Personal Reckoning Language

RECKON

(Draft 2a)

Robert L. Walton[*]

August 31, 2016

[*]Copyright 2016 Robert L. Walton. Permission to copy this document verbatim is granted by the author to the public. This document was partly inspired by teaching courses at Suffolk University with the help of Paul Ezust, and by the work of Thomas Cheatham and Stuart Shieber.

Table of Contents

1 Introduction

This document describes the programming language RECKON. RECKON, for what its worth, stands for ‘Personal Reckoning Language’, which hints at its purpose.

RECKON is designed for naive programmers: that is, for people who may never be able to program computers well. It is a fairly simple language with powerful data types that make it easier to write small programs that do a variety of tasks a person might want to do. Generally the tasks fall into the categories of calculating things (taxes, probabilities, statistics) or simulating things (computer games, construction designs, mathematics demonstrations). Included are:

Calculations that might be done with a spreadsheet.
Drawing pictures.
Simulating popular board games and creating new ones.
Creating simple computer games, including dialog games.
Computing and analyzing documents.
Doing elementary algebra and calculus problems.
Doing matrix calculations.
Calculating basic probabilities and statistics.
Simulating two and three dimensional objects.
Simulating simple electrical, mechanical, chemical, and biological systems.
Solving problems in elementary logic.

There are many computer languages that have some powerful data type that adapts them for a specific kind of computation. RECKON tries to combine these. Some previous computer languages that have influenced RECKON, and the data types they particularly support, are:

Various Spreadsheets	Spreadsheets
Various Data Base Languages	Data Bases
Various Script Languages	Documents
MATLAB	Matrices
Mathematica	Mathematical Formulae
TCL	Character Strings and Lists
Lisp	Words and Phrases
PROLOG	Logical Expressions

RECKON is <u>not</u> designed to be a computer-efficient language. It is designed to be person-efficient, and to do small calculations rapidly enough with inexpensive modern computers.

2 Remarks

RECKON was created as an answer to the question: what programming language should you teach beginning programming students most of whom do not have the talent or will not develop the inclination to become serious programmers? The initial answer, that it does not matter provided you implement some powerful types of data in the language you choose, has a flaw. The flaw is that without the right powerful data types, the language will be useless to the students after the course is over. So what is needed is a programming language that will be useful to students after a first course in programming, and the essence of such a language is the integration within it of many powerful and useful data types.

The basic principles of the RECKON design were developed by the author while teaching the intended customers of RECKON.[1] The language should have as few parts as possible, to cut down on the amount of detail that must be remembered to use the language, but conversely, there is no limit to the conceptual complexity of any well-used part.[2] The language should have powerful data types, well integrated into the syntax of the language. As much as possible, statement executions in the language should have visible effect.

RECKON is also based on the '***Syntactic Hypothesis***', which is that people reason syntactically and not in any other way, and therefore syntax matters a lot. In particular, different kinds of reasoning require different syntactic support.

The current version of RECKON is not stable, because it has not been implemented, and because, unlike most programming languages, RECKON has lots of subtle important interactions between its various features. The hope is that after implementation and experimentation a stable sensible version of RECKON, integrating all its data types and supporting each with special syntax, can be achieved.

3 Overview

RECKON has two major kinds of data: expressions and blocks. Numbers are the simplest expressions. More complex expressions are math expressions or document expressions. A block is a set of variables and pieces of code. Each variable can have a value, which is an expression. Each block has a unique indentifier, which by itself can be used as an expression that names the block. The code contains expressions that can be evaluated under appropriate circumstances to produce values for variables.

[1]Specifically, while teaching CS121 at Suffolk University using the C programming language.

[2]There was no problem teaching recursion, but it was better not to teach many different looping constructs.

In RECKON a ‘description’ is a block that has an associated ‘type’. For example, there may be a description ‘george’ with type ‘person’. All descriptions of the same type have the same code, and many of the same variables, but typically have different variable values.

You can use RECKON as a calculator by typing into it expressions to be evaluated, assignments of values to variables, and definitions of functions, predicates, and blocks. Some examples involving numbers are:

```
> 9
9
> 9 + 8
17
> x = 9
9
> y = 9 + 8
17
> x + y
26
> f (b, c, x) = b * x + c
f (b, c, x) evaluates to (b * x + c)
> f (10, x, y)
179
```

Here the ‘**>** ’ at the beginning of some lines is the RECKON ***prompt*** that tells you its OK to input an expression to be evaluated. Except for this prompt, lines beginning with ‘**>** ’ are input lines to a RECKON interpreter, and other lines are output lines.

Note that in RECKON names must be separated by whitespace. Thus ‘**b * x**’ names three things: the variable ‘**b**’, the operator ‘*****’, and the variable ‘**x**’. In contrast, ‘**b*x**’ would name one thing, a variable (with a strange name). However parentheses such as ‘**(**’ and ‘**)**’ need not be separated from anything else by whitespace, and separators such as ‘**,**’ need not be separated by what is <u>before</u> them by whitespace.

At somewhat the opposite extreme from numbers are words, phrases, sentences, and paragraphs. You can calculate with these ‘***document expressions*** ’.[3]

```
> g = `hello'
```
hello

[3]In what follows, if we followed LISP, each **[]** would cancel one **` '**, so **``[g]''** would evaluate to **`[g]'** and **``[[g]]''** would evaluate to **`hello'**. But we do the reverse of this, so in RECKON **``[g]''** evaluates to **`hello'** and **``[[g]]''** evaluates to **`[g]'**.

```
> `[g] there'
```
hello there
```
> z = `I thought he said `[g]'.'
```
I thought he said ‘hello’.
```
> notice = ``This document is meant to be read.
+            Reading this document is good, but...
+
+            [z].''
```
This document is meant to be read. Reading this document is good, but…

I thought he said ‘hello’.
```
> `When you add [x] and [y] you get [x+y].'
```
When you add 9 and 17 you get 26.
```
> ``This is a paragraph.
>
>   And a second paragraph.
>   With thress sentences.
>   Use {tt|"`` ''"|} instead of {tt|"` '"|} if the text
>   includes blank lines.''
```
This is a paragraph.

And a second paragraph. With three sentences. Use `` `` '' `` instead of `` ` ' `` if the text includes blank lines.

Modern math computes with expressions, and not just numbers. You can compute with ***math expressions*** in RECKON.

```
> f = {{ 10 x ^ 2 - 3.67 x - 0.04 }}
```
$10x^2 - 3.67x - 0.04$
```
> h = (- 0.96 + 0.67 x) in x
```
$-0.96 + 0.67x$
```
> (f + h) in x
```
$10x^2 - 3x - 1$
```
> solve (f + h = 0) for x
```
$x = (-0.2, 0.5)$
```
> (f + h) at (x = (3, 4, 5))
(78.95, 145.28, 231.61)
> g = {{ integral (x ^ 2 dx) }}
```
$\int x^2 dx$

```
> simplify g
```
$\frac{1}{3}x^3$
```
> v = g from (x = 1) to (x = 5)
41 1/3
> out = 'The value of {{[g] from (x = 1) to (x = 5)}}
>        is [v].'
```
The value of $\int_{x=1}^{x=5} x^2 dx$ is $41\frac{1}{3}$.

Matrices are another kind of data that you can compute with:

```
> V =# {| 1  2  3 |}
1 2 3
> M =#:
+      0  1   1
+     -1  1   0
+      1  0  -1
 0  1   1
-1  1   0
 1  0  -1
> V ^ T
1
2
3
> M * V ^ T
 5
 1
-2
```

Another kind of datum you can compute with in RECKON is the block. A ***block*** contains a set of variables, each of which can have a value which is either an expression or the identifier of a block. A block can also have code, which contains expressions that are evaluated under appropriate circumstances to produce values for the block's variables.

In RECKON a ***description*** is a block that has an associated ***type***. For example, there may be a description named 'Jack' with type 'person'. All descriptions of the same type have the same code, and many of the same variables, but typically have different variable values. For example:

```
> a person:
+      name = 'Jack'
+      weight = 123 lb
+     height = 5 ft 9 in
+     age = 23 yr 2 mo
```

ID	type	name	weight	height	age
@1000000	person	Jack	123 lb	5 ft 9 in	23 yr 2 mo

```
> a person:
+     name = 'Jill'
+     weight = 110 lb
+     height = 5 ft 7 in
+     age = 21 yr 8 mo
```

ID	type	name	weight	height	age
@1000001	person	Jill	110 lb	5 ft 7 in	21 yr 8 mo

```
> all persons
```

ID	type	name	weight	height	age
@1000000	person	Jack	123 lb	5 ft 9 in	23 yr 2 mo
@1000001	person	Jill	110 lb	5 ft 7 in	21 yr 8 mo

```
> the person Jack
```

ID	type	name	weight	height	age
@1000000	person	Jack	123 lb	5 ft 9 in	23 yr 2 mo

```
> the person named Jack's height
5 ft 9 in
> the weight of the person named Jack
123 lb
> @1000001
```

ID	type	name	weight	height	age
@1000001	person	Jill	110 lb	5 ft 7 in	21 yr 8 mo

```
> the weight of @1000001
110 lb
```

It is possible to add code to a type such as '**`person`**'. This has the affect of adding the code to all blocks that are descriptions of that type. For example:

```
> for every person:
+     body mass index = 703.06958 * weight in lbs
+                          / (height in inches)^2
> all persons
```

ID	type	name	weight	height	age	body mass index
@1000000	person	Jack	123 lb	5 ft 9 in	23 yr 2 mo	18.1637
@1000001	person	Jill	110 lb	5 ft 7 in	21 yr 8 mo	17.2282

One can use definitions to define expressions that compute values:

```
> sum from X through Y <-- is an integer X, is an integer Y:
+     'Sum of integers from X through Y.'
+     if ( X > Y ):
+         value = 0
+     else:
+         value = X + sum (X+1) through Y
> sum from 5 through 10
45
> all (sums from X through Y)
```

ID	parent	X	Y	value
@1000002		5	10	45
@1000003	@1000002	6	10	40
@1000004	@1000003	7	10	34
@1000005	@1000004	8	10	27
@1000006	@1000005	9	10	19
@1000007	@1000006	10	10	10
@1000008	@1000007	11	10	0

```
> sum from 1 through 2
3
> all (sums from X through Y)
```

ID	parent	X	Y	value
@1000002		5	10	45
@1000003	@1000002	6	10	40
@1000004	@1000003	7	10	34
@1000005	@1000004	8	10	27
@1000006	@1000005	9	10	19
@1000007	@1000006	10	10	10
@1000008	@1000007	11	10	0
@1000009		1	2	3
@1000010	@1000009	2	2	2
@1000011	@1000010	3	2	0

Executions can be examined in detail, because when an expression is computed, the block that computes it is remembered for some time. However, as memory is finite, eventually these computations are forgotten as they can always be regenerated if needed.

The sum above was computed by recursion: to compute the sum of 5 through 10 one adds 5 to the sum of 6 through 10. One can also compute sums by iteration.

```
> sum from X through Y <-- is an integer X, is an integer Y:
+     `Sum of integers from X through Y.'
+     first sum = 0
+     if ( X <= Y ):
+         next sum = sum + X
+         next X = X + 1
+     else:
+         value = sum
> sum from 5 through 10
45
> all (sums from X through Y)
```

ID	previous	next	X	Y	sum	value
@1000012		@1000013	5	10	0	
@1000013	@1000012	@1000014	6	10	5	
@1000014	@1000013	@1000015	7	10	11	
@1000015	@1000014	@1000016	8	10	18	
@1000016	@1000015	@1000017	9	10	26	
@1000017	@1000016	@1000018	10	10	35	
@1000018	@1000017		11	10	45	45

One can also iterate over data.

```
> average weight of X <-- list X of persons:
+     first count = 0
+     first sum = 0
+     if X = ():
+         if count = 0:
+             value = error `Cannot average 0 things.'
+         else:
+             value = sum / count
+     else:
+         next count = count + 1
+         next sum = sum + the weight of (first X)
+         next X = rest X
> average weight of (all persons)
116.5 lbs
> z = 10 + average weight of (all persons named Bill)
```

```
error `Cannot average 0 things.'
      occurred during: average weight of
                       (all persons named Bill)
      occurred during: 10 + ditto
      occurred during: z = ditto
```

In case you wonder how some of the above works, here are some hints.

RECKON tends to ignore word endings: thus ‘**`person`**’ and ‘**`persons`**’ are to RECKON the same word. RECKON can even be told that ‘**`person`**’ and ‘**`people`**’ are the same word. On the other hand, ‘**`Jack's`**’ is treated an abbreviation of two separate words ‘**`Jack`**’ and ‘**`'s`**’, where ‘**`'s`**’ is a separate word by itself.

Expressions are just strings of words, numbers, and subexpressions. Subexpressions must be parenthesized unless they are delimited by operators.

Lists of values can be stored in ***lists***, which are computed by comma separated lists in parentheses. Thus

```
> (the person named Jill, the person named Jack)
```

ID	**type**	**name**	**weight**	**height**	**age**
`@1000001`	person	Jill	`110 lb`	`5 ft 7 in`	`21 yr 8 mo`
`@1000000`	person	Jack	`123 lb`	`5 ft 9 in`	`23 yr 2 mo`

```
> (the person named Jack, the person named Jill)
```

ID	**type**	**name**	**weight**	**height**	**age**
`@1000000`	person	Jack	`123 lb`	`5 ft 9 in`	`23 yr 2 mo`
`@1000001`	person	Jill	`110 lb`	`5 ft 7 in`	`21 yr 8 mo`

```
> raw (all persons)
(the person named Jack, the person named Jill)
> really raw (all persons)
(@1000000, @1000001)
```

The ‘**`raw`**’ form of a value represents the value as you could input it in a way that reveals its internal structure. Thus ‘**`raw(all persons)`**’ denotes the list of all persons, and it is the list structure that is revealed, not the structure of the elements of the list.

‘**`the person named Jack`**’ is a printed representation of the internal name of a description. Such printed representations are chosen automatically from the set of all possible representations, which in this case include ‘**`the person weighing 123 lbs`**’, ‘**`the person named Jack weighing 123 lbs`**’, and so forth.

The ‘**`really raw`**’ form of a value identifies descriptions by their IDs.

A single non-list value is equivalent to a list with one element. Lists cannot have other lists as elements; instead attempts to compute such lists are ***flattened***:

```
> x = (1,(2,3),4)
(1,2,3,4)
> first x
1
> rest x
(2,3,4)
> rest (rest x)
(3,4)
> rest (rest (rest x))
4
> rest 4
()
```

For this reason RECKON lists are sometimes called '***flat lists***'.

RECKON has different kinds of quotes or brackets. The brackets `{{...}}` and `` `...' `` turn evaluation off, while `[...]` turns evaluation on. The brackets `` `...' `` turn recognition of operators (e.g., + and =) off, and turn recognition of sentence and phrase terminators (e.g., '`.`' and '`;`') on, while `[...]` and `{{...}}` turn recognition of operators on, and turn recognition of sentence and phrase terminators off.

RECKON can store information as expressions. For example:

```
> (a person named `Jack') is husband of
+     (a person named `Jill') <--
> Y is wife of X <-- X is husband of Y
```

Here '`<--`' means 'is asserted' or 'is asserted if'. All the assertions that have been made can be queried:

```
> (a person named `Jill') is wife of
+     (a person named `Jack') ?
true
> (a person named `Jack') is wife of
+     (a person named `Jill') ?
false
> (a person named X) is wife of
+     (a person named `Jack') ?
```

```
X = `Jill'
> X is wife of (a person named `Jack') ?
X = (a person named `Jill')
> @1000001 is wife of (a person named `Jack') ?
true
> @1000001 = (a person named `Jill') ?
true
> @1000001 = (a person named `Jack') ?
false
```

You can also put tables into RECKON:

```
> ----------------------------------------------------------
> | header 1 | header 2 | header 3 | header 4 | header 5 |
> ----------------------------------------------------------
> | entry 11 | entry 12 | entry 13 | entry 14 | entry 15 |
> | entry 21 | entry 22 | entry 23 | entry 24 | entry 25 |
> | entry 31 | entry 32 | entry 33 | entry 34 | entry 35 |
> ----------------------------------------------------------
>
```

header 1	**header 2**	**header 3**	**header 4**	**header 5**
entry 11	entry 12	entry 13	entry 14	entry 15
entry 21	entry 22	entry 23	entry 24	entry 25
entry 31	entry 32	entry 33	entry 34	entry 35

Here the table ends with a blank line.

Usually tables are put in separate files that can be input by the '**`input`**' statement. If the above table were put in a file named **`F`**, the statement

```
> input F
```

would have the same effect as typing the table and its following blank line in directly had above. The table following blank line does not have to be included in the file.

One can use '**`=`**' in place of '**`-`**' and '**`||`**' in place of '**`|`**' to get double-lines in place of single lines. One can abbreviate by omitting all but the first 5 '**`-`**'s or '**`=`**'s on a line containing just these characters.

Lines can be continued with indented lines, but if the text of an entry is too large, the input can become messy. This can be alleviated in some cases by using macros. For example, if we changed the file **`F`** to contain:

```
-----------------------------------
| H  1 | H  2 | H  3 | H  4 | H  5 |
-----------------------------------
| E 11 | E 12 | E 13 | E 14 | E 15 |
| E 21 | E 22 | E 23 | E 24 | E 25 |
| E 31 | E 32 | E 33 | E 34 | E 35 |
-----------------------------------
H: header
E: entry
```

we get the same table as we had before. Here **`H`** is a macro replaced by the text '**`header`**', and **`E`** is a macro replaced by the text '**`entry`**'.

Entries can be glued together to make larger entries using the symbols '**`<`**', '**`>`**', and '**`^`**'. If an entry consists of just '**`<`**', this means the entry is to be glued to the entry on its left. Similarly '**`>`**' means to glue the entry to the entry on its right, and '**`^`**' means to glue to the entry above. For example, if the file **`F`** contained:

```
-----------------------------------
| H  1 | H  2 | H  3 | H  4 | H  5 |
===================================
| E 11 | E 12 | ABOT |   <  | E 15 |
-----------------------------------
| E 21 | E 22 |   >  |   ^  | E 25 |
-----------------------------------
| E 31 | E 32 | E 33 | E 34 | E 35 |
-----------------------------------
H: header
E: entry
ABOT: a bit of text
```

we would have:

```
> input F
```

header 1	**header 2**	**header 3**	**header 4**	**header 5**
entry 11	entry 12	a bit of text		entry 15
entry 21	entry 22			entry 25
entry 31	entry 32	entry 33	entry 34	entry 35

Gluing is transitive, so in the above 4 entries are all glued together.

Entries have associated formats. Explicit formats may be given to override a computed default format. A format consists of a word containing format items surrounded by '**/**'s. A format that applies to all entries in a row may be placed at the end of the row line. For columns, a line all of whose entries are formats for their columns may be given (the surrounding '/'s may be omitted for some but not all of these formats). A format may be given for a particular entry by making it the first word of the entry. And format may be given for the entire table on a line preceding the table.

For example, if the file **F** contained:

```
------------------------------------
| H  1 | H  2 | H  3 | H  4 | H  5 | /ib/
------------------------------------
| /l/  | /r/  | c    | P    | cb   |
|   11 |   12 |   13 | 5.4% | E 15 |
| E 21 | E 22 | E23  | 0.8% | E 25 |
| E 31 | E 32 | E 33 | 0.09 | E 35 | /!b/
------------------------------------
H: header
E: entry
E23: /b/ entry 23
P: /.0u%r/
```

we would have:

```
> input F
```

header 1	***header 2***	***header 3***	***header 4***	***header 5***
11	12	13	5.4%	**entry 15**
entry 21	entry 22	**entry 23**	0.8%	**entry 25**
entry 31	entry 32	entry 33	9.0%	entry 35

A short guide to some format items is:

`r`	right adjust
`l`	left adjust
`c`	center adjust
`b`	bold face
`i`	italicize
`.0`$^{+n}$	right adjust with n decimal places
`u%`	multiply by 100 and append a % sign
`u$`	add a $ sign
`i,`	add commas to integer parts of numbers

An entry may have four formats: a table format, a column format, a row format, and an entry format. These are combined as by set union, except that conflicts are resolved in favor of the entry format first, the row format second, the column format third, and the table format forth. Putting a '`!`' in front of a format item negates that item if it was included in the set union by a less favored format ('`!b`' for the last row above negates the '`b`' for the last column). Also, defaults are computed in case no format items are given for a particular format aspect: for example, if no adjust format is given entries are centered unless they are in a column whose entries all end with numbers some containing decimal points, in which case a default '`.0`$^{+n}$' format item is inferred.

Table entries can be computed from other table entries and from macros that are not actually in the table. The computational expressions are represented by '`[ ]`' bracketed lists included within entries. Names which are words beginning with '`<`' and ending with '`>`' can be assigned to rows, columns, and entries, after the manner of formats. For example, if the file **`F`** contains:

```
    ----------------------------------
    | Value   | Factor | Contribution |
    ----------------------------------
    | <V>     | <F>    |              |
    | /u$/    | /u%/   | /u$.00/      |
    | 100     | 20     | C            | <FIRST>
    | 200     | 5      | C            |
    | 300     | 8      | C            |
    ----------------------------------
    | /lb/ Total | <   | T            |
    ----------------------------------
    C: [(@,<V>) * (@,<F>)]
    T: [sum([<FIRST>, ..., @-1], @)]
```

then

```
> my table = input F
```

Value	**Factor**	**Contribution**
$100	20%	$20.00
$200	5%	$10.00
$300	8%	$24.00
Total		$54.00

Here **`(I,J)`** denotes the value of the cell at the **`I`**'th row and **`J`**'th column, **`@`** when used in **`I`** denotes the row of that cell, and when used in **`J`** denotes the column of that cell. So **`(@,<V>)`**

denotes the value of the cell in column **`<V>`** of the same row as the cell whose denotation contains the **`(<V>,@)`**.

Here **`[sum([<FIRST>, ..., @-1], @)]`** denotes the sum of the all the values in the column of a cell from the **`<FIRST>`** row to the row before that containing the cell. '**`@-1`**' denotes the row before the one containing the cell, and '**`[`**I, ..., J]' denotes the list of indices from **`I`** through **`J`**.

The last example also illustrates that a table is a description and its identifier can be made the value of a variable, such as '**`my table`**'. The expression '**`input F`**' has as its value the last thing in the file **`F`**, which in this example is the table. So '**`my table = input F`**' assigns the identifer of this table to the variable '**`my table`**'.

It is also possible to make a table whose rows are all the descriptions of a given type. Suppose file **`F`** contains:

```
-----------------------------------------------------
| Name    | Weight     | Height       | Age         | BMI   | /ib/
-----------------------------------------------------
| <name>  | <weight>  | <height>     | <age>       | B
| /l/     | /U`lb'/   | /U`ft_in'/   | /u`yr'.0/   | /.2/
| George  |    205    | 6ft 3in      | 25          |
| Mary    |    135    | 5ft 5in      | 26          |
----------------------------------------------
BMI: Body Mass Index
B: <body mass index>
ROW TYPE: person
SORT ROWS BY: name
```

then

```
> input F
```

Name	**Weight**	**Height**	**Age**	**Body Mass Index**
George	205 lb	6 ft 3 in	25	25.62
Jack	123 lb	5 ft 9 in	23	18.16
Jill	110 lb	5 ft 7 in	22	17.23
Mary	135 lb	5 ft 5 in	26	22.46

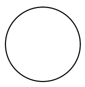
Here the '**`ROW TYPE`**' value '**`person`**' declares that the rows of the table are exactly all descriptions of type '**`person`**', while the '**`SORT ROWS BY`**' value '**`name`**' declares that the rows are to be sorted on the value of their '**`name`**' variable. The column names must match the description variable names: e.g., the column name **`<body mass index>`** matches the description variable name '**`body mass index`**'. Data rows in the table are added to the set of descriptions of type '**`person`**'. You can see this via:

```
> all persons
```

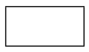
ID	type	name	weight	height	age	body mass index
`@1000000`	person	Jack	`123 lb`	`5 ft 9 in`	`23 yr 2 mo`	`18.1637`
`@1000001`	person	Jill	`110 lb`	`5 ft 7 in`	`21 yr 8 mo`	`17.2282`
`@1000047`	person	George	`205 lb`	`6 ft 3 in`	`25 yr 0 mo`	`25.6230`
`@1000048`	person	Mary	`135 lb`	`5 ft 5 in`	`26 yr 0 mo`	`22.4649`

RECKON also supports pictorial data that are expressions displayed as pictures:

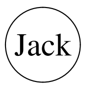
```
> x = {{circle 0.4}}
```



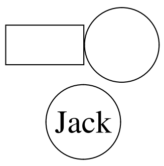
```
> y = {{rectangle (0.4,0.2)}}
```



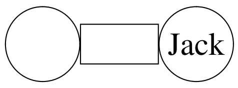
```
> z = {{(circle 0.4) labeled 'Jack'}}
```



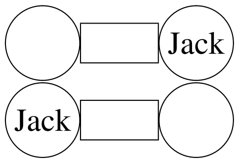
```
> {{([x] right of [y]) above [z]}}
```



```
> {{row([x],[y],[z])}}
```



```
> p = {{column (row([x],[y],[z]), row([z],[y],[x]))}}
```



```
> raw x
{{circle 0.4}}
> raw p
{{column (row (circle 0.4,
               rectangle (0.4,0.2),
               (circle 0.4) labeled 'Jack'),
          row ((circle 0.4) labeled 'Jack',
               rectangle (0.4,0.2),
               circle 0.4))}}
```

You can also change how an expression is displayed.

```
> display ( P ) <-- person ( P ) has name ( X ):
+     value = {{oval (0.4,0.2) labeled [X]}}
> (a person named 'Jack')
Jack
> ` [a person named 'Jill'] is wife of
+   [a person named 'Jack'] '
` Jill is wife of Jack '
```

Displays can be used to make demonstrations:

```
> for every demo:
+     on a demo with angle X:
+         angle = X
> x = a demo with angle 30 degrees
```

ID	type	angle
@1000043	demo	30 degrees

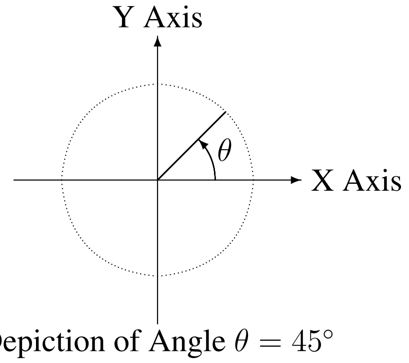
```
> for every demo:
+     on update THIS to X:
+         next angle = X
+     on increment THIS by X:
+         next angle = angle + X
> update x to 40 degrees
```

ID	type	angle
@1000044	demo	40 degrees

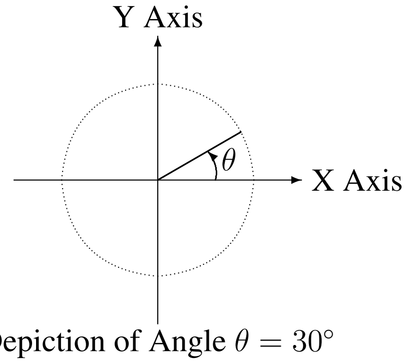
```
> increment x by 5 degrees
```

ID	**type**	**angle**
`@1000045`	demo	45 degrees

```
> display ( D ) <-- demo ( D ) with angle ( X ):
+     c = {{circle 1.0 dotted center (0.0,0.0)}}
+     x axis = {{arrow from (-0.75,0.0) to (0.75,0.0)}}
+     y axis = {{arrow from (0.0,-0.75) to (0.0,0.75)}}
+     line = {{line from (0.0,0.0)
+                    to [(0.5*cos X, 0.5*sin X)]}}
+     arc = {{arc arrow from (0.3,0.0)
+                        to [(0.3*cos X, 0.3*sin X)]}}
+     theta = {{Greek th}}
+     value = {{label (
+                'Depiction of Angle [theta]=[X]',
+                overlap (
+                  [c],
+                  [x axis] labeled 'X Axis',
+                  [y axis] labeled 'Y Axis',
+                  [line],
+                  [arc] labeled '[theta]' ) ) }}
> show x
```
See 'Depiction of Angle $\theta = 45°$'
```
> increment x by -15 degrees
```
See 'Depiction of Angle $\theta = 30°$'



Depiction of Angle $\theta = 45°$



Depiction of Angle $\theta = 30°$

In this example we first define a 'constructor' of the form '**`a demo with angle X`**' to make new **`demo`** descriptions, and then we define two 'methods', namely '**`update THIS to X`**' and '**`increment THIS by X`**', to change a **`demo`** description. Changing a **`demo`** description is like iterating a loop to make a new description.

