The Layered Programming Languages (Draft 1)

Robert L. Walton[1]

December 26, 2022

[1]This document is dedicated to the memory of Professor Thomas Cheatham of Harvard University.

1 Motivation

The layered languages are ideally a set of compatible computer languages. Some guiding principals are:

> ***Syntax Hypothesis*** People reason syntactically rather than semantically, so language syntax is very important, and the ability to extend syntax to support new library functionality is very important.
>
> ***Distributed Modularity*** Enhancements made by non-communicating users should fit together without too much labor.[2]

Another idea, from which the term 'Layered' originally derives, is to create a compatible set of three programming languages, low level, middle level, and high level, based on the principals:

Type Checking Segregation Hypothesis A strongly typed-checked general-purpose computer-efficient language is impossible. What is possible is to segregate code non-type-checkable code into small inline library functions with code that uses these functions being strongly type-checked.

Interpreted Language Efficiency Hypothesis An interpreted programming language is more efficient for the programmer than a compiled language, perhaps because of quicker testing turnaround and more capable logging features. The majority of code in a typical computer system can be written in a interpretive language if it is integrated with a compiled language.

So given this the low level language would segregate code into small inline library functions that are not type-checkable, while code that uses these functions would be type-checked. The middle level language would add an extensive library to this low level language and integrate parts of this library into the syntax. The high level language would add an interpretive language integrated with the middle level language.

However there are other agendas for the layered languages. One is to build an interpretive language that integrates the tools a non-professional programmer might used in daily life. This means integrating a classic programming language with a text processing language, a spreadsheet-like language, a mathematics language, etc.

Given all this, this manual describes the lexical scanner, parser, symbol table, etc. shared by the various '***layered languages***'.

2 Lexical Scanning

The L-, M-, and H-Languages are each encoded as UTF-8 text files (UTF-8 is an encoding of UNICODE that extends ASCII). These files are read and converted to a sequence of lexemes by a process known as lexical scanning. A lexeme is a UTF-8 character string: e.g., '`hello`', '`+`', '`;`', '`9.200`' are four lexemes.

[2] Solutions like using regular expressions for lexical analysis and LL parsers do not appear to meet this criteria.

The lexical scanner is written in the M-Language and can be replaced.[3] This section describes the standard scanner which is driven by the lexical tables and is capable of scanning lexemes of common languages such as C and C++.

The standard scanner attempts to be both machine and human efficient. It attempts to be ***machine efficient*** by being deterministic[4] and attempts to be ***human efficient*** by supporting algorithms that people can readily understand and hopefully quickly learn to mimic.[5]

The standard scanner also tries to promote '***distributed modularity***', meaning that enhancements made by non-communicating users are more likely to fit together.[6]

The standard scanner operates on an input stream of 32-bit UNICODE characters. Each character is associated with a ***position*** that specifies the location of the character within its file. Specifically, a position encodes a line-within-file number and byte-within-line offset. The standard scanner identifies a lexeme as a consecutive sequence of input stream UNICODE 32-bit characters with corresponding consecutive position values.

The ***line number*** of a character is 1 plus the number of *line-feeds* preceding the character. The ***byte offset*** of a character is the number of bytes before the character but after the first *line-feed* preceding the character, or after the beginning of input if there is no such *line-feed*. The first line in a file has line number **`1`** and the first character in a line has byte offset **`0`**.

The standard scanner also computes an ***indent*** for each input character. This is the sum of the columns taken by all the characters before the character and after the first *line-feed* preceding the character, or after the beginning of input if there is no such *line-feed*. The first character in a line has indent **`0`**.

For the purpose of computing indents, characters in UNICODE subcategories **`Mn`** (combining marks) and **`Me`** (enclosing marks), and *control-characters* that are not *horizontal-space-characters*, take zero columns. All other characters take one column, except for horizontal tab, which takes from 1 to 7 columns, with tabs being set every 8 columns in the line, so the indent of the character after a horizontal tab is always divisible by 8.

However for the purposes of printing a line, the column in which a character is printed depends upon both the position of the character and the format in which the line is printed, and may not be the same as the character's indent. For example, horizontal tab may be printed as 1 to 7 blank columns or in a single column as $^{H}_{T}$.

The standard scanner also produces a translation of each lexeme. This may just be a copy of the 32-bit characters in the lexeme, or it may have some changes, or 'translations', of some of these characters. For example, the lexeme **`"a line<LF>"`** may have a translation that omits the **`"`**'s and converts the four characters **`<LF>`** into a single linefeed character.

When the standard scanner identifies a lexeme it also identifies a '***type***' for the lexeme. For ex-

[3] Currently the scanner is actually written in C++.

[4] Unlike regular expression scanners.

[5] A possibly non-verifiable goal.

[6] Regular expression grammars seem too unruly to meet this criterion.

ample, the lexeme **9.35** may have type '**numeric**' while the lexeme **"abc"** may have the type '**quoted string**'.

At its most fundamental level the standard scanner identifies lexical atoms in the input text. A ***lexical atom*** is a single character (e.g., the letters in an identifier or word), a short string of characters (e.g., the 4-character atom **<LF>** in a quoted string), or in some cases a longer string of characters containing a repetition of characters from some character set (e.g., an arbitrarily long string of hexadecimal digits).

Lexical atoms are read left to right and grouped by the scanner into ***lexemes***. Lexical atoms are the unit of backup in the scanner; recognition of a lexical atom can fail after some number of characters, but once it is recognized, the scanner cannot back up past a lexical atom. This permits limited look ahead in lexical scanning. For example, in some application the string **<LF>X** may begin with the lexical atom **<LF>**, while the string **<LFX** may begin with the lexical atom **<**. However, the syntax of lexical atom patterns is simple, so such look ahead is limited. Also, in most cases actual lexical atoms are very short, and actual backup is rare.

Each atom is translated individually to produce an ***atom translation***. Just as the sequence of atoms in a lexeme comprise the lexeme, the sequence of their atom translations comprises the translation of the lexeme. Most atoms translate to themselves. An atom translation may be the empty string, in effect omitting the atom from the lexeme translation. For example, the lexeme **"x"** might consist of three atoms, the first and last of which are **"** and translate into the empty string, whereas the middle atom **x** translates to itself. Similarly **"<LF>"** might also consist of three atoms, with the middle atom **<LF>** translating to a single line feed character.

The scanner has a state containing a current point in the input UNICODE character stream, a ***current lexical table*** identifier, a ***current lexeme type*** which may be '**NONE**', a ***return stack*** of lexical table identifiers, and a ***translation buffer*** holding translations of atoms. A ***lexical table*** is a set of atom patterns that match atoms, and for each pattern an ***instruction*** that determines what to do when the pattern is matched. There is also a ***default instruction*** associated with the lexical table that determines what to do when no table pattern matches the next characters in the input stream.

Most ***atom patterns*** are fixed length sequences of character patterns that match any sequence of input characters of the same length as the atom pattern if each input character of the sequence matches the corresponding character pattern. A ***character pattern*** is simply a specification of a set of characters, e.g., letters or digits. Thus one can construct an atom pattern that will recognize single letters, or one that will recognize the sequence of characters such as the 4-character sequence **<LF>** which matches the atom pattern **"<" "LF" ">"** (in which the quoted string has been broken into three parts to prevent **<LF>** being treated as representing a single line feed character) or the 4-character sequence **<0A>** which matches the atom pattern

"<0<hex-digit>>"

where **<hex-digit>** names a character pattern matching characters in the ranges **0-9** and **A-F**.

Atom patterns can also contain the special indicator **<repeat>** that allows an immediately preceding character pattern to be repeated zero or more times. Thus the pattern

"<hex-digit><repeat>"

recognizes any sequence of 1 or more **<hex-digit>**'s (a single **<hex-digit>** followed by zero or more repetitions).

The atom patterns of a single lexical table must not conflict: no character string can be recognized by two atom patterns of the same lexical table. However, a shorter atom pattern in a table may match the initial segment of an atom matched by a longer atom pattern in the table, in which case the longer pattern is used for the atom.

The scanner operates by identifying the longest sequence of characters beginning at the current input character stream point that matches an atom pattern of the current lexical table. This sequence of characters becomes the next atom, and an instruction associated with its matching atom pattern is then executed. If no atom pattern can be matched, the default instruction associated with the lexical table is executed instead.

A single lexical table instruction can do many things. It can optionally reduce the length of a matched atom, translated the matched atom to a different character string, replace the matched atom and its translation with a match from an atom table, and change the current lexical table. An instruction can fail if the atom translation does not match a given pattern or if an atom table fails to match any atom in the input. An ***instruction group*** is a sequence of instructions such that the first successful instruction will be the effective instruction in the group. An instruction group, instead of a single instruction, may be associated with an atom pattern or be the default instruction of a lexical table.

There are four kinds of lexical tables: ***master*** tables, ***lexeme*** tables, ***sublexeme*** tables, and ***atom*** tables. ***Master tables*** are used to dispatch to lexeme or sublexeme tables which typically recognize lexemes. Atom tables are used to recognize atoms whose patterns are logically the union of a finite set of patterns.

Thus a simple scanner might have one master table that dispatches to different lexeme tables, one to recognize identifiers, one to recognize numbers, one to recognize whitespace, and one to recognize comments.

Lexeme tables have ***types***. When the current lexical table becomes a lexeme table, the current lexeme type is set equal to the type of that table. The '**output** *type*' and '**output NONE**' instruction components can also be used to change the current lexeme type.

Sublexeme tables are just like lexeme tables except they have no associated type and do not reset the current lexeme type.

Upon finishing the execution of an instruction, if the current lexical table is a master table, the currently accumulated lexeme is output with the current lexeme type, if that is not '**NONE**', or the currently accumulated lexeme is discarded if the current lexeme type is '**NONE**'. The current lexeme type is reset to '**NONE**' whenever the current lexical table becomes a master table.

Thus if a lexeme table of type '**whitespace**' cannot recognized any atom and the default instruction associated with it changes the current lexical table to a master table, the lexeme accumulated

so far will be output as a lexeme of type '**whitespace**'.

When it is between lexemes the current lexical table in the scanner state is always a master table. Master tables correspond to lexical contexts. A well designed language has only one main master table, plus a few small master tables to handle error situations such as premature end-of-file; see Appendix A ^{p127}. Legacy languages may have more master tables: C/C++ has several to handle special lexical scanning of preprocessing lexemes; see Appendix B ^{p142}.

The standard scanner can report errors in one of two ways. It may simply output a lexeme with a type that indicates an error. For example, the character **** appearing outside a quoted string might be returned as a 1-character lexeme of type '**misplaced character**'. Alternatively a single atom may be reported as an error of a given type without interrupting the normal operation of the scanner. For example, if **<H>** appears in a quoted string, it may be reported as an erroneous atom of type '**unrecognized escape**'. Aside from error reporting, this atom will be processed normally; it might, for example, be omitted from the lexeme translation or be represented in that translation by the Unicode Replacement Character (hex code **FFFD**).

The algorithm for scanning a lexeme is given on in Figure 1 ^{p8}.

A single ***instruction*** can do the following:

- invoke an atom table to replace the matched atom and translate it
- reduce the length of the matched atom (before it is copied to the **translation buffer**) (the length can be reduced to **0**)
- translate the reduced length atom in the **translation buffer** to a different character string (otherwise the translation of an atom is a copy of the atom)
- make instruction success conditional on the translated atom matching an atom pattern
- announce the reduced length (untranslated) atom as an erroneous atom of a given type
- change the **current lexeme type** (the type can be changed to **NONE**)
- change the **current lexical table** - optionally the previous table ID can be pushed into the **return stack**, or optionally the new table ID can be popped from this stack
- force the atom table that contains the instruction to fail to match any atom

Details and examples are given in the next section.

2.1 Lexical Programs

A ***lexical program*** is a set of character pattern definitions and lexical tables. A lexical program can include another lexical program. The top level lexical program must begin with a master table that serves as the initial lexical table.

To scan a lexeme:
input: **current lexical table** which must be a **MASTER** table
input position in input UNICODE character stream
set **current lexeme type** = **NONE**
make **return stack** empty
make **translation buffer** empty
loop:
if **current lexical table** is a **MASTER** table and
current lexeme type is not **NONE**:
terminate loop
if **current lexical table** is a lexeme table:
set **current lexeme type** to the lexeme table type
find the longest atom beginning at the **input position**
that matches an atom pattern in the **current lexical** table, if any
if an atom was found:
copy the atom to the end of the **translation buffer**
and update the **input position** to point after the atom
execute the **instruction** associated with the atom pattern used
(the simplest **instruction** is '**accept**' which does nothing)
else:
execute the **default instruction** associated with
the **current lexical table**
(the simplest of these just change the **current lexical table**)
output: a lexeme consisting of the **translation buffer** contents
(which may be empty)
with the **current lexeme type** as its type
(which may not be **NONE**)
the **current lexical table** (which is a **MASTER** table)
the **input position** (which has moved to just after the lexeme)

Figure 1: Algorithm for Scanning a Lexeme

The lexemes of a lexical program are defined in terms of the following character classes:

horizontal-space-character ::= character in UNICODE category **Zs** (includes single space)
| *horizontal-tab-character*
vertical-space-character ::= *line-feed* | *carriage-return*
| *form-feed* | *vertical-tab*
space-character ::= *horizontal-space-character* | *vertical-space-character*

graphic-character ::= character in UNICODE category **L**, **M**, **N**, **P**, or **S**
control-character ::= character in UNICODE category **C** or **Z**

isolated-separating-character ::= character in UNICODE category **Ps**, **Pi**, **Pe**, or **Pf**; includes **{ ([« »]) }**
separating-character ::= **|** | *isolated-separating-character*

leading-separator-character ::= **`** | **¡** | **¿**
trailing-separator-character ::= **'** | **!** | **?** | **.** | **:** | **,** | **;**

quoting-character ::= **"**

letter ::= character in UNICODE category **L**
decimal-digit ::= **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9**
digit ::= character in UNICODE category **Nd** (includes *decimal-digits*)
lexical-item-character ::= *graphic-character* other than *separating-character* or **"**

Comments may be placed at the ends of lines:

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

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

Lexemes may be separated by *white-space*, which is a sequence of *space-characters*, but, with some exceptions mentioned just below, is not itself a lexeme:

white-space ::= *space-character*$^+$
horizontal-space ::= *horizontal-space-character*$^+$
vertical-space ::= *vertical-space-character*$^+$

The following is a special virtual lexeme:

indent ::= virtual lexeme inserted just before the first *graphic* character on a line

Indent lexemes have no characters, but do have an ***indent***, which is the indent of the graphic character after the indent lexeme. The ***indent*** of a character is the number of columns that precede the character in the character's physical line. *Control-characters* other than *horizontal-space-characters* take zero columns, as do characters of classes **Mn** (combining-marks) and **Me** (ending marks). All other characters take one column, except for tabs, that are set every 8 columns. Indent lexemes are used to form logical lines and blocks (p10 and 3.2 ^{p31}).

One kind of *vertical-space* is given special distinction:

line-break ::= *vertical-space* containing exactly one *line-feed*

This is the *line-break* lexeme.

Non-*indent*, non-*line-break white-space*, such as occurs in the middle of text or code outside comments, is discarded and not treated as a lexeme. Such *white-space* may be used to separate lexemes.

Horizontal-space-characters other than single space are illegal inside *quoted-string* lexemes (defined below). *Vertical-space* that has no *line-feeds* is illegal (see below). *Control-characters* not in *white-space* are illegal. Characters that have no UNICODE category are *unrecognized-characters* and are illegal:

misplaced-horizontal-space-character ::= *horizontal-space-character*, other than ASCII single space
misplaced-vertical-space-character ::= *vertical-space-character* other than *line-feed*
illegal-control-character ::= *control-character*, but not a *horizontal-space-character* or *vertical-space-character*
unrecognized-character ::= character with no UNICODE category or with a category other than **`L`**, **`M`**, **`N`**, **`P`**, **`S`**, **`C`**, or **`Z`**

Sequences of these characters generate warning messages, but are otherwise like *horizontal-space*:

misplaced-horizontal ::= *misplaced-horizontal-space-character*$^+$
misplaced-vertical ::= *misplaced-vertical-space-character*$^+$
illegal-control ::= *illegal-control-character*$^+$
unrecognized ::= *unrecognized-character*$^+$

Misplaced-horizontal only exists inside a *quoted-string*, but the other three sequences can appear anywhere. When they occur, these sequences generate warning messages, but otherwise they behave like *horizontal-space*. Specifically, outside *quoted-strings* and *comments* these sequences can be used to separate other lexemes, just as *horizontal-space* can be used, whereas inside *quoted-strings* and *comments* these sequences do nothing aside from generating warning messages.

Lexemes are tagged with the position (p4) and indent (p4) of their first character and of the first character after the lexeme. The second or both of these characters may be an imaginary end-of-file character after the end of the input.

The lexemes in a lexical program are specified in Figure 2 ^{p12}. This specification assumes there are no illegal characters in the input; see text above to account for such characters.

The symbol '**`:::=`**' is used in syntax equations that define lexemes or parts of lexemes whose syntactic elements are character sequences that must not be separated by *white-space*. The symbol '**`::=`**' is used in syntax equations that define sequences of lexemes that may and sometimes must be separated by *white-space*.

There is a special ***end-of-file*** lexeme that occurs only at the end of a file.

The lexemes of a file are organized into ***logical lines***. Most of these consist of a single non-indented

input line. Indented lines continue the current logical line. A '**;**' outside parentheses ends a logical line, as does an *indent* lexeme with zero indent or an an *end-of-file* lexeme. Since indented lines continue the current logical line, *indent* lexemes with non-zero indent do not end the current logical line.

After each logical line is formed, *indent*, *comment*, *line-break*, *end-of-file*, and logical line ending '**;**' lexemes are deleted from the logical line. If the result is non-empty, it becomes a '***program statement***'. In syntax equations such as those of Figure 2 ^{p12}, the end of a logical line is denoted by '**;**', but in examples it is usually denoted by line indentation or the end of file.

A '***comment line***' is a logical line that consists of a single *comment* that is followed by a *line-break* and either an *indent* or *end-of-file*, or is followed by just an *end-of-file*. It is an error for a logical line that is not a comment line to begin with a *comment*. *Comments* can otherwise be included at any non-beginning position in any logical line.

A special *character-representative* can consist of a UNICODE character name surrounded by angle brackets. Examples are **<NUL>**, **<LF>**, **<SP>**, **<NBSP>**. There are three other cases: **<Q>** represents the doublequote **"**, **<NL>** (new line) represents a line feed (same as **<LF>**), and **<UUC>** represents the '***unknown UNICODE character***' which in turn is used to represent illegal UTF-8 character encodings.

A special *character-representative* can also consist of a hexadecimal UNICODE character code, which must begin with a digit. Thus **<0FF>** represents **ÿ** whereas **<FF>** represents a form feed.

The Bracketed Subexpression Recognition Pass merges ***quoted string lexemes*** separated by one or two '**#**' marks if they are in the same logical line (see p59 for more details). Thus

```
"This is a longer sentence"#
    #" than we would like."
"And this is a second sentence."
```

is equivalent to

```
"This is a longer sentence than we would like."
"And this is a second sentence."
```

Quoted string concatenation is useful for breaking long quoted string lexemes across line continuations. But there is an important case where there is not an exact equivalence between the merged and unmerged versions. **"<" # "LF" # ">"** is not equivalent to **"<LF>"**. The former is a 4-character quoted string, the characters being **<**, **L**, **F**, and **>**. The latter is a 1-character quoted string, the character being a line feed.

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

lexeme ::= *numeric-word* | *word* | *natural* | *number* | *numeric*
| *mark* | *separator* | *quoted-string*
| *indent* |*line-break* | *comment* | *end-of-file*

strict-separator ::= *isolated-separating-character* | **|**$^{+}$

leading-separator ::= **`**$^{+}$ | **¡**$^{+}$ | **¿**$^{+}$

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

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

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

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

special-character-representative ::= **<** { *upper-case-letter* | *digit* }$^{+}$ **>**

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

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

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

numeric-word ::= *sign*$^{?}$ **nan** | *sign*$^{?}$ **inf** [where *letters* are case insensitive]

word ::= *middle-lexeme* that contains a *letter* before any *digit*
and is not a *numeric-word*

natural ::= *decimal-digit*$^{+}$ not beginning with **0** | **0**
[but lexical type may be changed; see p13]

number ::= *sign*$^{?}$ *integer-part exponent-part*$^{?}$ [that is not a *natural*]
| *sign*$^{?}$ *integer-part*$^{?}$ *fraction-part exponent-part*$^{?}$
[but lexical type may be changed; see p13]

numeric ::= *middle-lexeme* that contains a *digit* before any *letter*
and is not a *natural* or *number*

integer-part ::= *decimal-digit*$^{+}$

fraction-part ::= **.** *decimal-digit*$^{+}$

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

sign ::= **+** | **−** ***exponent-indicator*** ::= **e** | **E**

mark ::= *middle-lexeme* not containing a *letter* or a *digit*

indent ::= see p9 *line-break* ::= see p10

comment ::= see p9 *end-of-file* ::= see p10

Figure 2: Lexeme Program Lexemes

separator '**¿**', the *middle-lexeme* '**4,987**', and the four *trailing-separators* '**?**', '**,**' '**,**' and '**::**'.[7]

It is possible to break *words*, *numerics*, and *marks* across lines by adding **#** to the end of the first part and separately to the beginning of the second part. The two parts will be concatenated after deleting these **#**'s. Thus

```
This is a continued-middle-#
    #lexeme.
```

is equivalent to

```
This is a continued-middle-lexeme.
```

However, *naturals*, *numbers*, and *numeric-words* cannot be broken into parts.

A *numeric-word*, *natural*, or *number* lexeme is a C/C++ constant, and conversely a C/C++ constant representing decimal number and not ending in a *decimal-point* or representing an IEEE floating point special value (e.g., `NaN` or `Inf`) is a *numeric-word*, *natural*, or *number* lexeme. All these lexemes are given an IEEE double precision number value after the manner of C/C++, and then their lexical type is changed as follows:

- If the value is not a finite number, the new type is *numeric-word*. For example, this applies to `1e500` which converts to the same value as `+inf`.
- If the value is an integer in the range $[0, 10^{15})$ the new type is *natural*. For example, this applies to `1e3` which converts to the same value as `1000`.
- Otherwise the new type is *number*. For example, this applies to `10000000000` which converts to the same value as `1e10`.

In contrast, a *numeric* represents a character string and in this is like a *word*.

The syntax of a lexical program is specified in Figures 3 p14 and 4 p15.

Note that *atom-patterns* are written as if consecutive *quoted-strings* had **#** marks between them. As a convenience, the **#**'s can be omitted in the context of an *atom-pattern*.

An example lexical program that recognizes whitespace and quoted strings is given in Figure 5 p16. **<LF>** is the only *character-representative* recognized in quoted strings by this example.

A ***lexical-program*** is a file consisting of a sequence of *character-pattern-definitions*, *lexical-table-definitions*, and *program-inclusions*. These respectively define character pattern sets and lexical tables, and include lexical program units of previously defined lexical programs. Programs have *program-names* that are *quoted-strings* representing file names relative to one of several directories input separately to the lexical scanner.

[7] It is possible to use a strictly left to right scan with backup over lexical atoms to perform the right to left removal of *trailing-separators* from a *lexical-item*, by recognizing a sequence of *trailing-separator-characters* not followed by a *lexical-item-character* that is not a *trailing-separator-character* as an atom that is backed up over and rescanned: see Appendix A p127.

name ::= { *word* | *natural* | *quoted-string* }*
word
{ *word* | *natural* | *quoted-string* }*

lexical-program ::= **`begin`** *program-name* **`lexical program;`**
lexical-program-unit*
`end` *program-name* **`lexical program`**;

program-name ::= *quoted-string*

lexical-program-unit ::= *cpat-definition*
| *lexical-table-definition*
| *program-inclusion*

program-inclusion ::= **`include`** *program-name remove-clause** **`;`**

remove-clause ::= **`remove`** *lexical-table-name*
| **`remove "`***cpat-name***`"`**

Note: '*cpat*' abbreviates '*character-pattern*'

cpat-definition ::= **`"`***cpat-name***`" =`** *cpat-expression* **`;`**

cpat-name ::= **`<`***letter*$^+${**`-`***letter*$^+$}***`>`**
but not **`<`***upper-case-ASCII-letter*$^+$**`>`**

special-cpat-name ::: = **`<UNICODE-`***category***`>**
| **`<UNICODE-`***subcategory***`>**
| **`<others>`**

category ::: = UNICODE category, e.g. **`L`**

subcategory ::: = UNICODE subcategory, e.g. **`Lu`**

cpat-expression ::= *cpat-term* { **`|`** *cpat-term* }*
| *cpat-term* { **`&`** *cpat-term* }*

cpat-term ::= *cpat-factor* | **`~`** *cpat-factor*

cpat-factor ::= **`"`***character-representative***`"`**
| **`"`***character-representative***`-`***character-representative***`"`**
| **`"`***cpat-name***`"`**
| **`(`** *cpat-expression* **`)`**

lexical-table-definition ::= **`begin`** *lexical-table-name lexical-table-kind* **`table ;`**
lexical-table-entry*
default-instruction-group
`end` *lexical-table-name lexical-table-kind* **`table ;`**

lexical-table-kind ::= **`master`** | **`lexeme`** | **`sublexeme`** | **`atom`**

lexical-table-name ::= *name* except **`SCAN ERROR`** or **`NONE`**

Figure 3: Lexeme Program Syntax: Part I

default-instruction-group ::= *empty* | *instruction-group*
lexical-table-entry ::= *atom-pattern instruction-group* **;**
| *atom-pattern* **;**
atom-pattern ::= *atom-term*$^{+}$
atom-term ::= **"***atom-factor*$^{+}$**"**
atom-factor ::= *atom-primary repeat*$^{?}$
atom-primary ::= *character-representative* | *cpat-name* except *repeat*
repeat ::= **<repeat>** | **<repeat-***decimal-digit*$^{+}$**>**
instruction-group ::= *instruction* { **else** *instruction* }*
instruction ::= *instruction-component*$^{+}$
instruction-component ::= **accept**
| **match** *atom-table-name*
| **keep** *natural*
| **translate to** *translation-string*
| **translate hex** *natural natural*
| **translate oct** *natural natural*
| **translate name** *natural natural*
| **require** *atom-pattern*
| **error** *type-name*
| **output** *type-name*
| **output NONE**
| **goto** *master-table-name*
| **goto** *lexeme-table-name*
| **goto** *sublexeme-table-name*
| **call** *lexeme-table-name*
| **call** *sublexeme-table-name*
| **return**
| **fail**
type-name ::= *name* except **SCAN ERROR** or **NONE**
atom-table-name ::= *lexical-table-name*
lexeme-table-name ::= *lexical-table-name*
sublexeme-table-name ::= *lexical-table-name*
master-table-name ::= *lexical-table-name*
translation-string ::= *quoted-string*

Figure 4: Lexeme Program Syntax: Part II

```
begin example lexical program;

"<whitespace-char>" = " " | "<HT>"
                    | "<LF>" | "<VT>" | "<FF>";

begin main master table;
    "<whitespace-char>" goto whitespace;
    "<Q>" translate to "" goto quoted string;
    "<others>" output misplaced character;
    output end of file;
end main master table;

begin whitespace lexeme table;
    "<whitespace-char>" accept;
    goto main;
end whitespace lexeme table;

begin quoted string lexeme table;
    "<Q>" translate to "" goto main;
    "<LF>" keep 0 goto premature end of line;
    "<" "LF" ">" translate to "<LF>";
    "<others>" accept;
    goto premature end of file;
end quoted string lexeme table;

include premature tables;

end example lexical program;

begin premature tables lexical program;
    // Stuff to include.

begin premature end of line master table;
    "<LF>" output premature end of line
           goto main;
end premature end of line master table;

begin premature end of file master table;
    output premature end of file
           goto main;
end premature end of file master table;

end premature tables lexical program;
```

Figure 5: Example Lexeme Program

2.1.1 Lexical Program Inclusion

A ***program-inclusion*** names a previously defined lexical program whose units are to be included in the current lexical program at the point of the *program-inclusion*. There is a simple example of a program inclusion at the end of the **`example`** lexical program on p16. Identical definitions may be repeated without error, as can happen when one program includes two programs each of which include the same third program. A *program-inclusion* can have ***remove-clauses*** each of which removes a named definition from the set of included definitions. This is the only way to resolve conflicts between definitions.

2.1.2 Character Pattern Definitions

A ***character-pattern-definition*** (abbreviated as ***cpat-definition***) assigns a set of characters to a *character-pattern-name*. The latter is just a string of letters and hyphens (**`-`**) surrounded by angle brackets (**`< >`**) with hyphens occurring only between letters. Examples: **`<digit>`** and **`<line-break>`**. However, the *character-pattern-name* defined by a *character-pattern-definition* cannot contain only upper case letters, as this could conflict with some *character-representative* (some *special-character-pattern-names* which have builtin definitions do contain only upper case letters).

The character set is denoted by a ***character-pattern-expression***. The simplest character pattern expression is just a single *character-representative* in quotes, which denotes the character set containing only the quoted character. Examples: **`"0"`** and **`"+"`**. The next simplest is a list of *character-representatives* in **`[ ]`** brackets. This represents the character set consisting of the characters represented in the list. Examples each representing a set of 2 characters: **`"[+-]"`**, **`"[eE]"`**, **`"[<LF><CR>]"`**. Within the list the hyphen (**`-`**) is special when it is between two *character-representatives*. In this case it represents all the characters whose character codes are between those of the two characters represented on either side of the hyphen. Examples: **`"[0-9]"`** and **`"[A-Z]"`**. The characters are ordered according to their 32 bit unsigned UNICODE value, and when a hyphen is used between two characters, the first must have a smaller code than the second.

A *character-representative* is a representative of a single UNICODE character that can be used inside quotes (**`"`**). Most graphic UNICODE characters represent themselves. Other characters are represented by character sequences surrounded by the angle brackets **`<`** and **`>`** (which are also the less than and greater than signs). For example, a line feed can be represented by any of the following: **`<LF>`**, **`<0A>`**, **`<00A>`**, ..., or **`<00000000A>`**.

A quoted *character-pattern-name* denotes the character set associated with that name by a previous *character-pattern-definition*, or in the case of *special-character-pattern-names*, by one of the following rules:

`<UNICODE-`C**`>`**	all characters in UNICODE category C
`<UNICODE-`SC**`>`**	all characters in UNICODE subcategory SC
`<others>`	see page 20

The **|**, **&**, and **~** operators and can be used to take unions, intersections, and complements of character sets. The complement is relative to the universe of all 32 bit unsigned integer UNICODE character codes. Thus:

~"[b-y]" denotes the same character set as **"[<0>-a]" | "[z-<0FFFFFFFF>]"**.

Parentheses may be used in *character-pattern-expressions* as is normal in algebraic expressions. Note that the **|** and **&** operators cannot be used together in the same expression without intervening parentheses as neither takes precedence over the other. This is done to prevent ambiguity.

Some example *character-pattern-definitions* are:

```
"<hex-digit>" = "[0-9a-fA-F]";
"<letter>" = "<UNICODE-L>";
"<non-line-break-char>" = ~ "<LF>" & ~ "<VT>" & ~ "<FF>";
"<quotable-char>" = "[<020>-<07E>]" & ~ "<Q>";
"<whitespace-char>" = " " | "<LF>" | "<VT>" | "<FF>" | "<HT>";
"<source-character>" =
    ( "[<020>-<07E>]" & ~ "$" & ~ "@" & ~ "`" )
    | "<whitespace-char>";
```

2.1.3 Lexical Tables

A *lexical-table-definition* specifies a *lexical-table-name*, the *lexical-table-kind* of the table, a set of *lexical-table-entries* each containing an *atom-pattern* and optionally an *instruction-group*, and an optional *default-instruction-group* that is executed when no atom is recognized by any of the table entries.

The *lexical-table-kinds* are ***master***, ***lexeme***, ***sublexeme***, or ***atom***. In a master table *atom-patterns* are typically associated with **goto** or **call** instructions that change the current table from the master table to a lexeme or sublexeme table. In a lexeme or sublexeme table *atom-patterns* are typically associated with **accept** instructions which accept the matched atom as part of the current lexeme. In a lexeme or sublexeme table the default instruction is typically a **goto** or **return** back to a master table, and because this changes the current lexical table to a master table, it causes any accumulated lexeme to be output with its lexeme type being the current lexeme type if that is not '**NONE**', or it causes any accumulated lexeme to be discarded if the current lexeme type is '**NONE**'.

