

# Calculation and Simulation Total Language Environment

## CASTLE

(Draft 1a)

Robert L. Walton\*

July 2, 2005

---

\*Copyright 2004 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, and by the work of Thomas Cheatham and Stuart Shieber.

## Table of Contents

1	Introduction . . . . .	5
2	Remarks . . . . .	6
3	Overview . . . . .	6
4	Lexical Scans . . . . .	16
4.1	Pre-Lexemes . . . . .	20
4.1.1	Lexical Matching and Pre-Lexical Context . . . . .	21
4.1.2	Character Disambiguation Rules . . . . .	22
4.1.3	Pre-Lexeme Examples . . . . .	22
4.1.4	White Space Conversion . . . . .	23
4.2	Lexemes . . . . .	24
4.2.1	Number Lexemes . . . . .	25
4.2.1.1	Decimal Numbers . . . . .	25
4.2.1.2	Radix Numbers . . . . .	26
4.2.1.3	Ratios . . . . .	27
4.2.1.4	Scientific Numbers . . . . .	28
4.2.1.5	Unit Numbers . . . . .	28
4.2.2	Post Separators . . . . .	29
4.2.3	Indentation Lexemes . . . . .	30
4.3	Lexical Parsing . . . . .	33
4.3.1	Lexeme Replacement . . . . .	34
4.3.2	Number Unit Grouping . . . . .	36
4.3.3	Unit Multiplier Insertion . . . . .	37
5	Expression Parsing . . . . .	38
5.1	Subexpressions . . . . .	38
5.2	Expression Structure Overview . . . . .	39
5.3	Raw Expressions . . . . .	42
5.4	Operators . . . . .	44
5.4.1	Operator Definitions . . . . .	44
5.4.1.1	Operator Fixity . . . . .	44
5.4.1.2	Operator Name . . . . .	45
5.4.1.3	Operator Precedence . . . . .	45
5.4.1.4	Operator Associativity . . . . .	45

5.4.1.5	Operator Parser . . . . .	46
5.4.1.6	Operator Control Flags . . . . .	47
5.4.1.7	Operator Wrapper . . . . .	47
5.4.1.8	Operator Subdefinitions . . . . .	47
5.4.2	Operator Definition Syntax . . . . .	48
5.4.3	Operator Selection . . . . .	49
5.4.4	Post Operator Selection Processing . . . . .	53
5.4.5	Equivalence to Classical Parsing . . . . .	54
5.5	Qualifiers . . . . .	60
5.6	Text Parsing . . . . .	60
5.6.1	Section, Paragraph, and Sentence Parsing . . . . .	61
5.6.2	Text with Format Separators . . . . .	62
6	Expression Graphs . . . . .	62
6.1	Expression Graph Notation . . . . .	63
6.2	Pure Unification . . . . .	65
6.3	Raw Expression Trees . . . . .	67
6.4	Call Nodes and the Call Check . . . . .	71
6.5	Paths and Witnesses . . . . .	72
6.6	Call Unification . . . . .	74
6.7	Expression Graph Representation . . . . .	76
6.8	Expression Graph Implementation . . . . .	79
7	Expression Evaluation . . . . .	80
7.1	Expression Definitions . . . . .	82
7.2	The Evaluation Algorithm . . . . .	82
7.3	Contexts . . . . .	85
8	Blocks . . . . .	87
8.1	Block Syntax . . . . .	87
8.2	Block Variable Names . . . . .	89
8.3	Block Evaluation . . . . .	90
8.4	Default Statements . . . . .	92
8.5	Iteration . . . . .	93
9	Objects . . . . .	94
10	Side Effects . . . . .	97
11	Debugging . . . . .	98

12 Design Notes . . . . .	99
13 To Do . . . . .	99

# 1 Introduction

This document describes the programming language CASTLE. CASTLE, for what its worth, stands for ‘Calculation and Simulation Total Language Environment’, which hints at its purpose.

CASTLE is designed for naive programmers: that is, for people who may never be able to program computers well. It is a 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.
- 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. CASTLE tries to combine these. Some previous computer languages that have influenced CASTLE, and the data types they particularly support, are:

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

CASTLE is not 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

CASTLE was created as an answer to the question: what programming language should you teach beginning programming students who do not have the talent or inclination to become good 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 CASTLE design were developed by the author while teaching the intended customers of CASTLE.<sup>1</sup> 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.<sup>2</sup> 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.

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

## 3 Overview

CASTLE 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 an piece of code. Each variable can have a value, which is an expression. The code contains expressions that can be evaluated under appropriate circumstances to produce values for variables.

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

You can use CASTLE as a calculator by typing into it expressions to be evaluated, assign-

---

<sup>1</sup>Specifically, while teaching CS121 at Suffolk University using the C programming language.

<sup>2</sup>There was no problem teaching recursion, but it was better not to teach many different looping constructs.

ments of values to variables, and definitions of functions and predicates. Some examples involving numbers are:

```
> 9
9
> 9 + 8
17
> x = 9
9
> y = 9 + 8
17
> x + y
26
```

Here the ‘>’ at the beginning of some lines is the CASTLE *prompt* that tells you its OK to input an expression to be evaluated.

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

```
> g = 'hello'
'hello'
> '<<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.’
```

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

```
> f = {10x^2 - 3.67x - 0.04}
{10x2 - 3.67x - 0.04}
> h = (- 0.96 + 0.67x) in x
```

```

{-0.96 + 0.67x}
> (f + h) in x
{10x2 - 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)}
{∫ x2dx}
> 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 $\frac{1}{3}$ .'
> raw out
[-SENTENCE- the value of
      {(integral (x ^ 2 * dx)) from (x = 1) to (x = 5)}
      is 124/3 .]

```

[TBD: Can you overload {} as both math expression brackets and code brackets?]

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

In CASTLE an **object** is a block that has an associated **type**. For example, there may be an object named 'Jack' with type 'person'. All objects 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' 9"
+   age = 23 yr 2 mo }

```

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

```

> a person {
+   name = 'Jill'

```



```

+   weight = 110 lb
+   height = 5' 7"
+   age = 21 yr 8 mo }

```

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

```

> all persons

```

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

```

> the person Jack

```

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

```

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

```

ID	type	name	weight	height	age
@1000001	person	Jill	110 lb	5' 7"	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 objects 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' 9"	23 yr 2 mo	18.163738
@1000001	person	Jill	110 lb	5' 7"	21 yr 8 mo	17.228258

One can use definitions to define expressions that compute values:

```

> sum from X through Y <-- integer X, 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	X	Y	value
@1000002	5	10	45
@1000003	6	10	40
@1000004	7	10	34
@1000005	8	10	27
@1000006	9	10	19
@1000007	10	10	10
@1000008	11	10	0

```

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

```

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

```

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

CASTLE tends to ignore word endings: thus 'person' and 'persons' are to CASTLE the same word. CASTLE can even be told that 'person' and 'people' are the same word. 'Jack's', on the other hand, 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 and subexpressions. Subexpressions must be parenthe-

sized unless they consist of a single word, or 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' 7"	21 yr 8 mo
@1000000	person	Jack	123 lb	5' 9"	23 yr 2 mo

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

ID	type	name	weight	height	age
@1000000	person	Jack	123 lb	5' 9"	23 yr 2 mo
@1000001	person	Jill	110 lb	5' 7"	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 ‘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 an object. Such printed representations are chosen automatically from the set of all possible representations, which in this case include ‘the person weighing 123 lbs’ and ‘the person named Jack weighing 123 lbs’.

The ‘really raw’ form of a value identifies objects 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 CASTLE lists are sometimes called ‘*flat lists*’.

CASTLE has many different kinds of quotes or brackets. Some of these, {...}, ‘...’, and {{...}}, turn evaluation off, while <<...>>, <<<...>>>, <<<<...>>>>, etc. turn evaluation on. Some, ‘...’ and {{...}}, turn recognition of operators (e.g., + and =) off, while others, <<...>> and {...}, turn recognition of operators on. Quotes – ‘...’, ‘‘...’’, ‘‘‘...’’’, etc. – also do other things, like insert implicit operations (e.g., -SENTENCE-).



CASTLE 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
```

CASTLE 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’}
```

```

      (Jack)
> {( <<x>> right of <<y>>) above <<z>>}
      [ ] ( )
      (Jack)
> {row(<<x>>,<<y>>,<<z>>)}
      ( ) [ ] (Jack)
> p = {column (row(<<x>>,<<y>>,<<z>>), row(<<z>>,<<y>>,<<x>>))}
      ( ) [ ] (Jack)
      (Jack) [ ] ( )
> 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 the 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

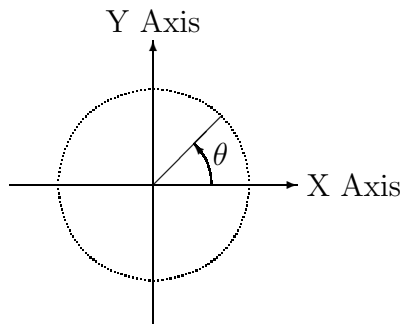

| ID       | type | angle      |
|----------|------|------------|
| @1000044 | demo | 40 degrees |


> 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 (0.7,0.0) to <<(0.3*cos X, 0.3*sin X)>>}
+   theta = {Greek th}
+   value = {column (
+     overlap (
+       <<c>>,
+       <<x-axis>> labeled 'X Axis',
+       <<y-axis>> labeled 'Y Axis',
+       <<line>>,
+       <<arc>> labeled '<<theta>>' ),
+     label 'Depiction of Angle <<theta>>' ) } }
> x

```

Depiction of Angle  $\theta$ 

```
> show x label {Greek th}
See  $\theta$ 
> update x to 40 degrees
See  $\theta$ 
> increment x by -5 degrees
See  $\theta$ 
```

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

Next we define how to display a **demo** object. Then we use the ‘show x label {Greek th}’ command to cause the **demo** object value of x to be displayed in a separate window labeled ‘ $\theta$ ’. Every time this **demo** object changes, the  $\theta$  window is updated, and every time the object is to be printed, ‘See  $\theta$ ’ is printed instead.

TBD: example of a simple game.

The rest of this document is a reference manual for CASTLE.

## 4 Lexical Scans

A CASTLE program is a sequence of characters which is scanned from left to right to produce a sequence of *pre-lexemes*. The sequence of pre-lexemes is then scanned from left to right to produce a sequence of *lexemes*. The sequence of lexemes may then be subject to a left to right *lexical parsing* scan to produce a final set of lexemes that is input for expression parsing. Lexical parsing is suppressed in certain contexts, e.g., within quotes.

For example, the input ‘x = 7’ 5.5”’ (here the “ quotes are not part of the input) contains



```
for every board:
  on a board of dimension S <--:
    'Make a board of size SxS.'
    size = S
    'Allowed vessels have lengths 2 (destroyer), 3 (cruiser),
      5 (battleship). vessels(L) is number of length L.'
    array vessels of size 5 with initial element 0
    'maximum-vessels(L) is maximum number of vessels of
      length L.'
    array maximum-vessels of size 5 with initial element 0
    maximum-vessels(2) = 5
    maximum-vessels(3) = 2
    maximum-vessels(5) = 1
    'state(I,J) is 'none', 'miss', or 'hit' iff shell has not
      struck square (I,J), struck square (I,J) but that square
      had no ship, or struck square (I,J) and hit a ship at
      that square.'
    array hit of size (S,S) with initial element 'none'
    'vessel(I,J) is the vessel at (I,J)'
    array vessel of size (S,S) with initial element 'none'
```

Figure 1: Code for the Battleship Game, Part I

```
direction vector of D <--:
  'given a direction N, NE, E, SE, S, SW, W, NW, return a vector
  with unit components in the given direction.'
  if D == 'N':
    value = (1,0)
  else if D == 'NE':
    value = (1,1)
  else if D == 'E':
    value = (1,0)
  else if D == 'SE':
    value = (1,-1)
  else if D == 'S':
    value = (0,-1)
  else if D == 'SW':
    value = (-1,-1)
  else if D == 'W':
    value = (-1,0)
  else if D == 'NW':
    value = (-1,1)
```

Figure 2: Code for the Battleship Game, Part II

```
for every vessel:
  on a vessel of length L with direction D from (I,J) on B
    'make a vessel of length L positioned in direction D
      from origin (I,J) on boards B; directions are N,
      NE, E, SE, S, SW, W, NW'
    length = L
    direction = D
    origin = (I,J)
    vector = direction vector of D
    destination = L * vector + origin
    board = B
    conflict =:
      first p = origin
      first k = 0
      next k = k + 1
      next p = p + vector
      TBD
    if L < 2 or L > 5:
      value = error 'bad length <<L>>'
    else if maximum-vessels(L) of B >= vessels(L) of B:
      value = error 'two many vessels of length <<L>>
        on board <<B>>'
    else:
      vessels(L) of B += 1
```

Figure 3: Code for the Battleship Game, Part III

the pre-lexeme '5.5";', which is split into 3 lexemes '5.5 " ;', and then during lexical parsing '\*' operators are inserted before or after units, and '+' operators are inserted between consecutive numbers with different units, and parentheses are inserted around sequences of consecutive numbers with different units, to produce from the input the following string of 12 lexemes:

x = ( 7 \* ' + 5.5 \* " ) ;

## 4.1 Pre-Lexemes

*Pre-lexemes* are defined as follows:

```

pre-lexeme ::= pre-word
                | opening-mark
                | closing-mark
                | format-separator
                | white-space

pre-word ::= word-character word-character*

word-character ::= letter | digit
                  | + | - | * | / | \ | ~ | @ | # | $ | % | ^ | & | = | | < | > | _ | " | ' | | ! | ? | ; | : | , | .

letter ::= lower-case-letter | upper-case-letter

lower-case-letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

upper-case-letter ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

opening-mark ::= ( | [ | { | opening-quote | opening-angle

closing-mark ::= ) | ] | } | closing-quote | closing-angle

opening-quote ::= opening-quote-character opening-quote-character*

opening-quote-character ::= '

closing-quote ::= closing-quote-character closing-quote-character*

closing-quote-character ::= '