Next we define how to display a **`demo`** description. Then we use the '**`show x`**' command to cause the **`demo`** description value of **`x`** to be displayed in a separate window. Every time this **`demo`** description changes, the window is updated, and every time the description is to be printed, 'See 'Depiction of Angle $\theta = \ldots°$' ' is printed instead, where 'Depiction ... ' is the **`label`** part of the display.

In extended example is given in the next section. The remainder of this document is a reference manual for RECKON.

4 Example

```
ship codes = ['S', 'D', 'C', 'B', 'A']
ship names = ['Submarine', 'Destroyer', 'Cruzier',
              'Battleship', 'Aircraft Carrier']
max ship length = 5
max tries = 100    // max tries to position ship on a board

on a board of size S <-- is an integer S:

    // Make a board of size SxS.
    size = S

    // state(R, C) is 'untouched', 'miss', 'hit',
    // or 'sunk' (if final hit on ship).
    state = an array of size (S, S)
            with initial element 'untouched'
    // ship(R, C) is ID of ship at (R, C) or 'none'.
    ship = an array of size (S, S)
           with initial element 'none'
    ships = 0  // Number of ships
    sunk  = 0  // Number of ships sunk

for every board:
```

```
on shoot at (R, C) on THIS <--
        is an integer R,  0 <= R < size,
        is an integer C,  0 <= C < size:
    if state(R,C) != ‘untouched’:
        value = ‘no effect’
    else if ship(R, C) == ‘none’:
        value = ‘miss’
    else:
        s <== ship(R, C)
        if hit(s) == ‘sunk’:
            update sunk = sunk + 1
            value = ‘sunk’
        else:
            value = ‘hit’
    if value != ‘no effect’:
        state(R, C) <== value

// Return true iff (DIRECTION,START) is an OK (empty)
// position for a new ship of length L.
//
on test position (DIRECTION, START) on THIS
        for ship of length L
    <-- is a pair of integers DIRECTION,
        is a pair of integers START,
        is an integer L, 1 <= L <= max ship length:

    (R,C) = START
    if R < 0 OR R >= size:
        value = ‘false’
    else if C < 0 OR C >= size:
        value = ‘false’
    else if ship(START) of B != ‘none’:
        // Position conflicts with previous ship
        value = ‘false’
    else if L == 1:
        value = ‘true’
    else
        value = position ( DIRECTION,
                              START + DIRECTION,
                              L - 1 )

    // Return a random empty place, (DIRECTION,POSITION),
    // where a ship of length L may be placed.
    //
    on find position on THIS for ship of length L
            <-- is an integer L, 1 <= L <= max ship length:
        value = loop:
            first tries = 1
            direction =
                random [ (-1, -1), (-1, 0), (-1, 1), (0, 1),
                         (1, 1), (1, 0), (1, -1), (0, -1) ]
            position = ( random ( 0 .. S - 1 ),
                         random ( 0 .. S - 1 ) )
            if tries > max tries:
                error `board too full to place ship'
            else if test position (direction, position) on THIS
                    for ship length L:
                value = (direction, position)
            else:
                update tries = tries + 1

    // Given an empty place (DIRECTION,START) for a ship S of
    // length L, install ship S on the board.
    //
    on place ship S of length L at (DIRECTION, START) on THIS
            <-- is a ship S,
                is a pair of integers DIRECTION,
                is a pair of integers START,
                is an integer L, 1 <= L <= max ship length:
        while L > 0:
            ship(START) <== S
            next START = START + DIRECTION
            next L = L - 1
        update ships = ships + 1

on a ship of length L on board B <--
```

```
        is an integer L, 1 <= L <= max ship length:
    // Make a ship of length L on board B.

    (direction, position) =
        find position on B for ship of length L
    place THIS of length L at (direction, position)
          on B

    length = L
    board = B
    hits = 0  // Number hits on ship so far

for every ship:
    on hit ( THIS ):
        update hits = hits + 1
        if update hits == length:
            value = 'sunk'
        else:
            value = 'not sunk yet'

on a board view for (BOARD) PLAYER
        ~ with square size SQSIZE ? <--
        board ( BOARD ),
        PLAYER :=: owner | opponent,
        SQSIZE :=? 0.25in,
        is a number SQSIZE, 0.05in <= SQSIZE:
    board = BOARD
    player = PLAYER
    square size = SQSIZE

    line width = 0.02in
    grid layer = B#00001#
    ship layer = B#00010#
    hit layer = B#00100#
    sunk layer = B#01000#
    letter layer = B#10000#
    plane = a plane with depth 5
            with x size (   2 * line width
                          + size * SQSIZE )
            with y size (   2 * line width
                          + size * SQSIZE )

    start = line width / 2
    stop = size * SQSIZE + line width / 2

    rcd = start
    rc = 0
    while rc < size:
        draw line in layer (grid layer)
                  on plane (plane)
                  with line width (line width)
                  from ( start, rcd ) to ( stop, rcd )
        draw line in layer (grid layer)
                  on plane (plane)
                  with line width (line width)
                  from ( rcd, start ) to ( rcd, stop )
        next rcd = rcd + SQSIZE
        next rc = rc + 1

    r = 0
    while r < size:
        next r = r + 1
        c = 0
        while c < size:
            next c = c + 1
            if ship(r, c) != none
                draw ship (ship(r, c)) on THIS

    on display ( THIS ) <--:
        value = plane

for every ship:

    // Update ships positions in board view.
    //
    on show ( THIS ) on ( VIEW ) <--
```

```
            is a board view ( VIEW ):
        assert board == VIEW’s board
        player = VIEW’s player
        plane = VIEW’s plane
        sqsize = VIEW’s square size
        hlwidth = VIEW’s line width / 2
ship layer = VIEW’s ship layer
hit layer = VIEW’s hit layer
sunk layer = VIEW’s sunk layer
letter layer = VIEW’s letter layer
        L = length
        P = position

        T = a 2D transform
            with expansion by ( sqsize, sqsize )
            with translation by (hlwidth, hlwidth )
        center = ( sqsize / 2, sqsize / 2 )
        while L > 0:
            state = board’s state(P)
            layers = 0
    if player == ‘owner’:
        layers = ship layer
    if state == ‘sunk’ or hits == L:
        layers = hit layer + sunk layer
    else if state == ‘hit’
        layers = hit layer

    draw filled polygon in (layers)
 on plane (plane)
 with vertices
      [T⋆P, T⋆(P+(0,1)), T⋆(P+(1,1) ),
    T⋆(P+(1,0))]

            if player == ‘owner’ or state == ‘sunk’:
                draw text (ship codes(length))
                     in (ship letter layer)
                     on plane (plane)
                     at (T⋆P + center)
```

```
next P = P + direction
next L = L - 1
```

5 Lexemes

Input text is a sequence of characters. This is scanned from left to right and top to bottom to produce a sequence of lexemes, which include words, marks, separators, numerics, and quoted strings.

For example, the input

```
x = 7ft 1_3/4 in;
y = `This is (we think) a sentence.'
```

contains in order the following lexemes:

`x`	word	`y`	word
`=`	mark	`=`	mark
`7ft`	numeric with embedded unit	`` ` ``	(leading) separator
`1_3/4`	numeric	`This`	word
`in`	unit	`is`	word
`;`	(trailing) separator	`(`	(strict) separator
		`we`	word
		`think`	word
		`)`	(strict) separator
		`a`	word
		`sentence`	word
		`.`	(trailing) separator
		`'`	(trailing) separator

Lexemes are defined more specifically as follows:

lexeme ::= *middle-lexeme* | *separator* | *quoted-string* | *comment*

strict-separator :::= **(** | **)** | **[** | **]** | { | } | **|**$^+$

strict-separator-character :::= **(** | **)** | **[** | **]** | { | } | **|**

quoted-string :::= " *character-representative** "

character-representative :::= *graphic-character* other than "
| *single-space-character*
| *special-character-representative*

graphic-character :::= character that prints some mark

lexical-item :::= *lexical-item-character*$^{+}$ not beginning with **//**

lexical-item-character :::= *graphic-character* other than *strict-separator-character* or **"**

lexical-item :::= *leading-separator** *middle-lexeme*$^{?}$ *trailing-separator**

middle-lexeme :::= *lexical-item*
not beginning with a *leading-separator*
or ending with a *trailing-separator*

leading-separator :::= **`**

trailing-separator :::= **'** | **!**$^{+}$ | **?**$^{+}$ | **.**$^{+}$ | **:**$^{+}$ | **;** | **,**

separator ::= *strict-separator*
| *leading-separator*
| *trailing-separator*

comment :::= **//** *comment-character**

comment-character :::= *graphic-character* | *horizontal-space-character*

Lexemes may be separated by ***white-space***, which is a sequence of white-space characters (single space, horizontal tab, form feed, etc.), but is not itself a lexeme.

white-space :::= *white-space-character*$^{+}$

white-space-character :::= *horizontal-space-character*
| *vertical-space-character*

horizontal-space-character :::= ***single-space-character*** | ***horizontal-tab-character***

vertical-space-character :::= ***line-feed-character***
| ***vertical-tab-character***
| ***form-feed-character***

The character set is UTF-8 encoded UNICODE.

The symbol '**:::=**' is used in syntax equations that define lexemes or parts of lexemes whose syntactic elements are character sequences that must not be separated by *white-space*. The symbol '**::=**' is used in syntax equations that define sequences of lexemes that may and sometimes must be separated by ***white-space***. *Comments* are treated as lexemes, but are discarded before the lexeme stream is transmitted to the parser.

Non-*white-space* control characters are ignored, as if they did not exist, except that a warning message is issued. However no warning is issued for ***carriage-return-characters*** that are next to a *vertical-space-character*.

The tabs stops used by the ***horizontal-tab-character*** are set every 8 columns.

`<NUL>`	nul	**`<DC1>`**	device control 1
`<SOH>`	start of heading	**`<DC2>`**	device control 2
`<STX>`	start of text	**`<DC3>`**	device control 3
`<ETX>`	end of text	**`<DC4>`**	device control 4
`<EOT>`	end of transmission	**`<NAK>`**	negative ack.
`<ENQ>`	enquiry	**`<SYN>`**	synchronous idle
`<ACK>`	acknowledge	**`<ETB>`**	end of transmission block
`<BEL>`	bell	**`<CAN>`**	cancel
`<BS>`	backspace	**`<EM>`**	end of medium
`<HT>`	horizontal tab	**`<SUB>`**	substitute
`<LF>`	line feed	**`<ESC>`**	escape
`<VT>`	vertical tab	**`<FS>`**	file separator
`<FF>`	form feed	**`<GS>`**	group separator
`<CR>`	carriage ret	**`<RS>`**	record separator
`<SO>`	shift out	**`<US>`**	unit separator
`<SI>`	shift in	**`<SP>`**	single space
`<DLE>`	data link escape	**`<DEL>`**	delete
`<NL>`	new line (equals **`<LF>`**)	**`<Q>`**	double quote (`"`)

special-character-representative ::= one of the above
| **<** *decimal-digit hexadecimal-digit** **>**

Figure 1: Special Character Representatives

Every ***vertical-space-character*** causes the current position to return to the beginning of a new line. ***Horizontal-space-characters*** before a *vertical-space-character* are ignored, as if they did not exist. Warning messages are issued for ***form-feed-characters*** and ***vertical-tab-characters*** that occur when the column position is not at the beginning of a line.

The definition of a ***middle-lexeme*** is unusual: it is what is left over after removing ***leading-separators*** and ***trailing-separators*** from a ***lexical-item***. The lexical scan first scans a *lexical-item*, and then removes *leading-separators* and *trailing-separators* from it. Also *trailing-separators* are removed from the end of a *lexical-item* by a right-to-left scan, and not the usual left-to-right scan which is used for everything else. Thus the *lexical-item* ‘**`` `4,987,,::' ``**’ yields the *leading-*

separator ‘**`**’, the *middle-lexeme* ‘**4,987**’, and the four *trailing-separators* ‘**,**’ ‘**,**’, ‘**::**’, and ‘**'**’.

Middle-lexemes, *separators*, and *quoted-strings* can represent ***symbols*** that store character strings. The character strings represented by *middle-lexemes* and *separators* are exactly the character strings of these lexemes. Some *middle-lexemes* can alternatively represent numbers.

The string of characters represented by a ***quoted-string*** is recognized by a left-to-right scan in which each *character-representative* is translated into one UNICODE character. Most characters simply translate to themselves, but *special-character-representatives* are sequences of characters that represent just one UNICODE character. The beginning and ending quotes (**"**) are <u>not</u> included in the represented character string.

Figure 1 $^{p\,31}$ defines *special-character-representatives*. Mnemonics are given for ASCII control characters and the **"** character. For other characters, any hexadecimal number in **< >** brackets that begins with a decimal digit is interpreted as representing the UNICODE character whose character code is the given hexadecimal number. Thus that **<FF>** represents the ASCII form feed, while **<0FF>** represents the UNICODE ÿ character which has character code FF in hexadecimal.

Middle-lexemes are further classified as follows:

> ***middle-lexeme*** ::= *word* | *numeric* | *mark*
>
> ***word*** ::= *middle-lexeme* that
> (1) contains a letter
> and (2) does not contain any **#**
> and (3) does not have any digit before its first letter
> and (4) does not have any digit adjacent to a character that is neither a letter nor a digit
>
> ***numeric*** ::= *middle-lexeme* that
> (1) contains a digit or both a **#** and a letter
> and (2) is not a *word*
>
> ***mark*** ::= *middle-lexeme* that is not a *word* or *numeric*

There are many special kinds of *numerics*:

	Section	Examples	
natural-numeric	5.1 $^{p\,34}$	**`43`**	**`9,587`**
integral-numeric	5.1 $^{p\,34}$	**`+43`**	**`-9,587`**
fractional-numeric	5.2 $^{p\,35}$	**`4/3`**	**`-1_1/3`**
scientific-numeric	5.3 $^{p\,35}$	**`4.3`**	**`-0.0098`**
date	5.5 $^{p\,38}$	**`4may2003`**	**`Fri-May-03`**
time-interval	5.6 $^{p\,39}$	**`4:30`**	**`4:30:23.78`**
time	5.6 $^{p\,39}$	**`4:30pm`**	**`20may05:12:30pm-EDT`**
radix-numeric	5.4 $^{p\,36}$	**`x#4b3e#`**	**`b#1.00110101001#e+3`**

Some other kinds of special *middle-lexemes* are:

	Section	Examples	
unit	5.7 $^{p\,42}$	**`mi`**	**`mi/hr`**
format	5.8 $^{p\,44}$	**`/.1/`**	**`/10.1c/`**

Non-*numerics* can be used to form names of variables, constants, and functions:

name ::= { *word* | *quoted-mark* | *quoted-separator* | *quoted-word* }
{ *word* | *quoted-mark* | *quoted-separator* | *quoted-word* | *natural-index* }

quoted-mark :::= " *mark* "
quoted-separator :::= " *separator* "
quoted-word :::= " *word* "
natural-index :::= `0` | *non-zero-digit digit** [there can be at most 15 *digits*]

Note that digits can appear adjacent to leters in a name, or in *natural-indices* that cannot begin a name, but not elsewhere. *Natural-indices* cannot have high order zeros. '**`#`**'s can appear only in *quoted-marks*.

There is no difference between a *word* and a *quoted-word* in the name being represented, e.g., no difference between **`X`** and **`"X"`**. However, sometimes words must be quoted to prevent them from being recognized as operators, e.g., **`"AND"`**.

If a name is to be used as a constant it must be `` ` ' `` quoted. However, not everything so quoted is a name.

Some examples are:

```
"+" = 5
X = "+" + 1
Y15 = `X + 3'
Z0 = `X'
```

```
    T = `"A line.<LF>"'
```

These statements define a variable named **"+"** with value **5**, a variable named **X** with value **6**, and a variable named **Y15** with value the text **X + 3**, i.e., the vector whose 3 elements are the symbols **X**, **+**, and **3**. **Z0** is a variable whose value is a single symbol whose character string contains the single character **X**. **T** is a variable whose value is the single symbol whose character string is that of the *quoted-string* **"A line.<LF>"**.

Some *names* have builtin definitions, such as the words '**ft**' and '**inch**' which in some contexts denote the units 'foot' and 'inch' (see Section 5.7 $^{p\,42}$), and '**PI**' which in some contexts denotes the mathematical constant π (see Figure 2 $^{p\,35}$).

Special lexemes are described in the following sections. Some of these represent numbers. When translated into internal computer form, which is IEEE double precision floating point, integers in the range

$$-9,007,199,254,740,992 \quad .. \quad +9,007,199,254,740,992$$

are exactly represented, and numbers with absolute values in the range

$$2.2250738585072014 \times 10^{-308} \quad .. \quad 1.7976931348623157 \times 10^{+308}$$

are represented with a relative error of $2^{-53} \approx 1.110223 \times 10^{-16}$, or 15.95 decimal digits precision. Numbers with absolute value less than $2.2250738585072014 \times 10^{-308}$ are represented as zero.

Numbers with absolute value equal to or greater than $1.7976931348623158 \times 10^{+308}$ are represented a $+$ or $-$ infinity, which can also be represented by the builtin constants **+INF** and **-INF**: see Figure 2 $^{p\,35}$.

Internally $+0$ and -0 are distinct, though in most computations they behave the same. On output both are represented by **+0**.

5.1 Integral Numerics

A ***natural-numeric*** is a *numeric* with contains only digits and commas. An ***integral-numeric*** is a *natural-numeric* optionally preceeded by a *sign*:

natural-numeric :::= *digit*$^+$ | *digit*$^?$ *digit*$^?$ *digit* { **,** *digit digit digit* }$^+$

integral-numeric :::= *sign*$^?$ *natural-numeric*

sign ::= **+** | **-**

Note that commas, if present, must appear every 3 digits.

`PI`	3.141592653589793	mathematical π
`MATH E`	2.718281828459045	mathematical e, Euler's constant
`+INF`	+ infinity	number too positive to represent
`-INF`	- infinity	number too negative to represent

Figure 2: Builtin Constants

5.2 Fractional Numerics

A ***fractional-numeric*** is an *integral-numeric* followed by either a single `/` and one or more *digits*, or by a _, one or more *digits*, a `/`, and one or more *digits*:

fractional-numeric :::= *integral-numeric* `/` *digit*$^+$
| *integral-numeric* _ *digit*$^+$ `/` *digit*$^+$

Some examples are:

```
1/3          +48/97          42_3/8          -10_7/10
```

Here '_' denotes 'and', as in **`10`** and **`7/10`**'ths, so the last example equals **`-10.7`**.

Internally a fraction is stored as an IEEE double precision floating point number. If an internally stored number can print as a fraction with a small denominator more accurately than it can be printed with some given number of decimal places, it may in some contexts be printed as a fraction. Thus in these contexts a number input as **`1/3`** will print as **`1/3`** and not as **`0.333333`**.

The fraction **`0/0`** denotes the IEEE ***NaN*** value, which stands of 'Not a Number'. The fraction N**`/0`** for $N > 0$ denotes **`+Inf`**, or plus infinity, while **`-`**N**`/0`** denotes **`-Inf`**, or minus infinity.

5.3 Scientific Numeric

A ***scientific-numeric*** is a *numeric* consisting of an *integral-numeric* optionally followed by a *fractional-part* which is optionally followed by an *exponent-part*:

scientific-numeric ::= *integer-numeric fractional-part*$^?$ *exponent-part*$^?$

fractional-part :::= **`.`** *digit*$^+$ | **`.`** { *digit digit digit* **`,`** }$^+$ *digit digit*$^?$ *digit*$^?$

exponent-part ::= *exponent-indicator sign*$^?$ *digit*$^+$

exponent-indicator ::= **`e`** | **`E`**

Commas must be every three *digits* from the end of a *integral-numeric* or the beginning of a *fractional-part*. If the *scientific-numeric* contains a decimal point, it must be preceeded by a digit (write **`0.5`** and not **`.5`**). An *exponent* shifts the decimal point (including the implied decimal point at the end of the *scientific-numeric* that has no decimal point); a positive exponent shifts right and a negative exponent shifts left. There may be at most one point. Commas may be used before the point without being used after the point, and vice versa. Exponents cannot contain commas.

Some examples are:

```
123e0     123e-2     123e-321        1,234e9
123E0     123E-2     123E-321        0.123,456e-3
.123e3    .123e+1    0.123,456e-3    1,234.567890e6
```

5.4 Radix Numerics

A ***radix-numeric*** is the same as a *scientific-numeric*, except that it is prefaced (after any *sign*) by a *radix-indicator*, the non-exponent digits are surrounded by '**`#`**'s, and the permissible *digits* and comma spacings depend upon the radix indicator:

radix-numeric ::=
sign$^?$ *radix-indicator* **`#`** *radix-natural-part radix-fractional-part*$^?$ **`#`** *exponent-part*$^?$

sign ::= **`+`** | **`-`**

radix-indicator ::= **`B`** | **`b`** | **`O`** | **`o`** | **`D`** | **`d`** | **`X`** | **`x`**

radix-natural-part :::= *radix-digit*$^+$ { `,` *radix-digit*$^+$ }* [see text]

radix-fractional-part :::= `.` *radix-digit*$^+$ { `,` *radix-digit*$^+$ }* [see text]

exponent-part ::= *exponent-indicator sign*$^?$ *digit*$^+$

exponent-indicator ::= **`e`** | **`E`**

The *radix-indicators* have the following denotations:

Radix Indicator	**Meaning**	**Digits**	**Comma Positions**
B or **b**	binary	**0**, **1**	every 2, 3, 4, or 5 digits
O or **o**	octal	**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**	every 2, 3, or 4 digits
D or **d**	decimal	**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7** **8**, **9**	every 3 digits
X or **x**	hexa-decimal	**0**, **1**, **2**, **3**, **4**, **5**, **6**, **7** **8**, **9**, **A**, **B**, **C**, **D**, **E**, **F**, **a**, **b**, **c**, **d**, **e**, **f**	every 2 or 4 digits

Only certain *radix-digits* are permitted: e.g., the digit **2** is not allowed in **B#...#** and the digit **E** is not allowed with **D#...#**. Only certain comma spacings are permitted, and comma spacing must be consistent. Thus **x#A,C9,BD,8F#** is permitted but **x#A,C9B,D8F#** is not (3-spacing not allowed for hexa-decimal), and **x#A,C9,BD8F#** is not permitted (2-spacing combined with 4-spacing is not consistent). There may be at most one decimal point. Commas may be used before the point without being used after the point, and vice versa, but if used in both places, the comma spacing must be the same.

Some example *radix-numbers* are:

```
b#10110100#     o#7534201#     d#9758#       x#E8A932B#
b#1011.0100#    o#75.34201#    d#97.58#      x#E8,A9.32B#
b#101101#E2     o#0.7753#      d#97.58#e5    X#0.fe8a#e-4
B#10,1101#E+2   O#2,3456#E-3   D#0.1235#     X#E8A.7CCD,83#
B#101,101#e2    O#0.2345,6#    D#0.123,5#    X#E8A,932B.7C#
```

Just as for *scientific-numbers* $^{p\,35}$ the *exponent-part* shifts the decimal point right (if +) or left (if -) by some number of digits (in the given radix). The *digits* in the *exponent-part* are always decimal.

5.5 Dates

A ***date*** is lexeme that denotes an integer which gives the ***Julian Day Number*** of the day times the number of seconds in one day ($60 * 60 * 24 = 86,400$).[4] Julian Day **0** is assigned to January 1, 4713BC. Numerically, **01/01/4713BC == 1jan4713BC == 4713BC == 0**, and every increase of this date by one day adds **86,400**. We interpret the number of a given date as corresponding to 12:00am on the date, the time the designated day begins.[5]

[4] On days in which there are leap seconds, the last second of the day is either double-length or zero-length, following the Network Time Protocol (NTP).

[5] Julian Day Numbers are used for dates in astronomy, but the astronomers start the day at 12noon instead of 12am. Day 2,457,538 begins at 12noon on 29may2016 and ends at 12noon on 30may2016, so 12:00am on 29may2016 corresponds to Julian Day 2,457,537.5 in astronomy, but to 2,457,538 * 86,400 in our system.

A variety of different date formats may be used:

date :::= *weekday-modifier*$^?$ *date-numeric era*$^?$
  | *weekday-modifier*$^?$ *day month-name year-name era*$^?$
  | *weekday-modifier*$^?$ *month-name* **`-`** *year-name era*$^?$
  | *weekday-modifier*$^?$ *month-name short-year-name era*$^?$
  | *year-name era*
  | *short-year-name*

weekday-modifier :::= *weekday* **`-`**

weekday :::= **`Mon`** | **`Monday`**
  | **`Tue`** | **`Tues`** | **`Tuesday`**
  | **`Wed`** | **`Wednesday`**
  | **`Thu`** | **`Thur`** | **`Thursday`**
  | **`Fri`** | **`Friday`**
  | **`Sat`** | **`Satday`**
  | **`Sun`** | **`Sunday`**

date-numeric :::= *month*`/`*day*`/`*year*

month :::= *digit*$^+$

day :::= *digit*$^+$

year :::= *digit digit*$^+$

month-name ::= **`jan`** | **`Jan`** | **`January`**
  | **`feb`** | **`Feb`** | **`February`**
  | **`mar`** | **`Mar`** | **`March`**
  | **`apr`** | **`Apr`** | **`April`**
  | **`may`** | **`May`** | **`May`**
  | **`jun`** | **`Jun`** | **`June`**
  | **`jul`** | **`Jul`** | **`July`**
  | **`aug`** | **`Aug`** | **`August`**
  | **`sep`** | **`Sep`** | **`September`**
  | **`oct`** | **`Oct`** | **`October`**
  | **`nov`** | **`Nov`** | **`November`**
  | **`dec`** | **`Dec`** | **`December`**

year-name :::= *year* | *short-year-name*

short-year-name :::= **`'`** *digit digit*$^+$

era ::= **`AD`** | **`BC`** | **`ADG`** | **`ADJ`** | **`BCG`** | **`BCJ`**

Weekdays are superfluous; if they are input and are wrong, a warning message is issued.

The default *era* is **`AD`**. A '**`G`**' at the end of an *era* indicates the date is in the Gregorian Calendar, and a '**`J`**' indicates the date is in the Julian Calendar. If neither is given, the date is Gregorian if it is on or after 15 October 1582, and Julian otherwise. There is no year **`0`**; **`1AD`** is immediately preceeded by **`1BC`**.

