Lexical Analysis

$Revision: 2.8 $

Compiler Tools Group
Department of Electrical and Computer Engineering
University of Colorado
Boulder, CO, USA
80309-0425

Table of Contents

The purpose of the lexical analyzer is to partition the input text, delivering a sequence of *comments* and *basic symbols*. Comments are character sequences to be ignored, while basic symbols are character sequences that correspond to terminal symbols of the grammar defining the phrase structure of the input (see Section 'Context-Free Grammars and Parsing' in Syntactic Analysis).

A user must define the forms of comments and the forms of all basic symbols corresponding to non-literal terminal symbols of the grammar. Eli can deduce the form of a literal terminal symbol from the grammar specification.

The definition consists of one or more type-'`gla`' files. Each line of a type-'`gla`' file describes a set of character sequences. If a line begins with an identifier followed by a colon (`:`), then all of the character sequences described by the line are instances of the non-literal terminal symbol named by that identifier; otherwise they are comments.

Here is an example of a type-'`gla`' file:

```
HexInteger:  $0[Xx][0-9A-Fa-f]+
             $!  (auxEOL)
Identifier:  C_IDENTIFIER
```

The first line of this specification uses a regular expression to define a hexadecimal integer as a zero, followed by the letter `X` (either upper or lower case) and one or more hexadecimal digits represented in the usual way. In the second line, one form of comment is defined by a regular expression and the name of a C routine. The C routine will be invoked when the regular expression has been matched. This approach allows the user to define character sequences operationally when a declarative definition is tedious or does not support appropriate error reporting.

Since certain lexical structures are common to many languages, Eli provides a library of definitions that can be invoked simply be giving their names. `C_IDENTIFIER`, in the third line, is such an invocation. The effect of the third line is to define the form of the basic symbol `Identifier` as that of an identifier in C: a letter or underscore followed by some sequence of letters, digits and underscores.

Chapter 1 defines the usage, form and content of specifications provided by the user as type-'`gla`' files. Those specifications may refer to canned descriptions, which are defined in Chapter 2. Chapter 3 presents the default processing of spaces, tabs and newlines and explains how to define other strategies. The treatment and meaning of literal terminal symbols is discussed in Chapter 4, and Chapter 5 explains how a generated lexical analyzer can be made insensitive to the case of letters. Complex lexical analysis problems may require modification of the behavior of the generated module; Chapter 6 discusses the possibilities.

1 Specifications

A specification consists of a *regular expression*, possibly the name of an *auxiliary scanner*, and possibly the name of a *token processor*. Sequences of input characters are classified initially on the basis of the regular expression they match. If the line containing the regular expression also contains the name of an auxiliary scanner, then that scanner is invoked after the regular expression has been matched. An auxiliary scanner may lengthen or shorten the character sequence being classified. If the line containing the regular expression also contains the name of a token processor, then that token processor is invoked after any auxiliary scanner. A token processor may change the initial classification of the sequence, and may also calculate a value representing the sequence.

Specifications are provided in type-'`gla`' files whose contents obey the following phrase structure:

```
File: Specification NewLine / File Specification NewLine .
Specification: ScanSpec / ScanSpec '[' TokenProcessorName ']' .
ScanSpec: Pattern / Pattern '(' AuxilaryScannerName ')' .
Pattern: RegularExpression / CannedSpecificationName .
CannedSpecificationName: Identifier .
AuxilaryScannerName: Identifier .
TokenProcessorName: Identifier .
```

An `Identifier` is defined as in C, and a type-'`gla`' file may contain arbitrary empty lines, C comments and pre-processor directives. Comments may also be written as character sequences enclosed in braces (`{ }`) that do not themselves include braces.

The remainder of this chapter explains each of these components of the description in detail.

1.1 Regular Expressions

A regular expression is a pattern that defines a set of character sequences: If the regular expression matches a particular sequence then that sequence is a member of the set; otherwise it is not a member. Here is a summary of Eli's regular expression notation:

`c` matches the character `c`, unless `c` is space, tab, newline or one of `\ " . [ ] ^ ( ) | ? + * { } / $ <`

`\c` matches `c` (see Section 1.1.1 [Matching Operator Characters], page 4)

`"s"` matches the sequence `s` (see Section 1.1.1 [Matching Operator Characters], page 4)

`.` matches any character except newline

`[xyz]`	matches exactly one of the characters `x`, `y` or `z`
`[^xyz]`	matches exactly one character that is *not* `x`, `y` or `z`
`[c-d]`	matches exactly one of the characters whose ASCII codes lie between the codes for `c` and `d` (inclusive)
`(e)`	matches a sequence matched by `e`
`ef`	matches a sequence matched by `e` followed by a sequence matched by `f`
`e|f`	matches either a sequence matched by `e` or a sequence matched by `f`
`e?`	matches either an empty sequence or a sequence matched by `e`
`e+`	matches one or more occurrences of a sequence matched by `e`
`e*`	matches either an empty sequence or one or more occurrences of a sequence matched by `e`
`e{`m`,`n`}`	matches a sequence of no fewer than m and no more than n occurrences of a sequence matched by `e`

Each of the regular expressions `e?`, `e+`, `e+` and `e{`m`,`n`}` matches the longest sequence of characters consonant with its definition.