Atom tables permit sets of atom patterns to be used in several places without copying all the patterns. Details are on p21.

The lexical scanner is called by another program (e.g., a parser) to scan a single lexeme. The state of the scanner between calls is the current position in the input character sequence and the identifier of the master table to be used to scan the first atom of the next lexeme. This state can be changed between calls by the caller, and in particular, the master table can be changed. If it is not change, the master table for the first lexeme scan is the first master table in the lexical program, and for

subsequent scans it is the last master table of the previous scan. The first master table in a lexical program is typically named '**main**', as it is in the **example** lexical program on p16.

A lexical table consists of *lexical-table-entries* each containing an *atom-pattern* and an *instruction-group*. The ***atom-pattern*** is matched against the remaining input characters. The pattern consists of a sequence of *character-representatives*, each of which match only themselves, and *character-pattern-names*, each of which match any character in the named set of characters. Thus

```
"<0<hex-digit><hex-digit>>"
```

matches 5-character atoms consisting of a **<** followed by an **0** followed by two characters each in the character set named by **<hex-digit>** followed by a **>**. Note that this *atom-pattern* only matches atoms that are 5 characters long.

An *atom-pattern* can be broken up into separate parts each of which match consecutive segments of the atom. For example, the above atom pattern could be written as

```
"<" "<" "<hex-digit>" "<hex-digit>" ">"
```

This can be of more than cosmetic use, as in the case of the two atom patterns

```
"<" "LF" ">"          "<LF>"
```

The first of these patterns matches the 4 characters '**<**', '**L**', '**F**', and '**>**', while the second pattern matches a single line-feed character. Similarly **"[<digit>]"** represents a single character whereas **"[" "<digit>" "]"** represents a sequence of three characters.

The special character pattern name **<repeat>** can be used in an atom pattern to match zero or more occurrences of the previous character pattern. There must be a previous character pattern. That pattern can be **<others>**. The number of repetitions is the maximum number of characters in the input that match the previous character pattern. As an example use, the atom pattern **"\x<hex-digit><repeat>"** will recognize an atom consisting of the characters **\x** followed by one or more characters matching the character pattern **<hex-digit>**. This particular atom pattern will match a C++ hexadecimal escape sequence.

The special character pattern name **<repeat-**N**>**is similar but matches zero through N occurrences of the previous character pattern, where N is a sequence of *decimal-digits* not beginning with **0**.

Note that if C_1**<repeat>**C_2 is part of an atom pattern where C_1 and C_2 are character pattern names or character representatives, then any atom recognized by the pattern must contain some number of C_1 characters followed by a C_2 character that is not also a C_1 character. For example, the pattern **"X<repeat>X"** cannot recognize the atom **XXX** because the string **XXX** will match **X<repeat>** leaving nothing to match the final **X** in the pattern.

The order of *atom-patterns* in a *lexical-table* is significant. If the pattern "PC_1Q_1" is followed in the table by "PC_2Q_2" and C_1 and C_2 are not syntactically identical as *character-patterns*, then C_2

is effectively replaced by C_2 **& ~** C_1. That is, the characters in the set denoted by C_1 are in effect removed from the characters in the set C_2. Here P, Q_1, and Q_2 are arbitrary and may be empty.

The special character pattern name **<others>** can be used in an atom pattern to match any character, which by the previous paragraph means any character not previously matched. For example, if "P**<others>**Q" follows "PC_1Q_1", "PC_2Q_2", and "PC_3Q_3", in a *lexical-table*, **<others>** will match any character not matched by C_1, C_2, or C_3. There are several examples of this on p16.

The atom pattern "" which would recognize zero length atoms is not permitted, but the *default-instruction-group* at the end of the lexical table executes in the equivalent case where no atom pattern in the table matches the remaining input. In particular the *default-instruction-group* always executes if the input has reached its end. This fact is used in both the '**main master table**' and '**quoted string lexeme table**' on p16.

2.1.4 Lexical Instructions

A ***lexical-table-entry*** contains an *atom-pattern* and an optional ***instruction-group*** that consists of one or more *instructions* separated by **else**'s. Most *instruction-groups* contain just a single *instruction*, so we will explain this first. *Instruction-groups* with more than one instruction contain instructions that can fail, causing the failing instruction to become a no-operation and the next instruction in the instruction group to be executed instead.

An ***instruction*** in an *lexical-table-entry* specifies actions to be taken when an atom is recognized by the *atom-pattern* of the entry. The *instruction* consists of a set of *instruction-components* each of which controls part of the atom processing.

The *default-instruction-group* at the end of an lexical table behaves as if a it was associated with the *atom-pattern* "" that recognizes zero length atoms. Note that actual *atom-patterns* are not allowed to be "".

The order of processing *instruction-components* in an *instruction* is:

Components that may cause their containing instruction to fail:

match, **translate oct**, **translate hex**, or **translate name**
require

Components that only execute if their containing instruction does not fail:

keep
translate to
error
output
goto, **call**, **return**, or **fail**

An *instruction* with no *instruction-components* simply accepts the matched atom and copies it into the translation. In order to indicate this, the instruction may be written as if it had a sin-

gle '**accept**' instruction component. If this component is present in an *instruction*, no other *instruction-components* may be present in the *instruction*. Omitting the *instruction-group* of a *lexical-table-entry* is the same as writing '**accept**' for the *instruction-group*.

2.1.4.1 Match Instruction Components and Atom Tables. The '**match** *atom-table-name*' *instruction-component* invokes the named ***atom table*** to recognize an atom and provide its translation. Any previously matched atom and translation are ignored, and the atom table *atom-patterns* are used to rematch the current atom, provide a translation of the atom, and determine whether the containing *instruction* fails.

Atom tables match only a single atom and implicitly return to their invoking *instruction* after matching one atom. The **output**, **goto**, **call**, and **return** *instruction-components* cannot be used in an atom table. However, the **fail** *instruction-component* can only be used in an atom table, and when executed indicates that the atom table invoking *instruction* has failed, so that *instruction* becomes a no-operation and the next *instruction* in the invoking *instruction-group* (i.e., the *instruction* after the '**else**' following the failed *instruction*) is executed.

accept, **match**, **keep**, **require**, and any of the **translate** *instruction-components* can be used in an atom table *instruction*. An atom table instruction may contain a **match** that invokes another atom table, but recursion is prohibited, and as **match** uses the **return stack**, a very deep nesting of **match**'es may exceed the stack limit (which is at least **32**).

If an instruction invokes an atom table and the atom table does not fail, the original atom match that caused the instruction to be executed is replaced by the atom matched by the atom table, and the atom translation is that provided by the atom table.

An instruction that invokes an atom table fails if the atom table fails or if the instruction has a **require** component that fails when applied to the atom translation produced by the atom table. If the instruction fails, the original atom match is reinstated (the instruction becomes a no-operation) before the next *instruction* in the *instruction group* executes.

The **error** instruction component can also be used in an atom table, but if the atom table fails, or if a subsequent **require** causes a **match** that invoked the atom table to fail, the effects of the **error** instruction component will not be undone. Normally the only effect is to announce an error.

Atom tables may have a *default-instruction-group*. However, this only makes sense if it is just a '**fail**', or if it is contains **match** components. If an atom table is invoked and none of its *atom-patterns* match an atom, and if the table has no *default-instruction-group*, then a scan error is signaled (i.e., '**fail**' is not implicit).

2.1.4.2 Translate Hex/Oct/Name Instruction Components. The **translate hex/oct/name** components are applied to a matched atom to produce a single UNICODE character translation of the atom.

An instruction can contain at most one of the following components:

```
match
translate hex
translate oct
translate name
```

This is because all four of these components independently produce an atom translation.

Note that if any previous instruction in an instruction group contains a **match** component, that instruction must have failed, and the atom as it was when the instruction group started to execute will have been reinstated, so **translate hex/oct/name** components are always applied to the matched atom as it was when the containing instruction group started to execute, before any **match** components in the group executed.

Also note that any **keep** component executes after **translate hex/oct/name** components, and so does not affect the matched atom used by these components.

The '**translate hex** m n' *instruction-component* ignores the first m and last n characters of the matched atom and, viewing the rest as hexadecimal digits, converts these into an 32 bit unsigned integer UNICODE character code, which becomes the 1-character atom translation. An example is the lexical table entry

```
"<0<hex-digit><hex-digit>>" translate hex 2 1;
```

which might be used to accept a character representative in a quoted string and copy the appropriate hexadecimally represented character code into the lexeme translation. If characters that are supposed to be hexadecimal digits are not, the **translate-hex** component fails (but in our example this cannot happen because the definition of **<hex-digit>** restricts these characters appropriately). The '**translate hex** instruction component accepts the characters **a-f**, **A-F**, and **0-9** as hexadecimal digits (but **<hex-digit>** in the example might not accept all these).

The '**translate oct** m n' *instruction-component* is identical but views the rest of the characters as octal and not hexadecimal. An example use would be the lexical table entry

```
"\<oct-digit><oct-digit><oct-digit>" translate oct 1 0;
```

The '**translate name** m n' *instruction-component* is similar but views the rest of the characters as the UNICODE name (a.k.a., UNICODE abbreviation alias) of a character. An example use would be the lexical table entry

```
"<<upper-case-ASCII-letter><repeat>>" translate name 1 1;
```

which would translate the atom '**<LF>**' to a line feed. Note that UNICODE names consist of ASCII upper case letters and digits, with a letter being first. In addition to standard UNICODE character names, **NL** is recognized as the name of the line feed character, **Q** is recognized as the name of the **"** character, and **UUC** is recognized as the name of the '***Unknown UNICODE Character***' that replaces erroneous UTF-8 character encodings on input.

2.1.4.3 Require Instruction Components. The '**require** *atom-pattern*' *instruction-component* tests whether the atom translation matches the given *atom-pattern*, and if no, causes the *instruction* to fail. A **require** should only appear in an *instruction* which is followed by an '**else**' in an *instruction-group*. A **require** should only appear in an instruction that also contains a **match**, **translate hex**, **translate oct**, or **translate name** instruction component, and is only useful for testing the translations produced by these latter components.

An example use would be the lexical table entry

```
"\u<hex-digit><hex-digit><hex-digit><hex-digit>"
    translate hex 2 0
    require "<letter>" else
    translate to "" error misplaced character in identifier;
```

which might be used to process an atom of the form **\uXXXX** in an identifier, accepting the atom as a legitimate representation of a character if that character is a letter, and otherwise designating the atom as an error while not putting anything in the lexeme translation (see below for the definition of the '**error**' instruction component).

The last *instruction* in an *instruction-group* must never fail, and so cannot contain a **require** *instruction-component*.

A **require** component executes before any **translate to** component (p23 below), so the translation tested by **require** can be replaced by **translate to**.

2.1.4.4 Keep Instruction Components. The '**keep** n' *instruction-component* causes the atom length to be shortened to n UNICODE characters. For example, the lexical table entry

```
",<digit>" keep 1;
```

will recognize a 1-character atom consisting of just a comma as long as the comma is followed by character in the **<digit>** character pattern character set. Atoms may not be lengthened by **keep**.

A **keep** component in the same instruction as a **match** component applies to the atom matched by the atom table invoked by the **match** component.

2.1.4.5 Translate To Instruction Components. A '**translate to** *translation-string*' *instruction-component* specifies a *quoted-string* of characters, the *translation-string*, that becomes the translation of the atom. A *translation-string* may be empty, as in the lexical table entry

```
"<Q>" translate to "" goto main;
```

which can be used to recognize the last " of a quoted string, avoid copying anything into the lexeme translation, and then switch to the '**main**' lexical table. See the example on p16.

The **translate to** component is executed after any **match**, **translate hex**, **translate oct**, **translate name**, or **require** components are executed, and overrides the translations produced by these components. It is therefore possible to produce a translation, test it with **require**, and if the test is successful, override the translation with **translate to**. An example is

```
"<"  "<digit>" match escaped char
               require "<ascii-character>"
               error ascii escaped character
               translate to ""
```

which announces that any escape sequence producing an ASCII character is an erroneous atom and replaces that atom by the empty translation so it is as if the atom did not exist in the input. One could also use '**translate to "?"**' to replace the atom by a question mark.

If an instruction does not contain any of the components **match**, **translate hex**, **translate oct**, **translate name**, or **translate to**, the translation of an atom defaults to a copy of the atom itself, after any **keep** component has been used to change the length of the atom.

2.1.4.6 Error Instruction Component. An '**error** *type-name*' *instruction-component* announces the current atom as an erroneous atom of the given *type*. The atom remains part of the current lexeme, and its translation is governed independently by **match** and **translate** instruction components (or the absence of such). The following are some examples:

```
"\<others>" error bad escape sequence;
"\<others>" translate to "" error bad escape sequence;
"\<others>" translate to "<DEL>" error bad escape sequence;
```

These might be entries in an lexical table for a quoted string. All identify an atom consisting of a backslash followed by any character that does not match any character pattern C that is in a lexical table atom pattern of the form **"**$C\ldots$**"** where C is a character pattern other than **<others>** (see p20). The first entry copies the entire 2-character atom, including the backslash, into the lexeme translation. The second entry copies nothing (the empty string **""**) into the lexeme translation. The third entry copies the DEL character (hex code **7f**) into the lexeme translation. An alternative is the entry

```
"\<others>" keep 1 error bad escape;
```

that reduces the atom size to 1 character, just the backslash, which is announced as an error without its following character, and copied to the lexeme translation. The character following the backslash remains in the input and will part of the next atom recognized. Another alternative is

```
"\" match bad escape translate "" error bad escape;
. . .
<any> = <0-<0FFFFFFFF>>;       // Matches any character.
begin bad escape atom table;
```

```
"\<any><any><any>"
end bad escape atom table;
```

in which the lexical table entry will be invoked if the next character in the input is **** and no longer lexical table atom pattern of the form **"\...** " is matched, and as a result the **** and the next three characters will be identified as a '**bad escape**' erroneous atom and translated to the empty character sequence.

The routine that announces an error atom is given the position of the atom in the input stream, so characters surrounding the atom may also be included in the announcement. For example, the line containing the atom may be printed with marks under the characters of the atom.

An erroneous atom is different from an erroneous lexeme. An erroneous lexeme is simply a lexeme whose *type* indicates to the user of the lexical scanner that the lexeme is erroneous. As such an erroneous lexeme is handled by the lexical scanner just as any other lexeme would be: i.e., the lexical scanner has no special knowledge of erroneous lexemes and no special operations for detecting or announcing them.

2.1.4.7 Output Instruction Component.

An '**output** *type-name'* *instruction-component* sets the ***current lexeme type***.

The **output** *instruction-component* is used to change the type of the lexeme to be output from the type of the table containing the instruction. For example, at the top of p16 the **main** master table has '**output end of file;**' as its *default-instruction-group*, thereby outputting a zero length lexeme with '**end of file**' type whenever the input is at the end of file.

As another example, the '**main**' master table at the top of p16 contains the entry:

```
"<others>" output misplaced character;
```

which causes an unrecognized character to be treated as a 1-character lexeme of type '**misplaced character**'. The translation of this lexeme is the character itself, but if the entry:

```
"<others>" translate to "<DEL>" output misplaced character;
```

had been used instead the translation would have been the ASCII delete (DEL) character.

Upon finishing the execution of an instruction, if the current lexical table is a master table, the currently accumulated lexeme is output with the current lexeme type, if that is not '**NONE**', or the currently accumulated lexeme is discarded if the current lexeme type is '**NONE**'. The current lexeme type is reset to '**NONE**' whenever the current lexical table becomes a master table.

The '**output NONE**' instruction can be used to reset the current lexeme type to '**NONE**' in order to discard the currently accumulated lexeme.

2.1.4.8 Goto, Call, Return, and Fail Instruction Components.

A '**goto** *lexical-table-name'* *instruction-component* switches the current lexical table after the current atom is processed. If the table switches to a master table and the current lexeme type is not '**NONE**', then the current lexeme

translation is output with the current lexeme type as its type.

A '**call** *lexeme-table-name* ' *instruction-component* is just like '**goto**' except that it also pushes the identifier of the lexical table in which the '**call**' occurs (the 'caller') into the return stack. The ***return stack*** is a stack of up to 32 lexical table identifiers used by the '**call**' and '**return**' instruction components and also by the '**match** *atom-table-name*' instruction components. Recursive calls are prohibited. Whenever a master table becomes the current lexical table, the return stack is cleared.

A '**return**' *instruction-component* is just like '**goto**' except that uses the lexical table identifier at the top of the return stack to determine the next lexical table. This identifier is also popped from the stack.

The return stack is set to empty whenever the current lexical table becomes a master table. A **call** cannot be used to call a master or atom table. A **return** can only appear in a lexeme or sublexeme table. A **goto** cannot be used to go to an atom table.

The '**call**' and '**goto**' instruction components can both be used in the same *instruction*, in which case the **goto** lexical table identifier is pushed into the return stack instead of the current lexical table identifier. The '**return**' instruction component cannot be used with '**call**' or '**goto**' (if '**return**' and '**call**' could be used together, they would have the same affect as a simple '**goto**')

The '**call**' and '**return**' instruction components permit a lexeme or sublexeme table to be shared among different master, lexeme, and sublexeme tables. For example, a lexeme table to scan a comment lexeme may be shared among several master tables; see Appendix B ^{p142} for examples.

The '**call**' and '**return**' instruction components add nothing new to the lexical scanning language, since they cannot be used recursively, and they can always be eliminated by making copies of the called tables with hard coded returns. But for certain languages these instruction components substantially reduce the size of the lexical program.

A '**fail**' *instruction-component* can only be used in an atom table and cannot be used with any other instruction components. It causes the instruction invoking the atom table to fail, but the instruction containing the '**fail**' succeeds.

The '**goto**', '**call**', and '**return**' instruction components cannot be used in an atom table.

3 Parsing

The layered languages parser attempts to promote human efficiency both by standardizing many aspects of syntax and by permitting substantial additions to syntax. Humans reason syntactically, and therefore new types of data and algorithm need to be supported by new syntax. But this new syntax needs to integrate with the old syntax, so the goal of the parser is to make possible syntactic

additions that integrate well with existing syntax.[8]

The parser is a sequence of passes each of which operates on a list of tokens. A token is either a lexeme, a subexpression, or an operator.

The top level parser pass, the bracketed subexpression recognition pass, recognizes:

- Logical lines, terminated either by line breaks with or without possible indented or non-indented continuation lines, or by logical line separators (e.g. '`;`'), or by blank lines. Note that logical lines with non-indented continuation lines that are terminated by blank lines are typically headed paragraphs.
- Indented paragraphs, introduced by line-ending indentation marks such as '`:`', and terminated by the end of indentation. An indented paragraph is a part of a logical line, typically at the end of its containing logical line, and in turn contains logical lines of its own.

  The top level is considered to be its own indented paragraph whose indentation mark is the file beginning and which ends at the file end.
- Bracketed subexpressions bounded by brackets such as '`(`' and '`)`'.
- Typed bracketed subexpressions bounded by typed brackets such as '`{b|`' and '`|b}`'.
- Typed separated subexpressions that are introduced by 'typed prefix separators' such as '`{s}`'. These are converted to typed bracketed subexpressions.
- Headed lines, that are logical lines which begin with a 'line header' that is a typed prefix separator that has the '`line`' prefix group. The line header may be implied. Headed lines are converted to typed bracketed subexpressions.
- Headed paragraphs, that are groups of logical lines that begin with a 'paragraph header' that is a typed prefix separator that has the '`paragraph`' prefix group. The paragraph header may be implied. These are converted to typed bracketed subexpressions.

The non-top-level passes operate on the subexpressions recognized by previous passes; for example, the operator pass operates on bracketed subexpressions and recognizes subexpressions bounded by operators.

Prefix separators, brackets, and indentation marks can be used to change syntax, so, for example, the syntax within a `{p}` paragraph may differ from the the syntax within a `{code}` paragraph, the syntax within a `[ ]` bracketed subexpression may differ from the syntax within a `` ` ' `` bracketed subexpression, and the syntax within a '`:`' introduced indented paragraph may differ from the syntax of its containing logical line. This is an important feature of the multi-pass organization of the parser.

Parsing reduces each recognized subexpression, including each top level logical line, to a single expression token.

[8] General LL parsers are too unruly to meet his criterion.

3.1 Tokens

A token has the following components:

type One of: A lexeme type.
BRACKETED
BRACKETABLE
PURELIST
PREFIX
MAPPED_PREFIX
IMPLIED_PREFIX
IMPLIED_HEADER
DERIVED
OPERATOR

For a lexeme, a translation of the lexeme may be recorded in the token **value** as a MIN string or MIN number.

Standardly word, mark, separator, and numeric lexeme type tokens have a **value** that is the MIN string equal to the lexeme characters; numeric-word, natural, and number lexeme type tokens have a **value** that is the lexeme characters converted by `strtod` to a MIN number (which is an IEEE double precision number); quoted string lexeme type tokens have a **value** that is the MIN string equal to the lexeme translation string; and comment, indent, line break, start of file, and end of file lexeme type tokens have no **value**.

The lexeme type of the token is modified as follows if the type is natural or number and thus the value is a MIN number. If the MIN number is an infinity, the token type is set to **numeric-word**. If the MIN number is an integer in the range $[0, 10^{15})$, the token type is set to **natural**. Otherwise the token type is set to **number**.

Standardly a quoted string token value is converted from a MIN string to an object having the MIN string as its only element and **<Q>** as its **.type**. This conversion is delayed until the converted version of the token value is needed, as when it must be copied to into an object representing an expression. Usually the quoted string token itself is discarded immediately after its value has been used.

Standardly non-indent, non-line-break whitespace lexemes and erroneous lexemes (which are announced when read) are not translated into tokens.

For an **OPERATOR**, the token **value** is the operator name recorded as a MIN string or a MIN label, or in the case where a bracketed subexpression is an operator, the **value** is the MIN object as described for **BRACKETED** tokens below. **OPERATOR** tokens are produced inside the operator pass and consumed by expression reformatters invoked by that pass.

BRACKETED, **BRACKETABLE**, and **PURELIST** tokens are used to represent subexpressions, and have MIN objects as their **value**.

The **value** of a **BRACKETED** token may have any attributes, including **.type**, **.initiator**, and **.terminator**.

A **PURELIST** token is equivalent to (and an optimization of) a **BRACKETED** token whose **value** is a MIN object that does not have any attributes (other than **.position**).

A **BRACKETABLE** token is equivalent to (and an optimization of) a **BRACKETED** token whose **value** is a MIN object that does not have a **.type**, **.initiator**, or **.terminator** attribute, but may have other attributes (besides **.position**). The **BRACKETABLE** type is given to tokens primarily by the operator parser pass.

If a **BRACKETABLE** or **PURELIST** token is to become the only element of a containing subexpression that will have no attributes other than **.type**, **.initiator**, or **.terminator**, then instead of creating a new **BRACKETED** token, any **.type**, **.initiator**, or **.terminator** from the containing subexpression is added to the **BRACKETABLE** or **PURELIST** token, the token type of this token is then changed to **BRACKETED**, and its token position and **.position** attribute are updated to the position of its containing subexpression.

A **PREFIX** token is a **BRACKETED** token that has been identified as a prefix separator. It can be converted to a **BRACKETED** token when its identity as a prefix separator is no longer useful. A **PREFIX** token necessarily has a **.type**, which is recorded in the **value_type** component of the token, but the token does not have an **.initiator** or **.terminator**.

A **MAPPED_PREFIX** token is a token that was created when when a mapped lexeme (see 3.6.2 ^{p53}) was mapped to a token that is a prefix. I differs from a **PREFIX** token only in that it cannot become an isolated header (see 3.7.12 ^{p99}).

An **IMPLIED_PREFIX** token is a prefix token that was inserted after another prefix token whose *bracket-type* bracket type table entry has an *implied-prefix*: see p87. It is just like a **PREFIX** token except that it is deleted and ignored if it heads a *prefix-n-list* with zero elements, and otherwise its value is replaced by a copy of its value before being used, as its value is shared with the bracket type table entry.

An **IMPLIED_HEADER** token is a prefix token like an **IMPLIED_PREFIX** but is a header token and appears at the beginning of a logical line, possibly after other header tokens. A header is a prefix whose *bracket-type-group* (a.k.a., prefix group) is either '**paragraph**' or '**line**. See 3.7.13 ^{p99} and 3.7.9 ^{p86}.

A **DERIVED** token is a non-lexeme that which has a token **value** that is not a MIN object. For example, a bracketed subexpression reformatter may produce a **DERIVED** token with a **value** that is a MIN special value or a MIN label. In certain circumstances the type of a lexeme token may be changed to **DERIVED** (e.g., when the token is a quoted string or numeric used as a component of an attribute label or value).

BRACKETED, **BRACKETABLE**, **PURELIST**, **PREFIX**, **IMPLIED_PREFIX**, **IMPLIED_HEADER** and **DERIVED** tokens are called ***expression tokens*** and **OPERATOR** and lexeme tokens are called non-expression tokens.

value The token **value**, as described above. This is a MIN string, number, label, special value, or object. For tokens that do not have **value**'s, this is **MISSING** (but for **DERIVED** type tokens the token my have a **value** that equals **MISSING**).

For quoted strings, this is the translation string of the lexeme, i.e., the string represented by the lexeme, and not the lexeme itself.

value_type This token component can be computed from other token components, and exists purely to optimize finding the **.initiator**, **.type**, or header group of the token **value**.

For a **BRACKETED** token whose token **value** has an **.initiator**, the value of this **.initiator**. For a **BRACKETED** token whose **value** was derived from a headed line or headed paragraph (3.7.10 ^{p95} and 3.7.11 ^{p95}) and has a **.type** with either the '**paragraph**' or '**line**' prefix group, the value of this prefix group. For a **PREFIX** token, the **.type** of the token **value** if there is one. Otherwise **MISSING**.

position The positions in the input file of the first character of the token and the first character after the token in the input text. A character position records a line number and the byte offset within a UTF-8 encoding of the line. The column containing the character can then be computed from the line itself, the format in which the line is being printed, and the byte offset of the character.

indent For an indent lexeme only, the indent of the first character after the lexeme. See p4.

Tokens are organized in a doubly threaded list. A file is translated into such a list of lexeme tokens that begins with a beginning-of-file lexeme token and ends with an end-of-file lexeme token. The lexeme tokens of a file are not read in all at once. When more tokens are needed, they are read from the file until an indent or end of file token has been read.

Lexemes are read by a lexical scanner which can be controlled by resetting the master table from which the scan of the next lexeme starts (see p6). In the parser, the master can be reset at the beginning of a logical line (see 3.2 ^{p31}).

As the parser identifies subexpressions, the tokens in the subexpression are replaced by a single expression token (of type **BRACKETED**, **BRACKETABLE**, **PREFIX**, **IMPLIED_PREFIX**, or **DERIVED**) that encodes the subexpression. Although at the top level an entire input file can be considered to be a list of logical lines (see below), the parser never forms a single token that would represent this list, but instead delivers top level logical line tokens one at a time for subsequent processing. These top level tokens appear in the the token stream as they are created, and never again accessed by the parser. They are just after the file beginning lexeme token, and may be removed from the token stream at any time by post-parser processing.

3.2 Logical Lines

Input is parsed into a sequence of ***logical lines***, each of which is a sequence of tokens. Once parsed, logical lines may be grouped into paragraphs. A paragraph with its logical lines may be nested inside a larger logical line.

In the input, non-blank physical lines begin with an indent lexeme token that contains an ***indent*** parameter that is the number of columns before the first graphic character in the physical line. All physical lines end with a line-break or end-of-file token. Blank physical lines have only this ending token, and do not have an indent token.

The ***indent*** of a non-blank physical line is the indent of its indent token.

A logical line can end with an indent token (see below for specific requirements), a line separator token outside parentheses (most commonly '`;`'), or an end-of-file token. The first logical line begins with the first token of the file. For logical lines at top level, or for all the logical lines in the same indented paragraph (see 3.3 [p36]), a logical line other than the first begins with the first token after the previous logical line.

Logical lines consisting of nothing but comment lexeme tokens and not ended by a line separator are called '***comment lines***'. Comment lines are discarded, as are empty logical lines not ended by a line separator.

After a logical line has been scanned, any indent, line-break, end-of-file, terminating line separator, or comment tokens are removed. The logical line is then compacted to form a MIN object to which an **`.initiator`** attribute is added that equals the special value **`min::LOGICAL_LINE()`** and a **`.terminator`** attribute is added that is either the terminating line separator or equals **`"<LF>"`**. The vector part of the MIN object can only be empty if the **`.terminator`** is a line separator.

More specifically, a logical line ends with one of the following as controlled by parser options defined in Figures 6 [p33], 7 [p34], and 8 [p35]:

- An indent token with indent equal to or less than the current indent.
- An indent token with indent less than the current indent.
- Any indent token.
- The first indent token after any completely blank (non-comment) line. This is called a '***paragraph break***'.
- A ***line separator***, e.g. '`;`'.
- The end of file. An end of file always terminates a logical line, regardless of option settings.

The above makes use of the notion of a ***current indent***. At top level this is just `0`. An indented paragraph changes the current indent to the indent of the first indent token within the indented paragraph. See 3.3 [p36] for more detail.

A logical line that begins after an indent (i.e., does not begin after a line separator) must begin at the current indent, else a warning message is produced. Since outside indented paragraphs the current indent is `0`, logical lines outside indented paragraphs must begin at indent `0`, unless they begin after a line separator. In particular, the first logical line of a file must begin at indent `0`.

End-At Options:

`end at le indent`
This parser option causes an indent token with indent less than or equal to the current indent to end a logical line. This means that logical lines can be continued by input lines with indent greater than the current indent, assuming other options permit.

`end at lt indent`
This parser option causes an indent token with indent less than the current indent to end a logical line. This means that logical lines can be continued by input lines with indent equal to or greater than the current indent, assuming other options permit.

`end at indent`
This parser option causes any indent to end a logical line. In this case continuation lines are not possible.

`end at paragraph break`
This parser option causes a blank line to end a logical line. If this option is not present, blank lines have no affect on parsed input.

This option is most often used with an implied paragraph header (3.7.13 ^{p99}) to turn sequences of lines between blank lines into headed paragraphs (3.7.11 ^{p95}).

`end at line separator`
This parser option allows a line separator to end a logical line. The line separator controlled is that of either the top level when not inside an indented paragraph (default is '`;`'), or of the smallest containing indented paragraph when inside an indented paragraph.

If this option is off or is turned off by an opening bracket or indentation mark or prefix separator, the controlled line separator will not be recognized anywhere in the the bracketed subexpression, indented paragraph, or prefix list.

`end at outer closing`
This parser option allows a bracketed subexpression or indented paragraph to be ended by the closing bracket of a containing bracketed subexpression in the same logical line. E.g., in '`[x(y]`' the '`]`' is permitted to end the subexpression '`(y`' by automatically inserting a '`)`' before the '`]`', while also announcing an error.

If this option is off, the only closing bracket that will be recognized is that of the innermost open bracketed expression.

Figure 6: **Parsing Options: Part I**

Enable Options:

`enable indented paragraph`
This parser option allows indentation marks beginning an indented paragraph to be recognized (if they end a physical line).

This option is on by default at top level (else parser commands could not be recognized). The option is automatically turned off inside a bracketed subexpression.

`enable header`
This parser option allows *typed-prefix-separators* that have the **`paragraph`** or **`line`** group, and therefore can be line or paragraph headers (see 3.7.10 ^{p95} and 3.7.11 ^{p95}).

This option is on by default at top level, but is automatically turned off inside a bracketed subexpression.

`enable table prefix`
This parser option allows *typed-prefix-separators* whose *bracket-type* has a selected bracket table entry. This option is off by default at top level.

`enable prefix`
This parser option allows all *typed-prefix-subexpressions* with no *elements* to be *typed-prefix-separators*. This option is off by default at top level.

Note: Unless **`enable header`** or an **`enable ... prefix`** option is on, '`@... = ...`' lines cannot be recognized.

Paragraph Options:

The following options apply only to bracket type table entries with the '**`paragraph`**' group.

`sticky`
An explicit paragraph header wit the '**`sticky`**' option replaces the paragraph implied header with a copy of itself. An explicit paragraph header without the '**`sticky`**' option replaces the paragraph implied header with the missing value at top level, or with the value from the **`define indentation mark`** command within an indented paragraph. See 6, p103.