A date consisting of just a *year-name* refers to January 1 of that year; e.g., **`'05`** and **`01/01/2005`** are the same date. A date consisting of just a *month-name* and a *year-name* refers to the first day of the month; e.g., **`May'05`**, **`May-05`**, and **`05/01/2005`** are the same date.

If no *era* is given and the *year* is **`999`** or less, the *year* is taken to denote the year nearest the current year which ends in the same digits as given in *year*. Thus in 2014 **`97`** denotes 1997 and **`114`** denotes 2114.

A warning message is issued if *month* is out of the range **`1..12`**, but the date is still computed letting **`0`** denote December of the previous year, **`13`** denote January of the next year, etc. A warning message is issued if *day* is **`0`** or too large for the given month and year, but again the date is still computed with **`0`** denoting the last day of the previous month, etc.

All *dates* are *numerics*, as either a digit comes before any letter, or a digit is next to a '**`-`**'. Some examples are:

```
01/01/2000   Jan-2000   jan-00   15mar44BC   Thursday-03sep1752
'09          101BC      101BCJ   101BCG      Mon-08/23/99AD
```

5.6 Times

A ***time*** is a *numeric* containing digits, possibly a decimal point, and one or two colons, optionally followed by a meridiem indication and/or a time zone, and optionally preceeded by a date. Dates and sometimes time zones are separated from the rest by '**`-`**'s.

time ::= *time-interval*
  | *time-of-day*
  | *date* **`-`** *time-of-day*

time-interval ::= *hour* **`:`** *minute* { **`:`** *seconds* }$^?$

time-of-day ::= *time-interval* { **`-`**$^?$ *meridian* }$^?$ { **`-`** *time-zone* }$^?$
  | *time-interval* **`-`**$^?$ *time-zone*

hour ::= *digit*$^?$ *digit*

minute :::= *digit digit*

seconds :::= *digit digit fractional-seconds*$^?$

fractional-seconds :::= `.` *digit*$^+$ | `.` { *digit digit digit* `,` }$^+$ *digit digit*$^?$ *digit*$^?$

meridiem ::= **`am`** | **`AM`** | **`pm`** | **`PM`** | **`noon`** | **`midnight`**

time-zone :::= **`UTC`**
| **`UTC`** *sign digit*$^?$ *digit* { `:` *digit digit* }$^?$
| *defined-time-zoned*

A *time-interval* represents the number $60 * (60 * hour + minute) + seconds$. A *minutes* or *seconds* value equal to or greater than 60 merely causes a warning message to be issued.

A *meridiem* limits the *time-interval* to be equal to or greater than **`1:00`** and strictly less than **`13:00`**; else a warning message is issued. A **`noon`** or **`midnight`** *meridiem* limits the *time-interval* to be equal to **`12:00`**; else a warning message is issued. The absence of a *meridiem* limits the *time-interval* to be equal to or less than **`24:00`**.

If **`am`** or **`AM`** is given, 12 hours is subtracted from *time-interval* values equal to or greater than **`12:00`**. If **`pm`** or **`PM`** is given, 12 hours is added to *time-interval* values that are strictly less than **`12:00`**. The time **`12:00noon`** is interpreted as **`12:00`** and the time **`12:00-midnight`** is interpreted as **`24:00`**.

A *time-zone* such as **`UTC-05:00`** refers to Greenwich Mean Time (**`UTC`**, Temps Universel Coordonné in French) minus 5 hours (**`-05:00`**) and is the same as Eastern Standard Time, or **`EST`**. When localities that use **`EST`** go on Daylight Savings Time, they convert to Eastern Daylight Time, or **`EDT`**, which is that same as **`UTC-04:00`**.

As another example, **`NZST`**, New Zealand Standard Time, is the same as **`UTC+12:00`**.

If a *time-zone* is given, the difference between the *time-zone* time and GMT is subtracted from the *time-interval*. Thus for **`UTC-5:00`** 5 hours is added to the *time-interval*, and for **`UTC+12:00`** 12 hours is subtracted from the *time-interval*. It is possible to get negative numbers, which indicate a time in the previous GMT day, or numbers equal to or larger than **`24:00`**, which indicate a time in the next GMT day.

The standard time zones pre-defined in RECKON are given in Figure 3.

Additional time zones can be defined, or existing time zone definitions can be replaced, by the statement:

`define time zone` *short-name* `'`*long-name*`'` **`UTC`**±*offset*

as for example

		UTC
HST	Hawaii Standard Time	-10:00
AKST	Alaska Standard Time	-09:00
AKDT	Alaska Daylight Time	-08:00
PST	Pacific Standard Time	-08:00
PDT	Pacific Daylight Time	-07:00
MST	Mountain Standard Time	-07:00
MDT	Mountain Daylight Time	-06:00
CST	Central Standard Time	-06:00
CDT	Central Daylight Time	-05:00
EST	Eastern Standard Time	-05:00
EDT	Eastern Datlight Time	-04:00
AST	Atlantic Standard Time	-04:00
ADT	Atlantic Daylight Time	-03:00
ART	Argentina Time	-03:00
BRT	Brasilia Time	-03:00
BRST	Brasilia Summer Time	-02:00
GMT	Greenwich Mean Time	+00:00
WET	Western European Time	+00:00

		UTC
BST	British Summer Time	+01:00
CET	Central European Time	+01:00
WAT	West Africa Time	+01:00
CEST	Central European Summer Time	+02:00
EET	Eastern European Time	+02:00
CAT	Central Africa Time	+02:00
EEST	Eastern European Summer Time	+03:00
EAT	Eastern Africa Time	+03:00
GST	Gulf Standard Time	+04:00
PKT	Pakistan Standard Time	+05:00
IST	India Standard Time	+05:30
CST	China Standard Time	+08:00
JST	Japan Standard Time	+09:00
AWST	Australia Western Standard Time	+09:00
AWDT	Australia Western Daylight Time	+10:00
ACST	Australia Central Standard Time	+09:30
ACDT	Australia Central Daylight Time	+10:30
AEST	Australia Eastern Standard Time	+10:00
AEDT	Australia Eastern Daylight Time	+11:00
NZST	New Zealand Standard Time	+12:00
NZST	New Zealand Daylight Time	+13:00

Figure 3: Pre-Defined Time Zones

```
define time zone WGST 'West Greenland Summer Time' UTC-02:00
define time zone AZOST 'Azores Summer Time' UTC+00:00
define time zone BST 'Bangladesh Standard Time' UTC+06:00
       // Replaces British Summer Time, UTC+01:00
define time zone VLAT 'Vladivostok Time' UTC+10:00
```

Short names may be capitalized or not in defining statements, and similarly in usage.

All *times* are *numerics*, as either a digit comes before any letter, or a digit is next to a '`-`'. Some examples are:

```
1:15                 12:00                 05:37:23
1:15am               12:00noon             05:37:23PM
1:15edt              12:00est              05:37:23WDT
1:15am-edt           12:00midnight-EET     05:37:23pm-WDT
May1975-1:15am-edt   05/04/1999-00:00EET   05/07/99-17:37:23WDT
may1975-1:15am-edt   04may'99-00:00EET     07May99-17:37:23WDT
```

5.7 Units

Values may have units, such as '**`ft`**' for 'feet' or '**`inch`**' for inch ('**`in`**' is useful as a preposition and is therefore not used as a unit). Units can be combined in a single lexeme by using the operators '**`/`**' for division, '**`*`**' for multiplication, and '**`^`**' for exponentiation:

unit :::= *unit-product*
  | *unit* **/** *unit-product*
unit-product :::= *unit-factor*
  | *unit-product* ***** *unit-factor*
unit-factor :::= *unit-name*
  | *unit-name* **^** *natural-numeric*

Some examples are

```
ft          inch          m            kg
lb*ft       inch^3        kg*m         kg*m^2
ft/sec      ft/sec/sec    m/sec^2      lb*ft/sec
ft^2/sec    ft/sec*sec    sec^2/m^2    sec/lb*ft
```

Note that '**`*`**' takes precedence over '**`/`**', and '**`^`**' takes precedence over '**`*`**'.

Units can be separate *word* lexemes, or they can be glued to a preceding *scientific-numeric* lexeme, from which they will be automatically split. In addition several *glued-lexemes* of this kind can themselves be glued together:

glued-lexeme :::= *scientific-numeric* $\_^{?}$ *unit* { $\_^{?}$ *scientific-numeric* $\_^{?}$ *unit* }*
  | other

Thus '**`5inch`**' is a *glued-lexeme* that will be split into '**`5 inch`**', '**`4ft3inch`**' will be split into '**`4 ft 3 inch`**', and '**`4ft_3inch`**' and '**`4_ft_3_inch`**' will also be split into '**`4 ft 3 inch`**'.

Units may be defined by statements of the form:

`define unit` *unit-name* { **=** *unit-expression* }$^{?}$

Here

> ***unit-expression*** ::= *unit-quotient*
> | *unit-quotient* **+** *unit-number-constant*
> | *unit-quotient* **−** *unit-number-constant*
>
> ***unit-quotient*** ::= *unit-product*
> | *unit-quotient* **/** *unit-number-constant*
> | *unit-quotient* **/** *unit-product*
> | *unit-number-constant* **/** *unit-product*
>
> ***unit-product*** ::= *unit-factor*
> | { *unit-number-constant* *unit-name* }$^+$
> | *unit-number-constant* ***** *unit-factor*
> | *unit-product* ***** *unit-number-constant*
> | *unit-product* ***** *unit-factor*
>
> ***unit-factor*** ::= *unit-name* { **^** *natural-numeric* }$^?$
>
> ***unit-number-constant*** ::= *scientific-numeric* without any leading *sign*

Note that operators in *unit-expressions* are separate lexemes. Also, the second operand of '**+**', and '**-**' must always be a *unit-number-constant*, and it is not possible to write non-integral or negative exponents.

Standard definitions are given in Figure 4 $^{p\,44}$.

A unit may appear in many '**define unit**' statements in order to describe its relation to other units. The statements must be consistent. An example of a definition consistent with the pre-defined definitions is:

```
define unit inch = yd / 36
```

This is automatically checked by computing:

```
inch = yd / 36 = (3 ft) / 36 = 3 * (12 inch) / 36
               = (36 inch) / 36 = 1 inch
```

Equations will be automatically derived from '**define unit**' statements when such can be done by linear algebra and simple substitution; e.g. '`yd = 36 inch`' and '`sqft = 144 inch^2`' will be derived automatically when needed.

```
define unit m                         // meter
define unit cm = 0.01m                // centimeter
define unit inch = 2.54cm             // inch
define unit ft = 12 inch              // foot
define unit yd = 3ft                  // yard
define unit sqft = ft^2               // square foot
define unit cuft = ft^3               // cubic foot
define unit degC                      // degree Centigrade
define unit degF = 1.8degC + 32       // degree Farenheit
define unit degK = degC + 273.15      // degree Kelvin
define unit g                         // gram
define unit oz = 28.349523g           // ounce
define unit lb = 16oz                 // pound
define unit sec                       // second
```

Figure 4: Pre-Defined UNITS

5.8 Formats

A ***format*** tells how to print a value, usually a number, but sometimes a symbol (character string). *Formats* are *lexemes* that begin and end with '**/**'.

Some examples are (where ␣ denotes a single space character):

Value	Format	Printout	Value	Format	Printout
`5.4321`	`/0.4/`	`5.4321`	`Hello`		`Hello`
`5.4321`	`/0.4rm/`	5.4321	`Hello`	`/rm/`	Hello
`5.4321`	`/0.4ss/`	5.4321	`Hello`	`/ss/`	Hello
`5.4321`	`/0.2b/`	**5.43**	`Hello`	`/b/`	**Hello**
`5.4321`	`/w5sc/0.2/`	␣5.43	`Hello`	`/rw7sc/`	␣␣HELLO
`5.4321`	`/cw6sl/0.2/`	␣*5.43*␣	`Hello`	`/w7/sl/`	␣*Hello*␣
`5.4321`	`/w6i/0.2/`	␣␣*5.43*	`Hello`	`/lw7i/`	*Hello* ␣␣

A *format* is a *word* or *numeric* consisting of a sequence of *format-items* surrounded by and optionally separated by slashes '**/**':

format ::= **/** *format-item* { **/**$^?$ *format-item* }* **/**

The following is a complete list of *format-items*. All *format-items* contain either a letter or a digit, so all formats are either *words* or *numerics*. The slashes (`/`) between *format-items* in a *format* may be omitted if no ambiguity arises.

The items are presented in groups whose items, unless otherwise noted, conflict with each other; see the rules below for resolving conflicts.

Font Style Group (non-monospace unless indicated)

`ty`	typewriter (monospace) [default]
`rm`	roman (normal non-monospace)
`ss`	sans-serif
`b`	bold
`i`	italic
`sl`	slanted
`sc`	small caps

Font Size Group

`fs`<*ratio*>	adjust the font size to *ratio*% of normal.

Background Color Group

`gray`	light gray background
`white`	white background [default]

Adjust Group

`r`	right adjust [default for numbers]
`l`	left adjust
`c`	center adjust [default for symbols]

Numeric Group

`.`<*places*>	output with exactly *places* decimal places
`0.`<*places*>	ditto but with `0` in the units position if less than `1`
`0`	ditto but with zero *places* and <u>no</u> decimal point
`p`<*precision*>	output with exactly *precision* significant decimal digits, but avoid using exponent if it is more compact not too, suppress low order fraction `0`'s, and suppress decimal point if there are no fraction digits [default is '`p6`']
`T'`*time-format*>`'`	format time (or date) as per below; conflicts with `% $ , () +` and `-`

Fraction Group

`f`<*denominator*>	when a number is about to be output with a fractional part, prefer a fractional representation with denominator less than or equal to the value given, if that is more precise.
`f2^`<*exponent*>	ditto but denomenator must be a power of 2 equal to or less than $2^{exponent}$.
`!f`	do not use fractional representations with demonimators [default]

Unit Group

`u%`	multiply numbers by 100 and add a % sign
`u$`	add $ sign to numbers; make **`0.2`** the Numeric Group default
`u'`<*unit*>**`'`**	the numbers are to be expressed in the given <*unit*>'s, where <*unit*> is as per Section 5.7 $^{p\,42}$, but the <*unit*> itself is not to be included in the entry
`U'`<*unit*>**`'`**	ditto but the <*unit*> itself is to be included in the entry
`!u`	no unit [default]

Sign Group

`s()`	indicate negative numbers by parentheses instead of sign
`s+`	indicate positive and zero numbers by sign, + or -
`s-`	indicate negative numbers by sign (-) and positive by absence of sign [default]

Integer Comma Group

`i,`	add commas to integral parts of numbers [default]
`!i,`	undo ditto

Fraction Comma Group

`f,`	add commas to fractional parts of numbers
`!f,`	undo ditto [default]

Width Group

`w`<*columns*>	adjust width by given number of *columns*
<*columns*>	ditto
`!w`	set width to 'only as wide as necessary' [default]

Height Group

`h`<*lines*>	adjust height by given number of *lines*
`!h`	set height to 'only as high as necessary' [default]

Horizontal Margin Group (`left` and `right` do not conflict)

`side`<*points*>	adjust space to left and right of entry by *points*
`left`<*points*>	adjust space to left of entry by *points*
`right`<*points*>	adjust space to right of entry by *points*
	[default is '`side3`']

Vertical Margin Group (no conflicts)

`top`<*points*>	adjust space at top of entry by *points*
`bottom`<*points*>	adjust space at bottom of entry by *points*
	[default is '`top3bottom3`']

To 'adjust' a value V by a parameter P in the above is to set V to P if P has no sign, or to add P to the current V if P has a sign. Thus '`w7`' sets the width to 7 columns while '`w-2`' subtracts two columns from the current width and '`w+2`' adds two columns to the current width.

The *format-items* are applied left to right, with a *format-item* cancelling conflicting *format-items* to its left. In applying this rule, however, *format-items* that adjust a value by adding or subtracting a parameter to the value do not conflict with any *format-item*.

In some contexts there are multiple formats applied in a given order. For example, in tables the table format, column format, and row format are applied in order, so column format items override conflicting table format items, and row format items override conflicting table and column format items. A format item does not override another item with which it does not conflict.

A *time-format* is a string of any characters surrounded by `` ` ' ``s that are allowed inside a *middle-lexeme* in which the following ***time-codes*** are replaced as indicated in Figure 5. Note it is in the nature of formats that the `` ` ' ``s are not at the ends of the *middle-lexeme* and are therefore not recognized as separators.

`\`	delete the ‘`\`’ and keep the following character without giving it a special interpretation (`\'` can be use to include `'` in a <*time-format*>)
`dd`	day of month: **`01`** ... **`31`**
`ddd`	day of year: **`01`** ... **`366`**
`ww`	day of weak: **`01`** ... **`07`**
`www`	day of weak: **`sun`** ... **`sat`**
`Www`	day of weak: **`Sun`** ... **`Sat`**
`WWW`	day of weak: **`SUN`** ... **`SAT`**
`mm`	month: **`01`** ... **`12`** if next to **`/`**
`mmm`	month: **`jan`** ... **`dec`**
`Mmm`	month: **`Jan`** ... **`Dec`**
`MMM`	month: **`JAN`** ... **`DEC`**
`yy`	year: **`00`** ... **`99`**
`yyyy`	year: **`0000`** ... **`9999`**
`xm`	meridian: **`am`** or **`pm`**
`XM`	meridian: **`AM`** or **`PM`**
`hh`	hour: **`00`** ... **`12`** if xm or XM in *time-format* or: **`00`** ... **`23`** if xm and XM <u>not</u> in *time-format*
`mm`	minute: **`00`** ... **`59`** if next to **`:`**
`ss`	second: **`00`** ... **`59`**
`_`	single space
`zzz`	local time zone (e.g. **`est`**), and adjust time to be in that zone
`ZZZ`	local time zone (e.g. **`EST`**), and adjust time to be in that zone
`gmt`	do not replace, and adjust time to be gmt
`GMT`	do not replace, and adjust time to be gmt

Figure 5: Time Format Codes

Some example *time-formats* and associated times are:

/T‘mm/dd/yyyy’/	02/21/2014
/T‘dd/mm/yyyy’/	21/02/2014
/T‘mm/yyyy’/	02/2014
/T‘www_dd_mmm_\’yy’/	fri 21 feb ’14
/T‘Www_ddmmmyy’/	Fri 21feb14
/T‘hh::mm:ss’/	13:27:58
/T‘hh::mm:ssXM’/	1:27:58PM
/T‘hh::mm:ss’/	00:27:58
/T‘hh::mm:ssXM’/	12:27:58AM
/T‘hh::mm:ss’/	12:27:58
/T‘hh::mm:ssXM’/	12:27:58PM
/T‘Www_Mmm_dd_hh:mm:ss_ZZZ_yyyy’/	Fri Feb 21 13:27:58 EST 2014
/T‘Www_Mmm_dd_hh:mm:ss_GMT_yyyy’/	Fri Feb 21 18:27:58 GMT 2014
/T‘Www-ddMmmyy-hh:mm:ss-ZZZ’/	Fri-21Feb14-13:27:58-EST

6 Logical Lines and Paragraphs

Lexemes are contained in ***physical lines***, and are tagged with the line and column numbers of their first character and of the first character following the lexeme (or of an imaginary end-of-file character). Physical lines that begin with a comment can contain only that comment, and are called ***comment lines***. The beginning column number of the first lexeme in a non-blank, non-comment physical line is called the ***indent*** of the physical line. This indent can be used to group lines into ***indented subparagraphs***, and to indicate that one physical line is a continuation of a current ***logical line***. Lines containing only whitespace are called ***blank lines***, and can be used to separate paragraphs.

There are two kinds of paragraphs, code and text.

A ***code paragraph*** consists of a sequence of logical lines, each beginning at the same indent, the ***paragraph indent***. Each logical line can be continued by a physical line with greater indent than the paragraph indent that is called a ***continuation line***. Thus in a code paragraph the physical lines:

```
This is a line
     which is continued
       // A comment.
   and continued.
And this is a second line.
```

represent two logical lines containing the lexeme sequences:

```
    This is a line which is continued and continued .
```

and

```
    And this is a second line .
```

A ***text paragraph*** consists of a sequence of lexemes taken from a sequence of physical lines each with an indent at least as great at the paragraph indent. Thus in a text paragraph the physical lines:

```
    This is a line. // With a comment
    And this is a second line.
```

represent the single lexeme sequence:

```
    This is a line . And this is a second line .
```

In a text paragraph, all lines but the first are called ***continuation lines***, and the entire paragraph is a single logical line.

A ***top level paragraph*** is a paragraph whose paragraph indent is `0`.

A '`:`' or '`::`' lexeme at the end of a physical line begins a sequence of ***indented subparagraphs***. If the indent of the next non-blank, non-comment physical line exceeds the current paragraph indent, the indent of this line becomes the new current paragraph indent, and this line begins the first physical line of a new paragraph. Otherwise the sequence of indented subparagraphs is empty.

The sequence of indented subparagraphs ends just before the next non-blank, non-comment physical line with less indent than the new current paragraph indent, or just before the end of file if that occurs first.

At any point there is a ***current paragraph type*** in addition to the current paragraph indent. This type specifies the current paragraph kind: code or text.

The paragraph type can be changed by a paragraph header that begins the first a non-blank, non-comment physical line at the beginning of a file or following a blank line. A paragraph header can be a prefix (p64) of the form

$$\texttt{\{}\ \textit{paragraph-type-name}\ \{\ \texttt{:}\ \textit{paragraph-attributes}\ \}^{?}\ \texttt{\}}$$

Such are called ***explicit paragraph headers***.

Or the paragraph header may be implied by the first non-whitespace characters of the physical line; for example:

`*@`*digit*	implies	**`{raw}`**	kind is code
`-----`	implies	**`{table}`**	kind is code
`=====`	implies	**`{table}`**	kind is code

Such are called ***implied paragraph headers***.

Brackets enclose subexpressions but do not change the current paragraph indent. Some brackets enclose a sequence of text paragraphs. For example, `` ` ` `` `''` enclose text paragraphs of '**`{quoted}`**' type, but the current paragraph indent of these is the same as that of the logical line containing the `` `` `` opening bracket.

If the first a non-blank, non-comment physical line at the beginning of a file or following a blank line does not begin with an explicit or implicit paragraph header, the line either continues the previous paragraph, or begins a new paragraph of the current paragraph type. For example, if the current paragraph type is '**`code`**', the previous paragraph is continued, whereas if it is '**`quoted`**', a new paragraph of '**`quoted`**' type is begun.

At the beginning of a file the current paragraph indent is **`0`** and the current paragraph type is **`{code}`**. The first indented paragraph in a sequence introduced by a physical line ending with '**`:`**' or '**`::`**' also has type **`{code}`**, unless the '**`:`**' or '**`::`**' is preceeded by special indentation marks. The first paragraph introduced by the `` `` `` opening bracket has the '**`{quoted}`**' type.

As an example, consider a file beginning with:

```
``This is a quoted text paragraph.  And
a second sentence therein.

And this is a second quoted paragraph
   with one sentence.''

This is a logical line.
And another
    logical line.
And a logical line ending with an indented code paragraph:

    // A comment line that can be ignored.
    The first logical line of the
        indented paragraph.

    The second logical line
        of the indented paragraph.
    The third logical line of the indented
        paragraph containing an indented
        code subparagraph:
        // A warning message will be issued
        // for these comment lines.

            Line 1 of subparagraph.

            Line 2 of subparagraph.

        that is not at the end of this third logical
             line.
             // A comment line that can be ignored.
    The fourth logical line of the indented paragraph.
    The fifth logical line of the indented
        paragraph ``And a quoted text paragraph inside the
    logial line.''
        but not at the end of the logical line.
// A comment line that can be ignored.
The logical line after the indented paragraph.
```

The first is a '**`{code}`**' paragraph that contains a `` ` ` `` `''` bracketed sequence of '**`{quoted}`**' paragraphs. The quoted paragraphs are text paragraphs and while they are being read the requirement for indenting continuation lines of a logical line is suspended. The current paragraph type, '**`{code}`**', is saved when ``` `` ``` is encountered and restored when `''` is encountered, and the requirement for indenting continuation lines is also restored by the `''`.

The third logical line of this paragraph ends with an indented paragraph, which is also a '**`{code}`**' paragraph because it was introduced by a physical line ending '`:`. This indented paragraph has 5 logical lines. The third contains within it (not at its end) another indented subparagraph. The fifth contains within it (and not at its end) a ``` `` ``` `''` bracketed '**`{quoted}`**' paragraph, whose lines merely need to start at or after the current paragraph indent.

A warning message is issued for any comment line whose indent is less that that of the next non-blank, non-comment physical line. This applies in the example to the comment lines just before the first logical line of the subparagraph.

There are some special rules for gluing together lexemes within a logical line. When reading these rules, recall that in a text paragraph all lines but the first are continuation lines, while in a code paragraph, lines with greater indent than the current paragraph indent are continuation lines.

1. Two consecutive ***quoted-strings*** in a logical line are concatenated after *character-representatives* have been replaced by the characters they represent.

   Thus **`"<" "LF" ">"`** is equivalent to a single *quoted-string* lexeme with the 4 characters '**`<`**', '**`L`**', '**`F`**', and '**`>`**', whereas **`"<LF>"`** is a *quoted-string* with only 1 character, a line feed.

2. If a physical line which is immediately followed by a continuation line ends with a *middle-lexeme* that contains 5 or more characters ending with a '**`-`**' (which counts as one of the 5 characters), and the continuation line begins with a *middle-lexeme*, the two *middle-lexemes* will be glued together with the line ending '**`-`**' deleted. The *middle-lexeme* on the continuation line may also begin with a '**`-`**', which will not be deleted.

   For example,

   ```
   This is a word-
        -that-is-con-
        tinued.
   ```

   translates to the lexeme squence

   ```
   This is a word-that-is-continued .
   ```

   and

   ```
   See http://forecast.weather.gov/MapClick.php?-
             lat=42&lon=-
             -71#.V036u7
   ```

   translates to the 2 lexeme squence

   ```
   See
   http://forecast.weather.gov/MapClick.php?lat=42&lon=-71#.V036u7
   ```

The following virtual lexemes are used in the rest of this document:

bip	begin indented paragraphs	***eip***	end indented paragraphs
bll	begin logical line	***ell***	end logical line

The sequence

indentation-mark bip . . . eip

denotes use of a physical line ending lexeme (or sequence of several), the ***indentation-mark***, followed by one or more indented paragraphs (the . . .), followed by an end to the indented paragraphs. Example *indentation-marks* are '`:` and '`::`'. The *indentation-mark* determines whether the indented paragraphs are code or text. If code, the . . . may be a sequence of logical lines.

7 Low Level Data

A low level ***datum*** in RECKON is either an atom, an object, or an arrow.

Expressions, blocks, and descriptions are higher level data layered on top of these low level data. An expression is either an atom or a particular kind of object. A block is a particular kind of object, and a description is a particular kind of block.

An ***atom*** is a symbol, a number, or a label. A ***label*** is a sequence of 0 or more atoms. A single symbol or number is frequently treated as if were a label whose only element is the symbol or number itself.

An ***object*** is just a place in memory, and is like a dot on a blank page. It can be the source or target of arrows, and it is different from every other object and from every atom. But it is nothing more.

However, as a place in memory, an object has a identifier. Objects are assigned ***raw object identifiers*** of the form '`@`N' where N is a natural number (non-negative integer). Raw object identifiers of this form are assigned only to objects that must be named in an output stream (e.g., printed output). The first object named in an output stream is assigned the identifier **`@1`**, the second object named in the output stream is assigned the identifier **`@2`**, and so forth. The same object may be assigned different identifiers in different output streams.[6]

There are two kinds of arrows: single and double.