In a type-'`gla`' file, each regular expression is delimited on the left by `$` and on the right by white space:

```
$a57D
$[0-9]+
$[a-zA-Z_][a-zA-Z_0-9]*
```

The first example matches the single character sequence `a57D`, while the second matches a sequence of one or more digits. The third describes C-style identifiers: an initial letter or underscore, followed by zero or more alphanumeric characters or underscores.

1.1.1 Matching operator characters

A regular expression consists of *text characters* (which match the corresponding characters in the input character sequences) and *operator characters* (which specify repetitions, choices and other features). The operator characters are the following:

```
\ " . [ ] ^ ( ) | ? + * { } / $ <
```

Space, tab, newline and characters appearing in this list are *not* text characters; every other character *is* a text character.

If an operator character is to match an instance of itself in the input sequence then it must be marked in the regular expression as being a text character. This can be done by preceding it with backslash (`\`). Any occurrence of an operator character (including backslash) that is preceded by backslash loses its operator status and is considered to be a text character. The text characters space, tab and newline are represented as `\040`, `\t` and `\n` respectively; `\b` represents the text character "backspace". Any character except the ASCII NUL (code 0) can also be represented by a backslash followed by the ASCII code for the character written as a sequence of up to three octal digits (the representation of a space character always has this form).

A sequence of operator characters can be used as a sequence of text characters by surrounding the sequence with double quote operators (`"`):

```
xyz"++"
"xyz++"
```

Both of these patterns match the string `xyz++`. As shown, it is harmless but unnecessary to quote a character that is not an operator.

Backslash is also effective within a sequence surrounded by double quote operators, and must be used to mark backslash, quote and white space:

```
"\t\\\040\"\040.\040[\040]\040^"
```

This pattern matches an initial segment of the operator character display at the beginning of this section.

1.1.2 Character classes

A *character class* is a pattern that defines a set of characters and matches exactly one character from that set. The simplest character class representation is the period (`.`), which defines the set of all characters except newline. Character classes can also be represented using the operator pair `[ ]`. `[Abc]` defines the set of three characters `A`, `b`, and `c`.

Within square brackets, most operator meanings are ignored. Only four characters are special: `\`, `-`, `^` and `]`. In particular, the double quote character (`"`) is not considered special and therefore cannot be used to surround a sequence of operator characters. The \ character provides the usual escapes within character class brackets. Thus [[\]] matches either [or], because \ causes the first] in the character class representation to be taken as a normal character rather than the closing bracket of the representation. The following specification causes an error, however:

```
[["]"]
```

The quote is not special in a class, so the first] is the closing bracket of the set. The second " is therefore outside the definition of the class, and is taken as the beginning of a quoted string containing the second]. Since there is no closing quote for this string, it is erroneous.

If the first character after the opening bracket of a character class is ^, the set defined by the remainder of the class is complemented with respect to the computer's character set. Using this notation, the character class represented by . can be described as [^\n].

If ^ appears as any character of a class except the first, it is not considered to be an operator. Thus [^abc] matches any character except a, b, or c but [a^bc] or [abc^] matches a, b, c or ^.

Within a character class representation, - can be used to define a set of characters in terms of a range. For example, a-z defines the set of lower-case letters and A-Z defines the set of upper-case letters. The endpoints of a range may be specified in either order (i.e. both 0-9 and 9-0 define the set of digits). Ranges can also be defined in terms of specific ASCII codes: \041-\0176 is the set of all visible ASCII characters. Using - between any pair of characters that are not both upper case letters, both lower case letters, or both digits defines an implementation-dependent set and will generate a warning.

Any number of ranges can be used in the representation of a character class. For example, `[a-z0-9<>_]` will match any lower case letter, digit, angle bracket or underline while [^a-zA-Z] will match any character that is not a letter. If it is desired to include the character - in a character class, it should either be escaped with \ or it should occupy the first or last position. Thus [-+0-9] will match +, - or any digit, as will [0-9\-+].

1.1.3 Building complex regular expressions

Single characters, character strings and character classes are all simple regular expressions. Each matches a particular set of character sequences. More complex patterns are built from these simple regular expressions by concatenation, alternation and repetition. The components of a complex pattern may be grouped by enclosing them in parentheses; a parenthesized expression behaves like a simple regular expression in further compositions.

Components must not be separated by white space, because white space terminates a regular expression.

When a complex regular expression is written as a sequence of components, the resulting pattern will match a sequence of characters consisting of a subsequence matching the first component, followed by a subsequence matching the second component, and so on:

```
[1-9]\.[0-9][0-9]
```

This complex expression has four components: three character classes and the text character `.` (the backslash converts the operator character `.` to a text character). It matches character sequences like `2.54` and `9.99`, but not `0.59`, `45.678` or `1x23`.