`continuing`
A blank line ends a headed paragraph unless the paragraph header has the '**`continuing`**' option and the first logical line after the blank line does not begin with an explicit paragraph header. See 2, p101.

Figure 7: **Parsing Options: Part II**

Option groups:

default options	**end at le indent** **end at line separator** **end at outer closing** **enable indented paragraph** **enable header**
other end at options	all '**end at ...**' options
default end at options	**end at le indent** **end at line separator** **end at outer closing**
non-default end at options	all '**end at**' options except those in '**default end at options**'
other enable options	all '**enable ...**' options
default enable options	**enable indented paragraph** **enable header**
non-default enable options	all '**enable**' options except those in '**default enable options**'

Default options are also the parser options set when the a parser is initialized.

Figure 8: **Parsing Options: Part III**

Note that in applying the above comment lexeme indentation is not ignored and the indent before a comment may end a logical line (or set the current indent of an indented paragraph, or end an indented paragraph: see 3.3 ^{p36}). However a warning message is issued if a comment lexeme that is not inside a comment line (and therefore is in a non-comment logical line) is not indented more than the current indent.

Prefix separators, brackets, and indentation marks, all of which can be used to change syntax, can change the options that control what terminates a logical line. Thus if at top level logical lines end with a line break followed by a non-indented line (i.e., an indent token with **0** indent), but the opening bracket **```** changes the parser options so that the current logical line is continued by any physical line regardless of indent (and the corresponding closing bracket **'''** restores the top level options), then the input:

```
This logical line contains ```
This quoted
```

```
paragraph is inside a logical
line!''',
    which is a subexpression.
```

is parsed to the logical line

```
This logical line contains @ which is a subexpression.
```

where **@** represents the subexpression

```
```This quoted paragraph is inside a logical line!'''
```

As another example, if the indentation mark '**:**' does not change the top level line ending options, but the prefix separator **{p}** changes these options so that only a blank line, the beginning of a line indented less than the current indent, or an end of file terminate a logical line, then:

```
This logical line contains:
    {p} This is a headed paragraph
    in the indented paragraph.

    {p}
    And this is a second
    headed paragraph.
This is the next logical line.
```

is parsed to two logical lines, the first containing an indented paragraph whose sole elements are:

```
{p|This is a headed paragraph in the indented paragraph "."|p}
```

and

```
{p|This is a second headed paragraph "."|p}
```

Note that a closing bracket or a prefix separator can not force the end of a logical line. Subexpressions bracketed by opening and closing brackets must be inside a logical line. Similarly subexpressions delimited by prefix separators must be inside a logical line, with the exception that paragraphs headed by certain prefix separators can consist of a sequence of logical lines.

The master table (p6) used to scan the lexemes of a logical line is called the '***lexical master***' of the line. This can be reset at the beginning of the logical line, and will be effective for all lexemes in the logical line after the indent token preceding the logical line and through and including any indent token that follows the logical line. See 3.7.13 ^{p99} for more details.

3.3 Paragraphs

There are two kinds of ***paragraphs***: indented and headed. Paragraphs consist of sequences of logical lines, or in the case of some headed paragraphs, of a single logical line.

An ***indented paragraph*** begins with an indentation mark at the end of a physical line or just before a comment that ends a physical line, and is most frequently the last thing in the logical line to
```

which this physical line belongs. The indentation mark resets the ***current indent*** for the duration of the indented paragraph to the indent of the next indent token (the indent of the next non-blank, non-comment line), if this indent is greater than the current indent at the time the indentation mark is encountered. If the indent is not greater, or if there is an end of file before any indent token, the indented paragraph is empty and contains no logical lines.

The current indent during the duration of an indented paragraph is also referred to as the ***paragraph indent*** of that indented paragraph. The paragraph ends with the first indent token that has a smaller indent (first non-blank, non-comment line with smaller indent), or with an end of file.

Top level parsing behaves as if input were in an indented paragraph with **0** paragraph indent and a fictitious indentation mark named '**TOP LEVEL**'.

A logical line that begins after an indent (i.e., does not begin after a line separator) must begin at the paragraph indent, else a warning message is produced. In particular, the first logical line of a file must begin at indent **0**.

Details on indented paragraphs are given in 3.7.6 ^{p68}.

A ***headed paragraph*** begins with explicit or implicit ***paragraph header,*** which is a prefix separator such as the '**{p}**'s in

```
{p} This is a
paragraph.

{p}
And another
paragraph.
```

The type of the paragraph header, **p** in this example, must appear in the parser's ***bracket type table*** (3.7.9 ^{p86}), and be marked there as a paragraph header (i.e., must have the '**paragraph**' group).

A paragraph header must appear at the beginning of a logical line that is in '***paragraph beginning position***', which means that the logical line is at the beginning of a file, at the beginning of an indented paragraph, or immediately after a blank line, with intervening comment lines ignored. A headed paragraph ends with the next blank line that follows a logical line (there is an exception if the paragraph bracket type table entry has the '**continuing**' option), or when the indented paragraph that contains the headed paragraph ends (this can be at an end of file).

Details on headed paragraphs are given in 3.7.11 ^{p95}.

While an indented paragraph is being scanned, there is a '***implied header***' that if not **MISSING** is inserted at the beginning of each logical line in the indented paragraph that does not have its own beginning header. This implied header may be a paragraph header (3.7.11 ^{p95}) or a line header (3.7.10 ^{p95}).

There are also two '***lexical master***' parameters, which when not missing specify the lexical master used to scan logical line lexemes (see p36). This scan can introduce ***special lexemes*** at the beginning of the logical line that are based on the first characters in the logical line, and these lexemes

can be mapped to paragraph or line headers (see 3.6.2 [p53]).

The two lexical master parameters are the '***paragraph lexical master***' which is used for logical lines in paragraph beginning position and the '***line lexical master***' which is used for other logical lines.

The default lexical master is named '**`DEFAULT`**'. It is assumed that all lexical masters check for special lexemes at the beginning of a logical line, and after reporting any, transfer to this default lexical master to scan the rest of the line and any subsequent lines for which lexical master names are not provided.

A paragraph header changes both the implied header and lexical masters temporarily while the logical lines of the headed paragraph are being scanned (3.7.13 [p99]).

3.4 Parser Symbol Tables

Sequences of lexemes encoded as tokens are looked up in ***parser symbol tables***. A parser symbol table is conceptually a stack of ***parser definitions***.

There are different symbol tables for different parser passes, as each pass looks up only pass specific definitions. Some passes have more than one symbol table.

Each parser definition has the following components:

`label` A non-empty sequence of 'symbols' that identifies the definition.

A ***symbol*** is a MIN string representing a work, mark, or separator; or the MIN number representing a natural or number. MIN values representing quoted string or numeric-word lexemes are not symbols.

Labels are symbol table '***keys***', that is, they are what is looked up in a symbol table.

If a label has just one symbol, it is represented by the symbol. Otherwise it is represented by a MIN label whose elements are the symbols.

`type` The type of the definition. Each parser pass has one (or sometimes more) symbol tables containing definitions of types particular to the pass. For example, the bracketed subexpression recognition pass has a symbol table containing symbols of type **`opening bracket`**, **`indentation mark`**, and **`line separator`**, among others.

`selectors` A set of ***parsing selectors*** that determine if the definition is active. See text.

level Block nesting level, 0, 1, 2, ..., of the parser block containing the definition that made this symbol table entry. When the parser block ends, this entry will be removed from the symbol table stack. The top level is level 0, which is outside all parser blocks.

Lookup in a parser symbol table takes as input a token sequence that is part of a line (for the bracketed subexpression recognition pass) or subexpression (for other passes), and finds an active symbol table entry with a label whose symbols match an initial segment of the token sequence. Only symbol tokens, with lexeme types word, numeric, mark, separator, natural, or number, can match label symbols. These symbols are matched for equality with the table entry label symbols. When a match is found, the initial segment of tokens used in the match is called a '***matched label***'.

At any point the longest active match is chosen. Entries may not be active because they have the wrong type or wrong selectors (see below) for the current parser pass or context. Entries may be activated only when other entries are recognized in the current context, as when afix operators become active only after an associated non-afix operator is recognized in the same subexpression (see p108).

In the bracketed subexpression recognition pass matched labels may not span physical line boundaries. In later passes, physical line boundary tokens are no longer present, and matched labels can span physical line boundaries, but cannot cross bracketed subexpression boundaries.

The context of a symbol table lookup includes a set of ***parsing selectors***. In order to be active, a symbol table entry's set of selectors must have some selector in common with the parsing selectors. A selector set is represented as a 43-bit unsigned integer value with the bit in position 1**<<**N being on if the N+1'st selector is in the set. Thus there can be at most 43 distinct selectors used by any set of parser definitions. The selectors that can be currently used are given names that are sequences of symbols, and must be referenced by these names.

The parsing selectors may change when an explicit opening bracket, indentation mark, or prefix separator is recognized, and remain changed until the associated explicit closing bracket, end of indented paragraph, or end of prefix separator list is recognized. The parser also has top level set of selectors used to begin parsing a top level logical line, and these top level selectors can be changed by parser commands.

There are no other mechanisms for setting or changing the parsing selectors.

The following parsing selectors are builtin:

TOP LEVEL
This is always on when a top level logical line is parsed, and off by default when indented paragraph logical lines or a bracketed subexpression is parsed. While the default may be overridden, great care should be used in doing so.

LINE LEVEL
This is always on when a logical line is parsed, and off by default when a bracketed subexpression is parsed. While the default may be overridden, great care should be used in doing so.

data
This selects symbol table entries that parse raw data. It is used to enable symbol table entries that parse prefix separators and the type/attribute parts of typed brackets.

atom
This selects only the symbol table entries that parse labels and special values (e.g., **[< >]** and **[$ $]**). It is used to enable labels and special values to appear in a list of lexemes.

One selector group is also defined:

other selectors all selectors

For a few parser symbol tables lookup does not use a sequence of tokens, but instead uses a prescanned sequence of symbols, such as the type of a prefix separator. An example of such a symbol table is the bracket type table (3.7.9 p86).

The parser symbol tables act like stacks; as new parsing definitions are encountered, they are are 'pushed' into the symbol tables. These stacks can be popped, removing the parser definitions in reverse order. When a definition is popped, its effects are removed from the symbol tables. Definitions are popped by the ends of parser blocks: see p41.

It is also possible to '***undefine***' parser symbol table entries by clearing some or all of their selectors. One can also think of 'undefine's as being pushed into and popped from symbol tables, where pushing corresponds to clearing selectors from some entries and popping corresponds to restoring those selectors to their previous state.

3.5 Parser Commands

Parser commands are logical lines within a *parser-command-paragraph* that change parser tables. The syntax of a parser commands in general and in particular those associated block definition, parser trace flags, and parser selectors is specified in Figures 9 p41 and 10 p42.

A *parser-command-paragraph* is an indented paragraph on a line by itself that has the indentation mark '***PARSER*:**', Thus '***PARSER*:**' must appear by itself in a top level physical line, and the indented logical lines that follow are *parser-commands*.

A *parser-command-paragraph* is first completely parsed, and then the commands it contains are executed in order. The ***PARSER*:**' indentation mark sets the parser selectors to '**[data]**', so only the '**data**' selector is present when parser commands are parsed. It is possible, but not advisable, to use parser commands to alter the way that input is parsed if only the '**data**' selector is present.

Parser commands make use of *simple-names* that are just a *word* optionally followed by a sequence of *words* and/or *naturals*. Parser commands also make use of *quoted-keys*, that are arbitrary sequences of *word*, *mark*, *separator*, and *natural* lexemes enclosed in quotation marks to form a *quoted-string* lexeme. This embedding of lexemes in a *quoted-string* lexeme serves to keep the

parser-command-paragraph ::= **`*PARSER*:`**
parser-command*

parser-test-paragraph ::= **`*PARSER* *TEST:`**
logical-line*

parser-command ::= *parser-block-command* | *parser-input-command*
| *parser-top-level-command* | *parser-trace-command*
| *parser-bracketed-command* | *parser-pass-command*
| *parser-operator-command* | *parser-standard-command*
| *parser-selector-command*

simple-name ::= *word* { *word* | *natural* }*
key ::= *symbol*$^{+}$
symbol ::= *separator* | *mark* | *word* | *natural* | *number* | *numeric*
quoted-key ::= **`"`***key***`"`**
partial-name ::= **`""`** | *quoted-key*
parser-flag ::= *simple-name*
parser-flag-list ::= **`[ ]`** | **`[`** *parser-flag* { **`,`** *parser-flag* }* **`]`**
parser-flag-modifier-list ::= **`[`** *parser-flag-op parser-flag*
{ **`,`** *parser-flag-op parser-flag* }* **`]`**
parser-flag-op ::= **`+`** | **`-`** | **`^`**
parser-flag-spec ::= *parser-flag-list* | *parser-flag-modifier-list*
selector ::= *parser-flag* as per p45
selectors ::= *parser-flag-list* with only selector names (p45)
selector-spec ::= *parser-flag-spec* with only selector names (p45)
selector-flags ::= *parser-flag-list* with only selector names (p45)
option ::= *parser-flag* as per p33
option-spec ::= *parser-flag-spec* with only option names (p33)
option-flags ::= *parser-flag-list* with only option names (p33)

Figure 9: **Parser Command Syntax: Part I**

embedded lexemes from being given a special interpretation by the parser when the parser command is parsed.

Parser commands may be organized into ***parser-blocks*** of the form:

```
*PARSER*:
    begin block block-name
    parser-command
. . . . . .
*PARSER*:
    end block block-name
```

parser-block-command ::= **`begin block`** *block-name*
| **`end block`** *block-name*
block-name ::= *simple-name*
parser-selector-command ::= **`define selector`** *selector*
| **`print selector`** *partial-name*
parser-top-level-command ::=
`define top level parsing selectors` *top-level-selectors*
| **`define top level parsing options`** *top-level-options*
| **`define top level line separator`** *top-level-line-separator*
| **`define top level`** { **`paragraph`** | **`line`** }? **`lexical master`**
lexical-master-name
| **`print top level`**
top-level-selectors ::= *selector-spec* without any '**`TOP LEVEL`**'
or '**`LINE LEVEL`**' selector [p41]
top-level-options ::= *option-spec* with only '**`end at ...`**'
and '**`enable ...`**' options [p41]
top-level-line-separator ::= *quoted-key* | **`NONE`**
lexical-master-name ::= *quoted-key* | *simple-name* not containing '**`with`**'
parser-input-command ::= **`input`** *input-flag-spec*
| **`print input`**
input-flag ::= *parser-flag* as per p49
input-flag-spec ::= *parser-flag-spec* with only *input-flag* names
parser-trace-command ::= **`trace`** *trace-flag-spec*
| **`print trace`**
trace-flag ::= *parser-flag* as per p50
trace-flag-spec ::= *parser-flag-spec* with only *trace-flag* names
parser-standard-command ::= **`define standard`** *standard-flag-spec*?
standard-flag ::= *parser-flag* as per p44
standard-flag-spec ::= *parser-flag-spec* with only *standard-flag* names
and no **`^`** *parser-flag-op*

Figure 10: **Parser Command Syntax: Part II**

When a parser block ends, all changes to the parser tables made in the block are erased, so subsequent parsing is as if the block had never existed. The *block-name* merely serves as an error check; it is not possible to have one '**`end block`**' command end more than one parser block.

The parser has several sets of ***parser-flags***:

A set of parser flags can be specified by giving a *parser-flag-list* or a *parser-flag-modifier-list*.

When a ***parser-flag-modifier-list*** is used, the flag set is made by modifying the current flag set (trace flags, selectors, or options) according to the instruction's '*parser-flag-op parser-flag*' pairs. These pairs are interpreted as follows:

+ *parser-flag*	set the named flag
- *parser-flag*	clear the named flag
^ *parser-flag*	complement (flip) the named flag

In either the case of a *parser-flag-list* or *parser-flag-modifier-list* no flag may be named more than once.

Each particular kind of flag can also have flag groups, which are names of sets of flags. If a flag group name is used, it is effectively replaced by the names of all the flags in the group, except those flags explicitly named in the *parser-flag-list* or *parser-flag-modifier-list* in which the flag group name appears. So for example, the flag modifier list

```
[+ other end at options, - end at paragraph break]
```

is equivalent to a *parser-flag-list* setting all end-at options except **`end at paragraph break`** which is cleared. This is because the group name **`other end at options`** denotes the set of all end-at options, but since **`end at paragraph break`** is explicitly mentioned in this *parser-flag-modifier-list*, it is deleted from the set in this context.

3.5.1 Parser Standard Commands

The *parser-standard-command*:

`define standard` *standard-flag-spec*?

executes builtin parser commands that together define the ***standard parser***. If the *standard-flag-spec* is omitted, all such commands are executed. Otherwise only those selected by the *standard-flag-spec* are executed.

The flags specified by the *standard-flag-spec* are called *standard-flags* and are defined in Figure 11. There are two kinds of flags: ***components*** which execute parser commands, and ***qualifiers*** which modify the components. In the Figure, the qualifiers that modify a component are listed in [] brackets after the component definition. E.g., the `math` qualifier modifies the actions of the `brackets` component which defines standard untyped brackets.

Standard Qualifiers

code	Enables definitions using the '**code**' selector. (p45).
text	Enables definitions using the '**text**' selector. (p45).
math	Enables definitions using the '**math**' selector. (p45).
id	Enables identifier definitions (e.g., '**@**').
table	Enables table definitions (e.g., '**{table}**').

Standard Components

block	Executes a '**begin block standard**' command (p44).
top level	Executes top level definitions (p48) and ID character definitions (p53). [code,text,math,id,table]
concatenator	Executes concatenator definitions (p59) and middle break definitions (p60).
lexeme map	Executes lexeme map definitions (p56). [id,table]
brackets	Executes untyped bracket definitions (p68) and typed bracket definitions (p83) [code,text,math]
indentation marks	Executes indentation mark definitions (p73).
bracket types	Executes bracket type definitions (p93).
control operators	Executes control operator definitions (p114). [code]
iteration operators	Executes iteration operator definitions (p116). [code]
assignment operators	Executes assignment operator definitions (p115). [code,math]
logical operators	Executes logical operator definitions (p117). [code,math]
comparison operators	Executes comparison operator definitions (p117). [code,math]
arithmetic operators	Executes arithmetic operator definitions (p118). [code,math]
bitwise operators	Executes bitwise operator definitions (p119). [code]

Figure 11: **Parser Standard Flags**

Standard '`block`' Component Parser Command

```
begin block standard
```

If this is executed, standard parser definitions can be canceled by an '**end block standard**' parser command.

3.5.2 Parser Selector Commands

The only builtin selector names are '**TOP LEVEL**', '**LINE LEVEL**', '**data**', and '**atom**'. Other selector names must be defined by the *parser-selector-command*:

define selector *selector*

When the parser block containing a selector name definition command ends, the selector name is erased. There may be at most 43 selectors, including the builtin selectors, defined at any one time.

The selector print command:

print selector *partial-name*

prints all selectors whose names contain the *partial-name*. In particular,

print selector ""

prints all selectors. The block in which each selector is defined is also printed. Here a *partial-name* is a sequence of lexemes that is matched to any subsequence of lexemes in a selector name, not just an initial subsequence. There is no matching of partial lexemes.

Initial Parser Selector Commands

```
define selector TOP LEVEL
define selector LINE LEVEL
define selector data
define selector atom
```

Standard Parser Selector Commands

Parser Command	Given
`define selector code`	`+ code` qualifier and '**code**' selector is used
`define selector text`	`+ text` qualifier and '**text**' selector is used
`define selector math`	`+ math` qualifier and '**math**' selector is used

If a selector qualifier is given, but the selector is not used in any executed standard parser command, the selector will not be created.

If a selector qualifier (e.g. '`+ math`') is not given, the corresponding selector (e.g. '`math`') will be automatically deleted from every standard parser command containing it, and if this makes the command meaningless, the command will be automatically ignored and not executed. E.g., the command

```
define bracket "{{" ... "}}" [code, text, math]
      with parsing selectors [ + math, - code, - text ]
```

will be ignored if the '`+ math`' standard qualifier was not given.

3.5.3 Parser Top Level Commands

Top level parsing of a file recognizes top level logical lines and headed paragraphs as if they were in an indented paragraph, called the ***top level indented paragraph***. Each logical line or headed paragraph is formed into a MIN object which is processed as it is produced, and is not combined into a bigger object by the parser.

In particular, a top level logical line that contains parser commands is parsed first, and then executed to change parser tables. These changes are then in effect for parsing subsequent logical lines or headed paragraphs. Each **`begin block`** command effectively creates a new top level indented paragraph and the corresponding **`end block`** command terminates this paragraph. After each logical line that contains parser commands, the top level indented paragraph of the innermost parsing block that has not ended is resumed.

The parameters of the current top level indented paragraph can be modified by **`define top level`** commands and printed by the **`print top level`** command.

Note that logical lines containing parser commands must be top level, and parameters of indented paragraphs that are not top level cannot be modified.

The parsing selectors and parsing options in effect at the beginning of a top level logical line not inside a headed paragraph are those of the current top level indented paragraph, and can be modified or set by:

`define top level parsing selectors` *selector-spec*
`define top level parsing options` *top-level-options*

as per Figure 10 [p42]. Note that the selectors **`TOP LEVEL`** and **`LINE LEVEL`** are automatically set at the beginning of every such top level logical line, and cannot be set or modified by **`define top level`** commands.

By default the **`TOP LEVEL`** selector is cleared when indented paragraphs or bracketed subexpressions are being parsed, and the **`LINE LEVEL`** selector is cleared by default when bracketed subexpressions are being parsed. While these defaults may be overridden, great care must be taken in doing so.

Either of the two lexical master parameters (p38) can be set by the commands:

`define top level paragraph lexical master` *lexical-master-name*
`define top level line lexical master` *lexical-master-name*
`define top level lexical master` *lexical-master-name*

The first command sets the paragraph lexical master, the second command sets the line lexical master, and and the third command sets both lexical master parameters to the same value.

Indented paragraphs have optional line separators (p32), and the line separator of the current top level indented paragraph can be set by:

define top level line separator *quoted-key*
define top level line separator NONE

The first command sets the line separator, and the second disables it.

Top level implied headers are always missing and cannot be set.

The top level print command:

print top level

prints the top level indented paragraph parameters for the the innermost current parsing block. However, each extant parsing block has its own set of top level indented paragraph parameters, and these are stored in virtual indentation mark data definitions that can be printed by:

print indentation mark "*TOP* *LEVEL*"

See 3.7.6 [p68] for more indented paragraph parameter details.

When the parser is initialized the top level indented paragraph has no parsing selectors or lexical masters and has the '**default**' parsing options (p35).

Initial Top Level Parser Commands

```
define top level parsing selectors [data]
define top level parsing options [default end at options]
define top level line separator NONE
define top level lexical master DEFAULT
```

Standard 'top level' Component Parser Commands

Parser Command	Given Qualifiers
`define top level parsing selectors [+ code]`	`+ code`
`define top level parsing selectors [+ text]`	`- code, + text`
`define top level parsing selectors [+ math]`	`- code, - text, + math`
`define top level parsing selectors [+ data]`	`- code, - text, - math`
`define top level line separator ";"`	`+ code`
`define top level paragraph lexical master PARAGRAPH-CHECK`	`+ id, + table`
`define top level paragraph lexical master TABLE-CHECK`	`- id, + table`
`define top level paragraph lexical master DATA-CHECK`	`+ id, - table`
`define top level line lexical master DATA-CHECK`	`+ id`

3.5.4 Parser Input Commands

The current set of ***parsing input flags*** may be given or modified by the *parser-input-command*:

`input` *input-flag-spec*

Input flags modify how the parser inputs lexemes. The input flag names are builtin and are listed in Figure 12 [p49].

When a parser is initialized, its **`enable numeric words`** and **`enable naturals`** flags are set, and all the other input flags are cleared.

A numeric containing commas that has the format of a C++ number not ending in '`.`' is made into a number lexeme by removing the commas provided the commas are allowed as per the **`enable integer commas`** and **`enable fraction commas`** input flags.

The print command:

`print input`

prints the state of the current parser input flags.

The end of a parser block erases the effects of any *parser-input-commands* executed during the block.

enable numeric words Any *numeric-word*, *natural*, or *number* lexeme whose value is a **NaN** or infinity is retyped as a *numeric word* if this flag is set. If cleared, the lexeme is retyped as a *number*.

enable naturals Any *numeric-word*, *natural*, or *number* lexeme whose value is an integer in the range $[0, 10^9)$ is retyped as a *natural* if this flag is set. If cleared, the lexeme is retyped as a *number*.

enable integer commas If set, commas are allowed every 3 digits in the integer part of a numeric lexeme formatted as a C++ number not ending in '.'. See p48.

enable fraction commas If set, commas are allowed every 3 digits in the fraction part of a numeric lexeme formatted as a C++ number not ending in '.'. See p48.

Figure 12: **Parsing Input Flags**

3.5.5 Parser Trace Commands

The current set of ***parsing trace flags*** may be given or modified by the *parser-trace-command*:

trace *trace-flag-spec*

Trace flags cause trace printouts to the printer the parser uses for error messages. The trace flag names are builtin and are listed in Figure 13 ^{p50}.

When a parser is initialized, its **warnings** flag is set, and all its other trace flags are cleared.

The print command:

print trace

prints the state of the current parser trace flags.

The end of a parser block erases the effects of any *parser-trace-commands* executed during the block.

3.5.6 Parser Define/Undefine Commands

In the following sections various parser symbol table '**define**' commands are described. These push entries into the parser symbol tables. The general syntax of these is

define *type name selectors ...*

and an example is

```
define bracket "(" ... ")" [operator, text] ...
```

The *type*, '**bracket**' in the example, identifies the parser pass into whose symbol tables entries are to be pushed. The *name*, '**"(" ... ")"**' in the example, gives *quoted-keys* that become the entry labels. The *selectors*, '**[operator, text]**' in the example, specify the selector set of

warnings Print warning messages when the parser detects a parse error. A warning message also indicates what action the parser took to 'fix' the error. This flag is set when the parser is initialized.

parser input When the parser reads a token from the input, a description of the token is printed.

parser output When the parser produces a output line token, the token value is printed.

parser commands Upon successfully finishing the execution of a parser command, the command is printed (if the command is in error the error printout will print the command).

bracketed subexpressions Enables tracing for the parser bracketed subexpression recognition pass.

operator subexpressions Enables tracing for the parser operator recognition pass.

subexpression elements The printing of a value by the '**parser output**' or '... **subexpressions**' trace flags, or of a logical line in a *parser-test-paragraph*, includes a direct rendition of the value with implicit brackets and graphic representations of non-graphic characters shown. This is the default.

subexpression details The printing of a value by the '**parser output**' or '... **subexpressions**' trace flags, or of a logical line in a *parser-test-paragraph*, includes a detailed rendition of the value with all attributes shown.

subexpression lines The printing of a value by the '**parser output**' or '... **subexpressions**' trace flags, or of a logical line in a *parser-test-paragraph*, includes the lines containing the value with the value producing subexpression within these lines underlined.

keys During parsing of a subexpression by a trace-enabled parser pass, the keys found in the pass symbol tables are printed. If a key is rejected after being found, a rejection message is also printed.

Figure 13: **Parsing Trace Flags**

the entries to be pushed.

The end of a parser block pops all entries pushed into parser symbol tables during the block.

For each parser symbol table **`define`** command there is a companion parser **`undefine`** command that effectively undoes the parser **`define`** command by clearing selector bits in the designated symbol table entries. Its syntax is

`undefine` *type name selectors*

and it clears the specified selector bits from all symbol table entries with the same *type* and *name*. When the block containing a parser **`undefine`** command ends, the effects of that command are undone by restoring the cleared selector bits to their previous state.

Thus the end of a parser block erases the effects of any parser **`define`** and **`undefine`** commands executed during the block.

3.5.7 Parser Test Paragraph

A *parser-test-paragraph* may be used to test the parser. It is an indented paragraph on a top level logical line by itself that has the indentation mark '**`*PARSER* *TEST:`**'. Thus '**`*PARSER* *TEST:`**' must appear by itself in a top level physical line, and the indented logical lines that follow are the lines whose parsing is to be tested.

A *parser-test-paragraph* is first parsed using the current top level selectors, and then information about the each logical line of the paragraph is printed according to the setting of the trace flags:

```
subexpression elements
subexpression details
subexpression lines
```

A *parser-test-paragraph* has no effect other than to produce the indicated printout.

3.6 Token Mapping

When a token is created, it may be mapped by the parser input routine in two ways. An *ID* token is mapped to a preallocated stub, and a mapped lexeme may be mapped to a MIN object.

3.6.1 IDs

An ***ID*** is a kind of variable with a **`min::gen`** value. The syntax of an ID is:

ID ::= *numeric-ID* | *symbolic-ID*
numeric-ID :::= *ID-character ID-number*

symbolic-ID ::= *ID-character ID-symbol*

ID-character ::= a unicode character that is a parser parameter (see below);
it must be a non-digit, non-letter that can begin a numeric or word
should be '`@`' unless the lexical analyzer is changed: see p37

ID-number ::= *non-zero-ASCII-digit ASCII-digit**

ID-symbol ::= **<** *ASCII-letter ASCII-letter-or-digit** **>**

An ID can be used before it is defined if it refers to an object or a long string. When this is done, the ID is given a value that is a '***preallocated stub***' that will be filled in later. If an ID is given a **`min::gen`** value that is not an object or long string, the ID must be defined before it is used (otherwise any use before the definition will become ***dangling***).

The **`data`** reformatter (p89) is used to define input IDs. Printer formatting is used to define output IDs. The same format is used for input and output definitions. *Numeric-IDs* can be used for either input or output, but *symbolic-IDs* can only be used for input.

When *numeric-IDs* are being used for output, their *ID-numbers* are assigned automatically by the printer being used (each parser has an associated printer). You can just output an object with a format that allows the object to be represented by a *numeric-ID*, and if the object has been previously assigned as the value of a *numeric-ID*, that will be output instead of the object, whereas if the object has never been assigned as the value of a *numeric-ID*, a new *numeric-ID* will be created to point to the object and output that instead of the object. Separately you can command the printer to output definitions of all the *numeric-IDs* which it has created in this manner.

For each printer there is a one-to-one correspondence between objects whose *numeric-IDs* have been output and the objects, so no object is ever referenced by two distinct *numeric-IDs*. Each printer has an ***ID map*** which maintains this one-to-one correspondence.

When *numeric-IDs* are being used for input, they are defined relative to the lexeme scanner being used, and their definitions must be in the input (these definitions will be processed by the **`data`** reformatter: p89). However a *numeric-ID* may be used in the input before its definition is input if the *numeric-ID* references an object or long string. An input *numeric-ID* may be defined to have the same value as another input *numeric-ID*.

Numeric-IDs once defined cannot be redefined.

Symbolic-IDs can be used for input after the same manner that *numeric-IDs* are used for input. But unlike *numeric-IDs*, the symbolic ID table for a scanner can be completely cleared, e.g., by inputting the mark '`@@@@`' at the beginning of line as per p54. This allows *symbolic-IDs* to be reused.

On input there are two separate ID maps, one for *numeric-IDs* and one for *symbolic-IDs*. Only the latter may be cleared.

An *ID* is recognized on input when its token is being constructed, and its token value, which would normally be its lexeme string, is replaced by its ID value, which is looked up in the parser's scanner's ID map. If there is no previous value, a preallocated stub (see above) is created and

assigned as the value.

The *ID-character* is a parser parameter that can be defined or printed by:

parser-ID-character-command
::= **`define ID character`** *ID-character-representative*
| **`print ID`**

ID-character-representative ::= *quoted-character* | **`disabled`**

quoted-character ::= *quoted-string* with exactly one *character-representative*.

Initial ID Character Parser Command

```
define ID character disabled
```

Standard '`top level`' Component ID Character Parser Command

Parser Command	Given Qualifiers
`define ID character "@"`	`+ id`

However, the *ID-character* '**`@`**' is also built into the part of the input lexical scanner that generates the special lexemes **`DATA`** and **`RAW-DATA`**, which in turn are mapped (3.6.2 ^{p53}) to prefixes that invoke the **`data`** reformatter (p89), so changing the *ID-character* will not work well for input unless the lexical scanner is also changed.