A ***single arrow*** is an arrow from an object to either another object or to an atom. The arrow has a label, which is a sequence of symbols and natural numbers.

A ***double arrow*** is a double headed arrow between two objects. It has a separate label for each direction, with each label being a sequence of symbols and natural numbers. A double arrow is equivalent to a related pair of single arrows going in opposite directions between the same two objects. The difference between a double arrow and a pair of single arrows is that it is possible to delete only one direction of a pair of single arrows, but when deleting a double arrow, both directions are deleted.

An ***arrow label*** is an atom that encodes a *name* (p33) or a single *natural-index* (p33). If the former, the atom is either a single symbol, or is a label with 2 or more elements, the first of which is a symbol, and the remainder of which are either symbols or natural indices.

Two arrows leaving the same object may have the same label. Thus an object and an arrow label together name a set of arrows sourced at the object. We say that the object and label together

[6]A possible implementation is to give objects that have been assigned identifiers in an output stream a hidden system defined output stream specific attribute (p55) equal to the object's identifier integer. Another implementation uses a hash table per stream.

specify a set of values, each of which is an atom or an object.

Arrow flags may be attached to arrow labels. More precisely, a set of arrow flags is defined for each object and each arrow label, and these flags apply to all arrows sourced at the object that have the given label. The standard flags are the ***dot flag*** (`.`), and the ***maybe flag*** (`?`). Arrows with a dot flagged label are not to be output when their source is output. Targets of arrows with a maybe flagged label may be garbage collected (made to disappear automatically, see 15$^{p\,138}$) if they cannot be reached without traversing an arrow whose label has a maybe flag.

We will give examples in the next section along with a basic way of representing sets of objects in text. In the rest of this document arrows are called ***attributes***, arrow labels are called ***attribute labels***, arrow flags are called ***attribute flags***, and arrow targets are called ***attribute values***. Also '***attribute L of object O***' denotes the set of all values (arrow targets) of attributes of object O (arrows sourced at O) which have the attribute label (arrow label) L.

A double arrow is called a ***double attribute***. When viewed from an object at one end of the double arrow, the double arrow is an attribute of that object, the label of the arrow directed away from that object is the ***attribute label*** of the double attribute, the object at the other end is the ***attribute value*** of the double attribute, and the label of the arrow directed toward the object is the ***reverse attribute label*** of the double attribute.

For a given object, an attribute label may label both single and double arrows. A distinction is made between the set of values of the single arrows, and the set of values of the double arrows. The values of double arrows, of course, must all be objects, while the values of the single arrows may be either objects or atoms.

7.1 Object Representations

A set of objects can be written to a text file or read from a text file. When this is done, a textual representation of the object set exists in the file. The most basic representation of an object is the ***object representation***, which we now describe.

An ***raw object representation*** is a special case of an *object-representation*. This special case makes minimal use of the parser and is used when data is written to a file with minimal formatting of the data.

A *object-representation* is a logical line beginning with a *represented-object-identifier* lexeme, which consists of either '`@`' or '`!@`', followed by digits. Use of '`!@`' turns operator and text parsing off and makes the representation '***raw***'. The use of just '`@`' instead leaves operator and text parsing on and makes the representation '***cooked***'. Bracket parsing is on in either case.

object-representation
:= *bll* *represented-object-identifier* *object-assignment-operator*
object-element*
{ `::` *bip* { *bll attribute-representation ell* }* *eip* }$^?$ *ell*
:= *bll* *represented-object-identifier* `=` *object-element* *ell*

represented-object-identifier :::= *raw-object-identifier* | *cooked-object-identifier*
raw-object-identifier :::= `!` *object-identifier*
cooked-object-identifier :::= *object-identifier*
object-identifier :::= `@` *digit*$^+$

object-assignment-operator ::= `=` | `=>` | `=>>`

object-element ::= *object-identifier* | *atom-name* | *bracketed-list*
atom-name ::= *word* | *quoted-string* | *number-name* | *bracketed-label*
number-name ::= *integral-numeric* | *fractional-numeric* | *scientific-numeric*
| *radix-numeric* | *date* | *time*
bracketed-label ::= `{*` *atom-name** `*}`
bracketed-list ::= see p64

attribute-representation ::= *single-attribute-representation*
| *double-attribute-representation*

single-attribute-representation
::= *single-attribute-label* *attribute-label-flags*$^?$ `=` *single-attribute-values*
| *single-attribute-label*
| `no` *single-attribute-label*

single-attribute-label ::= *name*$^{p\,33}$ | *natural-index*$^{p\,33}$

attribute-label-flags ::= `[` *flag-lexeme** `]`

flag-lexeme ::= *word* | *mark* | `.` | `?` | `!` | *quoted-string*

single-attribute-value ::= *atom-name** | *object-identifier* | *bracketed-list*
single-attribute-values ::= *single-attribute-value*
| `{*` *single-attribute-value* { `,` *single-attribute-value* }* `*}`

double-attribute-representation ::=
double-attribute-label attribute-label-flags$^?$
`=` *double-attribute-values*
`=` *reverse-attribute-label attribute-label-flags*$^?$

double-attribute-label ::= *name* $^{p\,33}$ | *natural-index* $^{p\,33}$
reverse-attribute-label ::= *name* $^{p\,33}$ | *natural-index* $^{p\,33}$

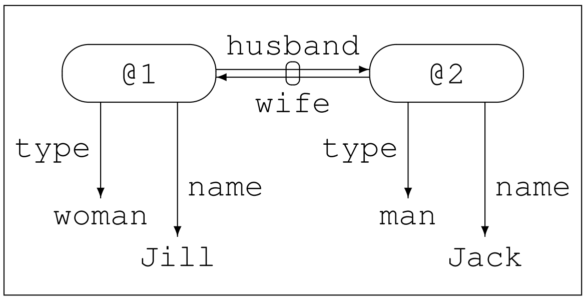
double-attribute-value ::= *object-identifier*
double-attribute-values ::= *double-attribute-value*
| **{*** *double-attribute-value* { **,** *double-attribute-value* }* ***}**

Raw *object-representations* only parse the punctuation given above plus brackets and separators. Assuming that by custom all brackets and separators contain a *mark* or *separator* lexeme, it is never necessary to quote a *word* in a raw *object-representation*. This can be exploited by quoting just *marks*, *separators*, and *numeric* symbols when outputting raw *object-representations*.

For example:

```
!@1 =::
    type = woman
    name = Jill
    husband = @2 = wife
!@2 =::
    type = man
    name = Jack
```

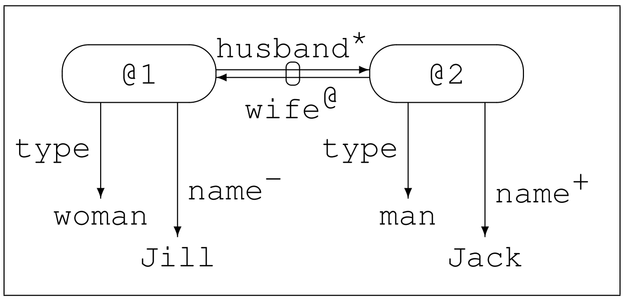
These are raw *object-representations* of an object pair that can be represented pictorially as:



There are two single attributes of object **`@1`** (arrows sourced at **`@1`**), one attribute labeled **`type`** whose value (target) is the atom **`woman`**, and one attribute labeled **`name`** whose value is the atom **`Jill`**. There are two similar single attributes from object **`@2`**. There is a double attribute (double arrow) between the two objects which has the label **`husband`** when going from **`@1`** to **`@2`** and the label **`wife`** when going in the reverse direction.

It is possible to place ***attribute flags*** on attribute labels by putting flag characters in a *flag-lexeme* inside **`[ ]`** brackets after a label. It may be necessary to use a *quoted-string flag-lexeme* in order to include separator characters or digits as flag characters. The following is the same as the above example except that flags have been added to some of the attributes:

```
!@1 =::
    type = woman
    name[-] = Jill
    husband[*] = @2 = wife[@]
!@2 =::
    type = man
    name[+] = Jack
```



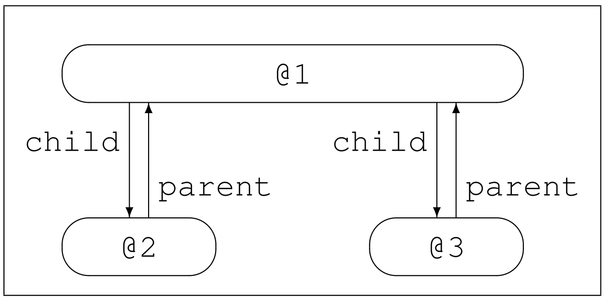
In the picture the attribute flags have been added as superscripts on the attribute labels, and in the text the flags have been added inside **[ ]** brackets that follow attribute labels.

Several attributes of the same object (arrows sourced at the object) may have the same attribute label. An example of this, in which object **@1** has two attributes labeled **child**, is:

```
!@1 =::
    child = @2 = parent
    child = @3 = parent
```

or

```
!@1 =::
    child = {* @2, @3 *}
          = parent
```



We say that the value of the **child** attribute of **@1** is the set to two elements, **@2** and **@3**.

The differences between **=**, **=>**, and **=>>** in a *object-representation* relate to what is done when an object or attribute label previously exists.

= indicates that the object being represented should not previously exist, or if it does exist, must not have been defined by any previous *object-representation* (it may have been defined as the value of an attribute). Second, any attribute label represented in the *object-representation*, if it previously exists because it was part of a *double-attribute-representation* of another *object-representation*, must be represented with exactly the same flags as it already has.

At the other extreme, **=>>** adds to existing objects. The object being represented can previously exist. Any attribute representation in the *object-representation* creates a new attribute value to be added to the set of values of the given attribute label (which is created if it did not previously exist). Any flags on an attribute label are added to the flags of the label if that label already exists.

=> is like **=>>** except that the *object-representation* cannot add new values to previously existing attributes of the object being represented. More precisely, the *attribute-label* of any *attribute-representation* in the *object-representation* must not have previously (before the *object-representation* is read) been an *attribute-label* of the represented object. Thus **=>** is used to introduce new attributes to an existing object.

However, if **=>** is used to add a double attribute to object $O1$ whose value is object $O2$, then while the *double-attribute-label* of the added attribute must not have previously existed for $O1$, the *reverse-attribute-label* may have previously existed for $O2$.

Double attributes must have only one representation. If they are given two representations, one for each end of the attribute (double headed arrow), two identical double attributes (two double headed arrows with the same end points and labels) will be created. Usually one end of a double attribute is thought of as the primary end, and its object representation is used to include the sole representation of the double attribute.

The *object-identifier* **@0** is special; it always names a particular object, the **.GLOBAL** object, whose attributes are called ***global variables.*** One of the global variables is named **.GLOBAL** and has as its value the **.GLOBAL** object itself, a situation which can be achieved by the *object-representation*:

```
!@0 =>::
     .GLOBAL = @0
```

Labels beginning with '**.**' are reserved for use by the RECKON system, and should not be defined by RECKON users. **.GLOBAL** is an example of such a label.

An *attribute-value* that is a sequence of two or more *atom-names* abbreviates a single *bracketed-label* atom that can be otherwise represented by placing the *atom-names* in **{* *}** brackets. Thus the *attribute-representations*

```
X = "{" "["          and          X = {* "{" "[" *}
```

are equivalent. However, an *attribute-value* consisting of just one *atom-name* consists of just the named atom.

Attribute-labels, which are *names*, are also sequences of *atom-names*, and also represent label atoms if there are two or more *atom-names* in the label. If there is just one *atom-name*, the label is the named atom. However, names cannot contain *bracketed-labels*, so

```
X Y = 1          cannot be rewritten as          {* X Y *} = 1
```

An attribute may have a single value, or a set of two or more values. In the latter case, the values are written separated by commas (**`,`**) and surrounded by **`{* *}`** brackets (the same as are used to surround the *atom-names* of several element label atoms). Thus

```
!@83 =>::
    X = Y 1                                        !@83 =>::
                     can be rewritten as               X = {* Y 1, Y 2 *}
!@83 =>::
    X = Y 2
```

A *single-attribute-representation* of the form '**`X = true`**' can be abbreviated as just '**`X`**', while one of the form '**`X = false`**' can be abbreviated as just '**`no X`**'. Thus

```
@49 =>::                                 @49 =>::
    X = true        can be rewritten as      X
    Y = false                                no Y
```

The RECKON parser ($8^{p\,70}$) parses *object-representations*. However, within *object-representations* that begin with **`@`** the parser only recognizes brackets and separators, and does not recognize operators, numeric units, formats, sentence and phrase terminators, etc. The parser does recognize lists as described in the next sections. In order to ensure that the parser does not mistakenly recognize *separators* and *marks* as delimiting lists, both *separators* and *marks* must be quoted as per **`"`** when they are used in *atom-names*.

7.2 Lists

Objects are implemented so that attributes whose labels are small strictly positive integers can be accessed with more efficiency than other attributes. If an object has attributes with consecutive labels from **`1`** to N, with no gaps, then these values are said to form a ***list*** that is represented by the object. An example is

```
!@93 =::
    1 = this
    2 = is
    3 = a
    4 = sentence
    .type = s
    .initiator = "`"
    .terminator = "'"
```

```
        capitalize = true
        end mark = "."
```

which represents the list

```
    this is a sentence
```

but also has some additional attributes, labeled **`.type`**, **`.initiator`**, **`.terminator`**, **`capitalize`**, and **`end mark`**.

An alternative raw *object-representation* to the one just given is:

```
    !@93 = this is a sentence::
        .type = s
        .initiator = "`"
        .terminator = "'"
        capitalize = true
        end mark = "."
```

The *object-elements* just after the *object-assignment-operator* in an *object-representation* specify values of the attributes labeled **`1`**, **`2`**, **`3`**, ... In this case the attributes labeled **`1`**, **`2`**, **`3`**, and **`4`** are given respectively the atoms denoted by the *atom-names* '**`this`**', '**`is`**', '**`a`**', and '**`sentence`**'. Additional attribute values can be given by following the *object-list* by a line ending '**`::`**' that begins an *attribute-representation-subparagraph*, which in this case gives values to the **`.type`**, **`.initiator`**, **`.terminator`**, **`capitalize`**, and **`end mark`** attributes.

Another alternative raw representation to the ones just given is:

```
    !@93 = `{s} this is a sentence'::
        capitalize = true
        end mark = "."
```

This works because bracket parsing, explained in the next section, is turned on within a raw *object-representation*, and parses the ` ` ' ` quotes and '**`s`**' prefix separator to produce the **`.initiator`**, **`.terminator`**, and **`.type`** attributes.

An alternative cooked representation to the ones just given is:

```
    @93 = `This is a sentence.'
```

The absence of an initial '**`!`**' and the presence of the ` ` ' ` quotes turns text parsing on, and this computes the '**`type`**', '**`capitalize`**', and '**`end mark`**' attributes.

Notice that the second form of object representation

bll represented-object-identifier **`=`** *object-element ell* is ambiguous with the first form which permits multi-element lists and extra attributes. The second is always preferred when it applies, and simply assigns the object identifier to the *object-element*. The first form, were it applied, would assign the identifier to a list whose single element is the *object-element*.

Thus

```
!@93 = (X Y Z)::
    weight = 55
```

means

```
!@93 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "("
    .terminator = ")"
    weight = 55
```

and does not mean

```
!@93 =::
    1 = @94
    weight = 55
!@94 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "("
    .terminator = ")"
```

Similarly

```
!@93 = "this is a"
       " long symbol with spaces in it"
```

means that `@93` identifies not an object, but a symbol atom, and

```
!@93 = @94
```

means that `@93` is an alternative designation for whatever `@94` designates.

Also

```
!@93 = X
```

assigns the identifier **@93** to the symbol **X**, but

```
!@93 = X::
    Y = 5
```

means

```
!@93 =::
    1 = X
    Y = 5
```

which has a single element list.

Note that single element lists can be represented using **{| |}** as described in the next section. For example,

```
!@93 = {| X |}
```

means

```
!@93 =::
    1 = X
```

It is also true that calling

```
!@93 = X
```

an object representation is a misnomer, as its really an atom representation.

7.3 Bracket Parsing

A *bracketed-list* represents an object in its own right, with the brackets typically becoming the **.initiator** and **.terminator** attributes of the object, and the elements between the brackets becoming the list elements of the object.

If operator, text, suffix, and units parsing are enabled, they operate on the elements of each separate list, after *bracketed-lists* have been parsed. In raw *object-representations*, operator, text, suffix, and units parsing are disabled, while in cooked *object-representations*, they are enabled. In any case, parsing *bracketed-lists* is done first before any other kind of parsing.

The syntax of a *bracketed-list* is:

bracketed-list ::=

```
      ( prefix-0-list )
    | [ prefix-0-list ]
    | [ [ prefix-0-list ] ]
    | { | prefix-0-list | }
    | { [ prefix-0-list ] }
    | { ` prefix-0-list ' }
    | { ! prefix-0-list ! }
    | ` prefix-0-list '
    | ` ` prefix-0-list ' '
    | ` ` ` prefix-0-list ' ' '
    | { ( prefix-0-list ) }
    | { { prefix-0-list } }
    | { : prefix-0-list : }
    | { ? prefix-0-list ? }
    | { mark-type prefix-0-list reflected-mark-type }
    | { name-type attributes? | prefix-0-list | }
    | { name-type attributes? | | }
    | { | prefix-0-list | name-type attributes? }
    | { | | name-type attributes? }
    | { name-type attributes? |   prefix-0-list | name-type attributes? }
    | { }
```

In the above: '**| |**' may be used to abbreviate '**| |**' (two separate '**|**'s)
{ } may be used to abbreviate **{ | | }** (the empty list: see below)

mark-type ::= *mark*
reflected-mark-type ::= reflection of the matching *mark-type* [see text]

name-type ::= *name* $^{p\,33}$

type ::= *mark-type* | *name-type*

```
list-opening  ::=  ( | [ | [ [ | ` | ` ` | ` ` ` | { | | { ` | { : | { ! | { ? | { ( | { [ | { {
                 | { mark-type
                 | { name-type attributes? |

list-closing  ::=  ) | ] | ] ] | ' | ' ' | ' ' ' | | } | ' } | : } | ! } | ? } | ) } | ] } | } }
                 | reflected-mark-type }
                 | | name-type attributes? }
```

prefix-n-list ::= { *prefix-n prefix-n+1-list* }* | *simple-list*

prefix-n ::= *prefix*

prefix ::= **{** *mark-type* **}** | **{** *name-type attributes*$^{?}$ **}**

All *prefix-n*'s in a *prefix-n-list* have the same type.

simple-list ::= *object-element**
object-element ::= see p56

attributes ::= **:** *attribute-representation* { **,** *attribute-representation* }*
| **: :** *bip* { *bll attribute-representation ell* }* *eip*

attribute-representation ::= see p56

Bracketed-lists containing *prefixes* are described at the end of this section. For now, we give examples of lists that contain no *prefixes*.

A *list-closing* must match the *list-opening* of the last unclosed list. The matching *list-openings* and *list-closings* that do not involve *types* are:

(	matches	**)**	**\`**	matches	**′**
[	matches	**]**	**\`\`**	matches	**′′**
[[	matches	**]]**	**\`\`\`**	matches	**′′′**
{|	matches	**|}**	**{(**	matches	**)}**
{{	matches	**}}**	**{[**	matches	**]}**
{\`	matches	**′}**	**{:**	matches	**:}**
{!	matches	**!}**	**{?**	matches	**?}**

These *list-openings* and *list-closings* that do not involve *types* may not be split across physical lines, even though some consist of two lexemes that may be separated by horizontal whitespace. These *list-openings* and *list-closings* become the **.initiator** and **.terminator** of the object represented by the *bracketed-list*, with the exception of the **{| |}** brackets, which enclose a pure list that has no attributes other than its list elements.

Thus, for example:

```
!@53 = {| X Y Z |}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
```

```
!@53 = ``X Y Z′′
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "`" "`"
    .terminator = "′" "′"
```

```
!@53 = ( X Y Z )
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "("
    .terminator = ")"
```

```
!@53 = {{X Y Z}}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "{" "{"
    .terminator = "}" "}"
```

```
!@53 = `X Y Z'
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "`"
    .terminator = "'"
```

```
!@53 = {`X Y Z'}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .initiator = "{" "`"
    .terminator = "'" "}"
```

A *bracketed-list* with a *mark-type* is similar to an untyped *bracketed-list*, but the *mark-type* becomes the **`.type`** attribute of the list, and there is no **`.initiator`** or **`.terminator`** attribute. Such a *bracketed-list* consists of a '**`{`** *mark-type*' list opening and a '*reflected-mark-type* **`}`**' list closing surrounding the list elements. The *mark-type* is a *mark* lexeme. The *reflected-mark-type* is this same lexeme with the order of characters reveresed and the characters **`<`** and **`>`** interchanged, if they occur. If the *mark-type* and *reflected-mark* type are identical, the *mark-type* becomes the **`.type`** attribute value for the list. But if they are not identical, the label atom

`{*` *mark-type reflected-mark-type* **`*}`**

becomes the **`.type`** attribute value.

Thus we have the examples:

```
!@53 = {+ X Y Z +}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = "+"
```

```
!@53 = {<+ X Y Z +>}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = "<+" "+>"
```

```
!@53 = {$$ X Y Z $$}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = "$$"
```

```
!@53 = {$⋆ X Y Z ⋆$}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = "$⋆" "⋆$"
```

Bracketed-lists with *named-types* but no *attributes* are similar but have the syntax:

{ *name-type*$^?$ **|** *prefix-0-list* **|** *name-type*$^?$ **}**

where the *name-type* must not be completely omitted, and if given twice, must be identical in both instances. Examples are:

```
!@53 = {T| X Y Z |}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = T
```

```
!@53 = {T| X Y Z |T}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = T
```

```
!@53 = {| X Y Z |T}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = T
```

```
!@53 = {"*" 1| X Y Z |}
```

is equivalent to

```
!@53 =::
    1 = X
    2 = Y
    3 = Z
    .type = "*" 1
```

Bracketed-lists with *named-types* can have *attributes*. These are either comma separated *attribute-representations* placed after a `:` following the *name-type*, or *attribute-representations* on one logical line each in an indented paragraph introduced by a physical line ending `::` placed just after the *name-type*. So, for example:

```
!@27 = X Y Z::
    .type = T
    J = 1
    K = B 1
    L = {* 1, B 1 *}
```

is equivalent to

```
!@27 = {| X Y Z | T::
             J = 1
             K = B 1
             L = {* 1, B 1 *}
        |}
```

is equivalent to

```
!@27 = {| X Y Z | T: J = 1, K = B 1, L = {* 1, B 1 *} }
```

is equivalent to

```
!@27 = { T: J = 1, K = B 1, L = {* 1, B 1 *} | X Y Z |}
```

A beginning '`{||`' is equivalent to '`{| |`', that is, in this case the two '`|`' are treated as separate 1-character lexemes, and the '`| |`' denotes an empty list. Similarly '`||}`' is equivalent to '`| |}`' and denotes an empty list.

The brackets '`{`' and '`}`' match, even if the *type-names* that follow '`{`' and preceed '`}`' fail to match. The latter case is a type match error.

Simple-lists may be parsed after they have been read. Operators, including separators such as comma '`,`' are recognized by such a parse. See 8 $^{p\,70}$ for details.

The *named-type* `""` is specially treated as denoting a missing `.type` attribute. Thus

```
!@27 = X Y Z::
    J = 1
    K = B 1
```

is equivalent to

```
!@27 = {| X Y Z | ""::
             J = 1
             K = B 1
        |}
```

The empty list can be denoted by any of '`{"" | | }`', '`{ | | }`', or the abbreviation '`{ }`'.

7.4 Prefixes

Prefixes are an unusual kind of separator that must appear at the beginning of the list as well as between elements of the list, and their appearance at the beginning of the list announces their use as a separator for the list and also establishes their precedence relative to other *prefix* separators. More explicitly, the *type* of the first *prefix-n* in a *prefix-n-list* specifies that all *prefixes* of that *type* shall be *prefix-n*'s, and *prefixes* of other *types* shall not be *prefix-n*'s, where *n* = *0*, *1*, *2*, *3*, ... is the priority of the *prefix* in the sense of operator hierarchy.

Algorithmically, during a left to right scan of a *prefix-0-list*, there is a stack of '***open list prefix types***'. Initially we are at the beginning of the *prefix-0-list* and the stack is empty. Then:

1. When a *prefix* of a type that is not in the stack is scanned, and the *prefix* follows a *prefix* or is at the beginning of the *prefix-0-list*, the type of the *prefix* is pushed into the stack, and a new sublist of that type beginning with the *prefix* is started. This sublist is a *prefix-n-list* where *n* is the depth of the stack just before the type is pushed.

2. When a *prefix* of a type that is in the stack is scanned, the stack is popped until the type is has been popped. Sublists of popped types are terminated. The type of the *prefix* (which was just popped) is pushed into the stack, and a new sublist of that type beginning with the *prefix* is started. This sublist is a *prefix-n-list* where *n* is the depth of the stack just before the type is pushed.

3. When a *prefix* of a type that is not in the stack is scanned, and the *prefix* follows a non-*prefix*, the *prefix* is announced as an error and ignored as if it did not exist. In particular, what immediately follows the *prefix* will still be treated as following a non-*prefix*.

4. Each sublist is associated with a *prefix* that starts the sublist. The type and attributes of this *prefix* become the type and attributes of the sublist.

Thus we have the following example parse of a *prefix-0-list*:

Text	Separator	Syntactic Category

`{p} {s} This is a sentence.`
`    {s} And another.`
`{p} {s} And a new {foo} paragraph.`
	`{p}`	*prefix-0-list*

`{s} This is a sentence.`
`{s} And another.`
	`{s}`	*prefix-1-list*

`This is a sentence.`	(none)	*prefix-2-list* ::= *simple-list*
`And another.`	(none)	*prefix-2-list* ::= *simple-list*
`{s} And a new {foo} paragraph.`	`{s}`	*prefix-1-list*
`And a new {foo} paragraph.`	(none)	*prefix-2-list* **`{foo}`** is in error and is ignored (deleted).

The *prefix-0-list* in this example is equivalent to:

```
{|  {p|  {s| This is a sentence "." |}
         {s| And another "." |} |}
    {p|  {s| And a new paragraph "." |} |} |}
             // `{foo}' deleted
```

8 Parsing

Input is read and parsed to form objects from *bracketed-lists*. These objects are then further parsed to find operators, separators, numeric units (e.g., **`lb`** and **`$`**), and some other constructions such as possessives (**`George's weight`** become **`George 's weigth`**). The input to this parsing is a list of ***tokens***, which initially are all lexemes. But as the parsing proceeds, bracketed lists are replaced by tokens that are ***expression objects*** (e.g., objects which may have **`.initiators`** and **`.terminators`**), and parsing proceeds on a list of tokens some of which are lexemes and some of which are objects.

Parsing at this point is a top down process. The parser is organized into ***passes*** which are applied to the token list in order. The first pass always handles bracketed expressions. Then the brackets of a subexpression establish a parsing ***context*** which determines which subsequent passes will be run on the token list of the backeted subexpression. For an ***operator expression***, the passes handle operators, numeric units, and possessives. For a ***text expression***, a pass handles phrases and sentences. If a pass succeeds it may discover subexpressions, which are parsed independently as they are discovered.

In general a pass modifies the list of tokens it is given, both replacing sublists by expression objects and possibly reordering the list.

