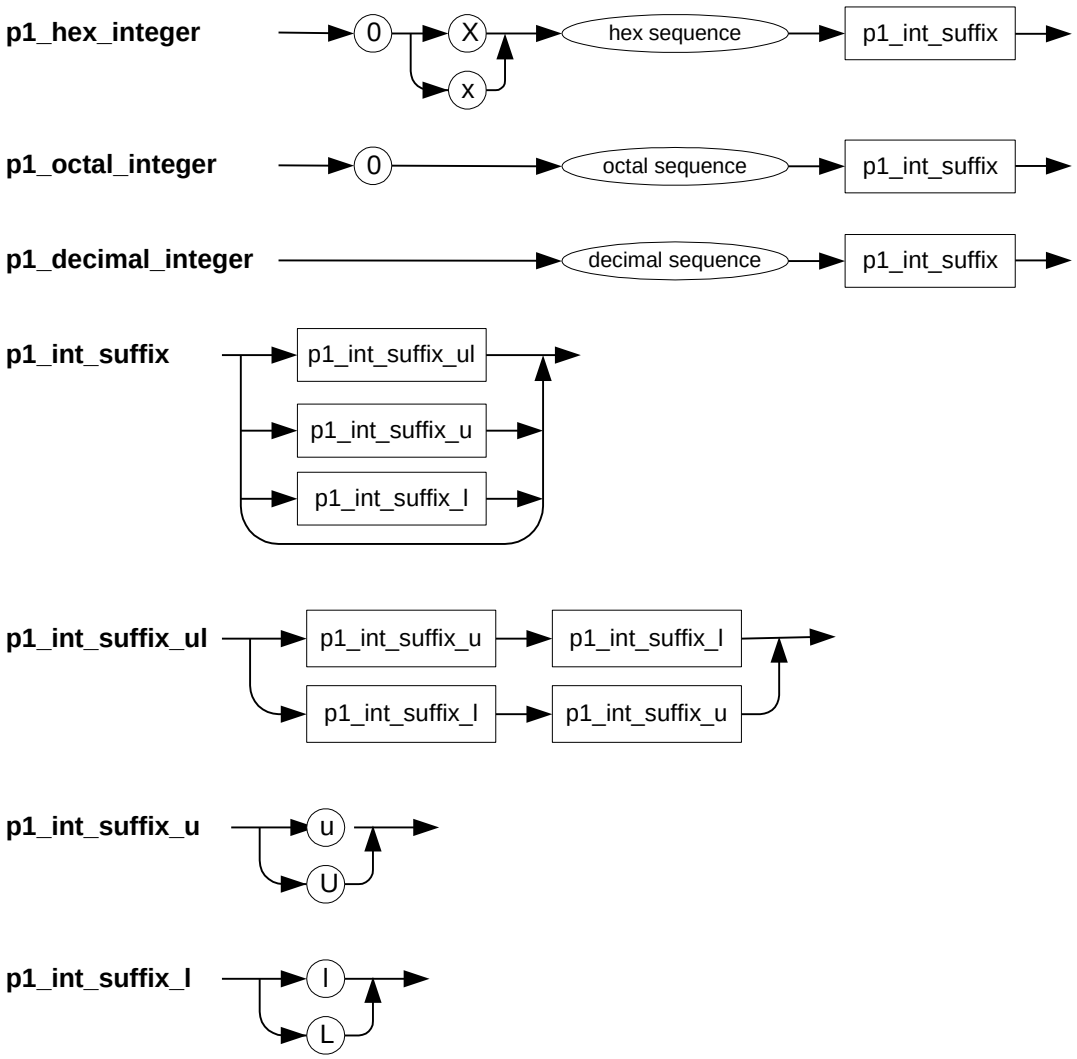


Figure 9. Pass-1 grammar rules (2)

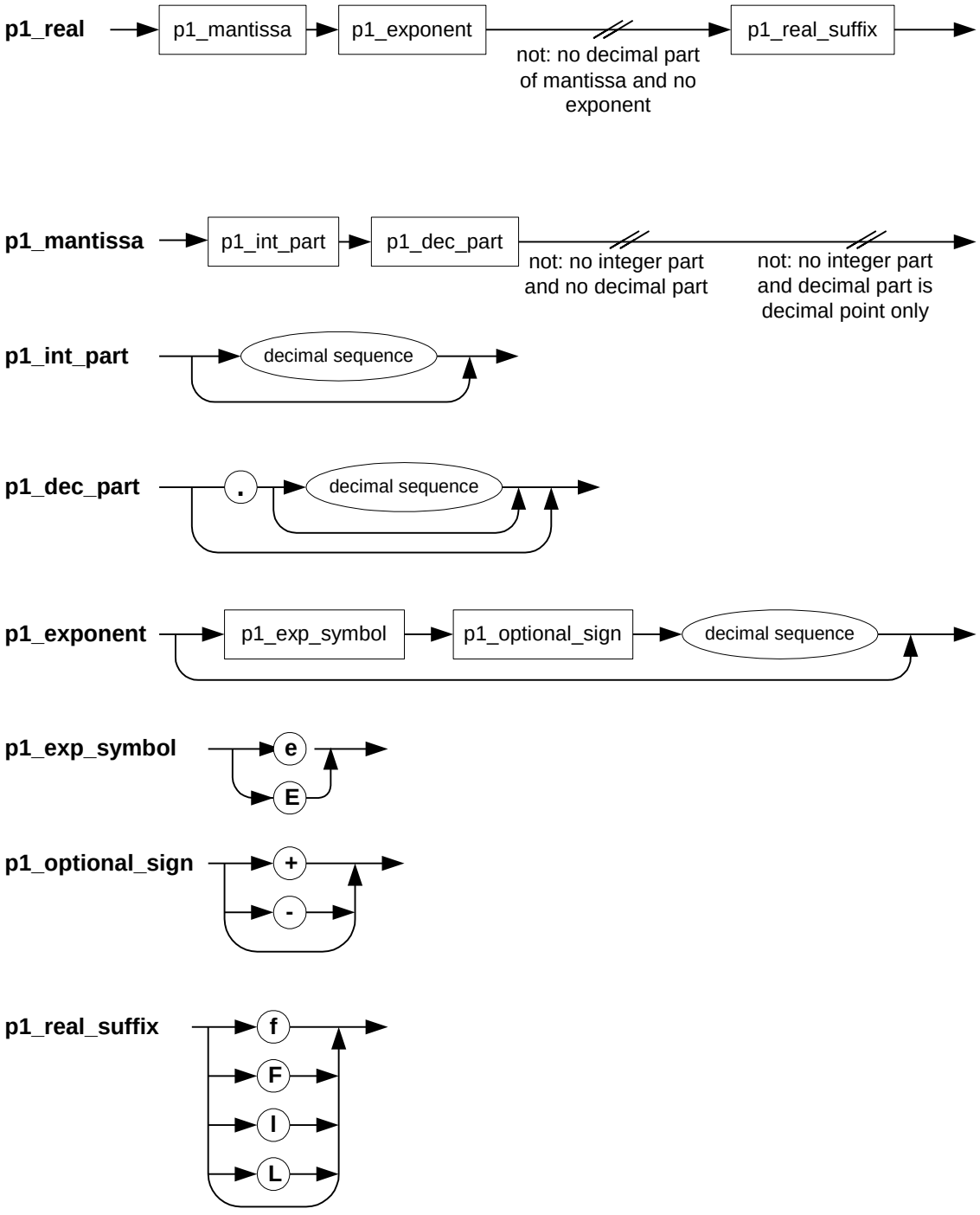


Figure 10. Pass-1 grammar rules (3)

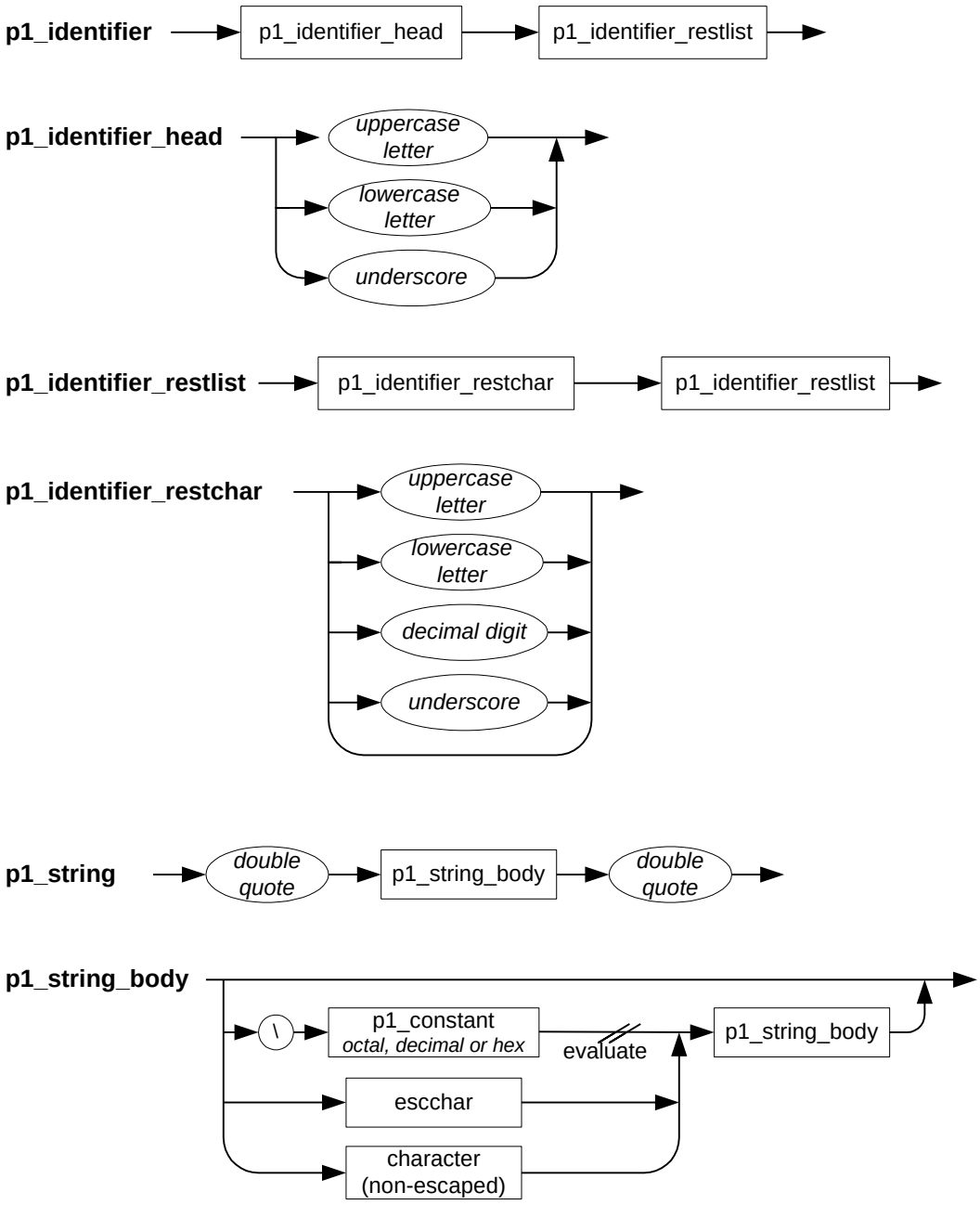


Figure 11. Pass-1 grammar rules (4)

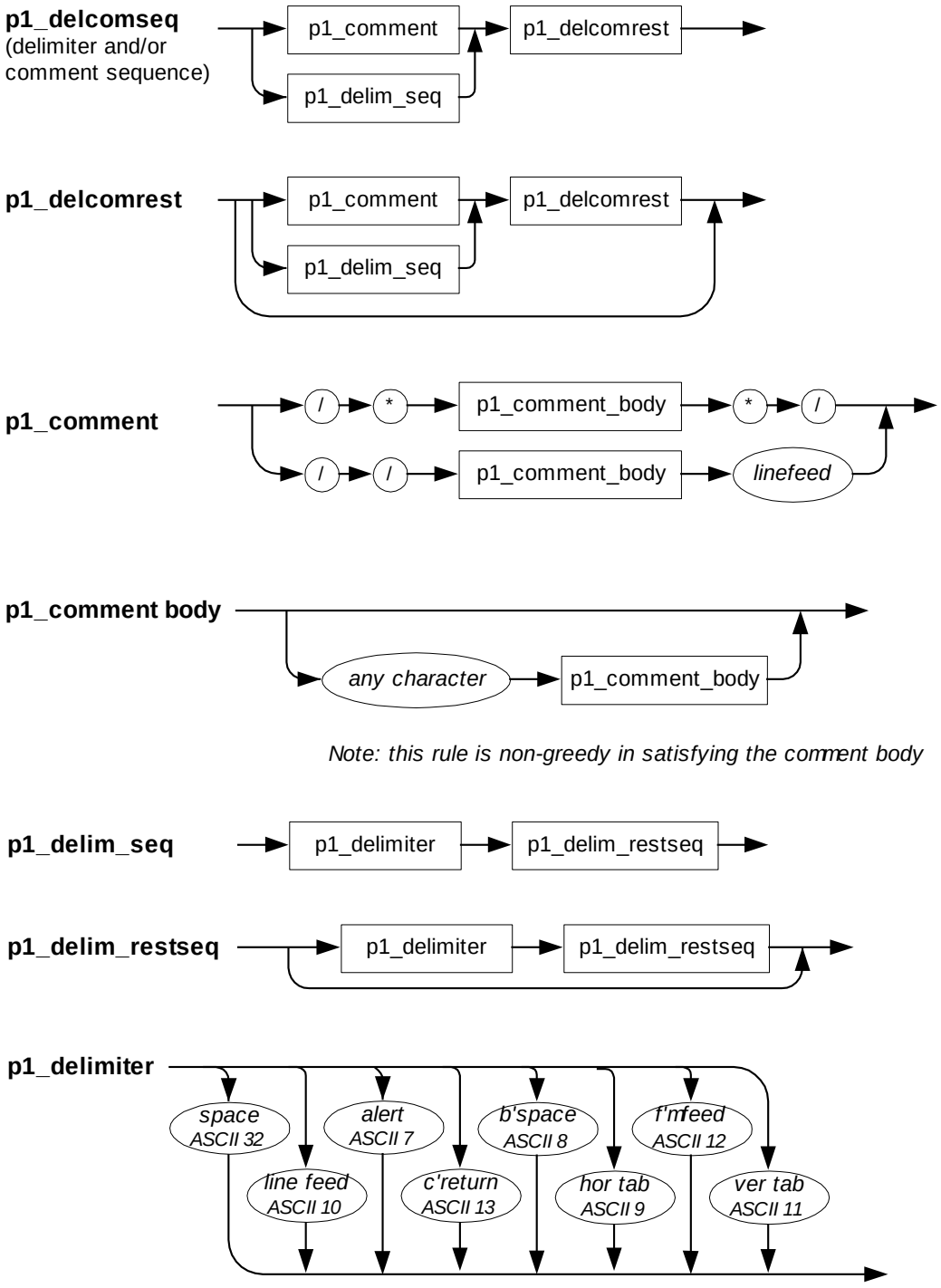


Figure 12. Pass-1 grammar rules (5)

4. Operator definitions

4.1 Operator overview

Operators lie in a layer between pass-1 and expression parse modules. They are called by the expression parser in order to combine operands into expressions. Expression parsing and operator parsing work with pass-1 output as their raw material. A successful parse produces operator attributes such as name, precedence, position, associativity, arity, and morphology. The operator definition system does not prescribe *type* or *lvalue* requirements on operands. This processing is left to a validator module.

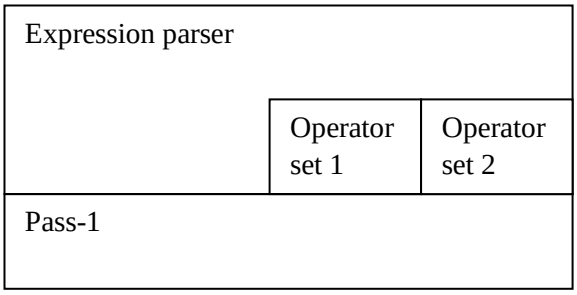


Figure 13. Operator layer

It is a feature of GP4 that various sets of operators can be defined by the user. The expression parser takes as a parameter the names of all the sets of operators to be used in an expression parse call. For example,

```
ex_expr([cc,fz,sm],...)
```

will look for an expression involving the operator sets cc (the C/Java set), fz (the fuzzy/probabilistic operator set) and sm (the state machine operator set).

4.2 Operator attributes

The attributes of an operator are as follows.

Name

Operators are given a name that can be distinct from their production sequence. This allows for aliases as well as disambiguation of homonyms (i.e. overloaded operators such as ++). It provides isolation of much of the complete software system from syntax changes. In Prolog it is also more convenient to have alphanumeric names rather than non-alphanumeric ones, since the latter require quoting.

For example, the C style pre-increment and post-increment operators both have the production "++" , and are called `preinc` and `postinc` respectively.

Precedence

An alternative term for precedence is *priority*. We adopt the convention that the higher the precedence, the sooner they are **bound** to their arguments, regardless of the order in which they appear in the expression. Precedence (but not only precedence) determines what the structure of an expression is. Note that this does **not** mean that they will necessarily be **evaluated** sooner, although this is sometimes perforce the case.

Under standard precedence conventions, multiplication and division have a higher precedence that addition and subtraction. In the expression

```
a+b+c*d-e
```

the multiplication binds terms `c` and `d`. The expression should be read as

```
a+b+(c*d)-e
```

Obviously, `c*d` will need to be evaluated before its result can be combined with other terms, but it would typically be permissible to add `a` and `b` before multiplying `c` and `d`.

Position

The position of an operator can be

- **prefix**, as in `++i`
- **dyadic infix**, as in `i+j`
- **postfix**, as in `a++, foo(bar), arr[6]`
- **triadic infix,** as in `a?b:c`

Note how **postfix** operators `()` and `[]` come in two parts, **circumfixing** their argument.

Associativity

Associativity determines the binding order of a sequence of terms with operators of equal precedence. Associativity can be

- **left associative**, as in
  - `a+b+c+d`, equivalent to `(a+b)+c)+d`
  - `a++++`, equivalent to `(a++)++`
    (conceivable, but not legal C because `a++` is never an lvalue, whatever its type – even if it is of type int*)
  - `a[b][c]`, equivalent to (a[b])[c]
  - `a?b:c?d:e` , equivalent to `(a?b:c)?d:e` (but in C this operator is *right* associative)

- **right associative**, as in
  - `a=b=c=d`, equivalent to `a=(b=(c=d))`
  - `!!b`, equivalent to `!(!b)`
  - `a?b:c?d:e`, equivalent to `a?b:(c?d:e)`

- **non-associative** This applies when an operator does not associate with operands containing operators of equal precedence to this operator. There are no examples of this kind of operator in C, but in some languages (including Prolog) the expression

  ```
  a=b=c
  ```

  would be illegal on grounds of non-associativity.

We can interpret left associativity as meaning: the operand on the left of this operator must contain operators of the same or higher precedence than this operator. The operand on the right of this operator must contain operators of strictly higher precedence.

Similarly right associativity means: The operand to the right of this operator must contain operators of the same or higher precedence than this operator. The operand on the left of this operator must contain operators of strictly higher precedence.

Note that `a?b?c:d:e` is unambiguously equivalent to `a?(b?c:d):e`.

Arity

The **arity** specifies how many terms bind to the operator. The arity can be:

- **monadic** (also known as unary), as in `a++`
- **dyadic** (also known as binary) as in `a+b`, `a[b]`
- **triadic** (also known as ternary) as in `a?b:c`

Note: we regard `a(b,c,d)` as a dyadic operator *function_call* operating on the operands `a` (the *function name*) and `b,c,d` (the *argument list*). The argument list is itself a data structure that could be regarded as being constructed by a **polyadic** operator. GP4 represents the argument list as a Prolog list. An alternative would have been to build a left-associative tree using the comma operator.

Similarly the array operator, as in `a[b],` is regarded as a dyadic operator with operands `a` and `b`.

Morphology

The morphology can be

- **keyword** type, where the operator is an identifier-conformant keyword, e.g. `fand` (fuzzy and).
- **symbol** type, where the operator is defined in terms of non-alphanumeric symbols, e.g. `->` .
- **keyword-assignment type**, where the operation is combined with assignment, e.g `fand=` .
- **symbol-assignment type**, where the operation is combined with assignment, e.g. `+=` .
- **brackets**, used to override precedence rules. They are hard coded into the expression parser. Precedence-overriding brackets are to be distinguished from function argument brackets, which are defined as an operator.

The morphology is of no fundamental consequence in the GP4 system; the production rules simply use the pass-1 tokens required. These are typically ASCII codes and `[p1_id,` *identifier*`]` tokens.