quote ::= opening-quote | closing-quote

opening-angle ::= opening-angle-character opening-angle-character
                  opening-angle-character*

```

*opening-angle-character* ::= <  
*closing-angle* ::= *closing-angle-character* *closing-angle-character*  
*closing-angle-character*<sup>\*</sup>  
*closing-angle-character* ::= >  
*angle-character* ::= < | >  
*opening-character* ::= ( | [ | { | ' | <  
*closing-character* ::= ) | ] | } | ' | >  
*format-separator* ::= *format-separator-character* *format-separator-character*<sup>\*</sup>  
*format-separator-character* ::= |  
*white-space* ::= *white-space-character* *white-space-character*<sup>\*</sup>  
*white-space-character* ::= *horizontal-space-character*  
| *vertical-space-character*  
*horizontal-space-character* ::= *space* | *horizontal-tab* | *carriage-return*  
*vertical-space-character* ::= *line-feed* | *vertical-tab* | *form-feed*

The following sections give rules involving pre-lexemes.

#### 4.1.1 Lexical Matching and Pre-Lexical Context

*Opening-marks* and *closing-marks* are both pre-lexemes and also lexemes. Rules for matching pre-lexemes in a pre-lexeme sequence are the same as rules for matching lexemes in a lexeme sequence. Here we will state the rules for lexemes, and leave it to the reader to reformulate them for pre-lexemes.

**Lexeme Matching Rule.** An *opening-mark* with  $N$  characters  $C$  must have a matching *closing-mark* with  $N$  characters each the mirror of  $C$ . Here the mirror of ' is ', the mirror of { is }, the mirror of [ is ], the mirror of ( is ), and the mirror of < is >. Each *closing-mark* must match exactly one *opening-mark*, each *opening-mark* must match exactly one *closing-mark*, and an *opening-mark* must precede its matching *closing-mark*.

Two lexemes are said to be **matched** if and only if they are matched opening and closing marks.

**Matched Lexeme Nesting Rule.** If one lexeme in a pair  $P_2$  of matched lexemes is in between the lexemes of another pair  $P_1$  of matched lexemes, then both lexemes in  $P_2$  must be in between the lexemes of  $P_1$ . In this case  $P_2$  is said to be **nested** inside of  $P_1$ .

During the scan a character  $C$  is said to be in the **lexical context** of a pair  $P$  of matched lexemes if and only if  $C$  is between the matched lexemes of  $P$ , and  $C$  is not between any other pair of matched lexemes that is nested inside of  $P$ . For example, in  $\{ \mathbf{x} \ [ \ \mathbf{y} \ ] \ \mathbf{z} \}$ ,  $\{ \}$  are in the outermost lexical context,  $\mathbf{x} \ [ \ ] \ \mathbf{z}$  are in the middle lexical context whose matched lexemes are  $\{ \}$ , and  $\mathbf{y}$  is in the innermost lexical context whose matched lexemes are  $[ \ ]$ .

For a sequence of pre-lexemes, **matched** pre-lexemes and **pre-lexical context** are defined as for a sequence of lexemes.

#### 4.1.2 Character Disambiguation Rules

Several characters in pre-lexemes are ambiguous in the pre-lexeme syntax equations. The following rules disambiguate these characters.

**Opening Quote Rule.** An **opening-quote-character** must be preceded by a *white-space-character*, an *opening-character*, or a *format-separator-character*. Otherwise it is a *word-character*.

**Closing Quote Rule.** A **closing-quote-character** must be part of a sequence of *closing-quote-characters* that is of exactly the right length to be the matching pre-lexeme for the last previous unmatched *opening-mark*, which must be an *opening-quote*. Otherwise the potential *closing-quote-character* is a *word-character*.

**Format Separator Rule.** A *format-separator-character* must be in the pre-lexical context of a pair of matched *quotes*. Otherwise it is a *word-character*.

**Angle Rule.** An *angle-character* must be either preceded by or followed by a copy of itself. Otherwise it is a *word-character*.

#### 4.1.3 Pre-Lexeme Examples

Quote Examples:

Input String	Pre-Lexeme Sequence with <i>spaces</i> represented by _'s
I said 'Hello'.	I _ said _ ' Hello ' _ .
Re'op 'tis. But!	Re'op _ 'tis. _ _ But!
'Like 'tis'.	' Like _ ' tis'.
''Like 'tis''.	'' Like _ 'tis '' _ .
'Like me''.	' Like _ me''.
'' 'Hello' is a word.'''	'' _ ' Hello ' _ is _ a _ word. ''

Other rules of CASTLE limit the semantic content of *white-space*. In particular, there is no problem putting space between the '' and ' in the last example (5.6.1 <sup>p61</sup>).

Angle and Format Separator Examples:

Input String	Pre-Lexeme Sequence with <i>spaces</i> represented by _'s
x <= y	x _ <= _ y
x <<= y	x _ << = _ y      Note << is unmatched <i>opening-mark</i> .
x = y z	x _ = _ y z
x = 'y z'	x _ = _ ' y   z '

#### 4.1.4 White Space Conversion

A *white-space* pre-lexeme does not have exactly the same characters that were input to create it, unlike other pre-lexemes. The sequence of *white-space-characters* input to create a *white-space* pre-lexeme is modified as follows to create the pre-lexeme:

**Line End Spaces Rule.** All *horizontal-space* characters preceding a *vertical-space* character are deleted. Thus spaces at line ends are ignored.

**Carriage Return Rule.** If the pre-lexeme contains a *carriage-return* but no *vertical-space-character*, it is in error. Each *carriage-return* and all *horizontal-spaces* preceding it are deleted. Thus the pre-lexeme has no *carriage-returns*.

**Horizontal Tab Rule.** Each *horizontal-tab* is replaced by *spaces* assuming that horizontal tab stops are set every 8 columns after applying the Carriage Return Rule (p23). Thus the pre-lexeme has no *horizontal-tabs*.

Note that these rules do not alter the printed appearance of the *white-space* pre-lexeme, assuming that each *vertical-space-character* causes printing to return to the beginning of the line after the vertical space is executed. The input may or may not contain *carriage-returns*

immediately before or after *vertical-space-characters* with no effect.

After these rules are applied, a *white-space* pre-lexeme consists of zero or more *vertical-space-characters* followed by zero or more *space* characters. It is not possible to have an empty *white-space* pre-lexeme; the pre-lexeme either has a *vertical-space-character*, which none of the above rules can delete, or it has a *space* character which cannot be deleted, because, in the absence of a *vertical-space-character*, the input to the pre-lexeme cannot have a *carriage-return*, and therefore *horizontal-space-characters* cannot be deleted by the rules.

*White-space* pre-lexemes in the pre-lexical context of a pair of matched *quotes* become lexemes. *White-space* pre-lexemes that are not in the pre-lexical context of a pair of matched *quotes* are discarded, and do not become lexemes, but in certain pre-lexical contexts they may be used to create indentation lexemes (4.2.3<sup>p30</sup>).

The rules for forming lexemes from pre-lexemes ensure that all non-empty sequences of *vertical-space* characters are equivalent outside the pre-lexical context of a pair of matched *quotes*.

Inside matched *quotes* that have no *format-separator* the only distinction made between non-empty sequences of *vertical-space* characters concerns whether or not they represent blank lines. A *white-space* lexeme whose only *vertical-space* character is a single *line-feed* represents just the end of a non-blank line, with no following blank lines. A *white-space* lexeme with some other sequence of *vertical-space* characters represents the end of a non-blank line that is followed by one or more blank lines. Thus all sequences of *vertical-space* characters that consist of other than just a single *line-feed* are equivalent, inside matched quotes without a *format-separator*.

Inside matched *quotes* *white-space* lexemes after a *format-separator* are retained literally.

## 4.2 Lexemes

The sequence of pre-lexemes is converted to a sequence of **lexemes** according to the following rules, which we will describe in order:

- Numbers
- Post Separators
- Indentation Lexemes

The syntax equations defining a lexeme are:

$$\mathbf{lexeme} ::= \mathit{word} \mid \mathit{separator} \mid \mathit{opening-mark} \mid \mathit{closing-mark} \mid \mathit{white-space}$$



$$\textit{separator} ::= \textit{format-separator} \mid \textit{post-separator}$$
$$indentation\text{-}lexeme ::= ; \mid \{ \mid \}$$

*Post-separators* are single characters removed from the ends of *pre-words*. A *word* is what is left of a *pre-word* after any *post-separators* are removed from its end. *Words* and *post-separators* are defined below in 4.2.2<sup>p29</sup>.

The *indentation-lexemes* are lexemes implied by indentation, and are not distinguishable from explicit lexemes. They are defined below in 4.2.3<sup>p30</sup>.

*White-space* pre-lexemes become lexemes if they appear in the pre-lexical context of a pair of matched *quotes*. All other *white-space* pre-lexemes are discarded when pre-lexemes are scanned to lexemes.

*Format-separators*, *opening-marks*, and *closing-marks* are both pre-lexemes and lexemes.

### 4.2.1 Number Lexemes

The rule for splitting a *pre-word* into a *word* and a *separator* makes reference to *words* that are *numbers*, saying that splitting is preferred if the *word* resulting from the split is a *number*. Some of the syntax equations defining ***numbers*** are as follows.

$$number ::= real-number \mid unit-number$$
$$\textit{real-number} ::= \textit{decimal-number} \mid \textit{radix-number} \mid \textit{ratio} \mid \textit{scientific-number}$$

**4.2.1.1 Decimal Numbers.** *Decimal numbers* are sequences of digits with optional commas, an optional decimal point, and an optional sign.

$$\textit{decimal-number} ::= \textit{unsigned-decimal-number} \mid \textit{sign unsigned-decimal-number}$$
$$sign ::= + \mid -$$
$$\begin{array}{lcl} \textit{unsigned-decimal-number} & ::= & \textit{decimal-natural} \\ & | & \textit{decimal-natural} . \textit{decimal-natural} \\ & | & . \textit{decimal-natural} \end{array}$$
$$\textit{decimal-natural} ::= \textit{decimal-digits} \mid \textit{decimal-natural} , \textit{decimal-digits}$$
$$decimal-digits ::= digit\ digit^*$$

Decimal Number Examples:

123	-123	+123	1,234	-1,234,567
123.0	-.123	+0.0	1,234.987654	1,234.987,654
1,2	-1.86,54	1,234567.89	12345.678,9	+1.234567,892

(Lexemes in the last line give errors when converted to numbers.)

In a *decimal-number* lexeme, the decimal point must be followed by a digit, and commas must be surrounded by digits.

In addition, commas must be located every 3 digits from the decimal point, or every 3 digits from the right end if there is no decimal point. If there are any commas at all, there must be commas every 3 digits in the integer part, while the fraction part may be comma free, or may contain commas every 3 digits. Failure to follow the rules of this paragraph will result in an error when the lexeme is converted to a number, but is not an error of lexeme formation. The last line of examples above are therefore legal lexemes that will give errors when converted to numbers.

**4.2.1.2 Radix Numbers.** *Radix-numbers* permit binary, octal, or hexadecimal radices to be used instead of decimal. In fact, other number representation schemes are permitted.

***radix-number*** ::= *unsigned-radix-number* | *sign unsigned-radix-number*

***unsigned-radix-number***

::= *radix-indicator* # *radix-number-mark* *radix-number-mark*\* #

***radix-indicator*** ::= *letter letter-or-digit*\*

***letter-or-digit*** ::= *letter* | *digit*

***radix-number-mark*** ::= *word-character* except { # | | < | > | ' | ' }

Radix Number Examples:

B#10110100#	O#77534201#	D#19758#	X#FE8A932B#
B#101101#	O#0.7753#	D#197.58#	X#0.fe8a932b#
B#10,1101#	O#12,3456#	D#0.123,5#	X#FE8A,932B.7CCD,83#
B#a5,7b63#	O#12/3456#	D#+0.1#	X#F.E8A932B.7CCD#

(Lexemes in the last line give errors when converted to numbers.)

The following *radix-indicators* are standard.

Radix Name	Radix Indicators	Allowed Digits	Allowed Digits Between Commas
binary	b B	0 1	4 or 8
octal	o O	0 1 2 3 4 5 6 7	3
decimal	d D	0 1 2 3 4 5 6 7 8 9	3
hexadecimal	x X	0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F	4 or 8

Standard *radix-indicators* permit a single decimal point, which must be followed by a digit. Standard *radix-indicator* rules for comma location are the same as for *decimal-numbers* (4.2.1.1<sup>p25</sup>), except the number of digits between commas may be 4 or 8 instead of 3.

*Radix-numbers* can be legal lexemes and still be unconvertible to numbers because their *radix-indicators* are undefined, they have *radix-number-marks* (e.g., digits) not defined for the given *radix-indicator*, they have too many decimal points, and so forth. These errors are detected when the lexemes are converted to numbers. The last line of *radix-number* examples above are legal lexemes that will give errors when converted to numbers. Note that standard radix indicators only allow commas, a decimal point, digits, and letters that are hexadecimal digits as *radix-number-marks*.

On the other hand, it is possible to define non-standard converters for converting *radix-number* lexemes to numbers, and thereby increase the set of legal number representations.

**4.2.1.3 Ratios.** A *ratio* consists of two strings of decimal digits, the numerator and denominator, separated by a slash ('/'), with an optional sign.

***ratio*** ::= *unsigned-ratio* | *sign unsigned-ratio*

***unsigned-ratio*** ::= *numerator* / *denominator*

***numerator*** ::= *decimal-natural*

***denominator*** ::= *decimal-natural*

Ratio Examples:

1/2      -3/4      +1,234/5      1,234/5,432      -53/000

(The last lexeme gives an error when converted to a number.)

If the denominator equals zero, the ratio is still a legal lexeme, but will cause an error when it is interpreted as a number. -53/000 is an example.

**4.2.1.4 Scientific Numbers.** A *scientific-number* is a *decimal-number* or a *radix-number* followed by an exponent.

$$\begin{aligned} \textit{scientific-number} &::= \textit{decimal-number exponent} \\ &\quad | \textit{radix-number exponent} \\ \textit{exponent} &::= \textit{exponent-indicator exponent-sign-and-digits} \\ \textit{exponent-indicator} &::= \textit{e} \mid \textit{E} \\ \textit{exponent-sign-and-digits} &::= \textit{decimal-digits} \mid \textit{sign decimal-digits} \end{aligned}$$

Scientific Number Examples:

123e0	-123e+2	+123e-321	1,234e9
123E0	-123E+2	+123E-321	-0.123,456e-3
.123e0	-.123e+2	-0.123,456e-3	-1,234.567890e6
X#a9#e0	B#1011#e-3	O#0.7753#e-5	X#0.FE8A,932B,E#e+5

Note that exponents cannot contain commas.

The digits in an exponent are always decimal, even when the number has a radix indicator such as B or X that indicates a different radix. However, the interpretation of the exponent depends upon the radix indicator. Standard radix indicators interpret the exponent as multiplication by their radix to the exponent power. Thus B#1101#e-2 and B#11.01# are both equal to 3.25.

**4.2.1.5 Unit Numbers.** *Unit-numbers* are just *decimal-numbers* with a *unit-indicator* prefixed or postfixed. A *sign* may be before or after a prefix *unit-indicator*.

$$\begin{aligned} \textit{unit-number} &::= \textit{pre-unit-indicator unit-base-number} \\ &\quad | \textit{sign pre-unit-indicator unsigned-unit-base-number} \\ &\quad | \textit{unit-base-number post-unit-indicator} \\ \textit{unit-base-number} &::= \textit{decimal-number} \\ \textit{unsigned-unit-base-number} &::= \textit{unsigned-decimal-number} \\ \textit{pre-unit-indicator} &::= \$ \mid \pounds \\ \textit{post-unit-indicator} &::= ' \mid " \mid \% \mid ^\circ \mid \textit{i} \mid \textit{j} \mid \textit{k} \\ \textit{unit-indicator} &::= \textit{pre-unit-indicator} \mid \textit{post-unit-indicator} \end{aligned}$$

However, unit numbers are not lexemes.

**Unit Number Rule.** If a *unit-number* is to be output as a *word* lexeme, then instead the *unit-number* is split into two *word* lexemes, one of which is the *real-number* part and the other of which is a 1-character *word* consisting of the *unit-indicator*.

Unit Number Examples:

Unit Number	Lexeme Sequence
\$5.71	\$ 5.71
£-5.71	£ -5.71
-\$2.20	\$ -2.20
15'	15 '
-2.543"	-2.543 "
72%	72 %
0.5i	72 i

Here 15' splits to produce ' as a 1-character word lexeme, and not as a closing quote lexeme.

A sign immediately before a *pre-unit-indicator* is moved to the *real-number* part; -\$2.20 becomes \$ -2.20. The units 'i', 'j', and 'k' are used to specify complex imaginary or quaternion numbers: see p99.

Note that *radix-numbers*, *ratios*, and *scientific-numbers* cannot have units as part of their pre-lexeme. This is to reduce confusion: the units can always be given separated by a space from the number, as in '£ 0#7601#', '\$ 3/4', and '5e3 i'.

#### 4.2.2 Post Separators

Informally, a **post-separator** is a 1-character separator that immediately follows a word without intervening space. Examples are the comma, period, and the exclamation point. A word and any following post-separators are all part of a single pre-word. For example, 'Hello!' is a pre-word containing the word 'Hello' and the post-separator '!'.

There are two kinds of post separators: **weak** and **strong**. The Post Separator Rule given below tells when a *pre-word* ending in a *post-separator-character* must be split into a smaller *pre-word* and a 1-character *separator*. This rule makes reference to *words* that are *numbers*, saying that splitting is preferred when the *post-separator-character* is weak if the *pre-word* resulting from the split is a *number*.

The syntax equations required are:

**separator** ::= *format-separator* | *post-separator*

**post-separator** ::= *post-separator-character*

$$\textit{post-separator-character} ::= \textit{strong-post-separator-character} \mid \textit{weak-post-separator-character}$$
$$\textit{strong-post-separator-character} ::= , \mid ;$$
$$\textit{weak-post-separator-character} ::= ! \mid ? \mid : \mid .$$
$$\textit{pre-word} ::= \textit{word} \mid \textit{pre-word post-separator-character}$$

**Post Separator Rule.** A *pre-word* of 2 or more characters that ends with a *post-separator-character* is split into a smaller *pre-word* and a *post-separator* if (1) the *post-separator-character* is strong, or if (2) the smaller *pre-word* ends with a *strong-post-separator-character*, or if (3) the smaller *pre-word* is a *number*, or if (4) the smaller *pre-word* does not contain a copy of the *post-separator-character*. If the *pre-word* is not split, it becomes a *word* lexeme.

The following rule handles the case of 1-character *pre-words*.

**Isolated Post Separator Rule.** A *pre-word* that consists of a single *post-separator-character* is a *post-separator* lexeme.

Examples:

Pre-Word	Lexeme Sequence	Splits?
hello.	hello .	yes
X,	X ,	yes
h.e.l.l.o.	h.e.l.l.o.	no
e.g.	e.g.	no
5.0.	5.0 .	yes
1,234.5,	1,234.5 ,	yes
1,234.5,,	1,234.5 , ,	yes, twice
1,234.5.,	1,234.5 . ,	yes, twice
1,234.5.,.	1,234.5 , .	yes, twice
help!	help !	yes
!help!	!help!	no
help.!	help . !	yes, twice
.help.!	.help. !	yes, once
!.help.!	!.help.!	no

### 4.2.3 Indentation Lexemes

*Indentation-lexemes* are implied by indentation. The *indentation* of a line is the the number of *space* characters in the *white-space* pre-lexeme just before the first non-*white-space* pre-

lexeme of the line. Note that lines cannot be empty; empty lines are merged into *white-space* pre-lexemes. Recall that *white-space* pre-lexemes each consist of zero or more *vertical-space-characters* followed by zero or more *space* characters (p24).

At any given point in the scan converting pre-lexemes to lexemes, there is a stack of indentation records, called the ***indentation stack***. An ***indentation record*** is a number of columns, a pre-lexical context (p22), and a flag. The number of columns is called the ***indentation***. The flag is called the ***implicit bracket flag***, and indicates whether or not an implied { lexeme was inserted at the same time the indentation record was pushed onto the indentation stack.

Initially the indentation stack contains a single indentation record with 0 indentation, the outermost pre-lexical context, and an off implicit bracket flag. The stack cannot become empty; any operation that would pop the last indentation off the stack announces an error and leaves the stack alone. Thus the bottommost indentation on the stack is always the same.

The indentation in the indentation record at the top of the indentation stack is called the ***current indentation***, and the pre-lexical context in that record is called the ***current indentation context***.

The rules for creating indentation records are such that the pre-lexical contexts in such records are either pre-lexical contexts of some { } bracket pair, or are the outermost pre-lexical context, which therefore behaves like the context of a { } bracket pair as far as indentation is concerned. A line beginning is said to be in a particular pre-lexical context if the first lexeme on the line, were it not a *closing-mark*, would be in that pre-lexical context. One may think of a ***line beginning*** as being a position on a line just before the first pre-lexeme on the line.

**Semi-Colon Rule.** If the beginning of the line is in the current indentation context, and if the indentation of the line is the current indentation, then a ; implied *separator* lexeme is output before any lexemes generated by the line are output, unless (1) the last lexeme output before the line was a ; *separator* lexeme, (2) the last lexeme output was a { *opening-mark* lexeme (including an implied *opening-mark* as in the Implicit-Bracket Rule below), or (3) the first non-*white-space* pre-lexeme of a line is }.

**Explicit-Bracket Rule.** If the last pre-lexeme of a line is {, then an indentation record is pushed onto the indentation stack just after the { lexeme is output. The indentation in the record is the indentation of the next line, the pre-lexical context is that in effect just after the {, and the implicit bracket flag is off. The pushed indentation record is popped just after its pre-lexical context ends; that is, just before reading the } pre-lexeme that matches the {

pre-lexeme that pushed the indentation record.

**Implicit-Bracket Rule.** If the last lexeme output for a line would be a separator `:` in the current indentation context, then a `{` implied *opening-mark* lexeme is output instead of the `:`, and an indentation record is pushed onto the indentation stack. The indentation of the record is the indentation of the next line, the implicit bracket flag of the record is on, and the pre-lexical context of the record is that of the `:`. The pushed record is popped (1) just before the first line such that the line beginning is in the record's pre-lexical context and the line's indentation is less than the record's indentation, or (2) at the end of the input pre-lexeme stream. When the record is popped, a `}` implied *closing-mark* lexeme is output. It is an error if this implied *closing-mark* does not match the implied *opening-mark* associated with the popped record, because of intervening explicit opening and closing marks.

In order to avoid subtle errors created by indentation, there is a **minimum-indentation** parameter and the following rule.

**Minimum-Indentation Rule.** The indentation of any line whose beginning is in the current indentation context (the first pre-lexeme of the line would be in the current indentation context if it did not end a sub-context) must equal the current indentation or differ from it by at least the value of the **minimum-indentation** parameter. The **minimum-indentation** parameter defaults to 4.

Indentation Lexeme Examples:

Input String	Output Lexemes
hi { x; y z; w }	hi { x ; y z ; w }
hi { x y z w }	hi { x ; y z ; w }
hi { x y z w }	hi ; { x ; y z ; w }



<pre> hi:     x     y z     w </pre>	<pre> hi {     x     ; y z     ; w } </pre>
<pre> hi {     x     y 'this is another pre-lexical context. ' foo bar     w } </pre>	<pre> hi {     x     ; y ' this is another pre-lexical context . ' foo bar     ; w } </pre>
<pre> hi {     x     { ho hum } y     z } </pre>	<pre> hi {     x     ; { ho     ; hum } y     ; z } </pre>

### 4.3 Lexical Parsing

After pre-lexemes are scanned to produce lexemes, lexemes may be scanned to perform the following:

Lexeme Replacement (Spelling Regularization, Possessive Splitting) (4.3.1 <sup>p34</sup>)

Number Unit Grouping (4.3.2 <sup>p36</sup>)

Unit Multiplier Insertion (4.3.3 <sup>p37</sup>)

We describe these below in order.

The scan that performs the above actions is called ***lexical parsing***, because it does initial parse steps at the level of individual lexemes and short sequences of lexemes, and because it is controlled by definitions in the parsing stack (p40).

Lexical parsing is not performed in certain contexts, such as in the lexical context of a pair of matched *quotes* (see the table on p40).

### 4.3.1 Lexeme Replacement

The lexeme replacement process replaces one lexeme by a string of zero or more lexemes. Replacement may be done by either a dictionary or a function. Replacement is controlled by *lexeme replacement definitions* in the parsing stack (p40).

There are two kinds of lexeme replacement definitions: dictionary and function. These have the syntax:

```

LEXEME-DEFINITION3 ::= LEXEME-DICTIONARY-DEFINITION
                        | LEXEME-FUNCTION-DEFINITION

LEXEME-DICTIONARY-DEFINITION
  ::= define lexeme dictionary DICTIONARY-NAME
                        LEXEME-DICTIONARY-ENTRIES

DICTIONARY-NAME ::= [ { word | separator } { word | separator } * ]

LEXEME-DICTIONARY-ENTRIES ::= ( lexeme-dictionary-entry-list )
lexeme-dictionary-entry-list
  ::= LEXEME-DICTIONARY-ENTRY
      | lexeme-dictionary-entry-list , LEXEME-DICTIONARY-ENTRY

LEXEME-DICTIONARY-ENTRY
  ::= [ replaced-lexeme ==> replacing-lexemes ]

replaced-lexeme ::= { word | separator }

replacing-lexemes ::= { word | separator } *

LEXEME-FUNCTION-DEFINITION
  ::= define lexeme function FUNCTION-NAME

FUNCTION-NAME ::= [ { word | separator } { word | separator } * ]

```

which correspond to the expression definitions (7.1<sup>p82</sup>):<sup>4</sup>

```

define lexeme dictionary NAME ENTRIES <-- list ENTRIES

define lexeme function NAME <--

```

A *lexeme dictionary entry* gives a lexeme that is to be replaced and a sequence of lexemes that replace it. For example, the entry

---

<sup>3</sup>Capitalized syntactic categories represent CASTLE expressions (5<sup>p38</sup>).

<sup>4</sup>These particular dictionary entries tell us that NAME and ENTIREs are evaluated arguments (7<sup>p80</sup>).