When the components of a complex regular expression are separated by the operator `|`, the resulting pattern will match a sequence of characters that matches at least one of the components:

```
[A-Za-z]|[1-9][0-9]&
```

This complex expression has two immediate components: a character class and a complex expression that is the result of concatenating two character classes and a single character. The complete expression matches character sequences like `B` and `10&`, but not `X11` or `A&`.

Concatenation takes precedence over alternation in constructing a complex regular expression, so this example is equivalent to `[A-Za-z]|([1-9][0-9]&)`. Parentheses can be used to group the expression differently:

```
([A-Za-z]|[1-9][0-9])&
```

This complex expression also has two immediate components, but they are a parenthesized expression and a single character. The complete expression matches character sequences like `B&` and `10&`.

When a complex regular expression consists of a single component followed by the operator `?`, the resulting pattern will match either an empty sequence or a sequence of characters that matches the component:

```
(-|\+?)[1-9]
```

Here the operand of `?` is the text character `+`. This complex expression matches character sequences like `-1`, `+2` and `3`. In each case, the pattern matches the longest sequence of characters consonant with its definition.

The `?` operator takes precedence over both concatenation and alternation. If its operand is a complex expression involving either of these operations, that complex expression must be parenthesized.

When a complex regular expression consists of a single component followed by the operator `+`, the resulting pattern will match a sequence of characters that matches one or more successive occurrences of a sequence matching the component:

```
[0-9]+
```

This complex expression has one immediate component: a character class. It matches character sequences like `0` and `1019`. In each case, the pattern matches the longest sequence of characters consonant with its definition.

The `+` operator takes precedence over both concatenation and alternation. If its operand is a complex expression involving either of these operations, that complex expression must be parenthesized.

When a complex regular expression consists of a single component followed by the operator `*`, the resulting pattern will match a sequence of characters that matches an empty sequence or one or more successive occurrences of a sequence matching the component:

```
[1-9][0-9]*
```

This complex expression has two immediate components: a character class and a complex expression whose operator is `*`. That complex expression, in turn, has a single character class component. The complete expression matches character sequences like `1` and `2992`, but not `0` or `0101`. In each case, the pattern matches the longest sequence of characters consonant with its definition.

The `*` operator takes precedence over both concatenation and alternation. If its operand is a complex expression involving either of these operations, that complex expression must be parenthesized. For example, `([1-9][0-9])*` would match character sequences like `1019` and `2992`, but not `1` or `123`.

When a complex regular expression consists of a single component followed by the operator `{`m`,`n`}` (m and n integers), the resulting pattern will match a sequence of characters that matches no fewer than m and no more than n successive occurrences of a sequence matching the component:

```
[A-Za-z][A-Za-z0-9]{0,5}
```

This complex expression has two immediate components: a character class and a complex expression whose operator is `{0,5}`. That complex expression, in turn, has a single character class component. The complete expression matches character sequences like `A` and `xyzzy`, but not `identifier` or `01July`. In each case, the pattern matches the longest sequence of characters consonant with its definition.

The `{`m`,`n`}` operator takes precedence over both concatenation and alternation. If its operand is a complex expression involving either of these operations, that complex expression must be parenthesized. For example, `([1-9][0-9]){1,2}` would match character sequences like `10` and `2992`, but not `1`, `123` or `123456`.

1.1.4 What happens if the specification is ambiguous?

When more than one expression can match the current character sequence, a choice is made as follows:

1. The longest match is preferred.
2. Among rules which match the same number of characters, the rule given first is preferred.

Thus, suppose we have the following descriptions:

```
Limit: $55
Speed: $[0-9]+
```

If the input text is `550kts` then the sequence `550` is classified as `Speed`, because `[0-9]+` matches three characters while `55` matches only two. If the input is `55mph` then both patterns match two characters, and the sequence `55` is classified as `Limit` because `Limit` was given first. Any shorter sequence of digits (e.g. `5kph`) would not match the regular expression `55` and so the `Speed` classification would be used.

When more than one type-'gla' file is provided, specifications in different files have no defined order. Thus if `Limit` and `Speed` appeared in different files, classification of the sequence `55` would be undefined. If an ambiguity between two descriptions is to be resolved on the basis of their order of appearance, they must be given within the same type-'gla' file.

1.2 Auxiliary Scanners

An auxiliary scanner is a routine to be invoked after the pattern described by the regular expression has been matched. The routine is passed a pointer to the matched string and the length of that string, and it returns a pointer to the first character that is not to be considered part of the string matched. Thus an auxiliary scanner may increase, reduce or leave unchanged the number of characters matched by the regular expression. This allows a user to specify operationally patterns that are tedious or impossible to describe using regular expressions (e.g. nested comments), or that require special operations during the match (e.g. sequences containing tabs or newlines — see Chapter 3 [White Space], page 25), or that would benefit from specialized error reporting.