The standard passes in order from first to last are:

Parser Pass	**Parsing Context**	**Usage**
`suffix`	`operator`	split suffixes from words
`operators`	`operator`	parse operators
`units`	`operator`	parse numeric units
`singularize`	`operator`	replace plurals by singulars
`text`	`text`	parse sentences and phrases

Which brackets select which parsing contexts is described in Section 8.2 $^{p\,77}$.

Tables such as this are constructed from parsing instructions which appear in a list called the parsing stack. Parsing instructions can be pushed into the stack and later popped from the stack in order to undo their affects, and they can be disabled and re-enabled while still in the stack. In this document we only describe a few of the many types of parsing instructions.

8.1 Parsing Examples

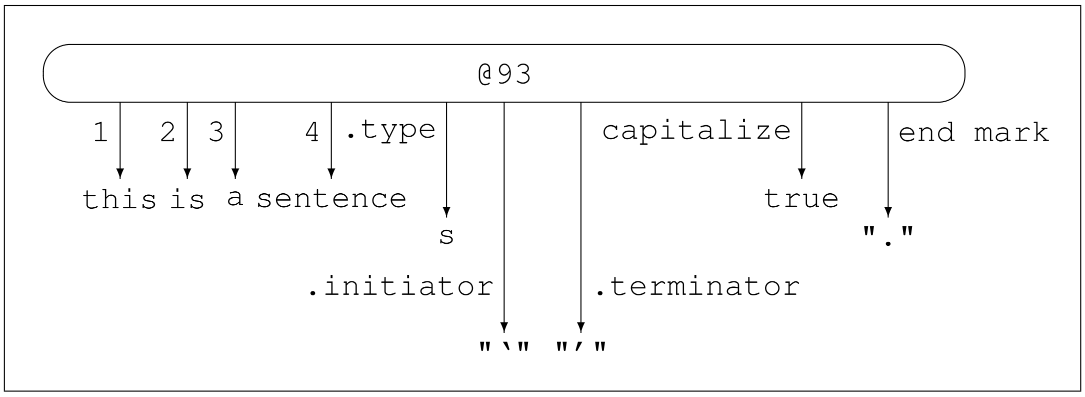
Prefixes, *types*, and *attributes* are explained below. First we give some examples involving just parsed untyped *bracketed-lists* without *attributes*. In some of these the fact is used that '**`{! !}`**' brackets translate to a list of **`.type "!"`**.

As a first example,

```
    @93 = `This is a sentence.'
```

is the cooked representation of the object

```
@93 =::
    1 = this
    2 = is
    3 = a
    4 = sentence
    .type = s
    .initiator = "`"
    .terminator = "'"
    capitalize = true
    end mark = "."
```

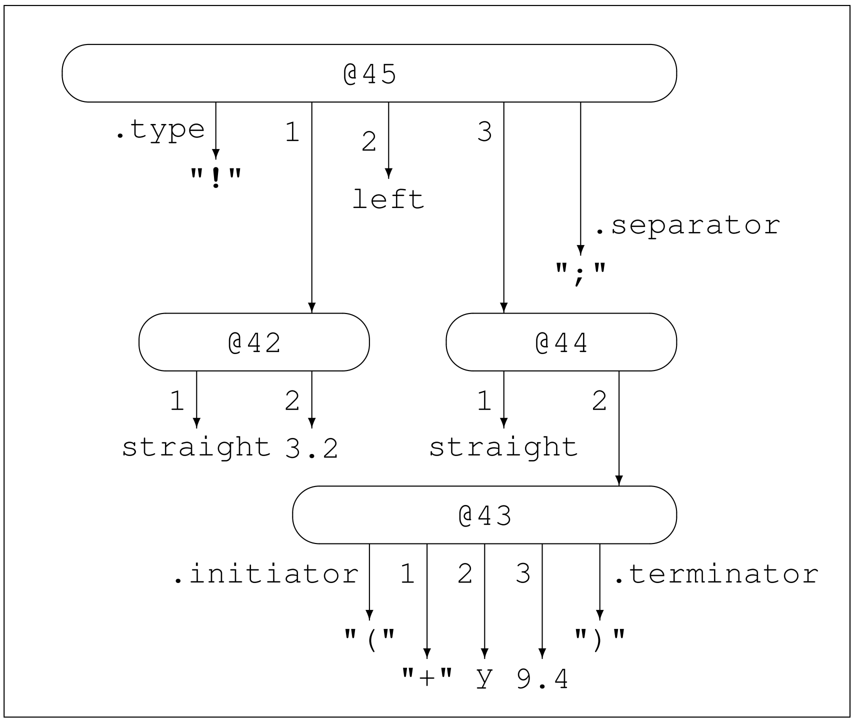


The second example contains simple code:

```
@45 = {! straight 3.2; left; straight (y + 9.4) !}
```

is the cooked representation of the objects

```
@42 =::
    1 = straight
    2 = 3.2
@43 =::
    1 = "+"
    2 = y
    3 = 9.4
    .initiator = "("
    .terminator = ")"
@44 =::
    1 = straight
    2 = @43
@45 =::
    1 = @42
    2 = left
    3 = @44
    .type = "!"
    .separator = ";"
```



A third example uses multiple indented lines to represent code, so that

```
@138 = {! function (x,y):

            if (x > y):
                return y
            else:
                return x
      !}
```

is the cooked representation of the object in Figures 6 and 7. Note that here the : indentation mark is used to introduce subparagraphs within the code, and not *attribute-representations*.

```
@130 =::
    1 = x
    2 = y
    .separator = ","
    .initiator = "("
    .terminator = ")"
@131 =::
    1 = ">"
    2 = x
    3 = y
    .initiator = "("
    .terminator = ")"
@132 =::
    1 = return
    2 = y
    .type = ":"
@133 =::
    1 = if
    2 = @131
    3 = @132
@134 =::
    1 = return
    2 = x
    .type = ":"
@135 =::
    1 = else
    2 = @134
@136 =::
    1 = @133
    2 = @135
    .type = ":"
    .separator = "<*>"
@137 =::
    1 = function
    2 = @130
    3 = @136
    .type = "!"
```

Figure 6: Example Code Object Representation I

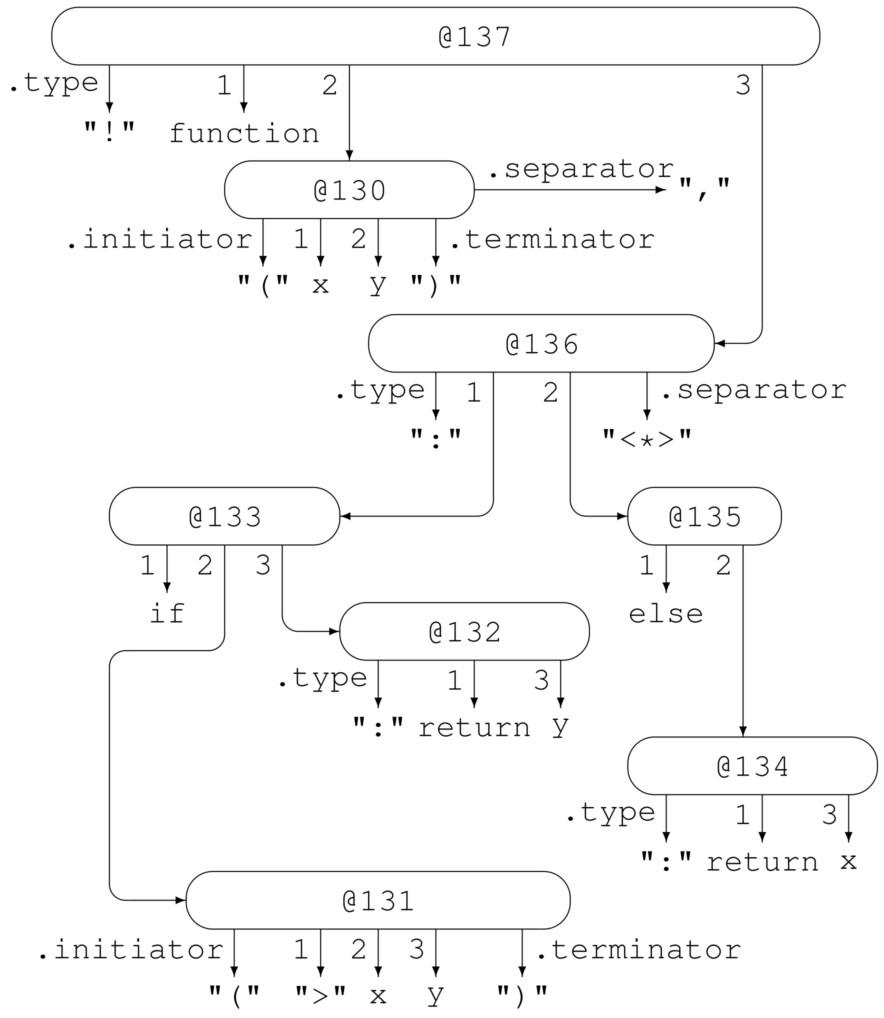


Figure 7: Example Code Object Representation II

Cooked representations may replace raw object identifiers in the description of other objects, as in

```
@291 =::
      text = 'This is a sentence.'
      outline = {! straight 3.2, left, straight (y + 9.4) !}
      min = {! function (x,y):

                  if (x > y):
                      return y
                  else:
                      return x
    !}
```

is the cooked representation of the object

```
@291 =::
    text = @93
    outline = @45
    min = @137
```

given the above examples.

One question left unanswered by the discussion so far is whether

```
@291 =::
    text A = 'This is a sentence.'
    text B = 'This is a sentence.'
```

represents

```
@291 =::
    text A = @93
    text B = @93
```

or instead

```
@291 =::
    text A = @93
    text B = @999
```

where object **@999** happens to have the same structure as object **@93**. The default is to make both **text A** and **text B** be the same object, **@93**, and to make that object 'immutable', meaning that it cannot be changed. The rule is that unless otherwise indicated, only immutable objects have the property that their cooked representations can replace their raw object identifiers in the cooked representations of other objects.

8.2 Parsing Contexts

During parsing both the parsing passes and definitions of parsing units such as operators are turned on or off by manipulating a ***parsing context***, which is just a set of ***parsing selectors***. Thus the passes and definitions appropriate to text processing are turned on by the '`text`' selector, and those appropriate to code parsing are turned on by the '`code`' selector.

The parsing context can only be changed by explicit brackets. So, for example, the `` `...' `` brackets always set the parsing context to just the '`text`' selector during the parse of the `` `...' `` bracketed list. As another example, '`(...)`' brackets do not change the parsing context but instead inherit it from their containing object.

The following is a list of the standard selector-changing brackets:

Brackets	**Context**	**New Context**	**Usage**
`{*...*}`	all	none	comment
`` `...' ``	all	`text`	text
`{!...!}`	all	`expression`, `code`	unevaluated code
`{{...}}`	all	`expression`, `math`	unevaluated math subexpression
`[...]`	`text`, `math`	`expression`	evaluated subexpression in text or math expression

According to this table, in any parsing context the `{*...*}` brackets turn off all selectors, the `` `...' `` brackets change the context to have just the '`text`' selector, the `{!...!}` brackets change the context to have both the '`expression`' and '`code` selectors, and so forth.

Brackets not listed do not change the parsing context. The `[...]` brackets only change the parsing context if their containing parsing context has the '`text`' or '`math`' selectors. None of the brackets change the context if there are no selectors on in the current parsing context, as is the case inside a `{*...*}` comment (but unquoted brackets are still recognized and must match inside the comment).

8.3 Parser Suffix Pass

The ***parser suffix pass*** splits suffixes from words so that they may be recognized as operators. Thus '`George's`' is split to become '`George 's`'.

The following are standard suffixes:

`'s   'S`

Suffixes split from words are usually recognized as operators by the parser operator pass.

8.4 Parser Operator Pass

The ***parser operator pass*** identifies operators in a subexpression and reformats subexpressions in which operators are found. Each operator can be enabled or disabled according to the current parser context.

The following are the standard operators enabled by various selectors.

Enabled by '`code`' selector:

Precedence	Reformatter	Kind	Name	Meaning
-2000	`separator`	`nofix`	`<*>`	separator
-1000	`separator`	`nofix`	`;`	separator
+0000	`define`	`nofix`	`<--`	define
		`afix`	`{: :}`	block
+1000	`right associative`	`nofix`	`=`	assignment
			`+=`	increment
			`-=`	decrement
			`*=`	multiply by
			`/=`	divide by

Enabled by '`operator`' selector:

Precedence	Reformatter	Kind	Name	Meaning
+2000	`separator`	`nofix`	`,`	separator

+3000	**`unary`**	**`nofix`**	**`NOT`**	logical not
+3000	**`infix`**	**`nofix`**	**`BUT NOT`**	logical but not
+4000	**`infix`**	**`nofix`**	**`AND`**	logical and
			`OR`	logical or
+5000	**`infix to(AND)`**	**`nofix`**	**`==`**	equal
			`/=`	not equal
			`!=`	ditto
			`<`	less than
			`<=`	less than or equal
			`=<`	ditto
			`>`	greater than
			`>=`	greater than or equal
			`=>`	ditto
+6000	**`summation(+,-)`**	**`infix`**	**`+`**	addition
			`-`	subtraction
+6100	**`binary`**	**`infix`**	**`/`**	division
+6200	**`infix`**	**`infix`**	**`*`**	multiplication
+6300	**`binary`**	**`infix`**	**`^`**	exponentiation
+10000	**`right associative`**	**`infix`**	**`OF`**	attribute
			`of`	selector
+11000	**`left associative`**	**`infix`**	**`'S`**	attribute
			`'s`	selector
none	**`unary`**	**`prefix`**	**`-`**	minus
		`prefix`	**`+`**	plus
		`prefix`	**`@`**	object identifier
none	**`implied ( subscript )`**	**`postfix`**	**`[...]`**	subscript

Enabled by ‘**`text`**’ selector:

Precedence	Reformatter	Kind	Name	Meaning
-10000	`terminator (s,cap,noend)`	`nofix`	`.`	(sub)sentence terminator
		`nofix`	`!`	
		`nofix`	`?`	
		`nofix`	`;`	
		`nofix`	`:`	

Each entry in these tables describes an '***operator parsing instruction***'. Parsing instructions appear in an ordered list, called the parsing instruction stack.

List ***parsing*** is performed by a left to right scan. At each point in the scan an attempt is made to find an operator instruction whose name begins at the scan point. Instructions are rejected if the are not enabled by a selector in the current parsing context.

Other instructions are rejected on the basis of their kind and the scan point position: for example, an infix instruction is rejected if the scan point is at the beginning of the list, at the end of the list, right after a just-recognized prefix operator, etc.

If there are non-rejected instructions, those with the longest names beginning at the scan point are selected, and among these, the one earliest in the parsing instruction stack is used, unless that instruction has the '**`undefined`**' flag, in which case the process is repeated with only shorter names allowed.

With this in mind the possible operator parsing instruction ***kinds*** are recognized according to the following rules:

`prefix`
: To be recognized a prefix operator must begin the list, or immediately follow a previously recognized prefix, infix, nofix, or afix operator of less precedence, or immediately follow a previously recognized prefix operator of equal precedence. Also the prefix operator may not end the list.

`postfix`
: To be recognized a postfix operator must immediately follow a non-operator, or a postfix, nofix, or afix operator of higher precedence, or another postfix operator of equal precedence.

`infix`
: To be recognized an infix operator must immediately follow a non-operator, or a postfix, nofix, or afix operator of higher precedence. Also the infix operator may not end the list.

`nofix`

A nofix operator is always recognizable.

`afix`
: To be recognized an afix operator must appear after another operator of equal precedence without any intervening operator of less precedence.

You will note that in the above table some operators defined to be **`nofix`** can in fact only be used in **`infix`** positions. This is done to minimize coding errors by recognizing misplaced operators. However, this cannot be done with operators which are different according to their kind, such as **`infix`** '`-`' and **`prefix`** '`-`'.

During the left to right scan, errors are detected and processed as follows:

after **`prefix`**
: If an operator is found after a prefix operator that has lower precedence than the prefix operator, or that has equal precedence and is not itself a prefix operator, an error operand is inserted between the two operators and an error is announced.

after **`infix`**
: If an operator is found after an infix operator that has precedence lower than or equal to that of the infix operator, an error operand is inserted after the infix operator and an error is announced.

after **`postfix`**
: If an non-operator is found after a postfix operator, or if an operator is found that has precedence higher than the postfix operator, or if a non-postfix operator is found that has precedence equal to the postfix operator, an error nofix operator of precedence one less than the postfix operator is inserted after the postfix operator, and an error is announced.

The output of the parse is obeys the following context free grammar:

expression-n	::=	*prefix-operand-n operand-continuation-n*$^?$
	|	*operator-continuation-n*
	|	*symbol*$^+$ not part of an operator
operand-continuation-n	::=	*infix-operator-n expression-n+1 operand-continuation-n*$^?$
	|	*infix-operator-n postfix-operand-n*
	|	*nofix-operator-n operator-continuation-n*$^?$
	|	*afix-operator-n operator-continuation-n*$^?$
operator-continuation-n	::=	*expression-n+1 operand-continuation-n*$^?$
	|	*postfix-operand-n*
	|	*nofix-operator-n operator-continuation-n*$^?$
	|	*afix-operator-n operator-continuation-n*$^?$
prefix-operand-n	::=	*prefix-operator-n expression-n+1*
	|	*prefix-operator-n prefix-operand-n*
postfix-operand-n	::=	*expression-n+1 postfix-operator-n*
	|	*postfix-operand-n postfix-operator-n*
prefix-operator-n	::=	prefix operator of precedence n
infix-operator-n	::=	infix operator of precedence n
postfix-operator-n	::=	postfix operator of precedence n
nofix-operator-n	::=	nofix operator of precedence n
afix-operator-n	::=	afix operator of precedence n

Here *expression-n* denotes an expression whose operators are of precedence n. Not reflected in the above is the rule that an afix operator cannot be the first operator in an expression.

An operator parsing instruction may associate an ***operator reformatter*** with an operator. After parsing a list, if the first operator in a resulting expression has an associated operator reformater, the expression is reformatted according to that reformater. For example, if the '`*`' operator has the '`infix`' reformatter, then expression '`5 * x * y`' will be reformatted as '`* 5 x y`'. If in addition the operator '`/`' has the same precedence as '`*`', the expression '`5 * x / y`' would be declared to be in error because the '`infix`' reformatter insists that all operators in an expression it reformats be the same.

The possible reformatters are:

`infix`

It is an error unless the expression has alternating operands and operators and begins and ends with an operand.

It is an error if all the operators in the expression are not the same operator.

The first of these operators is moved to the beginning of the expression and the remaining operators are removed from the expression.

`unary`
It is an error unless there is only one operator in the expression and it is at the beginning of the expression. The expression is not itself changed.

`binary`
It is an error unless there is only one operator in the expression and it is surrounded by operands. The operator is moved to the beginning of the expression.

`right associative`
It is an error unless the expression has alternating operands and operators and begins and ends with an operand.

Brackets are introduced so the operators are all binary and the rightmost is done first, and then all the binary operators are moved to the beginning of their subexpressions: e.g.,

```
x = y -= z += 5
```

becomes

```
= x {| -= y {| += z 5 |} |}
```

if '`=`', '`-=`', and '`+=`' have the same precedence and '`=`' has this reformatter.

`left associative`
Like **`right associative`** except that brackets are introduced so the leftmost is done first: e.g.,

```
x + y - z + 5
```

becomes

```
+ {|- {| + x y |} z |} 5
```

if '`+`' and '`-`' have the same precedence and '`+`' has this reformatter (but '`+`' and '`-`' have a different reformatter in the above tables).

`separator`
Only allowed for operators of '**`nofix`**' kind.

It is an error if all the operators in the expression are not the same operator.

Empty list operands (i.e., **`{||}`** lists) are inserted so that all the operators become infix. Then all the operators are removed and the operator name is made into the value of the '**`.separator`**' property of the list.

Thus if '**`,`**' is a nofix operator with this reformatter, the expression

```
, w x,, y, z,
```

is reformatted to

```
{"", .separator = ","| {||} {| w x |} {||} y z {||} |}
```

It is possible to make new reformatters using ***reformatter constructors*** that take arguments which are constant expressions. There are examples in the above tables; e.g., **`summation ('+', '-')`**.

The following are the reforrmatter constructors:

`infix to (` *and-op* **`)`**

This reformatter is used for infix comparison operators such as '**`<`**', '**`==`**', '**`<=`**'.

It is an error unless the expression has alternating operands and operators and begins and ends with an operand.

Each operator is executed separately from left to right and the results combined by the '*and-op*' operator. Thus

x1 op1 x2 op2 x3

becomes

```
and-op {| op1 x1 x2 |} {| op2 x2 x3 |}
```

except that temporaries are introduced to prevent non-end operands from being evaluated more than once, so the actual final reformatting of the example just given is

```
and-op {| op1 x1 {| = ##1 x2 |} |} {| op2 ##1 x3 |}
```

where '**`##1`**' denotes a temporary variable (see below $^{p\,85}$).

`summation (` *plus-op*, *minus-op* **`)`**

It is an error unless the expression has alternating operands and operators and begins and ends with an operand.

It is an error unless all the operators in the expression are either *plus-op* or *minus-op*.

Each '*minus-op operand*' subsequence of the expression is replaced by

plus-op { | *minus-op operand* | }

and then the expression is processed as by the **`infix`** reformatter, copying the first *plus-op* operator to the beginning of the expression while removing all the other *plus-op* operators.

Thus if '**`+`**' has this reformatter with arguments **`(+,-)`**, and '**`+`**' and '**`-`**' are infix operators with the same precedence, then the expression

```
w + x - y + z
```

is first changed to

```
w + x + {| - y |} + z
```

and finally reformatted as

```
+ w x {| - y |} z
```

`implied (` *implied-op* **`)`**

This reformatter is used for postfix operators which do not have ordinary names but are instead explicitly bracketed subexpressions named by their subexpression **`.type`** or **`.initiator`**.

No errors are detected. The *implied-op* is merely added to the beginning of the expression.

Sometimes ***temporaries*** are introduced by reformatting. The syntax involving temporaries is:

temporary-definition ::= *temporary subexpression*

temporary ::= **`##`** *integral-numeric* $^{p\,34}$

During reformatting, a subexpression may be replaced by a *temporary-definition* containing the subexpression. This defines the *temporary* as a name for the value of the subexpression. Then the *temporary* can be used in the reformatted expression to refer to this value. As an example, given the above tables

```
x == y + 3 == z
```

becomes

```
AND {| == x {| ##563 {| + y 3 |} |} |} {| == {| ##563 |} z |}
```

Here the temporary has been used to avoid computing **`y + 3`** more than once.

The *decimal-naturals* in *temporaries* are assigned so no two *temporary-definitions* ever have the same *temporary*.

8.5 Parser Units Pass

The function of the ***parser units pass*** is to identify numeric units, such as **`ft`** and **`lb`**, and turn them into multipliers and divisors. For example, '**`35 ft lb`**' is reformatted as '**`* 35 ft lb`**' and '`$100 per yr`' is reformatted as '`/ {| * $ 100 |} yr`'.

The following are standard units:

Unit	Singular	Plural	Fixity	Type	Equivalent
Gram*	`g`		postfix	continuous	
Pound	`lb`	`lbs`	postfix	continuous	`453.59237 g`
Ounce	`oz`		postfix	continuous	`28.349523 g`
Kilogram	`kg`		postfix	continuous	`1000 g`
Centimeter*	`cm`		postfix	continuous	
Meter	`m`		postfix	continuous	`100 cm`
Foot	`ft`		postfix	continuous	`30.48 cm`
Yard	`yd`		postfix	continuous	`91.44 cm`
Mile	`mi`		postfix	continuous	`160934.4 cm`
Second*	`sec`		postfix	continuous	
Minute	`min`		postfix	continuous	`60 sec`
Hour	`hr`		postfix	continuous	`3600 sec`
Degrees C*	`C`		postfix	continuous	
Degrees F	`F`		postfix	continuous	`1.8 C + 32`
Degrees K	`K`		postfix	continuous	`1.0 C + 273.15`
Person	`person`	`persons`, `people`	postfix	discrete	

The parser units pass attempts to parse a subexpression according to the following syntax:

units-subexpression ::= *unit-phrase*$^+$
| *unit-prefix-subexpression*
unitless-name
unit-suffix-subexpression$^?$
| *bifrucated-number*

unit-phrase ::= *unitless-name unit-suffix-subexpression*

unitless-name ::= *bifrucated-number*
| other *name*$^{p\,33}$ that does not contain any *unit* or *unit-subexpression*

bifrucated-number ::= *integer unsigned-fraction*

unsigned-fraction ::= *number* in the range $[0, 1)$

unit-prefix-subexpression ::= *prefix-unit*$^+$

unit-postfix-subexpression ::= *postfix-unit*$^+$
| *postfix-unit** **`per`** *postfix-unit*$^+$
| *unit-subexpression*

unit-subexpression ::= **`(`** *unit-expression* **`)`**

unit-expression ::= *unit-primary* { **`*`**$^?$ *unit-primary* }*
| *unit-primary* **`/`** *unit-primary*

unit-primary ::= *unit* | **`(`** *unit-expression* **`)`**

unit ::= *prefix-unit* | *postfix-unit*

If the pass finds any *unit* or *unit-expression* but the subexpression does not conform to the above syntax, an error message is issued. If there is no *unit* or *unit-expression* and the subexpression is not a *bifructed-number*, the pass leaves the expression alone. Otherwise the subexpression is reformatted according to the following rules:

(1) A '`+`' is placed between consecutive *unit-phrases*.

(2) A '`*`' is placed between consecutive *unit-primaries*, after any *unit-prefix-subexpression*, and before any *unit-postfix-subexpression*.

(3) Any '**`per`**' is replaced by '`/`', any *unit-postfix-subexpression* containing it is surrounded by `{| |}` brackets, and if there is no *postfix-unit* before the '**`per`**', a '`1.0`' is inserted before the '`/`'. Thus

```
40 lb per sec  ⟹ 40 * {| lb / sec |}
 40 per sec  ⟹ 40 * {| 1.0 / sec |}
```

(4) Then subexpressions containing infix `+`'s,`*`'s, or `/`'s are reformatted as per the operator pass. This moves infix operators to the beginning of the subexpression containing them.