Remarks

1. Mixed associativity at the same precedence level

Suppose we define operators *las* and *ras* as left associative and right associative operators respectively, at the same precedence level. How should we interpret: the following?

```
a ras b ras c las 2 las 3
```

Two parses fit the rules, as shown in the following figure:

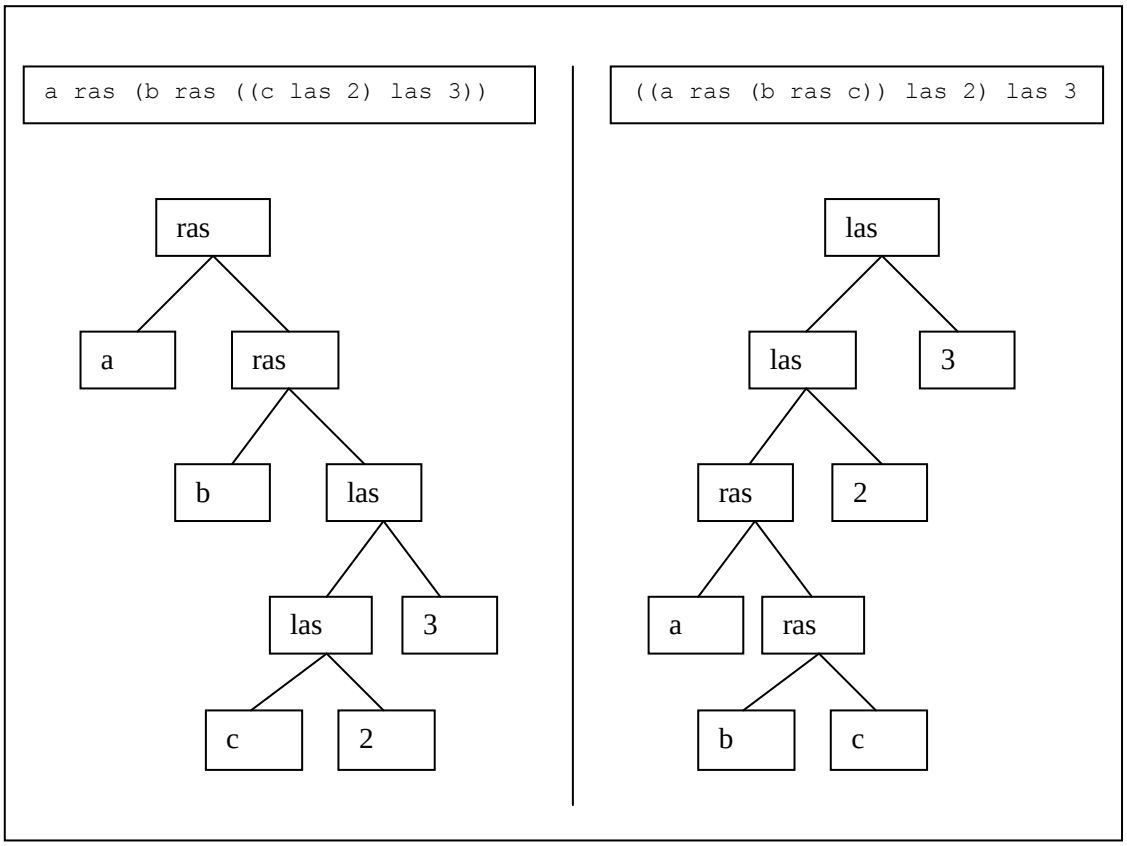


Figure 14. Mixed associativity

We observe that the first parse is what we would get if the **`las`** operator were to have a precedence greater than **`ras`**, and the second if the precedence of **`ras`** were to be greater than **`las`**. So either effect can be explicitly achieved by allocating different precedences. So:

We require that the user allocates different precedences to disambiguate this situation.

Another case is the expression

```
a las b las c ras 2 ras 3
```

If **las** and **ras** have the same precedence, this expression does not parse at all. Again, if the user allocates different precedences then it does parse, one way or another, whether the precedence of **las** is greater than that of **ras** or vice versa.

The reason that the parse `((a las b) las c) ras (2 ras 3)` is not obtained as a parse is that the first `ras` would have operands on both sides of equal precedence to its own precedence. The associativity requirement given above disallows this; the requirement states that the operand to the left must be of strictly higher precedence.

4.3 Operator definition format

GP4 defines operators as follows

`op_df(NTOKENS,OPSET,[op,PRECEDENCE,SHAPE,OPNAME])-->` *grammar rhs*

NTOKENS the number of tokens the operator consumes. This information is supplied by the user for efficiency reasons. It could be obtained by reverse-driving the predicate e.g. as follows to find the number of tokens in the "dyadic scope" operator, given that its name is `dscope`. Its production is seen to be "`::`", of length 2 tokens.

```
op_df(_,_,[_,_,_,dscope],WHAT,[]),
gn_length_list(WHAT,LEN).
```

giving

```
WHAT = [58,58] ,
LEN = 2
```

Here , the `dscope` (dyadic scope) operator has been made to yield its production, which is the ASCII for '`::`'. However, this is very inefficient in a critical part of the system (it requires a search through all operators), so for efficiency we have had to supply this information explicitly. A test in the test suite checks for the correctness of this parameter.

OPSET the name of the operator set to which this definition is to belong, e.g. `cc` for the C/Java set, `fz` for the fuzzy set etc.

PRECEDENCE the precedence level (high numbers for high precedence, =high priority)

SHAPE defines the **Arity**, **Position**, and **Associativity**.

- `[f,x]`[2] monadic, prefix, non- associative
- `[f,y]` monadic, prefix, right-associative
- `[x,f]` monadic, postfix, non-associative
- `[y,f]` monadic, postfix, left-associative
- `[x,f,x]` dyadic, infix, non- associative
- `[x,f,y]` dyadic, infix, right-associative
- `[y,f,x]` dyadic, infix, left- associative
- `[x,f,x,g,y]` triadic, infix, right-associative `a?(i?j:k)?(p?q:r)`
  this must be followed by `[continued]` in a definition with the same **OPNAME**
- `[f,argl]` dyadic, for argument LISTS, circumfixed by two productions.

  The definition comes in two parts
  - The left-circumfixing production
  - The right-circumfixing production; which must have a **SHAPE** of `[continued]` in a definition with the same **OPNAME**

  In practice, this is the operator that is used for a function call. The productions are "`(`" and "`)`". It dyadically combines the function to be called and the argument list.
- `[f,argi]` dyadic, but for argument ITEMS, circumfixed by two productions.

  The definition comes in two parts
  - The left-circumfixing production
  - The right-circumfixing production; which must have a **SHAPE** of `[continued]` in a definition with the same **OPNAME**

  In practice, this is the operator that is used for array indexing. The productions are "`[`" and "`]`". It dyadically combines the array to be indexed and the argument item.

The symbols used above (`f, x, y, argl, argi` etc.) are literals.

The arguments to operators are in general expressions. The `[f,argl]` interprets the comma as a separator between arguments, and each argument has to use operators that have a higher precedence than the comma operator. The `[f,argi]` operator accepts expressions containing the comma operator in the same way as any other operator.

OPNAME is the name given to the operator, e.g. `postinc` for the post-increment operator, ++.

[2] The [f,x] etc. notation is borrowed from the PROLOG way of specifying operators, as in e.g. [Clocksin. p.93].

grammar rhs defines the syntax, using standard Prolog Definite Clause Grammar rules. This production should *not* contain a Prolog cut, for backtracking reasons, and for reversibility reasons (the test script retranslates parses to what they were produced from).

Implementation-specific note: Operator `op_df/5` will in general be defined in more than one file (e.g. there may be one file for C-operators, one file for fuzzy operators). Some Prolog implementations, including WinProlog, will require a *multifile* declaration.

Note there are no restrictions on the order in which predicates are defined.

- There is no requirement to define e.g. **->** before **-** (leading substring issue).
- There is no requirement to define higher precedence operators before lower precedence ones.

Examples

```
/* A MONADIC EXAMPLE */
op_df(2,cc,[op,180,[f,y],mscope])    --> {name('::',[A,B])}, [A],[B].

/* SOME DYADIC EXAMPLES */
op_df(2,cc,[op,170,[y,f,x],imemsel]) --> {name('->',[A,B])}, [A],[B].
op_df(2,cc,[op,40,[x,f,y],asxmul])   --> {name('*=',[A,B])}, [A],[B].
op_df(1,fz,[op, 60,[y,f,x],fand])    --> [[p1_id,fand]].

/* TRIADIC OPERATOR IN TWO PARTS */
op_df(1,cc,[op,45,[x,f,x,g,y],aif]) -->  {name('?',[A])}, [A].
op_df(1,cc,[op,45,[continued],aif]) -->  {name(':',[A])}, [A].

/* FUNCTION CALL TYPE OPERATOR IN TWO PARTS */
op_df(1,cc,[op,170,[f,argl],fcall])    --> {name('(',[A])},[A].
op_df(1,cc,[op,170,[continued],fcall]) --> {name(')',[A])},[A].

/* ARRAY SUBSCRIPT OPERATOR IN TWO PARTS */
op_df(1,cc,[op,170,[f,argi],sqbr])     --> {name('[',[A])},[A].
op_df(1,cc,[op,170,[continued],sqbr])  --> {name(']',[A])},[A].
```

Additional definition required, defined in module `op_aa.pl`:

`op_info(START,MAXIMUM,INCREMENT)`

`START`: The START priority, i.e. the lowest priority in use

`MAXIMUM`: The MAXIMUM priority in use

`INCREMENT`: The priority increment (for efficiency: otherwise just use 1)

This clause is used by the expression parser to define its range of searching for operators at different precedence levels. It must be consistent with **all** operator sets. If the increment is set to 1, then this is safe, but inefficient if in fact the priority spacing is, say, 5 or 10. The

performance of the system is strongly related to the number of precedence levels that need to be covered.

4.4 Tables of operators defined for parsing

The fact that an operator has been defined for *parsing* is separate from whether an operator has been implemented for *evaluation*. Refer to section 10.3 for details of the implementation operators for evaluation.

Under the Language column, extensions absent in C or Java are provisionally defined by GP4, as a suggestion to new domain-specific languages if required.

4.4.1 Operator set "cc"

This set is based on C, some C++, Java and a few extensions.

Operation	Symbol	Definition parameter	Lang
Scope Resolution			
member scope	`::`	`[op,180,[y,f,x],dscope]`	C++
global scope	`::`	`[op,180,[f,y],mscope]`	C++
Primary Suffixes			
Function call	`()`	`[op,170,[f,argl],fcall]` `[op,170,[continued],fcall]`	C
Array indexing	`[]`	`[op,170,[f,argi],sqbr]` `[op,170,[continued],sqbr]`	C
Memory			
member select	`.`	`[op,170,[y,f,x],memsel]`	C
indirect member select	`->`	`[op,170,[y,f,x],imemsel]`	C
Various monadic			
address-of	`&`	`[op,160,[f,y],addrof]`	
dereference	`*`	`[op,160,[f,y],deref]`	C
reciprocal	`/`	`[op,160,[f,y],recip]`	GP4
plus	`+`	`[op,160,[f,y],mplus]`	C
minus	`-`	`[op,160,[f,y],mminus]`	C
bitwise not	`~`	`[op,160,[f,y],bnot]`	C
logical not	`!`	`[op,160,[f,y],lnot]`	C
pre-increment	`++`	`[op,160,[f,y],preinc]`	C
pre-decrement	`--`	`[op,160,[f,y],predec]`	C
post-increment	`++`	`[op,160,[y,f],postinc]`	C
post-decrement	`--`	`[op,160,[y,f],postdec]`	C

Memory pointer			
member pointer select	`->*`	`[op,150,[y,f,x],memptrsel]`	C++
indirect mem ptr select	`.*`	`[op,150,[y,f,x],imemptrsel]`	C++
Exponentiative			
exponentiation	`**`	`[op,145,[y,f,x],pwr]`	Fortran
Multiplicative			
multiplication	`*`	`[op,140,[y,f,x],xmul]`	C
division	`/`	`[op,140,[y,f,x],xdiv]`	C
modulo	`%`	`[op,140,[y,f,x],mod]`	C
Additive			
addition	`+`	`[op,130,[y,f,x],dplus]`	C
subtraction	`-`	`[op,130,[y,f,x],dminus]`	C
Shifting			
arithmetic shift right	`>>`	`[op,120,[y,f,x],asr]`	C
arithmetic shift left	`<<`	`[op,120,[y,f,x],asl]`	C
logical shift right	`>>>`	`[op,120,[y,f,x],lsr]`	Java
logical shift left	`<<<`	`[op,120,[y,f,x],lsl]`	GP4
circular shift right	`>>>>`	`[op,120,[y,f,x],csr]`	GP4
circular shift left	`<<<<`	`[op,120,[y,f,x],csl]`	GP4
Relational			
less than or equal	`<=`	`[op,110,[y,f,x],le]`	C
greater than or equal	`>=`	`[op,110,[y,f,x],ge]`	C
less than	`<`	`[op,110,[y,f,x],lt]`	C
greater than	`>`	`[op,110,[y,f,x],gt]`	C
equal	`==`	`[op,100,[y,f,x],eq]`	C
not equal	`!=`	`[op,100,[y,f,x],ne]`	C
Bitwise			
bitwise and	`&`	`[op, 90,[y,f,x],band]`	C
bitwise xor	`^`	`[op, 80,[y,f,x],bxor]`	C
bitwise eqv	`~^`	`[op, 80,[y,f,x],beqv]`	GP4
bitwise incl or	`|`	`[op, 70,[y,f,x],bior]`	C
Logical			
short-circuit and	`&&`	`[op, 60,[y,f,x],land]`	C
long-circuit and	`&&&`	`[op, 60,[y,f,x],lland]`	GP4
xor	`^^`	`[op, 55,[y,f,x],lxor]`	GP4
equivalence	`!^^`	`[op, 55,[y,f,x],leqv]`	GP4

short-circuit or	`||`	`[op, 50,[y,f,x],lior]`	C
long-circuit or	`|||`	`[op, 50,[y,f,x],llior]`	GP4
Arithmetic conditional			
arithmetic if	`? :`	`[op,45,[x,f,x,g,y],aif]` `[op,45,[continued],aif]`	C
Assignment			
assign	`=`	`[op,40,[x,f,y],assign]`	C
exponentiate-assign	`**=`	`[op,40,[x,f,y],aspwr]`	GP4
multiply-assign	`*=`	`[op,40,[x,f,y],asxmul]`	C
divide-assign	`/=`	`[op,40,[x,f,y],asxdiv]`	C
modulo-assign	`%=`	`[op,40,[x,f,y],asmod]`	C
add-assign	`+=`	`[op,40,[x,f,y],asplus]`	C
subtract-assign	`-=`	`[op,40,[x,f,y],asminus]`	C
bitwise-and-assign	`&=`	`[op,40,[x,f,y],asband]`	C
bitwise-xor-assign	`^=`	`[op,40,[x,f,y],asbxor]`	C
bitwise-equiv-assign	`!^=`	`[op,40,[x,f,y],asbeqv]`	GP4
bitwise-incl-or-assign	`|=`	`[op,40,[x,f,y],asbior]`	C
arith shift right assign	`>>=`	`[op,40,[x,f,y],asasr]`	C
arith shift left assign	`<<=`	`[op,40,[x,f,y],asasl]`	C
log'l shift right assign	`>>>=`	`[op,40,[x,f,y],aslsr]`	Java
log'l shift left asign	`<<<=`	`[op,40,[x,f,y],aslsl]`	GP4
circ shift right assign	`>>>>=`	`[op,40,[x,f,y],ascsr]`	GP4
circ shift left assign	`<<<<=`	`[op,40,[x,f,y],ascsl]`	GP4
Sequence			
sequence	`,`	`[op,10,[y,f,x],seq]`	C

Table 3. Operator set "cc"

Notes

1. We cannot differentiate between integer multiply/divide and real multiply/divide at this stage.
2. Not implemented are C keyword operators (`sizeof`), cast operators.
3. Not implemented are C++ keyword operators (`new, delete, throw`).
4. Not implemented are Java keyword operators (`instanceof`)

5. Java interprets
   - <logical item> & <logical item> as “long circuit and” (our `&&&`)
   - <integral item> & <integral item> as “bitwise and”
   - Similarly with “|”
6. The C bitfield symbol (:) is not used in expressions - only in declarations

4.4.2 Operator set "fz"

Operation	Symbol	Definition parameter	Lang
Probabilistic			
boost odds	`boost`	`[op,140,[y,f,x],boost]`	Dexios
depress odds	`depress`	`[op,140,[y,f,x],depress]`	Dexios
Fuzzy			
fuzzy not	`fnot`	`[op,160,[f,y],fnot]`	Dexios
fuzzy and	`fand`	`[op, 60,[y,f,x],fand]`	Dexios
fuzzy xor	`fxor`	`[op, 55,[y,f,x],fxor]`	Dexios
fuzzy equivalent	`feqv`	`[op, 55,[y,f,x],feqv]`	Dexios
fuzzy inclusive or	`fior`	`[op, 50,[y,f,x],fior]`	Dexios
Assignment			
boost-assign	`asboost`	`[op,40,[x,f,y],asboost]`	Dexios
depress-assign	`asdepr`	`[op,40,[x,f,y],asdepr]`	Dexios
fuzzy and assign	`asfand`	`[op,40,[x,f,y],asfand]`	Dexios
fuzzy xor assign	`asfxor`	`[op,40,[x,f,y],asfxor]`	Dexios
fuzzy equivalent assign	`asfeqv`	`[op,40,[x,f,y],asfeqv]`	Dexios
fuzzy incl or assign	`asfior`	`[op,40,[x,f,y],asfior]`	Dexios

Table 4. Operator set "fz"

Notes

- These operators were applied to a probabilistic/fuzzy inference engine in the Dexios project [Dexios].
- The semantics are defined in the chapter on expression evaluation.

4.5 Operator grammar

The terminals of these grammar rules are output tokens from pass-1 (not pass-1 rules themselves).

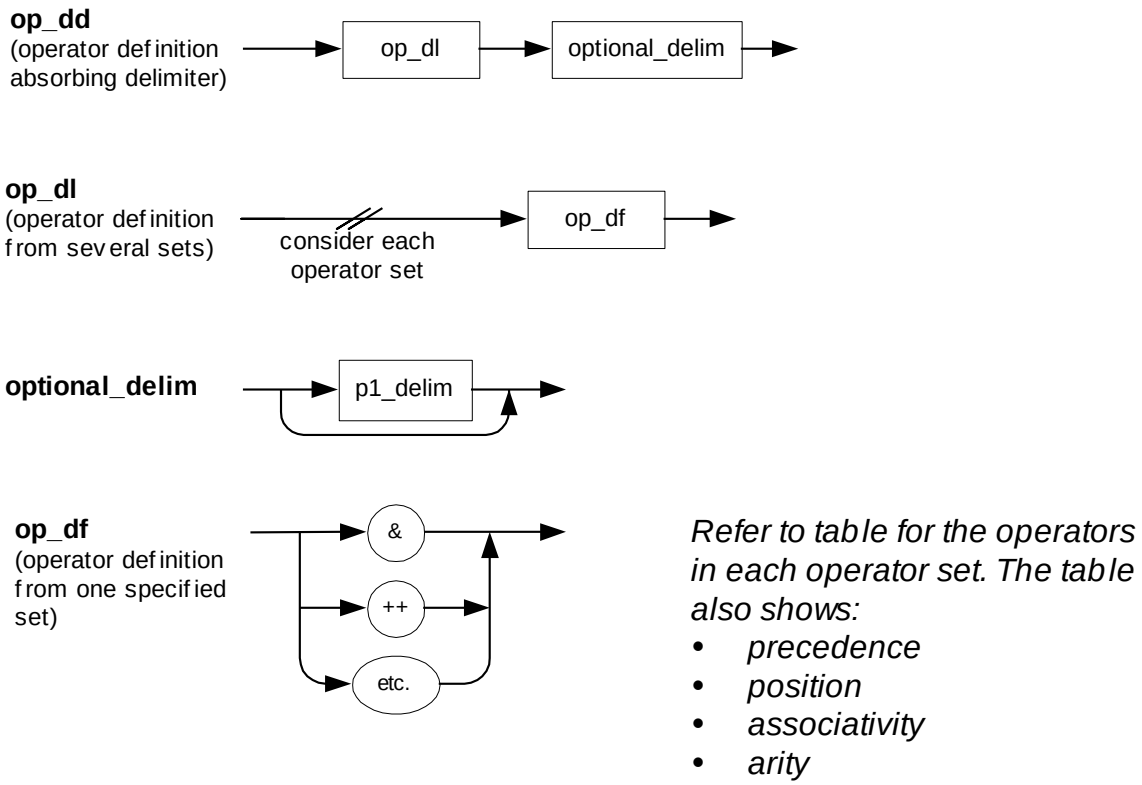
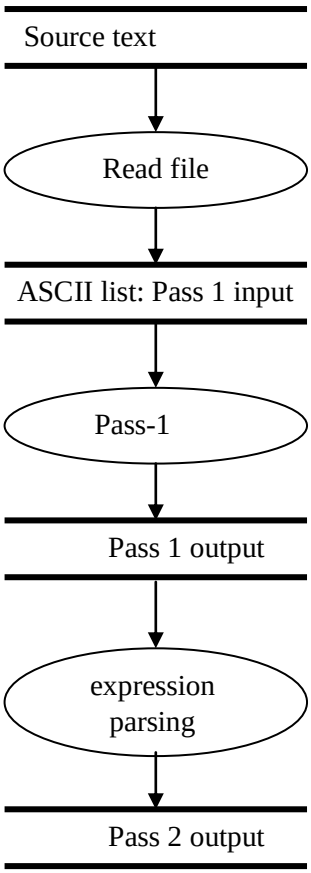


Figure 15. Operator grammar

5. Expression parsing

5.1 Overview

The task of expression parsing is to combine terms and operator sequences into expressions. This is shown in the following figure:




```
1+a*2-5
```

```
[49,43,97,42,50,45,53]
```

```
[[p1_co,int,none,10,1],43,[p1_id,a],42,[p1_co,
int,none,10,2],45,[p1_co,int,none,10,5]]
```

```
[ex_expr,
    [[ex_dyadic,dminus],
        [[ex_dyadic,dplus],
            [ex_co,int,1],
            [[ex_dyadic,xmul],
                [ex_id,a],
                [ex_co,int,2]
            ]
        ],
        [ex_co,int,5]
    ]
].
```

Figure 16. Overview of expression parsing

5.2 Some considerations

If expression parsing can be achieved in using Prolog Definite Clause Grammars, then parsing the rest of a language is probably relatively straightforward. The challenge is to parse expressions by consuming input tokens from the left only, i.e. without knowing in advance where the end of the expression is. The rest-string must be the token sequence starting at the first token that cannot belong to the expression.