3.6.2 Mapped Lexemes

A '***mapped-lexeme***' is a special kind of lexeme that has its own special lexeme type which allows a token with this type to be mapped by the '***mapped lexeme symbol table***' to a token with a different type and value. For example, a token with special lexeme type **`DATA`** can be mapped to a **`PREFIX`** type token with value '**`{data}`**'.

Entries in the mapped lexeme symbol table are managed by *parser-mapped-lexeme-commands*:

parser-mapped-lexeme-command
::= **`define mapped lexeme`** *lexeme-type-name* *parsing-selectors*
[**`with token value`** *token-value*]
[**`with lexical master`** *lexical-master-name*]
| **`undefine mapped lexeme`** *lexeme-type-name* *parsing-selectors*
| **`print mapped lexeme`** *partial-lexeme-type-name*

NOTE: The **`with ...`** clauses can appear in any order.

lexeme-type-name ::= *quoted-key*

partial-lexeme-type-name ::= *quoted-key*

quoted-key ::= see p41

token-value ::= *typed-bracketed-subexpression*
| *typed-prefix-separator*
| *untyped-bracketed-subexpression*

lexical-master-name ::= *quoted-key* | *simple-name* not containing '**`with`**'

Special lexemes do not (usually) have any associated character string, but they do have their own unique lexeme type. Although special lexemes indicate the presence of particular characters in the input, scanning a special lexeme (usually) will not consume most of these characters, which will then appear in the non-special lexeme that follows the special lexeme.

The standard special lexeme types:

`DATA`

Output by the **`DATA-CHECK`** and **`PARAGRAPH-CHECK`** lexical masters when the next input characters are '**`@`*D***', '**`@<`**', or '**`@@@@`**', where *D* is a non-zero ASCII digit. The recognized input characters become the first characters of the next lexeme. Typically mapped to the '**`{data}`**' prefix separator.

`RAW-DATA`

Output by the **`DATA-CHECK`** and **`PARAGRAPH-CHECK`** lexical masters when the next input characters are '**`!@`*D***' or '**`!@<`**'. The '**`!`**' is discarded and the other recognized input characters become the first characters of the next lexeme. Typically mapped to the '**`{raw data}`**' prefix separator.

`TABLE`

Output by the **`PARAGRAPH-CHECK`** and **`TABLE-CHECK`** lexical masters when the next 5 input characters are either '**`-----`**' or '**`=====`**'. The next input character becomes the first character of the next lexeme. Typically mapped to the '**`{table}`**' prefix separator.

`ROW`

Output by the **`ROW-CHECK`** lexical master when the next input character is '**`|`**', '**`-`**', or '**`=`**'. This next input character becomes the first character of the next lexeme. Typically mapped to the '**`{row}`**' prefix separator.

The standard lexical masters that recognize special lexemes all branch to the standard **`DEFAULT`** lexical master after finding a single special lexeme, or if there is no special lexeme in the input. The standard lexical masters are as follows, where *D* is a non-zero ASCII digit:

Lexical Master Name	Special Lexeme Trigger Characters	Special Lexeme Lexical Type	Characters Discarded
PARAGRAPH-CHECK	**@*D* @< @@@**	**DATA**	none
	!@*D* !@<	**RAW-DATA**	**!**
	-----	**TABLE**	none
	=====	**TABLE**	none
DATA-CHECK	**@*D* @< @@@**	**DATA**	none
	!@*D* !@<	**RAW-DATA**	**!**
TABLE-CHECK	**-----**	**TABLE**	none
	=====	**TABLE**	none
ROW-CHECK	**|**	**ROW**	none
	-	**ROW**	none
	=	**ROW**	none
DEFAULT	does not recognize any special lexemes		

When the bracketed subexpression parser first sees a lexeme it creates a token for that lexeme, setting the token type to the lexical type of the lexeme. The parser then looks this type up in the mapped lexeme symbol table, selecting entries active according to the top level parser selectors as set by **`define top level parsing selectors`** (p42) (these are not the current parser selectors as modified by brackets and indentations). If an entry is found, it is used to replace the token's type and value, and optionally to change the lexical master that will scan the next lexeme.

If the mapped lexeme symbol table entry has no *lexical-master-name*, the lexical master is not changed, which for standard lexical masters means the next lexeme will be scanned by the default lexical master. If the entry has no *token-value* (i.e., the entry's '`define mapped lexeme ...`' command has no '`with token value ...`' modifier), the token which selected the entry is deleted, though the lexical master may still be changed.

When a token value is changed by this process, the token is given a new token type computed from the new token value. If the *token-value* is an object with a **`.type`** but no elements, no **`.initiator`**, and no **`.terminator`**, the token is given the **`PREFIX`** token type. If the *token-value* is an object with no **`.type`**, no **`.initiator`**, and no **`.terminator`**, the token is given the **`BRACKETABLE`** token type. Otherwise if the *token-value* is an object it is given the **`BRACKETED`** token type. If the *token-value* is not an object, it is given the token type **`DERIVED`**.

A mapped lexeme is called a '***header lexeme***' if the parser symbol tables are configured so that the lexeme is only produced by looking at the first non-whitespace characters of a logical line and does not actually consume any of the characters it looks at. So for example, if the line begins with '**`========`**', a **`TABLE`** header lexeme indicating a table paragraph might be produced, followed

by a **ROW** header lexeme indicating a table row line, but none of the line beginning '**=**' characters will be skipped over while producing these header lexemes. In short, the scanner will peek ahead but not read ahead when producing header lexemes, as is normal for all mapped lexemes. Header lexemes are used to introduce headed paragraphs (3.7.11 ^{p95}) and headed lines (3.7.10 ^{p95}).

When a parser is initialized, no lexemes are mapped, and the parser's paragraph and line top level lexical masters (p38) are both set to the **DEFAULT** lexical master which does not recognize special lexemes.

Standard '`lexeme map`' Component Mapped Lexeme Parser Commands

Parser Command	Given Qualifiers
`define mapped lexeme "DATA" [TOP LEVEL]` `  with token value {data}`	`+ id`
`define mapped lexeme "RAW-DATA" [TOP LEVEL]` `  with token value {raw data}`	`+ id`
`define mapped lexeme "TABLE" [TOP LEVEL]` `  with token value {table}`	`+ table`
`define mapped lexeme "ROW" [TOP LEVEL]` `  with token value {row}`	`+ table`

Standard '`top level`' Component Parser Commands (as per p48 and p53)

Parser Command	Given Qualifiers
`define top level paragraph lexical` `  master PARAGRAPH-CHECK`	`+ id, + table`
`define top level paragraph lexical` `  master TABLE-CHECK`	`- id, + table`
`define top level paragraph lexical` `  master DATA-CHECK`	`+ id, - table`
`define top level line lexical` `  master DATA-CHECK`	`+ id`
`define ID character "@"`	`+ id`

3.7 The Bracketed Subexpression Recognition Pass

The bracketed subexpression recognition pass recognizes:

- indented paragraphs, consisting of a line-ending indentation mark (e.g. '**:**') followed by a sequence of indented logical lines, ended by a non-indented line or end of file: 3.7.6 ^{p68}
- logical lines, which may end with any line break, or with a line break not followed by an indented line, or with a line break followed by a counter-indented line, or with a line separator (e.g. '**;**'), or with any blank line, or with an end of file: 3.2 ^{p31}
- brackets (e.g. '**(**' and '**)**') and bracketed subexpressions, which must be inside logical lines:

3.7.5 [p63]

typed brackets (e.g. '`{b|`' and '`|b}`') and typed bracketed subexpressions, which must be inside logical lines: 3.7.7 [p73]

typed prefix separators (e.g. '`{p}`') and prefix lists that are lists of subexpressions between typed prefix separators, which must be inside logical lines: 3.7.8 [p83]

paragraph and line headers, which are specials kinds of typed prefix separators: 3.7.11 [p95] and 3.7.10 [p95]

mapped lexemes, which are special lexemes typically produced at the beginning of a logical line that are mapped onto paragraph or line headers or other typed prefix separators: 3.6.2 [p53]

headed lines, which are logical lines beginning with a line header: 3.7.10 [p95]

headed paragraphs, which are sequences of logical lines ended by a blank line, by the next headed paragraph, or by the end of a containing indented paragraph, where the whole headed paragraph begins with a paragraph header: 3.7.11 [p95]

broken middle lexemes, whose parts need to be merged together: 3.7.1 [p58]

split quoted strings, whose parts need to be merged together: 3.7.2 [p59]

The bracketed parser pass has three symbol tables:

Bracketed Symbol Table. Contains multi-lexeme keys found in the input:
opening brackets (e.g. '`(`') and corresponding closing brackets (e.g. '`)`'): 3.7.5 [p63]
indentation marks (e.g. '`:`') and corresponding line separators (e.g. '`;`'): 3.7.6 [p68]
typed openings (e.g. '`{`') corresponding closings (e.g. '`}`')
middles (e.g. '`|`') attribute begins (e.g. '`:`')
attribute separators (e.g. '`,`') attribute equals (e.g. '`=`')
attribute negators (e.g. '`not`'): 3.7.7 [p73]

Bracket Type Symbol Table. Contains information about bracket types: 3.7.9 [p86]

Mapped Lexeme Symbol Table. Maps some lexeme types to bracketed subexpressions that replace the lexemes: 3.6.2 [p53]

In bracketed subexpression recognition, multi-lexeme keys in the bracketed symbol table are not recognized in the input if their lexemes are not all on the same physical line. This is different from other parser passes, which are executed after line breaks have been removed.

Also bracketed symbol table entries should avoid keys that contain multi-character lexemes that end with *middle-break-begin* or begin with *middle-break-end* (see 3.7.1 [p58]). If such keys are used, they may not be recognized, or broken *middle-lexeme* parts may be misrecognized as parts of keys.

Like other parser passes, the bracketed subexpression recognition pass uses parsing selectors and parsing options. However, the bracketed subexpression recognition pass is the only pass that can change these selectors and options. Details are in the following subsections.

In addition the bracketed subexpression recognition pass maintains the parser current paragraph

indent, which indicates how much lines in the current paragraph should be indented. This indent may be changed by an indentation mark (see 3.7.6 [p68]).

Syntactically a *bracketed-subexpression* may take many forms:

bracketed-subexpression	::=	*untyped-bracketed-subexpression*	see 3.7.5 [p63]
	|	*indented-paragraph-subexpression*	see 3.7.6 [p68]
	|	*typed-bracketed-subexpression*	see 3.7.7 [p73]
	|	*typed-prefix-separator*	see 3.7.8 [p83]
	|	*prefix-n-list*	see p85 in 3.7.8 [p83]
	|	implied paragraph and line headers	see 3.7.11 [p95]
	|	mapped lexemes	see 3.6.2 [p53]
	|	*headed-lines*	see 3.7.10 [p95]
	|	*headed-paragraphs*	see 3.7.11 [p95]

However, some of these forms can only be recognized in particular contexts, e.g., *indented-paragraph-subexpressions* can only be recognized if their beginning indentation mark is at the end of a physical line or just before a comment that ends a physical line. See the indicated sections for details.

The parser commands specific to the bracketed subexpression recognizer are similarly divided:

parser-bracketed-command	::=	*parser-quoted-string-concatenator-command*	see p60
	|	*parser-middle-break-command*	see p59
	|	*parser-untyped-bracket-command*	see p64
	|	*parser-indentation-mark-command*	see p70
	|	*parser-typed-bracket-command*	see p80
	|	*parser-bracket-type-command*	see p86
	|	*parser-ID-character-command*	see p53
	|	*parser-mapped-lexeme-command*	see p53

The following subsections describe middle lexeme breaks, quoted string concatenation, the output of the bracketed subexpression parser pass, and each different kind of *bracketed-subexpression* and its related *parser-bracketed-commands*.

3.7.1 Middle Lexeme Breaks

The bracketed subexpression recognition pass merges middle lexemes if they are consecutive within a logical line, the first ends with a ***middle break begin*** (e.g., '**#**'), and the second begins with a ***middle break end*** (e.g., '**#**'). The middle break begin and end are deleted from the concatenation. Thus if logical lines end with a line break but permit indented continuation lines, then

```
This is a continued-middle-#
    #lexeme.
```

is equivalent to

```
This is a continued-middle-lexeme.
```

The middle break begin and end can be changed, or middle breaks can be disabled, by the **`define middle break`** command. Middle breaks of all current parser blocks can be printed by the **`print middle break`** command. The syntax of these commands is:

parser-middle-break-command ::=
::= **`define middle break`** *middle-break-name*
| **`print middle break`**

middle-break-name ::= "*middle-break-begin*" **`...`** "*middle-break-end*" | **`disabled`**

middle-break-begin :::= *mark* with at least 1 and at most 31 UTF-8 bytes

middle-break-end :::= *mark* with at least 1 and at most 31 UTF-8 bytes

mark ::= see p12

> WARNING: *middle-break-begin* and *middle-break-end* must be chosen so bracketed symbol table entries (p57) do not end with a *middle-break-begin* or begin with a *middle-break-end*.

If *middle-break-name* is '**`disabled`**', middle lexemes are never merged.

The end of a parser block erases the effects of any *define-middle-break-commands* executed during the block.

Initial Middle Break Parser Command

```
define middle break disabled
```

Standard '`concatenator`' Component Middle Break Parser Command

Parser Command	Given Qualifiers
`define middle break "#" ... "#'`	`+ concatenator`

3.7.2 Quoted String Concatenation

The bracketed subexpression recognition pass merges quoted strings if they are in the same logical line and are separated by one or two ***quoted string concatenators*** (e.g., '**`#`**'). Thus if logical lines permit indented continuation lines, then

```
"This is a longer sentence"#
    #" than we would like."
"And this is a second sentence."
```

is equivalent to

```
"This is a longer sentence than we would like."
"And this is a second sentence."
```

This is useful for breaking long quoted string lexemes across physical line boundaries. But there is an important case where there is not an exact equivalence between the merged and unmerged

versions. **"<" # "LF" # ">"** is not equivalent to **"<LF>"**. The former is a 4-character quoted string, the characters being **<**, **L**, **F**, and **>**. The latter is a 1-character quoted string, the character being a line feed.

The quoted string concatenator can be changed or disabled by the **define quoted string concatenator** command. Quoted string concatenators of all current parser blocks can be printed by the **print quoted string concatenator** command. The syntax of these commands is:

parser-quoted-string-concatenator-command ::=
::= **define quoted string concatenator** *quoted-string-concatenator*
| **print quoted string concatenator**

quoted-string-concatenator ::= *quoted-key* with a single component | **disabled** | **enabled**

quoted-key ::= see p41

If *quoted-string-concatenator* is '**disabled**', quoted strings are never concatenated. If it is '**enabled**', consecutive quoted strings are concatenated; that is, no concatenator lexeme needs to be placed between the strings.

The end of a parser block erases the effects of any *define-quoted-string-concatenator-commands* executed during the block.

Initial Quoted String Concatenator Parser Command

```
define quoted-string-concatenator disabled
```

Standard 'concatenator' Component Quoted String Concatenator Parser Command

Parser Command	Given Qualifiers
`define quoted-string-concatenator "#"`	`+ concatenator`

3.7.3 Bracketed Subexpression Parser Pass Input

The bracketed subexpression recognizer takes as input a stream of tokens most of which are lexemes. The exception are the tokens generated by mapping lexemes, as per p55 in 3.6.2 [p53].

When the bracketed subexpression recognizer calls another pass, e.g., the operator pass, to parse a sequence of tokens, that other pass may return one or more **BRACKETED**, **BRACKETABLE**, **PURELIST**, or **DERIVED** tokens. Similarly bracketed pass reformatters may return such tokens.

3.7.4 Bracketed Subexpression Parser Pass Output

The bracketed subexpression recognizer converts subexpressions to single **BRACKETED** or **PURELIST** tokens whose **value**'s are MIN objects, or into **DERIVED** tokens whose values are not MIN objects (e.g., they are labels or special values).

The final output of the parser is a sequence of **BRACKETED** tokens, one per top level logical line or headed paragraph. For each of these tokens, its '**value**' is a MIN object that encodes its logical line or headed paragraph.

For *untyped-bracketed-subexpressions* that are not reformatted, the brackets become the **.initiator** (e.g. '**(**') and **.terminator** (e.g., '**)**'). *Indented-paragraph-subexpressions* are similar, with the *indentation-mark* (e.g., '**:**') being the **.initiator** and the special value

[$ INDENTED_PARAGRAPH $]

being the **.terminator**. The elements of an indented paragraph are logical lines, whose **.initiator** is the special value

[$ LOGICAL_LINE $]

and whose **.terminator** is the line separator (e.g., '**;**') if there is one, or **"<LF>"** otherwise.

For *typed-bracketed-subexpressions* there is no implied **.initiator** or **.terminator**, though these may be given as explicit attributes. There is a **.type** attribute unless the type is represented by **""**. *Prefix-n-lists* are just an alternate form of *typed-bracketed-subexpressions*, and headed paragraphs are just a particular kind of *prefix-n-lists*.

The elements of a MIN object are the values of sub-subexpressions and may themselves the MIN objects, as when brackets are nested. The elements of indented paragraphs are always logical lines or headed subparagraphs.

If MIN object X has just one element that is a MIN object Y, X has no attributes other than **.type**, **.initiator**, and **.terminator**, and Y does not have any of these three attributes, then X and Y are merged to get a new object with the attributes of both X and Y and the elements of Y. This is implemented by giving the token type **BRACKETABLE** or **PURELIST** to Y as a temporary token type indicating that Y does not have any **.type**, **.initiator**, or **.terminator** attribute. Then when X is formed, the knowledge of the structure of X and the **BRACKETABLE** or **PURELIST** type of Y permit the two objects to be merged.

Consider the example:

```
( x, y 5, 8, w, p )
```

The token sequence

```
x "," y 5 "," 8 "," w "," p
```

is converted by the standard operator pass (see 3.9 p105) to a **BRACKETABLE** token with value

```
{"":  .separator = "," | x {| y 5 |} 8 w p |}
```

where `""` denotes a missing **`.type`**. If we denote this token value by Y, then the full expression is

```
{"":  .initiator = "(", .terminator = ")" | Y |}
```

and the above rule merges the attributes into Y to get

```
{"":   .initiator = "(", .terminator = ")", .separator = ","
      | x {| y 5 |} 8 w p |}
```

as the **`value`** of a **`BRACKETED`** token.

In the process of forming a MIN object, some tokens must be converted to object elements. The MIN values of these tokens become the object elements, except for quoted string and numeric lexemes. Quoted string and numeric lexemes are first converted to a MIN object whose **`.type`** is **`"<Q>"`** for a quoted string or **`#`** for a numeric, and whose single element is a MIN string equal to the the token **`value`**.

Note that natural token values are MIN numbers that are conversions of their lexeme string and not identical to this string.

Thus for example, the parser input

```
{| 005 "HELLO" 3.4 |}
```

produces a **`BRACKETED`** token whose **`value`** is a MIN object with no attributes and with the three elements that can be represented as:

`5`	MIN number	from natural token
`{"<Q>"| HELLO |}`	MIN object	from quoted string token
`{# 3.4 #}` or **`{"#"| 3.4 |}`**	MIN object	from numeric token

Untyped-bracketed-subexpressions may be reformatted to produce tokens of **`DERIVED`** type with **`value`**'s that may be MIN labels or special values. Examples are:

`[$ LOGICAL_LINE $]`	MIN special **`LOGICAL_LINE`** value
`[< A B 5 >]`	MIN label with three components, **`A`**, **`B`**, and **`"5"`** (this last a MIN string and not a MIN number)

The following is a review of the standard MIN object attributes mentioned above:

`.type`
A MIN string, or a MIN label whose elements are MIN strings, that equals

the *bracket-type* of a typed bracketed subexpression (3.7.7 [p73]),
or **`"<Q>"`** if the MIN object represents a quoted string lexeme.
or **`#`** if the MIN object represents a numeric lexeme.

The *bracket-type* `""` represents the absence of a **`.type`** attribute.

.initiator
A MIN string, or a MIN label whose elements are MIN strings, or a MIN special value, that equals

the *opening-bracket-name* of an untyped bracketed subexpression,
or the *indentation-mark-name* of an indented paragraph,
or **min::LOGICAL_LINE()** for a logical line

.terminator
A MIN string, or a MIN label whose elements are MIN strings, or a MIN special value, that equals

the *closing-bracket-name* of an untyped bracketed subexpression,
or **min::INDENTED_PARAGRAPH()** for an indented paragraph
or the terminator of a logical line:
either **"<LF>"**,
or a *line-separator-name* (e.g. **";"**)
or the terminator of sentence (e.g. **"."**)

.separator
A MIN string, or a MIN label whose elements are MIN strings, that equals the operator (e.g. **","**) that acts as a separator in a subexpression. Separators are not recognized by the bracketed subexpression parser pass, but by other passes (.e.g., the standard operator pass).

In addition to the attributes just given, a MIN object that is the value of a **BRACKETED** or **PURE-LIST** token will have a **.position** attribute that is a **min::phrase_position_vec** giving the within file positions of the lexemes which the MIN object and its elements represent. This is a hidden attribute that is not (normally) printed.

3.7.5 Untyped Bracketed Subexpressions

The following are examples of ***untyped bracketed expressions***, where ... denotes a list of elements:

```
 (  ...  )
(|  ...  |)
 [  ...  ]
[<  ...  >]
```

The general syntax is:

untyped-bracketed-subexpression ::= *opening-bracket* *element-list* *closing-bracket*
opening-bracket ::= *key* [e.g., '**(**' or '**[<**', as per *parser-untyped-bracket-command* p64]
closing-bracket ::= *key* [e.g., '**)**' or '**>]**', as per *parser-untyped-bracket-command* p64]
key ::= see p41
element-list ::= *prefix-0-list*
prefix-0-list ::= *simple-element-list*
| see p85
simple-element-list ::= *element**
element ::= *word* | *number* | *mark* | *separator* | *quoted-string*
| *untyped-bracketed-subexpression*
| *typed-bracketed-subexpression*
| *typed-prefix-separator* that is not a prefix: see p85
typed-bracketed-subexpression ::= see p76
typed-prefix-separator ::= see p84

Note: An *untyped-bracketed-subexpression element-list* is normally terminated by the expression's *closing-bracket*. It may be abnormally terminated by an outer *bracketed-subexpression's closing-bracket* (e.g. '**)**'), *typed-closing* (e.g. '**}**'), or *non-continuation-line-break*, but not by a *line-separator* (e.g. '**;**') or *typed-middle* (e.g. '**|**').

An ***untyped bracketed subexpression*** begins with an *opening-bracket* and ends with a matched *closing-bracket* as defined by a **parser define bracket** command, an example of which is

```
parser define bracket "(" ... ")" [math, text] ...
```

which specifies '**(**' to be an *opening-bracket* with '**)**' as its corresponding *closing-bracket* and '**math**' and '**text**' as the definition selectors.

The general syntax of *parser-untyped-bracket-commands* is:

parser-untyped-bracket-command
::= **define bracket** *bracket-name* *selectors*
[**with parsing selectors** *bracket-selectors*]
[**with parsing options** *bracket-options*]
[**with** *reformatter-name* **reformatter** *reformatter-arguments*$^{?}$]
| **undefine bracket** *bracket-name* *selectors*
| **print bracket** *partial-opening-bracket*

NOTE: The **with ...** clauses can appear in any order.

bracket-name ::= *opening-bracket* **...** *closing-bracket*
opening-bracket ::= *quoted-key*
closing-bracket ::= *quoted-key*

partial-opening-bracket ::= *quoted-key*
quoted-key ::= see p41
bracket-selectors ::= *selector-spec* [p41]
[Unless explicitly mentioned in *selector-spec*, '**`TOP LEVEL`**' and '**`LINE LEVEL`**' are automatically added as '**`-`**' selectors]
bracket-options ::= *option-spec* [p41]
with only '**`end at ...`**' options and '**`enable ...`**' options
[Unless explicitly mentioned in *option-spec*, '**`end at line separator`**', '**`enable indented paragraph`**, and '**`enable header`**' are automatically added as '**`-`**' options]
reformatter-name ::= *simple-name* not containing '**`reformatter`**'
reformatter-arguments ::= **`()`**
| **`(`** *reformatter-argument* { **`,`** *reformatter-argument* }* **`)`**
reformatter-argument ::= *quoted-key* | *simple-name* | *argument-sublist*
argument-sublist ::= *reformatter-arguments*

Brackets such as '**`(`**' and '**`)`**' are called '***untyped brackets***' to distinguish them from typed brackets such as '**`{b|`**' and '**`|b}`**' in which the type, in this case '**`b`**', is fairly arbitrary. See 3.7.7 ^{p73} for a description of typed brackets.

The '**`print bracket`** *partial-opening-bracket*' command prints all untyped bracket symbol table entries whose *opening-bracket* contains the *partial-opening-bracket* as a (not necessarily initial) subsequence of lexemes. Using **`""`** as a *partial-opening-bracket* will print all untyped bracket symbol table entries.

In the absence of a reformatter, an untyped bracketed subexpression is converted to a **`BRACKETED`** token whose **`value`** is a MIN object whose list elements are the components of the subexpression except for the brackets. The *opening-bracket* becomes the **`.initiator`** attribute of this object, and the *closing-bracket* becomes the **`.terminator`** attribute of the object. See below for *untyped-bracketed-subexpression* reformatters.

If a bracketed subexpression ends pre-maturely, before its *closing-bracket* is discovered, a parsing error is announced and the *closing-bracket* is inserted in the input just after the pre-maturely ended subexpression. Such a pre-mature end can be discovered by finding the *closing-bracket* of a containing *untyped-bracketed-subexpression* or *typed-bracketed-subexpression*, or the end of the logical line containing the *untyped-bracketed-subexpression*.

When an *opening-bracket* is recognized, the current parser selector set and options are saved in a stack. Then if the bracket definition contains *bracket-selectors*, a new selector set is computed by modifying the saved set according to the *bracket-selectors*. The '**`TOP LEVEL`**' and '**`LINE LEVEL`**' selectors are turned off by default. Similarly if the bracket definition contains *bracket-options*, these are used to compute a new set of end-at options. By default the '**`end at line separator`**', '**`enable indented paragraph`**', and '**`enable header`**' options are turned off. The saved selector set and options are restored after the *closing-bracket* is recog-

nized. However *closing-bracket* recognition does not depend upon the current parser selectors, but only on previous recognition of an associated *opening-bracket*.

An example using *bracket-options* to permit ```` ```...''' ```` to bracket arbitrary text with arbitrary indentation is:

```
parser define bracket "```" ... "'''" [code]
       with parsing options []
( An indented paragraph:
    ```This logical line continues on no
matter what, even with a spurious ), until
  the following is encountered- '''
    Second logical line of indented paragraph.
    ) // End of () bracketed subexpression
```

Several reformatters are provided that change the MIN value represented by an *untyped-bracketed-subexpression*:

`label` e.g.: **`[< >]`**

Requires that the subexpression elements be *words*, *naturals*, *numbers*, *numerics*, *quoted-strings*, or MIN label values. The value of an element is taken to be the MIN string equal to the lexeme string, the MIN number converted from the lexeme's string (for *naturals* and *numbers*), or the MIN label value of a non-lexeme element.

The subexpression is converted to a **`DERIVED`** token whose **`value`** is the value of the single element of the subexpression if the subexpression has exactly one element, and whose value is otherwise the MIN label made from the subexpression element values (an empty label is possible).

`special` e.g.: **`[$ $]`**

Requires that the expression consist of a single element with a string value that names a MIN special value (e.g., '**`MISSING`**' or '**`INDENTED_PARAGRAPH`**').

The subexpression is converted to a **`DERIVED`** token whose **`value`** is the special value named.

`multivalue (` *separator* **`)`** e.g.: **`{* *}`**

Requires that the subexpression elements be *words*, *naturals*, *numbers*, *numerics*, *quoted-strings*, or equal to the *separator* argument (e.g. **`","`**).