(6) Next nested subexpressions beginning with `*` are collapsed: e.g.,

```
* X {| * y z |} W ⟹ * X y z W
```

(7) Lastly the two *numbers* in each *bifrucated-number* are combined by addition. E.g.,

```
40 1/5 ⟹ 40.2
```

The following are examples of parser units pass subexpression reformatting:

```
40 lb 3 oz ⟹ + {| * 40 lb |} {| * 3 oz |}
   100 m per sec ⟹ * 100 {| / m sec |}
```

8.6 Parser Dequoting Pass

8.7 Parser Control

After input text is scanned into symbols and these are parsed into typed lists, each typed list may optionally be parsed further using instructions in a parsing table associated with the type of the list. A ***parsing table*** is a list of ***operator parsing instructions*** which have the form:

operator-parsing-instruction	::=	**`{ operator parsing instruction:`**
		`name` `=` *operator-name* `,`
		`kind` `=` *operator-kind*$^+$
		$\{$ `,` **`undefine`** $\}^?$
		$\{$ `,` **`type`** `=` *operator-type* $\}^?$
		$\{$ `,` **`precedence`** `=` *operator-precedence* $\}^?$
		$\{$ `,` **`reformatter`** `=` *operator-reformatter* $\}^?$
		`|}`
operator-name	::=	*word*$^+$
operator-kind	::=	**`prefix`** | **`infix`** | **`postfix`** | **`nofix`** | **`afix`**
operator-type	::=	*word*$^+$
operator-precedence	::=	32 bit signed integer
operator-reformatter	::=	**`infix`** | **`unary`**
	|	**`binary`** | **`infix to`**
	|	**`right-associative`** | **`left-associative`**
	|	**`separator`** | **`summation`**

8.7.1 Number Pair Recognition

A ***number pair*** is a pair of *decimal-numbers*, the first of which is in integer and the second of which is a fraction containing a slash.

number-pair ::= *decimal-integer decimal-numerator* **/** *decimal-denominator*

decimal-integer ::= see p**??**

decimal-numerator ::= see p**??**

decimal-denominator ::= see p**??**

Some examples of number pairs are:

```
1 1/2      55 3/4      174 15/16      1,234,567 2,875/3,408
```

When a *number-pair* is encountered during number parsing, it is surrounded by **`(# #)`** implicit brackets and a **`+`** is inserted between its two parts. Thus if **`41 3/4`** is input to number parsing, **`(# 41 + 3/4 #)`** will be output.

A number pair is illegal if either of its two parts is illegal.

Number pairs are recognized after scientific numbers have been recognized.

8.7.2 Number Unit Grouping

A *number-unit-group* is a sequence of one or more *number-unit-pairs* each of which consists of two lexemes: a *real-number* and a *unit-specifier*. The syntax equations are:

number-unit-group ::= *number-unit-pair*$^+$

number-unit-pair ::= *prefix-unit-specifier real-number*
| *real-number postfix-unit-specifier*

prefix-unit-specifier ::= *name*

postfix-unit-specifier ::= *name*

unit-specifier ::= *prefix-unit-specifier* | *postfix-unit-specifier*

name ::= see p33

In order to be recognized as a *unit-specifier* a *symbol* sequence must be defined as a *unit-specifier-name* by the following:

unit-specifier-definition
::= **`define unit specifier`** *unit-specifier-fixity*
unit-specifier parsing-selectors

unit-specifier-fixity ::= **`prefix`** | **`postfix`**

parsing-selectors ::= see p**??**

The ***number unit grouping*** phase of parsing recognizes *number-unit-groups* containing more than one *number-unit-pair*, places **`(# #)`** implied brackets around them, and places implied '**`+`**' operators between their *number-unit-pairs*.

For example, if '**hr**' and '**min**' are *postfix-unit-specifiers*, '**7 hr 30 min**' is a *number-unit-group* containing 2 *number-unit-pairs* that number unit grouping transforms into

```
(# 7 hr + 30 min #)
```

As another example, if '**ft**' and '**in**' are *postfix-unit-specifiers*, '**12ft 11 3/4in**' becomes

```
(# 12 ft + (# 11 + 3/4 #) in #)
```

Number unit groups are recognized after number pairs have been recognized.

8.7.3 Unit Multiplication Insertion

The ***unit multiplication insertion*** phase of parsing inserts a multiplication operator lexeme, '*****', before a *postfix-unit-specifier*, and after a *prefix-unit-specifier*, unless the point at which the '*****' is to be inserted is already occupied by a '*****' or '**/**' lexeme. This is done after number unit grouping (8.7.2 $^{p\,89}$).

For example, if '**sec**', '**ft**', and '**lb**' are *postfix-unit-specifiers*, then

```
    some function ( 3 ft 2 1/4 in / sec,
                    9 ft ^ 2 lb / sec,
                    $3.50 )
```

becomes

```
    some function ( (# 3 * ft + (# 2 + 1/4 #) * in #) / sec,
                    9 * ft ^ 2 * lb / sec,
                    $ * 3.50 )
```

Unit multiplication insertion is done after number unit groups have been recognized.

8.8 Text Parsing

Text parsing is similar to operator parsing except that brackets and most operators are 'named'. For example, '**<p>**' is a standard paragraph operator, and '**<indented p>**' is a standard paragraph operator that is a child of '**<p>**' and inherits attributes from '**<p>**'. As another example, '**<b|** ... **|b>**' are standard 'bold' brackets.

The named text brackets and operators have the syntax:

named-operator ::= **<** *text-type arguments-option labels-option* **>**

named-left-bracket ::= **<** *text-type arguments-option labels-option* **|**

named-right-bracket ::= **|** *text-type arguments-option labels-option* **>**

text-type ::= *name*

arguments-option ::= *empty* | **(** *argument-list* **)**

argument-list ::= *empty* | *argument* { **,** *argument* }*

argument ::= *name* | *number* | *quoted-string*

labels-option ::= *empty* | **:** *label* { **,** *label* }*

label ::= *name*

name ::= see p33

number ::= see p**??**

quoted-string ::= see p32

The names of text brackets and operators are referred to as '***text types***'. For example, '**`<p>`**' is a text operator of text type '**`p`**', and '**`<b|`** ... **`|b>`**' are text brackets of text type '**`b`**'. Text types have attributes such as '**`style`**', '**`weight`**', '**`indent`**', and '**`adjust`**', that are used to format the text for printing and display. For example '**`b`**' has a the '**`weight`**' attribute value '**`heavy`** and '**`n`**', as in '**`<n|`** ... **`|n>`**', has a '**`weight`**' attribute value '**`normal`**'.

Text types can also inherit attributes from other text types. To facilitate this there are also virtual text types that are never used in the text, but can be ancestors of other text types and serve as repositories of attributes. Lastly it is possible to give several names to the same text type, and to give names to attribute values (e.g., '**`default style`**' is a standard name for a '**`style`**' attribute value).

If a text type has no value for an attribute, or has the value '**`none`**', a piece of text of that type has no value for the attribute, and gets the attribute value if it needs it from the surrounding text, or from an ***initial value*** for the attribute if there is no surrounding text. Thus since '**`p`**' has no '**`weight`**' attribute value, surrounding a paragraph by '**`<b|`** ... **`|b>`**' produces a bold paragraph, and not surrounding a paragraph by text with any '**`weight`**' attribute gets the initial '**`weight`**' attribute, which is '**`normal`**'.

Although there are many standard text types, each with standard attributes, a large piece of text typically defines new types specific to that text and resets some attributes of the standard types in a way specific to the particular text.

Text surrounded by named brackets like '**<b|** ... **|b>**' is converted by the text parser into a list which has the **.text-type** and **.label** extra attributes supplied by the named brackets. Similarly text prefaced by a named operator like '**<p>**' is converted to such a list. So, for example, the text

```
<p>
This is a <b|bold|b> sentence.
```

is converted by the text parser to

```
@1 = @2 {| .text-type = p |}
@2 = this is a @3 sentence :|
     .text-type = s
     .initiator = capital
     .terminator = "."
@3 = bold {| .text-type = b |}
```

Here the named operator '**<p>**' prefaces the text included in **@1**, the unnamed operator '**.**' ends the text included in **@2**, and the named brackets '**<b|** ... **|b>**' surround the text included in **@3**. The unnamed operator '**.**' terminates a piece of text that is assigned the '**s**' text type, the '**capital**' **.initiator**, and the '**.**' **.terminator**. Here the text type '**s**' denotes a 'sentence', and the result would be the same if the named operator '**<s>**' were explicitly included just before '**This**', as in

```
<p>
<s> This is a <b|bold|b> sentence.
```

A piece of text can be given a *label* by including the *label* in the named bracket or named operator which gives the piece of text its text type. These labels are internal to the document, and not printed. They are used to create references from one place in the document to another. For example:

```
<p: porch definition>
. . . <b: porch|porch|b> . . .
. . . . . . . . .
. . . <r|porch|r> . . .
. . . . . . . . .
. . . <r|porch definition|r> . . .
. . . . . . . . .
```

in which the **<r| |r>** bracketed *labels* are references to the labeled bold text and labeled paragraph. For convenience '**<k|***text***|k>**' can be used to abbreviate '**<b:** *text***|***text***|b>**', for example,

'**`<k|porch|k>`**' could be used above instead of '**`<b: porch|porch|b>`**' (the text type '**`k`**' denotes a 'keyword').

The text parser converts text into a list whose elements are lexemes and sublists, where in general each sublist has elements that are lexemes and sublists. The text types end up as **`.text-type`** attributes of these lists and sublists. Similarly *labels* included in named brackets or named operators end up as **`.label`** attributes. Operators frequently cause additional reformatting of the parse output. *Arguments* in named operators and brackets are passed to reformatters which use them to control the reformatting and may insert them as various attributes in the parsed output. As an example, the text

```
<section(1)|A Section Header|>
<p>
<s: HERE>An important sentence.
Another sentence.
```

is converted by the text parser to

```
@1 = @3 :|
    .text-type = section
    .level = 1
    .title = @2
@2 = A Section Header
@3 = @4 @5 {| .text-type = p |}
@4 = an important sentence :|
    .text-type = s
    .label = HERE
    .initiator = capital
    .terminator = "."
@5 = another sentence :|
    .text-type = s
    .initiator = capital
    .terminator = "."
```

Here the entire bracketed subexpression '**`<section(1)|`** ... **`|>`**' also is the lowest precedence operator which reformats everything that follows (and in general everything following up to the next operator of equal or lower precedence or a closing outer bracket) into a single list. It attaches the bracketed text to this list as the **`.title`** attribute. Also the '**`<section>`**' operator is programmed to attach its argument, '`1`' in this case, to the list as the **`.level`** attribute

In general named operators can be used in bracketed form to attach a bracketed expression as a

`.title` attribute to the same text the operator attaches its **`.text-type`** to. In general some named operators or brackets are programmed to take arguments which they may use to control formatting or compute text attribute values (such as **`.level`** in this example).

Note that closing brackets may omit their name, as '**`|>`**' does when it closes '**`<section(1)|`**' in this example. Its also allowed to use either an initial or final segment of the name in a closing named bracket, as in '**`<bold italic|`** ... **`|bold>`**' or '**`<bold italic|`** ... **`|italic>`**'. Arguments may be given in either the opening or closing bracket of a named bracket, but if given in both, must be identical in both. E.g.,

```
 <section(1)|... |section>|
 <section|... |section(1)>|
<section(1)|... |section(1)>|
```

are all acceptable and equivalent, but

```
<section(1)|... |section(2)>|
```

is illegal.

In the above example, '**`<p>`**' is a higher precedence operator than '**`<section>`**', and '**`<s>`**' is higher precedence than '**`<p>`**'. Also the sentence terminator '**`.`**' has the same precedence as '**`<s>`**' and these operators work together to form sentences. The '**`<s>`**' operator is optional and is only given in the example in order to assign a **`.label`** attribute to the first sentence. The second sentence has its '**`<s>`**' operator omitted.

Some text operators do <u>not</u> set the **`.text-type`** attribute, but instead set only a **`.separator`** attribute. The '**`,`**' operator is an example. Thus the text

```
this is, certainly, a phrase
```

is converted by the text parser to

```
@1 = @2 certainly @3 {| .separator = "," |}
@2 = this is
@3 = a phrase
```

Because the reformatted expression in this case is a list with no **`.text-type`** attribute, it can be assigned a **`.text-type`** attribute by a text operator of precedence lower than the precedent of '**`,`**'. Thus the text

```
<p>
```

```
    this is, certainly, a phrase
    <p>
    This is, certainly, a sentence.
```

is converted by the text parser to

```
    @1 = @2 @6
    @2 = @3 certainly @4 :|
        .separator = ","
        .text-type p
    @4 = this is
    @5 = a phrase
    @6 = @7 {| .text-type p |}
    @7 = @8 certainly @9 :|
        .separator = ","
        .initiator = capitalize
        .terminator = "."
        .text-type s
    @8 = this is
    @9 = a sentence
```

Unnamed brackets, namely '**(** ... **)**', '**[** ... **]**', and '**{** ... **}**', produce bracketed subexpressions just as they do for other parsers. Thus the text

```
    the book (Bartholomew 2043) previously mentioned
```

is converted by the text parser to

```
    @1 = the book @2 previously mentioned
    @2 = Bartholomew 2043 :|
        .initiator = "("
        .terminator = ")"
```

If you want to eliminate spacing where it would normally occur, you can use ***glue brackets***, '**<g|** ... **|g>**', which can also be implied by the absence of spacing in the input. Thus

```
    the book[Bartholomew 2043] previously mentioned
```

is equivalent to

```
    the <g|book[Bartholomew 2043]|g> previously mentioned
```

and both are converted by the text parser to

```
@1 = the @2 previously mentioned
@2 = book @3 {| .text-type = g |}
@3 = Bartholomew 2043 :|
     .initiator = "["
     .terminator = "]"
```

8.8.1 Standard Text Operators

The following are the standard text operators

precedence	reformatter	operator	meaning
-10000+100n	**text-prefix**	**<section(**n**)>**	section header
-1100	**text-prefix**	**<tr>**	table row
-1000	**text-prefix**	**<ti(**m**,**n**)>**	table item
-500	**text-prefix**	**<li>**	list item
0	**text-prefix**	**<p(**i**)>**	paragraph
100	**sentence**	**<s>**	sentence
		.	sentence terminator
		!	sentence terminator
		?	sentence terminator
200	**text-separator**	**;**	subsentence separator
		:	subsentence separator
300	**text-separator**	**-**	phrase separator
		--	phrase separator
400	**text-separator**	**,**	phrase separator

text-prefix **<section(**n**)>** **<tr>** **<ti(**m**,**n**)>** **<li>** **<p>**

These operators have different precedences, and so cannot appear together in the same expression.

The expression must have the form

$$\{ \textit{ operator subexpression subexpression}^\star \}^\star$$

That is, *operators* (e.g., '**<p>**') must begin the expression and may not be consecutive or end the expression. Each operator and the *subexpressions* following it are reformatted into a list whose **.text-type** attribute is given by the operator. If there is one *operator*, this reformatted list becomes the reformatted expression. If there is more than one, a list of these reformatted lists becomes the reformatted expression.

The list of *subexpressions* following an *operator* is parsed as an expression. The result is then converted to a list with no **`.text-type`** attribute; if it is not already such, it is made the sole element of a new list. Then the **`.text-type`** and **`.label`** attributes of this list are set from the *operator* and the result is the reformatted list for that *operator*.

If the operator brackets text, that text becomes the **`.title`** attribute value of the reformatted list.

The '**`<section>`**' operator takes an additional argument n (default **`0`**) which becomes the **`.level`** attribute of the reformated list. Also, $100n$ is added to the precedence of the operator.

The '**`<ti>`**' table item operator takes an additional arguments m and n (defaults **`1`** and **`1`**) which become the **`.rows`** and **`.columns`** attributes, respectively, of the reformated list. These specify that the table item is to span m rows and n columns. m can be omitted, as in **`<ti(`**n**`)>`**.

The '**`<p>`**' operator takes an additional argument i (default **`0`**) which becomes the **`.indent`** attribute of the reformated list. This specifies that the entire paragraph is to be indented.

[TBD: maybe not the **`.indent`** attribute.]

`sentence` **`<s>   .   ?   !`**

The expression must have the form

{ **`<s>`**-*option subexpression subexpression** *terminator* }*

where the *terminator* is one of '**`.`**', '**`?`**', or '**`!`**'.

That is, it must consist of a sequence of '***sentences***' each beginning with an optional '**`<s>`**', each ending with one of the *terminators*, and each being non-empty otherwise. Each sentence is reformatted into a list whose **`.text-type`** is '**`s`**', whose **`.initiator`** may be '**`capitalize`**', and whose **`.terminator`** is the *terminator* at the end of the sentence. If there is just one sentence, its reformatted list becomes the reformatted expression. If there is more than one, a list of these reformatted sentence lists becomes the reformatted expression.

The list of *subexpressions* in a sentence is parsed as an expression. The result is then converted to a list with no **`.text-type`**, **`.initiator`**, or **`.terminator`** attributes; if it is not already such, it is made the sole element of a new list. Then the **`.text-type`** attribute of this list is set to '**`s`**', the **`.label`** attribute of this list is set to any value provided by an optional '`<s:` *label* `>`' *operator*, and the `.terminator` is set to the *terminator*. If the first element of the list is a capitalized word, it is decapitalized and the list `.initiator` is set to '`capitalize`'. This list is then the reformatted list for the sentence.

`text-separator` `-  --  ,  ;  :`

Here '`-`' (or '`--`'), '`,`', and '`;`' (or '`:`') have different precedences, and so cannot appear together in the same expression.

The expression must have the form

$$\{ \textit{ subexpression subexpression}^\star \textit{ operator } \}^\star \textit{ subexpression}^+$$

where all the *operators* must be the same (i.e., if you mix '`;`' and '`:`' or '`-`' and '`--`' in the same expression, the expression will be declared illegal by the reformatter).

That is, the expression must consist of a sequence of non-empty *subexpression* lists separated by *operators*. The entire expression is reformatted as a list whose `.separator` is the *operator*. The elements of this list are the lists that are the *subexpressions* between operators, though if there is only one *subexpression* between two operators, it becomes an element of the final list.

8.8.2 Standard Text Brackets

bracket	meaning
`<b|`...**`|b>`**	boldface text (heavy weight)
`<n|`...**`|n>`**	normal text (normal weight)
`<i|`...**`|i>`**	italic text (TBD)
`<r|`...**`|r>`**	roman text (TBD)
`<em|`...**`|em>`**	emphasized text (TBD)
`<table|`...**`|table>`**	table (TBD)
`<list|`...**`|list>`**	list (TBD)
`<g|`...**`|g>`**	glued text (TBD)

8.8.3 Text Mark Attributes

OBSOLETE - incorporate into above

Text parsing is performed by the **`-TEXT-PARSER-`**, which is the parser for subexpressions of the `` `...' ``, etc. matchfix operators. The **`|`** format separator and sentence and paragraph ends are recognized by text processing, while operators, qualifiers, qualifier shortcuts, and the **`::?`**, **`<:>`**, **`::>`**, **`@@`**, and **`??`** marks are <u>not</u> recognized.

Text parsing is normally done in the context of a pair of matched *quotes*, and in this context *white-space* pre-lexemes become lexemes. Note that *white-space* lexemes all consist of zero or more *vertical space* characters followed by zero or more *single-space* characters (reference: TBD: make white space lexemes sometimes?). There are three kinds of *white-space* lexemes used by text parsing:

Spacer Lexemes. A ***spacer*** lexeme is a *white-space* lexeme containing only single spaces. Spacers are used in text parsing if they follow a **`|`** format separator on a line.

Line Separator Lexemes. A ***line separator*** lexeme is a *white-space* lexeme that contains a single *line-feed* character and no other *vertical-space* characters. Such lexemes separate non-blank lines, and are used by text parsing to end lines containing a **`|`** format separator.

Blank Line Lexemes. A ***blank line*** lexeme is a *white-space* lexeme that contains either two or more *line-feed* characters or contains a *vertical-space* character that is not a *line-feed* character. Such lexemes represent blank lines between non-blank lines, and are used by text parsing both to end lines containing a **`|`** format separator and to separate paragraphs.

8.8.4 Section, Paragraph, and Sentence Parsing

OBSOLETE - incorporate into above

If the text being parsed does not contain any format separators, the text is parsed into phrases, sentences, and paragraphs.

First the text is divided by blank line lexemes into paragraphs. The sequence of paragraphs comprises a section.

Then in each paragraph, sentence terminators are located. White-space lexemes in the paragraph are deleted after sentence terminators are located. Each sequence of lexemes or subexpressions ending in a sentence terminator is made into a sentence, and any non-empty sequence of lexemes or subexpressions following the last sentence terminator is made into a phrase. The paragraph is then a sequence of zero or more sentences possibly followed by a phrase. However, a paragraph cannot be empty.

The syntax of the result is:

section ::= **`[-SECTION-`** *paragraph*$^+$ **`]`**

paragraph ::= **[-PARAGRAPH-** *sentence*$^+$ **]**
| **[-PARAGRAPH-** *sentence** *phrase* **]**

sentence ::= **[-SENTENCE-** *sentence-non-terminator** *sentence-terminator* **]**

phrase ::= **[** *sentence-non-terminator sentence-non-terminator sentence-non-terminator** **]**
| *sentence-non-terminator*

sentence-terminator ::= **.** | **?** | **!**

sentence-non-terminator ::= *symbol* | *subexpression*

Note that a *phrase* with more than one *sentence-non-terminator* is a list, but a *phrase* with just one *sentence-non-terminator* is not a list, but just the single *sentence-non-terminator* by itself.

There are several rules that modify the description just given:

Sentence Terminator Rule. A ***sentence-terminator*** is any lexeme that is syntactically a sentence terminator, that is not preceded by a *white-space* lexeme, and that is followed by a *white-space* lexeme, a *closing-mark* lexeme, or the end of the lexeme sequence. All other lexemes and all subexpressions are ***sentence-non-terminators***.

Initial Capitalization Rule. A *word* consisting of an initial capital letter followed by zero or more lower case letters is converted to an all lower case word if it begins a sentence. A *word* consisting of an initial ^ followed by an upper case letter followed by zero or more lower case letters has the initial ^ removed.

Text Simplification Rule. If the **-TEXT-PARSER-** is to return a *section* with just one *paragraph* and that *paragraph* contains nothing but just one *sentence* or *phrase*, then just the *sentence* or *phrase* is returned. Otherwise, if a *section* with just one *paragraph* is to be returned, just the *paragraph* is returned.

Some examples follow:

```
'the wife of Bob'     parses as  [-PHRASE- the wife of Bob]

'She hit the ball.'   parses as  [-SENTENCE- she hit the ball .]

                                 [-PARAGRAPH-
'^Bill swung.                        [-SENTENCE- Bill swung .]
  But he missed!'     parses as      [-SENTENCE- but he missed !]]

                                 [-SECTION-
                                    [-PARAGRAPH-
'^I liked                              [-SENTENCE-
  the party.                               I liked the party .]]
                      parses as     [-PARAGRAPH-
  Later, we caught                     [-SENTENCE-
  the bus.'                                later , we caught
                                                   the bus .]]]
```

Note that capitalized words like proper names and '`I`' need to be prefixed by '`^` ' if they begin a sentence or phrase.

8.8.5 Text with Format Separators

TBD

8.9 The Parsing Algorithm

Parsing is done by a recursive descent left-to-right algorithm. A parser is called to parse each subexpression. This parser is given the following explicit arguments:

a list of lexemes to parse
an optional (closing) bracket definition
an optional (terminating) operator precedence

The bracket definition, if given, specifies a closing bracket which must appear in the list of lexemes. When the parser finds this bracket outside other brackets, the parser terminates the parse.

The operator precedence, if given, specifies that infix or nofix operators of this and lower precedence will terminate the parsing if they are encountered in the list of lexemes.

In addition the parser is given the following implicit arguments:

the parsing definition stack
the parsing selector set stack

The parser returns

the parsed expression
the unparsed final segment of the input lexeme list

When parsing is terminated by finding a closing bracket matching the closing bracket definition argument, then the part of the input lexeme list after this closing bracket is returned as the unparsed final segment of the input lexeme list. When parsing is terminated by finding a terminating operator of precedence equal to or lower than an operator precedence argument, the part of the input lexeme list beginning with this terminating operator is returned as the unparsed final segment of the input lexeme list. Otherwise the returned unparsed final segment of the input lexeme list is empty.

There are two standard kinds of parser: the operator parser and the text parser. These use somewhat different algorithms.

8.9.1 The Text Parser

The text parser divides the input into paragraphs that are separated by blank lines.

Paragraphs are classified as tabular or free-form according to whether their first line is a tabular format line. A ***tabular format line*** contains optional whitespace characters, followed by a '**+**', followed by any number of '**+**' or '**−**' characters, followed by a '**+**' that ends the line, except that superfluous whitespace characters are allowed after the 'line ending' +. An example of a tabular paragraph is:

```
+------------------------+---------+------+
 ice melt                 40 lbs    $4.50
 2x4's, 8ft               10       $27.70
 16d nails                 2 lbs     3.21
```

A paragraph that is not tabular is ***free-form***. Each free-form paragraph is scanned for explicitly bracketed subexpressions, phrase separators, and sentence terminators. The paragraph is divided into sentences using sentence terminators outside brackets, and each sentence is given **<* *>** implicit brackets. Each sentence is then divided into phrases if it has any phrase separators, and

9 Expression Evaluation

An expression is evaluated by first searching for an expression definition that matches the expression. Expression definitions have patterns that are matched to the expression.

An expression definition may have qualifiers. Some qualifiers are logical expressions that must evaluate to true in order for the expression definition match to succeed. Other qualifiers require that subexpressions match patterns or each other in order for the expression definition match to succeed. Still other qualifiers provide default values for missing arguments.

An expression definition may have an expression block that executes in order to produce a value for the expression if the definition matches. If a matching expression definition has no expression block, the expression evaluates to '`true`'. If no expression definition matches an expression, the expression evaluates to '`false`'.

Expression definitions are searched for in a context, which is a list of expression definitions and pointers to other contexts. Each point in the program has a lexical context, which is used by default for expression evaluation. By default expressions are evaluated in the lexical context of the place in the program code where the expression appears. In addition, there may be other contexts in which expressions may be evaluated that effectively implement either data bases or alternative programming languages.

When an expression definition is found whose pattern matches an expression, a set of qualifiers are established that must be satisfied to complete the match. There are five kinds of qualifiers: syntax equations, pattern equations, default equations, alternative equations, and guards. Syntax equations equate expressions that involve syntax variables, which are names beginning with a `$` followed by a capital letter. Pattern equations are like syntax equations but involve a pattern which can represent one of many possible syntax expressions. Default equations assign values to those syntax variables that where discarded when a pattern was unfolded to become the syntax expression it actually represented. Alternative equations specify alternatives for simple atomic clauses that may be in a pattern. Guards are just expressions that must evaluate to `true` in order for the match to succeed.

The syntax equations are solved during the matching process, to produce values for the syntax variables. Patterns are unfolded during the matching process to become the syntax expressions they represent. Guards are evaluated, and in some cases the value of a syntax variable value is evaluated. If the syntax equations or pattern equations are inconsistent and cannot be solved, or if a guard or syntax variable value that must be evaluated cannot be evaluated, or if a guard evaluates to `false`, the match fails. Evaluation of the guards or syntax variable values may result in additional matches which generate more qualifiers involving more syntax variables.

The matching process must consider all possible choices of which definitions to match to an expression being evaluated. If all choices fail, the expression evaluates to **`false`**. If only one choice succeeds, that choice is used to evaluate the expression. If several choices succeed, the situation becomes ambiguous, and the evaluation proceeds according to the matching mode, as is described below (9.5 $^{p\,124}$).

A syntax variable name beginning with **`$`** denotes an unevaluated expression. The same name without the **`$`** denotes the value of this unevaluated expression when it is evaluated in the context in which the unevaluated expression appears. This last value is called the '***evaluation***' of the unevaluated expression. Thus, for example, **`$X`** denotes an unevaluated expression and **`X`** denotes the evaluation of that expression.

As an example, given the syntax equation

```
foo(2+2) :=: foo($X)
```

(where **`:=:`** means 'is syntactically equal to'), we get the syntax equation

```
2+2 :=: $X
```

that defines **`$X`**. If there is then a guard '**`X is a number`**', the expression **`$X`**, namely '**`2+2`**', will be evaluated in its context, the context in which '**`foo(2+2)`**' appeared, to produce the value **`4`**, which will become the value of **`X`**, so the guard will become '**`4 is a number`**'.