An alternative approach that was considered is to identify the end of an expression with a weaker grammar that does not know about precedences etc., but which can identify the text of an expression. This would give us a handle on both ends of the expression. Then one could use productions such as

any_text_longest_first operator any_text_shortest_first

to pick out the operators at highest priority first and recursively descend. This gives left associativity. For right associative operators the rule would be

any_text_shortest_first operator any_text_longest_first

One would have to ensure the entire expression string is consumed on each call. The method has the advantage of being isomorphic to the way a human would tend to parse an expression. However, this method would be inefficient for long expressions, as it would have to split the string in a quadratically rising number of ways. The idea of applying this technique on the entire input string without identifying the end of an expression in advance can be ruled out *a fortiori* for this reason.

The goal of left-hand-consuming input tokens has been achieved, but many issues required special attention:

- grammar transformation to avoid left recursion
- achieving left associativity
- disambiguation of overloaded operators
- disambiguation of overloaded leading substrings in operators
- absorption of white-space
- run-time efficiency
- testability

Expressions can be parsed working from multiple operator sets per call, as mentioned in the section 4.1

The expression parser does not consider *type* or *lvalue* contexts. An example of an invalid statement in C because an *lvalue* is required is:

```
"hello world"=x;
```

Variables, constants and strings are considered with impartiality. A language system should provide a separate **validator** to check for mismatches in this respect.

One restriction in the above scheme is that incorrect type/lvalue contexts will not cause backtracking in expression parsing. However, computer languages (e.g. C) are usually (*always*, maybe), designed so that no backtracking in such a situation is required.

Comment not based on the current parsing strategy: The backtracking on type/lvalue mismatch issue would be of importance in the following situation. Suppose we were to relinquish the C strategy of *longest operator wins* and replace it by the *highest priority operator with an operand wins* strategy. Consider parsing an expression such as

```
c+++++d
```

If we define ++ to be a right associative prefix and left-associative suffix, then we can parse this as

```
((c++)++)+d  or  (c++)+(++d)  or  c+(++(++d))
```

However, if we cannot tell the expression parser that ++ does not yield an lvalue, then it will accept the first parse rather than backtracking to find the second one. A way to obtain the second parse is to define ++ as nonassociative – i.e. shape `[x,f]` and `[f,x]` in the definition described in section 4.3.

There is of course no ambiguity when white space is used:

```
c++ + ++d
```

5.3 Choice of expression grammar to implement

Figure 24 shows the grammar of expressions in the C language in a railroad diagram. The diagram does not consider context issues such as types and lvalues.

This grammar exhibits a certain lack of generality:

- Suffix expressions can only operate on primary expressions. This is why in C an indirect function call might be `(*pf)(x,y,z)`, although as it happens this is equivalent to `pf(x,y,z)`. The call cannot be `*pf(x,y,z)` because it is not possible (even if we could adjust operator precedences), given C syntax as defined by Figure 24, to arrange for the dereference operator to bind before the argument list suffix. Argument list suffixes can only bind to primary expressions, and `*pf` is not a primary expression.
- The operators **`.`** and **`->`** have a context restriction in that their right hand argument must be an identifier. This is in itself a desirable feature, but for a generic parser, we postpone type and lvalue constraints to a later validation phase.
- The suffix symbols **`[]`** and **`()`** are the only instances of single-argument-circumfixing operators and multiple-argument-circumfixing operators.

We generalise, for better of for worse, (where for the worse, we compensate with a validation module), and specify a grammar for GP4 with the following features:

- Postfix expressions can operate on any expression in principle, (i.e. providing the precedences allow it).

- There are no facilities for context restrictions based on type and lvalue considerations.
- The operator features described in Section 4 allow for multiple operators of the circumfixing suffix type.

The resulting expression grammar of GP4 is given in Figure 25. However, this description, although perfectly valid, does not bring out precedence and associativity issues, and does not reflect an implementation strategy. These issues are considered in the next subsection.

5.4 Addressing the tough issues

5.4.1 Grammar transformation to avoid left recursion

The grammar given in Figure 25 can be transformed so that there is no left recursion. The figure below illustrates left recursion:

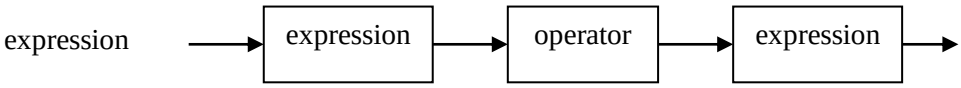


Figure 17. Left-recursive rule

Standard Prolog definite clause grammars cannot not cope with left recursion. Code such as the following (omitting parameters to the predicates a, b, and c which would normally be present)

```
a:-a,b.
a:-c.
```

can be read as:

> To prove "a", first prove "a" and then prove "b". If the previous strategy fails, prove "c" instead.

This results in infinite recursion on "a" - there is no way to break out. The fact that definite clause grammars are used makes no difference. If we have

```
expression-->
  expression,operator,expression.
```

there are hidden parameters; it is equivalent to

```
expression(S0,S):-
  expression(S0,S1),operator(S1,S2),expression(S2,S).
```

where `S0` is the string to be parsed, and `S` is the unused rest-string. The right-hand-side call to `expression(S0,S1)` works with the same input as the original left-hand-side call to `expression(S0,S)` - so no progress will be made in reducing the input string and so preventing infinite recursion.

The situation is different with e.g.

```
expression-->
 openbracket,expression,operator,expression,closebracket.
```

since by the time "`expression`" is evaluated on the right hand side, the string which it is passed is shorter than the original string, `openbracket` having "bitten something off".

For a discussion of grammar transformation in general, and with an example, refer to [Bennet, p.35].

The solution to transforming the grammar of GP4 lies in defining the arguments to operators as terms *at some precedence level*. As arguments to operators at the highest precedence level, terms are primitive (identifiers, constants or strings), and cannot involve expressions with other operators. A bracketed expression also behaves like a primitive term. When parsing an expression containing operators of various precedences, recursive calls to the *expression* goal take place, but they are at higher and higher levels, so there is no infinite recursion. It is where a primitive term is encountered that the recursion is broken, and part of the input string is "bitten off". If no primitive term is encountered, the level reaches a maximum level and the parse fails.

The result is the transformed grammar of Figure 26, Figure 27 and Figure 28 on pages 49, 50 and 51.

The transformed grammar is very close to the actual implementation. It incorporates some efficiency features. These are

- factored-out leading goals,
- a very important cut-fail sequence to prevent combinatorial explosion of backtracking.

These issues are discussed in a subsequent subsection.

The transformed grammar contains syntactic items *at a certain precedence level*, e.g. term(N). The starting level is given as zero as a typical value. The starting and maximum levels are set in the GP4 `op_aa.pl` module.

It is interesting to note the mutual recursion of ***expression*** and ***term*** in:

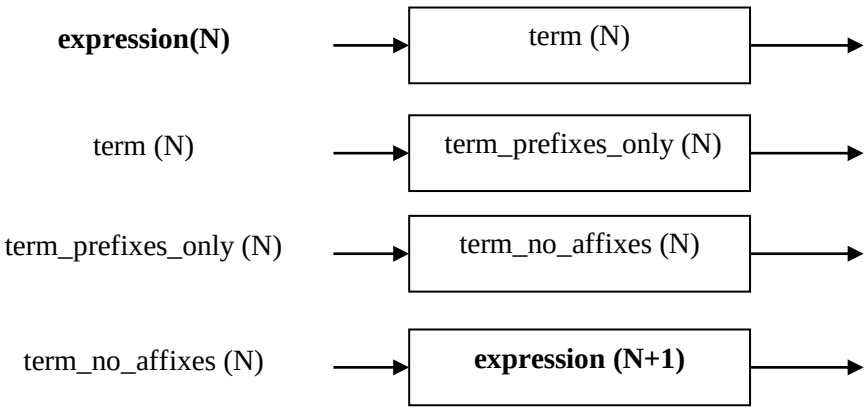


Figure 18. Mutual recursion of expression and term

An *expression* is defined by means of a *term*, and a *term* is defined by means of an *expression*. As discussed above, this mutual recursion is not infinite because of the associated "level" parameter. The last production above involving **expression (N+1)** is the only place in the grammar where the precedence level is increased.

The grammar also contains operations to associate sequences of terms. These operations apply to monadic operator sequences and dyadic term-operator-term-operator sequences.

Note also the introduction of the item **ex_identifier** (as opposed to a pass-1 identifier **p1_identifier**), defined in Figure 28, and used in the preceding figures. This allows us to specify identifiers that are not keyword operators, although the underlying pass-1 representation of each is identical. This does involve searching through all operators at every occurrence; the price is considerable but is affordable.

5.4.2 Achieving left associativity

It is not particularly difficult to left-associate a sequence of operators and terms, but if we were to try to work without sequences, we would be in trouble. The temptation is to observe that right associativity comes naturally, and to try to find something similar for left associativity. For example, the production following leads to natural right associativity.

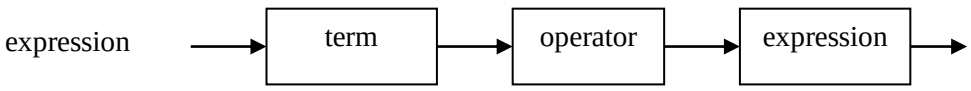


Figure 19. Natural right associativity

For left associativity, we cannot use the following because of left recursion.

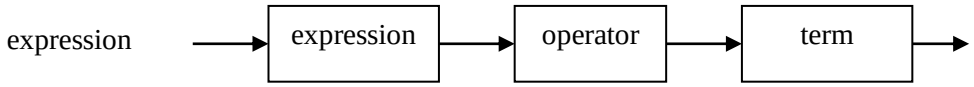


Figure 20. Left recursion hinders left associativity

It is tempting to consider using the right-associative rule and then transforming a right-associated structure into a left-associated one. The difficulty with this approach is knowing what part of an expression structure requires this transformation and what does not, as some parts of the parsed structure may be legitimately structured by the use of brackets.

The solution with term sequences appeared to be the most convenient, and it works well. At *some precedence level*, we see an expression as follows, and then apply a left-associate function to the list.

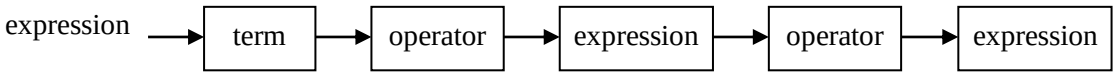


Figure 21. Expression at a precedence level

The "associate" function for triadic operators was not required as the C "arithmetic if" operator is right associative.

5.4.3 Disambiguation of overloaded operators

There are different types of overloading:

- monadic/dyadic overloading of **+** and **–**
- prefix/suffix overloading of **++ --**
- possible ambiguities with operator combinations, e.g. **c+++++1**
- overloading of leading substrings, e.g. **+** and **++**, **–** and **->**

A rule, which C imposes on us, is that the longest operators occurring in an input sequence (i.e. most characters) are taken to the exclusion of shorter ones. No regard is paid to operator precedence in this. So irrespective of precedences, `a&&b` will always be a "logical and", never be a "bitwise-and" and "address-of" combination, `a&(&b)`, (which happens to be invalid for typing reasons, but we are considering first principles here). Note that the precedence of both `&` operators is higher than `&&`, but this is not considered by the parser because of the existence of a longer operator in the input stream.

We still have to consider overloading at some operator length, e.g. the single-character operator "`&`". In almost all cases, a misinterpretation of a symbol will result in a failed parse and backtracking will take place to find a correct parse. Normally this will enable the dyadic `&`-operator to be distinguished form the monadic `&`-operator. Possible confusion could arise if two monadic and dyadic overloaded operators occur in succession, e.g. `a*&b`. In this case precedence and associativity play a role.

The parser will search and backtrack to find a fitting kind of term sequence. However, within precedence constraints the expression parser will take the first legal parse it can find. It is up to the operator definer to avoid ambiguities. The ways to constrain an expression are

- Operator precedence
- Operator associativity
- Operator symbol selection, verifying that overloading will not make multiple parses possible.

As already discussed, the following are *not* applicable to constraining expression parsing, only to post mortem error messages:

- Type restrictions
- Lvalue restrictions

The following are last resorts imposed on the user (as in C):

- Supply white space, e.g. **c++ + ++d** or **c++ ++ +d**
- Supply brackets, e.g. **(c++)+(++d)** or **((c++)++) +d**

5.4.4 Absorption of white space

Optional pass-1 white-space is absorbed by the items **ex_identifier**, **ex_constant**, **ex_string** and **op_dd**. From an expression parsing perspective, these items are the effective terminals.

5.4.5 Run-time efficiency

When a large number of operators are defined, then memory requirements and performance can be an issue. We have defined about 75 operators as "standard", and about 35 precedences as standard. Performance is acceptable but not fast on a 300 MHz machine (as sold in 1998); on a 3 GHz machine (as sold in 2003), it is good.

The GP4 expression grammar incorporates the optimisations described in the following subsections.

5.4.5.1 Factoring out duplicate leading goals

In PROLOG code, a duplicate leading goal can lead to inefficiency.
A duplicate leading goal occurs in the following Prolog code:

```
a:-b,c.
a:-b,d.
```

We can read this as: To prove "a", either prove "b" and then "c", or prove "b" and then "d". Now suppose we do this, starting by proving "b", then attempting to prove "c", but suppose also we fail to prove "c". We must now attempt to prove "b" and "d". But we have just proved "b"! Unfortunately, that information is now out of scope, and we must prove "b" again. This is the root cause of the inefficiency.

Assuming we are not interested in side-effects of a failed attempt at proving "a", a better way to define "a" would be

```
a:-b,z.
z:-c.
z:-d.
```

We can read this as: To prove "a", prove "b", then prove "z". To prove "z", prove "c", or if this fails, prove d instead. If, in a similar case to the one above, we apply this, we prove "b", then attempt to prove "c". The proof of "c" fails, but we now turn to proving "d" without re-proving "b".

We now give some Prolog code that demonstrates this point. Whenever "b" is called, it prints the letter **b** as a representation of doing some work. We add some parameters for realism. We are looking for a solution involving `a(4)`, which means finding `b(4),c(4)` or `b(4),d(4).` The way the goal has been defined, various values of `X` in `a(X)` will be generated before they are tested for the value we require, i.e. 4.

The following code calls b **eight** times:

```
go:-a(X),X=4,nl.

a(X):-b(X),c(X).
a(X):-b(X),d(X).

b(1):-write(b).
b(2):-write(b).
b(3):-write(b).
b(4):-write(b).

c(1). c(2). c(3).
d(1). d(2). d(3). d(4).

| ?- go.
bbbbbbbb
yes
| ?-
```

The following equivalent code (barring side effects) factors out the leading term b and only calls b **four** times:

```
a(X):-b(X),z(X).
z(X):-c(X).
z(X):-d(X).

b(1):-write(b).
b(2):-write(b).
b(3):-write(b).
b(4):-write(b).

c(1). c(2). c(3).
d(1). d(2). d(3). d(4).

| ?- go.
bbbb
yes
| ?-
```


The inefficiency can multiply into combinatorial explosion if it is not contained. Combinatorial explosion occurs where a duplicated leading term is defined in terms of lower level goals which are in turn defined using duplicate leading terms.

Duplicate leading goals arise in specifications such as the following:

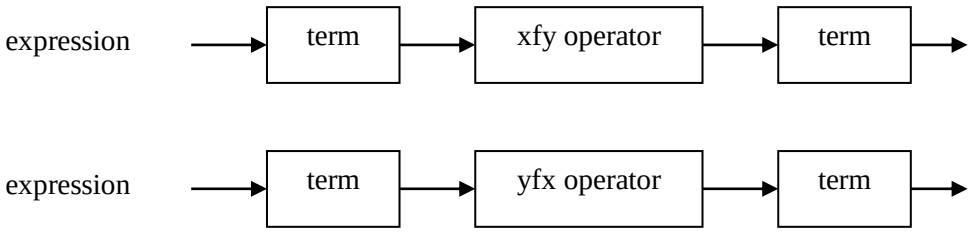


Figure 22. Duplicate goals

5.4.5.2 Controlling recursion incurred by precedence levels.

The expression grammar has a rule that a term at level N can be, via a few intermediate rules, an expression at level N+1. This in turn will lead to a search for an expression at level N+2. This process will continue up to the highest precedence level. Now suppose the input section in question is not an expression at all. The goal will fail at level MAX. Suppose we started at level 30, and suppose that MAX is 200. We will have stacked up expression calls at levels 31,32,33...200, and we will fail at level 200. The last rule attempted will be for a primary expression. Now if this fails at level 200, it will fail at all previous levels. The Prolog terminology for a committal to a particular solution path is to call the "cut" predicate. If after a cut, we call "fail", we exclude further attempts to satisfy the head goal (but not the calling goal above it). We can represent our desired behaviour by a cut-fail combination as shown in the following figure:

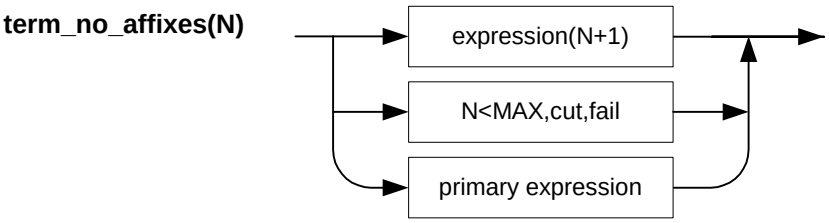


Figure 23. Cut-fail in a parse

Experiments showed that without this cut-fail combination, the parser was practically useless except for very simple expressions, but with it, long expressions can be parsed in very reasonable execution times. This control information is so important that it has been included in the syntax diagrams.

5.4.6 Testability

An important aspect to testing expressions is the ability to test hundreds of expressions easily. It is tedious and error-prone for the tester to have to predict the exact parsed output per test, and it consumes a lot of test-script "real-estate" to define tests this way.

To alleviate this situation an algebraic reconstruction predicate has been written that reconstructs a bracketed expression from the parse. The tester need only provide such bracketed expressions as oracles to tests. For example, the following defines a test, where `'a+b(a)++ rr'` is supplied to the parser, and `'a+(b(a)++)'` is the reconstruction after parse. The text `rr` belongs to the "rest-string".

The data for a test can be defined in terms of the input text and the reconstructed expression after parsing:

```
exzd(exs05, 'a+b(a)++ rr','a+(b(a)++)' ).
```

Section 12.5 describes the test framework, and how use a predicate such as the above in a test suite.

5.5 Representation of parsed expressions

Entire expressions after a parse are of the form

```
[ex_expr,EXPRESSION]
```

The parameter `EXPRESSION` is the expression body and is of the form

- `[ex_co,char,ASCII_CODE]`
- `[ex_co,int,INTEGER]`
- `[ex_co,real,REAL]`
- `[ex_str,LIST_OF_ASCII_CODES]`
- `[[ex_monadic,OPERATOR],OPERAND1]`
- `[[ex_dyadic,OPERATOR],OPERAND1,OPERAND2]`
- `[[ex_triadic,OPERATOR],OPERAND1,OPERAND2,OPERAND3]`

In the case of an argument list, an operand takes the form of a list of expression bodies. It can be the empty list. In all other cases, an operand is an expression body.

The OPERATOR names are not the sequences that defined the syntax of the operator, but the names as specified in **`op_df`** definitions (refer to Section 4).

The OPERANDS to a function call type operator are the function reference and the argument list where the operands can themselves be expression bodies.

The operands to an array dimension operator are the array reference and the dimension expression.

Notice that the pass-1 tokens for identifier, char, int, real and string, e.g. `[p1_id,IDENTIFIER]`, have been replaced by new expression parser ones, e.g. `[ex_id,IDENTIFIER]`.

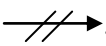
5.6 Expression grammar

Figure 24 shows an expression grammar for C, similar to that given in [C, p.496]. It contains left-recursive rules, and would require transformation before it could be implemented as a Definite Clause Grammar.

Figure 25 shows a slightly more powerful grammar than the grammar for C, but it is still left-recursive. It differs in the following respects:

- It generalises on the monadic suffix operator (rather than only allowing the explicit `++` and `--` operators).
- The `.`*identifier* and `->`*identifier* suffix expressions are widened by allowing `.`*expression* and `->`*expression*, which means that the `.` and `->` operators become ordinary dyadic operators.
- The sizeof operator is subsumed by any function call, which is handled by *expression* and *suffix expression* with the *arglist* operator.