```
[ people ==> person ]
```

causes the word ‘people’ to be replaced by the word ‘person’.

A *lexeme function* is a function that is called with a lexeme as its single argument and which returns either ‘false’ if the lexeme is not to be replaced or returns an expression consisting of a sequence of lexemes that are to replace the lexeme otherwise. For example, given the lexeme function definition

```
define lexeme function [replace people]
```

and the expression definition

```
replace people X <-- {
  if X == [people]:
    value = [person]
  else:
    value = false }
```

an appearance of the lexeme ‘fie’ will evaluate the expression ‘replace people fie’ which will return ‘false’ to avoid replacing the lexeme ‘fie’.

It is possible to temporarily void lexeme replacement definitions by placing canceling undefinitions in the parsing stack. These have the syntax:

```
LEXEME-UNDEFINITION ::= LEXEME-DICTIONARY-UNDEFINITION
                        | LEXEME-FUNCTION-UNDEFINITION
LEXEME-DICTIONARY-UNDEFINITION
  ::= undefine lexeme dictionary DICTIONARY-NAME
LEXEME-FUNCTION-UNDEFINITION
  ::= undefine lexeme function FUNCTION-NAME
```

which correspond to the expression definitions (7.1<sup>p82</sup>):

```
undefine lexeme dictionary NAME <--

undefine lexeme function NAME <--
```

Lexeme replacement is not recursive: the replacement lexemes are not themselves subject to replacement.

The following lexeme dictionaries and functions are defined in the initial parsing stack. Their definitions are ordered in the stack so the first given below is at the top of the stack and is the first that replaces lexemes.

**define lexeme dictionary [english lexeme dictionary]** This dictionary translates common English irregular plurals, to their singular form, and decomposes irregular possessives to their singular decomposed form. This dictionary also protects irregular singular forms that might be mistaken for regular plural forms (e.g., ‘news’). Some example translations are:

people → [person]  
 women → [woman]  
 geese → [goose]  
 fungi → [fungus]  
 news → [news]

**define lexeme function [english lexeme function]** This function translates common English standard plurals to their singular form, and decomposes standard possessives to their singular decomposed form. Some example translations are:

boys → [boy]  
 boy’s → [boy ’s]  
 boys’ → [boy ’s]  
 boxes → [box]

Because CASTLE, outside the context of quotes, insists on mapping different forms of a word to a single word, some subtleties of language are lost. For example, ‘people’ can be a singular word referring to a group of people, but CASTLE will standardly confuse it with ‘person’.

#### 4.3.2 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* (number without *unit-indicator*, p25), and a *unit-specifier*. The syntax equations are:

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

In order to be recognized as a *unit-specifier* a *word* sequence must be defined as a *UNIT-SPECIFIER-NAME* by the following:

**UNIT-SPECIFIER-DEFINITION**

```
::= define unit specifier  UNIT-SPECIFIER-FIXITY
                           UNIT-SPECIFIER-NAME
```

**UNIT-SPECIFIER-FIXITY** [ prefix ] | [ postfix ]

**UNIT-SPECIFIER-NAME** [ word word\* ]

It is possible to temporarily void unit specifier definitions by placing canceling undefinitions in the parsing stack. These have the syntax:

**UNIT-SPECIFIER-UNDEFINITION**

```
::= undefine unit specifier UNIT-SPECIFIER-NAME
```

The *number unit grouping* phase of lexical parsing recognizes *number-unit-groups* containing more than one *number-unit-pair*, places implied parentheses around them, and places implied ‘+’ operators between their *number-unit-pairs*. This is done after lexeme replacement (4.3.1<sup>p34</sup>).

For example, ‘7 ’ 30 ”’ is a *number-unit-group* containing 2 *number-unit-pairs* that number unit grouping transforms into ‘( 7 ’ + 30 ” )’.

### 4.3.3 Unit Multiplier Insertion

The *unit multiplier insertion* phase of lexical 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 (4.3.2<sup>p36</sup>).

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

```
some function ( 3 ft / sec, 9 ft ^ 2 lb / sec )
```

becomes

```
some function ( 3 * ft / sec, 9 * ft ^ 2 * lb / sec )
```

## 5 Expression Parsing

An expression is a sequence of words, separators, and subexpressions. A subexpression is a pair of matched lexemes and all the lexemes in between.

Expressions and subexpressions can contain operators. When they do, matched implied parentheses are inserted into the expressions or subexpressions according to rules of operator precedence and associativity, and these implied parentheses create new subexpressions.

Expressions and subexpressions that do not contain operators may contain argument lists and qualifying phrases. The order in which qualifying phrases appear does not matter, and sometimes the order of arguments in an argument list does not matter.

Expressions containing operators are restructured, both by inserting implied parentheses and in other ways. Subexpressions surrounded by particular matching lexemes, such as ‘*’*’, are restructured in special ways.

The process of restructuring expressions is called *expression parsing*. Expression parsing takes an expression as input, and produces as output a raw expression that contains nothing that will trigger further restructuring.

### 5.1 Subexpressions

The first step in expression parsing is to identify *subexpressions* within an expression, without restructuring the expression, by applying the following syntax equations.

**EXPRESSION** ::= *expression-item*\*

**expression-item** ::= *word* | *separator* | *subexpression*

**subexpression** ::= *opening-mark* **EXPRESSION** *closing-mark*

**Subexpression Rule.** The *opening-mark* lexeme that begins a subexpression must lexically match (4.1.1<sup>p21</sup>) the *closing-mark* lexeme that ends a subexpression.

The subexpressions identified by this rule have explicit opening and closing marks. Below we introduce subexpressions with operators that have implicit parentheses as their opening and closing marks.

The syntactic category name *EXPRESSION* is capitalized because in this document syntactic categories representing CASTLE expressions are given completely capitalized names.

## 5.2 Expression Structure Overview

Expression structure is affected by five special marks. The *optional argument mark* ‘::’ is used in expression definitions to separate arguments that are required from those that can be optional, as in the expression ‘find-max LIST ::? COMPARATOR’, in which the ‘LIST’ argument is required but the ‘COMPARATOR’ argument is optional. The <ireorder mark ‘<:>’ is used in expression definitions to separate arguments whose order can be switched, as in the expression ‘element LIST <:> INDEX’ in which the ‘LIST’ and ‘INDEX’ argument can be switched. The *remainder mark* ‘::>’ precedes a final argument in an expression definition that represents the list of remaining arguments in a use of the definition, as in the expression ‘max ::> ARGS’, where ARGS represents the list of all the arguments given to max. The *required qualifier mark* ‘@@’ in an expression signals that the following expression item is a required qualifier, as in the expression ‘sort x @@ with order ascending’, and is implied by qualifiers (e.g. with, has) in an expression. The *optional qualifier mark* ‘??’ in an expression signals that the following expression item is an optional qualifier, as in the expression ‘sort X ?? with order 0’. When default values are provided in definitions, required qualifier marks may be converted automatically to optional qualifier marks and an optional argument mark may be inserted into an argument list automatically (13<sup>p99</sup>).

Expressions are restructured if they contain operators (e.g., +, -, \*, /), qualifiers (e.g., with, has), or qualifier shortcuts (e.g., ascending, which is a shortcut for ‘with order ascending’). A *raw expression* (5.3<sup>p42</sup>) is a particular kind of expression that will not be restructured. The only special marks that may be in a raw expression are the five special marks: the optional argument mark (::), the reorder mark (&lt;:&gt;), the remainder mark (::&gt;), and the qualifier marks (@@, ??). Translating an expression into a raw expression is called <iexpression parsing.

Some operators are matchfix operators, which surround their single argument like parentheses. Some matchfix operators have special affects on parsing. An informal summary of the standard matchfix operators is:

	Turns					
	operators and qualifiers	lexical parsing	 text	<:> @@	::> ??	::? evaluation
Brackets						
(...)	—	—	—	—		—
{...}	—	—	—	—		off
[...]	off	off	off	off		off
[[...]]	off	off	off	on		off
<<...>>	on	on	off	on		on
'...'	off	off	on	off		off
''...''	off	off	on	off		off
'''...'''	off	off	on	off		off

Other matching lexemes may be made to affect parsing by introducing operator definitions (5.4.1 <sup>p44</sup>).

Parsing is controlled by the ***parsing stack***, which contains definitions of parsers, operators, qualifiers, and qualifier shortcuts, as well as definitions related to lexical parsing. An definitional operator (p47) may push definitions onto the parsing stack. After the expression containing the operator has been parsed, these definitions will be popped from the parsing stack.

The following is a list of the kinds of ***definitions*** that may be in the ***parsing stack***:

parser	5.2 <sup>p40</sup>
operator	5.4.1 <sup>p44</sup>
qualifier	13 <sup>p99</sup>
qualifier shortcut	13 <sup>p99</sup>
lexeme replacement	4.3.1 <sup>p34</sup>
unit specifier	4.3.2 <sup>p37</sup>

A ***parser definition*** names a parser and gives an optional evaluation mode. The most recent parser definition in the parsing stack names the parser used to parse expressions. Parser definitions have the syntax:

#### **PARSER-DEFINITION**

`::= define parser PARSER-NAME evaluation-mode-option`

***PARSER-NAME*** `::= [ word word* ]`



**evaluation-mode-option** ::= *empty* | **with evaluation** *evaluation-mode*

**evaluation-mode** ::= 'on' | 'off'

It is possible to temporarily void parser definitions by placing canceling undefinitions in the parsing stack. These have the syntax:

**PARSER-UNDEFINITION** ::= **undefine** *parser*

This undefinition, as long as it is in the stack, cancels the effect of the parser definition that would be used if the undefinition were not in the stack.

There are three standard *parsers*: the **marks parser-MARKS-PARSER-**, where only the optional argument mark (::?), reorder mark (<:>), remainder mark (::>), required qualifier mark (@@), and optional qualifier mark (??), are recognized; the **operators parser-OPERATORS-PARSER-**, where these marks are recognized along with operators, qualifiers, and qualifier shortcuts, and where lexical parsing (4.3<sup>p33</sup>) is performed; and the **text parser-TEXT-PARSER-**, where only the format-separator (|) and punctuation are recognized.

From the parsing point of view there are several types of expression restructuring that can occur. One, text restructuring, is done by the text parser (5.6<sup>p60</sup>), which is invoked by '' quotes, and does things like turn 'I am!' into [-SENTENCE- I am !]. Other kinds of restructuring are done by the operators parser (5.4<sup>p44</sup>). The operators '+', '-', and '\*' in the expression `x + y - 5 * z` cause this expression to be restructured as `[- [+ x y] [* 5 z]]`. The qualifier 'with' in the expression `sort x with order ascending` causes this expression to be restructured as `[sort x @@ with order ascending]`. The qualifier shortcut 'ascending' in the expression `sort x ascending` causes this expression to be restructured as `[sort x @@ with order ascending]`. In addition, lexical parsing (4.3<sup>p33</sup>) is performed by the operators parser.

In expressions that cannot contain operators, words and separators that would be recognized elsewhere as operators, qualifiers, or qualifier shortcuts are not recognized as such. Thus in `[word +]` the + is not recognized as an operator, and in `[item ascending]` `ascending` is not recognized as a qualifier shortcut.

Some of the matchfix operators above affect evaluation. Evaluation differs from parsing in that it is controlled by the 'wrapper functions' **-EVAL-ON-** and **-EVAL-OFF-** that are inserted automatically during parsing when the evaluation mode is changed (5.4.4<sup>p53</sup>). Thus in a context in which evaluation is on, the Y in 'Let X be <<Y>>.' will be evaluated but X will not be, because ' ' turns evaluation off, while << >> turns it back on. The mechanism for this is that the expression 'Let X be <<Y>>.' is parsed to become

```
[-EVAL-OFF- [-SENTENCE- let X be [-EVAL-ON- Y] .]]
```

and the evaluator, running during program execution, sees the `-EVAL-OFF-`, and does not evaluate the following subexpression, except for parts that have `-EVAL-ON-`.

Note also that if the expression designated by `-EVAL-ON-` evaluates to a list inside a list, flattening is used to merge the lists. Thus if `Y` has the value `'a banana'` when the last expression is evaluated, the result will be the same as evaluating `'Let X be a banana.'`, and not `'Let X be (a banana).'`.

Also note that matchfix operators that turn evaluation off (or on) do not produce `-EVAL-OFF-` (or `-EVAL-ON-`) wrappers if they are used in a context where the evaluation mode is already off (or on). Thus if `'Let X be a '<<Y>>'.'` is evaluated when `Y` is `'banana'`, the result is the same as evaluating `'Let X be a 'banana'.'`<sup>5</sup>

[TBD: Does this last example need work?]

### 5.3 Raw Expressions

A raw expression is an expression that contains no operators, qualifiers, qualifier shortcuts, or text parsing contexts that induce restructuring. Raw expressions can be directly represented using the `[]` matching lexemes and the `::?` optional argument, `<:>` reorder, `::>` remainder, `@@` required qualifier, and `??` optional qualifier marks.

When an expression is parsed, the output is a raw expression. The following are examples, in which the input expressions are assumed to appear in the context of `{}` matching lexemes (so operators, qualifiers, and qualifier shortcuts are turned on and text parsing is turned off):

---

<sup>5</sup>Classical programming languages would require that `'Let X be a '<< <<Y>> >>'.'` be evaluated to get this result. Two nested `<< >>`'s would be needed to overcome the two nested `' '`.

Input Expression	Raw Expression Equivalent
x	x
(+)	+
3 * x * x + 5	[+ [* [* 3 x] x] 5]
x = y = 10	[= x [= y 10]]
0 < x <= 5	[-COMPARE- 0 < x <= 5]
x = 'Hello <<y>>.'	[= x [-EVAL-OFF- [-SENTENCE- Hello [-EVAL-ON- y] .]]]

The syntax of raw expressions is defined as follows:

**RAW-EXPRESSION** ::= *word* | *separator* | *raw-subexpression*

**raw-subexpression**  
 ::= [ *raw-expression-head* *raw-argument*\* ]  
 | [[ *raw-expression-head* *raw-argument-list* *raw-qualifier-phrase*\* ]]

**raw-expression-head** ::= **RAW-EXPRESSION**

**raw-argument-list** ::= *raw-base-argument-list* *raw-argument-remainder-option*  
 | *raw-base-argument-list* *optional-argument-mark*  
*non-empty-raw-base-argument-list*  
*raw-argument-remainder-option*

**optional-argument-mark** ::= ::?

**raw-base-argument-list** ::= *empty* | *non-empty-raw-base-argument-list*

**raw-non-empty-base-argument-list**  
 ::= *raw-argument*  
 { *reorder-mark-option* *raw-argument* } \*

**reorder-mark-option** ::= *empty* | <:>

**raw-argument-remainder-option** ::= *empty* | ::> *raw-argument*

**raw-argument** ::= **RAW-EXPRESSION**

**raw-qualifier-phrase** ::= *qualifier-mark* *raw-qualifier-head* *raw-argument-list*

**qualifier-mark** ::= *required-qualifier-mark* | *optional-qualifier-mark*

**required-qualifier-mark** ::= @@

**optional-qualifier-mark** ::= ??

***raw-qualifier-head*** ::= *RAW-EXPRESSION*

Special marks are recognized only if `[ [ ] ]` surround a *raw-subexpressions*, and not if `[ ]` surround the subexpression.

The order of *raw-qualifier-phrases* in a *RAW-EXPRESSION* does not matter. The order of arguments separated by *reorder-marks* (`<:>`) does not matter. The *optional-argument-mark* (`::?`), *reorder-mark* (`<:>`), *remainder-mark* (`::>`), and *optional-qualifier-mark* (`??`) appear in patterns (7.1<sup>p82</sup>), but not in evaluable expressions.

## 5.4 Operators

Operators restructure expressions in which they occur. Operators are defined by operator definitions that can be added to the parsing stack. Certain operators are ‘definitional’ (p47), and it is just these operators which accept definitions to be pushed into the parsing stack.

### 5.4.1 Operator Definitions

***Operator definitions*** can be pushed onto the parsing stack, and are used by the operators parser. An operator definition specifies for each operator the following:

- Fixity
- Name
- Precedence
- Associativity
- Parser
- Control Flags
- Wrapper
- Subdefinitions

**5.4.1.1 Operator Fixity.** An operator has one of the following fixities:

<b>infix</b>	E.g., + in <code>x + 5</code> .
<b>prefix</b>	E.g., - in <code>- 5</code> .
<b>postfix</b>	E.g., ! in <code>x!</code> .
<b>matchfix</b>	E.g., <code>[   ]</code> in <code>[  x - 5  ]</code> .

An infix operator is placed between its two operands; a prefix operator is placed before its one operand; a postfix operator is placed after its one operand; and a matchfix operator

surrounds its one operand. Operands of prefix, infix, and postfix operators may not be empty; but the operand of a matchfix operator may be empty. An infix operator may not begin or end an expression; a prefix operator must begin an expression; a postfix operator must end an expression; and a matchfix operator has two parts: an opening operator that must begin an expression and a closing operator that must end the same expression.

**5.4.1.2 Operator Name.** An operator definition has a sequence of lexemes that is the *name* of the operator. The operator inside a subexpression is just this sequence of lexemes, except for matchfix operators, which have two sequences of lexemes, an ***opening operator name*** that must begin the subexpression and a ***closing operator name*** that must end the subexpression.

By abuse of language, the term ‘***operator***’ is often used as a synonym for ‘operator name’. Similarly ‘***opening operator***’ is used as a synonym for ‘opening operator name’ and ‘***closing operator***’ is used as a synonym for ‘closing operator name’.

The opening and closing names of a matchfix operator are bundled into a single operator name that is sequence of lexemes consisting of the opening operator name followed by a ‘...’ lexeme followed by the closing operator name. For example, the matchfix operator named ‘[ | ... | ]’ permits subexpressions like ‘[ | x - 5 | ]’. Here the opening operator name is ‘[|’ and the closing operator name is ‘|]’.

A matchfix operator name must begin with an opening mark and end with a matching closing mark. Non-matchfix operator names cannot contain opening or closing marks.

When a subexpression is tested for matchfix operators, any explicit opening and closing marks that begin and end the subexpression are included in the subexpression. Implicit parentheses, are not included. Thus a matchfix operator such as ‘( | ... | )’ can only be invoked with explicit parentheses, and not with implicit parentheses.

**5.4.1.3 Operator Precedence.** The precedence of an operator is an integer. Precedence is used in selecting which infix operators in an expression to consider (5.4.3<sup>p49</sup>). Only infix operators have precedence.

**5.4.1.4 Operator Associativity.** An infix operator has an *associativity* that is a sequence of lexemes. Three associativities, **left**, **right**, and **none**, have special meaning. If an infix operator has some other associativity, it is said to have ***named associativity***. Only infix operators have associativity.

If more than one infix operator of lowest precedence is selected in an expression by the operator selection algorithm(5.4.3<sup>p49</sup>), all the infix operators that are selected must have the identical operator definitions except for operator names, and the associativity in these definitions must not be **none**.

In this situation we consider the following subcases.

If all operators are **left** associative, all the infix operators but the rightmost are deselected, so it is as if operators to the left had higher precedence than those to the right.

If all operators are **right** associative, all the infix operators but the leftmost are deselected, so it is as if operators to the right had higher precedence than those to the left.

If all operators are of **named associativity**, all remain selected, and implicitly parenthesized subexpressions will be created between the operators as well as between the beginning of the expression and the first operator and between the last operator and the end of the expression. Then the associativity, as a lexeme sequence, is prepended to the expression. Thus if the infix operators `<` and `<=` have associativity **-COMPARE-**, the expression `0 < x - 3 <= 5` will be rewritten first as **[-COMPARE- 0 < (x - 3) <= 5]**, and then finally as **[-COMPARE- 0 < [- x 3] <= 5]**.

**5.4.1.5 Operator Parser.** A parser is the name of the function that is called with an expression as its single argument in order to parse the expression.

Parsers can be pushed onto the parsing stack. The parser used for subexpressions of an expression is the parser nearest the top of the parsing stack.

The **parser** of an operator definition is an optional word that names a parser. If present, a **PARSER-DEFINITION** (p40) naming the parser is pushed onto the parsing stack before subexpressions are parsed, and popped from the parsing stack after subexpressions have been parsed.

The following are standard parsers:

The **operators parser, -OPERATORS-PARSER-**. Lexical parsing is done. Then operators, qualifiers, qualifier shortcuts, and the `::?` optional argument, `<:>` reorder, `::>` remainder, `@@` required qualifier, and `??` optional qualifier marks are recognized. The special constructs of text parsing, e.g., the `|` format separator, are not recognized.

The **marks parser, -MARKS-PARSER-**. Lexical parsing is not done. The `::?` optional argument, `<:>` reorder, `::>` remainder, `@@` required qualifier, and `??` optional qualifier marks are recognized. Operators, qualifiers, qualifier shortcuts, and the special constructs of text parsing, e.g., the `|` format separator, are not recognized.

The *text parser*, **-TEXT-PARSER-**. Lexical parsing is not done. The special constructs of text parsing (5.6 <sup>p60</sup>) are recognized. Operators, qualifiers, qualifier shortcuts, and the ::? optional argument, <:> reorder, ::> remainder, @@ required qualifier, and ?? optional qualifier marks are not recognized.

**5.4.1.6 Operator Control Flags.** Operators can be associated with *operator control flags* that affect parsing of subexpressions of the operator.

**definitional** This flag may only be given for a right associative infix operator. It causes the left operand of the operator to be inspected to see if it is an expression beginning with the word ‘define’ or ‘undefine’. Such an expression is called a *definition*, and is assumed to push a definition into the parsing stack. See p40 for a list of the kinds of definition that may be pushed into the parsing stack.

If its left operand is not a definition, nothing special is done by a definitional operator.

If its left operand is a definition, the left operand is executed by the parser and may push definitions into the parsing stack, the expression containing the infix operator is replaced by just its right operand, the right operand is parsed, and then any definitions pushed while executing the left operand are popped off the parsing stack.

**5.4.1.7 Operator Wrapper.** An operator wrapper is a raw expression containing the lexeme ‘...’. If operator selection for an expression selects a single operator with a wrapper, the expression is restructured to be a copy of the wrapper with its ‘...’ lexeme replaced by the parse of the expression being restructured.

For example, if the matchfix operator ‘(\* ... \*)’ has the wrapper ‘[try ...]’, then the expression ‘(\* x + y \*)’ will be restructured to be ‘[try (x + y)]’. See p54.

**5.4.1.8 Operator Subdefinitions.** An operator definition can contain a list of definitions that are pushed onto the parsing stack before the subexpressions of the operator are parsed, and are popped from the stack after the subexpressions are parsed. To avoid confusion, these definitions are called *subdefinitions*.

Note that subdefinitions can include any definitions that can be pushed onto the parsing stack, and can also include undefinitions, that inactivate all definitions with particular names in the stack. See p40 for a list of the kinds of definition that may be pushed into the parsing stack.

### 5.4.2 Operator Definition Syntax

The syntax of operator definitions is:

```

OPERATOR-DEFINITION
    ::= define operator OPERATOR-LABEL operator-definition-qualifier*
OPERATOR-LABEL ::= [ operator-fixity operator-name ]
operator-fixity ::= prefix | infix | postfix | matchfix
operator-name ::= { word | separator } { word | separator }*
operator-definition-qualifier ::=
    | operator-precedence-qualifier
    | operator-associativity-qualifier
    | operator-parser-qualifier
    | operator-flags-qualifier
    | operator-wrapper-qualifier
    | operator-subdefinition-qualifier
operator-precedence-qualifier ::= with precedence INTEGER
operator-associativity-qualifier
    ::= with associativity OPERATOR-ASSOCIATIVITY
    | left
    | right
OPERATOR-ASSOCIATIVITY
    ::= [ { word | separator } { word | separator }* ]
operator-parser-qualifier ::= with parser PARSER-NAME
PARSER-NAME ::= [ { word | separator } { word | separator }* ]
operator-flags-qualifier
    ::= with flags OPERATOR-FLAGS
    | definitional
OPERATOR-FLAGS ::= [ { word | separator } { word | separator }* ]
operator-wrapper-qualifier ::= with wrapper OPERATOR-WRAPPER
OPERATOR-WRAPPER ::= [ { word | separator } { word | separator }* ]
operator-subdefinitions-qualifier ::= with subdefinitions subdefinitions-block
subdefinitions-block ::= { declaration-group }
declaration-group ::= see p87

```



which correspond to the definitions (7.1<sup>p82</sup>):

```

define qualifier with

define qualifier shortcuts
  ( [left => with associativity [left]],
    [right => with associativity [right]],
    [definitional => with flags [definitional]] )

define define operator LABEL
  ?? with precedence PRECEDENCE
  ?? with associativity ASSOCIATIVITY
  ?? with parser PARSER
  ?? with flags FLAGS
  ?? with wrapper WRAPPER
  ?? with subdefinitions SUBDEFINITIONS
  <--

```

It is possible to temporarily void an operator definition by placing a canceling undefinition in the parsing stack. This has the syntax:

#### ***OPERATOR-UNDEFINITION***