An expression cannot be evaluated if it contains any undefined syntax variables. For example, if we began with the syntax equation

```
foo(2+$Y) :=: foo($X)
```

and guard '**`X is a number`**', **`$X`** would be '**`2+$Y`**' which could not be evaluated. If at some later time in the matching process the syntax equation

```
$Y :=: 7
```

is generated, then the value of **`$X`** becomes '**`2+7`**' which can now be evaluated to produce the value **`9`** for **`X`** and allow the guard to become '**`9 is a number`**'.

In the above **`$X`** is an 'unevaluated syntax variable' and **`X`** is its 'associated' 'evaluated syntax variable'. It is possible to use **`X`** in a definition where **`$X`** is actually meant, but if that is done once, then it must be done for every occurrence of **`$X`** in the definition, and the only computation that may be performed on the (unevaluated) value of **`$X`** is to evaluate it to produce a value for **`X`**.

Except for choices of which definitions to match to which expressions, the entire matching process is monotonic. This means the order in which syntax equations are solved, patterns are unfolded, guards are evaluated, and unevaluated syntax variable values are evaluated does not matter.

In order to make the matching process monotonic, pattern unfolding must be monotonic. This means that if undefined syntax variables in a pattern equation whose pattern has been unfolded are given arbitrary values later, the pattern unfolding must still be valid and unambiguous.

Similarly guard evaluations must not have side effects. In other words, evaluating a guard cannot affect any future valuations. Side effects can only occur during block execution, so any blocks executed during the guard evaluation process must not have side effects.

Lastly, required evaluations of unevaluated syntax variable values must be monotonic. For example, given the syntax variable **`$X`**, if the value of **`X`** is required, the evaluation of the value of **`$X`** must be monotonic. In order to ensure this, an unevaluated expression is not evaluated until all syntax variables it contains have been given values, so that the expression to be evaluated does not contain any undefined syntax variables. Furthermore, evaluation of such an expression may not have side effects. Again this last means that any blocks executed during the evaluation of the expression must not have side effects. If an evaluation is needed, but the expression to be evaluated contains a syntax variable that never becomes defined, then the matching process fails for the expression definition, just as it would if a guard evaluated to false.

9.1 Expression Definitions

Expression definitions have the syntax:

expression-definition ::= *pattern* **`<--`** *qualifier-list*$^?$ *block*$^?$

pattern ::= see p110]

qualifier-list ::= *qualifier* { **`,`** *qualifier* }*

qualifier ::= *syntax-equation* | *pattern-equation* | *default-equation*
| *alternative-equation* | *guard*

syntax-equation ::= *syntax-expression* **`:=:`** *syntax-expression* [see p106]

pattern-equation ::= *syntax-expression* **`:=~`** *pattern* [see p110]

default-equation ::= *syntax-variable* **`:=?`** *syntax-expression* [see p116]

alternative-equation ::=
syntax-variable **`:=|`** *alternative* **`|`** *alternative* { **`|`** *alternative* }* [see p117]

alternative ::= *name* [see p33]

guard ::= *subexpression*

block ::= **{ }** | **{** *statement* { **;** *statement* }* **}**

statement ::= see p127

The ***pattern*** of an expression definition is the subexpression before the **<--**. If the expression definition with pattern p is being matched to the expression e, the pattern equation e **:=~** p is asserted and alternative equations are used to unfold the pattern. If this match succeeds, syntax and pattern equation qualifiers in the definition are asserted, and the definition guards are required to evaluate to **true**. Default equations are asserted as necessary.

Just before an expression definition is matched to an expression, the definition is copied, and the act of copying creates a new set of variables that are distinct from any previous variables. Thus if **$X** appears in a definition, each use of the definition will involve a different variable named **$X**.

9.2 Syntax Equations

Although pattern equations are asserted before syntax equations are asserted, a knowledge of syntax equations is required to understand pattern equations.

A ***syntax equation*** asserts syntactic identity between two ***syntax expressions***. A syntax expression is a syntax variable, an atom, or an object whose attribute values are single valued with these values being syntax expressions. Atoms are represented by *atom-names* (p56) and objects by *bracketed-lists* (p64) where **{| |}** brackets may be omitted if the object representation is surrounded by operators or other suitable delimiters (see p**??**).

The syntax of a syntax equation is:

syntax-equation ::= *syntax-expression* **:=:** *syntax-expression*

syntax-expression ::= *syntax-variable* | *atom* | *syntax-object*

syntax-variable ::= **$** *capitalized-word* as a single lexeme
| **$ (** *name* beginning with *capitalized-word* **)**
| *place-holding-syntax-variable*
| *evaluated-syntax-variable*

capitalized-word ::= *word* beginning with capitalized letter in which all letters are capitalized

place-holding-syntax-variable ::= **$$** [see 9.4.5$^{p\,121}$]

evaluated-syntax-variable ::= *capitalized-word* [see 9.4.6$^{p\,121}$]

> ***atom*** ::= *atom-name*
>
> ***syntax-object*** ::= *object* with single valued attributes
> whose attribute values are *syntax-expressions*

All the different forms of *syntax-variable* are abbreviations for the form:

`$(`*name* beginning with *captitalized-word*`)`

A syntax variable of the form `$`*captitalized-word* abbreviates `$(`*captitalized-word*`)`.

Syntax equations can be consistent or inconsistent according to the:

Syntax Equation Consistency Rules:

1. If either syntax expression has both an **`.initiator`** equal to `"("` and a **`.terminator`** equal to `")"`, its **`.initiator`** and **`.terminator`** attributes are removed and these rules are applied to the result.
2. If either syntax expression is an object with only a single attribute that is a list element, the object is replace by this list element, and these rules are applied to the result.
3. If one syntax expression is a syntax variable, the syntax equation is consistent.
4. If the two syntax expressions are both atoms and these atoms are equal, the syntax equation is consistent.
5. If the two syntax expressions are both objects, and these objects have the same attributes (including the same number of list elements), the syntax expression is consistent and the Propagation Rule below is applied.
6. If the two syntax expressions are both objects, these objects have some non-list element attributes, and both objects have the same non-list element attributes, the syntax expression is consistent and the Propagation Rule below is applied.

Syntax equations that are not consistent are ***inconsistent***.

Note that rules (1) and (2) may be applied recursively before the other rules are applied.

Thus the following are examples:

`5 :=: 6`	inconsistent; Rule (4) fails
`5 :=: 5`	consistent by Rule (4)
`5 :=: {|5|}`	consistent by Rules (2) and (4)
`5 + 6 :=: $X`	consistent by Rule (3)
`5 + 6 :=: ($X)`	consistent by Rules (1) and (3)
`5 + 6 :=: A + B`	consistent by Rule (5)
`[5 + 6] :=: [$X]`	consistent by Rule (6)
`[5, 6] :=: {{A, B, C}}`	consistent by Rule (6)

The following rule is applied when two objects are consistent according to Syntax Equation Consistency Rules (5) or (6):

Syntax Equation Propagation Rules: From a consistent syntax equation asserting the identity of two objects new equations are deduced as follows:

1. If consistency was deduced according to Syntax Equation Consistency Rule (5), a new equation is deduced for every attribute label shared by the two objects. Given such an attribute label, the values of the attribute for the two objects are asserted to be syntactically identical.
2. If consistency was deduced according to Syntax Equation Consistency Rule (6), a new equation is deduced for every non-list-element attribute label shared by the two objects. Given such an attribute label, the values of the attribute for the two objects are asserted to be syntactically identical.

   An additional syntax equation is asserted between the two objects with all their non-list-element attributes removed.

The deduced equations need not be consistent in order to be deduced.

For example, from

```
{| sort ($X) | foo = 5 } :=: {| sort ( 5 + 6 ) | foo = 10 }
```

we can deduce by propagation that

```
sort :=: sort
($X) :=: ( 5 + 6 )
   5 :=: 10
```

The deduced equations need not be consistent in order to be deduced. In this case the first two equations are consistent and the third is not.

In addition to the Propagation Rule, we apply the following, where e_1, e_2, and e_3 are subexpressions of the original syntax equations:

> ***Subexpression Symmetry Rule***: From e_1 `:=:` e_2 we deduce e_2 `:=:` e_1.
>
> ***Subexpression Transitivity Rule***: From e_1 `:=:` e_2 and e_2 `:=:` e_3 we deduce e_1 `:=:` e_3.

It is very important to notice that so far each subexpression of an original syntax equation is distinct in all the deduced equations. In particular, two different subexpressions are not necessarily `:=:` even if they are syntactically the same. For example, given the original syntax equation:

```
sort ($X, $Y) :=: sort ( 5, 5 )
```

we derive by propagation that

$$\begin{array}{rcl} \texttt{sort} & \texttt{:=:} & \texttt{sort} \\ \texttt{,} & \texttt{:=:} & \texttt{,} \\ \texttt{\$X} & \texttt{:=:} & \texttt{5}^1 \\ \texttt{\$Y} & \texttt{:=:} & \texttt{5}^2 \end{array}$$

where we have put superscripts ^{1} and ^{2} on the two **`5`**'s to indicate that they are different subexpressions of the original syntax equations. But our rules do not generate the syntax equation $\texttt{5}^1$ `:=:` $\texttt{5}^2$ and as a consequence they cannot generate **`$X := $Y`**.

There is one last rule. Given subexpressions v_1, v_2, e_1, and e_2 of the original syntax equations:

> ***Syntax Variable Identity Rule***:
> If v_1 and v_2 name the same syntax variable,
> then from v_1 `:=:` e_1 and v_2 `:=:` e_2 we deduce e_1 `:=:` e_2.

This if we modified the above example to:

```
sort ($X, $X) :=: sort ( 5, 5 )
```

we derive by propagation that

$$\begin{array}{rcl} \texttt{sort} & \texttt{:=:} & \texttt{sort} \\ \texttt{,} & \texttt{:=:} & \texttt{,} \\ \texttt{\$X} & \texttt{:=:} & \texttt{5}^1 \\ \texttt{\$X} & \texttt{:=:} & \texttt{5}^2 \end{array}$$

and by syntax variable identity that

$$5^1 \;\texttt{:=:}\; 5^2$$

since both occurances of **$X** are identified. This last is consistent.

But if instead we started we started instead with:

```
sort ($X, $X) :=: sort ( 5, 6 )
```

we would end with:

```
5 :=: 6
```

which is inconsistent.

We have given all the deduction rules used to solve syntax equations. Note that these rules do not create new syntax expressions: all the syntax expressions in any deduced equation were subexpressions in previous equations. Because of this, an algorithm that keeps making deductions until no new deductions can be made will stop quite quickly.

9.3 Patterns

A ***pattern*** is an object that represents one of a set of syntax expressions. Patterns are used in *pattern-equations*, which identify a *syntax-expression* with a *pattern*. The syntax of a pattern equations and patterns is:

pattern-equation ::= *syntax-expression* **:=~** *pattern*

pattern ::= *pattern-object*
| *syntax-expression* that is not a *pattern-object*

pattern-object ::= *object* all of whose attributes have single values,
whose list portion is a *pattern-list*,
and whose non-list-element attribute values
are *patterns*

pattern-list ::= *pattern-term* { *pattern-separator pattern-term* }*

pattern-separator ::= **~**

pattern-term ::= *pattern-clause pattern-qualifier*$^?$

pattern-clause ::= *pattern-factor*$^+$ [see text]

pattern-factor ::= *atom-name* | *syntax-variable* | *bracketed-pattern*

bracketed-pattern ::= *pattern* with **`.initiator`** and **`.terminator`** attributes

pattern-qualifier ::= **`?`** | **`??`** | **`...`** | **`...?`**

A '***key***' of a *pattern-clause* an *atom-name* that is a *pattern-factor* in the clause or in one of the alternatives of an *alternative-syntax-variable* that is in the clause (an *alternative-syntax-variable* is replaced by one of its alternatives when the pattern is unfolded: see 9.4.2$^{p\,117}$), or a key may be an *object* with an **`.initiator`** other than **`"("`** that is a *pattern-factor* in the clause (informally the **`.initiator`** is the key).

1. Every *pattern-clause* must contain a key.
2. The first *pattern-term* of a *pattern* may not have any *pattern-qualifier* other than '**`...`**' (so the first *pattern-clause* must be copied at the beginning of the unfolded pattern).

For example, the *pattern*

```
replace ($X) ~ in ($Y) ~ by ($Z) ?
```

contains the following keys: **`replace`**, **`in`**, and **`by`**.

A pattern represents one of many possible syntax expressions, which are called '***unfolded patterns***' or '***unfoldings***' of the pattern, and are made by concatenating the *pattern-clauses* in different ways. For example, the *pattern* just given has three unfoldings:

```
replace ($X) in ($Y) by ($Z)
or
replace ($X) by ($Z) in ($Y)
or
replace ($X) in ($Y)
```

The rules for unfolding are:

Pattern Unfolding Rules:

1. An unfolding of a *pattern* is a sequence of copies of *pattern-clauses* from the *pattern* taken in any order, except a copy of the first *pattern-clause* in the *pattern* must be first in the unfolded *pattern*. No *pattern-separators* or *pattern-qualifiers* are included in the unfolded *pattern*.
2. A *pattern-clause* not followed by a *pattern-qualifier* (**?**, **??**, **...**, or **...?**) in the *pattern* must be copied exactly once in the unfolded *pattern*.
3. A *pattern-clause* followed by '**?**' in the *pattern* is an ***optional pattern clause*** and must be copied exactly once or not at all in the unfolded *pattern*. See 9.4.1 $^{p\,116}$ for examples.
4. A *pattern-clause* followed by '**??**' in the *pattern* is an ***optional truth-value pattern clause***. It must end with a *pattern-factor* of the form '**(** *syntax-variable* **)**'. Like an optional *pattern-clause*, this *pattern-clause* must be copied exactly once or not at all in the unfolded *pattern*. The copy may omit the last *pattern-factor*, in which case the omitted *syntax-variable* will default to '**TRUE'**. Or the copy may both omit the last factor and add the factor '**not**' to the beginning of the copy, in which case the omitted *syntax-variable* will default to '**FALSE**'. See 9.4.3 $^{p\,118}$ for examples.
5. A *pattern-clause* followed by '**...**' in the *pattern* is a ***repeated pattern clause*** and must be copied one or more times in the unfolded *pattern*. See 9.4.4 $^{p\,120}$ for examples.
6. A *pattern-clause* followed by '**...?**' in the *pattern* is an ***optional repeated pattern clause*** and must be copied one or more times or not at all in the unfolded *pattern*. See 9.4.4 $^{p\,120}$ for examples.
7. A *syntax-variable* in a *pattern* that appears in an *alternative-equation* is called an ***alternative-syntax-variable*** and must be replaced in the unfolded *pattern* by one of the *alternatives* from the *alternative-equation*. These *alternatives* are sequences of *atoms*. If the replacement is a sequence of more than one atom, the replacement atoms are inserted in the unfolded pattern increasing the length of the unfolded pattern (and not simply replacing the *alternative-syntax-variable* by a single list object). See 9.4.2 $^{p\,117}$ for examples.
8. An attribute that does not have the **?** attribute flag must appear exactly once in the unfolded *pattern*.
9. An attribute that does have the **?** attribute flag and must appear exactly once or not at all in the unfolded *pattern*.

A consistent pattern equation produces a unique matching between the syntax expression on the right and an unfolding of the pattern on the left. Matching makes use of the concept of a '***termi-***

nating key' of a *pattern*. This is just a key of the *pattern* that can follow a '**`( )`**' bracketed element in an unfolding of the *pattern*. For example, given the pattern

```
replace ($X) ~ $WHERE ? ~ in ($Y) ~ by ($Z) ?
```

with **`$WHERE`** being an *alternative-syntax-variable* (see 9.4.2 $^{p\,117}$ for details) defined by

```
$WHERE :=: everywhere | at beginning | at end
```

then the following are terminating keys of the pattern: **`everywhere`**, **`at`**, **`in`**, **`by`**.

The rules governing the formation of the matching are:

Pattern Matching Rules:

1. If a syntax expression (equation left side) is not an object (it is a single atom or a syntax variable) it is replaced by a list object containing as its sole attribute a single list element equal to the original syntax expression before it is matched.
2. A matching is an order-preserving 1-1 correspondence between all the the list elements of the unfolded pattern (right side) and sequences of list elements of the syntax expression (left side). Each list element of the syntax expression must be included in one of the matched sequences.
3. Each atom of the pattern is matched to an identical atom of the syntax expression.
4. Each object of the pattern that does not have an **`.initiator`** equal to **`"("`**, or which has a terminator that is not **`")"`** or some other non-list-element attribute other than **`.separator`**, is matched to a single element of the of the syntax expression that is also an object. Both objects must have the same **`.initiator`** and **`.terminator`** values. The pattern object must have every non-list-element attribute that the syntax expression has. The syntax expression must have every non-list-element attribute that the pattern has unless the pattern attribute has the '**`?`**' attribute flag.
5. Each object of the pattern whose **`.initiator`** is **`"("`**, whose **`.terminator`** is **`")"`**, and which has no other non-list-element attributes, is either matched as in (4), or this object is matched to a (possibly empty) sequence of elements of the syntax expression such that the sequence contains no element that could be matched to any terminating key of the original folded pattern, and no element that is a syntax variable, and the sequence is followed by (a) the end of the syntax expression, or (b) an syntax expression element matched using rules (3) or (4).

Not allowing a syntax expression element sequence around which parentheses have been omitted to contain any initial key of the original unfolded pattern means that no pattern clause of the original unfolded pattern could possibly match some part of the syntax expression that overlapped the sequence.

Then the pattern equation consistency rule is:

Pattern Equation Consistency Rule:
A pattern equation is ***consistent*** if either:

1. Its right side is not a *pattern-object*, but if the pattern equation is changed to a syntax equation by replacing **:=~** by **:=:**, the syntax equation is consistent.
2. Its right side is a *pattern-object* and the syntax expression on the left side is not a syntax variable or an object containing a list element that is a syntax variable. Then the pattern equation is consistent if the pattern has a unique unfolding such that there is a matching, and that matching is unique for the given unfolding.

   If the syntax expression on the left side contains a syntax variable or an object list element that is a syntax variable, processing the pattern equation is delayed until these syntax variables are given values, which are then substituted into the pattern equation. However, if there is no longer any possiblity of one of these syntax variables getting a value, the pattern equation is declared to be inconsistent.

The following rule is applied when two list elements match according to Pattern Equation Consistency Rules:

Pattern Equation Propagation Rules: When a pattern equation is consistent, the following deduce new pattern equations or syntax equations.

1. If the pattern equation was replaced by a syntax equation according to Pattern Equation Consistency Rule (1), new syntax equations are deduced as per the Syntax Equation Propagation Rules (p108).
2. If consistency was deduced according to Pattern Equation Consistency Rule (2), and bracketed list elements were matched according to Pattern Matching Rule (4), a new pattern equation is deduced for every non-list-element attribute label shared by the two matched bracketed lists. Given such an attribute label, if its value in the syntax expression is **S** and its value in the pattern is **P** the pattern equation **S :=~ P** is asserted.

   An additional pattern equation is asserted between the two bracketed lists with all their non-list-element attributes removed.

3. If consistency was deduced according to Pattern Equation Consistency Rule (2), and a pattern bracketed list element was matched to a sequence of syntax expression elements according to Pattern Matching Rule (5), a new pattern equation is deduced between the sequence of syntax expression elements, made into an unbracketed list, and the pattern bracketed list with its bracket attributes removed.

The deduced equations need not be consistent in order to be deduced.

Besides the unfolding, pattern equations differ from syntax equations in that they permit **( )** parentheses to be omitted from the syntax expression if the latter does not contain any terminating keys. For example,

```
sin my variable :=~ sin ($X)
```

matches and deduces

```
my variable :=: $X
```

As another example, given the pattern equation:

```
replace {{b}} by {{5 + c}} in (exp 1) :=~
    replace ($X) ~ in ($Y) ~ by ($Z) ?
```

The pattern may be unfolded so the equation becomes:

```
replace {{b}} by {{5 + c}} in (exp 1) :=~
    replace ($X) by ($Z) in ($Y)
```

from which we deduce using the Pattern Propagation Rule:

```
     {| {{b}} |} :=~ $X
 {| {{5 + c}} |} :=~ $Z
           exp 1 :=~ $Y
```

Note that this results in pattern equations and not syntax equations. But when we apply the Pattern Equation Consistency Rules to these equations, they first turn into syntax equations, and then the Syntax Consistency Rules eliminate the superfluous **{| |}**'s giving:

```
    {{b}} :=: $X
{{5 + c}} :=: $Z
    exp 1 :=: $Y
```

The matching algorithm in Section 9.5 $^{p\,124}$ starts with a single pattern equation. It checks this for consistency, and if consistent, deduces new equations by propagation. Each deduced equation is then checked for consistency, and more equations are deduced. The process stops quickly because no new syntax expressions or pattern expressions are created. If an inconsistent equation is discovered, the process fails. Otherwise when the process ends any syntax variables in the original have been assigned values or may be given default values.

9.4 Expression Definition Matching

When an expression is to be evaluated, various candidate expression definitions are examined to see if any match the expression. This process of matching an expression and an expression definition is as follows.

First a copy of the expression definition is made with each syntax variable replaced by a new syntax variable that is distinct from every other syntax variable.

Then a pattern equation of the form

expression-to-be-evaluated **:=~** *expression-definition-pattern*

is asserted, and along with all the *qualifiers* in the expression definition.

The consistency of these equations is determined, and new equations are deduced using propagation and some other qualifier related rules described below. If no inconsistent equations are deduced, the match succeeds. Otherwise the match fails.

The following sections describe different types of *qualifiers* and some other specialized aspects of this matching process.

9.4.1 Default Equations

When a pattern is unfolded, syntax variables in optional pattern clauses not included in the unfolded pattern are given default values provided by default equation qualifiers. These qualifiers have the syntax:

> ***default-equation*** ::= *syntax-variable* **:=?** *default-value*
>
> ***default-value*** ::= *syntax-expression*

For example, given the equations:

```
sort x  :=~  sort ($X) ~ in direction ($Y) ?
    $Y  :=?  ascending
```

the pattern in the first equation will be unfolded to give the equation:

```
sort x  :=~  sort ($X)
```

and because the variable **$Y** was dropped during the unfolding, the default equation that defines it will promoted to become the syntax equation:

```
$Y  :=:  ascending
```

thereby establishing a value for **$Y**. Default equations that are not promoted to become syntax equations by this mechanism are ignored. This mechanism does not apply to repeated pattern clauses or optional repeated pattern clauses.

9.4.2 Alternative Equations

An *alternative-equation* specifies that a syntax variable must take one of several alternative values.

> ***alternative-equation*** ::=
> *syntax-variable* **:=|** *alternative* **|** *alternative* { **|** *alternative* }*
>
> ***alternative*** ::= *atom*$^{+}$

For example,

```
DIRECTION :=| in ascending order | in descending order
```

Note that the *alternatives* are simple *atom* sequences, and cannot include more complex objects or *syntax-variables*.

The *syntax-variable* in an alternative equation is called an ***alternative-syntax-variable***. When such a variable appears in a pattern it must be replaced by one of the *alternatives* from the *alternative-equation* when the pattern is unfolded. The replacing *alternative* is integrated into the unfolded pattern as a sequence of elements (and not a single element that is a list of atoms) replacing the *alternative-syntax-variable* in the pattern.

For example, given the above *alternative-equation* and the *pattern-equation*

```
sort my data in descending order :=~ sort ($X) ~ $DIRECTION ?
```

the pattern in thus equation can be unfolded to make the *pattern-equation*

```
sort my data in descending order
              :=~
  sort ($X) in descending order
```

that is consistent. Note that in the *pattern* the atom '**`in`**' is a terminating key, and so cannot be part of the list of elements matched to **`$X`**.

Whenever a pattern containing a *alternative-syntax-variable* is unfolded, a particular *alternative* replaces the variable in the pattern, and the syntax equation

alternative-syntax-variable **`:=:`** *alternative*

is asserted. Thus in the above example where the pattern was unfolded, the equation

```
$DIRECTION :=: in descending order
```

is asserted.

A default equation can also be given for an *alternative-syntax-variable*. Thus if in our example

```
$DIRECTION :=? in ascending order
```

were also asserted, then from the pattern equation

```
sort my data :=~ sort ($X) ~ $DIRECTION ?
```

on would deduce among other things:

```
$DIRECTION :=: in ascending order
```

9.4.3 Optional Truth-Value Pattern Clauses

An ***optional truth-value pattern clause*** (with '**`??`**' *clause-qualifier*) must end with a syntax variable v in parenthesis, and this variable is treated specially in the following ways. First, the equations:

```
v :=? false
v :=~ $TRUTH-VALUE
$TRUTH-VALUE :=| true | false
```

are asserted. These ensure that if the *optional truth-value pattern clause* is omitted when its containing *pattern* is unfolded, the variable is given the value '**`false`**', and also ensure that the variable is given the value '**`true`**' or '**`false`**'. Second, if the clause appears on the left with its ending '**`(`***v***`)`**' missing, instead of '**`{} :=~`** *v*' being asserted, '**`true :=~`** *v*' is asserted. And lastly, if the clause appears without its ending '**`(`***v***`)`**' but with '**`not`**' prefixed, it is recognized and '**`false :=~`** *v*' is asserted.

For example, consider the pattern equations:

```
             sort x :=~ sort ($X) ~ ascending ($ASCENDING) ??
   sort x ascending :=~ sort ($X) ~ ascending ($ASCENDING) ??
sort x not ascending :=~ sort ($X) ~ ascending ($ASCENDING) ??
sort x ascending yes :=~ sort ($X) ~ ascending ($ASCENDING) ??
```

These all produce the equations:

```
$ASCENDING :=? false
$ASCENDING :=~ $TRUTH-VALUE
$TRUTH-VALUE :=| true | false
```

The first pattern equation produces by the rule for default variables:

```
$ASCENDING :=: false
```

The second pattern equation produces by special rule:

```
$ASCENDING :=: true
```

The third pattern equation produces by special rule:

```
$ASCENDING :=: false
```

The fourth pattern equation produces by normal rules:

```
$ASCENDING :=: yes
```

and since we have

```
$ASCENDING :=~ $TRUTH-VALUE
$TRUTH-VALUE :=| true | false
```

we deduce

```
yes :=~ $TRUTH-VALUE
$TRUTH-VALUE :=| true | false
```

which is inconsistent.

9.4.4 Repeated Pattern Clauses

When a pattern is unfolded, a syntax variable in a repeated or optional repeated pattern clause is replaced in the unfolded pattern by new variables that represent elements of a list, and a new syntax equation setting the original variable to this list is created. For example, when the pattern in the equation

```
increment x by 5 by 10  :=~  increment ($X) ~ by ($Y) ...?
```

is unfolded, the following equations result:

```
increment x by 5 by 10 :=~
    increment ($X) by ($(Y 1)) by ($(Y 2))
$Y :=: ($(Y 1), $(Y 2))
```

With an optional repeated pattern clause it is also possible for the resulting list to be empty. Thus

```
increment x  :=~  increment ($X) ~ by ($Y) ...?
```

yields

```
increment x  :=~  increment ($X)
         $Y  :=:  ()
```

9.4.5 Place Holding Syntax Variables

A '**`$$`**' by itself can be used as a syntax variable called a ***place holding syntax variable*** that has the special property that each of its original equation occurrences is treated as if it were a different variable. The syntax is:

> ***place-holding-syntax-variable*** ::= **`$$`**

More specifically, whenever **`$$`** appears in an original equation, it is replaced by a unique new syntax variable name. For example, the pattern equation:

```
replace {b} by {5 + c} in (exp 1) :=:
    replace ($X) ~ by $$ ~ in $$
```

is treated as if it were

```
replace {b} by {5 + c} in (exp 1) :=:
    replace ($X) ~ by ($X.1$) ~ in ($X.2$)
```

where '**`$X.1$`**' and '**`$X.2$`**' are unique new syntax variable names created by the RECKON system.