The above grammar is transformed to a non-left-recursive grammar, which is moreover represented in an entirely feed-forward way. It is the expression grammar used by GP4. Various features were introduced:

- Expression grammar rules are parameterised with a precedence level, which is the precedence level of the operators used to combine terms in the grammar rule for the expression at that level.
- Term sequences are also parameterised with an associativity parameter.
- Some small non-grammar operations are performed, indicated by  .
  Examples:
  - to *left associate*, which basically transforms a+b-c+d into [[a+b]-c]+d]
  - to test for a property, such as the parameter ASSOC representing *xfy* (i.e. `[x,f,y]`) associativity.

The transformed grammar is shown in **Figure 26, Figure 27 and Figure 28**. This exercise, carried out from first principles, was the most interesting and challenging part of designing and implementing GP4.

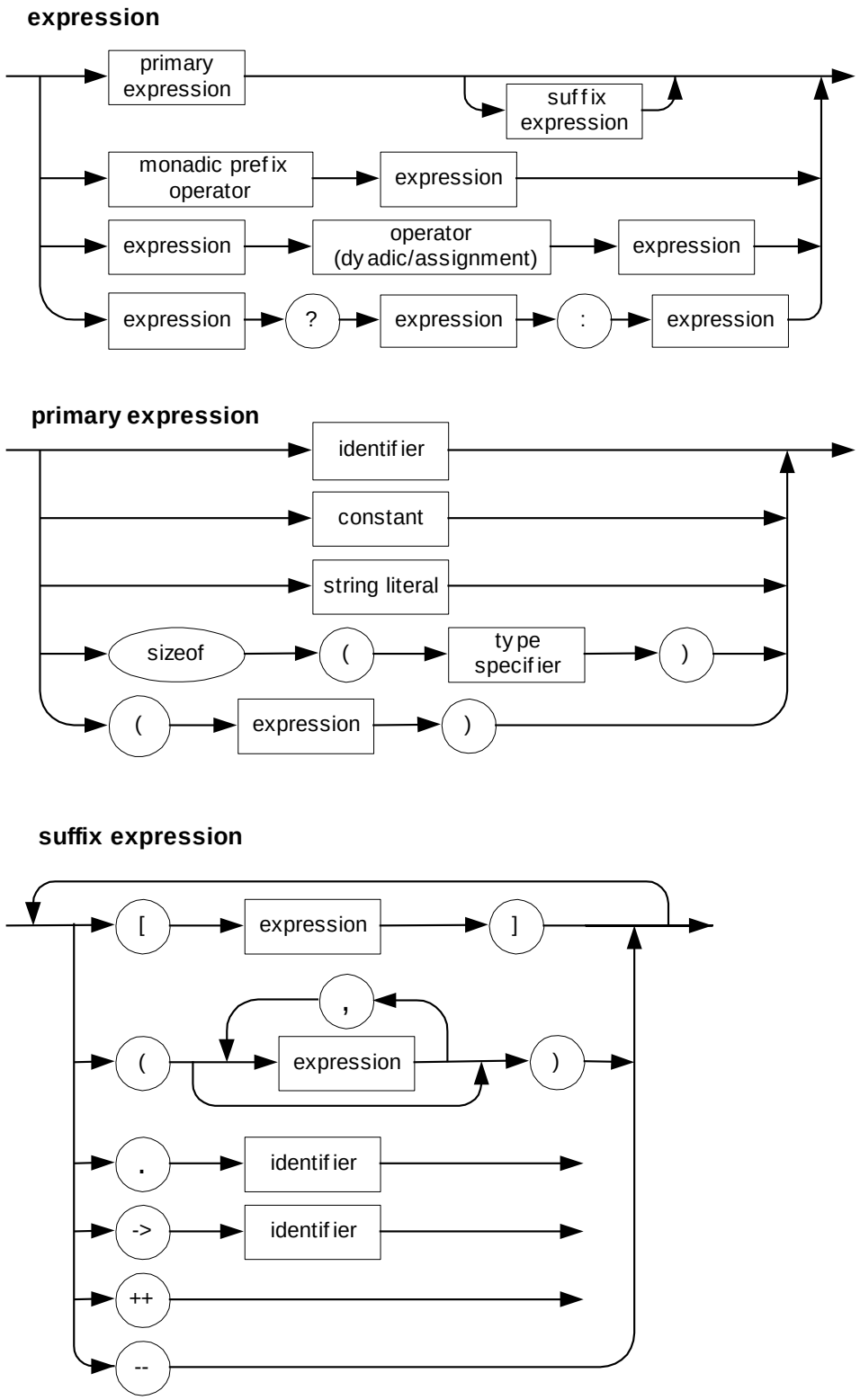


Figure 24. Expressions in C - left recursive specification

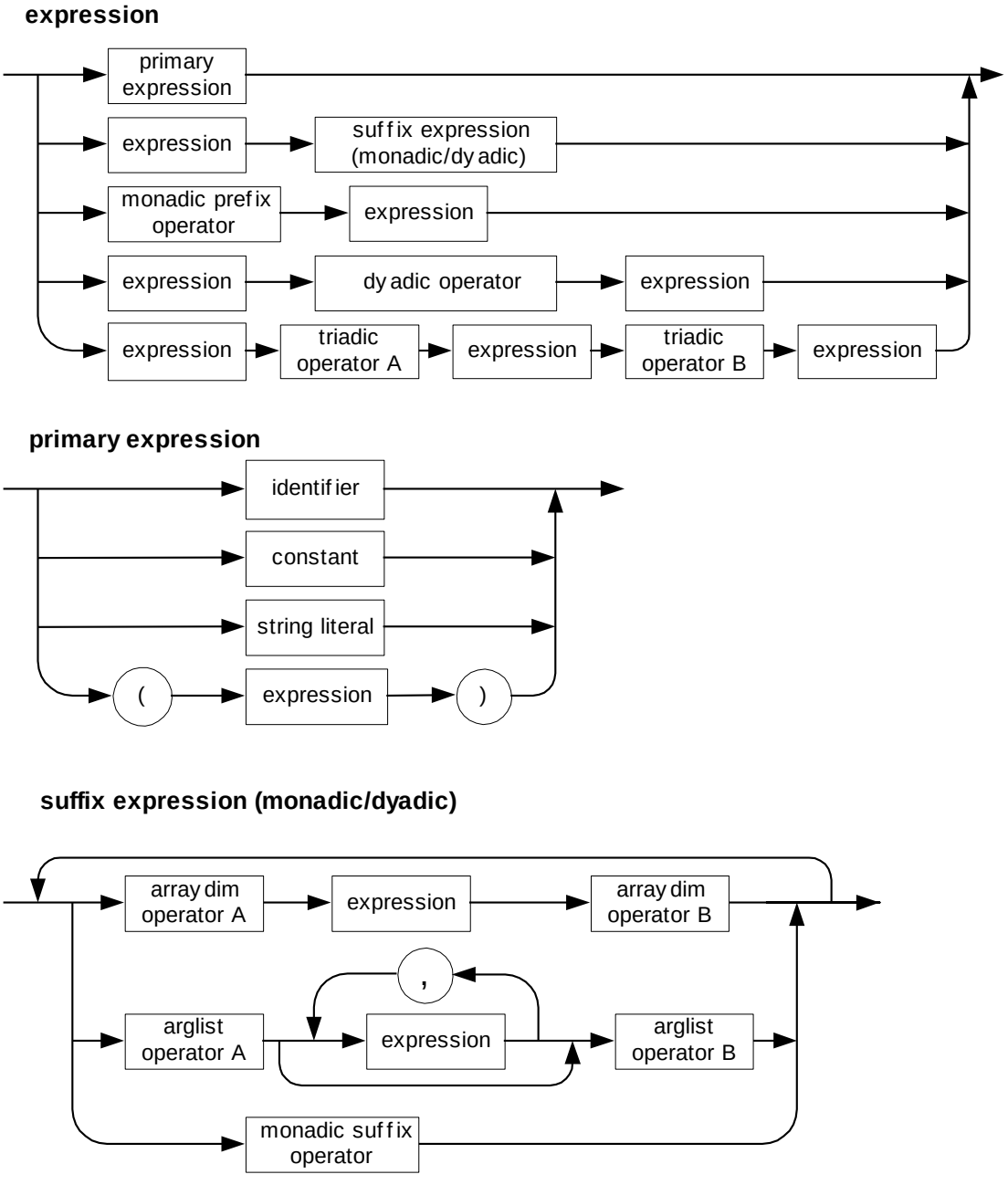


Figure 25. GP4 Expressions - left recursive specification

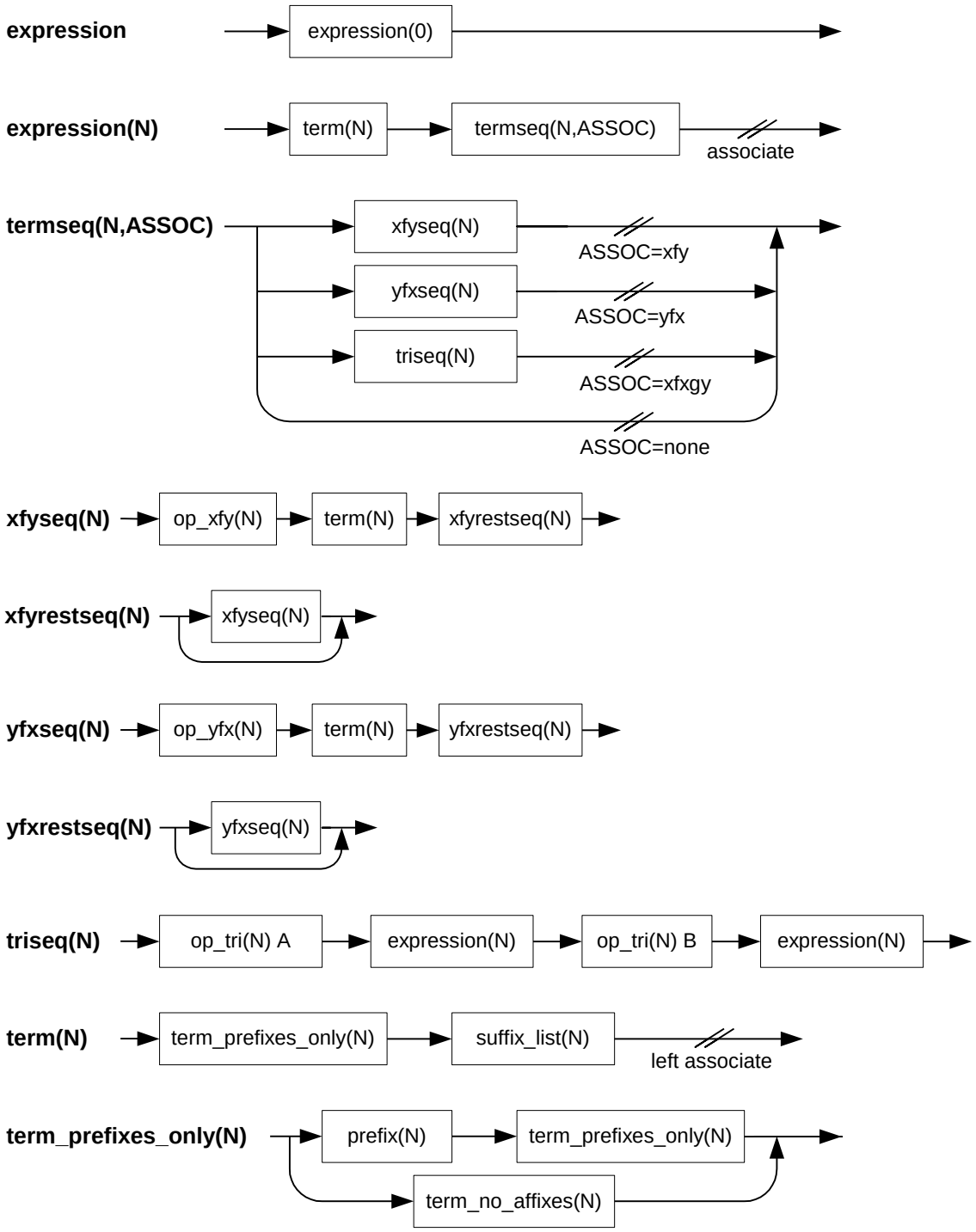
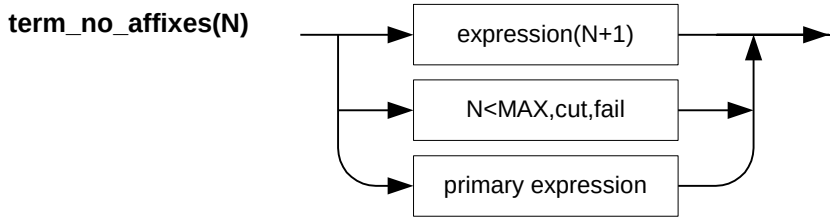


Figure 26. GP4 Expressions - Transformed grammar (1)



*The **cut,fail** combination is reached if the input stream cannot be parsed as expression(N+1).*

*If N=MAX, we ignore the **N<MAX,cut,fail** route and proceed to look for a primary expression in the input stream.*

*If N<MAX, we execute the **cut,fail** combination. This means that the syntactic item term_no_affixes(N) is considered to have failed to parse and no further options for it are to be examined.*

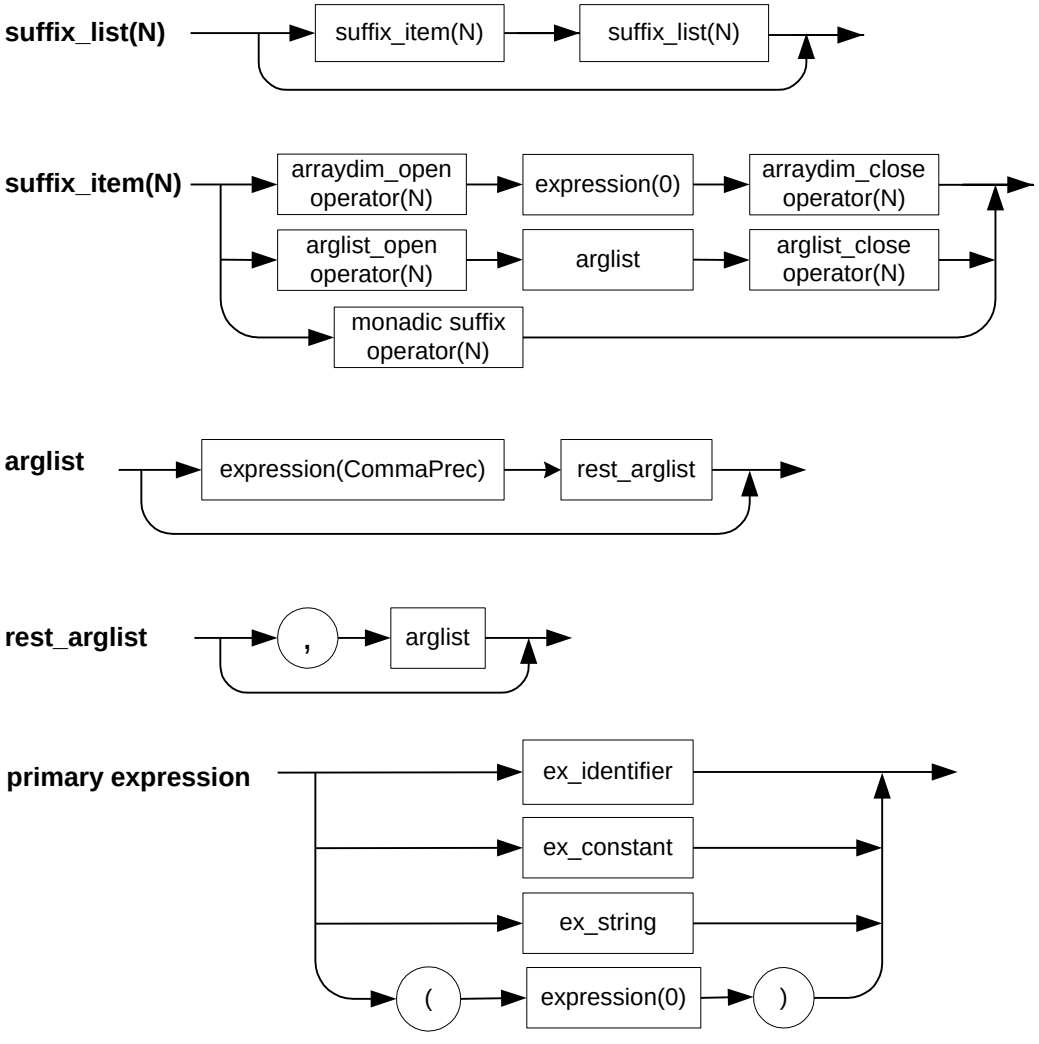


Figure 27. GP4 Expressions - Transformed grammar (2)

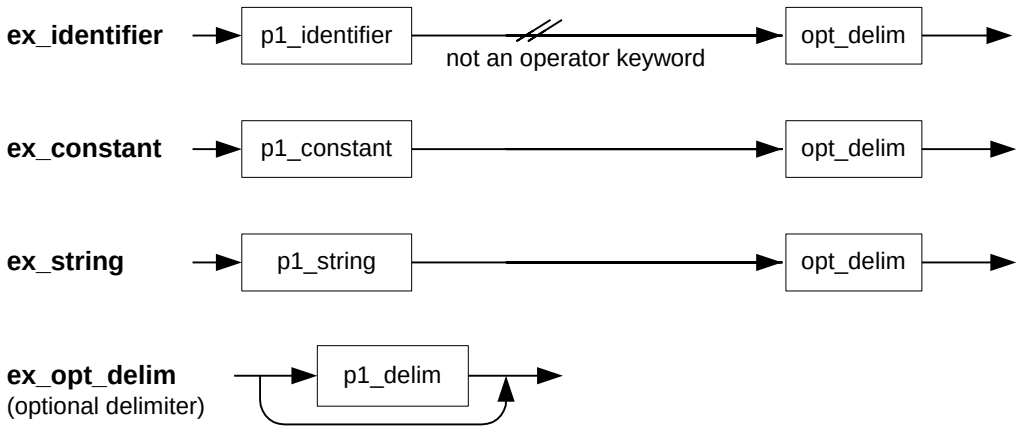


Figure 28. GP4 Expressions - Transformed grammar - (3)

6. Application-specific syntax definition

This module is named "sy_sm" in Figure 3. The _sm suffix stands for an arbitrary application.

This module contains:

- The statement delimiter for the application language being implemented.
- Prolog definite clause grammar (DCG) predicates describing statements
- "Write Summary" predicates that use a parsed statement to generate a one-liner summary so that limited output can be generated.

This information is used by the compiler control module, which itself has no knowledge of application specifics such as the above items.

Statement delimiter definition

The statement delimiter definition is as follows (for delimiter **-:-**):

```
sy_statement_delimiter(X):-
   name('-:-',X).
```

If it is not possible to separate portions of input by a delimiter, a non-ASCII symbol can be used so that the entire file will be read in one go, e.g.

```
sy_statement_delimiter([none]).
```

Syntax definitions

The syntax definitions work with pass-1 tokens, but should reference the following predicates from the expression-parsing module:

- `ex_identifier(OPSETLIST,[_,IDENTIFIER])`, where `IDENTIFIER` can be ground or non-ground.
- `ex_opt_delim(_)`, for optional comment/white-space sequences
- `ex_expr(OPSETLIST,E)`, to parse an entire expression (`E`) using operators in the operator sets given.

Statement parser

The compiler module works independently of any of any particular application, so every statement must be parsed by the same call. This call is

```
sy_statement(STATUS,OBJECT_STATEMENT,P1_STATEMENT,[]).
```

The input parameter is `P1_STATEMENT`. This is the pass-1 output list, consisting of pass-1 output tokens.

The empty list parameter `[]` implies that the entire input string must be used up in doing the parse.

The output parameter is `OBJECT_STATEMENT`. This consists of a nested list, built up by the parsing grammar rules that the user defines.

The `STATUS` parameter tells the compiler whether the parse was successful or not. The values used are

- `g_er` for a parse containing errors
- `g_ok` for a valid parse

A final possibility is that

- the parse call fails (in the Prolog sense)

The compiler takes appropriate action in each of these eventualities (and some additional ones): see Section 8.

A warning

The implementer should be judicious with the use of cuts in defining the syntax. Bear in mind that a cut freezes the "rest-string" parameter and that this may not be desirable under some circumstances.

Example

Statement delimiter

```
sy_statement_delimiter(X):-
   name('-:-',X).
```

Operator set definition (operators in use in expressions)

```
sy_opset([cc,fz]).
```

General statement: `sy_statement(S)`

```
sy_statement(STATUS,S) -->
  ex_opt_delim(_),
  sy_statement_heart(STATUS,S),
  ex_opt_delim(_),
{!}. /* rest-string is bound here, can cut */
```

Repertoire of statements

```
sy_statement_heart(STATUS,CO)--> sy_co(STATUS,CO).  /* constant           */
sy_statement_heart(STATUS,IS)--> sy_is(STATUS,IS).  /* initial survey     */
sy_statement_heart(STATUS,FS)--> sy_fs(STATUS,FS).  /* final survey       */
sy_statement_heart(STATUS,SG)--> sy_sg(STATUS,SG).  /* symptom group      */
```

etc.