```
 ::= undefine operator  OPERATOR-FIXITY  OPERATOR-NAME
```

### 5.4.3 Operator Selection

The operator selection algorithm is run by the the ***operators parser***, **-OPERATORS-PARSER-**, to check an expression for operators. This algorithm selects operators whose name appears in the expression, and then applies rules to de-select some selected operators. If at the end of the algorithm there is exactly one operator selected, the definition of that operator is used to restructure the expression. If there is more than one operator selected, it is a parse error unless the definitions of all selected operators are identical except for operator names, in which case the common part of these definitions is used to restructure the expression.

The operator selection algorithm uses the current parsing definition stack to determine which operators are defined. For this algorithm, the most important parts of each operator definition are the operator name and operator fixity.

The operator selection algorithm produces a result that is often the same as the more common

operator precedence rules for parsing expressions. However, in CASTLE an operator can change the parsing stack used to parse its operand subexpressions, and this allows CASTLE to use various grammars for subexpressions.<sup>6</sup>

The operator selection algorithm works on an expression viewed as a sequence of words, separators, and proper subexpressions, possibly surrounded by opening and closing marks. The proper subexpressions are bounded by opening and closing marks: see 5.1<sup>p38</sup>. Operators found in the expression cannot include any of the proper subexpressions.

The operator selection algorithm uses the concepts of prefix sequence and postfix sequence. A **prefix sequence** is a sequence of defined prefix operator names. A **postfix sequence** is a sequence of defined postfix operator names.

The **operator selection algorithm** executes the following steps in the order given, repeating each step until it does nothing new before proceeding to the next step.

1. If the expression begins with an implicit '(' and ends with an implicit ')', then discard the beginning '(' and ending ')'.  
 Similarly if the expression begins with an explicit '(' and ends with an explicit ')', and if there is no defined matchfix operator whose opening name begins the expression and whose closing name ends the expression, then discard the beginning '(' and ending ')'.

2. Select any defined matchfix operator whose opening name begins the expression and whose closing name ends the expression.
3. If two matchfix operators are selected, the opening operator name of the first is longer than the opening operator name of the second, and the closing operator name of the first is longer than the closing operator name of the second, then the second operator is deselected.

Thus if '[ ... ]' and '[ ... ]' are two selected matchfix operators, the second will be deselected.

4. If more than one matchfix operator is still selected, the parse is in error.
5. If a matchfix operator is selected, the algorithm terminates successfully at this point, without selecting any other operators. The part of the expression between the opening

---

<sup>6</sup>In practice we expect this may be equivalent to using an operator precedence grammar to parse everything outside explicit opening and closing marks, and only permitting matchfix operators to change grammars. But CASTLE currently has more capabilities than this, and it is a matter of experiment to see whether these extra capabilities are useful.

operator name and closing operator name is the sole operand of the matchfix operator. This operand is implicitly parenthesized, and may be empty.

6. Select all defined infix operators that occur in the expression.
7. If two selected infix operators overlap, and one includes and is longer than the other, deselect the shorter operator.
8. If one operator is selected at this point, and the operator name is the entire expression, de-select the operator and terminate the algorithm. Thus in the expression ' $x = (*)$ ', '\*' will be deselected when the subexpression ' $(*)$ ' is parsed.
9. De-select any infix operator that is included in a prefix sequence and not preceded by a non-empty postfix sequence if the infix operator is preceded by the beginning of the expression, by an infix operator selected at the end of the last step, or by a non-empty part of a prefix sequence that includes the infix operator.

De-select any infix operator that is included in a postfix sequence and not followed by a non-empty prefix sequence if the infix operator is followed by the end of the expression, by an infix operator selected at the end of the last step, or by a non-empty part of a postfix sequence that includes the infix operator.

Thus in the expression ' $x + - y$ ', '-' would be deselected as an infix operator by this step, as it is a prefix operator preceded by an infix operator that cannot be a postfix operator. However '+' would not be deselected, as it is not preceded by an operator or expression beginning.

10. The parse is in error if two consecutive or overlapping infix operators are still selected at this point, or if a still selected infix operator begins or ends the expression.
11. If two infix operators are selected, and one has strictly lower precedence than another, de-select the operator with higher precedence.
12. If several infix operators are still selected at this point, all these operators must have the same associativity, which must not be '**none**'. Otherwise the parse is in error.
13. If several infix operators are selected at this point and their common associativity is '**left**', all but the rightmost infix operator are deselected. Similarly if several infix operators are selected at this point and their common associativity is '**right**', all but the leftmost infix operator are deselected.

14. If several infix operators are still selected at this point, all these operators must have the same definitions except for operator names. Otherwise the parse is in error.
15. If any infix operators are still selected at this point, the algorithm terminates successfully. The parts of the expression between the operators, before the first operator, and after the last operator are the operands, and are implicitly parenthesized. No operand is empty.
16. Select all prefix operators that begin the expression, and all postfix operators that end the expression.
17. If two prefix operators are selected, and one is longer than the other, de-select the shorter. Similarly, if two postfix operators are selected, and one is longer than the other, de-select the shorter.
18. If both prefix and postfix operators are selected at this point, the postfix operators are deselected. Thus postfix operators are in effect given precedence over prefix operators.
19. If more than one operator is selected at this point, the parse is in error. This would be the result of ambiguity among prefix operators, or ambiguity among postfix operators.
20. If one operator is selected at this point, and the operator name is the entire expression, de-select the operator and terminate the algorithm. Thus in the expression ' $x = (+)$ ', '+' will be deselected when the subexpression ' $(+)$ ' is parsed.
21. If one operator is selected at this point, the algorithm terminates successfully. Note the operator must be prefix and begin the expression or postfix and end the expression. The part of the expression that is not the operator name is the sole operand, and is implicitly parenthesized. Note this operand cannot be empty.
22. Zero operators are selected at this point. If the expression contains any operator the parse is in error. This can happen, for example, if the only operator in the expression is an prefix operator not at the beginning of the expression, or a postfix operator not at the end. It can also happen in an expression like ' $* x$ ' in which the infix operator '\*' was deselected because it begins an expression.
23. The algorithm terminates successfully at this point with zero operators selected.

As noted above, the parsing stack can be changed before subexpressions are parsed, but the definitions of prefix, postfix, and infix operators before the change affect operator selection.

It is possible for parsing stack changes to produce anomalous seeming results. For example, if a parsing stack change introduced by the infix operator `+++` defines a new prefix operator named `*`, where `*` was previously as just an infix operator, then the expression `'x +++ * y'` would be illegal by step 10<sup>p51</sup> because `*` would not be defined as a prefix operator soon enough. In this case one would have to write `'x +++ (* y)'` to get a legal parse.

#### 5.4.4 Post Operator Selection Processing

Post operator selection processing is done after operator selection by the ***operators parser*** **-OPERATORS-PARSER-**, whether or not any operators are selected. After operators are selected without a parse error, all the selected operators have identical definitions except for operator name. The common part of these definitions, which we will refer to as 'the operator definition', is used to control post operator selection processing of the expression containing the operators. If no operators are selected, a definition containing no optional parts (no named associativity, no parser, no control flags, no wrapper, no subdefinitions) is used to control post operator selection processing.

Post operator selection processing is done in the following steps.

1. If the operator definition has a parser, this is pushed onto the parsing parser stack.
2. If the operator definition has subdefinitions, these are pushed onto the parsing definition stack.
3. Subexpressions are parsed left to right by calling the parser at the top of the parsing parser stack, and subexpressions are replaced by their parsed versions.

An exception is made if the operator definition has the **definitional** control flag (note the operator must be **right** associative) and if the first subexpression, after being parsed, is a definition that can be pushed onto the parsing stack. In this case, this definition is pushed onto the parsing stack, the second subexpression is parsed, the current expression is replaced by the parsed second expression, and parsing continues at step 6<sup>p54</sup> below.

4. If the operator definition has no wrapper, it is processed as follows:
  - (a) If the operator definition has named associativity, this associativity is prepended to the expression.
  - (b) Otherwise if the operator definition is not **matchfix**, any selected operator (there can be at most one) is moved to the beginning of the expression.

- (c) Otherwise if the operator definition is **matchfix**, the parse is in error.
- 5. If the operator definition has a wrapper, it is processed as follows:
  - (a) If the operator definition does not have named associativity, all selected operators are removed from the expression.
  - (b) The expression is replaced by its wrapper with the word `...` in the wrapper replaced by the expression.
- 6. If anything was pushed onto the parsing stack by steps 1 and 2 above, these things are popped from the stack.
- 7. The expression is returned as the result of parsing the expression.

#### 5.4.5 Equivalence to Classical Parsing

In this section we prove that parsing using the operator selection algorithm of 5.4.3<sup>p49</sup> is equivalent to the classical expression parsing algorithm in typical circumstances.

First we define classical parsing. A **classical parse** is a parse according to the syntax equations:

$$\begin{aligned}
 \textit{expression} &::= \textit{expression}_0 \\
 \textit{expression}_n &::= \textit{expression}_{n+1} \{ \textit{infix-operator}_n \textit{expression}_{n+1} \}^* \quad (0 \leq n \leq M) \\
 \textit{expression}_{M+1} &::= \textit{prefix-operator}^* \textit{expression}_{M+2} \\
 \textit{expression}_{M+2} &::= \textit{expression}_{M+3} \textit{postfix-operator}^* \\
 \textit{expression}_{M+3} &::= \textit{non-operator non-operator}^* \mid ( \textit{expression} ) \\
 \textit{infix-operator}_n &::= \textit{infix operator of precedence } n \quad (0 \leq n \leq M) \\
 \textit{infix-operator} &::= \textit{infix-operator}_0 \mid \dots \mid \textit{infix-operator}_M \\
 \textit{operator} &::= \textit{prefix-operator} \mid \textit{postfix-operator} \mid \textit{infix-operator} \\
 \textit{non-operator} &::= \textit{lexeme that is not an operator or parenthesis}
 \end{aligned}$$

We want to show that the parse of any expression that has an unambiguous classical parse is obtained by the operator selection algorithm of 5.4.3<sup>p49</sup>, and conversely, that if the operator selection algorithm obtains a parse, that parse is an unambiguous classical parse. In order to show these things we must assume:

- A1. All operators are defined in the parsing stack, and the parsing stack does not change during the expression parse.
- A2. There are no matchfix operators (if these begin and end with opening and closing marks and do not overlap other operators, they behave just like parentheses).
- A3. No two operators can overlap. (This is trivial if all operators are just a single lexeme).
- A4. No operator can overlap a non-operator.
- A5. No operator is both prefix and postfix.

Here we mean by ‘operator’ the lexemes that represent an operator, so one operator can be both infix and prefix. However no operator can be infix with two different precedences.

Assumption A5 reflects a minor distinction between the operator selection algorithm and classical parsing.

It will be convenient to use the following alternative parsing. A ***semi-classical parse*** is a parse according to the syntax equations:

$$\begin{aligned}
 \textit{expression} &::= \textit{primitive-expression} \{ \textit{infix-operator primitive-expression} \}^* \\
 \textit{primitive-expression} &::= \textit{prefix-operator}^* \textit{atomic-expression} \textit{postfix-operator}^* \\
 \textit{atomic-expression} &::= \textit{non-operator non-operator}^* \mid ( \textit{expression} )
 \end{aligned}$$

One can then prove the following:

**Lemma 5.1** *There is a 1-1 correspondence between classical and semi-classical parses of an expression such that corresponding parses have exactly the same infix-operators, prefix-operators, and postfix-operators.*

Proof: By induction on the size of the parse trees.

**Corollary 5.2** *If an expression has two different classical parses, some operator in the expression must have different fixities in the two parses.*

Proof: Two semi-classical parses of the expression that have the same operators with the same fixities must be identical.

**Lemma 5.3** *In a classical parse of an expression:*

- (a) *the expression is not empty,*
- (b) *the expression does not end with an operator that is not a postfix-operator,*
- (c) *the expression does not begin with an operator that is not a prefix-operator,*
- (d) *each prefix-operator is preceded by the beginning of the expression, or by a (, or by an infix-operator, or by a prefix-operator,*
- (e) *each postfix-operator is followed by the end of the expression, or by a ), or by an infix-operator, or by a postfix-operator,*
- (f) *no infix-operator is preceded by an infix-operator or a prefix-operator,*
- (g) *no infix-operator is followed by an infix-operator or a postfix-operator,*
- (h) *an expression<sub>n</sub> cannot contain outside parentheses any infix-operator<sub>m</sub> for  $m < n$ .*

Proof: Each part can be proved separately by induction on the syntax equations.

**Lemma 5.4** *In a semi-classical parse of an expression:*

- (a) *no primitive-expression is empty,*
- (b) *an infix- or postfix-operator must be preceded by a primitive-expression,*
- (b) *an infix- or prefix-operator must be followed by a primitive-expression.*

Proof: Each part can be proved separately by induction on the syntax equations.

**Theorem 5.5** *Under assumptions A1-A5<sup>p55</sup>, if an expression has a classical parse, the expression has two or more distinct classical parses (or in other words, has an ‘ambiguous’ classical parse) if and only if the expression contains a sequence ‘X Y’ of two operators such that X can be postfix or infix and Y can be prefix or infix.*

Proof: If the *expression* has two classical parses, these cannot have identical infix operators, is if they did, the fixities assigned to operators by the two parses would be the same, since the fixities assigned to an operator in a *primitive-expression* depends solely whether the operator is before or after the *non-operators* in the *primitive-expression*. Then by corollary 5.2 the two classical parses could not be distinct.



Therefore if the *expression* has two classical parses, some infix operator in the one parse must not be an infix operator in the other parse. Without loss of generality assume this operator is outside parentheses, is an infix operator in the first parse, and is a prefix operator in the second parse (the postfix case being handled by a symmetric argument). Take the first operator  $Y$  in the *expression* that has this property.  $Y$  cannot be at the beginning of the *expression*, for it is an infix operator in the first parse. Therefore  $Y$  must be preceded by an infix or prefix operator  $X$  in the second parse, by Lemma 5.3. But  $X$  cannot be an infix or prefix operator in the first parse, also by Lemma 5.3, so it must be a postfix operator in the first parse. Since  $X$  is postfix in the first parse, and must be prefix or infix in the second parse, it must be infix in the second parse by A5<sup>p55</sup>. So the *expression* contains a sequence of operators ‘ $X Y$ ’ such that  $X$  can be postfix or infix and  $Y$  can be prefix or infix.

Conversely, if the *expression* has a classical parse it has a semi-classical parse, and if it has a sequence ‘ $X Y$ ’ of operators the first of which may be infix or postfix and the second infix or prefix, it must by Lemma 5.4 be of the form:

$$\dots \textit{primitive-expression } X Y \textit{ primitive-expression } \dots$$

as  $X$  must be preceded and  $Y$  followed by a *primitive-expression*. Then we can either take  $X$  to be an infix operator and  $Y$  to be a prefix operator, or  $X$  to be a postfix operator and  $Y$  to be an infix operator, giving two distinct semi-classical parses, which in turn give two distinct classical parses.

QED

**Theorem 5.6** *If an expression has a unique unambiguous classical parse, a parse using the operator selection algorithm of 5.4.3<sup>p49</sup> will produce exactly this classical parse, provided assumptions A1-A5<sup>p55</sup> hold.*

Proof: By induction on the classical syntax equations. The main step is proving that given

$$\textit{expression}_n ::= \textit{expression}_{n+1} \{ \textit{infix-operator}_n \textit{expression}_{n+1} \}^*$$

the *infix-operator* <sub>$n$</sub> ’s are exactly the operators selected at the end of step 11<sup>p51</sup> of the operator selection algorithm. Each *infix-operator* <sub>$n$</sub>  is preceded by an *expression* <sub>$n+1$</sub>  that by Lemma 5.3 cannot end in a prefix or infix operator that is not also a postfix operator. And each *infix-operator* <sub>$n$</sub>  is followed by an *expression* <sub>$n+1$</sub>  that by Lemma 5.3 cannot begin with a postfix or infix operator that is not also a prefix operator. So no *infix-operator* <sub>$n$</sub>  is deselected

by step  $9^{p51}$  of the operator selection algorithm. It remains to show that no  $expression_{n+1}$  contains an infix operator outside parentheses that is not deselected by step  $9^{p51}$  or step  $11^{p51}$ .

No  $expression_{n+1}$  can contain an infix operator of precedence below  $n+1$  outside parentheses, by Lemma 5.3. But an  $expression_{n+1}$  might contain an infix operator of precedence below  $n+1$  outside parentheses if that operator is prefix or postfix in the classical parse, and we must show that these are deselected by step  $9^{p51}$  of the operator selection algorithm.

We will show that every *prefix-operator* or *postfix-operator* of the classical parse of  $expression_n$  that is outside parentheses and that can also be an infix operator is deselected by step  $9^{p51}$  of the operator selection algorithm. We will give the argument for *prefix-operators* and leave the symmetrical argument for *postfix-operators* to the reader.

By Lemma 5.3 a prefix operator outside parentheses is preceded by the beginning of the expression, or by an infix operator, or by another prefix operator. If it is preceded by the beginning of expression or by another prefix operator, it must be deselected by step  $9^{p51}$ , as by assumption A5<sup>p55</sup> a prefix operator cannot be a postfix operator. If it is preceded by an infix operator that can be a postfix operator, then we have a sequence of two infix operators, the first of which can also be a postfix operator, and the second of which can also be a prefix operator. But by Theorem 5.5 this would make the classical parse of the expression ambiguous, which is disallowed, so this case cannot happen.

QED

**Lemma 5.7** *Let the operator selection algorithm be run on an expression to produce some selected operators and some subexpressions, and let  $S$  be one of the subexpressions. Then let the operator selection algorithm be run on  $S$ . Under assumptions A1-A5<sup>p55</sup>, both algorithm executions select the same infix operators in  $S$  at the end of their respective step  $9^{p51}$ s.*

Proof: The infix operators in  $S$  are the same for both algorithm executions by the assumptions. We must prove that step  $9^{p51}$  deselects the same operators in  $S$  in both algorithm executions.

Suppose an operator is deselected by step  $9^{p51}$  in the algorithm run for the whole expression because the operator is a prefix operator preceded by a prefix operator that cannot be a postfix operator, or by an infix operator that cannot be a postfix operator, or by the beginning of the expression. Then this will also be true when the algorithm is run for the subexpression, though any preceding operator may be replaced by a subexpression beginning. A symmetrical argument applies to deselected postfix operators. So infix operators deselected when the algorithm is run on the whole expression are also deselected when the algorithm is run on the subexpression.

Now suppose an operator  $Y$  is deselected by step  $9^{p51}$  in the algorithm run for the subexpression  $S$  because the operator is a prefix operator preceded by a prefix operator that cannot be a postfix operator, or by an infix operator that cannot be a postfix operator, or by the beginning of the subexpression  $S$ . Then this will also be true when the algorithm is run for the whole expression, except possibly in the case where the operator is at the beginning of the subexpression but not at the beginning of the whole expression. In this case the operator is preceded in the whole expression by an operator  $X$  that must have been selected by the algorithm run on the whole expression. Since  $X$  is before  $Y$ ,  $X$  cannot have been selected as a postfix operator. Therefore  $X$  must be either an infix or prefix operator. If it is a prefix operator it cannot also be postfix, by  $A5^{p55}$ , so  $Y$  will be deselected by the algorithm on the whole expression. If it is an infix operator that is also a postfix operator, neither  $X$  nor  $Y$  would be deselected by step  $9^{p51}$  of the algorithm on the whole expression, and that algorithm would have failed at step  $10^{p51}$ , which it did not do. So  $X$  cannot be infix and postfix, and  $Y$  will be deselected when the algorithm is run on the whole expression. Therefore in all cases  $Y$  is deselected when the algorithm is run on the whole expression. A symmetrical argument applies when  $Y$  is infix and postfix. So infix operators deselected when the algorithm is run on the subexpression are also deselected when the algorithm is run on the whole expression.

QED

Note that use of assumptions A1-A4 in this proof could be replaced by the assumption that no infix operator that is not deselected overlaps a fractional part of another operator. This keeps operators from crossing the boundaries of subexpressions.

**Theorem 5.8** *If a sequence of lexemes is successfully parsed using the operator selection algorithm, and assumptions A1-A5<sup>p55</sup> hold, the resulting parse is the unique classical parse of the sequence, provided the resulting parse does not contain a subexpression consisting of just an operator (see steps  $8^{p51}$  and  $20^{p52}$  of the algorithm).*

Proof: By steps  $6^{p51}$  through  $10^{p51}$  and Lemma 5.7 the algorithm divides the expression into non-empty subexpressions separated by infix operators. By steps  $16^{p52}$  through  $23^{p52}$  each subexpression is a *primitive-expression*. Therefore the parse produced is a semi-classical parse. And by the various steps that select only one operator during each iteration of the algorithm, the result is a classical parse.

The classical parse can be shown to be unique by showing the *expression* does not contain a sequence of operators ' $X Y$ ' such that  $X$  can be infix and postfix and  $Y$  can be infix and prefix. If the *expression* contained such a sequence, both  $X$  and  $Y$  would not be deselected

by step  $9^{p51}$ , and the parse would be in error by step  $10^{p51}$ . Since the *expression* cannot contain such a sequence, by Theorem 5.5 the classical parse is unique.

QED

## 5.5 Qualifiers

TBD

## 5.6 Text Parsing

*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 not 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 (4.1.4<sup>p23</sup>). 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.

### 5.6.1 Section, Paragraph, and Sentence Parsing

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 ::=  [-PARAGRAPH- sentence 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 ::= word | separator | 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  $\sim$  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 .]
''^Bill swung. But he missed!''	parses as	[-PARAGRAPH- [-SENTENCE- Bill swung .] [-SENTENCE- but he missed !]]
''^I liked the party.  Later, we caught the bus.''	parses as	[-SECTION- [-PARAGRAPH- [-SENTENCE- I liked the party .]] [-PARAGRAPH- [-SENTENCE- 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.

### 5.6.2 Text with Format Separators

TBD

## 6 Expression Graphs

Expression graphs store expressions. An *expression graph* is a directed possibly cyclic graph with labels on all arrows and on some nodes, with a designated root node, and with a possibly empty set of graph node valued variables.

Expression graph arrow and node labels must be single words or separators. Labeled nodes cannot be the source of any arrow. Several arrows with the same source node may have the same arrow label.

A node that does not have a label and is not the source of any arrows is called a **null node**.

An expression graph variable has a name, a value, and an optional default. Variable names must be single words or separators. Both the value and the default are graph nodes. The default may be missing. Variable values are often, but by no means always, null nodes.

All nodes in an expression graph are reachable from either the root or from the value or default of one of the graph's variables.

If there are no variables and no node is the destination of more than one arrow, the graph is a tree. In this case it is called an **expression tree**.

There is a correspondence between raw expressions and certain expression trees, called **raw expression trees**, that is one-to-one, except that two raw expressions differing only in the order of their qualifier clauses and the order of their reorderable arguments correspond to the same expression tree.

There are several operations that convert raw expression trees into general expression graphs (6.7<sup>p76</sup>). All these use designated operators, such as '//', to identify expression graph variables and their values. The most commonly used conversion operation also uses the convention that words beginning with a capital letter or with '#' followed by a capital letter name expression graph variables.

## 6.1 Expression Graph Notation

The following is an example of the notation we will use to represent expression graphs:<sup>7</sup>

$$\left[ \begin{array}{l} \boxed{\{\text{ROOT}\}} \\ \text{fee} : \left[ \begin{array}{l} \boxed{\text{X}} \\ 51 \end{array} \right] \\ \text{fie} : \left[ \boxed{\text{X}} \right] \end{array} \right] \quad \left[ \begin{array}{l} \boxed{\text{X}} \\ 1 : \left[ \text{foe} \right] \\ 2 : \left[ \text{fum} \right] \end{array} \right] \quad \left[ \begin{array}{l} \boxed{\text{Y}} \\ \boxed{\text{Y}} \\ 1 : \left[ \boxed{\{1\}} \right] \\ 2 : \left[ * : \left[ \boxed{\{1\}} \right] \right] \end{array} \right]$$

In this notation, the representation of a graph node is a set of lines enclosed in [ ] brackets.

If the node is the value or default of some variables, the names of those variables, each enclosed in a box or double box, are the top lines of the representation of the node. A

---

<sup>7</sup>This notation is standard for feature structures: e.g., see Bob Carpenter, *The Logic of Typed Feature Structures*, 1992, Cambridge University Press

variable name is boxed if the node is the value of a variable, and double boxed if the node is the default of the variable. In our example,  $\boxed{X}$  and  $\boxed{Y}$  designate nodes that are variable values and  $\boxed{\boxed{X}}$  and  $\boxed{\boxed{Y}}$  designate nodes that are variable defaults. The *root node* is treated as if it were the value of a variable named ‘{ROOT}’, and is therefore designated by  $\boxed{\{ROOT\}}$  in our example. A single node can be the value or default of several variables: in the example one node is both the value and the default of Y.

The example expression graph has two variables, X and Y. While all nodes in the expression graph must be reachable from the root or from the value or default of some variable in the graph, all nodes need not be reachable from the root, The above example is a single expression graph with a root and two variables, and not three expression graphs.

Two nodes with the same boxed name in their representation are actually the same node in the graph, and similarly two nodes with the same doubly boxed name in their representation are actually the same node in the graph. When several nodes are the same in this sense, all but one of these must have only boxed or doubly boxed names in its representation.

Sometimes it is necessary to indicate that two nodes are the same for a node that is not the root or the value or default of any variable. This is done by creating an *expression graph pseudo-variable* whose name is a word or separator in curly brackets. In the example, {1} is such a pseudo-variable name, and of course {ROOT} is a pseudo-variable name. Pseudo-variable names are always boxed, and never double boxed.

If a node has a label, that is the last line in the node’s representation, and is immediately below any boxed or doubly boxed names. There are no other lines in the representation. In the above example, 51 labels the node that is the value of the variable X.

If a node is the source of arrows in the directed graph, representations of each arrow appear below any boxed or doubly boxed variable names in the representation of the node. Each arrow representation consists of the label of the arrow followed by a colon (:) followed by a representation of the target node of the arrow. This later is of course surrounded by [] brackets.

It is permitted to omit the [] brackets when only one thing is bracketed. Thus the above example can also be rendered as:

$$\left[ \begin{array}{l} \boxed{\{ROOT\}} \\ \text{fee} : \left[ \begin{array}{l} \boxed{X} \\ 51 \end{array} \right] \\ \text{fie} : \boxed{X} \end{array} \right] \quad \left[ \begin{array}{l} \boxed{X} \\ 1 : \text{foe} \\ 2 : \text{fum} \end{array} \right] \quad \left[ \begin{array}{l} \boxed{Y} \\ \boxed{Y} \\ 1 : \boxed{\{1\}} \\ 2 : * : \boxed{\{1\}} \end{array} \right]$$



## 6.2 Pure Unification

Informally, expression graph nodes represent expressions that may be thought of as containing information, and the unification of two nodes is then a node whose information content is the least upper bound of the information content of each of the two nodes taken separately.

CASTLE uses two forms of expression graph node unification. The pure form, called '*pure unification*', is just the standard labeled graph unification algorithm, and is specified in this section. The other form, which has a special merge operation for nodes representing calls on functions or builtin CASTLE operations, is called 'call unification', and is described in 6.6<sup>p74</sup>.

Unification of two expression graph nodes either succeeds and produces a unique expression graph node as its result, or fails and produces no result.

Unification algorithms use a set called the *merge set* of node pairs called *merge pairs*. For pure unification of two nodes, X and Y, this is initialized to the set containing only the one merge pair, (X,Y).

A unification algorithm simply extracts a merge pair (M,N) from the merge set, merges the two nodes named, M and N, and does this repeatedly until the merge set is empty. The algorithm succeeds if every pair of nodes that must be merged can be merged. When two nodes are merged, they become the same node. Merging two nodes can create new merge pairs. However, merging two nodes that are identical does nothing, and since expression graph memory is finite, the algorithm must terminate after all nodes are merged, if it does not terminate before then.

In merging two nodes M and N, one of these nodes, say M, is replaced by a forwarding pointer to the other node, say N, so every reference to M becomes a reference to N. N may also be modified by giving it a label or creating arrows that it sources. And as mentioned above, new merge pairs may be created.

If a unification algorithm is unsuccessful, all changes to expression graph memory are undone.

For pure unification of two nodes X and Y, the merge set is initially the single pair (X,Y). If unification is successful, X and Y are merged, and the result of the unification is the merged node, i.e., the node X, or equivalently the node Y.

It is possible to run unification with arbitrary initial merge sets.

For pure unification the *node merge algorithm* applied to the merge pair (M,N) is as follows:

1. Remove (M,N) from the merge set.

2. If M and N are the same node, terminate the node merge algorithm.
3. If M and N are labeled nodes with different labels, the unification algorithm fails.
4. If either M or N is a labeled node and the other node is the source of some arrows, the unification algorithm fails.
5. If there is an arrow label L such that one of the nodes M or N is the source of two or more arrows labeled L and the other node is the source of one or more arrows labeled L, the unification algorithm fails. Intuitively, when we try to match the arrows sourced at one of the two nodes with arrows sourced at the other of the two nodes so that matched arrows have the same arrow label, then we must get a unique answer.
6. If M is a labeled node with label L, and N is unlabeled, label N with L.
7. For every arrow sourced at M do the following. Let the arrow be labeled L and point at target node MT. If N is not the source of an arrow labeled L, make a new arrow labeled L pointing from N to MT. But if N is the source of an arrow labeled L with target NT, add the merge pair (MT,NT) to the merge set.
8. Replace M by a forwarding pointer that points at N. This means that henceforth any attempt to reference M will be forwarded to N, so M is effectively merged with N.

Note that the running time of the pure unification algorithm is bounded by

$$T(\text{number of nodes reachable from nodes in the initial merge set}) \\ \times \\ (\text{number of arrows sourced at these nodes})^2$$

where  $T$  is a small constant time (typically a few microseconds). The total number of nodes reachable from nodes in the initial merge set bounds the number of node merge operations, and the last factor bounds the time to form the union of the arrow labels of any two nodes that are being merged. The bound just given is typically a substantial overestimate, and can be improved to:

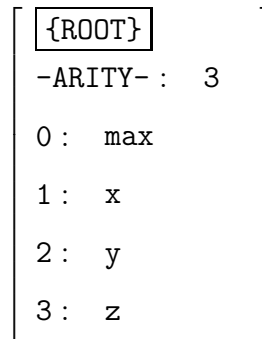
$$T \left( \frac{\text{number of nodes reachable from nodes in the initial merge set}}{\text{number of nodes reachable from these nodes after unification}} \right) \\ \times \\ (\text{maximum number of arrows sourced at any one node reachable after unification})^2$$

### 6.3 Raw Expression Trees

Raw expressions represent certain expression trees called *raw expression trees*. For example, the expression

$$[\text{max } x \ y \ z]$$

represents the raw expression tree



The raw expression is represented by the root node of a raw expression tree. The raw expression head, ‘max’, is represented by a node labeled ‘max’ that is the target of an arrow labeled ‘0’ from the root. Here ‘0’ is called the *head index* of the root node, the arrow it labels is called the *head arrow* of the root node, and the target of that arrow is called the *head* of the root node.

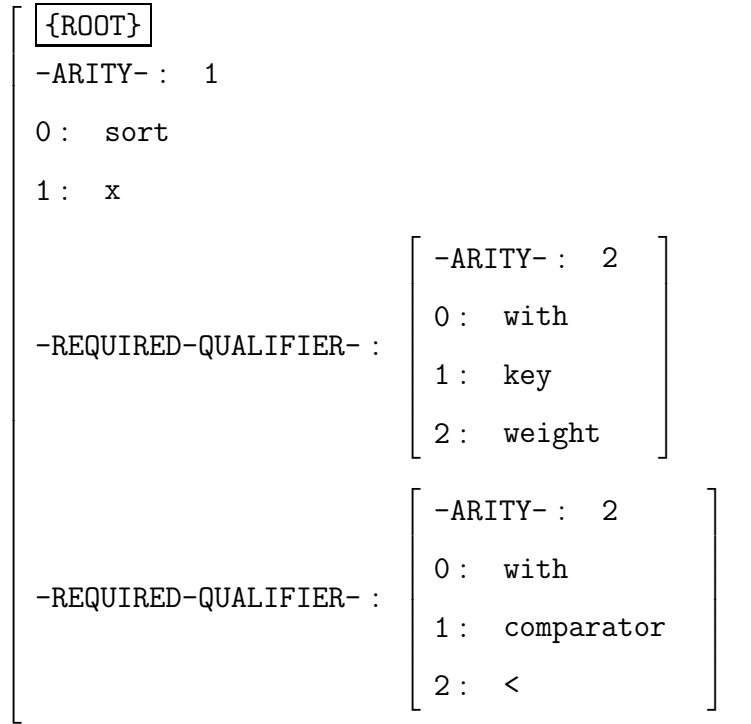
The three arguments become targets of three arrows from the root labeled ‘1’, ‘2’, and ‘3’. Each argument in this example is a single word, and is represented by a node labeled with that word. Here non-zero natural numbers are *un-reorderable argument indices* of the root that label *un-reorderable argument arrows* of the root, and the targets of these arrows are the *un-reorderable arguments* of the root. Reorderable and rest arguments are introduced below.

An arrow labeled ‘-ARITY-’ is added to the root node to point at a node labeled with the number of arguments in the raw expression. This arrow is called the *arity arrow* of the root node, its target is called the *arity target* of the root, and the label of that target is called the *arity* of the root.

A second example is the expression

$$[\text{sort } x \ @@ \ \text{with key weight } @@ \ \text{with comparator } <]$$

which is represented by the raw expression tree

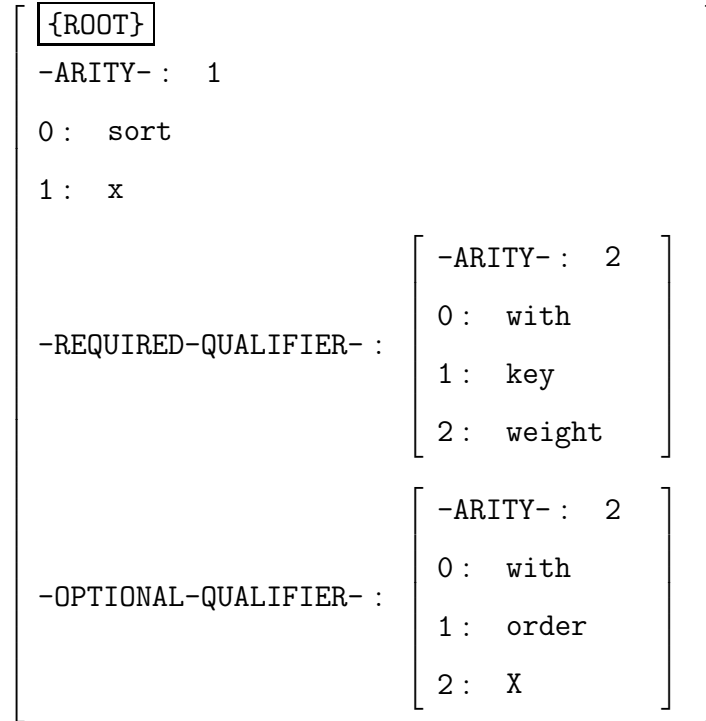


Here each qualifier becomes the target of an arrow labeled ‘**-REQUIRED-QUALIFIER-**’ because the ‘@@’ *required-qualifier-mark* is used (the arrow label would be ‘**-OPTIONAL-QUALIFIER-**’ if the ‘??’ optional-qualifier-mark had been used). The arrows with these labels are called *qualifier arrows* of the root node and their targets are called *qualifiers* of the root. In this case there are two *required qualifier arrows* and two *required qualifiers*. Each qualifier sources arity, head, and un-reorderable argument arrows analogous to those of the root node, and could source qualifier, reorderable argument (see below), and rest argument (see below) arrows.

A third example is the expression

[sort x @@ with key weight ?? with order X]

which is represented by the raw expression tree



This is similar to the last example except one qualifier is an ‘**-OPTIONAL-QUALIFIER-**’ because the ‘??’ *optional-qualifier-mark* is used. In this case there is one *required qualifier arrow*, one *required qualifier*, one *optional qualifier arrow*, and one *optional qualifier*. The only difference between required and optional qualifiers is the label on the arrow to the qualifier node, which is either ‘**-REQUIRED-QUALIFIER-**’ or ‘**-OPTIONAL-QUALIFIER-**’.

A fourth example is the expression

[fill Prefix ::? Matrix <:> Value Postfix ::> Runs]

which is represented by the raw expression tree

<div style="border: 1px solid black; display: inline-block; padding: 2px 5px;">{ROOT}</div>	
-ARITY- :	1-4
0 :	fill
1 :	Prefix
2-3 :	Matrix
2-3 :	Value
4 :	Postfix
-REST- :	Runs

Arguments appearing after the *optional argument mark*  $::?$  are optional. This is indicated by the arity ‘1-4’ that indicates that argument ‘1’ is the last required argument and there are ‘4’ arguments total. In general, the arity can have the form ‘ $m-n$ ’, where  $0 \leq m < n$ ,  $m$  is the *number of required arguments*,  $n$  is the *total number of arguments*, and  $n - m$  is the number of optional arguments.

If there are no optional arguments and the arity is just a natural number  $n$ , then  $n$  is both the number of required arguments and the total number of arguments.

Consecutive arguments separated by the *reorder mark*  $<:>$  are reorderable. These arguments are indicated by argument indices that describe the positions the arguments can occur in. Thus in the example the two arguments with index ‘2-3’ can each appear as either the 2nd or 3rd argument. In general, an argument index that is a range of the form ‘ $i-j$ ’, where  $1 \leq i < j \leq n$ , and  $n$  is the total number of arguments, indicates that the argument can be the  $i$ ’th through  $j$ ’th argument position. These argument indices are called the *reorderable argument indices* of the root node, the arrows they label are called *reorderable argument arrows* of the root, and the targets of these arrows are called *reorderable arguments* of the root.

Note that if an argument has index ‘ $i-j$ ’, then the  $i$ ’th and  $j$ ’th argument must be either both required or both optional. That is, if there are optional arguments and  $m$  is the number of required arguments, either  $j \leq m$  or  $m < i$ .

The argument after a *remainder mark*  $::>$  is given the special argument index ‘-REST-’ and names a list of all arguments that occur after the total number of arguments specified by the arity. Here -REST- is called the *rest argument index* of the root node, the arrow it labels is called the *rest argument arrow* of the root, and the target of this arrow is called the *rest argument*. The rest argument is not counted in the arity.

## 6.4 Call Nodes and the Call Check

During expression evaluation (7<sup>p80</sup>) the expression being evaluated is unified with a pattern in an expression definition. The unification used for this purpose is a variant called ‘call unification’ (6.6<sup>p74</sup>).

Call unification treats graph nodes called ‘call nodes’ differently than pure unification (6.2<sup>p65</sup>) does. A **call node** is a node that has an **arity arrow**. That is, it is the source of an arrow labeled **-ARITY-**. Call unification replaces the node merge algorithm (p65) with the call merge algorithm described below if one of the nodes being merged is a call node and the other is not a null node.

The call merge algorithm performs a subalgorithm called the **call check algorithm** that checks whether a call node is legal. Specifically, a call node N passes the call check algorithm if and only if the following are true:

1. The arity of N is either a natural number  $n$  (string of digits without high order zeros) or has the form  $m-n$  where  $m$  and  $n$  are natural numbers and  $m < n$ . In the second case  $m$  is called the **minimum arity** of N and  $n$  is called the **maximum arity** of N. In the first case  $n$  is both the minimum and the maximum arity of N.
2. N has exactly one head arrow.
3. There exists a 1-1 map between the set of un-reorderable and reorderable (but not rest) argument arrows of N and the set of natural numbers from 1 through  $n$ , where  $n$  is the maximum arity of N, such that if  $i$  is a natural number from 1 through  $n$ , the arrow mapped to  $i$  is either an un-reorderable argument arrow labeled  $i$ , or is a reorderable argument arrow labeled  $j-k$  where  $j \leq i \leq k$ .

Such a mapping is called a **argument order assignment** of N. If there are any reorderable arguments of N, there will be more than one argument order assignment of N.

4. N has at most one rest argument arrow. If N has a rest argument arrow, the rest argument of N is a null node (a node with no label that is the source of no arrows).
5. N can have any number of qualifier arrows (required or optional).
6. There are no arrows sourced at N other than those with labels enumerated above. That is, any arrow sourced at N is either an arity arrow, a head arrow, an argument arrow (un-reorderable, reorderable, or rest), or a qualifier arrow (required or optional).

7. The qualifiers of N (targets of qualifier arrows) pass the call check algorithm recursively. The call check algorithm allows cycles in the graph, e.g. the case that N and one of its qualifiers are the same node.

Note that raw expressions cannot represent call nodes that have qualifiers which in turn have qualifiers, but the conversions of 6.7<sup>p76</sup> can represent such call nodes. Because of the difficulty of representing them, such call nodes are of mostly theoretical interest.

## 6.5 Paths and Witnesses

A *non-empty path* is a sequence of one or more arrows in some expression graph, such that if A1 and A2 are consecutive arrows in the sequence, the target of A1 is the source of A2. The *path name* of the path is the sequence of words and separators that are the arrow labels of the arrows of the path. The source node of the first arrow in the path is the *source* of the path. The target node of the last arrow in the path is the *target* of the path. Note that the path name and source node of a path are together not adequate to determine the path in all cases, since we allow a node to source several arrows with the same label.

An *empty path* is just a node. This node is both the *source* of the empty path and the *target* of the empty path. The *path name* of the path is the empty sequence.

A *path name* is just a possibly empty sequence of words or separators. A *node label* is just any word or separator.

A *witness* is a path name and a node label. We use ‘*path-name: node-label*’ to name the witness. A *witness* of a node S in an expression graph is a witness P: L with path name P and node label L such that there is a path with source S, path name P, and target that has node label L.

For example, consider the expression graph of