An auxiliary scanner is invoked by giving its name, surrounded by parentheses (`( )`), on the same line as the associated regular expression:

```
$--  (auxEOL)
```

This specification invokes the auxiliary scanner `auxEOL` whenever a sequence of two dashes is recognized, and passes it a pointer to the first of the two dashes and a length of 2. As described below, `auxEOL` returns a pointer to the first character of the next line, after having updated the coordinate information. This specification is the implementation of the canned description `ADA_COMMENT`.

The remainder of this section describes the auxiliary scanners that are available in the Eli library, and also explains how to implement auxiliary scanners for tasks that are specific to your problem.

1.2.1 Available scanners

All of the auxiliary scanners described in this section can be used simply by mentioning their names in a specification line.

The name of the file containing each available auxiliary scanner is also given in this section. It is not necessary to examine this file in order to use the auxiliary scanner, but sometimes an existing auxiliary scanner can be useful as a starting point for solving a similar problem (see Section 1.2.2 [Building scanners], page 13).

`auxNUL` This routine is invoked automatically when the first character of a sequence is the ASCII NUL character, a pattern that cannot be specified by a regular expression. In that case, the character sequence matched by the associated pattern is an empty sequence. If information remains in the current input file, `auxNUL` returns a pointer to the empty sequence at the beginning of that information. Effectively, this is a pointer to the new information.

If information remains in the current input file, the library version of `auxNUL` (see Section 'Text Input' in Library Reference Manual) appends that information to the character sequence matched by the associated pattern, possibly relocating the character sequence matched by the associated pattern. It returns a pointer to the first character of the sequence matched by the associated pattern. Source code: '`$elipkg/Scan/auxNUL.c`'.

To obtain different behavior when the first character of a sequence is the ASCII NUL character, supply your own routine with the name `auxNUL` in a type-'`c`' file. The easiest way to do this is to copy the source code for the library routine into a local file and then modify it.

`auxEOF` This routine is invoked automatically when the first character of a sequence is the ASCII NUL character, a pattern that cannot be specified by a regular expression, and no information remains in the current input file. In that case, the character sequence matched by the associated pattern is an empty sequence.

The library version of `auxEOF` simply returns the argument supplied to it. Source code: '`$elipkg/Scan/auxEOF.c`'.

To obtain different behavior when the first character of a sequence is the ASCII NUL character, and no information remains in the current input file, supply your own routine with the name `auxEOF` in a type-'`c`' file. The easiest way to do this is to copy the source code for the library routine into a local file and then modify it.

`coordAdjust`

Leaves the character sequence matched by the associated pattern unchanged. Updates the coordinate information to reflect the tabs and newlines in that sequence. Source code: '`$elipkg/Scan/coordAdjust.c`'

`auxNewLine`

Leaves the character sequence matched by the associated pattern unchanged. Updates the coordinate information under the assumption that the last character of that sequence is a newline. (This is a special case that can be handled more efficiently than the general case, for which `coordAdjust` would be used.) Source code: '`$elipkg/Scan/auxNewLine.c`' `auxTab` Leaves the character sequence matched by the associated pattern unchanged. Updates the coordinate information under the assumption that the last character of that sequence is a tab. (This is a special case that can be handled more efficiently than the general case, for which `coordAdjust` would be used.) Source code: '`$elipkg/Scan/auxTab.c`'

`auxEOL` Extends the character sequence matched by the associated pattern to the end of the current line, including the terminating newline. Updates the coordinate information to reflect the new position. Source code: '`$elipkg/Scan/auxScanEOL.c`'

`auxCString`

Completes a C string constant when provided with the opening quote (`"`). Updates the coordinate information to reflect the tabs and newlines in that sequence. Source code: '`$elipkg/Scan/CchStr.c`'.

`auxCChar` Completes a C character constant when provided with the opening quote (`'`). Source code: '`$elipkg/Scan/CchStr.c`'.

`auxCComment`

Completes a C comment when provided with the opening delimiter (`/*`). Updates the coordinate information to reflect the tabs and newlines in the comment.

The comment is terminated by the delimiter `*/`, and may not contain nested comments.

Source code: '`$elipkg/Scan/Ccomment.c`'

`auxM2String`

Completes a string constant when provided with the opening quote, possibly followed by other characters. Updates the coordinate information to reflect the tabs in that sequence.

The string constant is terminated by an occurrence of the opening quote. If a newline or the end of the input text is reached before the constant terminates, `auxM2String` reports an error.

For Modula2, the opening quote is either the character `'` or the character `"`. This auxiliary scanner simply uses the first character of the string matched by the regular expression as the opening quote character, so it can complete any sequence of characters that is terminated by the first character, and is contained wholly within a single source line. Note that the characters matched by the regular expression are *not* re-scanned for a closing quote.

Source code: '`$elipkg/Scan/M2chStr.c`'

`auxM3Comment`

Completes a Modula2 or Modula3 comment when provided with the opening delimiter (`(*`). Updates the coordinate information to reflect the tabs and newlines in the comment.