The subexpression is divided into components by *separator* tokens and is converted to a **`BRACKETED`** token whose **`value`** is a list with one element for each non-missing component. A component that is a single element becomes the MIN string of the lexeme of that element, and a component with more than one element becomes a MIN label whose elements are the MIN strings of the component elements. The **`BRACKETED`** result token
```

gets the *separator* as its **.separator** attribute, and the opening and closing brackets (e.g. **{* *}**) as its **.initiator** and **.terminator** attributes.

The *separator* argument (e.g. **","**) must denote a single MIN string, and cannot denote a multi-component MIN label (this is purely a current implementation limitation).

text (*prefix* { , *quoted-terminator* }$^{+}$) e.g.: **`` ''**

where

prefix ::= *simple-name* | *quoted-key* e.g.: **s**

quoted-terminator :::= **"***terminator***"**

terminator :::= *mark* | *separator*

If the *prefix-n-list* does not contain a *terminator* lexeme, nothing is done. Note that if the *prefix-n-list* previously began with an explicit **{***prefix***}**, any *terminators* in that previous *prefix-n-list* would now be inside subexpressions, so this rule would apply and nothing would be done.

{*prefix***}** is pre-pended to the *prefix-n-list* and the new *prefix-n-list* is processed as if it had just been read. In particular, any reformatter for *prefix* is called and if there is none the elements following the **{***prefix***}** are collected into a typed bracketted subexpression with **.type** equal to *prefix*.

Initial Bracket Parser Commands

```
define bracket "(" ...  ")" [data]
define bracket "[" ...  "]" [data]
define bracket "" ...  "" [data]
define bracket "`" ...  "'" [data]
       with parsing selectors [atom]
define bracket "``" ...  "''" [data]
define bracket "[<" ...  ">]" [data, atom]
       with parsing selectors [atom]
       with label reformatter
define bracket "["...."]" [data, atom]
       with parsing selectors [atom]
       with special reformatter
define bracket "*" ...  "*" [data]
       with multivalue reformatter ( "," )
```

Standard '`brackets`' Component Bracket Parser Commands

All Commands Require at Least One of `+ code, + text, + math`
Selectors with Missing Standard Qualifiers are Deleted from Commands

Parser Command	Given Qualifiers
`define bracket "(" ... ")" [code, text, math]`	
`define bracket "[" ... "]" [code, text, math]`	
`define bracket "`" ... "'" [code, text, math]`	

`define bracket "[<" ... ">]" [code, text, math]`
`with parsing selectors [atom]`
`with label reformatter`	

`define bracket "[$" ... "$]" [code, text, math]`
`with parsing selectors [atom]`
`with special reformatter`	

`define bracket "{{" ... "}}"`
`[code, text, math]`
`with parsing selectors`
`[+ math, - code, - text]`
	`+ math`

`define bracket "``" ... "''"` `[data, code, text, math]` `with parsing selectors` `[+ text, - code, - math, - data]` `with text reformatter` `("s", ".", "?", "!", ":", ";")`	`+ text`

3.7.6 Indented Paragraph Subexpressions

Indented paragraph subexpressions are recognized if the '**`enable indented paragraph`**' option is on (p34; this option is always off inside a bracketed subexpression). The following is an example ***indented paragraph subexpression*** using '`:`' as the *indentation-mark*:

```
the first logical line
the second logical line that ends with
          an outer indented paragraph:
    the first logical line of this outer paragraph
    the second logical line of the outer paragraph which
        is continued on this indented line
    the third logical line which ends with
              an inner indented paragraph:
        the first logical line of the inner paragraph
        the second logical line of the inner paragraph
            which is continued on this indented line
```

```
        the first outer paragraph logical line after
            the inner paragraph
        the next logical line, which ends with another
            inner paragraph, but one with zero lines:
        the last logical line of the outer paragraph
    the logical after the outer paragraph
```

An ***indented paragraph subexpression*** begins with an *indentation-mark* (e.g. '`:`') which must be at the end of a physical line or just before a comment that ends a physical line. This is followed by an ***indented paragraph*** consisting of zero or more non-empty (see p70) logical lines called ***paragraph lines***. The paragraph has a ***paragraph indent***, which is the indent of the first non-blank physical line after the line containing the indentation mark, if this indent is greater than the current indent. But if this indent is less than or equal to the current indent, or if there is no non-blank physical line before an end of file, the paragraph is empty, and has no paragraph lines.

Comment lines (lines containing only comment lexemes) are 'logical lines' for the purposes of setting the paragraph indent and terminating an indented paragraph. Thus the following example:

```
a logical line ending with an indented paragraph:
    // comment line setting the paragraph indent
    the first logical line in the paragraph
// comment line terminating the paragraph
the first logical line after the paragraph
```

At the beginning of the paragraph, the ***current indent*** is set to the paragraph indent, and at the end of the paragraph, the current indent is restored to its previous value. The top level current indent is `0`.

The general syntax is:

indented-paragraph-subexpression ::= *indentation-mark line-break*
paragraph-line*
paragraph-end

paragraph-line ::= *logical-line*, possibly ended by a *line-separator*
[May be grouped into headed paragraphs: see 3.7.11 ^{p95}]

logical-line ::= see 3.2 ^{p31} [Includes headed lines: see 3.7.10 ^{p95}]

paragraph-end ::= indent lexeme with indent that is less than the paragraph indent
| end of file

indentation-mark ::= *key* [e.g., '`:`', as per *parser-indentation-mark-command* p70]

line-separator ::= *key* [e.g., '`;`', as per *parser-indentation-mark-command* p70]

Paragraph-lines are logical lines that end according to the setting of the end-at options; See p33. The setting of these options can be controlled by the specification of the *indentation-mark*.

Paragraph-lines may be headed lines (see 3.7.10 ^{p95}) or may be grouped into headed paragraphs (see 3.7.11 ^{p95}).

An indented paragraph subexpression is converted to a **BRACKETED** token whose **value** is a MIN object with list elements derived from the *paragraph-lines* of the paragraph. In the absence of a line header, a *paragraph-line* becomes a logical line object, which is a MIN object with **min:: LOGICAL_LINE()** as its **.initiator** and either the indented paragraph *line-separator* (see below) or **"<LF>"** as its **.terminator**. Headed lines (3.7.10 ^{p95}) have a **.type** instead of an **.initiator**, and either have no **.terminator** or have a *line-separator* as **.terminator**. After lines are parsed, a sequence of lines may be made into a headed paragraph which replaces the sequence of lines in the indented paragraph MIN object. A headed paragraph has a **.type** but no **.initiator** or **.terminator**. The indented paragraph MIN object itself has its *indentation-mark* (see below) as its **.initiator** and **min::INDENTED_PARAGRAPH()** as its **.terminator**.

Empty logical lines with no ending *line-separator* are always ignored as if they did not exist. This rule allows physical lines such as

```
This logical line is followed by an empty logical line;
```

to represent a single logical line, instead of two logical lines the second of which is empty. All retained logical lines are non-empty.

An *indentation-mark* is specified by a **define indentation mark** parser command, and may have an associated *line-separator* supplied by that definition. An example is

define indentation mark ":" ... ";" [code] ...

which specifies '**:**' to be an *indentation-mark* with '**;**' as its corresponding *line-separator* and '**code**' as the sole definition selector.

The general syntax of *parser-indentation-mark-commands* is:

parser-indentation-mark-command
::= **define indentation mark** *indentation-name parsing-selectors*
[**with parsing selectors** *indentation-mark-selectors*]
[**with parsing options** *indentation-mark-options*]
[**with implied header** *header*]
[**with** { **paragraph** | **line** }$^{?}$ **lexical master** *lexical-master-name*]
| **undefine indentation mark** *indentation-mark parsing-selectors*
| **print indentation mark** *partial-indentation-mark*
| **define indentation offset** *indentation-offset*
| **print indentation offset**

NOTE: The **with ...** clauses can appear in any order.

indentation-name ::= *indentation-mark* [**...** *line-separator*]

indentation-mark ::= *quoted-key*

line-separator ::= *quoted-key*

partial-indentation-mark ::= *quoted-key*

quoted-key ::= see p41

indentation-mark-selectors ::= *selector-spec* [p41]
[Unless they are explicitly mentioned in *selector-spec*, '**LINE LEVEL**' is automatically added as a '**+**' selector and '**TOP LEVEL**' is automatically added as a '**−**' selector]

indentation-mark-options ::= *option-spec* with only '**end at ...**' and '**enable ...**' options [p41]

header ::= *typed-prefix-separator* (p84)
with '**paragraph**' or '**line**' group (3.7.10 ^{p95}, 3.7.11 ^{p95})

lexical-master-name ::= *quoted-key* | *simple-name* not containing '**with**'

indentation-offset ::= unsigned integer

The *line-separator* may be used to end paragraph lines, as long as the **end at line separator** (p33) option permits. Line separators are only recognized outside explicitly bracketed subexpressions (untyped bracketed or typed bracketed) in the paragraph.

The '**print indentation mark** *partial-indentation-mark*' command prints all indentation mark symbol table entries whose *indentation-mark* contains the *partial-indentation-mark* as a (not necessarily initial) subsequence of lexemes. Using **""** as a *partial-indentation-mark* will print all indentation mark symbol table entries.

Indentation offsets of all current parser blocks can be printed by the **print indentation offset** command.

The **implied paragraph header** and **lexical master** parameters can be used to initialize line variables that affect the parsing of indented paragraphs: see 3.7.13 ^{p99}. If they are not explicitly included in the indentation mark definition command, they are treated as missing. If a lexical master is missing, the lexical master in effect at the beginning of a *paragraph-line* in an indented paragraph is the lexical master in effect at the end of the previous *paragraph-line*, or if there is no previous *paragraph-line*, the lexical master in effect when the paragraph's indentation mark is scanned.

If '**with lexical master**' is given in the '**define indentation mark**' command, both lexical master parameters are set to the same *lexical-master-name*. If '**with line lexical master**' is given but '**with paragraph lexical master**' is not, both lexical masters are set to the line lexical master and a parser warning message is announced.

If while parsing an indented paragraph a line is encountered whose indent is different from the current indent, but the difference is less than '***indentation offset***' columns (default 2), a parser warning message is announced. No corrective action is taken; the line may continue a logical line or end an indented paragraph according to the usual rules. Here the indentation offset can be changed with the **parser define indentation offset** command. If changed in a

parser block, the original value is reinstated at the end of the block.

When an *indentation-mark* is recognized, the current parsing selector set and parsing options are saved in a stack. Then a new selector set and new options are computed by modifying the saved selector set according to the *indentation-mark-selectors* and modifying the saved options according to the *indentation-mark-options* in the indentation mark definition. The saved selector set and options are restored after the end of paragraph is recognized. The **TOP LEVEL** selector is turned off by default when parsing indented paragraph lines, and the **LINE LEVEL** selector is turned on by default. Note that if no *new-selectors* are given, selectors other than **TOP LEVEL** and **LINE LEVEL** are not modified, and if no *indentation-mark-options* are given, options are not modified.

A new set of line variables is computed for use in parsing the indented paragraph, as per 3.7.13 [p99]. The line variables include the paragraph line separator, implied header, and lexical masters, as given in the indentation mark definition (any of these may be missing).

There is a ***TOP* *LEVEL*** indentation mark definition associated with each active parser block; the definition for the current parser block is the current ***TOP* *LEVEL*** definition. The parameters of this current definition are the top level parameters that can be set and printed as per Section 3.5.3 [p46]; these parameters govern top level parsing as if the top level were inside an indented paragraph with indentation **0**. However, when the parsing selectors for a ***TOP* *LEVEL*** indentation mark are printed, the **TOP LEVEL** and **LINE LEVEL** selectors are omitted, even though both are forced on a the beginning of each logical line.

When the parser is initialized, an initial **TOP LEVEL** parser block is created with its associated ***TOP* *LEVEL*** indentation mark definition. Each **begin block ...** creates the ***TOP* *LEVEL*** definition for its block by copying the current definition from the immediately containing parser block; the parameters of this new definition may then be reset by **define top level** commands as per Section 3.5.3 [p46];

Explicit ***TOP* *LEVEL*** indentation mark definitions are not permitted, but the definitions can be printed by the **print indentation mark ...** command.

Initial Indentation Mark Parser Commands

```
define indentation offset 2
define indentation mark "*PARSER*:" ...  ";" [TOP LEVEL]
       with parsing selectors [data]
define indentation mark "*PARSER* *TEST*:" ...  ";"
                        [TOP LEVEL]
```

Standard ‘`indentation marks`’ Component
Indentation Mark Parser Commands

Parser Command	Given Qualifiers

`define indentation mark ":" ... ";" [code]`
`  with parsing options`
`  [default end at options]`
`  with paragraph lexical master` P
`  with line lexical master` L
	+ code

`define indentation mark ":" ... ";" [text]`
`  with parsing options`
`  [default end at options]`
`  with paragraph lexical master` P
`  with line lexical master` L
`  with implied header {p}`
	+ text

`define indentation mark ":" ... ";" [data paragraph]`
`  with parsing selectors [data]`
`  with parsing options`
`  [default end at options]`	

where

P = `PARAGRAPH-CHECK`	if `+ id, + table`
P = `TABLE-CHECK`	if `- id, + table`
P = `DATA-CHECK`	if `+ id, - table`
P = inherit from context	if `- id, - table`
L = `DATA-CHECK`	if `+ id`
L = inherit from context	if `- id`

‘inherit from context’ means that the
‘`with ... lexical master ...`’ line is omitted

3.7.7 Typed Bracketed Subexpressions

The following are examples of ***typed bracketed subexpressions***, where ... denotes a list of elements:

{my type| ... |}
{my type| ... |my type}
{my type||}
{my type: my attribute = 5, my option = TRUE| ... |}
{my type: my attribute = 5, your option = FALSE| ... |my type}
{my type: my attribute = 5| ... |my option = TRUE: my type}
{my type| ... |my attribute = 5, my option}

```
{my type|  ...  |my attribute = 5, my option:  my type}
                        {|  ...  |}
                          {||}
                           {}
     {my attribute = 5, no your option|  ...  |}
  {"":  my attribute = 5, no your option|  ...  |""}
                        {+  ...  +}
```

The basic form is

```
{my type:  my attribute = 5, my option = TRUE|  ...  |my type}
```

from which various parts can be omitted or moved. Some or all of the attributes can be moved to the end. The attribute value '**= TRUE**' can be omitted (as in '**my option**'), and the attribute value '**= FALSE**' can be represented by omitting it and prefacing the attribute label with '**no**' (as in '**no your option**'). The type can be omitted from the end unless it is preceded by a single attribute with omitted **TRUE** value, as such an attribute will be mistaken for the type (i.e., in **{X|...|Y}**, **Y** is a type label and not an attribute label, and it is an error if **X** and **Y** are not the same). If the type is given at the end, it must match the type given at the beginning. If no type is given at the beginning, the type is missing and must not be given at the end. However, **""** can be used as a type at either end to denote a missing type (e.g., **{|...|""}** is legal and is the same as **{|...|}**).

For example, the following are equivalent:

```
{my option|a b c|no your option}
{my option: no your option|a b c|}
{my option: your option = FALSE |a b c|}
```

where '**my option**' is the **.type**, and not an attribute, while the following is in error because it has a different **.type** name at the end than it does at the beginning:

```
{my option|a b c|your option}
```

A type or attribute label may be a label, which is a sequence of one or more words, naturals, numbers, numerics, and quoted strings, not beginning with a natural or number. Note that marks and separators must be quoted. If there is only one element in the sequence, the MIN string it represents is the label, but if there is more than one element, the elements are concatenated into a MIN label.

Attribute values are similar but may begin with a natural or number, and may alternatively be bracketed subexpressions. In this last case, the attribute value is the bracketed subexpression, with one exception.

The exception is that when the brackets are '**{* *}**', the comma separated elements of the bracketed subexpression are each a distinct value of the attribute, which has a multi-set of values. Elements must be single bracketed subexpressions, words, naturals, numbers, numerics, or quoted strings that are made into MIN strings or MIN numbers, or sequences of words, naturals, numbers, numerics, and quoted strings that are made into labels. Missing elements, as indicated by the commas, are ignored, If there are no values, as in '**{* *}**' or '**{* , *}**', the attribute is not set.

Some examples of attribute values are:

```
            {my type:  x = 5, y = "5" |  ...  |}
{my type:  x = 5 tomatoes, y = [< 5 tomatoes >] |  ...  |}
{my type:  x = {* A, B *}, y = {* , "A",, B, *} |  ...  |}
            {my type:  y = {* ,, *} |  ...  |}
```

In each case, the values of **x** and **y** are equal.

If there are no elements, '**| |**' can be collapsed into a single lexeme '**||**'. '**{||}**' can be collapsed into '**{}**'.

If the type is a single *mark* lexeme and there are no attributes, the '**|**'s surrounding the elements may be omitted, in which case the *mark* is not quoted, but the *mark* must appear both before and after the elements. Similarly if the type is a label consisting of two distinct *mark* lexemes, and there are no attributes, the '**|**'s may be omitted with the two unquoted *marks* being placed before and after the elements, respectively. Except in these cases, type and attribute labels cannot contain separators or marks unless these are quoted.

The following are equivalent pairs of examples with *mark* types:

```
{+ 1 2 3 +}  {"+"| 1 2 3 |"+"}
{< 1 2 3 >}  {"<" ">"| 1 2 3 |"<" ">"}
```

An attribute can have flags which are specified in square brackets following the attribute label. Normally there is only one lexeme in the square brackets and it contains some combination of the 64 flag characters:

```
*+-/@&#=$%<>
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

More flags can be represented by natural number flag indices (e.g. **64**) which are separated from each other and from any flag character by comma separators. Multiple non-number flag lexemes, separated by comma separators, can also be used. Some examples are:

```
{my type: my attribute [H] = (3,4)|A B C|}
{my type: my attribute [H, 64] = (3,4)|A B C|}
{my type: my attribute [64] = (3,4)|A B C|}
{my type: my attribute [64, HJ, 256] = (3,4)|A B C|}
{my type: my attribute [64, H, J, 256] = (3,4)|A B C|}
```

The 64 flag characters listed above are associated with flag indices **0** through **63** in order: '*****' is associated with index **0** and '**Z**' with index **63**. Thus '**[+Y]**', '**[1, Y]**', '**[+, 62]**', and '**[62, 1]**' are equivalent as attribute flag specifiers.

The general syntax is:

typed-bracketed-subexpression
::= *typed-opening* *opening-type-and-attributes*$^{?}$
typed-elements
closing-type-and-attributes$^{?}$ *typed-closing*
| **{** *bracket-type-mark* *element-list* *bracket-type-mark* **}**
| **{** *bracket-type-mark-1* *element-list* *bracket-type-mark-2* **}**
| **{}**

typed-elements ::= *typed-middle* *element-list* *typed-middle*
| *doubled-typed-middle*

element-list ::= see p64

opening-type-and-attributes ::= *bracket-type-label*
| *bracket-type-label* *typed-attribute-begin* *attribute-list*

closing-type-and-attributes ::= *bracket-type-label*
| *attribute-list* but not *attribute-label*
| *attribute-list* *typed-attribute-begin* *bracket-type-label*

attribute-list ::= *attribute* { *typed-attribute-separator* *attribute* }*

attribute ::= *attribute-label* *attribute-flags*$^{?}$
typed-attribute-equal *single-attribute-values*
| *attribute-label* *attribute-flags*$^{?}$
typed-attribute-equal *double-attribute-values*
typed-attribute-equal *reverse-attribute-label*
| *attribute-label* *attribute-flags*
| *attribute-label*
| *typed-attribute-negator* *attribute-label*

label ::= { *word* | *numeric* | *quoted-string* } *label-component**

label-component ::= *word* | *natural* | *number* | *numeric* | *numeric-word* | *quoted-string*
but not *mark* or *separator*

bracket-type-label ::= *label*

bracket-type-mark ::= *mark*

bracket-type ::= *bracket-type-label* | *bracket-type-mark*
| MIN label with elements *bracket-type-mark-1* and *bracket-type-mark-2*

attribute-label ::= *label*

reverse-attribute-label ::= *attribute-label*

attribute-flags ::=

untyped-bracketed-subexpression (p64) with *attribute-flag-items* as *elements*
and *typed-attribute-flags-opening* as *opening-bracket*

attribute-flag-items ::= *attribute-flag-item*$^{?}$ { **,** *attribute-flag-item*$^{?}$ }*

attribute-flag-item ::= *word* or *mark* containing only letters and/or **`*+-/@&#=$%<>`**
| *attribute-flag-index*

attribute-flag-index ::= *natural*

single-attribute-values ::= *value* | *single-attribute-multivalue*

single-attribute-multivalue ::=
untyped-bracketed-subexpression (p64) with *values* as *elements*,
typed-attribute-multivalue-opening as *opening-bracket*,
and **`multivalue`** reformatter (p66)

double-attribute-values ::= *object-id* | *double-attribute-multivalue*

double-attribute-multivalue ::=
untyped-bracketed-subexpression (p64) with *object-ids* as *elements*,
typed-attribute-multivalue-opening as *opening-bracket*,
and **`multivalue`** reformatter (p66)

value ::= *label-component*$^{+}$ | *bracketed-subexpression*

object-id ::= **`@`** *ID-number* | **`@<`***ID-symbol***`>`**
This is a special kind of *label*.

ID-number ::= *non-zero-ASCII-digit* *ASCII-digit**

ID-symbol ::= *ASCII-letter* *ASCII-letter-or-digit**

Most of the following are defined by the *parser-typed-bracket-command* (p80):

typed-opening ::= *key* [e.g. '**`{`**']
typed-middle ::= *key* [e.g. '**`|`**']
typed-closing ::= *key* [e.g. '**`}`**']
typed-attribute-begin ::= *key* [e.g. '**`:`**']
typed-attribute-separator ::= *key* [e.g. '**`,`**']
typed-attribute-equal ::= *key* [e.g. '**`=`**']
typed-attribute-negator ::= *key* [e.g. '**`no`**']
typed-attribute-flags-opening ::= *key* [e.g. '**`[`**']
typed-attribute-multivalue-opening ::= *key* [e.g. '**`{*`**']
doubled-typed-middle ::= *typed-middle* *typed-middle* [e.g. '**`||`**']
as a single lexeme (when possible): see text

The following rules apply to the set of all *attributes* in a single *typed-bracketed-subexpression*:

1. An *attribute* that is just '*attribute-label*' is equivalent to:
attribute-label *typed-attribute-equal* **`TRUE`**

2. An *attribute* that is just '*typed-attribute-negator attribute-label*' is equivalent to:
   attribute-label typed-attribute-equal **`FALSE`**
3. An *attribute* that is just '*attribute-label attribute-flags*' sets just the attribute flags and does not set any attribute values
4. For a given *attribute-label*, at most one *attribute* containing *attribute-flags* may appear.
5. For a given *attribute-label*, at most one *attribute* containing *single-attribute-values* may appear.
6. For a given *attribute-label* and *reverse-attribute-label*, at most one *attribute* containing both labels may appear.

A *typed-bracketed-subexpression* is converted to a MIN object whose list elements are the components of the subexpression between *typed-middles* (e.g., '`|`'s). The *bracket-type*, if it is not missing (and therefore not `""`), becomes the **`.type`** attribute of this object. An *attribute-label* becomes an attribute name of the object, and *single-attribute-values* in an *attribute* become the values of the *attribute*'s attribute name. *Attribute-flags* in an *attribute* become the flags of the *attribute*'s attribute name. *Double-attribute-values* in an *attribute* become the values of the attribute name/reverse attribute name specified by the *attribute*'s *attribute-label* and *reverse-attribute-label*. The MIN object is given the hidden **`.position`** attribute specifying the position in the input text of the *typed-bracketed-subexpression* and the positions of each of its *elements*.

A *typed-bracketed-subexpression* with *bracket-type-marks* has just **`.type`** and **`.position`** attributes, and no other attributes. The **`.type`** is never missing. If the beginning and ending *bracket-type-marks* are identical, the *bracket-type* is just a MIN string equal to these *bracket-type-marks*, but if the two *bracket-type-marks* are not identical, the *bracket-type* is the MIN label whose first element is *bracket-type-mark-1* and whose second element is *bracket-type-mark-2*.

Note that *marks* cannot be part of a *label*, so a *bracket-type-mark* cannot be part of a *bracket-type-label*.

Note that a *typed-bracketed-subexpression* must contain a *typed-middle* (e.g., '`|`'), unless it has two *bracket-type-marks* or is the degenerate form '**`{}`**'. *Typed-prefix-separators* (3.7.8 ^{p83}) are syntactically similar to *typed-bracketed-subexpressions* but have no *typed-middle* and at most one *bracket-type-mark*, and therefore have cannot have any *elements*.

Note that the MIN object is not given **`.initiator`** or **`.terminator`** attributes, unless they are given as explicit attributes as in:

```
{ "": .initiator = "(", .terminator = ")" | 1 2 3 4 |}
```

If two *typed-middles* (e.g., '`|`'s) with no intervening space are lexically scanned as a single lexeme, then this new *doubled-typed-middle* lexeme (e.g., '`||`') can be used to represent an empty list in place of two *typed-middle* lexemes with an intervening space. Note that in this case the *typed-middle* is required to be a single lexeme.

If a typed bracketed subexpression ends pre-maturely, a parsing error is announced and any missing but required *typed-middle* (e.g. '**|**') or final *bracket-type-mark* and *typed-closing* (e.g. '**}**'), are inserted in the input just after the pre-maturely ended subexpression. Such a pre-mature end can be discovered by finding the *closing-bracket* of a containing untyped bracketed subexpression, the *typed-closing* of a containing typed bracketed subexpression with a different *typed-closing*, a *typed-closing* following a missing *typed-middle*, the end of the logical line containing the typed bracketed subexpression, or an end of file. However, the end must be enabled by the parsing options (e.g., a *closing-bracket* cannot pre-maturely end a subexpression if '**end at outer closing**' is not a parsing option).

When a *typed-opening* (e.g. '**{**') is recognized, the current parsing selector set and options become the ***context selectors and options*** of the *typed-bracketed-subexpression*. The current parsing selectors are then reset to the *typed-attribute-selectors* associated with the *typed-opening* (see below), while the options remain the same as the context options but with the '**end at line separator**', '**enable indented paragraph**', and '**enable header**' options turned off. These selectors and options are used to scan the *bracket-type* and *attributes*, but not the *elements*. *Type-attribute-selectors* typically consists of just the **data** selector.

When a *bracket-type-label* is encountered, it is looked up in the bracket type definition table (3.7.9 ^{p86}) using the context selectors. If an entry is found, the selectors and options used to scan the *elements* are computed by modifying the context selectors and options according to the *bracket-selectors* and *bracket-options* in the found entry. If no entry is found, the selectors and options used to scan the *elements* are computed by modifying the context selectors and options according to the *bracket-selectors* and *bracket-options* associated with the *typed-opening* by the **parser define typed bracket** command (see below).

A *bracket-type-mark* is also looked up in the bracket type table when it is encountered, and any entry found is used to modify the context selectors and options used to scan *elements* of its *typed-bracketed-subexpression*. When an initial *bracket-type-mark-1* is not the same as the final *bracket-type-mark-2*, and the two are combined into a MIN label that becomes the *bracket-type*, only the initial *bracket-type-mark-1* is looked up to find the bracket type definition.

The parsing selectors and options are reset to the context selectors and options after the subexpression ending *typed-closing* (e.g. '**}**') is recognized.

A *typed-opening* is specified by a **define typed bracket** parser command. An example is:

```
parser define typed bracket "{" ... "|" ... "|" ... "}"
    [text]
    with parsing selectors [- data]
    with parsing options [- end at outer closing]
    with attribute selectors [data]
    with attributes ":" ... "=" ... ","
    with attribute negator "no"
    with attribute flags initiator "["
    with attribute multivalue initiator "{*"
    with prefix selectors [text]
```

which specifies

‘**{**’ to be a *typed-opening*
‘**|**’ to be a *typed-middle*
‘**||**’ to be by implication the *doubled-typed-middle*
‘**}**’ to be a *typed-closing*
‘**text**’ to be the sole definition selector
‘**[- data]** ’ to be the *bracket-selectors*
‘**[- end at outer closing, - end at line separator,**
- enable indented paragraph, - enable header]’
to be the *bracket-options*;
here ‘**- end at line separator**’, ‘**- enable indented paragraph**’,
and ‘**- enable header**’ are implied.
‘**[data]**’ to be the *typed-attribute-selectors*
‘**:**’ to be the *typed-attribute-begin*
‘**=**’ to be the *typed-attribute-equal*
‘**,**’ to be the *typed-attribute-separator*
‘**no**’ to be the *typed-attribute-negator*
‘**[**’ to be the *typed-attribute-flags-opening*
‘**{***’ to be the *typed-attribute-multivalue-opening*

That a *typed-bracketed-subexpression* with the give *typed-opening* and no elements should not be treated as a *typed-prefix-separator* (see 3.7.8 [p83]) if the context selectors do not have the ‘**text**’ selector.

Note that the *typed-middle* must be repeated twice and both copies must be identical. Also, if *typed-middle* denotes a single string lexeme which when concatenated with itself forms another single string lexeme (as ‘**|**’ concatenated with itself forms ‘**||**’), this second doubled lexeme can be used in place of two consecutive *typed-middles* (as in ‘**{||}**’ being used in place of ‘**{| |}**’).

The general syntax of *parser-typed-bracket-commands* is:

parser-typed-bracket-command

::= **define typed bracket** *typed-bracket-name parsing-selectors*
[**with parsing selectors** *bracket-selectors*]
[**with attribute selectors** *typed-attribute-selectors*]
[**with parsing options** *bracket-options*]
[**with attributes** *typed-attribute-punctuation*]
[**with attribute negator** *typed-attribute-negator*]
[**with attribute flags initiator**
typed-attribute-flags-opening]
[**with attribute multivalue initiator**
typed-attribute-multivalue-opening]
[**with prefix selectors** *typed-prefix-selectors*]
| **undefine typed bracket** *typed-bracket-name*
parsing-selectors
| **print typed bracket** *partial-typed-opening*

typed-bracket-name
::= *typed-opening* **...** **...** *typed-middle* **...** *typed-middle* **...** *typed-closing*
| *typed-opening* **...** *typed-closing*

typed-opening ::= *quoted-key*

typed-middle ::= *quoted-key*

typed-closing ::= *quoted-key*

partial-typed-opening ::= *quoted-key*

quoted-key ::= see p41

bracket-selectors ::= see p65

bracket-options ::= see p65

typed-attribute-selectors ::= *selector-flags* [p41]

typed-attribute-punctuation ::=
typed-attribute-begin **...** *typed-attribute-equal* **...** *typed-attribute-separator*

typed-attribute-begin ::= *quoted-key*

typed-attribute-equal ::= *quoted-key*

typed-attribute-separator ::= *quoted-key*

typed-attribute-negator ::= *quoted-key*

typed-attribute-flags-opening ::= *quoted-key*

typed-attribute-multivalue-opening ::= *quoted-key*

typed-prefix-selectors ::= *selector-flags* [p41]

The '**print typed bracket** *partial-typed-opening*' command prints all typed bracket symbol table entries whose *typed-opening* contains the *partial-typed-opening* as a (not necessarily initial) subsequence of lexemes. Using **""** as a *partial-typed-opening* will print all typed bracket symbol table entries.

The following is an example **define typed bracket** command:

```
define typed bracket "\{" ... "|" ... "|" ... "\}"
                     [code, text, math]
        // `||' abbreviates `| |'
        // there are NO PARSING SELECTORS
        // there are NO PARSING OPTIONS
        with attribute selectors [data]
        with attributes ":" ... "=" ... ","
        with attribute negator "no"
        with attribute flags initiator "["
        with attribute multivalue initiator "\{*"
```

When '**with parsing options**' does not appear in a '`define typed bracket ...`' command, the parsing options used to scan any *elements* are the same as the context options modified by any parsing options found when the *bracket-type* is looked up in the bracket type table. Similarly when '**with parsing selectors**' does not appear in a '`define typed bracket ...`' command, the parsing selectors used to scan *elements* are the same as the context selectors modified by any parsing selectors found when the *bracket-type* is looked up in the bracket type table. Note, however, that '**- TOP LEVEL**' and '**- LINE LEVEL**' are implied by default in *bracket-selectors*, and '**- end at line separator**', ' **- enable indented paragraph**', and ' **- enable header**' are always implied by default in *bracket-options*.

The untyped bracket definitions for '**[...]**' and '**{*...*}**' given on p67, all with the '**data**' selector, are used to parse attribute flags and multivalue lists in any *typed-bracketed-subexpression* recognized by these definitions.

The current implementation supports two syntactic extensions of the above. First, more than 2 **|**'s may be used to switch back and forth between attribute scanning and element scanning, so that:

```
{T: a = 1, b = 2, c = 3, d = 4 | X Y Z | }
{T: a = 1, b = 2, c = 3 | X Y Z | d = 4 }
{T: a = 1, b = 2 | X Y | c = 3 | Z | d = 4 }
{T: a = 1, b = 2 | X Y | c = 3, d = 4 | Z | }
{T| X | a = 1, b = 2 | Y | c = 3, d = 4 | Z | }
```

are all equivalent. Second, in *attribute-flags*, commas do not have to be followed by whitespace, so that:

```
{T: a[PQ@*+,85,203] | X Y Z | }
{T: a[PQ@*+, 85, 203] | X Y Z | }
{T: a[203,P,Q,@,85,*,+] | X Y Z | }
{T: a[203,P, Q,@,85, *,+] | X Y Z | }
```

are all equivalent.

These two syntactic extensions may become permanent or be redacted in the future.

Initial Typed Bracket Parser Command

```
define typed bracket "{" ...  "|" ...  "|" ...  "}" [data]
       // `||' abbreviates `| |'
       // there are NO PARSING SELECTORS
       // there are NO PARSING OPTIONS
       with attribute selectors [data]
       with attributes ":" ...  "=" ...  ","
       with attribute negator "no"
       with attribute flags initiator "["
       with attribute multivalue initiator "{*"
       with prefix selectors []
```

Standard 'brackets' Component Typed Bracket Parser Command

This Command Requires at Least One of `+ code, + text, + math`
Selectors with Missing Standard Qualifiers are Deleted from the Command

```
define typed bracket "{" ...  "|" ...  "|" ...  "}"
                     [code, text, math]
       // `||' abbreviates `| |'
       // there are NO PARSING SELECTORS
       // there are NO PARSING OPTIONS
       with attribute selectors [data]
       with attributes ":" ...  "=" ...  ","
       with attribute negator "no"
       with attribute flags initiator "["
       with attribute multivalue initiator "{*"
```

3.7.8 Typed Prefix Separators

A ***typed prefix separator*** has the same syntax as a *typed-bracketed-subexpression* but with no *typed-middles* (e.g., '**|**'s) and therefore no *elements*, or with only a single *bracket-type-mark* and no *elements*. Some examples are:

{my type}
{my type: my attribute = 5, my option = TRUE}
{my type: my attribute = 5, your option = FALSE}
{my type: my attribute = 5, my option}
{my type: my attribute = 5, not your option}
{+}

The general syntax of a typed prefix separator is:

typed-prefix-separator

::= *typed-opening* *bracket-type-label* *opening-attributes*$^{?}$ *typed-closing*
| *typed-opening* *bracket-type-mark* *typed-closing*

typed-opening ::= see p77

typed-closing ::= see p77

opening-attributes ::= *typed-attribute-begin* *attribute-list*

typed-attribute-begin ::= see p81

attribute-list ::= see p76

bracket-type-label ::= see p76

bracket-type-mark ::= see p76

Typed-prefix-separators and *typed-bracketed-subexpressions* are parsed by the same algorithm which is driven by parameters from the same *parser-typed-bracket-command* for a given *typed-opening* (e.g., '**`{`**'). After the *typed-closing* (e.g., '**`}`**') is read, the parsed subexpression is classified as a *typed-prefix-separator* if all of the following are true:

1. The subexpression does not contain a *typed-middle* (e.g., '**`|`**').
2. The subexpression does not contain *bracket-type-marks*.
3. The subexpression contains a non-missing *bracket-type* (e.g., the subexpression is not '**`{}`**' or '**`{""}`**').

A *typed-prefix-separator* is classified as a ***prefix*** if in addition the following are true:

4. The subexpression context selectors (p79) and the *typed-prefix-selectors* (p81) associated with the *typed-opening* by the *parser-typed-bracket-command* have at least one selector flag in common (the *typed-prefix-selectors* default to all selectors on, so this requirement is met by default).
5. At least one of the following is true:
   (a) The '**`enable prefix`**' option is set in the context options.
   (b) The '**`enable table prefix`**' option is set in the context options and a bracket table entry was found for the *bracket-type*.
   (c) The '**`enable header`**' option is set in the context options and a bracket table entry with '**`paragraph`**' or '**`line`**' group was found for the *bracket-type*.

A *typed-prefix-separator* is converted to a MIN object which has the same structure as a typed bracketed subexpression with an empty element list. In particular, the *bracket-type* of the *typed-prefix-separator* becomes the **`.type`** attribute of the MIN object.