A statement syntax example: constant

```
/*-------------------------------------------------------------------------*/
/*                                                                         */
/* BRAND-X SYNTAX for CONSTANT                                             */
/* ===========================                                             */
/*                                                                         */
/* Prefix: sy_co                                                           */
/*                                                                         */
/* <constant statement> ::=                                                */
/*    constant     [<identifier> <expression>]*                            */
/*                                                                         */
/*-------------------------------------------------------------------------*/


/*-----------------------------------------------*/
/* Status ok                                     */
/*-----------------------------------------------*/
sy_co(STATUS,[nw_co,CL]) -->  /* will be called with no restlist */
      {sy_opset(OPSETLIST)},
   ex_identifier(OPSETLIST,[_,constant]),
```



```
      ex_opt_delim(_),
   sy_co_constant_list(STATUS,[_,CL]),
      ex_opt_delim(_).
   /*---------------------------------------*/
   /* No cut, so that if RESTLIST not empty, */
   /* backtracks to any-text error-parse     */
   /*---------------------------------------*/

sy_co_constant_list(STATUS,[nw_co_constant_list,[[I,E]|T]]) -->
      {sy_opset(OPSETLIST)},
   ex_identifier(OPSETLIST,[_,I]),
      ex_opt_delim(_),
   ex_expr(OPSETLIST,E),
      ex_opt_delim(_),
   sy_co_constant_list(STATUS,[_,T]).
   /*---------------------------------------*/
   /* No cut, so that if RESTLIST not empty, */
   /* backtracks to any-text error-parse     */
   /*---------------------------------------*/

sy_co_constant_list(ok,[nw_co_constant_list,[]])        -->
   [].

/*-----------------------------------------------*/
/* Status error                                  */
/*   for junk text in constant list              */
/*-----------------------------------------------*/
sy_co_constant_list(err,
   [nw_co_constant_list,
      ['**Error: constant list: <identifier> <expression> not found']]) -->
   ex_any_text_long(_).
```

7. Application specific data

This module is named "ap_sm" in Figure 3. The _sm suffix is an arbitrary application. The module defines certain texts that are used by the compile module:

- Compiler name
- Copyright text
- File extensions for source, listing and object files.
- Compiler-and-version text as one atom

Example

```
/*-------------------------------------------*/
/* application  definitions                  */
/*-------------------------------------------*/
ap_name('BRAND-X').
ap_copyright('Copyright (C) Philips Electronics N.V, 1999').
ap_version('1.00').
ap_extn(source,dxs).
ap_extn(object,dxo).
ap_extn(listing,dxl).

/*-------------------------------------------*/
/* ap_compiler_line                          */
/* ================                          */
/*   Returns an atom of                      */
/*   COMPILERNAME (Version X.nn)             */
/*-------------------------------------------*/
ap_compiler_line(CL):-
  ap_name(NAM),
  ap_version(VERSION),
  gn_append_atoms(NAM,' COMPILER (Version ',T1),
  gn_append_atoms(T1,VERSION,T2),
  gn_append_atoms(T2,')',CL).
```


8. The compiler control module

This module is named "cp" in Figure 3. The module does not contain any application-specific code or data. It draws on the following application-specific modules (refer to the relevant section for details):

- **`ap_xx`** for application data (e.g. compiler name)
- **`ci_xx`** for compiler verbosity settings
- **`op_xx`** for operator definitions
- **`sy_xx`** for syntax definitions (and also the statement delimiter)

The module works in a read-compile loop, working with a unique statement delimiter. This breaks up the input into separate chunks, which are compiled individually. However, it may not always be practical to do this because the statement delimiter must not occur in any other place such as a comment or string. There is no parse-as-you-read strategy. Under some circumstances, it may be necessary to read the entire source file in one go, perform pass-1 on it, and then maybe offer portions to pass-2 at a time, or to offer everything to pass-2.

The parsing return code of a statement (returns from *sy_statement*) can be the following

- `g_ok` **Success**, i.e. the parse worked and is marked as correct in a returned status parameter.
- `g_er` **Erroneous**, i.e. the statement is recognised as such but it contains syntax errors. It is up to the syntax implementer to provide for erroneous parses, e.g. by using "any-text" predicates to skip over failed parts of a statement. In a Prolog sense, the statement-parsing predicate succeeds with erroneous statements, but only because provision was made to detect erroneous parts of the statement. It is marked as erroneous in a returned status parameter. The compiler then knows to output an error message and to count the error.
- `g_ig` **Ignore**. The compiler should ignore this, except to reproduce it in listings (e.g. statement is null)
- `g_stop` **Stop**. The statement is in irrecoverable error, and compiler should stop, except that *sy_finalize* will still be called
- *`(fail)`* **Failed in the Prolog sense**. This means that the statement not recognised at all. The compiler will output an error message and the Pass-1 output.

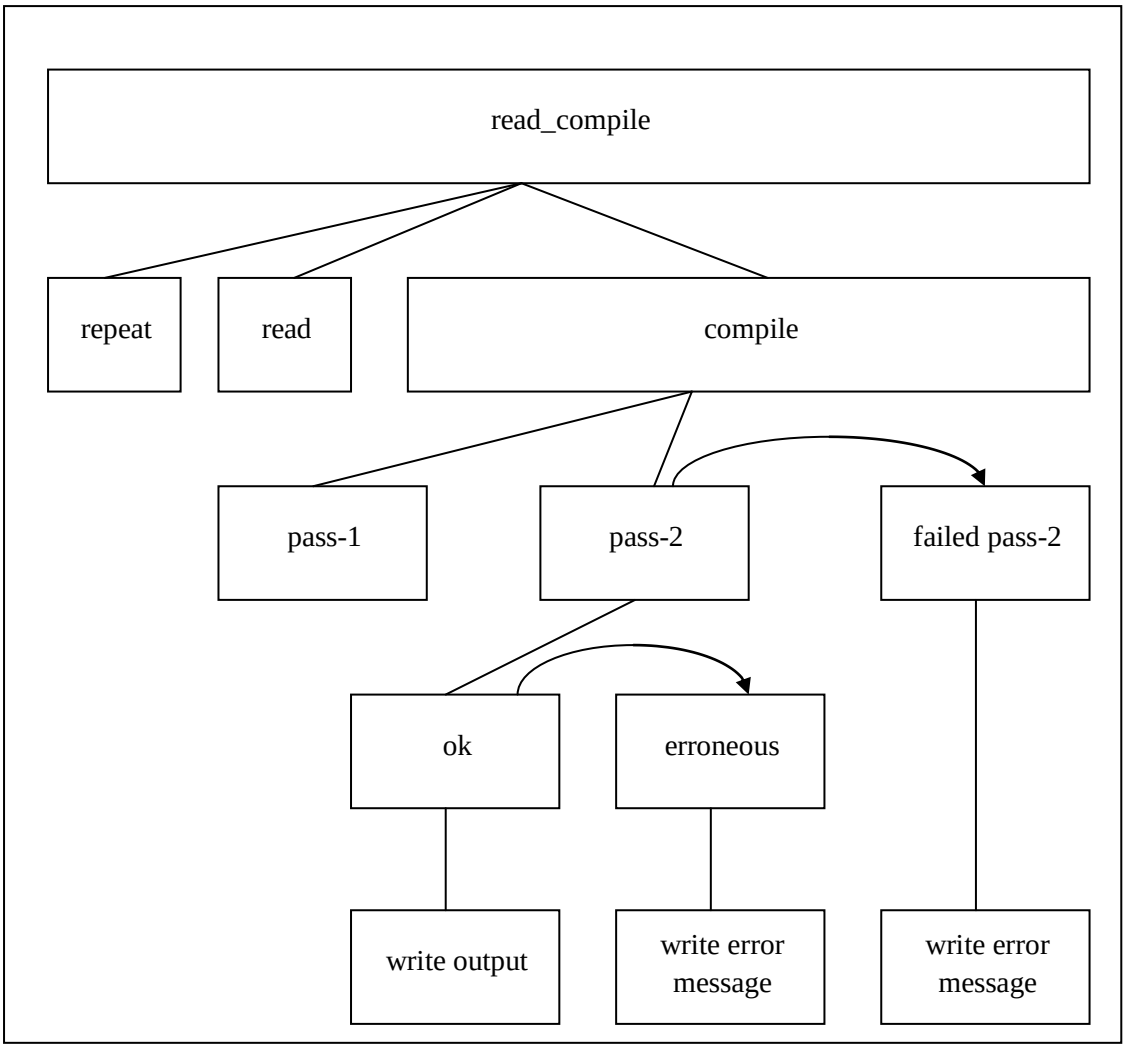


Figure 29. Compilation control in main loop

Figure 30 below shows the structure of the compiler control module in more detail. Shaded calls are those to the application-specific module.

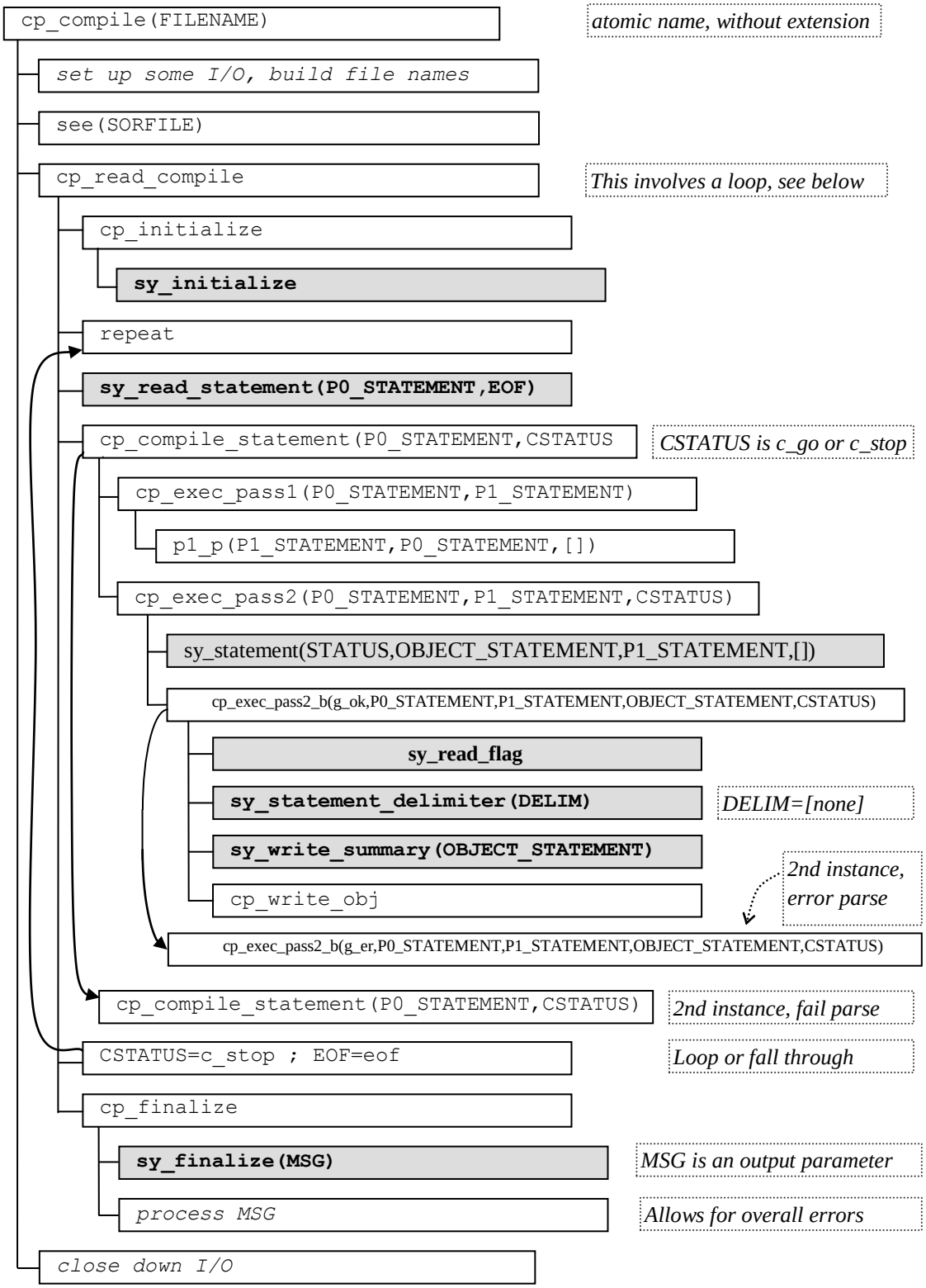


Figure 30. Main features of call graph in `cp` module

Examples of compilation

Source code

```
constant abc 6+4*8+3
         def 2*abc
-:-
  constant ghi 7+5*9+4
           rst ### // junk
           jkl 2*def
-:-
  errorstatement x y+1
-:-
  constant xyz abc+1
```

Object code (Prolog readable lists)

```
/*-------------------------------------------------------------*/
/*  BRAND-X COMPILER (Version 1.00)                            */
/*  Copyright (C) Philips Electronics N.V, 1999                */
/*-------------------------------------------------------------*/
oc_version('1.00').

nw_co(
[[abc,[ex_expr,[[ex_dyadic,dplus],[[ex_dyadic,dplus],[ex_co
,int,6],[[ex_dyadic,xmul],[ex_co,int,4],[ex_co,int,8]]],[
ex_co,int,3]]]],[def,[ex_expr,[[ex_dyadic,xmul],[ex_co,int,
2],[ex_id,abc]]]]]).

/*---------------------------------*/
/*  FAILED STATEMENT HERE          */
/*---------------------------------*/
oc_statement_error(1).

/*---------------------------------*/
/*  FAILED STATEMENT HERE          */
/*---------------------------------*/
oc_statement_error(2).

nw_co(
[[xyz,[ex_expr,[[ex_dyadic,dplus],[ex_id,abc],[ex_co,int,1]
]]]]).

oc_errorcount(2).
```


Verbose Listing

```
+-------------------------------------------------------+
|  BRAND-X COMPILER (Version 1.00)                      |
|  Copyright (C) Philips Electronics N.V, 1999          |
+-------------------------------------------------------+

+-------------------------------------------------------+
|                     STATEMENT                         |
+-------------------------------------------------------+
  constant abc 6+4*8+3
           def 2*abc

-:-
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 1 OUTPUT                       |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
<p1_delim>
[p1_id,constant]
<p1_delim>
[p1_id,abc]
<p1_delim>
[p1_co,int,none,10,6]
+
[p1_co,int,none,10,4]
*
[p1_co,int,none,10,8]
+
[p1_co,int,none,10,3]
<p1_delim>
[p1_id,def]
<p1_delim>
[p1_co,int,none,10,2]
*
[p1_id,abc]
<p1_delim>


+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 2 OUTPUT                       |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
    {nw_co}
            {abc}
            [ex_expr,
                [[ex_dyadic,dplus],
                    [[ex_dyadic,dplus],
                        [ex_co,int,6],
                        [[ex_dyadic,xmul],
                            [ex_co,int,4],
                            [ex_co,int,8]
                        ]
                    ],
                    [ex_co,int,3]
                ]
            ].
            {def}
            [ex_expr,
                [[ex_dyadic,xmul],
                    [ex_co,int,2],
                    [ex_id,abc]
```

```
                ]
            ].


+-----------------------------------------------------+
| *ERROR*     STATEMENT WITH LOCALIZED ERROR(S)       |
+-----------------------------------------------------+
  constant ghi 7+5*9+4
           rst ### // junk
           jkl 2*def

-:-
+- - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 1 OUTPUT                     |
+- - - - - - - - - - - - - - - - - - - - - - - - - - -+
<p1_delim>
[p1_id,constant]
<p1_delim>
[p1_id,ghi]
<p1_delim>
[p1_co,int,none,10,7]
+
[p1_co,int,none,10,5]
*
[p1_co,int,none,10,9]
+
[p1_co,int,none,10,4]
<p1_delim>
[p1_id,rst]
<p1_delim>
#
#
#
<p1_delim>
[p1_id,jkl]
<p1_delim>
[p1_co,int,none,10,2]
*
[p1_id,def]
<p1_delim>

+- - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 2 OUTPUT                     |
+- - - - - - - - - - - - - - - - - - - - - - - - - - -+
    {nw_co}
            {ghi}
            [ex_expr,
                [[ex_dyadic,dplus],
                    [[ex_dyadic,dplus],
                        [ex_co,int,7],
                        [[ex_dyadic,xmul],
                            [ex_co,int,5],
                            [ex_co,int,9]
                        ]
                    ],
                    [ex_co,int,4]
                ]
            ].
        {**Error: constant list: <identifier> <expression> not found}
```



```
+-------------------------------------------------------+
| *ERROR*         FAILED STATEMENT                      |
+-------------------------------------------------------+
  errorstatement x y+1

+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 1 OUTPUT                       |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
<p1_delim>
[p1_id,errorstatement]
<p1_delim>
[p1_id,x]
<p1_delim>
[p1_id,y]
+
[p1_co,int,none,10,1]
<p1_delim>


+-------------------------------------------------------+
|                     STATEMENT                         |
+-------------------------------------------------------+
  constant xyz abc+1

+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 1 OUTPUT                       |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
<p1_delim>
[p1_id,constant]
<p1_delim>
[p1_id,xyz]
<p1_delim>
[p1_id,abc]
+
[p1_co,int,none,10,1]
<p1_delim>


+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
|                   PASS 2 OUTPUT                       |
+- - - - - - - - - - - - - - - - - - - - - - - - - - - -+
    {nw_co}
            {xyz}
            [ex_expr,
                [[ex_dyadic,dplus],
                    [ex_id,abc],
                    [ex_co,int,1]
                ]
            ].

+-------------------------------------------------------+
|  BRAND-X:  COMPILATION COMPLETE -   2 ERROR(S)        |
+-------------------------------------------------------+
```

9. The command interpreter module

This module is a placeholder for a graphical user interface. A mature commercial product under Windows would probably require a graphical user interface for marketing reasons. But for a prototyping tool that needs flexibility in the face of changes, building a graphical user interface may be an expensive luxury. For the time being, the settings that a graphical user interface would acquire are simply defined in this module.

The ci_compile_option predicates determine the quantity of output (and indeed, whether any output is generated at all).

Compiler option settings

The following settings control the verbosity level of output in the listing file, on the screen and in the "object code" (i.e. parse output) file, respectively.

```
ci_compile_option(listing,LEVEL).
ci_compile_option(user,LEVEL).
ci_compile_option(object,LEVEL3).
```

Verbosity levels

The numerical level parameter is a verbosity level, used as follows:

Quantity of output for user/listing

5 =	global-summary	failure-detail	statement	pass-1-detail	pass-2-detail
4 =	global-summary	failure-detail	statement	pass-1-detail	
3 =	global-summary	failure-detail	statement		
2 =	global-summary	failure-detail	statement-summary		
1 =	global-summary				
0 =	none				

Quantity of output for "object" code (Prolog-readable output)

5 =	failure-count	failure-indication	OBJECT-CODE
4 =	failure-count	failure-indication	OBJECT-CODE
3 =	failure-count	failure-indication	OBJECT-CODE
2 =	failure-count	failure-indication	
1 =	failure-count		
0 =	none		

Note that a level of at least 3 is needed to obtain object code.

Example definitions

```
ci_compile_option(listing,5).
ci_compile_option(user,5).
ci_compile_option(object,3).
```

10. Expression evaluation

10.1 Introduction to the evaluation module

The essence of the expression evaluator is to take an argument such as

```
[[ex_dyadic,dplus],P1,P2]
```

which represents the dyadic add operator, to evaluate the two operands (by recursive application of the expression evaluator), and combine these evaluated operands by the operation being handled.

Standard implementations of the commonest operators have been implemented. Tri-valued logic has been used (true/false/unknown). Some arithmetic operators have been overloaded to provide string manipulation functions. Although this is not standard C practice, it provides a convenient basis for many a prototype language.

The following figure illustrates the evaluation process.