9.4.6 Evaluated Syntax Variables

Syntax variables take values that are syntax expressions which are not evaluated. Such syntax variables are called ***unevaluated syntax variables***. With every unevaluated syntax variable there can be associated an ***evaluated syntax variable*** whose name is made by omitting the **`$`** from the beginning of the unevaluated syntax variable name. Thus **`X`** is associated with **`$X`** and **`X 5`** is associated with **`$(X 5)`**. The syntax is:

> ***unevaluated-syntax-variable*** ::= **`$`** *capitalized-word* as a single lexeme
> | **`$ (`** *name* beginning with a *capitalized-word* **`)`**
>
> ***evaluated-syntax-variable*** ::= *name* beginning with a *capitalized-word*

Informally, an evaluated syntax variable is assigned the value obtained by evaluating its associated unevaluated syntax variable's value. In order for this to happen, the unevaluated syntax variable must be evaluatable. This means that its value, a syntax expression, must not contain any unevaluatable syntax variables.

More formally:

Evaluatable Definition: An unevaluated syntax variable **$**$v$ is evaluatable if a syntax equation **$**$v$ **:=:** e has been deduced and every syntax variable in e is evaluatable. In particular, **$**$v$ is evaluatable if e does not contain any syntax variables.

Here e is called the ***unevaluated value*** of **$**$v$, and the ***completed unevaluated value*** of **$**$v$ is e with all the syntax variables it contains replaced by their completed unevaluated values.

Evaluation Rule: If an evaluated syntax variable v appears in a syntax equation and its associated unevaluated syntax variable **$**$v$ is evaluatable, then the syntax equation v **:=:** E is deduced, where E is the value obtained by evaluating the unevaluated value of **$**$v$ in the context in which the unevaluated value of **$**$v$ appears. It is an error if this evaluation has side effects (p136).

In this case v may be replaced by E whereever v appears.

For example, given the code

```
sum ($X) and ($Y) <-- is a number X, is a number Y
    { value = X + Y }
. . . . .
z = sum (1+1) and (5+5)
```

the call to '**sum**' in the last line is matched with the definition of '**sum**' in the first two lines giving rise to a number of deduced equations, among which are:

```
sum (1+1) and (5+5) :=~ sum ($X) and ($Y)
$X :=: (1+1)
$Y :=: (5+5)
X :=: 2
Y :=: 10
```

The last two equations are required to deduce the value of the evaluated variables **X** and **Y** that are needed for the guards **is a number X** and **is a number Y** and the block code statement '**value = X + Y**'. These last two equations are obtained by the Evaluation Rule. For example, to obtain the value **2** for the evaluated variable **X**, the value '**(1+1)**' of the a associated unevaluated variable **$X** is evaluated. '**(1+1)**' is evaluatable because it contains no syntax variables and for this reason is also its own 'completed unevaluated value'.

Semantically, evaluated syntax variables are not permitted in patterns. However, syntactically they can be placed in patterns according to the following:

Evaluated Variable Promotion Rule: If an evaluated syntax variable v appears in an expression definition in places where only an unevaluated syntax variable can appear (e.g., in patterns or the left sides of default and alternative equations), it is replaced in these places by its associated unevaluated variable **$**$v$ when the expression definition is initially copied, provided the unevaluated **$**$v$ does not appear anywhere in the expression definition. Note that in places where it is ambiguous whether an evaluated or unevaluated variable is intended, the evaluated variable will be used.

If an unevaluated syntax variable **$**$v$ appears anywhere in an expression definition, its associated evaluated syntax variable v is never replaced by **$**$v$, and therefore the evaluated variable v must not appear anywhere an evaluated syntax variable cannot appear.

For example, the definition

```
for every counter:
    on increment THIS ~ by Y ? <-- is a number Y, Y :=? 1:
        next count = count + Y
```

is treated as if it were

```
for every counter:
    on increment $THIS ~ by $Y ? <-- is a number Y, $Y :=? 1:
        next count = count + Y
```

while the definition

```
for every counter:
    on increment THIS ~ by Y ? <-- is a number Y, $Y :=? 1:
        next count = count + Y
```

would be in error because **Y** is used in the pattern but **$Y** is used in the default equation.

9.4.7 Guards

A ***guard*** is an expression definition qualifier that can be any of the following:

1. An expression. The expression must evaluate to **true** if its containing expression definition is to match.

   If the Evaluated Variable Promotion Rule is using used for a variable in such a guard, it is assumed that the variable is unevaluated.

   For example, in

```
        for every counter:
            on increment $THIS ~ by $Y ?
                    <-- is a number Y, $Y :=? 1:
                next count = count + Y
```

the expression '**is a number Y**' is a guard.

2. A syntax or pattern equation. The equation is asserted.

   If the Evaluated Variable Promotion Rule is using used for a variable in such a guard, it is assumed that the variable is unevaluated.

   If the left side of a pattern equation consists of just a single syntax variable, or is an object with list element that is a syntax variable, these syntax variables must be replaced by their values before the pattern equation is processed. If processing gets to the point where this is no longer possible, the pattern equation is inconsistent.

   For example, in

   ```
           sort ($X) ($D) <-- $D :=~ $DIRECTION,
               $DIRECTION :=| ascending | descending
   ```

   the pattern equation '**$D :=~ $DIRECTION**' is a pattern equation guard that cannot be processed untion **$D** is given a value.

9.5 The Expression Matching Algorithm

The ***evaluation algorithm*** inputs an expression to be evaluated and a context. The context (9.6 $^{p\,125}$) provides a list of expression definitions that may match the expression. Evaluating the expression requires searching contexts for expression definitions that match the expression.

An expression to be evaluated may contain syntax variables that are assigned values during matching. The result of matching is both a list of values for the expression and an assignment of values for these variables.

More than one definition may match an expression. Different matchings may lead to different expression values and values of syntax variables in the expression.

Note that evaluations of guards and evaluations used to find values of evaluated syntax variables are required to have no side effects, so the order of these evaluations does not matter. Given the choices of which definitions are matched to which expressions, the outcome of the matching process is uniquely determined, and does not depend on the order of syntax equation deductions, guard evaluations, etc., or on the order in which the choices are tried.

Matching may be done in any of the following ***matching modes***:

`first-value`
: The first definition tried that successfully matches is the only one used. Definitions are tried in the order they are given in the contexts used in evaluation.

`all-values`
: All possible choices are tried and the successful results are collected in a set of results. Each result in this set consists of a value for the expression being evaluated and values for each syntax variable in that expression.

`consistent-values`
: All possible choices are tried and the successful results are collected in a set of results. Then these results are tested to see if they are pairwise equal. All the values of the expression being evaluated must be equal, and all the values of each syntax variable contained in the expression must be equal. If all expression values are equal and all values of any syntax variable are equal, one expression value is returned as the value of the expression, and one syntax variable value is returned as the value of each syntax variable. If some of the expression values or syntax variable values are unequal, an error value giving the context of the evaluation and the unequal values or values is returned as the value of the expression or syntax variable.

If the expression match search process yields no matches at all, the expression is given the value '**`false`**', and variables in the expression are not given any values.

9.6 Contexts

An expression is evaluated in a context. The context is searched for an expression definition whose pattern matches the expression being evaluated, and that definition is then used to evaluate the expression.

A ***context*** is either a list or a set of expression definitions and other contexts. Context elements that are expression definitions are matched to the expression being evaluated. Context elements that are themselves contexts are searched recursively.

Context lists are searched in order if the matching mode is **`first-value`**, and are searched exhaustively otherwise. ***Context sets*** are always searched exhaustively.

There is a ***current context*** which is the value of the global variable '**`CURRENT CONTEXT`**'.

An expression definition is an object (p54) that consists of a pattern, an optional list of qualifiers, and an optional block. An *expression-definition* (p105) computes an expression definition. More specifically, the top level operator of an *expression-definition* is **<--** and this returns an expression definition object.

An *expression-definition* given by itself adds the expression definition it computes to the beginning of the current context.

The expression

> **a context**

computes a new context. Contexts are lists of expression definitions, other contexts, and context marks. The expression

> **a context mark**

computes a new context mark.

The statements

> **add** *expression-definition* **to** *context* { **beginning** | **end** }$^?$
> **add** *context* **to** *context* { **beginning** | **end** }$^?$
> **add** *context-mark* **to** *context* { **beginning** | **end** }$^?$

add elements to the context beginning or end, where the default is to add to the beginning (which is searched first). The statement

> **pop** *context-mark* **from** *context* { **beginning** | **end** }$^?$

removes elements from the context beginning or end until the context mark is removed. The default is to remove from the beginning. It is an error if the context mark is not an element of the context.

Contexts are like files, and can be read-write, read-only, or write-only. They can also be treated as lists or sets when searched (p125). These context attributes can be changed by the statement

> **make** *context context-mode*
> ***context-mode*** ::= **READ-WRITE**
> | **READ-ONLY**
> | **WRITE-ONLY**
> | **SET-SEARCH**
> | **LIST-SEARCH**

The list of context elements can be copied from the context object by

> **the list of** *context*

The result is a list object whose elements are expression definition, context, and context marker objects. This list object is not part of the context object, and changing this list does not change the context object.

10 Blocks

A block is a set of variables, values for some of the variables, and code for computing these values. Blocks are optional parts of *expression-definitions* (9.1 $^{p\,105}$).

10.1 Block Syntax

The code of a block has the following syntax:

block ::= *bip group* { *sequence-break group* }* *eip*

sequence-break ::= *bll* **------*** *ell*
(a logical line containing 1 word consisting of 5 or more **-**'s)

group ::= *declaration-group* | *statement-group*

declaration-group ::= *declaration*$^+$

statement-group ::= *statement*$^+$

declaration ::= *bll expression-definition ell*

expression-definition ::= see p105

statement ::= *bll statement-qualifier*$^?$ *unqualified-statement ell*

statement-qualifier ::= **`first`** | **`always`** | **`default`**

unqualified-statement ::= *variable-assignment-statement*
| *variable-assignment-subblock*
| *pattern-assignment-statement*
| *guarded-statement*

variable-assignment-statement
::= *output-variable* **=** *value-expression*
| **(** *output-variable* { **,** *output-variable* }* **)** **=** *value-expression*

variable-assignment-subblock
::= *output-variable* **=** *temporary-subblock*
| **(** *output-variable* { **,** *output-variable* }* **)** **=** *temporary-subblock*

value-expression ::= *expression* [see p**??**]

output-variable ::= *variable-name* | *next-variable*

next-variable ::= **`next`** *variable-name*

variable-name ::= *name* [see p33]

pattern-assignment-statement ::= *pattern* **`~=:`** *value-expression*

pattern ::= see p110

temporary-subblock ::= **:** *block*

guarded-statement ::= *if-statement* | *when-statement*

if-statement ::= **`if`** *guard* *permanent-subblock*
{ *ell* *bll* **`else if`** *guard* *permanent-subblock* }*
{ *ell* *bll* **`else`** *permanent-subblock* }$^?$

when-statement ::= **`when`** *guard* *permanent-subblock*

guard ::= *value-expression* | *assignment-statement*

permanent-subblock ::= **:** *block*

10.2 Block Variables

In order to evaluate a block, the variables of a block must be identified. These have *variable-names* that are one of the following:

1. *Variable-names* ***inherited*** from a *expression-definition* containing the block.
2. *Output-variables* in *assignment-statements*.
3. Syntax variables in the *patterns* of *pattern-assignment-statements*.
4. The evaluated syntax variable names associated with unevaluated syntax variable names that name block variables.

All block variables are one of these kinds of variables.

If a *name* that is not the *variable-name* of a block variable appears in a *value-expression*, that *name* does not name a block variable, and it must be defined by matching it to an expression definition in the block context.

For example, in

```
sum from X through Y
        <--- is an integer X, is an integer Y:
    block
```

the variable names **X** and **Y** are inherited by the *block* and name block variables.

As another example, in

```
when y = sort x:
    z = first
```

y and **z** are *output-variables* and therefore name block variables.

As a last example, in

```
$X + $Y ~=~ `5 + ( 7 * y )'
```

the variable names **$X** and **$Y** are pattern syntax variable names that name block variables. The associated evaluated syntax variable names '**X**' and '**Y**' also name block variables.

Variables of a *permanent-subblock* are also variables of that subblock's containing block. For example, in:

```
if X < 5:
    z = X + 10
```

the variable **z** is a variable of both the permanent subblock and a variable of its containing block (that includes the '**if**' statement).

Variables of a *temporary-subblock* are <u>not</u> also variables of that subblock's containing block, unless they are *output-variables* of the containing block's *variable-assignment-subblock* statement that conains the subblock. For example, in:

```
z =:
    w = X + 10
    z = w - 5
    y = w + 5
```

the variable **z** is a variable of both the temporary subblock and its containing block (and is the same variable named as an output variable in two places), but the variables **w** and and **y** are variables of the subblock but <u>not</u> variables of its containing block.

To avoid confusion, variables of a *temporary-subblock* that are <u>not</u> also variables of its containing block may not have the same names as variables of the containing block.

10.3 Block Evaluation

Block evaluation proceeds according to the following rules:

1. ***Blocks*** are divided into ***groups*** by ***sequence-breaks***. The *groups* are evaluated in order: each group being completely evaluated before the next group is evaluated. No part of a *group* can evaluate until all previous *groups* of the block have completely finished evaluating. Once a *group* starts to evaluate, no part of a previous *group* can evaluate.

   A *block* terminates when its last group terminates.

2. The ***declarations*** of a ***declaration-group*** are evaluated in order. Evaluation of a *declaration* just adds its definition to the current context. The *declaration-group* terminates when its last *declaration* has evaluated.

3. Evaluation within a ***statement-group*** is driven by availability of variable values, and not by the order of the *statements* in the group.

4. Each ***block variable*** (see 10.2 $^{p\,128}$) can be assigned at most one value: it is an error if the variable is assigned a value more than once during a block evaluation, even if all the values assigned are the same.

5. Block variables named in an *value-expression* are called ***input-variables***. Each *value-expression* in a *statement-group* is evaluated $\underline{\text{only after all}}$ its input variables have be given values.

6. Input variables named in *value-expressions* within a subblock that are not block variables of the subblock are called '***inherited-variables***'. Each *temporary-subblock* in a *statement-group* is evaluated $\underline{\text{only after all}}$ its inherited variables have be given values.

7. The *output-variables* in a *variable-assignment-statement* are given values as soon as the *value-expression* of the statement is evaluated.

8. The *output-variables* in a *variable-assignment-subblock* are given values by name in the *temporary-subblock*. For example, in:

   ```
   z =:
       y = 5
       x = 6
       z = x + y
   ```

   the two instances of the *variable-name* **z** refer to the same block variable, which is set by the statement '**z = x + y**'.

9. The *pattern-variables* in a *pattern-assignment-statement* are given values as soon as the *value-expression* of the statement is evaluated.

10. None of the statements in an *if-statement*'s first *permanent-subblock* are evaluated until the block's *guard* has been evaluated to true. If the guard evaluates to false, no statement in the *permanent-subblock* is ever evaluated.

    An '**`else if`**' *guard* is not evaluated until all *guards* preceeding it in the *if-statement* have been evaulated to false. The *permanent-subblock* following an '**`else if`**' *guard* is not evaluated until the *guard* is evaluated to true.

    The *permanent-subblock* following an '**`else`**' is not evaluated until all *guards* preceeding it in the *if-statement* have been evaluated to false.

    An example application of these rules is:

    ```
    z = y1
    z = y2
    if X > 5:
      y1 = 9
    else:
      y2 = 10
    ```

    where **`X`** is an inherited variable name. If **`X > 5`** is true, the variable **`y1`** is given the value **`9`** and then **`z`** is given this value, while the variable **`y2`** is never given a value. If **`X > 5`** is false, the variable **`y2`** is given the value **`10`** and then **`z`** is given this value, while the variable **`y1`** is never given a value.

11. A *when-statement* is just like an *if-statement* that has no '**`else if`**' or '**`else`**' parts.

12. A *statement-group* terminates when all the *value-expressions* in it (and its subblocks) have either been evaluated or cannot be evaluated because of lack of an input variable value or because of *guards*. Once a *statement-group* terminates, no part of the group can be evaluated, even if another group later in the block defines that part's input variables.

    However, *statement-group* evaluation is additionally modified by *statement-qualifiers*: see Default Statements 10.4 *p* 132 and Iteration 10.5 *p* 133.

13. When a block that is not a subblock terminates, the ***value*** of the block is the value of its **`value`** variable, if any. Thus a one-statement block

    ```
    value = X + Y
    ```

which inherits the variables **X** and **Y** will return the sum of its inherited variables as the value of the block.

If a block that is not a subblock finishes without producing a value for a variable named **value**, the value of the block is **true**.

10.4 Default Statements

The **default** *statement-qualifier* modifies *statement-group* execution. A statement qualified by **default** is called a ***default statement***. Default statements in a *statement-group* are initially inactive, meaning that they are treated as if they do not exist,

If execution of a *statement-group* gets to the point where there is no possibility of any of the variables set by a default statement receiving a value from another statement, then the default statement is activated.

The '**default**' qualifier may only be applied to *assignment-statements*, and not to *guarded-statements*. A block variable may be a settable by at most one default statement in the block.

An example application of these rules is:

```
if x > 5:
    value = 'true'
default value = 'false'
```

If **x** gets the value **6**, then '**value**' is set to '**true**'. If **x** gets the value **4**, then '**value**' is set to '**false**'. If execution gets to the point where **x** cannot ever be given a value, then '**value**' is set to '**false**'.

The proof that a variable like **x** can never get a value is conservative. For example, in:

```
if x > 5:
    y = 10
if y > 5:
    x = 10
default x = 8
default y = 8
```

one can prove that **x** never gets a value only if one can prove that **y** never gets a value, and vice versa. As a consequence one can never prove either, and both default statements never activate.

10.5 Iteration

A block can ***iterate***, meaning that it generates another block that is the next block in a sequence of blocks. A block iterates if it produces a value for a block *next-variable*, which is a block variable whose name has the form '**`next`** V' for some ordinary variable name V.

The blocks in the sequence are called the ***iterations*** of the original *block*. Evaluation of a block can always be thought of as producing a sequence of iterations, though this sequence might include only one block which does not iterate.

The value of '**`next`** V' in a block iteration becomes the value of V in the block's next iteration.

A block that is going to iterate terminates only after <u>all</u> the block's *next-variables* have been given values. The next iteration of the block begins only after the previous iteration terminates.

Note that a *next-variable* output in a *permanent-subblock* is actually a block variable of the containing block, whereas a *next-variable* output in a *temporary-subblock* is a block variable of the subblock, and not its containing block, unless it is named as an *output-variable* of the *variable-assignment-subblock* statement containing the subblock. So *permanent-subblocks* can never iterate, though their containing blocks can, while *temporary-subblocks* can iterate.

A *next-variable* is named by a *variable-name* preceded by the '**`next`**' keyword. A *next-variable* is effectively a new variable with a name derived from the *variable-name*. We will used the notation '**`next`** V' to denote the *next-variable* made from the variable name V. The value of **`next`** V at the end of execution of the current block iteration becomes the value of V at the beginning of execution of the next block iteration. In order for there to be a next iteration, the current iteration must compute the value of some *next-variable*.

Next-variables can be both output and input variables. When used as input variables they are just like other input variables. The **`next`** unary operator can be applied to an expression that does not consist solely of a *variable-name*. When this is done, it is as if the **`next`** keyword had instead been prefaced to every *variable-name* in the expression. Thus '**`next(x+y)`**' is the equivalent of '**`next x + next y`**'.

If a default statement outputs a *next-variable*, the statement cannot become active unless some other *next-variable* has already been given a value. Thus the decision to iterate cannot be made inside a default statement.

If a *next-variable* is given the value of the expression '**`UNDEFINED`**', the variable will become undefined at the beginning of the next iteration. The expression '**`UNDEFINED`**' cannot be used to set a non-*next-variable*.

A **`first`** qualified *statement* is only visible in the first iteration of a block iteration sequence.

An **`always`** qualified *statement* is equivalent to the *statement* qualified by **`first`** plus additional *statements* of the form

`default next` V `=` V

for every *variable-name* V that can be assigned by the **`always`** qualified *statement*. Thus the **`always`** qualified *statement* will assign values on the first iteration, and these values will then be propagated to subsequent iterations.

11 Descriptions

A description is a typed block that can be used as a value. A description is an object with a type that inherits code from its type.

An ***description*** can be created by an *description-expression*:

> ***description-expression*** ::= { **`a`** | **`an`** } *type-name* { **`:`** *block* }$^?$
>
> ***type-name*** ::= *name* [see p33]

The *block* executes and the block variables it gives values to become ***components*** of the description. These components can be retrieved by expressions of the form:

> ***component-expression*** ::= **`the`** *variable-name* **`of`** *description*

where *variable-name* names a component of the *description*.

Description components can be updated by expressions such as:

> ***description-update-expression*** ::= **`for`** *description* **`:`** *block*

Here the *block* inherits existing components of the *description* and may add new components. The *block* in a *description-update-expression* may also define '***update variables***' that have names of the form '**`update`** v'. When the *block* terminates, if it has given a value to a block variable with such a name, the value becomes the new value of description component v. The one exception is the block variable '**`value`**', which if given a value becomes the value of the *description-update-expression*.

Code can be added to a description type by:

> ***type-update-statement*** ::= **`for every`** *type-name* **`:`** *block*

The block is added to the description type and any statement group in the block is immediately executed for every existing description of the type. Statement groups of a description type are also executed for a new description immediately after any *description-expression* block is executed.

Alternative syntax is usually used for creating and updating descriptions.

For creating descriptions, a '***constructor***' can be defined that is a function whose *expression-definition* begins with a *pattern-term* of the form:

{ **`a`** | **`an`** } *type-name* *bracketed-pattern**

When a constructor's *block* executes, it must not set the '**`value`**' variable. At the end of the block excution, a description of the given type will be created and returned as the value of the block. The variables of the block will be the components of the description. A big advantage of constructors over *description-expressions* is that constructors can have arguments that are inherited by the block that defines the description.

For updating descriptions, '***methods***' can be defined by syntax of the form:

`for every` *type-name* **`:`**
`on` *expression-definition*

The *expression-definition* must have the evaluated syntax argument **`THIS`** in its *expression-definition pattern*.

The *expression-definition* defines a function, but differs from the usual function definition as follows. First, the definition implicitly has the qualifier

`THIS is a` *type-name*

Second, when the function *block* executes, it behaves as if it were the block in a

`for` *type-name*: *block*

description-update-expression. It inherits the components of the **`THIS`** argument description, and can compute '**`update`** v' variables that update these components. A big advantage of methods over *description-update-expressions* is that methods can have arguments that are inherited by the block that updates the description.

11.1 Arrays

[TBD: Arrays are descriptions plus a vector; array element names are syntactic sugar.]

12 Side Effects

A ***side effect*** is an action that changes memory, inputs information from the outside world, or outputs information to the outside world. When an expression is evaluated, it may or may not have side effects.

The order in which side effects are executed is determined by *sequence-breaks* that divide a *block* into *groups* further (10.3 $^{p\,130}$). A programmer typically writes code so there is at most one side effect per *group*, in order to ensure side effects execute in the desired order.

TBD: could this last rule be enforced.

The ***side effect mode*** controls the execution of side effects. It has three settings: **`execute`**, **`delay`**, and **`error`**. In **`execute mode`** a side effect simply executes. In **`error mode`** an attempt to execute a side effect raises an error, and the side effect is not executed.

In **`delay mode`** input and memory change side effects execute, but put operations on an **`undo`** list that can undo their effects, while output side effects do not execute, but are instead put on a **`todo`** list.

There are two lists maintained that permit side effects to be delayed or undone. The **`todo`** list is a list of delayed output actions that have been delayed. The **`undo`** list is a list of input and memory change actions that can be undone. The position of these lists can be recorded and an undo operation can be performed that backs up to previously recorded positions by deleting actions from the end of the **`todo`** list and undoing actions on the end of the **`undo`** list.

To control the side effect mode there is a ***side effect mode stack***. This contains items each of which contains a side effect mode and positions in the **`todo`** and **`undo`** stacks. The side effect mode of the top item on the side effect mode stack is the effective side effect mode for current execution. Whenever an item is pushed to the side effect mode stack, the current positions of the **`todo`** and **`undo`** lists are recorded in the item.

The following statements operate on the side effect mode stack:

`begin executing side effects` Push a new item with side effect execute mode onto the side effect mode stack. Before the push the stack must be empty or the top item in the stack must have execute or error mode.

`end executing side effects` Pop the top item from the side effect mode stack. This top item must have execute mode.

`begin forbidding side effects` Push a new item with side effect error mode onto the side effect mode stack. Before the push the stack must be empty or the top item in the stack must have execute or error mode.

`end forbidding side effects` Pop the top item from the side effect mode stack. This top item must have error mode.

`delay side effects` Push a new item with side effect delay mode onto the side effect mode stack.

`commit side effects` Pop the top item from the side effect mode stack. This top item must have delay mode. It the resulting stack has a new top item that is not delay mode, discard the contents of the **`undo`** stack, and execute and then discard the contents of the **`todo`** stack.

`abort side effects` Pop the item from the the side effect mode stack. This top item must have delay mode. Consider the **`todo`** and **`undo`** list positions of the new item at the top of the stack, or take these positions to be the beginning of the lists if the stack has become empty. Remove elements from ends of the lists until these considered positions become the current list positions. When removing an element from the end of the **`undo`** list, perform the undo action specified by the element.

13 Debugging

Design:

Debugging is based on the notion that almost all RECKON programs will run quickly. Input checkpointing is used to record all inputs to a computation so the computation can be deterministically rerun. Detailed traces can be generated which explain for each value how it was generated. Values have a sequence number that identifies the point in the execution where they were generated. It is therefore possible to ask for a detailed accounting of how any value was generated, provided the run is short enough to be repeated once or a few times so the computer can turn the history tracing on appropriately.

14 Design Notes

These are some detailed design rules to be incorporated in the language definition later.

Blank Algebraic Operators. The blank, or missing operator, denotes `*` when it precedes a word, as in **`5x`** being equivalent to **`5*x`**. When it is between a preceding integer and a following ratio, it adds the ratio to the integer, as in **`41 1/3`**.

15 To Do

What about making a variable true if it has ANY value, for the purposes of a *when-statement*?

Describe matrix expression scanning that has no separators. Maybe it should not recognize operators outside parentheses?

Describe blank lines as paragraph operators in text parsing.

Should temporaries be full-fledged names.

Describe the quote evaluator.

Why doesn't '**`define qualifier xxx`**' mean define expressions equal to 'qualifier xxx'.

`missing(#X)` is true if **`#X`** is a null node. **`integer(#X)`** is true if **`#X`** is an integer.

Qualifier shortcuts should be conditioned on the first words of the expression being qualified. E.g., if these words are '**`define operator`**' the shortcut '**`left => with associativity [left]`**' would be defined.

Expressions to be evaluated can have sets of possible values.

Imaginary Units

Visible Side Effects

List Objects

Qualifier Definition

Qualifier Shortcut Definition

Automatic Optional Marks

Describe Garbage Collection

Manual Deletion, the stub is marked deleted, and gc's make pointers to the stub either NULL or point them at a standard deleted object so the deleted stub can be collected.

Index