If a *typed-prefix-separator* is a *prefix*, it becomes a **`PREFIX`** token that can head a *prefix-N-list*. Otherwise it becomes a **`BRACKETED`** token by itself that cannot head a *prefix-N-list*, but which can be a *prefix-N-list element*.

Prefixes are used to form *prefix-N-lists*. The following is an example in which the entire text is a single logical line, in spite of the lack of indentation of continuation lines, because `{p}` is a special paragraph prefix with '`+ end at paragraph break`' and '`- end at le indent`' options:

Text	Separator	Syntactic Category
`{p} {s} This is a sentence. {s} And another. {s} And yet another {foo} sentence.`	`{p}`	*prefix-0-list*
`{s} This is a sentence.`	`{s}`	*prefix-1-list*
`This is a sentence.`	(none)	*prefix-2-list*
`{s} And another sentence.`	`{s}`	*prefix-1-list*
`And another sentence.`	(none)	*prefix-2-list*
`{s} And yet another {foo} sentence.`	`{s}`	*prefix-1-list*
`And yet another {foo} sentence.`	(none)	*prefix-2-list* **`{foo}`** is in error and is ignored (deleted).

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

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

except that `{s}` is a special sentence prefix that invokes a the **`sentence`** reformatter (p92) which moves the `"."` into a `.terminator` attribute, so the example is finally equivalent to:

```
{p|  {s: .terminator = "."| This is a sentence |s}
     {s: .terminator = "."| And another |s}
     {s: .terminator = "."| And a yet another sentence |s} |p}
```

If we changed `{p}` to `{P}` and `{s}` to `{S}`, there would nothing special about the prefixes, the continuation lines would have to be indented, and `"."` would not be moved.

The general syntax of a prefix-n list is:

prefix-n-list ::= { *prefix-n prefix-(n+1)-list* }$^{+}$
| *simple-element-list*

prefix-n ::= *prefix*

prefix ::= see text above

simple-element-list ::= see p64

Note: All the *prefix-n*'s in a given *prefix-n-list* must have the same group. The group of a *prefix-n* is its *bracket-type* unless this has a bracket type definition with a non-missing **`group`** member: see p87.

The elements of a *prefix-n-list* are modified by taking each '*prefix-n prefix-(n+1)-list*' component and moving the *prefix-(n+1)-list* elements to the MIN object represented by *prefix-n*. So, for example,

```
{T} X Y Z
```

is just alternate syntax for

```
{T| X Y Z |}
```

3.7.9 Bracket Type Definitions

Unlike brackets and indentation marks, *bracket-types* (p76) do **not** have to be defined in order to be used. But a *bracket-type* can be defined so that it has special effects on parsing, particularly if it is the *bracket-type* of a *typed-prefix-separator*.

Bracket type definitions are stored in the '***bracket type table***' which is one of the parser symbol tables.

When a *typed-bracketed-subexpression* or *typed-prefix-separator* is parsed, its *bracket-type* (i.e., its **`.type`** if it has one) is looked up in the bracket type table using the selectors in effect at the beginning of the subexpression or separator parse (the context selectors). If a table entry is found, it is used to provide information that affects the parse of the *elements* of a *typed-bracketed-subexpression* or the *elements* of a *prefix-n-list* that begins with a *typed-prefix-separator*.

Entries in the bracket type table are managed by *parser-bracket-type-commands* whose general syntax is:

parser-bracket-type-command
::= **`define bracket type`** *bracket-type-name parsing-selectors*
[**`with parsing selectors`** *bracket-type-selectors*]
[**`with group`** *bracket-type-group*]
[**`with implied subprefix`** *implied-prefix*]
[**`with`** { **`paragraph`** | **`line`** }? **`lexical master`** *lexical-master-name*]
[**`with parsing options`** *bracket-type-options*]
[**`with`** *reformatter-name* **`reformatter`** *reformatter-arguments*?]
| **`undefine bracket type`** *bracket-type-name parsing-selectors*
| **`print bracket type`** *partial-bracket-type-name*

bracket-type-name ::= *quoted-key*

partial-bracket-type-name ::= *quoted-key*
quoted-key ::= see p41
bracket-type-selectors ::= *selector-spec* [p41]
bracket-type-group ::= *quoted-key* | *simple-name* not containing '**`with`**'
implied-prefix ::= *typed-prefix-separator* [p84]
lexical-master-name ::= *quoted-key* | *simple-name* not containing '**`with`**'
bracket-type-options ::= *option-spec* [p41]
reformatter-name ::= see p65
reformatter-arguments ::= see p65

The '**`print bracket type`** *partial-bracket-type-name*' command prints all bracket type symbol table entries whose *bracket-type-name* contains the *partial-bracket-type-name* as a (not necessarily initial) subsequence of lexemes. Using **`""`** as a *partial-bracket-type-name* will print all bracket type symbol table entries.

The components special to a bracket type definition are:

`parsing selectors`
The elements of a *typed-bracketed-subexpression* are parsed using the parsing selectors as they were at the beginning subexpression (the context selectors of the subexpression) as modified by any *bracket-type-selectors* found in a bracket type definition of the *bracket-type* of the subexpression, if a bracket type definition is found. If no definition is found, the *bracket-selectors* of the *parser-typed-bracket-command* for the *typed-opening* are used: see p79.

The elements of the *prefix-(n+1)-list* following a *prefix-n* are parsed using the parsing selectors as they were at the beginning of the containing *prefix-n-list* (the context selectors of the *prefix-n*) as modified by any *bracket-type-selectors* found in a bracket type definition of the *bracket-type* of the *prefix-n*. These same modified selectors are also used to parse any subsequent *prefix-m* in the list and look up its **`.type`** in the bracket type table, even if $m \leq n$. So the selector modifications should the chosen so that they do not affect parsing of *typed-prefix-separators* or lookup of bracket type table definitions.

More specifically, consider *prefix-m* in its *prefix-m-list*. Let S_1 be the selectors in effect at the beginning of the entire *prefix-m-list*. Let S_2 be the selectors in effect at the beginning of *prefix-m*, and let S_3 be the selectors in effect just after *prefix-m*. S_2 is used to parse *prefix-m* and look up its bracket type table entry. Then S_1 is modified by the bracket type table entry parsing selectors to produce S_3 (if no bracket type table entry was found, S_3 equals S_1). Note that if *prefix-m* is not at the beginning of its *prefix-m-list*, S_1 and S_2 may not be the same. Note that prefix options below do not have the same considerations as a bracket type table cannot modify options unless it has **`paragraph`** group, and a *prefix-m* with **`paragraph`** group must begin its containing *prefix-m-list* (which is a logical line).

`group`

If the *bracket-type* of a *prefix-n* has a bracket type definition with a non-missing *bracket-type-group*, this *bracket-type-group* is the group of the *prefix-n*. Otherwise the *bracket-type* is the group of the *prefix-n*.

In an expression of the form:

$$\{\ \textit{prefix-n prefix-(n+1)-list}\ \}^{+}$$

the *prefix-n*'s must all have the same group, but need not all have the same *bracket-type*.

The groups '**paragraph**', '**line**', and '**reset**' are special: see 3.7.10 ^{p95}, 3.7.11 ^{p95}, and 3.7.12 ^{p99}.

implied subprefix

If the *bracket-type* of a *prefix-n* has a bracket type definition with a non-missing *implied-prefix*, a copy of this *implied-prefix* is inserted immediately after the *prefix-n*, and becomes the head of a *prefix-(n+1)-list*. If this *prefix-(n+1)-list* list has no elements other than the *implied-prefix*, it is deleted and ignored.

Thus the effect is to insert the *implied-prefix* after the *prefix-n* unless the logical line or bracketed expression containing the *prefix-n-list* ends immediately after the *prefix-n* or unless an explicit prefix whose group is that of a preceding prefix (including the *implied-prefix*) follows the *prefix-n* in the input. However, there is one difference if there is an explicit prefix, and this is that the selectors used to parse the explicit prefix and find any associated bracket type table definition will be those modified by the *implied-prefix* and may differ from those that would be at the same input position had the *implied-prefix* not been inserted.

More specifically, consider a *prefix-m-list* that begins with an implied *prefix-m* followed immediately by an explicit *prefix-m*. Let SO_1 be the selectors and options in effect at the beginning of the entire *prefix-m-list*. Let SO_2 be the selectors and options in effect at the beginning of explicit *prefix-m*, and let SO_3 be the selectors and options in effect just after the explicit *prefix-m*. SO_2, which is SO_1 modified by the bracket type table entry of the implied *prefix-m*, is used to parse the explicit *prefix-m* and look up its bracket type table entry. Then the implied *prefix-m* is deleted (an implied prefix is deleted if its element list is empty) and SO_1 is modified by the explicit *prefix-m*'s bracket type table entry parsing selectors and parsing options to produce SO_3 (if no bracket type table entry was found, SO_3 equals SO_1). Note that prefix options are only affected if the *prefix-m*'s have the **paragraph** group, as only bracket type table entries with that group can have parsing options.

The *implied-prefix* of the bracket type definition will also be ignored unless one of the following is true, where the context options are the parsing options in effect at the point where the *implied-prefix* is to be inserted:

(a) The '**enable prefix**' option is set in the context options.

(b) The '**enable table prefix**' option is set in the context options and a bracket table entry was found for the *bracket-type* of the *implied-prefix*.

(c) The '**enable header**' option is set in the context options and a bracket table entry with '**paragraph**' or '**line**' group was found for the *bracket-type* of the *implied-prefix*.

If the *implied-prefix* is given, it must have the form of a *typed-prefix-separator*, but within its *parser-bracket-type-command* it must be parsed using a definition of its *typed-opening* with empty *typed-prefix-selectors*, so it will be recognized as if it were a *typed-bracketed-subexpression* with a *bracket-type* and no *elements*, as per item 4, p84. In particular, when the parser is initialized, the **TOP LEVEL** parser block is given a suitable typed bracket definition for **"{ ... "|" ... "|" ... "}"** (p83).

parsing options
'**with parsing options**' is only permitted for prefixes that have the '**paragraph**' group. The *bracket-type-options* are used to set the current parsing options and the **options** current line variable when an explicit paragraph header begins a paragraph: see 5 [p102].

line lexical master
'**with line lexical master**' is only permitted for prefixes that have the '**paragraph**' group. This parameter may be non-missing only for isolated paragraph headers and mapped paragraph headers, and is used to set the corresponding current line variable. See 3.7.12 [p99] and 3.7.13 [p99] for details.

3.7.9.1 Bracket Type Reformatters. The following reformatters may be used to change the MIN value represented by a *prefix-n-list* headed by a prefix with a bracket type definition:

data (*clear-sign*, *assign-sign*,
attribute-initiators, *equal-sign*, *negator*, *flags-opening*, *multivalue-opening* **)**

where

attribute-initiators ::= **()** | **(** *attribute-initiator* { **,** *attribute-initiator* }* **)**

The *prefix-n-list* must be a token list that could be represented by a headed logical line of the form:

{*data-prefix***}** *ID* *assign-sign* *element-list* *attribute-paragraph*$^{?}$

or

{*data-prefix***}** *clear-sign*

where

data-prefix ::= the prefix, e.g., '**data**'

ID ::= see p51
element-list ::= see p64
attribute-paragraph ::= *attribute-initiator* *end-of-physical-line* *attribute-line**
attribute-line ::= *attribute* (p76) as a logical line where:

typed-attribute-equal ::= *equal-sign*
typed-attribute-negator ::= *negator*
typed-attribute-flags-opening ::= *flags-opening*
typed-attribute-multivalue-opening ::= *multivalue-opening*

A subexpression of one of these forms typically arises from a logical line that begins with an *ID* or *clear-sign*. For such a line a special lexeme with a special lexeme type and empty character string is prefaced to the line, and the type of this special lexeme is mapped (3.6.2 ^{p53}) to the prefix of the subexpression, e.g., to **`{data}`** or **`{raw data}`**.

In order for this to work with the standard lexical program, the *ID-character* (p52) must be '**`@`**' and the *clear-sign* must be '**`@@@@`**'.

The subexpression must have one of the two above forms and contain either a *clear-sign* or an *assign-sign*. An *ID* must be either a preallocated stub or an object (that is presumed to have replaced a preallocated stub and be identified by an ID). The *prefix* must not have any associated prefix attributes other than **`.type`** or **`.position`**. If these requirements are not met, the **`data`** reformatter takes no action, and the *prefix-n-list* beginning with the *data-prefix* is retained as is in the input, but no error is announced.

Otherwise the subexpression will be reformatted and the entire subexpression will be deleted from the token list. The first form of subexpression defines or adds to the datum denoted by *ID*. The second form clears the symbolic ID table. If *ID* denotes an object, the object is given elements from the *element-list* and attributes from the *attribute-paragraph*.

An subexpression containing the *clear-sign* just clears the symbolic ID table: see p52.

If *ID* is a preallocated stub, it is initialized to an object with no elements or attributes. If *ID* is an object, the object must not have *elements*, *single-attribute-values*, or *attribute-flags*, except that it can have a **`.position`** attribute with its flags. The object may have *double-attribute-values* (produced by data reformatter invocations creating the objects listed in the *double-attribute-values*: see example just below). If these rules are violated, an error is announced and the subexpression is deleted.

In a subexpression containing an *ID* any *attribute-paragraph* before this reformatter is invoked to become a MIN object with *attribute-initiator* as **`.initiator`**, with **`min:: INDENTED_PARAGRAPH()`** as **`.terminator`**, and with logical line elements. Different *attribute-initiators* may be chosen to control how this paragraph is parsed: e.g., are operators recognized or not. This reformatter then further parses these logical lines using its reformatter arguments to treat each line as being formatted in the same manner as an

attribute in a *typed-bracketed-subexpression* (see p76).

Single-attribute-values or *attribute-flags* given for a particular *attribute-label* in an *attribute-paragraph* set the corresponding *attribute-values* or *attribute-flags* with the same *attribute-label* in the object identified by the *ID*.

Double-attribute-values given for a particular *attribute-label* and *reverse-attribute-label* in an *attribute-paragraph* add to the set of corresponding *double-attribute-values* with the same labels in the object identified by the *ID*. If a previous value matches a new value, the new value is not added (i.e., *double-attribute-values* are treated as sets and not multi-sets).

If there is an error in an attribute logical line, the error is announced and the announcement indicates that the logical line or the flags in it or the value in it is being ignored.

A simple example involving *double-attribute-values* is:

```
@1 :=:
    name = Jack
    wife = @2 = husband
@2 :=:
    name = Jill
    husband = @1 = wife
```

Here the line '`wife = @2 = husband`' creates object `@2` and gives it the '`husband = @1 = wife`' attribute so the line '`husband = @1 = wife`' does nothing and is redundant.

As a special case, if there is only one *element*, no *attribute-paragraph*, and *ID* is a preallocated stub, then the only element becomes the value assigned to *ID* in place of the list containing just this element. If this new value of *ID* is an object, a copy of the object replaces the preallocated stub, and previous references to *ID* will become references to this copy (but not to the object copied). Similarly if the new value is a quoted non-identifier string the preallocated stub is replaced by a copy of the non-identifier string (without the quotes).

However, if the new value is neither an object nor a non-identifier string, only future references to *ID* will be mapped to the new value, and previous references to the preallocated stub will continue to reference the untouched preallocated stub. In this case an error is announced and the previous references are left ***dangling*** and unusable.

For example:

@5 := Bill	sets **@5** to	the MIN string '**Bill**'
	instead of	the list whose only element is '**Bill**'
		previous uses of **@5** are dangling
@5 := (a b c)	sets **@5** to	(a b c)
	instead of	**{| (a b c) |}**
@5 := {| a b c |}	sets **@5** to	**{| a b c |}**
	instead of	**{| {| a b c |} |}**
@5 := a b c	sets **@5** to	**{| a b c |}**
	which is NOT a special case	

Dangling references will not cause garbage collection errors, but the **data** reformatter will generate a parser error message. Dangling references print as preallocated stubs with their ID number and a counter equal to one plus the total number of dangling references with that ID number. This information can be used for finding dangling references in the input.

In the unusual case where the *element-list* ends in a paragraph with the same paragraph initiator as an *attribute-paragraph*, and there is no *attribute-paragraph*, an empty *attribute-paragraph* must be added to make the *element-list* end as intended.

sentence *terminator-list*

where

terminator-list ::= **()** | **(** *quoted-terminator* { **,** *quoted-terminator* }* **)**
quoted-terminator ::= **"***terminator***"**
terminator ::= *mark* | *separator*

The *prefix-n-list* must have the form:

{*prefix prefix-attributes*$^{?}$**}** *element-list*

where

prefix ::= the prefix, e.g., '**s**'
prefix-attributes ::= *attribute-type-begin attribute-list* (see p76)
element-list ::= see p64

The *element-list* is parsed according to the rules:

element-list ::= *sentence** *phrase*$^{?}$
sentence ::= *phrase*$^{?}$ *terminator*
phrase ::= *phrase-element*$^{+}$
phrase-element ::= *element* other than *terminator*

The **sentence** reformatter converts the subexpression to a sequence of subexpressions,

one for each *sentence* and one for any non-empty final *phrase*. Each of these subexpressions has as elements the *phrase-elements* of the corresponding *sentence* or *phrase*, and has as attributes copies of the attributes of the original subexpression. In addition, each subexpression corresponding to a *sentence* has a **`.terminator`** attribute equal to the *terminator* of the *sentence*.

Initial Bracket Type Parser Commands

(none)

Standard '`bracket types`' Component Bracket Type Parser Commands That Do NOT Require Any Standard Qualifier

Selectors with Missing Standard Qualifiers are Deleted from the Commands

```
define bracket type "***" [TOP LEVEL, code, text]
       with group reset
define bracket type "data"
                [TOP LEVEL, data, code, math, text]
       with group line
       with parsing selectors [+ data paragraph]
       with data reformatter ( "=", "no", "[", "{*" )
define bracket type "raw data"
                [TOP LEVEL, data, code, math, text]
       with group line
       with parsing selectors [data, data paragraph]
       with data reformatter ( "=", "no", "[", "{*" )
```

Standard '`bracket types`' Component Bracket Type Parser Commands That REQUIRE the `+ text` Standard Qualifier

Selectors with Missing Standard Qualifiers are Deleted from the Commands

```
define bracket type "table" [code, text]
       with parsing selectors [text]
       with group paragraph
       with line lexical master ROW-CHECK
define bracket type "row" [text]
       with group line
```

Standard '`bracket types`' Component Bracket Type Parser Commands That REQUIRE the `+ text` Standard Qualifier

Selectors with Missing Standard Qualifiers are Deleted from the Commands

```
define bracket type "section" [code, text]
       with group paragraph
       with parsing selectors [+ text,
                               - code, - math, - data]
       with parsing options [+ end at lt indent,
                            + end at paragraph break,
                            - end at indent,
                            - end at le indent,
                            - end at line separator,
                            - end at outer closing]
       with implied subprefix s
define bracket type "p" [code, text]
       with group paragraph
       with parsing selectors [+ text,
                               - code, - math, - data]
       with parsing options [+ sticky,
                            + end at lt indent,
                            + end at paragraph break,
                            - end at indent,
                            - end at le indent,
                            - end at line separator,
                            - end at outer closing]
       with implied subprefix s
define bracket type "s" [text]
       with sentence reformatter
            ( ".", "?", "!", ":", ";" )
define bracket type "quote" [code, text]
       with group paragraph
       with parsing selectors [+ text,
                               - code, - math, - data]
       with parsing options [+ sticky,
                            + end at lt indent,
                            + end at paragraph break,
                            - end at indent,
                            - end at le indent,
                            - end at line separator,
                            - end at outer closing]
       with implied subprefix s
```

3.7.10 Headed Lines

A '***line header***' is a typed prefix separator that has the '**line**' group. Line headers may only appear at the beginning of a logical line, or immediately after a paragraph header, which itself must begin a logical line: see 3.7.11 [p95].

Line headers may be implied by the **implied subprefix** of a paragraph header (see p88 and 3.7.11 [p95]) or by the **implied header** line variable (see p100).

Line headers, including implied line headers, are only recognized if the **enable header** option is on at the beginning of the logical line.

A '***headed line***' is a logical line beginning with a line header. The parser's output for a headed line is a MIN object whose **.type** and attributes are taken from the line header and whose elements are taken from the rest of the headed line, except that if the logical line is ended by a *line-separator* this becomes he **.terminator** attribute of the MIN object. The MIN object has a **.type** but no **.initiator**, unlike the parser's output for a logical line that has no line header, which has no **.type** but has an **.initiator** equal to the special value **min::LOGICAL_LINE()**.

3.7.11 Headed Paragraphs

A '***paragraph header***' is a typed prefix separator that has the '**paragraph**' group. Paragraph headers may only appear at the beginning of a logical line in paragraph beginning position, that is, following a blank line or the beginning of an indented paragraph or file (where intervening comment lines are ignored).

Paragraph headers may be implied by the **implied header** line variable (see p100).

Paragraph headers, including implied paragraph headers, are only recognized if the **enable header** option is on at the beginning the paragraph header's logical line.

A '***headed paragraph***' is a logical line that begins with paragraph header, plus all the following logical or headed lines up until the headed paragraph end. A headed paragraph may be ended by a blank line (unless the paragraph is 'continuing'), or by the end of its containing indented paragraph, or by the end of the input file.

The parser's output for a headed paragraph is a MIN object whose **.type** and attributes are taken from the paragraph header and whose elements are the logical or headed lines in the headed paragraph. If the first logical line of the headed paragraph contains only the paragraph header, that logical line is omitted from the list of elements (so there are no blank logical lines).

As an example, suppose the parser is given the definitions.

```
*PARSER*:
    define bracket type "itemize" [code, text]
        with group paragraph
    define bracket type "item" [code, text]
        with group line
```

Then when given the input (with '**code**' or '**text**' a top level parsing selector):

```
{itemize: indent = 5em, mark = "*"}
{item}1 chicken
{item}1 tablespoon olive oil
{item}1 tablespoon paprika
```

the parser produces the MIN object:

```
{itemize: indent = 5em, mark = "*"|
    {item| 1 chicken |item}
    {item| 1 tablespoon olive oil |item}
    {item| 1 tablespoon paprika |item}
    |itemize}
```

Now suppose the parser is given the definitions:

```
*PARSER*:
    define bracket type "itemize" [code, text]
        with group paragraph
        with implied subprefix {item}
    define bracket type "item" [code, text]
        with group line
```

with a '**implied subprefix**'. Then when given given the input:

```
{itemize: indent = 5em, mark = "*"}
1 chicken
1 tablespoon olive oil
1 tablespoon paprika
```

the parser will produce the same MIN object.

Next suppose the parser is given the definitions:

```
*PARSER*:
    define bracket type "itemize" [code, text]
        with group paragraph
        with implied subprefix {item}
        with parsing options [+ sticky]
    define bracket type "item" [code, text]
        with group line
```

with a '**sticky**' option. Then when given given the input:

```
{itemize: indent = 5em, mark = "*"}
1 chicken
1 tablespoon olive oil
1 tablespoon paprika
```

```
a potato
1 tablespoon butter
```

the parser will produce two MIN objects, the first being as above and the second being:

```
{itemize: indent = 5em, mark = "*"|
    {item| a potato |item}
    {item| 1 tablespoon butter |item}
    |itemize}
```

If instead the parser is given the definitions:

```
*PARSER*:
    define bracket type "itemize" [code, text]
        with group paragraph
        with implied subprefix {item}
        with parsing options [+ continuing]
    define bracket type "item" [code, text]
        with group line
```

with a '**continuing**' option, then when given the above input the parser would produce the single MIN object:

```
{itemize: indent = 5em, mark = "*"|
    {item: 1 chicken}
    {item: 1 tablespoon olive oil}
    {item: 1 tablespoon paprika}
    {item: a potato}
    {item: 1 tablespoon butter}}
```

Note that the '**sticky**' and '**continuing**' options remain in effect until the next explicit (non-implied) paragraph header, or until the end of a containing indented paragraph.

The '**standard**' parser block contains the definitions:

```
*PARSER*:
    define parsing selectors [+ code]
    define top level paragraph lexical master PARAGRAPH-CHECK
    define mapped lexeme "TABLE" [TOP LEVEL]
           with token value {table}
    define mapped lexeme "ROW" [TOP LEVEL]
           with token value {row}
    define bracket type "table" [code, text]
        with parsing selectors [text]
        with group paragraph
        with line lexical master ROW-CHECK
    define bracket type "row" [text]
        with group line
```

To see how these definitions work, consider the example:

```
=============
| 1 chicken |
-------------
| 1 tablespoon olive oil |
-
| 1 tablespoon paprika
=
```

where the first line appears in paragraph beginning position (p99). Therefore at the beginning of the first line the top level paragraph lexical master, which is **PARAGAPH-CHECK**, is used. As this sees a string of 5 or more **=**'s, it outputs a **TABLE** special lexeme which is mapped to the **{table}** prefix. The '**table**' prefix in turn has a definition which changes the line lexical master to **ROW-CHECK**, and this is also used for the rest of the first line. As none of the **=**'s have been consumed by the **TABLE** special lexeme, this **ROW-CHECK** lexical master sees the first **=** and outputs a **ROW** special lexeme, which is mapped to a **{row}** prefix. After outputting the **ROW** special lexeme, the **ROW-CHECK** lexical master transfers to the **DEFAULT** lexical master which scans the normal lexemes in the line. Since neither special lexeme consumed any **=**'s, the one normal lexeme in the line contains all the **=**'s. At the beginning of the next logical line, the top level line lexical master defined by the paragraph header **{table}** is installed, which is **ROW-CHECK**. The final result is the same as if

```
{table}{row}=============
{row}| 1 chicken |
{row}-------------
{row}| 1 tablespoon olive oil |
{row}-
{row}| 1 tablespoon paprika
{row}=
```

had been input and is

```
{table| {row| ============= |row}
        {row| "|" 1 chicken "|" |row}
        {row| ------------- |row}
        {row| "|" 1 tablespoon olive oil "|" |row}
        {row| - |row}
        {row| "|" 1 tablespoon paprika |row}
        {row| = |row}
        |table}
```

At top level the paragraph and line lexical masters are set by the '**define top level**' commands, and the only way to get an implied header is to input a paragraph header with the '**sticky**' option. In contrast, an indented paragraph can begin with a paragraph and line lexical masters that are either set by the its '**define indentation mark**' command or are inherited from the

surrounding context, and the '**define indentation mark**' can optionally set an implied paragraph or line header. See 3.7.13 [p99] for details.

3.7.12 Isolated Headers

An ***isolated header*** is a reset header, or is a paragraph header that resets the **line lexical master** and is not an implied prefix or mapped prefix. A ***mapped prefix*** is an explicit prefix that was created when a mapped lexeme was mapped to a prefix. A ***mapped header*** is a mapped prefix that is a paragraph header. Isolated headers and mapped headers are the only headers that can reset the line lexical master to the value in the header's bracket type definition. A ***reset header*** is a prefix with the '**reset**' group.

An isolated header must appear at the beginning of a logical line and must be the only thing in its logical line (in particular, the logical line cannot end with a line separator). In determining the extent of the logical line containing the header, the **options** current line variable value modified by parameters from the header's bracket type definition is used.

An isolated paragraph header sets the current line variables to those of the containing indented paragraph or the top level if there is no containing indented paragraph, as modified by the parameters in the header's bracket type definition (one of which is the non-missing line lexical master).

A reset header sets the paragraph line variables to those of the containing indented paragraph or the top level if there is no containing indented paragraph.

An isolated paragraph header may not be implied or sticky.

A reset header's bracket type definition cannot have any parameters other than '**group**'.

A mapped paragraph header may set the line lexical master current line variable without the requirement that the mapped paragraph header be in a logical line by itself. The reason is that the special lexical master that produced the mapped paragraph header is assumed to jump to the line lexical master after producing the paragraph header, so the line lexical master takes effect immediately after the mapped paragraph header in its logical line. However this assumption is not verified by code.

3.7.13 Line Variables

In order to parse logical lines, the parser maintains sets of '***line variables***'. There is a separate set of line variables for each indented paragraph, and one additional set for the top level.

The line variables are described below. These descriptions make use of the notion that a logical line may be in '***paragraph beginning position***'. This means that the logical line follows a blank line, or is at the the beginning of its containing indented paragraph, or is at the beginning of the input file, with comment lines possibly intervening.

While parsing a logical line the parser uses the line variables:

selectors
The parsing selectors are reset to this value just after the indent token that begins a logical line.

options
The parsing options are reset to this value just after the indent token that begins a logical line.

implied header
May be missing or set equal to a paragraph or line header.

If this value is not missing a copy of the value is inserted as an implied header just after the indent token that begins a logical line. An implied header is an implied subprefix (p88) and is almost like an explicit prefix separator except that if it has no following *prefix-n-list* elements it is removed. In particular, if an implied header is immediately followed by an explicit (i.e., not implied) header with the same group, the implied header is removed.

A non-missing implied header must have a bracket type definition selected by the **selectors** line variable that has either the '**paragraph**' or '**line**' group.

Modifications to selectors and options made by an implied header should be chosen so that they do not affect the parsing of any explicit header that occurs immediately after the implied header, and do not affect the lookup of the bracket type of that explicit header. This is because if the logical line begins with an explicit header, that will be parsed and its bracket type table definition will be looked up using the selectors and options set by the implied header, and then the implied header will then be deleted (as there are no elements between it and the explicit header). But after the implied header is deleted, the explicit header is not re-parsed and its bracket type table definition is not re-looked-up.

paragraph lexical master
line lexical master
The lexical master is reset to one of these values just after the indent token that begins a logical line, unless the value is missing. The paragraph lexical master is used for logical lines in paragraph beginning position (p99) and the line lexical master is used for other logical lines.

By convention, non-default lexical masters revert to the default master after identifying any special lexemes at the location where the non-default lexical master is installed. If the lexical master is not set at the beginning of a logical line, by this convention the default lexical master will be in effect.

Note: The above line variables are called the '***current* line variables**'.

Note: Explicit paragraph headers and reset headers change the line variables, except for the paragraph lexical header. A change to the line lexical header does not take effect until the beginning of the logical line immediately after the logical line containing the explicit paragraph or reset header, so use of these headers is restricted: see 3.7.12 ^{p99}.

paragraph selectors
paragraph options
paragraph implied header
paragraph paragraph lexical master
paragraph line lexical master

These variables are called the '***paragraph line variables***'.

The paragraph line variables are copied to corresponding current line variables just before the current line variables are used for a line in paragraph beginning position, unless the **options** current line variable contains the '**continuing**' option.

The top level paragraph line variables, with the exception of the paragraph implied header, are the same as the parser selectors, options, and lexical masters, and can be changed by the **define top level** parser commands (p42). The top level paragraph implied header is normally missing, but may be changed by a '**sticky**' paragraph header as indicated below.

For indented paragraphs, paragraph selectors and options line variables are computed from the context of the indented paragraph modified according to parameters given in the the **define indentation mark** parser command (p70) for the mark that introduces the paragraph. The paragraph lexical masters and implied header line variables are set from the values provided in the **define indentation mark** command. The paragraph implied header may be changed while parsing the indented paragraph if a '**sticky**' explicit paragraph header is encountered.

An explicit paragraph header that sets the '**sticky**' option replaces the paragraph implied header with a copy of itself. An explicit paragraph header that clears the '**sticky**' option replaces the paragraph implied header with the missing value at top level, or with the value from the **define indentation mark** command within an indented paragraph.

Logical line processing is as follows:

1. A sequence of logical lines whose first line (and only the first line) is a logical line prefixed by a paragraph header is collected into a headed paragraph. Such a sequence ends (1) just before a logical line in paragraph beginning position if the **options** current line variable does not have the '**continuing**' option, or (2) just before a logical line that begins with an explicit paragraph header (such a line must be in paragraph beginning position), or (3) just before the end of the current indented paragraph or, at top level, the end of file.

2. If a logical line is in paragraph beginning position and the **options** current line variable does not have the '**continuing**' option, then the paragraph line variables are copied to the corresponding current line variables.

3. The parsing selectors, options, and lexical master used at the beginning of the logical line are set from the current line variables, specifically, the **selectors**, **options**, and **paragraph/line lexical master** current line variables.

4. If the **implied_header** current line variable value is not missing, the bracket type definition for this is looked up (using selectors now equal to the **selectors** current line variable). The bracket type definition must exist and have either the '**paragraph**' or '**line**' group. Note that an isolated header cannot be implied.