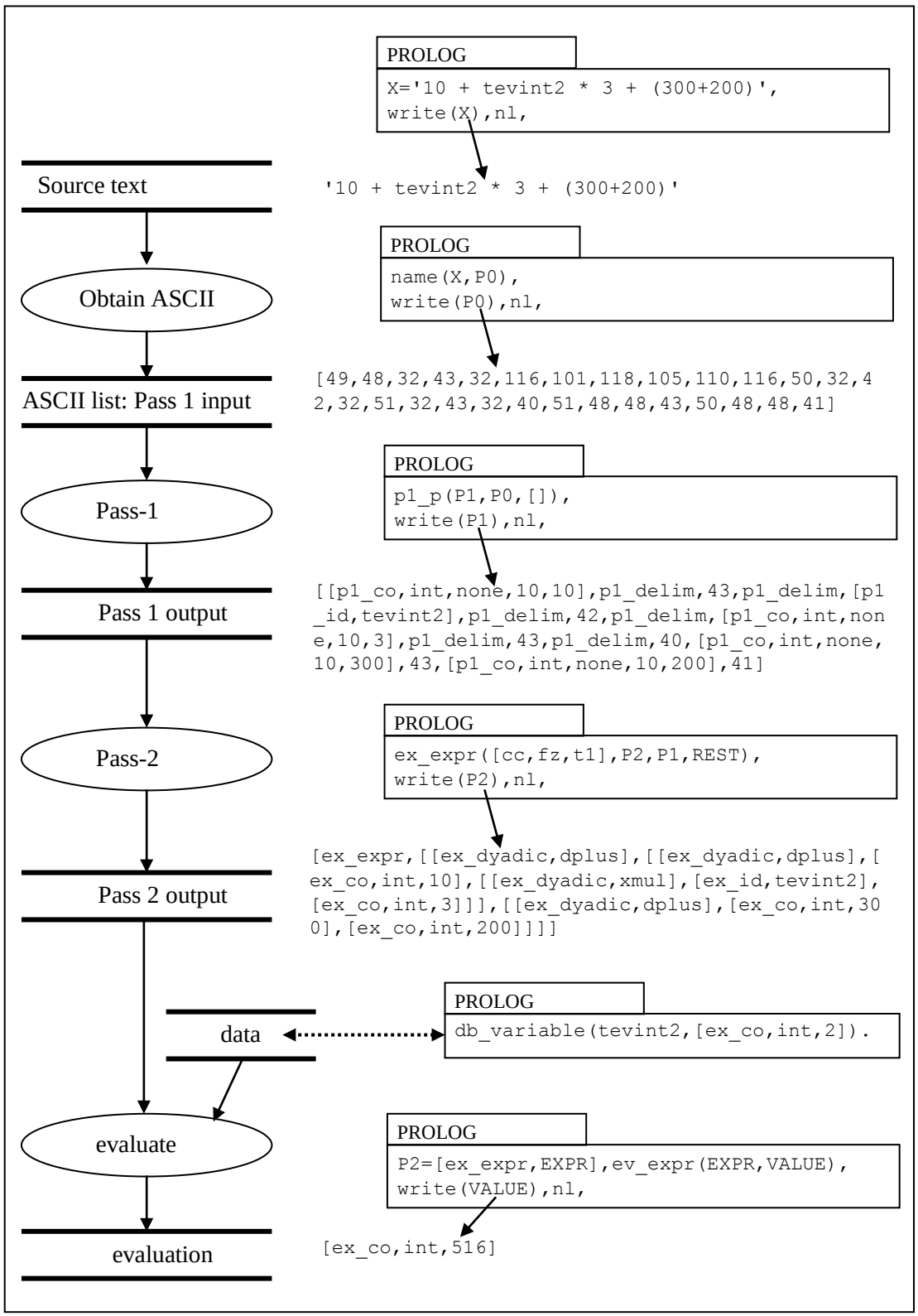


Figure 31. Evaluation call

10.2 Example of an evaluation call

The following example is basically the same as the one in Figure 31, but it contains some extra features:

- pretty-print of the expression
- reconstruction of the algebraic expression as a bracketed expression.
- explicit output of the "reststring", i.e. the unused tail of the parser input list, if any.

Data

```
db_variable(tevint2  ,[ex_co,int,2]).
```

Driver predicate

```
te(X):- /* PARSES AND EVALUATES AN EXPRESSION */
   nl,
   write('   X= '),           write(X),nl,
   name(X,P0),                write('  P0= '),write(P0),nl,
   p1_p(P1,P0,[]),            write('  P1= '),write(P1),nl,
   ex_expr([cc,fz,t1],P2,P1,REST), write('  P2= '),write(P2),nl,
                              write('REST= '),write(REST),nl,nl,
   ex_pp(ppp,P2,_),nl,
   write('RECONSTRUCTION='),ex_abr([cc,fz,t1],P2,OUT),
       write(''),write(OUT),write(''),nl,
   P2=[ex_expr,EXPR],
   ev_expr(EXPR,VALUE),
   write('VALUE='),write(VALUE),nl.
```

Prolog Query

```
| ?- te('10 + tevint2 * 3 + (300+200)').
```

Output

```
   X= 10 + tevint2 * 3 + (300+200)
  P0= [49,48,32,43,32,116,101,118,105,110,116,50,32,42,32,51,32,43,
       32,40,51,48,48,43,50,48,48,41]
  P1= [[p1_co,int,none,10,10],p1_delim,43,p1_delim,[p1_id,tevint2],
       p1_delim,42,p1_delim,[p1_co,int,none,10,3],p1_delim,43,p1_delim,40,
       [p1_co,int,none,10,300],43,[p1_co,int,none,10,200],41]
  P2= [ex_expr,[[ex_dyadic,dplus],[[ex_dyadic,dplus],[ex_co,int,10],
       [[ex_dyadic,xmul],[ex_id,tevint2],[ex_co,int,3]]],[[ex_dyadic,dplus],
       [ex_co,int,300],[ex_co,int,200]]]]
REST= []

[ex_expr,
    [[ex_dyadic,dplus],
        [[ex_dyadic,dplus],
            [ex_co,int,10],
            [[ex_dyadic,xmul],
                [ex_id,tevint2],
                [ex_co,int,3]
            ]
        ],
        [[ex_dyadic,dplus],
            [ex_co,int,300],
```



```
            [ex_co,int,200]
        ]
    ]
].

RECONSTRUCTION=(10+(tevint2*3))+(300+200)
VALUE=[ex_co,int,516]
```

Desciption of the evaluation predicate `ev_expr`

The predicate `ev_expr` takes as input argument the `ARG` of `[ex_expr,ARG]` which the expression parser returns. Section 5.5 lists these representations of expressions. We give some examples below:

Nonterminal items

- `[[ex_monadic,mminus],P1]` monadic
- `[[ex_dyadic,dplus],P1,P2]` dyadic
- `[[ex_dyadic,fcall],P1,PLIST]` dyadic, second argument is a list
- `[[ex_triadic,aif],P1,P2,P3]` triadic

Terminal items for an operand can be:

- `[ex_co,int,INTEGER]` e.g. `[ex_co,int,39]`
- `[ex_co,char,INTEGER]` e.g. `[ex_co,char,39]`
- `[ex_co,real,REAL]` e.g. `[ex_co,real,39.9]`
- `[ex_str,LIST]` e.g. `[ex_str,[61,62]]`
- `unknown`
- `[ex_id,IDENTIFIER]`

where the value of `IDENTIFIER` can be found from:

- `db_fixed_constant(IDENTIFIER,VALUE)`
- `db_model_constant(IDENTIFIER,VALUE)`
- `db_variable(IDENTIFIER,VALUE)`

Illegal arguments to operators (operands) give a result of 'unknown'

The output parameter is an item of type

- `[ex_co,int,INTEGER]`
- `[ex_co,char,INTEGER]`
- `[ex_co,real,REAL]`
- `[ex_str,LIST]`
- `unknown`

10.3 Operators implemented for evaluation

The following tables show the result for various kinds of operands. A specific value or type preceding a generic type takes precedence, e.g. where the "recip" operator takes a numeric parameter, read this as "other than 0". A "numeric" argument can be of type char, int or real. An argument can also be a string, or "unknown".

A numeric result from two numeric arguments will generally be of the most generic type (real is more general than int, which is more general than char).

Reminder: Results are wrapped, e.g. of form `[ex_co,TYPE,VALUE],` not just a plain `VALUE`.

Tables of operators follow.

Operator	Input Param 1 Type	Result
mplus *monadic plus*	numeric string unknown	identity operation identity operation identity operation
mminus *monadic minus*	numeric string unknown	negates reverses unknown
recip *reciprocal*	0 numeric string unknown	unknown real unknown unknown
lnot *logical not*	numeric OTHER	numeric int, by logical not unknown
fnot *fuzzy not*	numeric: 0..1 OTHER	numeric real 0..1, by fuzzy not unknown

Table 5. Monadic operators

Operator	Input Param 1 Type	Input Param 2 Type	Result
dplus *dyadic plus*	numeric string numeric string OTHER	numeric string string numeric OTHER	numeric, by addition string, concatenates string, concatenates a character string, concatenates a character unknown

dminus *dyadic minus*	numeric string numeric string OTHER	numeric string string numeric OTHER	numeric, by subtraction string, eliminates 1st substring P2 else =P1 string, eliminates 1st substring P2 else =P1as string string, eliminates 1st substring P2 else =P1 unknown
xmul *multiply*	numeric string numeric string OTHER	numeric string string numeric OTHER	numeric, by multiplication unknown string, reduplicated n times string, reduplicated n times unknown
xdiv *divide*	numeric: ANY real ANY int int char char OTHER	numeric: 0 (any form) ANY real int char int char OTHER	 unknown numeric, real (by REAL division) numeric, real (by REAL division) numeric, int (by INTEGER division) numeric, int (by INTEGER division) numeric, int (by INTEGER division) numeric, char (by INTEGER division) unknown
mod *modulo*	numeric: ANY real ANY int int char char OTHER	numeric: 0 (any form) ANY real int char int char OTHER	 unknown numeric, int (rounds the real, INTEGER division) numeric, int (rounds the real, INTEGER division) numeric, int (by INTEGER division) numeric, int (by INTEGER division) numeric, int (by INTEGER division) numeric, char (by INTEGER division) unknown
pwr *raise to the power*	numeric: real ANY int int char char OTHER	numeric: ANY real int char int char OTHER	numeric: real real int int int char unknown
eq *equality*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type int, =0 always int, =0 always int, 0 or 1. Case sensitive.
ne *inequality*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type int, =1 always int, =1 always int, 0 or 1. Case sensitive.
ge *greater than or equal*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type unknown unknown int, 0 or 1. Case sensitive, ASCII order. "A" >= "AB".

le *less than or equal*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type unknown unknown int, 0 or 1. Case sensitive, ASCII order. "AB" <= "A".
gt *greater than*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type unknown unknown int, 0 or 1. Case sensitive, ASCII order. "A" > "AB".
lt *less than*	numeric numeric string string	numeric string numeric string	int, 0 or 1. NB: operands need not be of same type unknown unknown int, 0 or 1. Case sensitive, ASCII order. "AB" < "A".
land *logical and* *(short-circuit)*	ANY 0 numeric OTHER	0 ANY numeric OTHER	int, 0 int, 0 int, 0 or 1 unknown
lior *logical inclusive or* *(short-circuit)*	ANY numeric≠0 numeric OTHER	numeric≠0 ANY numeric OTHER	int, 1 int, 1 int, 0 or 1 unknown
lxor *logical exclusive or*	numeric OTHER	numeric OTHER	int, 0 or 1 unknown
leqv *logical equivalence*	numeric OTHER	numeric OTHER	int, 0 or 1 unknown
fand *fuzzy and*	ANY 0 0..1 OTHER	0 ANY 0..1 OTHER	real, 0 real, 0 real, 0..1 Formula: P1*P2 unknown
fior *fuzzy inclusive or*	ANY 1 0..1 OTHER	1 ANY 0..1 OTHER	real, 1 real, 0 real 0..1 Formula:1- (1-P1)*(1-P2) unknown
fxor *fuzzy exclusive or*	0..1 OTHER	0..1 OTHER	real 0..1 Formula: (P1fand(fnot P2))+(P2 fand(fnot P1)) unknown
feqv *fuzzy equivalence*	0..1 OTHER	0..1 OTHER	real 0..1 Formula: (P1 fand P2)+((fnot Y)fand(fnot X)) unknown
boost *multiply up odds*	1 0 0..1 OTHER	ANY ANY numeric pos. OTHER	real 1 real 0 real 0..1 unknown
depress *divide down odds*	1 0 0..1 OTHER	ANY ANY numeric pos. OTHER	real 1 real 0 real 0..1 unknown

Table 6. Dyadic operators

11. Function calls

11.1 How functions are called

The parse of a function call is as follows (an example)

Source

```
format(0, 1+2, x)
```

Parse

```
[ex_expr,
    [[ex_dyadic,fcall],
        [ex_id,format],
        [
            [ex_co,int,0],
            [[ex_dyadic,dplus],
                [ex_co,int,1],
                [ex_co,int,2]
            ],
            [ex_id,x]
        ]
    ]
].
```

The expression evaluator handles the function call operator by evaluating the function parameters and calling a Prolog function with a user-specified implementation name, which is always of signature

```
somefunction(RETURNVALUE,PARAMETERLIST)
```

The parameter list can be the empty set. Otherwise, the elements it contains should be of the return types

- `[ex_co,int,INTEGER]`
- `[ex_co,char,INTEGER]`
- `[ex_co,real,REAL]`
- `[ex_str,LIST]`
- `unknown`

The return value should also be one of these types.

Example:

To implement

```
foo(X,Y,Z)
```

the evaluation of the function name, which is just `[ex_id,foo],` must yield e.g.

```
[fu_fname,fi_foo]
```

This is achieved if there is a predicate somewhere of:

```
fu_function(foo,[fu_fname,fi_foo]).
```

There must be no name clash between function names and variables, so `foo` must not also be a variable.

The implementor writes

```
fi_foo(RETURNVALUE,[X,Y,Z]):-...
```

The name "`fi_foo`" is arbitrary, as long as it is used consistently, and could be just "`foo`", but a convention to avoid accidental name clashes is to use the prefix `fi_`, giving `fi_foo`.

11.2 Functions implemented

For more detailed explanation of a function, refer to the source code. This table serves to indicate what is available.

Function	Input Param 1 Type	Input Additional Parameters Type	Result	Notes
abs *absolute value*	numeric OTHER		numeric unknown	same type as input
maximum *maximum of several (1 or more)*	numeric OTHER	numeric OTHER	numeric unknown	type is same as of the maximum
minimum *minimum of several (1 or more)*	numeric OTHER	numeric OTHER	numeric unknown	type is same as of the minimum
round *round to nearest integer*	numeric OTHER		numeric, int unknown	
round_down *round down*	numeric OTHER		numeric, int unknown	rounds towards -∞, not necessarily towards zero
round_up *round up*	numeric OTHER		numeric, int unknown	rounds towards +∞, not necessarily away from zero
exp *e to the power*	numeric OTHER		numeric, real unknown	
exp10 *10 to the power*	numeric OTHER		numeric, real unknown	

ln *natural logarithm*	numeric OTHER		numeric, real unknown	
log *log base 10*	numeric OTHER		numeric, real unknown	
member *membership*	numeric OTHER	numeric (4x) OTHER	numeric, real unknown	P2,P3,P4,P5 define a trapezium
bayes *bayesian updating*	numeric OTHER	numeric OTHER	numeric, real unknown	P1=Prior probability of hypothesis P2, P3, P4 = evidence data = Prior, Sufficiency, Necessity Similarly additional triplets
format (I) *format an integer into a string*	numeric OTHER	numeric, int unknown	string unknown	P2=field width -ve = left justify 0 = just justify +ve = right justify
format (II) *format a real into a string*	numeric OTHER	numeric, int	string unknown	P2=field width, as above P3=number of decimals (truncated)
length *length of a string*	string OTHER		numeric, int unknown	
upper_case *convert string to upper case*	string OTHER		string unknown	
lower_case *convert string to lower case*	string OTHER		string unknown	

Table 7. Function calls

12. The library modules

The library modules described in this section are Prolog modules that contain predicates that could be of use in *any* application, (not just parsing and language prototyping).

The purpose of the following tables is not to give exact details, but to show what is available. The tables are not exhaustive of all predicates defined in the modules; they rather cover those that should have external visibility and could be of general use.

12.1 Module "aa" (System Dependent)

The purpose of this module is that all code that is dependent on a particular system of Prolog should be in this module. Also, any code that is dependent on the operating system should also be in this module. The current Prolog system is WinProlog [WinPro]. The current operating system is Windows-NT. Should a port be needed to other systems, then ideally only this module need be examined.

The main areas of system dependency are

- i/o
- operations on reals, mathematical functions

The convention in this module is to prefix predicate names with the prefix that they would naturally take if they were in their "native" module: ar=arithmetic, gn=general, io=input/output etc.

The arithmetic modules work with unwrapped Prolog data, i.e. just Prolog integers, reals etc. This is in contrast to the GP4 operators and functions, which work with GP4-wrapped data items, (i.e. [ex_co, real,REAL] etc.). Arithmetic routines with invalid parameters normally fail (they do not return *unknown*). This applies to e.g. the logarithm of a negative number.

Predicate	Function
io_eof(X)	Defines the character that is returned by get0(X) at END OF FILE
ar_number(X)	Succeeds if its argument is an integer or real
ar_mod(X,Y,Z)	Arguments Y and Z must be of integral type Z:= X modulo Y X mod -Y = +(X mod Y)

ar_div(X,Y,Z)	Arguments Y and Z must be of integral type Z:= X/Y in integer arithmetic X div -Y = -(X div Y) The following always holds: (A div B) * B + (A mod B) = A
ar_real(X)	Succeeds if its argument is a real (but not an integer)
ar_round(X,Y)	X must be a real Y:=nearest integer to X
ar_round_down(X,Y)	X must be a real Y:=first integer <= X. Rounds towards $-\infty$, not towards zero
ar_round_up(X,Y)	X must be a real Y:=first integer >= X. Rounds towards $+\infty$, not away from zero
ar_power(X,Y,Z)	X must be a positive real $Z:=X^Y$

Table 8. Table of predicates in module "aa"

Additional predicates that are/may also be system dependent, but which are not housed in module "aa".

- `gn_bag2set1(BAG,SET)`. We keep this next to its companion `gn_bag2set2`.
- The use of `name(REAL)` in `fi_format`. Behaviour in other systems has not been investigated.

12.2 Module "ar" (Arithmetic)

Many arithmetic predicates are Prolog-system dependent, and so are housed in module "aa" (Section 12.1).

Predicate	Function
ar_raise(A,B,C)	C:= A^B. B must be an integer. This predicate is used to build the value of a constant (integer and real) from its ASCII string in pass-1 parsing.
ar_for(N,M,X)	like BASIC **`FOR X = N TO M`**, or C **`for (x=n; x<=m; x++) {}`** `N and M must be integral`
ar_next_integer(N)	Produces integers 1,2,3 .... infinity.

Table 9. Table of predicates in module "ar"

12.3 Module "gn" (General)

Some of these predicates are in-built into most Prolog Systems, but for portability, standard implementations have been included in the **gn** module. In the case of **gn_not**, the semantics may be different to the in-built semantics of some systems.

A number of these predicates are well-known techniques from the literature. A primary source is [Clocksin].

Predicate	Function
gn_member(M,LIST)	Membership of a list [Clocksin, p.55]
gn_append(L1,L2,L3)	Append lists [Clocksin, p.63]
gn_append_atoms(A1,A2,A3)	Append atoms, concatenating direction only
gn_not(P)	Succeeds iff a call to P fails. [Clocksin, p.87] The semantics may be different to the in-built semantics of some systems. A common application is proving non-membership: `gn_not((gn_member(M,LIST))).`
gn_asserta(X)	Same as standard `asserta(X),` but guarantees a cut after it.
gn_assertz(X)	Same as standard `assertz(X),` but guarantees a cut after it.
gn_retract(X)	Same as standard `retract(X),` but guarantees a cut after it.
gn_retractall(X)	Retracts all matching predicates. [Clocksin, p.179]
gn_length_list(LIST,LEN)	Returns the length of a list
gn_revzap(L1,L3)	Reverses a list efficiently [Clocksin p.150]
gn_delete(X,L,M).	Delete all occurrences of element X in list L to produce list M. [Clocksin, p.141]
gn_last(LIST, LONGHEAD,SHORTTAIL).	Split list into long-head (a list) and short-tail (last element) Also reverse-drivable to join up a list.
gn_call_list(LIST)	Call each predicate in LIST
gn_duplicate(S,N,SSS)	Duplicate a list S, N times, (as one equally flat list).
gn_sublist(X,Y)	Succeeds if X is a sublist in Y [Clocksin p.151]
gn_bag2set(BAG,SET)	Convert bag to set (eliminate duplicates) implemented using gn_bag2set1 or gn_bag2set2
gn_bag2set1(BAG,SET)	Convert bag to set using WinProlog in-built predicate. Sorts alphabetically.
gn_bag2set2(BAG,SET)	Convert bag to set from first principles, removes rightmost duplications
gn_make_set(BAG,SET)	Convert bag to set from first principles, removes rightmost duplications *This is an alternative to the above. A performance comparison has not yet been carried out.*

gn_findall(X,G,L)	Constructs a list L of all the objects X such that the goal G is satisfied. Recursive find (i.e. in G) is supported. [Clocksin, p.163]
gn_insert(X,OLDLIST,NEWLIST, COMPARATOR)	Insert an element into a sorted list, using a supplied reference to a comparator predicate.
gn_merge_sort(INLIST,OUTLIST, COMPARATOR)	Sort a list. It seems to be quite efficient. COMPARATOR can be: ▪ for numbers, a straight operator, e.g. >` ▪ for strings, a predicate, e.g. `gn_less_string` ▪ for atoms, a predicate, e.g. `gn_less_atom` ▪ user defined
gn_univ(X,Y)	Recursive application of the univ (=..) operator. N.B. Not designed for functors taking 3 or more arguments

Table 10. Table of general predicates

12.4 Module "io" (Input/Output)

Some of these routines perform i/o, while others support i/o operations without actually performing any i/o.