The comment is terminated by the delimiter `*)`, and may contain nested comments.

Source code: '$elipkg/Scan/M3comment.c'

`auxPascalString`

Completes a string constant when provided with the opening quote, possibly followed by other characters. Updates the coordinate information to reflect the tabs in that sequence.

The string constant is terminated by an occurrence of the opening quote that is not immediately followed by another occurrence of the opening quote. (Thus the opening quote character may appear doubled within the string.) If a newline or the end of the input text is reached before the constant terminates, `auxPascalString` reports an error.

For Pascal, the opening quote is the character `'`. This auxiliary scanner simply uses the first character of the string matched by the regular expression as the opening quote character, so it can complete any sequence of characters that is terminated by a single occurrence of the first character, and not by two successive occurrences of that character, and is contained wholly within a single source line. Note that the characters matched by the regular expression are *not* re-scanned for a closing quote.

Source code: '$elipkg/Scan/pascalStr.c'

`auxPascalComment`

Completes a Pascal comment when provided with the opening delimiter (either `{` or `(*`). Updates the coordinate information to reflect the tabs and newlines in the comment.

A comment is terminated by either the delimiter `}` or the delimiter `*)`, regardless of the opening delimiter. Comments may *not* be nested.

Source code: '$elipkg/Scan/pascalCom.c'

`Ctext` Completes a C compound statement when provided with the opening brace (`{`). Updates the coordinate information to reflect the tabs and newlines in the compound statement.

A compound statement is terminated by the matching close brace (`}`). Compound statements may be nested, and unmatched braces may be embedded in C strings, character constants or comments.

Source code: '$elipkg/Scan/Ctext.c'

1.2.2 Building scanners

All auxiliary scanners obey the same interface conventions:

```
extern char *Name(char *start, int length);
/* Auxiliary scanner "Name"
```

```
*   On entry-
*     start points to the first character matching the associated
*       regular expression
*     length=number of characters matching the associated
*       regular expression
*   On exit-
*     Name points to the first character that does not belong to the
*       character sequence being classified
***/
```

Unless otherwise stated, `Name>=start` on return, and all characters in the half-open interval `[start,Name)` are in memory.

Any auxiliary scanner that passes over tabs or newline characters must update coordinate information (see Section 3.1 [Maintaining the Source Text Coordinates], page 25). In addition, if the character following a newline is an ASCII NUL then the source buffer must be refilled (see Section 'Text Input' in Library Reference Manual). The easiest way to develop an auxiliary scanner is therefore to start with one from the library that solves a similar problem. Source file names for all of the available auxiliary scanners are given in the previous subsection. To obtain a copy of (say) the source code for `auxNUL` as file '`MyScanner.c`' in your current directory, give the Eli request:

```
-> $elipkg/Scan/auxNUL.c > MyScanner.c
```

After modifying '`MyScanner.c`', simply add its name to your type-'`specs`' file to make it available.

1.3 Token Processors

A token processor is a routine to be invoked after the pattern described by the regular expression has been matched, and after any associated auxiliary scanner has been invoked. It is passed a pointer to the matching character sequence, the length of that sequence, a pointer to an integer variable containing the classification, and a pointer to an integer variable to hold a value representing the character sequence. The token processor may change the classification, and may compute a value to represent the sequence.

A token processor is invoked by giving its name, surrounded by brackets (`[ ]`), on the same line as the associated regular expression:

```
Integer: $[0-9]+  [mkint]
```

This specification invokes the token processor `mkint` whenever a sequence of digits is recognized. The arguments are a pointer to the first digit, the length of the digit sequence, a pointer to an integer variable containing the classification code for `Integer`, and a pointer to an integer variable to hold a value representing the digit sequence. As described below, `mkint` leaves the character sequence and its classification unchanged and sets the value to the decimal integer denoted by the digit sequence. This specification is the implementation of the canned description `PASCAL_INTEGER`.

This section describes the token processors that are available in the Eli library, and also explains how to implement token processors for tasks that are specific to your problem.

1.3.1 Available processors

All of the token processors described in this section can be used simply by mentioning their names in a specification line.

The name of the file containing each available token processor is also given in this section. It is not necessary to examine that file in order to use the token processor, but sometimes an existing token processor can be useful as a starting point for solving a similar problem (see Section 1.3.2 [Building Processors], page 16).

`c_mkchar`
: Assumes that the character sequence has the form of a C character constant. Sets the value to the integer encoding of that character constant. Does not alter the initial classification. Source file: '`$elipkg/Scan/CchStr.c`'.

`c_mkint`
: Assumes that the character sequence has the form of a C integer constant. Sets the value to the integer represented by that constant. Does not alter the initial classification. Source file: '`$elipkg/Scan/int.c`'.

`EndOfText`
: This processor is invoked automatically when the end of the input text is reached. It assumes that the character sequence is empty, and does nothing. Source file: '`$elipkg/Scan/dflteot.c`'.