```
[sort x @@ with key weight @@ with comparator <]
```

on p68. The root of this expression graph has ‘0: sort’ as one of its witnesses and ‘-REQUIRED-QUALIFIER- 0: with’ as another. There are actually two distinct paths corresponding to this second witness.

Two witnesses are said to be *incompatible* if they have the same path name but have different labels. Two witnesses are said to be *compatible* if they are not incompatible: that is, if they have different path names, or if they have both the same path names and the same



labels. For example, ‘0: sort’ and ‘1: x’ are compatible while ‘0: sort’ and ‘0: switch’ are incompatible.

It is not true that all witnesses for a node in an expression graph are compatible. This is because we allow one node to source several arrows with the same label. In the example expression graph on p68, ‘-REQUIRED-QUALIFIER- 1: key’ and ‘-REQUIRED-QUALIFIER- 1: comparator’ are incompatible witnesses for the value of {ROOT}. However, for an expression graph in which no node sources two arrows with the same arrow label, it can be easily proved that all witnesses of every node are compatible.

Two expression graph nodes, S1 and S2, possibly the same node, and possibly in different expression graphs, are said to be *witness compatible* if and only if every witness for S1 is compatible with every witness for S2. Otherwise S1 and S2 are said to be *witness incompatible*.

For an integer  $N \geq 0$ , an N-witness is a witness whose path name has at most N arrow labels. Two expression graph nodes, S1 and S2, are said to be *N-witness compatible* if and only if every N-witness for S1 is compatible with every N-witness for S2. Otherwise S1 and S2 are said to be *N-witness incompatible*.

It is possible for a node to be 1-witness incompatible with itself, without being witness incompatible with many other nodes. For example, a node whose only witnesses are the incompatible witnesses ‘1: fee’ and ‘1: fie’ would be witness compatible with any node that was not the source of any arrow with arrow label ‘1’.

It can be easily proved that if two expression graphs can be unified by pure unification (6.2<sup>p65</sup>), then any two nodes that are merged during unification are 1-witness compatible. This is because step 5<sup>p66</sup> of the node merge algorithm (p65) means that if the nodes have incompatible 1-witnesses then the nodes each source only one arrow with the witness arrow label, and therefore the targets of these two arrows must merge, which is forbidden by step 3<sup>p66</sup> as according to the incompatible 1-witnesses the targets have unequal labels.

It can even be proved that if two expression graphs can be unified by pure unification (6.2<sup>p65</sup>), then any two nodes that are merged during unification are witness compatible. This is because if there are two incompatible witnesses of the two nodes, there would be two paths with the same path name, one for each witness, with consecutive nodes on the two paths being merged, because step 5<sup>p66</sup> requires that along each of the two paths each node sources exactly one arrow with the next arrow label on the paths. The two paths would end at two nodes that would have different labels, but would have to be merged, causing the unification algorithm to fail by step 3<sup>p66</sup>.

In the unification algorithm, arrow labels are used to match arrows to be merged. More

specifically, when two nodes M and N are merged, arrows sourced at M are matched with arrows sourced at N by arrow label. If this match cannot be done, step 5<sup>p66</sup> of the unification algorithm causes unification to fail. In order to merge qualifier nodes, we need to avoid this failure, and we do this in 6.6<sup>p74</sup> by using some of the the 1-witnesses of the qualifier nodes to make the match. The 1-witnesses we use include the qualifier head, if that is a node with a label, and any un-reorderable non-rest qualifier argument that is simply a node with a label.

## 6.6 Call Unification

**Call unification** is a variant of pure unification (6.2<sup>p65</sup>) that differs from pure unification when merging nodes one of which is a call node and the other of which is not a null node (node with no label that sources no arrows). Call unification is used in the expression evaluation algorithm (7.2<sup>p82</sup>).

Call unification is identical to pure unification (6.2<sup>p65</sup>) with two differences. The first difference is that when one of two nodes being merged is a call node (6.4<sup>p71</sup>), and the other node is not a null node, call unification replaces the node merge algorithm (p65) with the call merge algorithm described below.

The second difference is that call unification makes choices during its execution, because the call merge algorithm makes choices. Each set of possible choices over the whole call unification algorithm leads to a separate algorithm execution that either succeeds or fails.

The **call merge algorithm** for a merge pair (M,N) in the merge set, where either M or N or both is a call node, and neither is a null node, with a particular choice, is as follows.

1. Remove (M,N) from the merge set.
2. If either M or N fails to pass a call check, the unification algorithm fails.
3. If M and N are the same node, terminate the call merge algorithm.
4. A call node is said to be **indeterminate** if it has an optional, reorderable, or rest argument or an optional qualifier. Otherwise the node is said to be **determinate**.

If both M and N are indeterminate, the unification algorithm fails.

Otherwise, if just N is indeterminate, switch M and N so that N is always determinate.

Later in this call merge algorithm M will be replaced by a forwarding pointer to N, and N will not be given any optional, rest, or reorderable arguments, so the result of the merge will be determinate.

5. If M is determinate and the arities of M and N are not equal, the unification algorithm fails.  
If M has more required arguments than N has arguments, the unification algorithm fails.
6. For the rest of this algorithm, let MT be the total number of arguments (p70) of M and NT be the total number of arguments of N.
7. If M has a rest argument R, then For each natural number I from MT+1 through NT, an arrow labeled I-MT is added that has R as its source and the I'th argument of N as its target. Note that before this is done R is a null node (step 4<sup>p71</sup> of the call check algorithm on p71). If NT ≤ MT, this step does nothing (R remains a null node).
8. Choose an argument order assignment (p71) for M. For 1 ≤ I ≤ MT, let MI let the assigned I'th argument of M.
9. Let T be the minimum of MT and NT. For 1 ≤ I ≤ T, let NI be the I'th argument of N, and add the merge pair (MI,NI) to the merge set. For T < I ≤ MT (if T < MT), add an arrow labeled I from N to MI.
10. Let MH be the head of M and NH be the head of N. Add a merge pair (MH,NH) to the merge set.
11. If there is any qualifier arrow of M whose target is 1-witness compatible with the target of more than one qualifier arrow of N, the unification algorithm fails. Ditto with M and N switched: if there is any qualifier arrow of N whose target is 1-witness compatible with the target of more than one qualifier arrow of M, the unification algorithm fails.
12. For each qualifier arrow MA of M whose target is 1-witness compatible with the target of exactly one qualifier arrow NA of N, let MQ be the target of MA and NQ be the target of NA. Add (MQ,NQ) to the merge set.
13. For each qualifier arrow MA of M whose target is 1-witness compatible with the targets of no qualifier arrows N, let MQ be the target of MA, and make an arrow from N to MQ that has the arrow label -REQUIRED-QUALIFIER.
14. Replace M by a forwarding pointer that points at N. This means that henceforth any attempt to reference M will be forwarded to N, so M is effectively merged with N.

## 6.7 Expression Graph Representation

Raw expressions can be used to represent expression graphs. This is done by defining *graph creation conversions* that convert raw expression trees to expression graphs.

A graph creation conversion is performed by first converting the raw expression to an expression tree plus a set of variables where each variable has not a single value or single optional default, but instead has a set of one or more values and zero or more defaults. Then all the values of each variable are unified using pure unification, and all the defaults of each variable with defaults are similarly unified. The result after unification is an expression graph.

There are two kinds of graph creation conversion: explicit and implicit. In implicit conversion words beginning with a capital letter or with ‘#’ followed by a capital letter are taken to be variable names if they appear where a node label would appear. In explicit conversion such words are not given special treatment.

In converting a raw expression to an expression tree, the rules for representing raw expressions as expression trees (6.3<sup>p67</sup>) are followed, with the following exceptions:

[// *variable-name variable-value*]

This expression converts to the same tree that the *variable-value* converts to, and adds the root node of this tree to the value set of the named variable.

[// *variable-name*]

This expression converts to a tree consisting of a single null root node, and adds this node to the value set of the named variable.

*variable-name*

For implicit conversion this is treated as the equivalent of ‘[// *variable-name*]’ if *variable-name* begins with a capital letter or with ‘#’ followed by a capital letter. Otherwise this converts to a tree with a single node labeled with a word that is not in general a variable name.

[~~ *variable-name variable-default*]

This expression converts to the same tree that the *variable-default* converts to, and adds the root node of this tree to the default set of the named variable. Thus ~~ is like // except that the node is added to the default set instead of the value set of the variable.

[#### { *arrow-label target-value* }\*]

This expression converts to a tree whose root is an unlabeled node that is the source of arrows labeled with the *arrow-labels*. The target of each arrow is the tree obtained by converting the *target-value* raw expression that follows the arrow’s

*arrow-label.*

[++++ *expression-graph expression-graph*\*

This expression converts to the tree that the first *expression-graph* converts to. The other *expression-graphs* are also converted to trees. All the *expression-graphs* may contribute variables, variable values, and variable defaults, and the values and defaults for each variable will be merged by the unification at the end of the conversion.

After the expression tree is created, its root node becomes the root of the expression graph resulting from conversion.

Then pure unification is performed as described in 6.2<sup>p65</sup>, except that the merge set is initialized to merge pairs that merge the values of each variable and the defaults of each variable. It is a conversion error if this unification fails or if any variable has an empty set of values before unification is attempted. After unification a variable has a default if and only if its set of default values before unification was non-empty.

There are two kinds of graph creation conversions: explicit and implicit. In ***implicit conversions*** what appear to be nodes labeled by a word L beginning with a capital letter or with ‘#’ followed by a capital letter are treated as the equivalent of ‘[// L]’. Thus L is a variable name, and when used like a node label in the raw expression, L represents a null node that is a value of the variable named L. For ***explicit conversions*** no such special provision is made for words that are used like node labels.

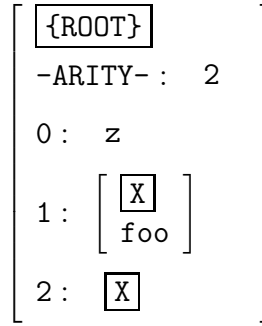
The exact operation names used in graph creation conversions may be different from those given above. The names to be used are given as arguments to the conversion operations, with the names used above as defaults. These default names and the descriptive names of the operations are:

//	<b><i>graph variable value operation</i></b>
~~	<b><i>graph variable default operation</i></b>
####	<b><i>graph source construction operation</i></b>
++++	<b><i>graph concatenation operation</i></b>

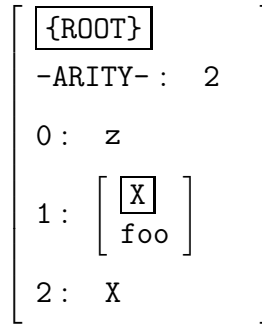
***Pseudo-variable*** names (p64) of the form ‘{word-or-separator}’ can be used with the ‘//’ operation to merge nodes that are not the value or default of any real variable. This use simply creates variables with names that are not single words or separators, and these variables are discarded at the end of conversion. The pseudo-variable ‘{ROOT}’ is always given the root node of the expression tree as one of its values, but may be given other values by this method. Pseudo-variables cannot be given defaults.

Some examples of expression trees represented by raw expressions and the expression graphs they convert to are as follows:

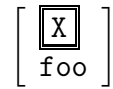
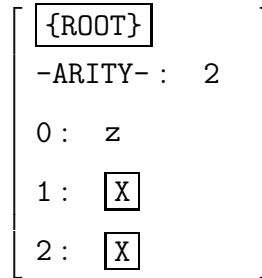
[z [/ X foo] X]  
implicit conversion

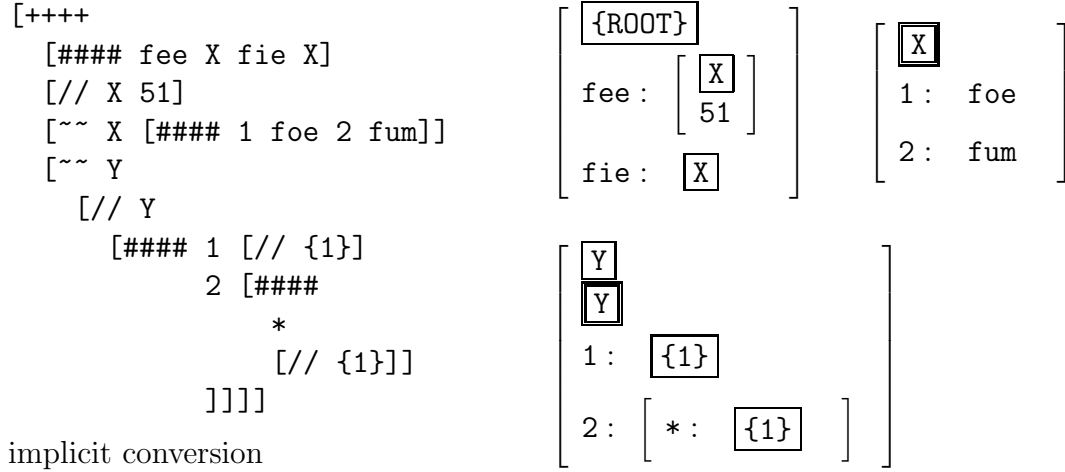


[z [/ X foo] X]  
explicit conversion



[++++  
[z X X]  
[~~ X foo]]  
implicit conversion





## 6.8 Expression Graph Implementation

In this section we give ideas that may or may not be used to implement CASTLE expression graphs.

CASTLE may have a single memory of nodes and arrows that holds all expression graphs. We will call this the expression graph memory.

There may be at most one node in expression graph memory that has a given label  $L$ . That is, all nodes labeled  $L$ , for a particular  $L$ , are the same node.

Giving a null node a label  $L$  is then implemented by forwarding the null node to the node with label  $L$ .

A frequent operation is copying expression graphs as part of expression evaluation (7.2<sup>p82</sup>). To make this more efficient, an expression graph can be separated into two parts: an immutable part, and a mutable part. The mutable part is just a vector of node references, one per node in the expression graph. The immutable part is just a set of immutable nodes, with associated arrows, except that arrow targets may be indices of node references in the mutable vector. Thus there are now two kinds of nodes in memory: normal nodes, which include those that are labeled and those that are mutable in the normal way, and immutable nodes. The latter can only be pointed at by a node reference in a mutable vector.

Node references in mutable vectors are like nodes. They can be null, forwarded, or point at an immutable node associated with the mutable vector. In the last case, the node reference and immutable node together behave like a mutable node, and we say the node reference is ‘*coupled*’ to the immutable node.

During unification arrows are sometimes added to a node. To make this work for a node reference coupled to an immutable node, a node reference may also point to a structure that stores a pointer to an immutable node plus a list of extra arrows added to that node. The node reference is still coupled to the immutable node, but the node reference now represents a new node that has more arrows than the immutable node.

An arrow target can be a normal node, that is, either a labeled node or a normal mutable node. An arrow target may not be an immutable node unless that node is never forwarded and arrows are never added to the node. Instead the arrow target be a node reference coupled to the immutable node.

In CASTLE things can be arranged so the nodes to be evaluated, which are guard nodes in expression definitions or nodes in expression definition blocks, are never forwarded. They are determinant (p74), and in any call unification at most one node is being evaluated, while the other is the expression defined by an expression definition. However, in CASTLE arrows may be added to an expression being evaluated (Step 13 p75). So in CASTLE probably no immutable node can point directly at another immutable node.<sup>8</sup> Luckily CASTLE is not intended to run large programs. However, it should be possible to compile code that will check whether an expression defined by a given definition unifies with an arbitrary expression to be evaluated. Call unification is just a specification for a kind of pattern matching, and pattern matching specifications can be compiled into classical imperative code.

## 7 Expression Evaluation

An expression is evaluated by first searching for an expression definition that matches the expression. Expression definitions also have guards, which are expressions that must evaluate to true in order for the expression definition match to succeed. 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. There is also a global context to which definitions may be added or from which they may be deleted. The block of an expression definition

---

<sup>8</sup>CASTLE differs from PROLOG in that in PROLOG arrows are not added to nodes during unification so immutable guard nodes can point directly at other immutable guard nodes.



usually evaluates expressions in the lexical context of its expression definition, but may evaluate expressions in the lexical context of the expression being evaluated by the expression definition.

When an expression definition is found that matches an expression, a copy of the expression definition is made, and a part of this copy called the pattern is unified with the matched expression. This unification defines variables in the expression definition and null nodes in the matched expression that are values of variables outside the expression definition. Expression definition variables that receive no value in this way, but which have default values, are given the default values. However, values received from unification may be incomplete, in that they may be expressions which include null nodes that represent unknown values of other variables. When the guards of the expression definition are matched in turn with other expression definitions, and unified with patterns in copies of these matched definitions, the incomplete values may become complete. For this reason, matching of guards with definitions and unification of guards is done as much as possible before any blocks are evaluated.

The expression being evaluated may also have null nodes that represent unknown values of variables. During evaluation these null nodes may be unified with patterns in expression definitions, and cease to be null nodes. This process is known as ‘*completing the expression*’ being evaluated. An expression with no null nodes is said to be *complete*, whereas an expression with null nodes is said to be *incomplete*.

After an expression definition pattern is unified with an expression the definition matches, and after default values are assigned, some variable values are evaluated and replaced by their evaluated values. These variables are called evaluated variables, and typically have names beginning with a capital letter, whereas other variables are called unevaluated variables, and typically have names beginning with ‘#’ followed by a capital letter. Evaluation of evaluated variable values happens before guard expressions are matched to definitions, so guard expressions see only the evaluated value of evaluated variables. The values which are evaluated in this manner must not be incomplete, and their evaluation must have no visible side effects.

After all guards are matched with definitions and unified, blocks are evaluated, beginning with any blocks in the definitions matched to guards. If any of these guard blocks fails to produce the value true, the expression definition match fails, and the search for other matching definitions continues. Evaluations of guard blocks are required to have no visible side effects.

## 7.1 Expression Definitions

*Expression definitions* have the syntax:

**EXPRESSION-DEFINITION** ::= *pattern* <-- *guard-list-option* *block-option*  
**pattern** ::= *EXPRESSION*  
**guard-list-option** ::= *empty* | *guard-list*  
**guard-list** ::= *guard* { , *guard* }<sup>\*</sup>  
**guard** ::= *EXPRESSION*  
**block-option** ::= *empty* | *block*  
**block** ::= { *statement* { ; *statement* }<sup>\*</sup> }

*Blocks* and *statements* are further defined in 8<sup>p87</sup>.

In use, an expression definition being matched to an expression is copied, and then the pattern of the copy is call unified (6.6<sup>p74</sup>) with the expression to be matched. If unification fails, the definition does not match. If unification succeeds, variables bound to null values are given their default values, evaluated variable values are evaluated, and then the matching process continues by searching for definitions that match the guards.

An expression definition is an expression graph which may have variables as well as graph nodes. When an expression definition is copied, the copy has its own variables that are distinct from the variables in the original expression definition or in any other expression definition copy.

## 7.2 The Evaluation Algorithm

The *evaluation algorithm* inputs an expression to be evaluated and a context. The context (7.3<sup>p85</sup>) provides a list of expression definitions that may match the expression.

An expression to be evaluated may contain variables that are assigned values during matching. The result of evaluation is both a value for the expression and an assignment of values for these variables.

More than one definition may match an expression. More than one definition may match a guard of a definition, and different guard definitions may lead to different completions of values of variables in the expression being evaluated. In matching a definition, more than one argument order assignment (p71) may allow the definition to be matched. So expression

evaluation is a search process to find a choice of expression definitions and argument order assignments which leads to success, and more than one choice may succeed.

Note that the guards of an expression are always matched from left to right, so order of guard matching is not a choice. Similarly guard block evaluation order is fixed, left to right, depth first, so this is not a choice. Lastly evaluation of evaluated variable values and of guard blocks are required to have no side effects, so these can be undone trivially when part of the search fails.

Evaluation may be done in any of the following modes:

**first-value**

The first definitions in contexts and first argument order assignments tried that lead to successful matches are the only ones used. Definitions are tried in the order they are given in the contexts used in evaluation. The order in which argument order assignments are tried is implementation dependent.

**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 null node in that expression.

**consistent-values**

All possible choices are tried and the successful results are collected in a set of values. 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 null node contained in the expression must be equal. If all values of the expression or of a null node are equal, one of these values is returned as the value of the expression or null node. If some of the values are unequal, an error value giving the context of the evaluation and the unequal values is returned as the value of the expression or null node.

**consistent-reordering**

Definitions are tried in the order they are given in the contexts used in evaluation, and the first definitions that match are used. However, for each definition all possible argument order assignments are tried. The results are collected in a set of results that is tested for pairwise equality. All the values of the expression being evaluated must be equal, and all the values of each null node contained in the expression must be equal. If all values of the expression or of a null node are equal, one of these values is returned as the value of the expression or null node. If some of the values are unequal, an error

value giving the context of the evaluation and the unequal values is returned as the value of the expression or null node.

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 changed.

In the following description of algorithms we use a ‘***choose operation***’ as if there were an oracle that allowed us to make appropriate choices. In actuality a search process is used, as indicated above.

Evaluating an expression uses subalgorithms for matching an expression with a definition and for evaluating a block.

The following subalgorithm is used to match an expression to be matched with an expression definition.

1. Make a copy of the expression definition. Variables in this copy are distinct from other variables of the same name in memory.
2. If there are reorderable arguments in the expression definition pattern, choose an argument order.
3. Call unify (6.6<sup>p74</sup>) the pattern (7.1<sup>p82</sup>) from the definition copy with the expression to be matched, using the chosen argument order if any. If the unification fails, the definition does not match. If the unification succeeds, it provides values for the variables in the pattern and for null nodes in the matched expression.
4. For any pattern variable bound to a null node, rebind the variable to its default value, if the variable has a default value.
5. For any pattern variable whose name does not begin with ‘#’, evaluate the value of that variable to obtain a new variable value. Then simultaneously replace all the evaluated variable values by their new values. Each replacement is done by forwarding a variable value graph node to the new value graph node.

These variable value evaluations are not permitted to have visible side effects (13<sup>p99</sup>).

The subalgorithm for evaluating a block is given in 8.3<sup>p90</sup>.

Evaluating an expression is as follows:

1. Choose an expression definition in the appropriate context that matches the expression to be evaluated, and match the definition with the expression.

Note the evaluated variable value evaluations of 5 above are done before guard expression matching is done.

2. For each guard of the expression definition, from left to right, choose an expression definition in the appropriate context that matches the guard, and match the definition with the guard.
3. For each guard that has a block, evaluate the block, working in left to right depth first order of guards. If a guard block evaluates to something other than ‘true’, declare the choices made so far to be a failure, and continue the search. These guard block evaluations are not permitted to have visible side effects (13<sup>p99</sup>).
4. If the definition matched to the expression to be evaluated has no block, the value of the expression is ‘true’. If it has a block, the block is evaluated to produce a value and possibly side effects (8.3<sup>p90</sup>).

It is always an error if a null node is to be evaluated. In particular, the value of an evaluated variable just before the value is to be evaluated must not be a null node.

### 7.3 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. **Context sets** are searched exhaustively, and it is always an error if matching definitions are found in more than one element of a set.

An expression definition is an object (9<sup>p94</sup>) of **expression-definition** type that consists of a pattern, a list of guards, an optional block, and an optional context. An *EXPRESSION-DEFINITION* expression (p82) computes an expression definition that has no context. A context can be added to the expression definition later by executing:

```
set EXPRESSION-DEFINITION context CONTEXT
```

Once a context has been added to an expression definition, the context cannot be changed.

A new expression definition can be made from an old one by the expression:

**an expression-definition** *EXPRESSION-DEFINITION* *CONTEXT*

The new expression definition has a different context, given by the second argument, than the first argument does. The special value **UNDEFINED** can be given as the *CONTEXT* argument to create a new expression definition whose pattern, guards, and block are the same as those of *EXPRESSION-DEFINITION* argument but whose context part is missing and can be set later.

There are two kinds of contexts: *immutable* and *mutable*. An immutable context cannot be modified. A mutable context can have its list (or set) modified. Each kind can be either a set or a list.

A context can be created by:

```
a context ( CONTEXT-ELEMENT { , CONTEXT-ELEMENT } )
    [ is list LIST
      | is mutable MUTABLE
      | with contexts missing CONTEXTS-MISSING ]1
```

in which each *CONTEXT-ELEMENT* is either an expression definition or a context, and *LIST*, *MUTABLE*, and *CONTEXTS-MISSING* are either ‘**false**’ (the default) or ‘**true**’. If *CONTEXTS-MISSING* is **false** (the default), any expression definition *CONTEXT-ELEMENTS* that do not have their contexts set will have them set equal to the context being created.

Thus given expression definitions that do not have their contexts set, the default is to create an immutable context set whose expression definitions have the new context and therefore can reference each other. Blocks containing *EXPRESSION-DEFINITIONs* perform exactly this kind of context creation.

The *CONTEXT-ITEM* list of a context object is a list object (p99) which can be extracted from the context object by

**the list of** *CONTEXT*

This list object is a list or set according to whether or not the context is a list or set, and the list object is immutable or mutable according to whether or not the context object is immutable or mutable. If mutable, altering the list object will alter its containing context object, and this is in fact the only way to mutate a mutable context.

The above expressions are defined by the expression definitions (7.1<sup>p82</sup>):

```

an expression-definition {
  on an expression-definition EXPRESSION-DEFINITION CONTEXT
    <-- expression-definition EXPRESSION-DEFINITION,
      context CONTEXT
  on set SELF context CONTEXT <-- context CONTEXT
}
a context {
  on a context CONTEXT-ELEMENT-LIST
    ?? is list ( LIST ~ false )
    ?? is mutable ( MUTABLE ~ false )
    ?? with context missing ( CONTEXT-MISSING ~ false )
    <-- list CONTEXT-ELEMENT-LIST of context-items
}
context-item X <-- expression-definition X
context-item X <-- context X

```

## 8 Blocks

A block is a set of variables, values for some of the variables, and code for computing these values.

### 8.1 Block Syntax

The code of a block has the following syntax:

```

block ::= { group { sequence-break group }* }
sequence-break ::= ----* (1 word consisting of 3 or more -'s)
group ::= declaration-group | statement-group
declaration-group ::= declaration { ; declaration }*
statement-group ::= statement { ; statement }*
declaration ::= expression-definition
                  | method-definition
                  | empty
expression-definition ::= EXPRESSION-DEFINITION (see p82)

```

*method-definition* ::= **on** *EXPRESSION-DEFINITION*  
*statement* ::= *qualified-statement* | *empty*  
*qualified-statement* ::= *assignment-statement*  
                               | *unguarded-subblock*  
                               | *guarded-statement*  
                               | *statement-qualifier qualified-statement*  
*statement-qualifier* ::= **first** | **always** | **default**  
*assignment-statement* ::= *variable-assignment-statement*  
                               | *pattern-assignment-statement*  
*variable-assignment-statement* ::= *output-variable* = *right-side*  
*right-side* ::= *right-side-expression*  
                   | *right-side-expression* => *output-variable* { , *output-variable* } \*  
*right-side-expression* ::= *EXPRESSION*  
*output-variable* ::= *variable-name* | *next-variable*  
*input-variable* ::= *variable-name* | *next-variable*  
*next-variable* ::= **next** ( *variable-name* )  
*variable-name* ::= *word*  
*pattern-assignment-statement* ::= *assignment-pattern* ~~ *right-side*  
*assignment-pattern* ::= *EXPRESSION*  
*unguarded-subblock* ::= { *statement-group* }  
*guarded-statement* ::= *if-statement* | *when-statement*  
*if-statement* ::= **if** *guard* *guarded-subblock*  
                       *else-if-continuation* \*  
                       *else-continuation-option*  
*else-if-continuation* ::= **; else if** *guard* *guarded-subblock*  
*else-continuation-option* ::= *empty* | **; else** *guard* *guarded-subblock*  
*when-statement* ::= **when** *guard* *guarded-subblock*  
*guard* ::= *right-side*  
*guarded-subblock* ::= { *statement-group* }



A block is parsed when the statement that contains it (typically an *EXPRESSION-DEFINITION*, p82) is parsed. Because the ; operator used as a statement separator in a *block* is by default **definitional** (p47), in addition to *statements* a *block* can contain definitions that control parsing but do not become part of the parsed block. Such definitions are omitted from the above syntax.

When the pattern and guard parts of an expression definition are parsed, they are parsed together to create an expression graph, and implicit conversion (p77) of variable names is used to produce variables for that graph. Any *block* part of the expression definition is parsed separately to produce a separate expression graph that has no expression graph variables (it does have variables in a different sense: see 8.2<sup>p89</sup>). Implicit conversion is not used for any part of *block* parsing, except for *assignment-patterns*, *expression-definitions*, and *method-definitions*, each of which becomes a separate expression graph with its own separate expression graph variables [TBD: how is this represented and done?].

## 8.2 Block Variable Names

After parsing, the *variable-names* in the code of a block are identified. Each *variable-name* is a subexpression consisting of a single *word*, but not all such subexpressions are *variable-names*. Some *variable-names* may be **inherited** from a statement containing the block. Some are *output-variables* in *assignment-statements* or *guards*. Some are expression graph variables of *assignment-patterns*. All the *variable-names* in the block are inherited or can be identified by looking at the left sides of *assignment-statements* or at the *output-variables* following => in *assignment-statements* or *guards*.

When identifying *variable-names* in a *block*, *statements* in a *subblock* of the *block* are treated as if they were in the *block* proper. Thus variables named in the *subblock* are variables named in the *block*.<sup>9</sup>

If a *block* is part of an *EXPRESSION-DEFINITION* (p82), the names of expression graph variables of the *pattern* and *guard-list* expression tree become *variable-names* inherited by the *block*. Thus in

```
sum from X through Y <-- integer X, integer Y block
```

the variable names X and Y are inherited by the *block*.

An *assignment-pattern* is parsed to an expression graph with its own expression graph vari-

---

<sup>9</sup>CASTLE does not have the notion of name space nesting, because the {} brackets that would indicate nesting can be implied by the : construction.

ables. These variables are called **pattern variables**, and their names become *variable-names* of the *block* containing the *assignment-pattern*. Thus in the statement

$$X + Y \sim\sim '5 + ( 7 * y )'$$

the variable names *X* and *Y* are pattern variable names that are *variable-names* of the *block* containing this *pattern-assignment-statement*.

The variable named before the = in a *variable-assignment-statement* is an *output-variable*. Variables named after => following an expression in a *right-side* are also *output-variables*. In the statements

```
when sort x to y => y:
    z = first
```

*y* and *z* are *output-variables*.

From the point of view of block evaluation there is no distinction between variable names beginning with # and other variable names. The only distinction occurs in *EXPRESSION-DEFINITION patterns*, where the values obtained by call unification for input variables whose names do not begin with # are replaced by their evaluations (p84).

TBD: Arrays and array element names.

### 8.3 Block Evaluation

**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. However, evaluation within a **statement-group** is driven by availability of variable values, and not by the order of the *statements* in the group.

Each variable in a block 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.

The **input-variables** of a *right-side-expression* are all variables in the expression that are not *output-variables* or *pattern-variables* of the *assignment-statement* or *guard* containing the expression. Each *right-side-expression* in a *statement-group* is evaluated only when all its input variables have values. When the *right-side-expression* of an *assignment-statement* is evaluated, the variables named on the left side of the statement and any *output-variables* named after a => in the *right-side* are given values. When the *right-side-expression* of a *guard* is evaluated, any *output-variables* named after a => in the *guard* are given values.

None of the statements in an *if-statement guarded-block* are evaluated until the block's *guard* has been evaluated to true. If the guard evaluates to false, no statement in the *guarded-block* is ever evaluated. An *else-if-continuation guard* is not evaluated until all *guards* in any preceding *if-statement* or *else-if-continuations* in the same *guard-statement* have been evaluated to false. The *guarded-block* of an *else-continuation* is evaluated if and only if all *guards* in any preceding *if-statement* or *else-if-continuation* in the same *guard-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.

A *when-statement* is just like an *if-statement* that has no *else-if-continuation* or *else-continuation*. [TBD: what about making a variable true if it has ANY value, for the purposes of a *when-statement*?]

A *statement-group* terminates when all the *right-side-expressions* in it 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, group evaluation is additionally modified by *statement-qualifiers* (8.4<sup>p92</sup>, 8.5<sup>p93</sup>).

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. There are no *guards* or block variables in a declaration group (the variables of a definition are not variables of the block).

**Empty-statements** and **empty-declarations** do nothing when they evaluate.

The *groups* of a *block* are evaluated in order. 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.

When a block terminates, the *value* of the block is the value of its **value** variable, if any. Thus the 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 finishes without producing a value for a variable named **value**, the value of the block is **true**.

Block evaluation is additionally modified by *statement-qualifiers* and *next-variables* (8.4<sup>p92</sup> and 8.5<sup>p93</sup>).

## 8.4 Default Statements

The **default** *statement-qualifier* modifies group execution. A statement qualified by **default** is called a **default statement**. Default statements in a *group* are initially inactive, meaning that they are treated as if they do not exist,

After the *group* without its default statements finishes executing, any default statement in the *group* that has no output or pattern variable with a value is made active. All default statements that can be made active in this way are made active at the same time. Then the *group*, which now consists of all non-default statements and all active default statements, continues executing, until no more non-default or active default statement can be executed.

Since all default statements that become active in a group become active at the same time, it is possible for an error to occur if two such statements assign values the same variable.

An example application of these rules is:

```
{
  if X > 5:
    value = 'true'
  default value = 'false'
}
```

where  $X$  is an inherited variable name. If  $X > 5$  is true, the value of the block is ‘true’ and the **default** *assignment-statement* remains inactive, but if  $X > 5$  is false, the **default** *assignment-statement* becomes active and the value of the block is ‘false’.

## 8.5 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 does not produce a value of a **value** variable and if it does produce a value for a *next-variable*. The sequence of blocks are called the *iterations* of the sequence. 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.

A *next-variable* is named by a *variable-name* that is the operand of a **next** unary operator. A *next-variable* is effectively a new variable with a name derived from the *variable-name*. We will use 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*, and the current iteration must not compute the value of the **value** variable.

*Next-variables* can be both output and input variables. When used as input variables they are just like other input variable. The **next** 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** operator had instead been applied 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.

A statement of the form

$$\text{default next}(V) = V$$

is implied in the last group of a block for every inherited *variable-name*  $V$  provided no other default statement in the last group of the block outputs **next**( $V$ ). Thus inherited variables are normally propagated unchanged from one iteration to the next.

If a *next-variable* is given the value of the expression ‘**UNDEFINED**’, the variable will become undefined at the beginning of the next iteration. Thus the statement

`default next(V) = UNDEFINED`

for an inherited *variable-name*  $V$  will keep the inherited variable from being propagated from one iteration to the next. 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 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.

## 9 Objects

An object is a typed block that can be used as a value. A typed block, or object, has a type and inherits code from its type.

An *object* can be created by an *OBJECT* expression:

**OBJECT** ::= { **a** | **an** } *type-name block-option*  
*type-name* ::= *word*

The code in the *block* becomes a permanent part of the object. The variable values defined by this code become **components** of the object. These components can be retrieved by expressions of the form:

**the** *variable-name* **of** *object*

Objects are updated by adding code to them either permanently or temporarily. Whenever code is added to an object, the code executes as much as it can to define variable values that are then components of the object. When the code can execute no further (8.3<sup>p90</sup>, 8.4<sup>p92</sup>, 8.5<sup>p93</sup>), any variables that do not have values become undefined components. If some

of undefined components are given values later, any code that is a permanent part of the object may restart and produce additional variable values that define additional components.

Recall that the concept of being a variable of a block of code is defined syntactically and independently of which variable values the block actually computes (8.2<sup>p89</sup>). The components of an object are more precisely the syntactic variables of the code added permanently to the object, whether or not these variables have values defined by code added permanently or temporarily to the object. Variables of code added temporarily that are not syntactically variables of code added permanently are not components of the object, and are not visible outside the temporarily added code.

TBD: need a way to declare an object component without otherwise referencing it in permanent code.

The permanent code of an object all belongs to the same group. Blocks of permanent code cannot have *sequence-breaks*, and therefore have only one *group*. Code that is added temporarily to an object can have several groups, and the permanent code of the object behaves as if it were a separate group executed after the temporary code is executed.

A statement of the form:

**for every** *type-name block*

adds the code in its block permanently to every object of the given type, including both existing objects and objects that are created in the future. The code in the block is associated with the type and is said to be ‘*inherited*’ by every object of that type.

A statement of the form:

**update** *OBJECT block*

where *OBJECT* evaluates to an object adds the code in the *block* temporarily to the object. The code is added and all code in the object executes until it can do nothing more. Then the temporary code is removed from the object.

When code is added to an object and executes, the statement qualifier **default** acts when execution stops to enable execution of **default** statements, as per 8.4<sup>p92</sup>. Similarly the **next** operator can be used to create iterations of the object. Each iteration replaces the previous iteration of the object wherever the object is referenced. Thus if two variables equal the object, and one is used to cause the object to iterate, both variable values will become the new iterated object.

A statement of the form

**default next**(*C*) = *C*

is implied in the permanent code for every component name  $C$  of an object, provided no other default statement in the permanent code outputs `next( $C$ )`. Thus components are normally propagated unchanged from one object iteration to the next. A statement of the form

`default next( $C$ ) = UNDEFINED`

can be used to override this behavior to make  $C$  have no value at the beginning of the next object iteration.

When an iteration of an object is created by temporary code, the sequence of actions is:

- (1) Add the temporary code to the object.
- (2) Execute all object code until it terminates.
- (3) If any `next` component values were defined by the execution in step (2), create the next iteration of the object, giving this iteration only the permanent code of the object.
- (4) In the next iteration of the object execute all code until it terminates.

Code can be temporarily added to an object by invoking methods. A ***method*** is defined by a ***method definition*** that is an expression definition which is part of the code of an object. The method is callable from code inside or outside the object code block. The syntax for a method definition is:

**on** *expression-definition*

When the method definition is called, the method code block is added temporarily to the code that contained the method definition. The object whose code this is may be referred to within its code by the name **SELF**, which may or may not also be an argument of the method definition. If it is an argument, the method definition has an implicit guard that requires the **SELF** argument value to be equal to the object whose code contains the method definition.

If there is no **SELF** argument, a new object is created when the method is called, and the permanent code plus method code is executed for that object. The permanent code must contain the method definition in this case. Such a method is called a **constructor**, and the object created is said to be constructed by the method execution. Constructor executions usually return the object they construct, but this is not required by the language.



Constructors usually have patterns that begin with ‘{ a | an } *type-name*’, where *type-name* names the type of object they construct, but this is not required by the language either.

TBD: protection; protection zones.

## 10 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 (8.3<sup>p90</sup>). 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. If 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.

## 11 Debugging

Design:

Debugging is based on the notion that almost all CASTLE 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.

## 12 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`.

## 13 To Do

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.

How does one know that an expression is a definition, in 5.4.4<sup>p53</sup>.

Imaginary Units

Visible Side Effects

List Objects

Qualifier Definition

Qualifier Shortcut Definition

Automatic Optional Marks

## Index

- ++++, 77
- , 87
- ARITY-, 67
  - of call node, 71
- MARKS-PARSER-, 41, 46
- OPERATORS-PARSER-, 41, 46
  - operator selection, 49
  - post operator selection, 53
- OPTIONAL-QUALIFIER-, 69
- REQUIRED-QUALIFIER-, 68, 69
- REST-, 70
- TEXT-PARSER-, 41, 47, 60
- //, 76
- ::>, 39, 70
- ::?, 39, 70
- ;
  - statement separator, 89
- <:>, 39, 70
- =, 88
- =>, 88
- ??, 39, 43
- #, 77, 84
- ####, 76
- ~~, 76
- ^, 61
- ~=~, 88
- a, 94
- abort side effects, 98
- all-values, 83
- always, 88, 94
- ambiguous, 56
- an, 94
- Angle Rule, 22
- angle-character, 21
- argument order assignment, 71
- arity, 67
- arity arrow, 67
  - of call node, 71
- arity target, 67
- assignment-pattern, 88
- assignment-statement, 88
- associativity
  - of operator, 45
  - Assumption A1, 55
  - Assumption A2, 55
  - Assumption A3, 55
  - Assumption A4, 55
  - Assumption A5, 55
- begin executing side effects, 97
- begin forbidding side effects, 98
- blank line, 60
- block, 8, 82, 87
  - evaluation, 90
- block-option, 82
- call check algorithm, 71
- call merge algorithm, 74
- call node, 71
- call unification, 74
- Carriage Return Rule, 23
- carriage-return, 21
- choose operation, 84
- classical parse, 54
- closing operator, 45
- closing operator name, 45
- Closing Quote Rule, 22
- closing-angle, 21
- closing-angle-character, 21

- closing-character, 21
- closing-mark, 20
- closing-quote, 20
- closing-quote-character, 20, 22
- commit side effects, 98
- compatible
  - witness, 72
- complete
  - expression, 81
- completing the expression, 81
- component, 94
- consistent-reordering, 83
- consistent-values, 83
- context, 85
- context, 86
- control flag
  - of operator, 47
- coupled, 79
- current indentation, 31
- current indentation context, 31
  
- decimal number, 25
- decimal-digits, 25
- decimal-natural, 25
- decimal-number, 25
- declaration, 87
  - evaluation, 91
- declaration-group, 48, 87
  - evaluation, 91
- default, 88, 92
- default statement, 92
- define, 47
- define operator, 48
- definition, 47
  - in parsing stack, 40
- definition
  - of operator, 47
- definitional, 48
  - control flag, 53
- definitional flag, 47
- delay mode, 97
- delay side effects, 98
- denominator, 27
- determinate, 74
- DICTIONARY-NAME, 34
- digit, 20
- document expression, 7
  
- else, 88
- else if, 88
- else-continuation-option, 88
- else-if-continuation, 88
- empty-declaration
  - evaluation, 91
- empty-statement
  - evaluation, 91
- end executing side effects, 98
- end forbidding side effects, 98
- error mode, 97
- evaluation algorithm, 82
- evaluation-mode, 41
- evaluation-mode-option, 41
- execute mode, 97
- explicit conversion
  - to expression graph, 77
- Explicit-Bracket Rule, 31
- exponent, 28
- exponent-indicator, 28
- exponent-sign-and-digits, 28
- EXPRESSION, 38
- expression definition, 82
- expression graph, 62
- expression graph pseudo-variable, 64, 77
- expression parsing, 38, 39
- expression tree, 63
- EXPRESSION-DEFINITION, 82

- expression-definition, 87
- expression-definition**, 85, 86
- expression-item, 38
- false**, 80
- first**, 88, 94
- first-value**, 83
- fixity**
  - of operator, 44
- flat list, 13
- flattened, 12
- for every**, 95
- form-feed, 21
- Format Separator Rule, 22
- format-separator, 20, 21
- format-separator-character, 21
- FUNCTION-NAME, 34
- graph concatenation operation, 77
- graph creation conversion, 76
- graph source construction operation, 77
- graph variable default operation, 77
- graph variable value operation, 77
- group, 87
  - evaluation, 90
- guard, 82, 88
- guard-list, 82
- guard-list-option, 82
- guarded-statement, 88
- guarded-subblock, 88
- head, 67
- head arrow, 67
- head index, 67
- Horizontal Tab Rule, 23
- horizontal-space-character, 21
- horizontal-tab, 21
- i**
- as unit, 29
- if**, 88
- if-statement, 88
- immutable
  - context, 86
- implicit bracket flag, 31
- implicit conversion
  - to expression graph, 77
- Implicit-Bracket Rule, 32
- incompatible
  - witness, 72
- incomplete
  - expression, 81
- indentation, 31
  - of line, 30
- indentation record, 31
- indentation stack, 31
- indentation-lexeme, 25
- indeterminate, 74
- infix**, 44, 48
- inherit, 95
- inherited
  - variable-name, 89
- Initial Capitalization Rule, 61
- input-variable, 88, 90
- is list**
  - of context
  - of **context**, 86
- is mutable**
  - of context
  - of **context**, 86
- Isolated Post Separator Rule, 30
- iterate, 93
- iteration, 93
- j**
  - as unit, 29
- k**

- as unit, 29
- left**, 48
- left** associative, 46
- letter, 20
- letter-or-digit, 26
- lexeme, 16, 24
- lexeme dictionary entry, 34
- lexeme function, 35
- Lexeme Matching Rule, 21
- lexeme replacement definition, 34
- LEXEME-DEFINITION, 34
- LEXEME-DICTIONARY-DEFINITION, 34
- LEXEME-DICTIONARY-UNDEFINITION, 35
- LEXEME-DICTIONARY-ENTRIES, 34
- LEXEME-DICTIONARY-ENTRY, 34
- lexeme-dictionary-entry-list, 34
- LEXEME-FUNCTION-UNDEFINITION, 35
- LEXEME-FUNCTION-DEFINITION, 34
- LEXEME-UNDEFINITION, 35
- lexical context, 22
- lexical parsing, 16, 33
- line beginning, 31
- Line End Spaces Rule, 23
- line separator, 60
- line-feed, 21
- list, 12
  - context, 85
- list**
  - of context, 86
- lower-case-letter, 20
- marks parser, 41, 46
- matched
  - lexemes, 21
  - pre-lexemes, 22
- Matched Lexeme Nesting Rule, 21
- matchfix**, 44, 48
- matching
  - lexemes, 21
- math expression, 7
- maximum arity, 71
- merge pair, 65
- merge set, 65
- method, 96
- method definition, 96
- method-definition, 88
- minimum arity, 71
- minimum-indentation**, 32
- Minimum-Indentation Rule, 32
- mutable
  - context, 86
- N-witness compatible, 73
- N-witness incompatible, 73
- name**
  - of operator, 45
- named associativity, 45, 46
- nested
  - lexemes, 21
- next**, 88, 93
- next-variable, 88
- node label, 72
- node merge algorithm, 65
- none**
  - associativity, 46
- null node, 63
- number, 25
- number of required argument, 70
- number unit grouping, 37
- number-unit-group, 36
- number-unit-pair, 36
- numerator, 27
- OBJECT, 94

- object, 8, 94
- of
  - the ... of, 94
- on, 88, 96
- opening operator, 45
- opening operator name, 45
- Opening Quote Rule, 22
- opening-angle, 20
- opening-angle-character, 21
- opening-character, 21
- opening-mark, 20
- opening-quote, 20
- opening-quote-character, 20, 22
- operator, 45
- operator control flag, 47
- operator definition, 44
- operator selection algorithm, 50
- OPERATOR-ASSOCIATIVITY, 48
- operator-associativity-qualifier, 48
- OPERATOR-DEFINITION, 48
- operator-definition-qualifier, 48
- operator-fixity, 48
- OPERATOR-FLAGS, 48
- operator-flags-qualifier, 48
- OPERATOR-LABEL, 48
- operator-name, 48
- operator-parser-qualifier, 48
- operator-precedence-qualifier, 48
- operator-subdefinitions-qualifier, 48
- OPERATOR-UNDEFINITION, 49
- OPERATOR-WRAPPER, 48
- operator-wrapper-qualifier, 48
- operators parser, 41, 46
  - operator selection, 49
  - post operator selection, 53
- optional argument mark, 39, 70
- optional qualifier, 69
- optional qualifier arrow, 69
- optional qualifier mark, 39
- optional-argument-mark, 43
- optional-qualifier-mark, 43, 69
- output-variable, 88
- paragraph, 61
- parser
  - standard, 41
- parser**
  - of operator, 46
- parser definition, 40
- PARSER-DEFINITION, 40
- PARSER-NAME, 40, 48
- PARSER-UNDEFINITION, 41
- parsing
  - expression, 38, 39
  - lexical, 33
- parsing stack, 40
  - kinds of definition in, 40
- path
  - empty, 72
  - non-empty, 72
- path name, 72
  - of empty path, 72
  - of non-empty path, 72
- pattern, 82
- pattern variable, 90
- pattern-assignment-statement, 88
- phrase, 61
- Post Separator Rule, 30
- post-separator, 29
- post-separator-character, 30
- post-unit-indicator, 28
- postfix**, 37, 44, 48
- postfix sequence, 50
- postfix-unit-specifier, 36
- pre-lexeme, 16, 20
- pre-lexical context, 22



- pre-unit-indicator, 28
- pre-word, 20, 30
- precedence**
  - of operator, 45
- prefix**, 37, 44, 48
- prefix sequence, 50
- prefix-unit-specifier, 36
- prompt, 7
- pseudo-variable
  - expression graph, 64, 77
- pure unification, 65
  
- qualified-statement, 88
- qualifier, 68
- qualifier arrow, 68
- qualifier mark
  - optional, 39
  - required, 39
- qualifier-mark, 43
- quote, 20
  
- radix-indicator, 26
- radix-number, 26
- radix-number-mark, 26
- ratio, 27
- raw expression, 39
- raw expression tree, 63, 67
- raw-argument, 43
- raw-argument-list, 43
- raw-argument-remainder-option, 43
- raw-base-argument-list, 43
- RAW-EXPRESSION**, 43
- raw-expression-head, 43
- raw-non-empty-base-argument-list, 43
- raw-qualifier-head, 44
- raw-qualifier-phrase, 43
- raw-subexpression, 43
- real-number, 25
  
- remainder mark, 39, 70
- reorder mark, 39, 70
- reorder-mark-option, 43
- reorderable argument, 70
- reorderable argument arrow, 70
- reorderable argument index, 70
- replaced-lexeme, 34
- replacing-lexemes, 34
- required qualifier, 68, 69
- required qualifier arrow, 68, 69
- required qualifier mark, 39
- required-qualifier-mark, 43, 68
- rest argument, 70
- rest argument arrow, 70
- rest argument index, 70
- right**, 48
- right** associative, 46
- right-side, 88
- right-side-expression, 88
- root node, 64
  
- scientific-number, 28
- section, 61
- SELF**, 96
- semi-classical parse, 55
- Semi-Colon Rule, 31
- sentence, 61
- Sentence Terminator Rule, 61
- sentence-non-terminator, 61
- sentence-terminator, 61
- separator, 25, 29
- sequence-break, 87, 90
- set
  - context, 85
- side effect, 97
- side effect mode, 97
- side effect mode stack, 97
- sign, 25

- source
  - of empty path, 72
  - of non-empty path, 72
- space, 21
- spacer, 60
- statement, 88
- statement-group, 87
  - evaluation, 90
- statement-qualifier, 88
- strong**
  - post-separator, 29
- strong-post-separator-character, 30
- subdefinition, 47
- subdefinitions-block, 48
- subexpression, 38
- Subexpression Rule, 38
- target
  - of empty path, 72
  - of non-empty path, 72
- text parser, 41, 47
- text parsing, 60
- Text Simplification Rule, 62
- the**
  - the ... of**, 94
- todo**
  - list, 97
- total number of argument, 70
- true**, 80
  - block value, 92
- type, 8
- type-name, 94
- un-reorderable argument, 67
- un-reorderable argument arrow, 67
- un-reorderable argument index, 67
- undefine**, 47
- UNDEFINED, 93
- argument of
  - an expression-definition, 86
- undo**
  - list, 97
- unguarded-subblock, 88
- unification
  - call, 74
- unit multiplier insertion, 37
- Unit Number Rule, 29
- unit-base-number, 28
- unit-indicator, 28
- unit-number, 28
- unit-specifier, 36
- UNIT-SPECIFIER-DEFINITION, 37
- UNIT-SPECIFIER-FIXITY, 37
- UNIT-SPECIFIER-NAME, 37
- UNIT-SPECIFIER-UNDEFINITION, 37
- unsigned-decimal-number, 25
- unsigned-radix-number, 26
- unsigned-ratio, 27
- unsigned-unit-base-number, 28
- update**, 95
- upper-case-letter, 20
- value
  - of block, 92
- value** variable, 92
- variable-assignment-statement, 88
- variable-name, 88
- vertical-space-character, 21
- vertical-tab, 21
- weak**
  - post-separator, 29
- weak-post-separator-character, 30
- when**, 88
- when-statement, 88
- white-space, 20, 21, 23

- white-space-character, 21
- with associativity, 48
- with contexts missing
  - of context
  - of context, 86
- with flags, 48
- with parser, 48
- with precedence, 48
- with subdefinitions, 48
- with wrapper, 48
- witness, 72
  - of node, 72
- witness compatible, 73
- witness incompatible, 73
- word-character, 20
- wrapper**
  - of operator, 47