Predicate	Function
Input	
io_read_line(LINE)	reads up to a line feed (ASCII code 10) and returns the list
io_read_line(LINE,EOF)	reads up to a line feed (ASCII code 10) and returns the list gives end-of-file indication =eof when end of file read in the line =ok when no end of file is read in the line
io_read_in(STATEMENT,DELIM,EOF)	statement read with arbitrary look ahead to statement delimiter. P1 (output): statement read, as a list P2 (input): The delimiter, as a list, e.g. from name('-:-',DELIM) P3 (output): End-of-file indication =ok The current statement is not followed by an end-of-file =eof The current statement is followed by an end of file
low level routines	
io_wr_oc	write Prolog-style open-comment: /* since write('/*') may be interpreted as start comment after *write('

io_wr_cc	write Prolog-style close-comment: * / since write('*/') may be interpreted as end comment after *write('*
io_writeq(X)	write quoted
io_write_repeat(N,X)	write X N times.
io_tab_sp(X)	tab X spaces, with protection against negative argument
io_length_atom(ATOM,LEN)	length of an atom
io_length_int(INT,LEN)	length of an integer
writing a single item	
io_write_atom(ATOM,FIELD,J)	write an atom in a field ATOM: atom to write FIELD: field length (including sign) J r=right justified, l=left justified
io_write_int(INT,FIELD,J)	write an integer in a field INT: integer to write FIELD: field length (including sign) J r=right justified, l=left justified
io_write_real(REAL,[A,B])	write a real in format [A,B] 1 place for sign (can be used as extra place for digit) A places for integer part 1 place for decimal point B places for decimal part
Writing a list	
io_wlist(LIST)	write each element of a list
io_wlistc(LIST)	write list commented, i.e. /* list items */
io_wlist_nl(LIST)	write elements of a list on new lines
io_put_list(LIST)	put elements of a list
io_put_list_limited(N,LIST)	put initial elements of a list in a limited field of size N
io_pp(X)	pretty print of a list. Based on [Clocksin, p.97]
io_long_list(X)	write long list in limited width (ugly print) uses io_long_list_width to define output width (default =60).
io_long_tail(X)	similar to `io_long_list(X)` but without outermost brackets.

logging	
io_log_setlevel(X)	setting of threshold Level at and below which calls io_log are processed. Guide: 0 No logging 1 Only very high level messages 5 Moderate detail 9 Maximum detail
io_log(LEVEL,MESSAGE,DATA)	logging call LEVEL: Level (must be at or below threshold to appear) MESSAGE: The message to be written DATA: Extra data to go with message

Table 11. Module "io"

12.5 Permutation and tree walking

Permutations

We do not necessarily generate all permutations. An [n,k] indication means that when permuting a list of n elements, *any* k embedded elements will exhibit all k! ordering of themselves somewhere in the permuted orderings of the original list. We show the permutations of `[a,b,c,d]` as generated.

- **`gn_permute_k1`** The [n,1] solution: no permutations - just the original
  `[a,b,c,d]`

- **`gn_permute_k2`** The [n,2] solution: forwards and backwards
  `[a,b,c,d],[d,c,b,a]`

- **`gn_permute_k3a`** An [n,3] solution: 2n permutations
  `[a,b,c,d],[b,c,d,a],[c,d,a,b],[d,a,b,c],[d,c,b,a],[c,b,a,d],[b,a,d,c],[a,d,c,b]`

- **`gn_permute_1`** Full permutations - algorithm 1
  `[a,b,c,d],[a,b,d,c],[a,c,b,d],[a,c,d,b],[a,d,b,c],[a,d,c,b],[b,a,c,d],[b,a,d,c],`
  `[b,c,a,d],[b,c,d,a],[b,d,a,c],[b,d,c,a],[c,a,b,d],[c,a,d,b],[c,b,a,d],[c,b,d,a],`
  `[c,d,a,b],[c,d,b,a],[d,a,b,c],[d,a,c,b],[d,b,a,c],[d,b,c,a],[d,c,a,b],[d,c,b,a]`

- **`gn_permute_2`** Full permutations - algorithm 2
  `[a,b,c,d],[b,a,c,d],[b,c,a,d],[b,c,d,a],[a,c,b,d],[c,a,b,d],[c,b,a,d],[c,b,d,a],`
  `[a,c,d,b],[c,a,d,b],[c,d,a,b],[c,d,b,a],[a,b,d,c],[b,a,d,c],[b,d,a,c],[b,d,c,a],`
  `[a,d,b,c],[d,a,b,c],[d,b,a,c],[d,b,c,a],[a,d,c,b],[d,a,c,b],[d,c,a,b],[d,c,b,a]`

Permutation walking

There is an anlogy in the recursion structure technique with the pretty print technique, which is more familiar, and, we feel, worth illustrating. A list L can be printed with an initial indentation of 0 by calling `io_pp(L,0)` (which is a default call made if `io_pp/1` is used, i.e. with just one parameter, the list.

```
io_pp([H|T],I):-
  !,
  J is I+3,
  io_pp(H,J),
  io_ppx(T,J).

io_pp(X,I):-
  tab(I),
  write(X),nl.


io_ppx([],_).

io_ppx([H|T],I):-
  io_pp(H,I),
  io_ppx(T,I).
```

Output

```
?- io_pp([a,b,[c,d,[e,f],g,h,[i,j]],k,l]).
   a
   b
      c
      d
         e
         f
      g
      h
         i
         j
   k
   l
```

Figure 32. io_pp

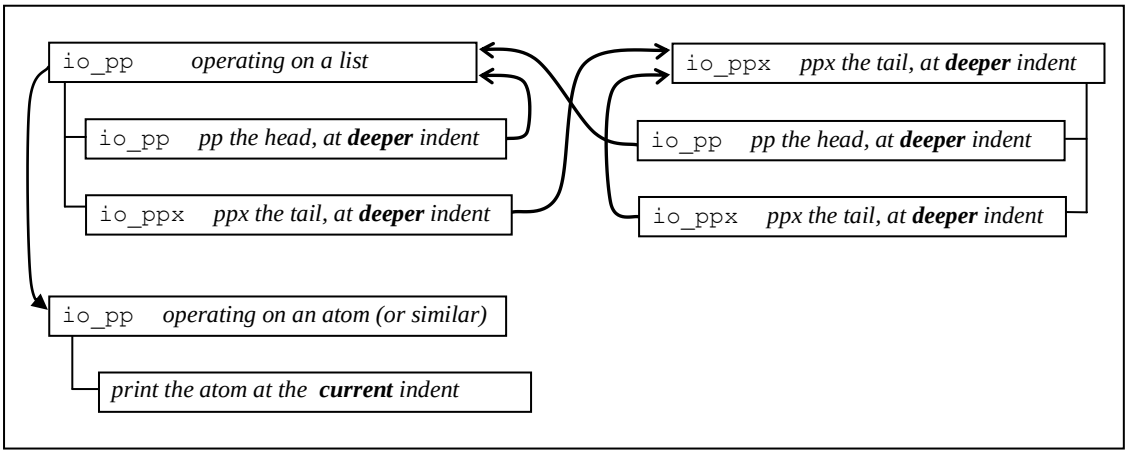


Figure 33. Call graph of io_pp

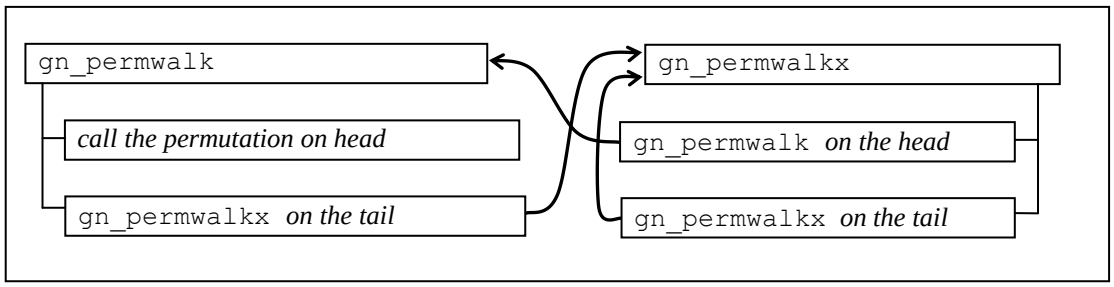


Figure 34. Basic permutation walking

Note that the predicate under consideration is for permutation ***walking***; the permutation itself is done by full or partial permutations. We can use any of the permutation generation predicates previously mentioned.

Predicate gn_permwalk walks a nested list structure generating permutations of parts of the structure as follows:

- NONLISTS (ATOMS, STRUCTURES, NUMBERS,..) are not affected
- LISTS are handled according to their first atom, which is a code
  - LISTS beginning with `'$pm_y'` are permuted at top level and walked at lower levels
  - LISTS not beginning with `'$pm_y'` are not permuted at the current level, but are walked at lower levels looking for lower level permutations

The reason for the symbol `'$pm_y'` beginning with a `$` (and so needing quoting in Prolog) is that is should not be met with by accident in the list structure. If user variables are not be allowed to begin with a `$`, then clashes can be prevented.

Identified `'$pm_y'` atoms are replaced by `'$pm_d'` (but these are removed on flattening - see below).

Module **`gnz_pm.pl`** contains some demonstrations.

A call to

```
gn_permwalk_find(X,Y,gn_permute_1)
```

with

```
X=['$pm_y',a,[b,c],['$pm_y',d,[e1,e2]],f]
```

generates 48 permutations. The outer permutable list has 4 user-elements; the inner permutable list has 2 elements, giving a total of 4! x 2! =48. The first and last two permutations are:

```
Y1= [$pm_d,a,[b,c],[$pm_d,d,[e1,e2]],f]
Y2= [$pm_d,a,[b,c],[$pm_d,[e1,e2],d],f]
Y47=[$pm_d,f,[$pm_d,d,[e1,e2]],[b,c],a]
Y48=[$pm_d,f,[$pm_d,[e1,e2],d],[b,c],a]
```

The solutions can be flattened separately using `gn_permflat`, or they can be flattened intrinsically by using gn_permwalk_flat_find. A call to

```
gn_permwalk_flat_find(X,Y,gn_permute_1)
```

with

```
X=['$pm_y',a,[b,c],['$pm_y',d,[e1,e2]],f],
```

gives 48 solutions again, the first and last two being

```
Y1= [a,[b,c],d,[e1,e2],f]
Y2= [a,[b,c],[e1,e2],d,f]
Y47=[f,d,[e1,e2],[b,c],a]
Y48=[f,[e1,e2],d,[b,c],a]
```

Sometimes the user must supply `'$pm_d'`s to obtain the right final structure (inserting a `'$pm_d'` in every high-level list that has been wrapped for the purpose of defining what is macro-manipulated). The following is derived from the set-transit example:

```
gnz_pmw20a:-
  gnz_pm_data(20,X),
  gn_permwalk_flat_find(X,Y,gn_permute_1),
  io_wlist_nl(Y),nl.

gnz_pm_data(20,X):-
  X=[ba,['$pm_y',BAA,BAB]],
  BAA=['$pm_d',baa,['$pm_y',BAAA,BAAB]],
  BAB=['$pm_d',bab,['$pm_y',BABA,BABB]],
  BAAA=['$pm_d',baaa,['$pm_y',baaaa,baaab]],
  BAAB=['$pm_d',baab,['$pm_y',baaba,baabb]],
  BABA=['$pm_d',baba,['$pm_y',babaa,babab]],
  BABB=['$pm_d',babb,['$pm_y',babba,babbb]].
```

128 permutations:

```
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,bab,baba,babaa,babab,babb,babba,babbb]
[ba,baa,baaa,baaaa,baaab,baab,baaba,baabb,bab,baba,babaa,babab,babb,babbb,babba]
...
[ba,bab,babb,babbb,babba,baba,babab,babaa,baa,baab,baabb,baaba,baaa,baaaa,baaab]
[ba,bab,babb,babbb,babba,baba,babab,babaa,baa,baab,baabb,baaba,baaa,baaab,baaaa]
```

Note that this permutation-walk would work even if the 'leaves' were lists (providing they did not contain `'$pm_y'`s or `'$pm_d'`s).

12.6 Each/One tree walking

We can walk a tree stating that for some sublists we wish to take one element at a time per solution. The predicates are `gn_eo_walk` and `gn_eo_walk_find`. The tag to request one element of a sublist is `'$from_one'`; in solutions it is replaced by `'$eo_d'` (standing for (each/one walker done).

Module `gnz_eo.pl` contains some demonstrations:
A simple example:

```
gnz_eo1.
RAW-linear=[a,[$from_one,p,q],z]

Walked=     [a,[$eo_d,p],z]
Flattened=  [a,p,z]

Walked=     [a,[$eo_d,q],z]
Flattened=  [a,q,z]
```

The following example is from a transition selection example, for hierarchical nondeterminism in the case of race and fork nondeterminism. The data involves user `'$eo_d'`s.

The data:

```
gnz_eo_data(2,X):-
 FROM_EACH='$eo_d',
 X= [FROM_EACH,
       ['$from_one',
          ['$from_one',a5,a7],
          ['$from_one',a9]],
       ['$from_one',
          ['$from_one',a10,a11],
          [FROM_EACH,
            ['$from_one',a12],
            ['$from_one',a13]]]].
```

The output, walked and flattened

```
| ?- gnz_eo2.
RAW-linear=
[$eo_d,[$from_one,[$from_one,a5,a7],[$from_one,a9]],[$from_one,
  [$from_one,a10,a11],[$eo_d,[$from_one,a12],[$from_one,a13]]]]

Walked=   [$eo_d,[$eo_d,[$eo_d,a5]],[$eo_d,[$eo_d,a10]]]
Flattened=[a5,a10]

Walked=   [$eo_d,[$eo_d,[$eo_d,a5]],[$eo_d,[$eo_d,a11]]]
Flattened=[a5,a11]

Walked=   [$eo_d,[$eo_d,[$eo_d,a5]],[$eo_d,[$eo_d,[$eo_d,a12],[$eo_d,a13]]]]
Flattened=[a5,a12,a13]
```

```
Walked=   [$eo_d,[$eo_d,[$eo_d,a7]],[$eo_d,[$eo_d,a10]]]
Flattened=[a7,a10]

Walked=   [$eo_d,[$eo_d,[$eo_d,a7]],[$eo_d,[$eo_d,a11]]]
Flattened=[a7,a11]

Walked=   [$eo_d,[$eo_d,[$eo_d,a7]],[$eo_d,[$eo_d,[$eo_d,a12],[$eo_d,a13]]]]
Flattened=[a7,a12,a13]

Walked=   [$eo_d,[$eo_d,[$eo_d,a9]],[$eo_d,[$eo_d,a10]]]
Flattened=[a9,a10]

Walked=   [$eo_d,[$eo_d,[$eo_d,a9]],[$eo_d,[$eo_d,a11]]]
Flattened=[a9,a11]

Walked=   [$eo_d,[$eo_d,[$eo_d,a9]],[$eo_d,[$eo_d,[$eo_d,a12],[$eo_d,a13]]]]
Flattened=[a9,a12,a13]
```


13. Regular expressions

Regular expression processing is offered as a general utility; it is not required for parsing. It is useful in testing, as it allows a flexible pattern match of one text string against another.

13.1 Basic usage

```
re_regexp(REGEXP,PARSE,INPUT,REST)
```

`REGEXP`	(input)	a nested list structure defining the regular expression
`PARSE`	(output)	the resulting parse, representing the input as divided up by the regular expression
`INPUT`	(input)	a list of ascii character codes, e.g. as obtained by `name('abc',INPUT)`, giving `INPUT=[97,98,99]`
`REST`	(output)	unused input characters in the parse

Example (from the zz_re.pl file)

```
| ?- rezdem(10).
ATINPUT=abcdcdcdgys
REGEXP= [ab,[zeroormoren,cd],[or,e,f,g],[not,x]]
PARSE=  [ab,[p_zeroormoren,[cd,cd,cd],3],[p_or,g,2],[p_not,y]]
REST=   [115]
```

The regexp can be read as: characters "ab", followed by zero or more occurrences of "cd", followed by an e,f, or g, followed by a character that is not "x".

The parse and remainder can be read as a match comprising in sequence:

- A *literal* term match consisting of the characters "ab"
- A *zero-or-more* term match consisting of 3 occurrences of "cd"
- An *or* term match, consisting of a "g", which is the 3rd matching option (first=0, second=1, third=2,...).
- A *not* term match, consisting of a "y".
- The remainder of the string consists of an "s" (in an ASCII list as [115]).

Repertoire of regular expression terms

In the description that follows,

`ATOM,A1,A2`	atomic representation of characters of any length e.g. `''` (i.e. null), `abc`
`AC`	an atomic representation of 1 character, e.g. `a`
`T`	additional range segments in the tail of the list
`N`	for repeating terms: repeat count for 'or' clauses: the Nth alternative (0,1,2,...)

`REGEXP`	`[ITEM,ITEM,ITEM,..]`
`PARSE`	`[P_ITEM,P_ITEM,P_ITEM,..]` or `[]`. Some `P_ITEM`s may be nested lists.

ITEM matched	P_ITEM in parse	MATCH
TERMINALS		
`[endlist]`	`[p_endlist]`	
`ATOM`	`ATOM`	1+ chars: 'x' or longer
`[not,AC]`	`[p_not,AC]`	1 char
`[anychar]`	`[p_anychar,AC]`	1 char
`[anycharsn]`	`[p_anycharsn,ATOM]`	0+ chars NONGREEDY: '' or more
`[anycharsg]`	`[p_anycharsg,ATOM]`	0+ chars GREEDY : '' or more
`[range,AC1,AC2|T]`	`[p_range,AC]`	1 char
`[notrange,AC1,AC2|T]`	`[p_notrange,ATCH]`	1 char
NONTERMINALS		
`[zeroormoreg,ITEM]`	`[p_zeroormoreg,P_ITEMLIST,N]`	GREEDY N items
`[zeroormoren,ITEM]`	`[p_zeroormoren,P_ITEMLIST,N]`	NONGREEDY N items
`[oneormoreg,ITEM]`	`[p_oneormoreg, P_ITEMLIST,N]`	GREEDY N items
`[oneormoren,ITEM]`	`[p_oneormoren, P_ITEMLIST,N]`	NONGREEDY N items
`[or,ITEM,ITEM,...]`	`[p_or,P_ITEM,N]`	Nth (0,1,2..) alternative
`[checkis,ITEM]` no consumption of input	nothing contributed to parse	ITEM must succeed
`[checknot,ITEM]` no consumption of input	nothing contributed to parse	ITEM must fail

Table 12. Regular expression items and parses

Pattern Matching Strategy

See [Expect], p. 108, p.113, p.137 - but we do not conform to that here.

Issues:

1. Match earliest starting position
2. Match left-most branch
3. Matched longest string
4. Subexpressions from left to right

Greediness

- GREEDY: repeating items absorb as much as they can but backtracking if subsequent terms fail will cause SHORTENING.
- NONGREEDY: repeating items absorb as little as they can, but backtracking if subsequent terms fail will cause LENGTHENING.

Our strategy

1. Left-most branch taken if it matches at all is taken
2. Match earliest starting position
3. We provide GREEDY and NONGREEDY strategies

Note: There is *no* term or parse for the start of a list. It would serve no purpose here. If it were implemented, it would be e.g.

`ITEM=`	`[startlist]`	(terminal) start of list
no: `P_ITEM=`	`[startlist]`	matched `[startlist]`

Note that we *do* have `[endlist]`, which may force backtracking on nongreedy terms.

Advanced examples from the test suite

Showing nongreediness

```
tcre([anycharsn,5],re_regexp(REGEXP,PARSE,INPUT,REST),
(PARSE=EXPECT,REST=[])):-
    name('philosophic',INPUT),
    REGEXP=[ [anycharsn],     hi, [anycharsn],            [endlist] ],
    EXPECT=[ [p_anycharsn,p], hi, [p_anycharsn,losophic], [p_endlist] ].
```

Showing greediness

```
tcre([anycharsg,6],re_regexp(REGEXP,PARSE,INPUT,REST),
(PARSE=EXPECT,REST=[])):-
    name('philosophic',INPUT),
    REGEXP=[ [anycharsg],            hi, [anycharsg],     [endlist] ],
    EXPECT=[ [p_anycharsg,philosop], hi, [p_anycharsg,c], [p_endlist] ].
```

Showing NESTED REGEXP terms and greediness

```
tcre([zeroormoreg,5],re_regexp(REGEXP,PARSE,INPUT,REST), PARSE=EXPECT):-
    name('abAAAAAde',INPUT),
    REGEXP=[ ab,[zeroormoreg,[range,'A','Z']],'AAde' ] ,
    EXPECT=
            [ab,
            [p_zeroormoreg,[[p_range,'A'],[p_range,'A'],[p_range,'A']],3],
            'AAde'].
```

13.2 Greedy and nongreedy algorithms

We compare greedy and nongreedy algorithms with analogous syntax diagrams. In the diagrams below, the ***vertical order of railroad lines*** is significant, being equivalent to the order in which PROLOG tries to satisfy a predicate.