: To obtain different behavior when the end of the input text is reached, supply your own routine with the name `EndOfText` in a type-'`c`' file. The easiest way to do this is to copy the source code for the library routine into a local file and then modify it.

`mkidn`
: Looks the character sequence up in the identifier table (see Section 'Unique Identifier Management' in Library Reference Manual). If it is not in the table, it is added with its classification unchanged. Otherwise `mkidn` changes the initial classification to the classification given by the identifier table. (The identifier table can be initialized with pre-classified character strings, see Chapter 4 [Literal Symbols], page 29.)

In any case, `mkidn` sets the value to the (unique) index of the character sequence in the character storage module (see Section 'Character String Storage' in Library Reference Manual). Source file: '`$elipkg/Scan/idn.c`'.

`mkint` Assumes that the character sequence consists of one or more decimal digits. Sets the value to the integer denoted by that sequence of digits. Does not alter the initial classification. Source file: '`$elipkg/Scan/int.c`'.

`mkstr` Stores a new copy of the character sequence in the character storage module and sets the value to the index of that copy (see Section 'Character String Storage' in Library Reference Manual). Does not alter the initial classification. Source file: '`$elipkg/Scan/str.c`'.

`modula_mkint`

Assumes that the character sequence consists of one or more hexadecimal digits, possibly followed by a radix marker. Sets the value to the integer denoted by that sequence of digits, interpreted in the given radix. Does not alter the initial classification.

Valid radix markers are `B` and `C` (indicating radix 8), and `H` (indicating radix 16). Sequences of digits not followed by a radix marker are assumed to be radix 10.

Source file: '`$elipkg/Scan/M2int.c`'.

1.3.2 Building processors

All token processors obey the same interface conventions:

```
extern void Name(char *start, int length, int *class, int *intrinsic);
/* Token processor "Name"
 *   On entry-
 *     start points to the first character of the sequence being classified
 *     length=length of the sequence being classified
 *     class points to a location containing the initial classification
 *     intrinsic points to a location to receive the value
 *   On exit-
 *     class points to a location containing the final classification
 *     intrinsic points to a location containing the value (if relevant)
 ***/
```

The token processor can change the classification of the character sequence. It may carry out any computation whatsoever, involving arbitrary modules, to obtain the information it needs. Eli generates a file called '`termcode.h`' that contains `#define` directives specifying the classification code for each symbol appearing before a colon at the beginning of a line in a type-'`gla`' file. Thus if `name: ...` is a line in a type-'`gla`' file, a processor can use the following sequence to change the classification of any character sequence, including one that is initially classified as a comment, to `name`:

```
#include "termcode.h"
...
   *class = name;
...
```

All comments are classified by the value of the symbol `NORETURN`, exported by the lexical analyzer module in file 'gla.h'. A token processor can cause the character sequence matched by its associated regular expression to be considered a comment by setting the classification to `NORETURN`:

```
#include "gla.h"
...
   *class = NORETURN;
...
```

The easiest way to develop a token processor is to start with one from the library that solves a similar problem. Source file names for all of the available token processors are given in the previous subsection. To obtain a copy of (say) the source code for `EndOfText` as file 'MyProcessor.c' in your current directory, give the Eli request:

```
-> $elipkg/Scan/dflteot.c > MyProcessor.c
```

After modifying 'MyProcessor.c', simply add its name to your type-'specs' file to make it available.

2 Canned Symbol Descriptions

For many applications, the exact structure of the symbols that must be recognized is not important or the problem description specifies that the symbols should be the same as the symbols used in some other situation (e.g. identifiers might be specified to use the same format as C identifiers). To cover this common situation, Eli provides a set of canned symbol descriptions.

To use a canned description, simply write the canned description's identifier in a specification instead of writing a regular expression. For example, the following type-'gla' file tells Eli that the input text will contain C-style identifiers and strings, Ada-style comments, and Pascal-style integers:

```
Identifier: C_IDENTIFIER
            ADA_COMMENT
String:     C_STRING_LIT
Integer:    PASCAL_INTEGER
```

`Identifier`, `String` and `Integer` would appear as non-literal terminal symbols in the context-free grammar defining the phrase structure of this input text (see Section 'How to describe a context-free grammar' in Syntax Analysis).

The available canned descriptions are defined later in this section. All of these definitions include a regular expression, and some include auxiliary scanners and/or token processors. An auxiliary scanner or token processor specified by a canned description can be overridden by nominating a different one in the specification that names the canned description. For example, the canned description `PASCAL_STRING` includes the token processor `mkstr` (see Section 1.2.1 [Available scanners], page 10). This token processor stores multiple copies of the same string in the character storage module. The following specification overrides `mkstr` with `mkidn`, which stores only one copy of each distinct string:

```
Str: PASCAL_STRING [mkidn]
```

The auxiliary scanner `auxPascalString`, included in the canned description, is not overridden by this specification.

The remainder of this section characterizes the canned descriptions that are available in the Eli library, and also gives their definitions.