   The **implied_header** is treated as missing unless one of the following options is present in the **options** current line variable:

   enable prefix
   enable table prefix
   enable header

   If the bracket type definition has the '**paragraph**' group, the implied prefix is an implied paragraph header. If it has the '**line**' group, the implied prefix is an implied line header.

   If the **implied_header** has the '**paragraph**' group and the logical line is not in paragraph beginning position, an error is announced and the **implied_header** is treated as missing.

   A non-missing implied header is inserted at the beginning of the logical line. Then the parsing selectors and parsing options in its bracket type definition are used to modify the current selectors and options. In addition, if the implied header has '**paragraph**' group, the new selectors and options are stored in the current line variable selectors and options, and the implied subprefix of the bracket type definition is stored in the current line variable implied header variable if it has the '**line**' group when its bracket type is looked up with the new selectors current line variable value (this implied subprefix will also be inserted after the implied paragraph header in its logical line). These new current line variable values are used for subsequent lines until the current line variables are reset from the paragraph line variables (e.g., at beginning of the next logical line that is in paragraph beginning position).

5. The first part of the rest of the logical line is then parsed. If it begins with a prefix that has a bracket type definition with either the '**paragraph**', '**line**', or '**reset**' group, then that is an explicit paragraph, line, or reset header.

   If an explicit paragraph or reset header is found immediately after implied headers, the implied headers are deleted. If an explicit line header is found immediately after implied headers, an implied line header is deleted, but an implied paragraph header is not deleted.

   When implied headers are deleted, the parsing selectors, options, and current line variables are restored to their state before the deleted implied headers were inserted. However the selectors and options used to parse the explicit header and look up its bracket type definition are not the restored parsers and options, but rather the selectors and options established by the implied headers. Therefore implied headers should not change selectors and options sufficiently to affect parsing explicit headers or looking up their bracket type definitions.

   If a line or non-isolated paragraph explicit header is found, then after any implied headers are deleted and the state set by the current indentation mark or top level definition has been

restored, the explicit header's bracket type definition is used to modify the parsing selectors and options, and if the header is a paragraph header, the current line variable selectors, options, and implied header, and if in addition the header is a mapped paragraph header with a non-missing line lexical master, the current line variable line lexical master. Here the implied header is copied from the explicit header's bracket type definition only if the explicit header is a paragraph header and the implied header has the '**`line`**' group when its bracket type definition is looked up with the new selectors current line variable.

If an isolated paragraph header is found, the header's bracket type definition modifies the current line variables selectors, options, line lexical master, and implied header. Since an isolated header must be on a logical line by itself, processing proceeds to the next logical line.

If a reset header is found, the selectors, options, and line variables are just restored to the values they were set to by the current indentation mark definition or top level definition. Then the reset header is deleted, and its line is treated as a blank line, so the following logical line is in paragraph beginning position.

6. If an explicit paragraph header sets the '**`sticky`**' option flag, then the **`paragraph implied header`** is set to a copy of the explicit paragraph header. Note that an isolated header cannot be implied and therefore cannot be sticky.

   If the explicit paragraph header does not set the '**`sticky`**' option flag, then the **`paragraph implied header`** is reset to the value it had at the beginning of the current indented paragraph, or to missing if the logical line is top level.

7. The remainder of the logical line is then parsed. If the logical line is empty except for implied headers, any implied headers are deleted and the logical line is deleted.

Note that parsing definitions cannot change during the parsing of a logical line, an indented paragraph inside a logical line, or any headed paragraph. Line variables, however, may change, and new sets of line variables are created by indented paragraphs.

3.8 Parser Passes

After a subexpression has been identified by the top level bracketed subexpression recognition pass, a sequence of passes is run on the subexpression. Which passes are in the sequence is determined by the parser pass stack and parser selectors.

The ***parser pass stack*** is a list of parser passes with a set of selectors associated with each pass. A pass in this list is active if it has a selector in common with the parsing selectors computed by the top level bracketed subexpression recognition pass. Each pass calls the next active pass in the parser pass stack on subexpressions recognized by the calling pass. For passes like the operator pass, this is done on subexpressions recognized by the pass which contain no operators, and after

the called pass returns, the operator pass compacts any subexpression with zero or more than one token into a single **PURELIST** token.

The parsing selectors computed by the opening bracket of the bracketed subexpression become the parsing selectors while the subexpression is being parsed. These are used to determine activity of the parser passes run on the bracketed subexpression and also the activity of the parser definitions (i.e., parser symbol table entries) used by these parser passes. Note that only the brackets surrounding a bracketed subexpression can change parsing selectors, and then can only change them within the bracketed subexpression. Operators that bound implicit subexpressions cannot change these parsing selectors.

The parser pass stack can be altered and inspected by the following parser definitions:

parser-pass-command
::= **define pass** *parser-pass-name parsing-selectors parser-pass-stack-location*
| **undefine pass** *parser-pass-name*
| **print pass**

parser-pass-stack-location ::= **after** *previous-parser-pass-name*
| **before** *next-parser-pass-name*
| **at end**

parser-pass-name ::= *simple-name*

previous-parser-pass-name ::= *parser-pass-name*

next-parser-pass-name ::= *parser-pass-name*

The set of passes that may be run is builtin, and cannot be changed. The following are permitted *parser-pass-names*:

top
This refers to the bracketed subexpression recognition pass which is always at the top of the parser pass stack, and can only be used as a *previous-parser-pass-name* to place a pass just below it on the stack.

operator
Parses expressions with computational operators (e.g. **+** and *****).

lexeme replacement
Replaces sequences of lexemes with other sequences of lexemes (e.g., replaces plurals by singulars).

radix number recognition
Recognizes numbers with non-decimal radixes.

scientific number recognition
Recognizes numbers with exponents.

number pair recognition
Recognizes pairs of numbers (e.g. **4 1/2**).

number unit grouping
Groups numbers and numeric units (e.g. **4ft 5in**).

unit multiplication insertion
Inserts multipliers between numbers and numeric units (e.g. **4*ft** and **$*4.99**).

The **parser define pass** statement sets the *parsing-selectors* of the named parser pass and installs that pass in the parser pass stack at the location specified. If the pass was previously on the stack, it is removed from the stack and then reinserted into the stack without altering any pass-specific symbol tables. The *parser-pass-name* and *next-parser-pass-name* cannot be '**top**'. Any *previous-parser-pass-name* or *next-parser-pass-name* given must name a pass already in the stack that is different from the pass being installed.

The **parser undefine pass** statement removes the named parser pass from the stack, if it is in the stack, and does nothing otherwise. The '**top**' pass cannot be undefined.

The **parser undefine pass** statement destroys any symbol tables associated with the pass being undefined. First undefining and then redefining a pass effectively clears this symbol table. Thus first undefining and then redefining the '**operator**' pass clears the symbol table set by '**parser define operator ...**' commands. On the other hand, a '**parser define pass operator ...**' statement may be used to change the '**operator**' pass selectors or position in the pass stack without clearing the operator symbol table.

The **parser print pass** statement prints the parser pass stack passes with their associated selectors.

The input and output of a parser pass is a sublist of the list of all tokens. When a parser pass is called, it is provided with a pointer to the first token of this sublist, and a pointer to the first token after the sublist. The pass may edit the sublist. The pass is responsible for calling the next pass down in the parser pass stack, and may edit the sublist before and/or after calling this next pass.

The bracketed subexpression recognition pass, also known as the '**top**' pass, which is the first pass called for each top level input line, recognizes bracketed subexpressions, and for each recognized bracketed subexpression calls the next active lower pass for the token list of the subexpression and then replaces this token list, which may have been edited by the called lower pass, by a single MIN object.

3.9 The Operator Parsing Pass

The ***operator parsing pass*** is an expression parser pass that uses operators to restructure expressions. Operators are defined by operator definitions that can be added to the operator parsing definition stack. List separators, such as '**,**', are treated as operators, and have operator definitions.

3.9.1 Operator Expression Syntax

The operator parsing pass identifies operators and parses expressions using operator flags and operator precedence. Operator definitions assign flags (prefix, infix, postfix, etc.) and integer precedences to operators.

The operator flags **`initial`**, **`left`**, **`right`**, **`final`**, **`afix`**, and **`line`** are described in Figure 14. Flag groups are names used in operator definitions for sets of flags. For example, a **`prefix`** operator is an operator with the **`initial`** and **`right`** flags, a **`postfix`** operator is an operator with the **`left`** and **`final`** flags, and an **`infix`** operator is an operator with the **`left`** and **`right`** flags.

Operators have precedences in the range [L,H], where L = -1,000,000 and H = +1,000,000. By convention, precedence H is reserved for postfix operators and precedence H-1 is reserved for prefix operators, but this is just convention and some prefix or postfix operators may have other precedences. The precedence L-1 is reserved for the 'error operator' which is inserted during parsing to 'fix up' errors found during parsing.

Given this, expressions have the following syntax, where an *P-expression* is an expression all of whose operators that are outside brackets have precedence equal to or greater than P:

expression ::= *(L-1)-expression*

P-expression ::= *P-final-expression*
| *P-initial-operator P-expression*$^{?}$

P-final-expression ::= *P-middle-expression*
| *P-final-expression*$^{?}$ *P-final-operator*

P-middle-expression ::= { *(P+1)-expression* | *P-middle-operator)* }$^{+}$

(H+1)-expression ::= *non-operator*$^{+}$

P-operator ::= operator of precedence P

P-initial-operator ::= *P-operator* with **`initial`** flag

P-final-operator ::= *P-operator* with **`final`** flag

P-middle-operator ::= *P-operator* with neither **`initial`** nor **`final`** flag

where in an *P-expression*:
P is any precedence in the range [L-1,H];
no two *(P+1)-expressions* may be adjacent;
each *P-operator* with a **`left`** flag must be preceded by an *(P+1)-expression*;
each *P-operator* with a **`right`** flag must be followed by an *(P+1)-expression*;
any operator with an **`afix`** flag must not be the first *P-operator* in an *P-expression*;
no operator may have both a **`initial`** and a **`left`** flag;
no operator may have both a **`right`** and a **`final`** flag;
no operator may have both a **`initial`** and an **`afix`** flag;

Essentially the expression being parsed is organized into *P-expressions* where P is the precedence

of the *P-expression*. Generally a *P-expression* consists of a sequence of *(P+1)-expressions* and operators of precedence P. If a *P-expression* begins with a *P-initial-operator*, the entire *P-expression* after its initial operator is an operand of the initial operator. If a *P-expression* does not beginning with a *P-initial-operator* ends with a *P-final-operator*, the entire *P-expression* before its final operator is an operand of the final operator.

Whether or not an operator is recognized is determined by the left context of the operator. Thus in '`x + + y`' the first + is recognized as an infix operator because it is preceded by an operand, and the second + is recognized as a prefix operator because it is preceded by another operator (the first +) of lower precedence. However, in '`x * * y`' the second $\star$ is not recognized as an operator, because only infix $\star$ is defined, and what is immediately to the left of the second $\star$ is another operator (the first $\star$) of equal precedence. Thus the second $\star$ is treated as a non-operator, (e.g., as if there were a variable named '`* y`').

A consequence of this is that while an operator may have both an infix and a prefix definition, it should not have both an infix and a postfix definition, as both infix and postfix operators both have the same left context (and in that context whichever was defined later will be selected).

There can be multiple operator definitions with the afix flag and the same *operator-name* but with different precedences that are all selected by the current parsing selectors. The most recent definition will be used that has the same precedence as a previous operator in the expression with no intervening operator of lower precedence.

3.9.2 Operator Commands

Parser-operator-commands modify and print the operator parsing definition stack:

parser-operator-command ::= *parser-operator-definition*
| *parser-operator-print-command*

parser-operator-definition
::= **define operator** *operator-name* *parsing-selectors* *operator-flag* *operator-flag**
with precedence *precedence*
[**with** *reformatter-name* **reformatter** *reformatter-arguments*$^?$]
| **undefine operator** *operator-name* *parsing-selectors* *operator-flag* *operator-flag**
with precedence *precedence*

operator-name ::= *simple-operator-name*
| **bracket** *bracket-name*
| **indentation mark** *indentation-mark-name*

simple-operator-name ::= *quoted-key*
quoted-key ::= see p41
bracket-name ::= see p64

Operator Flags:

`initial` Operator must be the first thing in any subexpression containing it.

`left` Operator must follow an operand (e.g., a non-operator or a subexpression formed with operators of higher precedence). That is, operator must have a left operand.

`right` Operator must precede an operand (e.g., a non-operator or a subexpression formed with operators of higher precedence). That is, operator must have a right operand.

`final` Operator must be the last thing in any subexpression containing it.

`afix` Operator must be preceded in any subexpression containing it by another operator of the same precedence.

`line` Current parsing selectors must contain **`LINE LEVEL`** in order for the operator to be recognized.

Operator Flag Groups:

`prefix` The flags **`initial`** and **`right`**.

`infix` The flags **`left`** and **`right`**.

`postfix` The flags **`left`** and **`final`**.

`nofix` No flags.

Figure 14: **Operator Flags and Flag Groups**

indentation-mark-name ::= see p70

operator-flag ::= **`initial`** | **`left`** | **`right`** | **`final`** | **`afix`** | **`line`** | *operator-flag-group*

operator-flag-group ::= **`prefix`** | **`infix`** | **`postfix`** | **`nofix`**

precedence ::= *integer* in the range [L, H]
where L = -1,000,000 and H = +1,000,000

reformatter-name ::= see p65
reformatter-arguments ::= see p65

parser-operator-print-command
::= **`print operator`** *partial-operator-name*

partial-operator-name ::= *quoted-key*

An ***operator definition*** specifies for each operator the following:

Name and Selectors
Operator Flags

Precedence
Reformatter

A *simple-operator-name* is matched to lexemes in a subexpression in order to identify occurrences of the operator. The other forms of *operator-name* permit bracketed subexpressions with particular kinds of brackets to be parsed as if they were operators. Thus in '**x[5] = 0**' the bracketed subexpression '**[5]**' can be a postfix operator, and in:

```
int y:
    if x == 0:
        y = 5
    else:
        y = 6
```

the indented paragraph:

```
            :
        y = 5
```

introduced by the indentation mark '**:**' can be an afix operator which with the prefix '**if**' operator brackets the *conditional-expression* '**x == 0**'.

When a bracketed subexpression (e.g., **[...]**) is identified as an operator, only the opening bracket (e.g., '**[**') is checked. Since it is unusual for two defined brackets to have the same opening and different closings, this usually is inconsequential.

Operators have ***operator flags*** that affect parsing of subexpressions of the operator: see Figure 14. The actual flags are the union of those given in the definition. Thus '**infix left**' is the same as '**infix**'.

The definition must have a *precedence*.

The *reformatter-name* in an *operator-definition* names a function that is called after all other parsing has been done to reformat a subexpression whose first operator is the defined operator. For example, given the subexpression '**(x, y, z)**', the bracket parser recoginizes the '**( )**' brackets and calls the operator parser on the subexpression '**x, y, z**'. The operator parser recoginizes the '**,**' operator and calls its reformatter which changes this subexpression to the BRACKETABLE expression '**{|x y z| .separator = ","}**'. The bracket parser, seeing that this subexpression is BRACKETABLE, adds the '**( )**' brackets to it, making:

```
{|x y z| .separator = ",", .initiator = "(", .terminator = ")"}
```

However, most reformatters do not actually reformat: they just check for errors. Thus for '**x / y**' the binary reformatter for '**/**' does nothing, but for '**x / y / z**' it announces an error because '**/**' appears in a subexpression with more than one operator.

The optional *reformatter-arguments* are arguments to the reformatter function. For example, the '**summation**' reformatter takes two arguments, '**(** *plus-op*, *minus-op* **)**', which are usually '**("+","-")**'.

A *parser-operator-print-command* prints all operator table entries whose *simple-operator-name*,

opening-bracket-name, or *indentation-mark-name* contains the *partial-operator-name* as a (not necessarily initial) subsequence of lexemes. Using `""` as a *partial-operator-name* will print all operator symbol table entries.

The possible operator reformatters are specified in 3.9.5 ^{p123}.

3.9.3 Standard Operators

The standard operators are given in Figures 15, 16, and 17.

precedence	selectors	reformatter	flags	operator	meaning	class
0000	code	**control**	**prefix line**	**if**	conditional	C
				else if		
			initial line	**else**	terminating conditional	
		(none)	**afix right line**	**:**	conditional modifier	
			afix line	**:** indentation mark		
			postfix line		assignment or loop	
1000		**assignment**	**left line**	**=**	assignment	S
		binary	**infix line**	**+=**	increment	A
				-=	decrement	
				***=**	multiply by	
				/=	divide by	
				|=	include	B
				&=	mask	
				^=	flip	
				<<=	shift left	
				>>=	shift right	
	math	**binary**	**infix**	**=**	assignment	S

Class	Standard Component
A:	**arithmetic operators**
B:	**bitwise operators**
C:	**control operators**
S:	**assignment operators**

Figure 15: Standard Operators: Part 1

precedence	selectors	reformatter	flags	operator	meaning	class
2000	code math	**separator**	**nofix**	,	separator	S
3000	code	**unary**	**prefix line**	**do**	iterator	I
				while		
				until		
		iteration		**repeat**		
				at most		
		(none)	**afix line**	**times**	iterator modifier	

Class	Standard Component
S:	**assignment operators**
I:	**iteration operators**

Figure 16: Standard Operators: Part 2

prece-dence	selec-tors	reformatter	flags	operator	meaning	class
10000	code math	**selector**	**infix**	**if**	selector	E
		(none)	**afix infix**	**else**		
11000		**binary**	**infix**	**BUT NOT**	logical and not	L
11100		**infix**	**infix**	**AND**	logical and	
				OR	logical or	
11200		**unary**	**prefix**	**NOT**	logical not	
12000		(none)	**infix**	**==**	equal	P
				!=	not equal	
				<	less than	
				<=	less than or equal	
				>	greater than	
				>=	greater than or equal	
13000		**infix**	**infix**	**+**	addition	A
				−	subtraction	
	code	**infix**		**|**	binary or	B
				&	binary and	
				^	binary exclusive or	
13100	code math	**binary**	**infix**	**/**	division	A
13200		**infix**	**infix**	*****	multiplication	
13300		**binary**	**infix**	******	exponentiation	
	code			**<<**	left shift	
				>>	right shift	
H-1	code math	**unary**	**prefix**	**+**	no-op	
				−	negation	
	code			**~**	binary complement	B

Class	Standard Component
A:	**arithmetic operators**
B:	**bitwise operators**
E:	**selection operators**
L:	**logical operators**
P:	**comparison operators**

Note: All operators in an expression with **infix** reformatter must be identical, except that **+** and **−** are allowed in the same expression. E.g., **AND** and **OR** are not both allowed in the same expression; and only one of **|**, **&**, and **^** is allowed in an expression.

Figure 17: Standard Operators: Part 3

The following parser standard components (see 3.5.1 p43) can be used to define these standard operators.

Standard 'control operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "if" [code] prefix line`
`  with precedence 0`
`  with control reformatter ( ":", has condition )`
	`+ code`

`define operator "else if" [code] prefix line`
`  with precedence 0`
`  with control reformatter ( ":", has condition )`
	`+ code`

`define operator "while" [code] prefix line`
`  with precedence 0`
`  with control reformatter ( ":", has condition )`
	`+ code`

`define operator "until" [code] prefix line`
`  with precedence 0`
`  with control reformatter ( ":", has condition )`
	`+ code`

`define operator "else" [code] prefix line`
`  with precedence 0`
`  with control reformatter ( ":" )`
	`+ code`

`define operator ":" [code] afix line` `  with precedence 0`	`+ code`

`define operator indentation mark ":" [code]`
`  afix line`
`  with precedence 0`
	`+ code`

`define operator indentation mark ":" [code]`
`  postfix line`
`  with precedence 0`
	`+ code`

Standard 'assignment operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "=" [code] left line`
`  with precedence 1000`
`  with assignment reformatter ( ":" )`
	`+ code`

`define operator "=" [math] infix`
`  with precedence 1000`
`  with binary reformatter`
	`+ math`

`define operator indentation mark ":" [code]`
`  postfix line`
`  with precedence 1000`
	`+ code`

`define operator indentation mark ":" [code]`
`  afix line`
`  with precedence 1000`
	`+ code`

`define operator "," [code, math] nofix`
`  with precedence 2000`
`  with separator reformatter`
	`+ code` or `+ math`

Standard 'iteration operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "do" [code] prefix line`
`  with precedence 3000`
`  with unary reformatter`
	`+ code`

`define operator "while" [code] prefix line`
`  with precedence 3000`
`  with unary reformatter`
	`+ code`

`define operator "until" [code] prefix line`
`  with precedence 3000`
`  with unary reformatter`
	`+ code`

`define operator "repeat" [code] prefix line`
`  with precedence 3000`
`  with iteration reformatter ( "times" )`
	`+ code`

`define operator "at most" [code] prefix line`
`  with precedence 3000`
`  with iteration reformatter ( "times" )`
	`+ code`

`define operator "times" [code] afix line` `  with precedence 3000`	`+ code`

Standard 'selection operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "if" [code, math] infix`
`  with precedence 10000`
`  with selector reformatter`
`  ( "if", "else" )`
	`+ code` or `+ math`

`define operator "else" [code, math] afix infix` `  with precedence 10000`	`+ code` or `+ math`

Standard 'logical operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "BUT NOT" [code, math] infix`
`  with precedence 11000`
`  with binary reformatter`
	`+ code` or `+ math`

`define operator "AND" [code, math] infix`
`  with precedence 11100`
`  with infix reformatter ( "AND" )`
	`+ code` or `+ math`

`define operator "OR" [code, math] infix`
`  with precedence 11100`
`  with infix reformatter ( "OR" )`
	`+ code` or `+ math`

`define operator "NOT" [code, math] prefix`
`  with precedence 11200`
`  with unary reformatter`
	`+ code` or `+ math`

Standard 'comparison operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "==" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

`define operator "<=" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

`define operator ">=" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

`define operator "!=" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

`define operator "<" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

`define operator ">" [code, math] infix`
`  with precedence 12000`
`  with infix reformatter`
	`+ code` or `+ math`

Standard 'arithmetic operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "+=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "-=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "*=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "/=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "+" [code, math] infix`
`with precedence 13000`
`with infix reformatter ( "+", "-" )`
	`+ code` or `+ math`

`define operator "-" [code, math] infix`
`with precedence 13000`
`with infix reformatter ( "+", "-" )`
	`+ code` or `+ math`

`define operator "/" [code, math] infix`
`with precedence 13100`
`with binary reformatter`
	`+ code` or `+ math`

`define operator "*" [code, math] infix`
`with precedence 13200`
`with infix reformatter ( "*" )`
	`+ code` or `+ math`

`define operator "**" [code, math] infix`
`with precedence 13300`
`with binary reformatter`
	`+ code` or `+ math`

`define operator "+" [code, math] prefix`
`with precedence 999999`
`with unary reformatter`
	`+ code` or `+ math`

`define operator "-" [code, math] prefix`
`with precedence 999999`
`with unary reformatter`
	`+ code` or `+ math`

Standard 'bitwise operators' Component
Operator Definition Parser Commands

Parser Command	Given Qualifiers

`define operator "|=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "&=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "^=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "<<=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator ">>=" [code] infix line`
`with precedence 1000`
`with binary reformatter`
	`+ code`

`define operator "|" [code] infix`
`with precedence 13000`
`with infix reformatter ( "|" )`
	`+ code`

`define operator "&" [code] infix`
`with precedence 13000`
`with infix reformatter ( "&" )`
	`+ code`

`define operator "^" [code] infix`
`with precedence 13000`
`with infix reformatter ( "^" )`
	`+ code`

`define operator "<<" [code] infix`
`with precedence 13000`
`with binary reformatter`
	`+ code`

`define operator ">>" [code] infix`
`with precedence 13000`
`with binary reformatter`
	`+ code`

`define operator "~" [code] prefix`
`with precedence 999999`
`with unary reformatter`
	`+ code`

3.9.4 The Operator Parsing Algorithm

The operator parsing pass runs the ***operator parsing algorithm*** that accepts as input a sequence of tokens which we call an 'expression' and edits this input, changing it into a single **BRACKETABLE** token if it contains any operators, and calling subsequent passes on the expression if it does not contain any operators.

3.9.4.1 Algorithm Outline. The operator parsing algorithm runs a left to right scan which:

1. Identifies sequences of tokens that are operators, and replaces each by a single **OPERATOR** token.
2. Identifies maximal non-empty sequences of tokens between **OPERATOR** tokens, calls subsequence passes on each, and if these do not turn the sequence into a single token that is **BRACKETTED**, **BRACKETABLE**, or **PURELIST**, packages the sequence into a single **PURELIST** token. The result is a single 'non-operator' token.
3. Identifies subexpressions using operator precedence and flags.
4. Processes each subexpression, innermost first, treating the entire expression as a subexpression. For each subexpression, the reformatter of the first operator in the expression is called. If there is no reformatter, or the reformatter chooses not to collect the expression into a single **BRACKETABLE** token, then the expression is made into a single **PURELIST** token unless the expression is already a single token of **BRACKETED**, **BRACKETABLE**, or **PURELIST** type.

The left-to-right scan always has a current position. All operators to the left of the current position have been identified as per 3.9.4.2 [p120]. All tokens between two identified operators have been collected into a single non-operator token. Subexpression recognition is done after operators and non-operators have been thus determined, and can be done most simply by a second left-to-right scan.

3.9.4.2 Operator Identification. Operator identification determines whether a sequence of lexemes beginning at the current position is an operator by looking up operator definitions. Only operator definitions selected by the current parsing selectors are considered. Longer operators are preferred over shorter operators, and among these, operators defined later are preferred over operators defined earlier.

When an operator definition is selected, it is accepted or rejected according to its flags and the token immediately to its left, which may be an operator or non-operator token. The flags of the operator definition and of any operator to its left determine whether the operator is accepted or rejected, according to the rules in Figure 18. If an operator is accepted, its is made into a single token and the search for an operator at the current position terminates.

If an operator definition has the **afix** flag, the definition is rejected unless all of the following would be true after insertion of the afix definition's operator:

(1) There is an operator preceding the inserted operator that has the same precedence as the inserted operator.

(2) There is no operator of lower precedence between this preceding operator and the inserted operator.

If an operator definition D of precedence Q is not rejected by the rule above, the first applicable entry in the following table is used to accept or reject the operator. The token immediately to the left of the current position is T, and if it is an operator, it has precedence P.

T Type	T Flags	D flags	Condition	Result
non-operator		initial		reject
non-operator		other		accept
operator	right	left		reject
operator	right	initial	P > Q	reject
operator	right	initial	P < Q	accept
operator	right + initial	initial	P = Q	accept
operator	right + other	initial	P = Q	reject
operator	right	other	P < Q	accept
operator	right	other	P $\geq$ Q	reject
operator	final	initial		reject
operator	final	left	P > Q	accept
operator	final	left	P < Q	reject
operator	final	left + final	P = Q	accept
operator	final	left + other	P = Q	reject
operator	final	other	P > Q	accept
operator	final	other	P $\leq$ Q	reject
operator	other	initial	P < Q	accept
operator	other	initial	P $\geq$ Q	reject
operator	other	left	P > Q	accept
operator	other	left	P $\leq$ Q	reject
operator	other	other		accept

Figure 18: **Operator Flag Rules**

The ends of the whole expression are treated as if they were special operators with no flags and precedence L-2.

If an operator is identified, its lexemes are collected into an **OPERATOR** token. If no operator is identified, the current position is moved one token to the right.

When an operator is identified, the non-**OPERATOR** tokens between it and the preceding operator are treated as an operator-free subexpression. This is a '***non-operator subexpression***' if it is not empty. For each non-operator subexpression (including the whole expression if it has no operators), if it is a single token of **BRACKETED**, **BRACKETABLE**, or **PURLIST** type, it is left alone. Otherwise the next pass is called, and then the subexpression is made into a single **PURELIST** token. Thus each non-operator subexpression is replaced by a single non-operator token. Note that by construction, two non-operator tokens cannot be next to each other.

If no operator is identified at the current position, and the previous position contains an operator with the **final** flag, it is a parsing error which is 'fixed up' by inserting an **ERROR'OPERATOR** of precedence L-1 and no flags before the current position.

If the whole expression ends with an operator that has the **right** flag, it is a parsing error which is 'fixed up' by inserting a non-operator **ERROR'OPERAND** before the current position.

3.9.4.3 Subexpression Processing. After operators and their fixity have been determined, the expression is parsed into subexpressions according to the syntax given in 3.9.1 [p106]. At this point, all operators in a subexpression have the same precedence, and all non-operators are single tokens of **BRACKETED**, **BRACKETABLE**, or **PURLIST** type.

If a subexpression contains one or more operators, the operator parsing pass first calls the reformatter associated with the first operator in the subexpression, if any. Then if the reformatter requests, or if there is no reformatter, and if the subexpression does not consist of a single **BRACKETED**, **BRACKETABLE**, or **PURLIST** token, the subexpression is compacted into a single **PURELIST** token.

The bracketed subexpression recognition pass may merge brackets into **BRACKETABLE** tokens: see p61. Thus given '**(x, y)**', the '**separator**' reformatter applied to '**x, y**' will return

```
@1 := x y::
          .separator = ","
```

so without this merging the bracketed subexpression recognition pass would produce

```
@2 := @1::
          .initiator = "("
          .terminator = ")"
@1 := x y::
          .separator = ","
```

but instead it produces

```
@1 := x y::
          .initiator = "("
          .terminator = ")"
```

```
.separator = ","
```

3.9.5 Operator Reformatters

In describing the effects of operator reformatters we use notation such as

```
- x + y * z    ⟹  (- x) + (y * z)
x AND y OR z   ⟹  error
```

Here ⟹ means 'is reformatted as'. Furthermore, the parentheses introduced by the rewrite are implied, which means that the lists they bracket have no **.initiator** or **.terminator**, and the parentheses are merely written here to indicate sublists in the expression structure.

Reformatters can detect parsing errors, such the fact that the two operators in '**x AND y OR z**' are not identical (when both are infix operators with the same precedence).

Note that many errors that one might expect to be handled by reformatters are not so-handled because the operators are not recognized. For example, no operators are recognized in:

x NOT y	**NOT** is not in prefix position
/ y	**/** is not in infix position

With the above in mind, the operator reformatters are:

control (*delimiter* { **, has condition** }$^{?}$ **)** **if else if else**

Typically used with **prefix line** operators (e.g., **if**).

The expression must have one of the forms:

op condition$^{?}$ *delimiter statement*

or

op condition$^{?}$ *delimiter*
statement*

The *delimiter* is typically **:**. The *condition* expression must be present if and only if **has condition** is a reformatter argument. This is the case for **if** and **else if** but not for **else**.

assignment **=**

Typically used with line assignment operator (e.g., **=**).

The expression must be postfix or binary. Specifically, it must consist of an operand followed by the operator followed by an optional operand.

separator **,**

Typically used with **nofix** operators (e.g., **,** or **;**).

All operator values must be identical, must be MIN names, and will become the **.separator** attribute of the expression.

It is required that any two operands be separated by an operator, but operators my be consecutive and may begin or end the subexpression.

Empty list operands are inserted to make all operators infix (if some are originally nofix). Then the operators are deleted, the resulting list of operands is made into a **BRACKETABLE** token with a MIN object value that has a **.separator** attribute whose value is the first operator. Examples with '**,**' as operator are:

```
x , y    ⟹  x y      with .separator ","
, y      ⟹  {} y     with .separator ","
x ,      ⟹  x {}     with .separator ","
x , , y  ⟹  x {} y   with .separator ","
```

iteration (*second-operator* **)** **repeat at most times**

Typically used with a **prefix** operator (e.g., **repeat**) and a second **afix** operator (e.g. **times**).

The expression must have the form '*operator operand operator*' with the second *operator* being equal to the reformatter argument (e.g., **times**).

declare (*second-operator* **)** **<-- :**

Typically used with a **nofix** operator (e.g., **<--**) with **final afix** operator following (e.g. **:**).

If the expression does not begin with a non-operator, an empty list **{}** is inserted at the beginning of the expression. If what is now the third token of the expression does not exist or is an operator, an empty list **{}** is inserted as the third token. If a fourth token now exists, it is checked to be sure it equals the *second-operator* reformatter argument followed by an operand followed by the end of the expression, or it is an indented paragraph whose *indentation-mark* equals the *second-operand* reformatter argument.