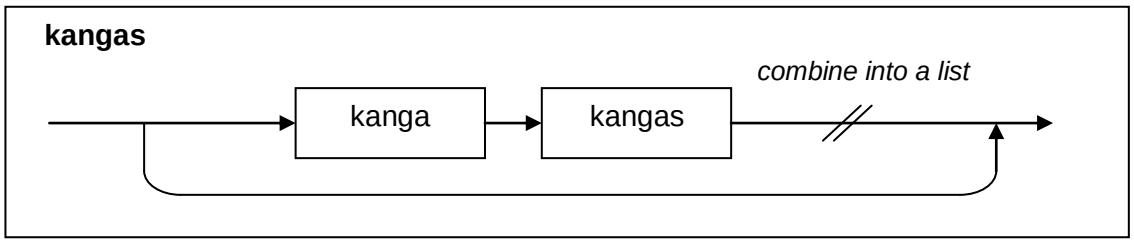


Figure 35. Zero or more (greedy)

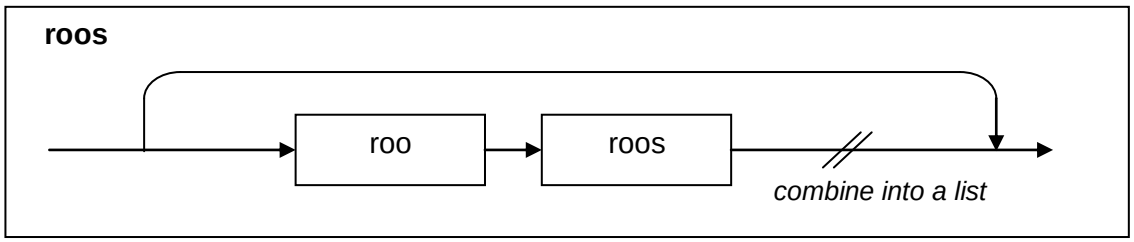


Figure 36. Zero or more (nongreedy)

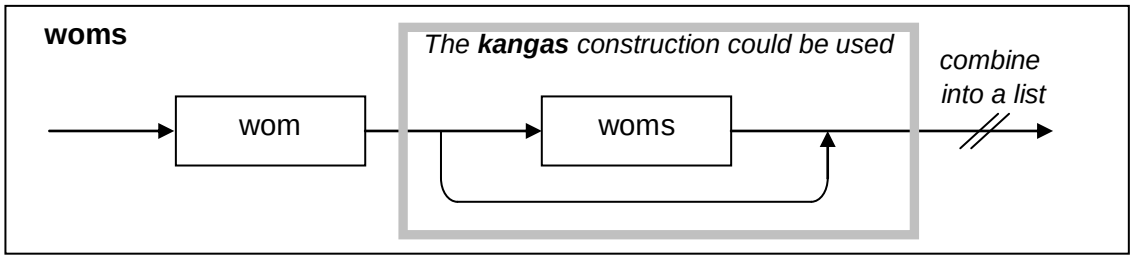


Figure 37. One or more (greedy)

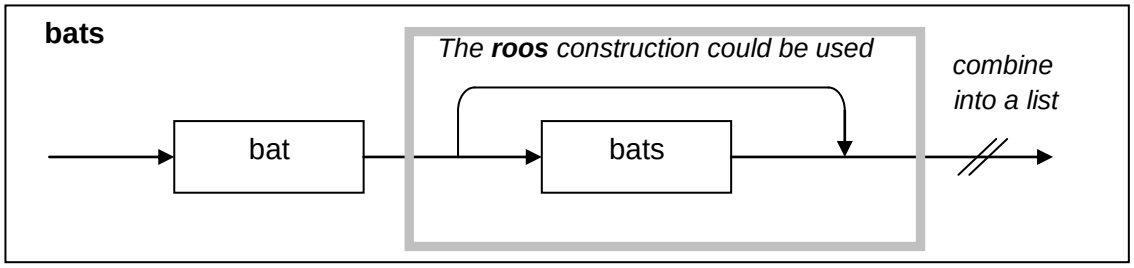


Figure 38. One or more (nongreedy)

The greedy algorithm is implemented as follows:

gn_append_greedy

```
/*-------------------------------------------------------------------------*/
/* gn_append_greedy                                                        */
/* ================                                                        */
/* Description:                                                            */
/*   ALL parameters are lists                                              */
/*   Appends lists, but order of solutions is opposite to gn_append        */
/*                                                                         */
/*   We use a grammar rule for a repeating item,                           */
/*     and provide a wrapper to adjust parameter calling order             */
/*                                                                         */
/*   Works fine to split a list, long L1 first                             */
/*                                                                         */
/*   Does not NATURALLY work with L1 and L3 uninstantiated, e.g.           */
/*      gn_append_greedy(L1,[c,d],L3])                                     */
/*   so we handle exceptionally in this case                               */
/*-------------------------------------------------------------------------*/
gn_append_greedy(L1,L2,L3):-
   var(L1),var(L3),            /* handle exceptionally */
   gn_append(L1,L2,L3).

gn_append_greedy(L1,L2,L3):-
   gn_app_elems(L1,L3,L2).

gn_app_elems([EL|ELS]) -->
   gn_app_elem(EL),
   gn_app_elems(ELS).

gn_app_elems([]) -->
   [].

gn_app_elem(EL) -->
   [EL].
```

Example call

```
| ?- gn_append_greedy(L1,L2,[a,b,c]).
L1 = [a,b,c] ,
L2 = [] ;

L1 = [a,b] ,
L2 = [c] ;

L1 = [a] ,
L2 = [b,c] ;

L1 = [] ,
L2 = [a,b,c]
```

13.3 Module "tf" (Test Framework)

13.3.1 Introduction

The test framework produces a test report in the presentation style of DejaGnu [DejaGnu], but the scripting is specifically geared to PROLOG predicate testing, (not standard-I/O-executable testing as in the case of DejaGnu). The framework enables automatic execution of tests, where each test defines its own pass/fail criterion and a log is produced of all tests with their PASS/FAIL status. Ideally, stub control, driven by the script, is also possible. For the GP4 Prolog applications there has been no need to test with stubs - this may be due to the intrinsic local scope of Prolog variables and the bottom-up way of development. However, one way to provide for control over stubs is at assert predicates for them before a SUT (System Under Test) call with the "asserta" predicate, and retract them after the test. To prevent access to the real function stubbed, the last of such asserted stub predicates would include a catch-all and cut-fail combination.

The following figure shows how automated test execution works.

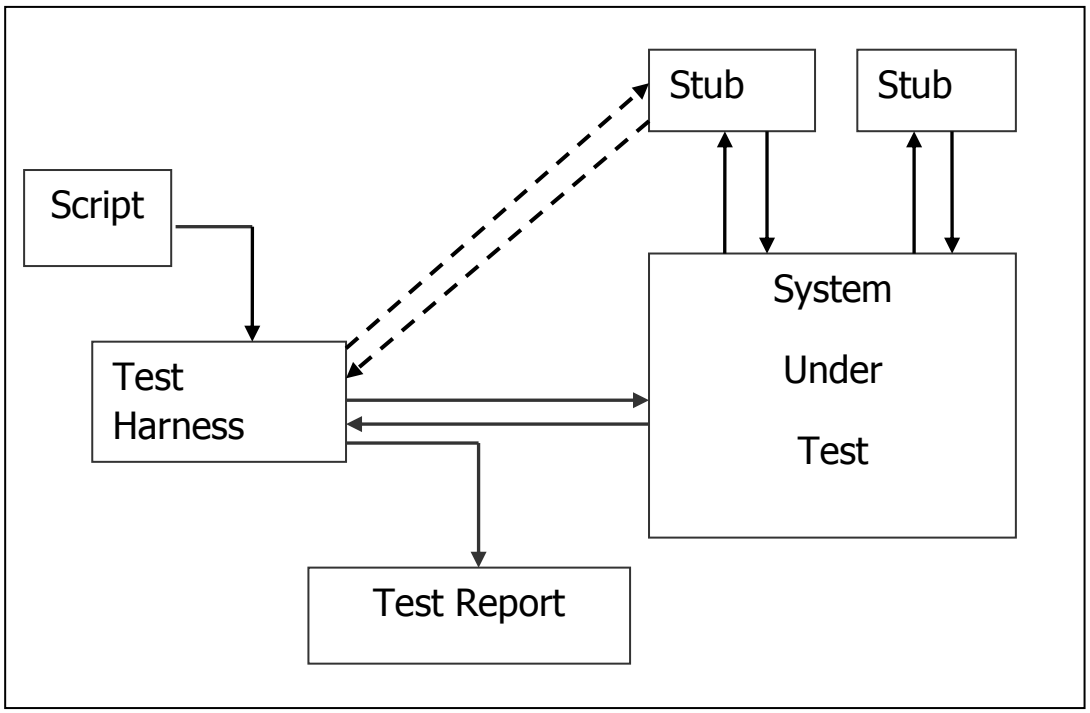


Figure 39. Automatic test execution

In the case of Prolog, the "script" takes the form of a predicate, `tc`, that is repeatedly instantiated by the test harness.

13.3.2 Description of test case definition predicate tc

The test case definition predicate is

```
tc(TESTNAME,DESCRIPTION,PREDICATE,CONDITION)
```

TESTNAME is the test name in list format, e.g. `[gp4, re, or, 3].` This might represent

`gp4` =	Package (General Prolog Parsing and Prototyping Package)
`re` =	Predicate under test = re_regexp = regular expressions
`or` =	"inclusive or" processing
`3` =	test number 3 of the above functionality

DESCRIPTION is the test description as atomic (single-quoted) text.

PREDICATE is the predicate to be called for the test. It can consist of a conjunction, which is standard Prolog, e.g. `(dothis(A,Y),dothat(Y,Z)).`

CONDITION is a predicate call that must evaluate to true for the test to pass. If the test simply tests for a successful call of `PREDICATE`, then `CONDITION` should be set to **true**. `CONDITION` can also be a conjunction, which is standard Prolog, e.g. `(X>Y,happywith(Y)).`

Notes:

- If a single **tc** predicate can be instantiated many times, then it defines *several* tests.
- If the **PREDICATE** in a call can be instantiated several times, then it defines *several* tests.
- If the **PREDICATE** in a call can produce several solutions from an instantiation, only the first solution is taken and the predicate defines *one* test. However, it is possible for the whole `PREDICATE` to be considered as succeeding when the core of it has been made to backtrack over *n* solutions (see section 13.3.3).

Feature 1: **CONDITION** can be set to **dontrun** so as not to run the test, nor count it in the log summary.

Feature 2: **CONDITION** can be set to the atom **negate**. This is interpreted as meaning that the **PREDICATE** must fail. It saves having to write `gn_not((...))` around the SUT predicate.

Feature 3: **CONDITION** can be set to the atom **negate_msg.** This is as with being set to **negate**, but in the test log, the message

```
**This test provokes an error message (but not a fail!) - please ignore
```

will be output. This is to warn the reader of the test report that an error message has been provoked as part of the test. It applies negation to the `PREDICATE`, as it is intended for test

predicate calls that fail, (in the Prolog sense), but produce a diagnostic message. It does not mean that the test should produce a FAIL status! A test script using **`negate`** and **`negate_msg`** should run with every test producing PASS (if the SUT works correctly).

13.3.3 Support predicates for the nth backtracking result.

It is sometimes desirable to obtain the result of a second, third, or in general nth call to a predicate on backtracking. To support this, the **`PREDICATE`** should be a conjunction of

```
tf_failNtimes_init, sut_call(...), tf_failNtimes(N)
```

The predicate `tf_failNtimes(N)` will fail N times and then succeed, and any variables instantiated by *sut_call*(..) in that last backtracked call will be available for examination in **`SCONDITION`**.

The predicate `tf_failNtimes_init` should be initialised in every test using `tf_failNtimes(N).`

13.3.4 Running tests: `runtest`

To run all tests, the goal

```
runtest.
```

or

```
runtest(SELECTOR).
```

is given. This

- runs all or a subset of the tests
- logs results

The `SELECTOR` parameter

This is a *selector* for group/name of test(s) to run. If **`SELECTOR`** is omitted, all tests are run. A test will be run if it is compatible with the selector parameter of `runtest` - the selector must be equal or shortened in the tail (or=`[]`). For example, if

```
TESTNAME = [gp4,re,or,3]
```

and

```
SELECTOR =[gp4,re]
```

then the test will be run.

An extra option to define tests to be run as one batch is to nest alternative selector elements in the list, e.g.

```
runtest([gp4,tf,[callfail,badtest,condfail,negfail]]).
```


This runs the same tests as

```
runtest([gp4,tf,callfail]).
runtest([gp4,tf,badtest]).
runtest([gp4,tf,condfail]).
runtest([gp4,tf,negfail]).
```

A non-ground guard

When checking a return value in an equality condition, e.g.

```
(ACTUAL=EXPECTED)
```

we guard against condition checking on uninstantiated parameters, which can easily happen, typically when a parameter is misspelled. For this reason, a check is made on conditions of the type

```
X=Y
```

but ***not***

```
X\=Y,  X>Y,  X<Y,  X>=Y,  X=<Y
```

The check made is that `X` and `Y` are instantiated.

The check also applies to such terms in composite conditions, e.g.

```
(A=B, C>D, dothis(P,Q), E=F)
```

If the check fails, the test fails.

Workarounds if this is not convenient: Use

```
VEXP is 1.4E2
```

rather than

```
VEXP=1.42
```

or

put `VEXP =1.42` in the right hand side (i.e. after the ":-")

or

put `VEXP =1.42` in conjunction with the SUT call rather than in the condition

Additional `CONDITION` options

- If `CONDITION` = `dontrun`, the case will be rejected (not a test fail!)
- If `CONDITION` = `negate`, the test will be run with gn_not(P4)
- If `CONDITION` = `negate_msg`, the test will be run with gn_not(P4) and produce a message indicating that error messages have been provoked

Caution

In many situations, there is no need to have a body to the test case clause. But sometimes it is convenient to instantiate an expected result there, e.g.

```
tc(....(ACTUAL=EXPECTED)):-EXPECTED=...
```

Take care with what done is in such a body, as

- it is picked up by the test harness and executed even if the particular test is not selected
- it is executed by showtest (see below).

A good rule is *never to do anything more than instantiating complex expected values.*

***Listing and counting tests:* `showtest`**

Call as

```
showtest
```

or

```
showtest(SELECTOR)
```

Counting placeholders

Placeholders have a PREDICATE of true and a CONDITION of true. They are useful to keep the number of tests to round numbers, which is useful in keeping count of the number of tests so as to ensure all are run. In order to see the number of real tests, the number placeholders can be counted by

```
tf_count_placeholders
```


13.3.5 Examples and test reports

Test cases (passing)

```
/*-------------------------------------------------------------------------*/
/* Test for gn_bag2set    Group name: gn_bag2set                           */
/*                                                                         */
/*   Version1: sorts alphabetically                                        */
/*   Version2: Maintains the right-to-left order in the bag                */
/*-------------------------------------------------------------------------*/
tc([gp4,gn,bag2set,1],
   'Test of Bag to Set (Algorithm 1)',
   gn_bag2set1([d,b,q,c,b,b,c,a],SET),
   SET=[a,b,c,d,q]
   ).

tc([gp4,gn,bag2set,2],
   'Test of Bag to Set (Algorithm 2)',
   gn_bag2set2([d,b,q,c,b,b,c,a],SET),
   SET=[d,b,q,c,a]
   ).
```

Running the tests

```
?- runtest([gp4,gn,bag2set,[1,2]]).

Start of Tests. Test selector = [gp4, gn, bag2set, [1, 2]]

PASS           [gp4, gn, bag2set, 1]
  Seq. number: 1
  Description: Test of Bag to Set (Algorithm 1)
  Predicate:   gn_bag2set1([d, b, q, c, b, b, c, a], _G334)
  Condition:   = [a, b, c, d, q]
                 [a, b, c, d, q]
  Comment:     Pass

PASS           [gp4, gn, bag2set, 2]
  Seq. number: 2
  Description: Test of Bag to Set (Algorithm 2)
  Predicate:   gn_bag2set2([d, b, q, c, b, b, c, a], _G334)
  Condition:   = [d, b, q, c, a]
                 [d, b, q, c, a]
  Comment:     Pass


Number of passes=2
Number of fails= 0

End of Tests
DATE OF TESTS:     19 Oct 2003 20:46:16/110
DURATION OF TESTS: 00h 00m 01s 260ms

Yes
?-
```

Example with fails

Here we have seeded two errors. One is in the `PASSCONDITION`, and one in the SUT call itself to make it fail.

Test cases (failing)

```
tc([gp4,gn,bag2set_err,1],
   'Test of Bag to Set (Algorithm 1)',
   gn_bag2set1([d,b,q,c,b,b,c,a],SET),
   SET=[a,b,c,d,qq]                        /* seeded here */
   ).

tc([gp4,gn,bag2set_err,2],
   'Test_gn_bag2set2_1',
   gn_bag2set2(not_a_list,SET),            /* seeded here */
   SET=[d,b,q,c,a]
   ).
```

Running the tests

```
?- runtest([gp4,gn,bag2set_err]).

Start of Tests. Test selector = [gp4, gn, bag2set_err]

**FAIL**       [gp4, gn, bag2set_err, 1]
  Seq. number: 1
  Description: Test of Bag to Set (Algorithm 1)
  Predicate:   gn_bag2set1([d, b, q, c, b, b, c, a], _G307)
  Condition:   = [a, b, c, d, q]
                 [a, b, c, d, qq]
  Comment:     Condition failed

**FAIL**       [gp4, gn, bag2set_err, 2]
  Seq. number: 2
  Description: Test_gn_bag2set2_1
  Predicate:   gn_bag2set2(not_a_list, _G283)
  Condition:   = _G221
                 [d, b, q, c, a]
  Comment:     Call failed


Number of passes=0
Number of fails= 2

End of Tests
DATE OF TESTS:     19 Oct 2003 20:57:28/840
DURATION OF TESTS: 00h 00m 00s 720ms

Yes
?-
```


14. Extent of implemented features

A few less common operators have not been implemented in version 1.0. They are listed below.

14.1 Grammar productions

The following parsing features have not been implemented in Version 1.0, as they are not required for the initial applications envisaged. They could easily be added as required.

- Monadic operators of type `[f,x]`.
- Monadic operators of type `[x,f]`.
- Dyadic prefix operators of the type (castfunction)argument
- Dyadic prefix operators of the type [operand1]operand2
- Dyadic operators of type `[x,f,x]`.

These are less common operators: all the common operators have been implemented, e.g. monadic operators of type `[f,y]` have been implemented. Of the above, the only one "C" uses is *(castfunction)argument*.

14.2 Operator definition for parsing

All the operators defined in section 4.4 have been implemented for parsing.

For efficiency reasons, the maximum number of tokens an operator may use (at the time of writing) is 6, and the maximum number of tokens of an *overloaded* operator is 4. The module imposing these restrictions is op_aa.pl.

14.3 Operator evaluation

Only the operators listed in section 10.3 have been implemented for evaluation .

14.4 Function call evaluation

The functions listed in section 11.2 have been implemented for evaluation.

15. References

STATECRUNCHER documentation and papers by the present author

Main Thesis	[StCrMain]	The Design and Construction of a State Machine System that Handles Nondeterminism

Appendices

Appendix 1	[StCrContext]	Software Testing in Context
Appendix 2	[StCrSemComp]	A Semantic Comparison of STATECRUNCHER and Process Algebras
Appendix 3	[StCrOutput]	A Quick Reference of STATECRUNCHER's Output Format
Appendix 4	[StCrDistArb]	Distributed Arbiter Modelling in CCS and STATECRUNCHER - A Comparison
Appendix 5	[StCrNim]	The Game of Nim in Z and STATECRUNCHER
Appendix 6	[StCrBiblRef]	Bibliography and References

Related reports

Related report 1	[StCrPrimer]	STATECRUNCHER-to-Primer Protocol
Related report 2	[StCrManual]	STATECRUNCHER User Manual
Related report 3	[StCrGP4]	GP4 - The Generic Prolog Parsing and Prototyping Package *(underlies the STATECRUNCHER compiler)*
Related report 4	[StCrParsing]	STATECRUNCHER Parsing
Related report 5	[StCrTest]	STATECRUNCHER Test Models
Related report 6	[StCrFunMod]	State-based Modelling of Functions and Pump Engines

References

[Aho] Alfred V. Aho, Jeffrey D. Ullman
Principles of Compiler Design
Addison-Wesley Publishing Company, 1977. ISBN 0-201-10073-8
(This book is well-known as *The Dragon Book*).

[Baker 95] M.L. Baker and D.C. Yule
Automation of Software Testing:
A Case Study on a Real-Time Embedded System"
Philips PRL Technical Note 3373, September 1995

[Bennet] J.P. Bennet
Introduction to Compiling Techniques
McGraw Hill

[C] Peter A. Darnell and Philip E. Margolis
C - A Software Engineering Approach
Springer-Verlag, 2nd edition, 1990.
ISBN 0-387-79389-3 & 3-540-97389-3.

[CHSM] Paul J. Lucas
An Object-Oriented System for Implementing Concurrent, Hierarchical, Finite State Machines.
MSc. Thesis, University of Illinois at urbana-Champaign, 1993

[Clocksin] W.F. Clocksin & C.S. Mellish
Programming in Prolog, 2nd Edition
Springer Verlag, 1984

[DejaGnu] R. Savoye
The DejaGnu Testing Framework
The Free Software Foundation, 1993

[Dexios] G.G. Thomason & P. van Loon
DEXIOS User and Reference Manual
Philips CFT/Automation Report UDR-ITD-X88-GT193-gt, 1989

[Expect] Don Libes
Exploring Expect
O'Reilly, November 1996. ISBN 1-56592-090-2

[WinPro] WinProlog, Logic Programming Associates Ltd
http://www.lpa.co.uk

[Yule 97] D.C. Yule
Automatic State-Based Testing
Philips PRL Technical Note TN 3611, 1997 / DVD Document V19 C4 S41