2.1 Available Descriptions

Each of the identifiers in the following list is the name of a canned description specifying the lexical structure of some component of an existing programming language. Here they are simply characterized by the role they play in that language. A complete definition of each, consisting of a regular expression, possibly an auxiliary scanner name, and possibly a token processor name, is given in the next section.

When building a new language, it is a good idea to use canned descriptions for lexical components: Time is not wasted in deciding on their form, mistakes are not made in their implementation, and users are familiar with them.

The list also provides canned descriptions for spaces, tabs and newlines. These white space characters are treated as comments by default. If, however, you define any pattern that will accept a white space character in its first position, this pattern overrides the default treatment and that white space character will be accepted *only* in contexts that are specified explicitly (see Chapter 3 [Spaces Tabs and Newlines], page 25). For example, suppose that the following pattern were defined and that no other patterns contain spaces:

```
Separator:  $\040+#\040+
```

In that situation, a space will be accepted only if it is part of a `Separator`. To treat spaces that are not part of a `Separator` as comments, include the canned description `SPACES` as a comment specification:

```
Separator:  $\040+#\040+
            SPACES
```

Note that only a white space character that appears at the beginning of a pattern loses its default interpretation in this way. In this example, neither the tab nor the newline appeared at the beginning of a pattern and therefore tabs and newlines continue to be treated as comments.

`C_IDENTIFIER, C_INTEGER, C_FLOAT, C_STRING_LIT, C_CHAR_CONSTANT, C_COMMENT`

Identifiers, integer constants, floating point constants, string literals, character literals, and comments from the C programming language, respectively.

`PASCAL_IDENTIFIER, PASCAL_INTEGER, PASCAL_REAL, PASCAL_STRING, PASCAL_COMMENT`

Identifiers, integer constants, real constants, string literals, and comments from the Pascal programming language, respectively.

`MODULA2_INTEGER, MODULA2_CHARINT, MODULA2_LITERALDQ, MODULA2_LITERALSQ, MODULA2_COMMENT`
: Integer constants, characters specified using character codes, string literals delimited by double and single quotes, and comments from the Modula-2 programming language, respectively.

`MODULA3_COMMENT`
: Comments from the Modula-3 programming language.

`ADA_IDENTIFIER, ADA_COMMENT`
: Identifiers and comments from the Ada programming language.

`AWK_COMMENT`
: Comments from the AWK programming language.

`SPACES` Sequence of one or more spaces.

`TAB` A single horizontal tab.

`NEW_LINE` A single newline.

2.2 Definitions of Canned Descriptions

Eli textually replaces a reference to a canned description with its definition. If a user nominates an auxiliary scanner and/or a token processor for a canned description, that overrides the corresponding nomination appearing in the definition of the canned description.

The following is an alphabetized list of the canned descriptions available in the Eli library, with their definitions. Use this list as a formal definition, and as an example for constructing specifications. (`C_FLOAT` and `PASCAL_REAL` have definitions that are too long to fit on one line of this document. Each is, however, a single line in the specification file.)

```
ADA_COMMENT
        $-- (auxEOL)


ADA_IDENTIFIER
        $[a-zA-Z](_?[a-zA-Z0-9])* [mkidn]


AWK_COMMENT
        $# (auxEOL)
```

```
C_COMMENT
          $"/*" (auxCComment)

C_CHAR_CONSTANT
          $' (auxCChar) [c_mkchar]

C_FLOAT   $((([0-9]+\.[0-9]*|\.[0-9]+)((e|E)(\+|-)?[0-9]+)?)|
          ([0-9]+(e|E)(\+|-)?[0-9]+))[fFlL]? [mkstr]

C_IDENTIFIER
          $[a-zA-Z_][a-zA-Z_0-9]* [mkidn]

C_INTEGER
          $([0-9]+|0[xX][0-9a-fA-F]*)[lLuU]? [c_mkint]

C_STRING_LIT
          $\" (auxCString) [mkstr]

MODULA2_COMMENT, MODULA3_COMMENT
          $\(\* (auxM3Comment)

MODULA2_CHARINT
          $[0-9][0-9A-Fa-f]*C [modula_mkint]

MODULA2_INTEGER
          $[0-9][0-9A-Fa-f]*[BH]? [modula_mkint]

MODULA2_LITERALDQ
          $\" (auxM2String) [mkstr]

MODULA2_LITERALSQ
          $\' (auxM2String) [mkstr]
```

```
PASCAL_COMMENT
          $"{"|"(*" (auxPascalComment)

PASCAL_IDENTIFIER
          $[a-zA-Z][a-zA-Z0-9]*  [mkidn]

PASCAL_INTEGER
          $[0-9]+  [mkint]

PASCAL_REAL
          $(([0-9]+\.[0-9]+)|((e|E)(\+|-)?[0-9]+)?)|([0-9]+(e|E)(\+|-)?[0-9]+)
          [mkstr]

PASCAL_STRING
          $' (auxPascalString) [mkstr]

SPACES    $\040+

TAB       $\t (auxTab)

NEW_LINE  $\n (auxNewLine)
```

3 Spaces, Tabs and Newlines

An Eli-generated processor examines its input text sequentially, recognizing character sequences in the order in which they appear. At each point it matches the longest possible sequence, classifies that sequence, and then begins anew with the next character. If the first character of a sequence is a space, tab or newline then the default behavior is to classify the sequence consisting of that character and all succeeding spaces, tabs and newlines as a comment. This behavior is consistent with the definitions of most programming languages, and is reasonable in a large fraction of text processing tasks.