For example, if **<--** is a nofix operator with this reformatter having **:** as its *second-operator*, and if the mark **:** is an afix operator with the same precedence, and if the *indentation-mark* **:** is similarly an afix operator with the same precedence:

```
x <-- y : a b c  ⟹  x <-- y : (a b c)
x <-- : a b c    ⟹  x <-- {} : (a b c)
<-- y : a b c    ⟹  {} <-- y : (a b c)
x <--            ⟹  x <-- {}

x <-- u v w:        x <-- ( u v w ):
      a b c     ⟹        a b c
```

Because **:** is an afix operator with same precedence as **<--**, '**x <-- y : a b c**' does not end up as '**x <-- x (y : a b c)**'.

selector (*first-operator*, *second-operator* **)** **if else**

Typically used with **infix** operators (e.g., **if** and **else**).

Operators and operands must alternate, with the first and last subexpression elements being operands.

There must be an even number of operators, the first two as given by the reformatter arguments, and all the operators must alternate between these two possibilities.

infix { **(** *operator* { **,** *operator* }* **)** }$^?$ **AND OR + - | & ^ ***

Should only be used with operators whose precedence contains only **infix** operators.

Checks that operands and operators alternate, and that the expression begins and ends with an operator.

Checks that all the operators in the expression are listed as arguments to the reformatter. But if the reformatter has no arguments, does not check the operators.

right associative { **(** *operator* { **,** *operator* }* **)** }$^?$

Identical to the **infix** reformatter, except after checking the expression, inserts implied brackets into the expression from right to left making all operators binary with the rightmost operator innermost (executed first).

Thus if **=**, **+=**, and ***=** are all infix operators of the same precedence with this reformatter and reformatter arguments `("=", "+=")`, then:

```
y = z       ⟹  y = z
x = y += z  ⟹  x = (y += z)
x = y *= z  ⟹  error
```

left associative { **(** *operator* { **,** *operator* }* **)** }$^?$

Identical to the **infix** reformatter, except after checking the expression, inserts implied brackets into the expression from left to right making all operators binary with the leftmost operator innermost (executed first).

Thus if **+**, **-**, and **++** are all infix operators of the same precedence with this reformatter and reformatter arguments `("+", "-")`, then:

```
y + z       ⟹  y + z
x + y - z   ⟹  (x + y) - z
x + y ++ z  ⟹  error
```

unary **NOT + - ~**

Typically used with **prefix** operators.

Checks that the subexpression has exactly two elements, an operator followed by an operand. Examples where **NOT** is a prefix operator with this reformatter:

```
NOT x       ⟹  NOT x
NOT NOT x   ⟹  NOT (NOT x)
NOT         ⟹  NOT ERROR'OPERAND
```

binary **= BUT NOT << >> / ****

Typically used with **infix** operators.

The subexpression must have exactly three elements, an operand followed by an operator followed by an operand. Examples:

```
x BUT NOT y  ⟹  x [< BUT NOT >] y
x / y        ⟹  x / y
x ^ y        ⟹  x ^ y
x / y / z    ⟹  error
```

Note that in these examples '**BUT NOT**' is a single label element of the resulting subexpression, and not two word elements of the subexpression.

A Standard Lexical Program

```
// This file was automatically generated from
// ll_lexeme_standard.lexcc.


begin standard lexical program;

"<horizontal>" = "<UNICODE-CATEGORY-Zs>"
               | "<HT>";

"<misplaced-horizontal>" =
          "<horizontal>"
        & ~ "<SP>";

"<vertical>" = "<CR>" | "<VT>" | "<FF>"

"<graphic>" = "<UNICODE-CATEGORY-L>"
            | "<UNICODE-CATEGORY-M>"
            | "<UNICODE-CATEGORY-N>"
            | "<UNICODE-CATEGORY-P>"
            | "<UNICODE-CATEGORY-S>"

"<control>" = "<UNICODE-CATEGORY-C>"
            | "<UNICODE-CATEGORY-Z>"

"<illegal-control>" = "<control>"
                    & ~ "<horizontal>"
                    & ~ "<vertical>"
                    & ~ "<LF>"

"<isolated-separator>"
        = "<UNICODE-CATEGORY-Ps>"
        | "<UNICODE-CATEGORY-Pi>"
        | "<UNICODE-CATEGORY-Pe>"
        | "<UNICODE-CATEGORY-Pf>"

"<repeating-separator>" = "|"

"<separator>"
        = "<isolated-separator>"
        | "<repeating-separator>"
```

```
"<leading>"
        = "`"
        | "<0A1>"  // Inverted !
        | "<0BF>"  // Inverted ?

"<trailing>"
        = "'" | "!" | "?" | "." | ":"
        | "," | ";"

"<letter>" = "<UNICODE-CATEGORY-L>"

"<sign>" = "+" | "-"

"<exponent-indicator>" = "e" | "E"

"<digit>" = "<UNICODE-CATEGORY-Nd>"

"<mark>" = (   "<UNICODE-CATEGORY-P>"
             | "<UNICODE-CATEGORY-S>"
             | "<UNICODE-CATEGORY-M>"
             | "<UNICODE-CATEGORY-N>" )
         & ~ "<digit>"
         & ~ "<separator>"
         & ~ "<Q>"

"<middle>" =   "<graphic>"
           & ~ "<trailing>"
           & ~ "<separator>"
           & ~ "<Q>"

"<mark-middle>" = "<mark>" & ~ "<trailing>"

"<directly-quotable>" = (     "<graphic>"
                          & ~ "<"
                          & ~ "<Q>" )
                      | "<SP>"

"<escape>" = "[A-Z]" | "[0-9]"


begin start file master table;
```

```
   output start file goto begin line
end start file master table;


begin begin line master table;

   "<horizontal><repeat><graphic>"
                keep 1
                output indent
                goto horizontal space
   "<horizontal><repeat>"
                output horizontal space

   "<graphic>"
        keep 0
        output indent
        goto within line

   "<LF>" goto line break

   "<vertical><repeat><LF>" goto line break
   "<vertical><repeat>"
                output misplaced vertical

   "<illegal-control><repeat>"
        output illegal control

   "<others><repeat>"
        error unrecognized character

   output premature end of file goto end of file
        // End of file not following a line
        // break

end begin line master table;


begin horizontal space sublexeme table;

   "<horizontal>" accept;

   goto within line;
```

```
end horizontal space sublexeme table;


begin data check master table;

   "@[1-9]"     keep 0
                output data
                goto within line;
   "@<"         keep 0
                output data
                goto within line;
   "@@@@"       keep 0
                output data
                goto within line;

   "!@[1-9]"    keep 1
                output raw data
                goto within line;
   "!@<"        keep 1
                output raw data
                goto within line;

   goto within line;

end data check master table;


begin table check master table;

   "====="      keep 0
                output table
                goto row check;

   "-----"      keep 0
                output table
                goto row check;

   goto within line;

end table check master table;
```

```
begin paragraph check master table;

   "@[1-9]"     keep 0
                output data
                goto within line;
   "@<"         keep 0
                output data
                goto within line;
   "@@@@"       keep 0
                output data
                goto within line;

   "!@[1-9]"    keep 1
                output raw data
                goto within line;
   "!@<"        keep 1
                output raw data
                goto within line;

   "====="      keep 0
                output table
                goto row check;

   "-----"      keep 0
                output table
                goto row check;

   goto within line;

end paragraph check master table;


begin row check master table;

   "|"          keep 0
                output row
                goto within line;

   "="          keep 0
                output row
                goto within line;
```

```
    "-"          keep 0
                 output row
                 goto within line;

    goto within line;

end row check master table;


begin within line master table;

    "//" goto comment;
    "/"  goto mark;

     "<horizontal><repeat>"
         output horizontal space;

     "<LF>" goto line break

     "<vertical><repeat><LF>" goto line break
     "<vertical><repeat>"
                 output misplaced vertical

     "<illegal-control><repeat>"
         output illegal control

     "<isolated-separator>" output separator;
     "|<repeat>" output separator;
     "`<repeat>" output separator;
     "<0A1><repeat>" output separator;
         // Inverted !
     "<0BF><repeat>" output separator;
         // Inverted ?

     ".[0-9]" goto fraction part;
     "." keep 0 goto test trailing;
     "<trailing>" keep 0 goto test trailing;

     "[Nn][Aa][Nn]" goto numeric word;
     "[Nn]" output word goto finish middle;
     "[Ii][Nn][Ff]" goto numeric word;
```

```
    "[Ii]" output word goto finish middle;
    "<letter>" output word goto finish middle;

    "0" goto leading zero;

    "[1-9]" goto natural;

    "<digit>" output numeric goto finish middle;

    "<sign>[0-9]" goto integer part;
    "<sign>.[0-9]" goto fraction part;
    "<sign>[Nn][Aa][Nn]" goto numeric word;
    "<sign>[Ii][Nn][Ff]" goto numeric word;
    "<sign>" goto mark;
    "<mark-middle>" goto mark;

    "<Q>" translate to "" goto quoted string;

    "<others><repeat>"
        error unrecognized character

    output premature end of file
    goto end of file
        // End of file not following a line
        // break

end within line master table;


begin test trailing sublexeme table;
    // First atom of the lexeme has not been
    // scanned but is known to begin with a
    // trailing character.

    "<trailing><repeat><letter>"
                output word
                goto goto finish middle;
    "<trailing><repeat><digit>"
                output numeric
                goto finish middle;
    "<trailing><repeat><mark-middle>"
                goto mark
```

```
    "<trailing><repeat>"
                keep 0
                goto separator;

end test trailing sublexeme table;


// The below tables are entered from the master
// table with the first atom scanned, unless
// otherwise indicated.

begin comment lexeme table;

    "<graphic>" accept
    "<horizontal>" accept

    "<LF>" keep 0 goto within line
    "<vertical><repeat><LF>" keep 0
                             goto within line
    "<vertical><repeat>"
        error mispaced vertical
    "<illegal-control><repeat>"
        error illegal control
    "<others><repeat>"
        error unrecognized character
    goto within line
    // which will output premature end of file

end comment lexeme table;


begin line break lexeme table;

    "<vertical>" accept;

    "<others>" keep 0 goto begin line

    goto end of file

end line break lexeme table;
```

```
begin separator lexeme table;
    // First atom is not scanned; next lexeme
    // must be a trailing separator if it has
    // the syntax of such.

    "!<repeat>" goto finish separators
    "?<repeat>" goto finish separators
    ".<repeat>" goto finish separators
    ":<repeat>" goto finish separators
    "'<repeat>" goto finish separators
    "," goto finish separators
    ";" goto finish separators

    output NONE goto within line;

end separator lexeme table;


begin finish separators master table;
    // Come here to end lexeme when next lexeme
    // must be a trailing separator if it has
    // the syntax of such.

    goto separator;

end finish separators master table;


begin finish middle sublexeme table;
    // Come here when type of middle lexeme has
    // been set by an `output' instruction.

    "<middle>" accept;

    "<trailing><repeat><middle>" accept;

   goto within line;

end finish middle sublexeme table;


begin numeric word lexeme table;
```

```
    // One of sign? Nan or sign? Inf scanned.

    "<middle>" output word goto finish middle;
    "<trailing><repeat><middle>"
               output word
               goto finish middle;

    goto within line;

end numeric word lexeme table;


begin mark lexeme table;
    // Middle lexeme with no letter or digit yet
    // scanned.  Will be mark if none found.

    "<letter>" output word goto finish middle;

    "<digit>" output numeric goto finish middle;

    "<mark-middle>" accept;

    "<trailing><repeat><letter>"
                output word
                goto finish middle;
    "<trailing><repeat><digit>"
                output numeric
                goto finish middle;
    "<trailing><repeat><mark-middle>" accept;

    goto within line;

end mark lexeme table;


begin leading zero sublexeme table;

    "<middle>"
        keep 0 goto integer part
        // Includes second digit.
        // Includes exponent.
    "<trailing><repeat><middle>"
```

```
        keep 0 goto integer part
        // Includes fraction.

    output natural goto within line;

end leading zero sublexeme table;


begin natural lexeme table;

    "[0-9]" accept;

    ".[0-9]" goto fraction part;
    ".<middle>"
        output numeric goto finish middle;
    ".<trailing><middle>"
        output numeric goto finish middle;

    "<exponent-indicator>[0-9]"
              goto exponent part;
    "<exponent-indicator><sign>[0-9]"
              goto exponent part;
    "<exponent-indicator>"
              output numeric
              goto finish middle;

    "<middle>" output numeric
              goto finish middle;
    "<trailing><repeat><middle>"
              output numeric
              goto finish middle;

    goto within line;

end natural lexeme table;


begin integer part sublexeme table;
    // sign digit or 0 plus middle lexeme
    // continuation scanned

    "[0-9]" accept;
```

```
    ".[0-9]" goto fraction part;
    ".<middle>"
        output numeric goto finish middle;
    ".<trailing><middle>"
        output numeric goto finish middle;

    "<exponent-indicator>[0-9]"
                goto exponent part;
    "<exponent-indicator><sign>[0-9]"
                goto exponent part;
    "<exponent-indicator>"
                output numeric
                goto finish middle;

    "<middle>" output numeric
                goto finish middle;
    "<trailing><repeat><middle>"
                output numeric
                goto finish middle;

    output number
    goto within line;

end integer part sublexeme table;


begin fraction part sublexeme table;
    // `. digit' scanned with prefix compatible
    // with a number.

    "[0-9]" accept;

    "<exponent-indicator>[0-9]"
                goto exponent part;
    "<exponent-indicator><sign>[0-9]"
                goto exponent part;
    "<exponent-indicator>"
                output numeric
                goto finish middle;

    "<middle>" output numeric
```

```
                goto finish middle;
    "<trailing><repeat><middle>"
                output numeric
                goto finish middle;

   output number;
   goto within line;

end fraction part sublexeme table;


begin exponent part sublexeme table;
    // `{e|E} sign? digit' scanned after prefix
    // compatible with a number.

    "[0-9]" accept;

    "<middle>" output numeric
                goto finish middle;
    "<trailing><repeat><middle>"
                output numeric
                goto finish middle;

   output number;
   goto within line;

end exponent part sublexeme table;


begin quoted string lexeme table;

    "<Q>" translate to "" goto within line;
            // End quoted string.

    "<" "[0-9]" ">"
        translate hex 1 1
    "<" "[0-9]<escape><repeat>" ">"
        translate hex 1 1
        else error unrecognized escape
             translate to "<UUC>"
    "<" "[A-Z]" ">"
             translate name 1 1
```

```
            else error unrecognized escape
                 translate to "<UUC>"
        "<" "[A-Z]<escape><repeat>" ">"
              translate name 1 1
            else error unrecognized escape
                 translate to "<UUC>"
        "<"
            accept;


        "<directly-quotable>" accept;

        "<LF>"
            keep 0 goto premature end of string
        "<vertical><repeat><LF>"
            keep 0 goto premature end of string
        "<vertical><repeat>"
            error mispaced vertical
        "<misplaced-horizontal><repeat>"
            error misplaced horizontal
        "<illegal-control><repeat>"
            error illegal control
        "<others><repeat>"
            error unrecognized character

        goto premature end of string;

    end quoted string lexeme table;


    begin premature end of string master table;

        output premature end of string
        goto within line;

    end premature end of string master table;


    begin end of file master table;
        output end of file goto stop
    end end of file master table;
```

```
begin stop master table;
end stop master table;


end standard lexical program;
```

B C++ Lexical Program

```
// This file was automatically generated from
// ll_lexeme_c++.lexcc.

begin c++ lexical program;

// Before this lexical program is used the input
// should be preprocessed to
//
//    (1) Replace trigraph sequences.
//    (2) Eliminate carriage-returns next to a
//        newline (others become misplaced)
//    (3) Eliminate sequences of the form:
//        backslash newline
//
// After this lexical program is used the output
// should be post-processed to
//
//    (4) Perform C/C++ macro preprocessing
//        (macro expansion).
//    (5) Delete whitespace and newlines.
//    (6) Concatenate adjacent quoted strings
//        of the same character type (ordinary,
//        u, U, or L).

"<digit>" = "0-9";

"<non-zero-digit>" = "1-9";

"<oct-digit>" = "0-7";

"<hex-digit>" = "0-9" | "a-f" | "A-F";

"<ascii-letter>" = "a-z" | "A-Z";

"<letter>" = "<UNICODE-CATEGORY-L>"

"<non-ascii-letter>" =   "<letter>"
                       & ~ <ascii-letter>

"<ascii-character>" =   "<00-3F>"
```

```
// <non-spacing-combining-char> ::=
//         "<UNICODE-CATEGORY-Mn>"

// Universal characters are only permitted in
// identifiers, character literals, and string
// literals.  In identifiers they must represent
// non-ASCII-letters or non-ASCII combining
// characters, and the latter cannot be the
// initial character of an identifier.

"<identifier-non-digit>" = "_" | "<letter>";

"<identifier-char>" =
      "<identifier-non-digit>"
    | "<digit>"
    | "<non-spacing-combining_char>"

"<non-ascii-identifier-char>" =
      "<identifier-char>" & ~ <ascii-char>

// Because `u', `U', or `L' can start a char-
// acter or string literal, we must define:
//
"<u_U_L>" = "u" | "U" | "L";
"<identifier-non-literal>" =
          "<identifier-non-digit>"
        & ~ "<u_U_L>";

"<whitespace-char>" =
    " " | "<HT>" | "<VT>" | "<FF>";
    // newline is treated separately


// The following can begin an operator or punc-
// tuation mark.
//
// / not followed by / or * and . not followed
// by <digit> are handled separately.
//
"<op-char>" = "#" | "<" | ">" | ":" | "%" |
            | "?" | "+" | "-" | "*" | "="
```

```
              | "^" | "&" | "|" | "~" | "!" | ","
              | "(" | ")" | "[" | "]" | "{" | "}"
              | ";";

"<sign>" = "+" | "-" ;

"<u_U>" = "u" | "U" ;


// C/C++ preprocessing is hereafter abbreviated
// as `pp'.

// Alternative operators that are not identi-
// fiers are translated.  E.g., <: becomes [ in
// the lexeme translation.

// C++ punctuation are treated as `operators'.

// Pp numbers are divided into 5 categories
// (note that a `suffix' is an identifier):
//
//      decimal integer
//          nothing but digits not beginning
//          with 0, with an optional suffix
//      octal integer
//          nothing but octal digits beginning
//          with 0, with an optional suffix
//      hexadecimal integer
//          nothing but hexadecimal digits
//          refaced by 0x or 0X, with an
//          optional suffix
//      float
//          legal floating point #'s, with an
//          optional suffix
//      pp number
//          all other pp numbers

// Because of pp control lines, lexical scanning
// is slightly context dependent.  There are
// several contexts, each corresponding to a
// different master table:
//
```

```
//    initial master
//        Used in line beginning situations to
//        recognize the # token that introduces
//        a pp control line.
//    pp beginning master
//        Used to scan the beginning of a pp
//        control line after the # has been
//        scanned but before anything else is
//        scanned.
//    pp include master
//        Used to scan the pp-header token in a
//        #include line after #include has been
//        scanned.
//    normal master
//        Used to scan a non pp control line or
//        the rest of a pp control line after
//        any #include header.
//
// The whitespace lexeme types are:
//
//    newline
//        a single <NL>
//    horizontal-space
//        sequence of "<whitespace-char>"s
//        (excludes <NL>'s)
//    comment
//        "/*" comment (including /* and */)
//        "//" comment (including // but not
//                      ending newline)
//
// This allows line feeds to be used to end pp
// control lines.


// Characters that are not part of legal lexemes
// and which are outside character literal ''s
// and and quoted string literal ""s are made
// into error lexemes that can separate other
// lexemes in exactly the same way that white-
// space can.
//
// Note that ## cannot be used to move charac-
```

```
// ters inside a character or string literal ''
// or "".  Also, we do NOT allow ## to append a
// combining character to another character.
// Therefore ## cannot be used to create a legal
// pp token by concatenating a legal pp token
// with a stray character that is not part of a
// legal pp token.  Thus we can treat all stay
// characters as errors, instead of making them
// into pp tokens.

// Characters that are part of erroneous atoms
// INSIDE character literal ''s or quoted string
// ""s are announced as part of an erroneous
// atom which is translated to "" and ignored as
// if it did not appear at all in the input.

begin initial master table;

    "<whitespace-char>" call whitespace;

    // We only need handle /*...*/ comments
    // because only these can occur INSIDE
    // a pp directive line.
    //
    "/*" call "/*" comment;

    "#" output operator goto pp beginning;
    "##" output operator goto normal;

    "%:" translate to "#" output operator
         goto pp beginning;
    "%:%:" translate to "##" output operator
           goto normal;

    goto normal;

end initial master table;


begin pp beginning master table;

    "<whitespace-char>" call whitespace;
```

```
        // We only need handle /*...*/ comments
        // because only these can occur inside
        // pp directive line.
        //
        "/*" call "/*" comment;

        // "include" identifier must be followed by
        // one of the following for a header to be
        // recognized:
        //
        //     <whitespace-char>
        //     /* comment
        //     <...> header
        //     "..." header
        //
        "include<whitespace-char>"
            keep 7 output identifier
            goto pp include;
        "include/"
            keep 7 output identifier
            goto pp include;
        "include<"
            keep 7 output identifier
            goto pp include;
        "include<Q>"
            keep 7 output identifier
            goto pp include;

        goto normal;

    end pp beginning master table;

    begin pp include master table;

        "<whitespace-char>" call whitespace;

        // We only need handle /*...*/ comments
        // because only these can occur inside
        // pp directive line.
        //
        "/*" call "/*" comment;
```

```
    "<" translate to ""
        goto bracketed header name;

    "<Q>" translate to ""
        goto quoted header name;

    goto normal;

end pp include master table;


begin premature newline master table;

    output premature newline goto normal;

end premature newline master table;


begin premature end of file master table;

    output premature end of file goto initial;

end premature end of file master table;

begin whitespace lexeme table;

    "<whitespace-char>" accept;

    return;

end whitespace lexeme table;


begin "/*" comment lexeme table;

    "*/" return;
    "*" accept;

    "<other>" accept;

    translate_to "*/"
```

```
    goto premature end of file;

end "/*" comment lexeme table;


begin "//" comment lexeme table;

    "<NL>" keep 0 return;

    "<other>" accept;

    goto premature end of file;

end "//" comment lexeme table;


begin bracketed header name lexeme table;

    ">" translate to "" goto normal;

    "<NL>"
        keep 0 goto premature newline;

    "<other>" accept;

    goto premature end of file;

end bracketed header name lexeme table;


// Quoted header names are not the same as
// quoted strings, and cannot have escape
// sequences.
//
begin quoted header name lexeme table;

    "<Q>" translate to "" goto normal;

    "<NL>"
        keep 0 goto premature newline;

    "<other>" accept;
```

```
    goto premature end of file;

end quoted header name lexeme table;


begin normal master table;

    "<whitespace-char>" call whitespace;

    "/*" call "/*" comment;
    "//" call "//" comment;
    "/" match operator output operator
        // match should always succeed

    "<op-char>"
        match operator output operator
        // match should always succeed

    "<identifier-non-literal>"
        call identifier;
        // Also see "u/U/L..." etc. below.

    "u'" translate to ""
         output u char literal
         call char literal;
    "u<Q>" translate to ""
           output u string literal
           call string literal;
    "u"  call identifier;

    "U'" translate to ""
         output U char literal
         call char literal;
    "U<Q>" translate to ""
           output U string literal
           call string literal;
    "U"  call identifier;

    "L'" translate to ""
         output L char literal
         call char literal;
```

```
"L<Q>" translate to ""
       output L string literal
       call string literal;
"L"  call identifier;

"\<u_U>" match universal char
         require <non-ascii-letter>
         call identifier
    else match universal char
         require "<ascii-char>"
         output ascii universal char
    else match universal char
         output misplaced universal char
    else match short universal char
         output universal char
         // Match should always succeed.
"\" output misplaced char

"<non-zero-digit>" call decimal integer;

"0x" call hexadecimal integer;
"0X" call hexadecimal integer;
"0" call octal integer;

".<digit>" call fraction;
"." match operator output operator
    // match should always succeed

"'" translate to ""
    output char literal
    call char literal;

"<Q>" translate to ""
      output string literal
      call string literal;

"<NL>" output newline goto initial;

"<other>" output misplaced char;

output end of file;
```

```
end normal master table;


begin operator atom table;

    "("; ")"; "["; "]"; "{"; "}"; ";"; ","; "?";
    "~";

    "#";
    "##";

    "<:" translate to "[";
    "<%"; translate to "{";
    "<<="; "<<"; "<";

    ">>="; ">>"; ">";

    ":>" translate to "]";
    "::"; ":";

    "%>" translate to "}";
    "%:%:" translate to "##";
    "%:" translate to "#";
    "%="; "%";

    "..."; ".*"; ".";

    "++"; "+="; "+";

    "->*"; "->"; "--"; "-="; "-";

    "*="; "*";
    "/="; "/";
    "^="; "^";
    "!="; "!";
    "=="; "=";

    "&&"; "&="; "&";

    "||"; "|="; "|";

    fail;
```

```
end operator atom table;


begin identifier lexeme table;

    "<identifier-non-digit>" accept;
    "<digit>" accept;

    "\<u_U>" match universal char
             require "<non-ascii-letter>"
        else keep 0 return

    return;

end identifier lexeme table;

"<e_E> = "e" | "E";
"<identifier-non-digit-except-eE>" =
    "<identifier-non-digit>" & ~ "<e_E>";


begin decimal integer lexeme table;

    "<digit>" accept;

    "." goto fraction;

    "<e_E><sign><digit>" goto exponent;
    "<e_E><sign>" goto pp number;
    "<e_E><digit>" goto exponent;
    "<identifier-non-digit>" goto suffix;

    "\"      match universal char
             require "<non-ascii-letter>"
             goto suffix
        else keep 0 return

    return;

end decimal integer lexeme table;
```

```
begin octal integer lexeme table;

    "<oct-digit>" accept;

    "8" goto float integer;
    "9" goto float integer;

    "." goto fraction;

    "<e_E><sign><digit>" goto exponent;
    "<e_E><sign>" goto pp number;
    "<e_E><digit>" goto exponent;
    "<identifier-non-digit>" goto suffix;

    "\"      match universal char
             require "<non-ascii-letter>"
             goto suffix
         else keep 0 return

    return;

end octal integer lexeme table;

"<identifier-non-hex-digit>" =
    "<identifier-non-digit>" & ~ "<hex-digit>";


begin hexadecimal integer lexeme table;

    "<hex-digit>" accept;

    "<identifier-non-hex-digit>" goto suffix;

    "\"      match universal char
             require "<non-ascii-letter>"
             goto suffix
         else keep 0 return

    return;

end hexadecimal integer lexeme table;
```

```
// integer began with 0 and then included a
// non-octal digit.
//
begin float integer lexeme table;

    "<digit>" accept;

    "." goto fraction;

    "<e_E><sign><digit>" goto exponent;
    "<e_E><sign>" goto pp number;
    "<e_E><digit>" goto exponent;
    "<identifier-non-digit>" goto pp number;

    "\"      match universal char
             require "<non-ascii-letter>"
             goto pp_number
        else keep 0 goto pp number

    goto pp number;

end float integer lexeme table;


// Come here to process suffix at end of integer
// or float lexeme.  Similar to identifier but
// changes lexeme type to pp number when certain
// character sequences are encountered.
//
begin suffix sublexeme table;

    "<identifier-non-digit-except-eE>" accept;
    "<digit>" accept;

    "<e_E><sign>" goto pp number;
    "<e_E>" accept;

    "." goto pp number;

    "\<u_U>" match universal char
```

```
                require
                    "<non-ascii-identifier-char>"
                else keep 0 return;

        return;

    end suffix sublexeme table;

    begin pp number lexeme table;

        "<identifier-non-digit-except-eE>" accept;
        "<digit>" accept;

        "<e_E><sign>" accept;
        "<e_E>" accept;

        "." accept;

        "\<u_U>" match universal char
                require
                    "<non-ascii-identifier-char>"
                else keep 0 return;

        return;

    end pp number lexeme table;


    begin fraction lexeme table;

        "<digit>" accept;

        "." goto pp number;

        "<e_E><sign><digit>" goto exponent;
        "<e_E><digit>" goto exponent;
        "<e_E><sign>" goto pp number;
        "<identifier-non-digit>" goto suffix;

        "\<u_U>" match universal char
                require "<non-ascii-letter>"
                goto suffix
```

```
                else keep 0 return;

    return;

end fraction lexeme table;


begin exponent lexeme table;

    "<digit>" accept;

    "." goto pp number;

    "<e_E><sign>" goto pp number;
    "<identifier-non-digit>" goto suffix;

    "\<u_U>" match universal char
             require "<non-ascii-letter>"
             goto suffix
             else keep 0 return;

    return;

end exponent lexeme table;


begin char literal sublexeme table;

    "'" translate to "" return;

    "<NL>" keep 0 goto premature newline

    "\" match escaped char
        // match should always succeed

    "<other>" accept

    goto premature end of file

end char literal sublexeme table;
```

```
begin string literal sublexeme table;

    "<Q>" translate to "" return;

    "<NL>" keep 0 goto premature newline

    "\" match escaped char
        // match should always succeed

    "<other>" accept

    goto premature end of file

end string literal sublexeme table;


// This atom table is used when the next atom
// begins with \ and is in a character or string
// literal.  This atom table always succeeds,
// but may produce an erroneous atom with ""
// translation.
//
// We allow only a maximum of 8 hexadecimal
// digits after \x; 9 digits is an error.  The
// C++ standard permits any number of hexa-
// decimal digits.
//
begin escaped char atom table;

    "\n" translate to "<NL>";

    "\t" translate to "<HT>";

    "\v" translate to "<VT>";

    "\b" translate to "<BS>";

    "\r" translate to "<CR>";

    "\f" translate to "<FF>";

    "\a" translate to "<BEL>";
```

```
"\\" translate to "\";

"\?" translate to "?";

"\<Q>" translate to "<Q>";

"\'" translate to "'";

"\<oct-digit><oct-digit><oct-digit>"
    translate oct 1 0;
"\<oct-digit><oct-digit>"
    translate oct 1 0;
"\<oct-digit>"
    translate oct 1 0;

"\x<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit>"
    error too long hex escape
    translate to "";
"\x<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
    translate hex 2 0;
"\x<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit>"
    translate hex 2 0;
"\x<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
    translate hex 2 0;
"\x<hex-digit><hex-digit>"
  "<hex-digit><hex-digit>"
  "<hex-digit>"
    translate hex 2 0;
"\x<hex-digit><hex-digit>"
```

```
      "<hex-digit><hex-digit>"
        translate hex 2 0;
    "\x<hex-digit><hex-digit>"
      "<hex-digit>"
        translate hex 2 0;
    "\x<hex-digit><hex-digit>"
        translate hex 2 0;
    "\x<hex-digit>"
        translate hex 2 0;
    "\x" error ill formed escape
        translate to "";

    "\<u_U>" match universal char
        else match short universal char
             error short universal char
             translate to "";
             // This should always succeed

"\<NL>" keep 1
        error ill formed escape
        translate to "";

"\<other>" error ill formed escape
           translate to "";

"\" error ill formed escape
    translate to "";
    // In case \ followed by end of file

// This table should always succeed.

end escaped char atom table;


// This atom table is called when the next
// atom begins with \U or \u.  If the atom
// has the correct number of hexadecimal
// digits after the \U or \u, it is recognized
// and translated.
//
begin universal char atom table;
```

```
    "\U<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
        translate hex 2 0;

    "\u<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
        translate hex 2 0;

    fail;

end universal char atom table;


// This atom table is called when the next
// atom begins with \U or \u but there are too
// few hexadecimal digits following.  An atom
// is always recognized but is NOT translated.
//
begin short universal char atom table;

    "\U<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit>";
    "\U<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>";
    "\U<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>"
      "<hex-digit>";
    "\U<hex-digit><hex-digit>"
      "<hex-digit><hex-digit>";
    "\U<hex-digit><hex-digit>"
      "<hex-digit>";
    "\U<hex-digit><hex-digit>";
    "\U<hex-digit>";
    "\U";

    "\u<hex-digit><hex-digit>"
      "<hex-digit>";
```

```
        "\u<hex-digit><hex-digit>";
        "\u<hex-digit>";
        "\u";

        // The above should always succeed.

    end short universal char atom table;


    end c++ lexical program;
```

Index