Even though tabs and newlines are considered comments by default, some processing is needed to account for their effect on the source text position. Eli-generated processors define a two-dimensional coordinate system (line number and column index), which they use to link error reports to the source text (see Section 'Source Text Coordinates and Error Reporting' in Library Reference Manual).

White space may be significant in two situations:

1. Within a character sequence, such as spaces in a string
2. On its own, such as line boundaries in a type-'`gla`' file

Appropriate white space may be specified as part of the description of a complete character sequence (provided that it is not at the beginning) without disrupting the default behavior. (Coordinate processing for tabs and newlines must be provided if they are allowed within the sequence.) The default behavior is overridden, however, by any specification of white space on its own or at the beginning of another character sequence. Overriding is specific to the white space character used: a specification of new behavior for a space overrides the default behavior for a space, but not the default behavior for a tab or newline.

The following sections explain how coordinate processing is provided for newlines and tabs, and how to re-establish default behavior of white space on its own when white space can occur at the beginning of another character sequence.

3.1 Maintaining the Source Text Coordinates

The raw data for determining coordinates are two variables, `LineNum` (an integer variable exported by the error module, see Section 'Source Text Coordinates and Error Reporting' in Library

Reference Manual) and `StartLine` (a character pointer exported by the lexical analyzer). The following invariant must be maintained on these variables:

```
LineNum=Cumulative index of the current line in the input text
(Pointer to current character)-StartLine=index of the current character
    in the current line
```

This invariant must hold whenever the lexical analyzer begins to process a character sequence. It may be destroyed during the processing of that sequence, but must be re-established before processing of the next character sequence begins.

`LineNum` is initially 1, and must be incremented each time the lexical analyzer advances beyond a newline character in the input text. At the beginning of each line, `StartLine` must be set to point to the character position preceding the first character of that line. As the current character pointer is advanced, the condition on `StartLine` is maintained automatically unless the character pointer advances over a tab character.

A tab character in the input text represents one or more spaces, depending upon its position relative to the next tab stop, but it occupies only one character position. If the tab represents *n* spaces, *n-1* must be subtracted from `StartLine` to maintain the invariant.

Because the value of *n* depends upon the index of the current character and the settings of the tab stops in the line, Eli provides an operation `TABSIZE(i)` (defined in file '`tabsize.h`') to compute it. The argument `i` is the index in the current line of the character position beyond that containing the tab, and the result is the number of spaces that must be added to reach the next tab stop.

Suppose that `p` is a pointer to the current input character. Here is a code sequence that maintains the condition on `StartLine` when a tab is encountered:

```
#include "tabsize.h"
...
  if ((*p++) == '\t') StartLine -= TABSIZE(p - StartLine);
...
```

`TABSIZE` defines the positions of the tab stops. The default implementation provides tab stops every 8 character positions. A user changes this default by supplying a new version of the Eli library routine `TabSize`. The source code for the library version of this routine can be obtained by making the following request:

```
-> $elipkg/gla/tabsize.c > MyTabSize.c
```

After modifying the routine appropriately, add the name `MyTabSize.c` to your type-'`specs`' file.

The coordinate invariant is maintained automatically if no patterns matching tabs or newline characters are defined, and no auxiliary scanners that advance over tabs or newline characters are provided by the user. If such patterns or scanners are needed, then the user must define them in such a way that they maintain the coordinate invariant.

Three auxiliary scanners (`coordAdjust`, `auxTab` and `auxNewLine`) are available to maintain the coordinate invariant for a regular expression that matches tabs or newline characters (see Section 1.2.1 [Available scanners], page 10). While these auxiliary scanners could be invoked by user-defined auxiliary scanners that advance over tabs or newline characters, it is often simpler to include the appropriate code to maintain the coordinate invariant.

For an example of the use of code in an auxiliary scanner to maintain the coordinate invariant, see the library version of `auxNUL`.

3.2 Restoring the Default Behavior for White Space

When a pattern beginning with a space, tab or newline character overrides the default behavior for that character, the character will only be accepted as part of an explicit pattern. The default behavior can be restored by using one of the canned descriptions `SPACES`, `TAB` or `NEWLINE` respectively (see Section 2.1 [Available Descriptions], page 20):

```
Define:  $\040+define
         SPACES
```

Here the pattern for `Define` overrides the default behavior for space characters. If this were the only specification, spaces in the input text would only be accepted if they occurred immediately before the character sequence `define`. By adding the canned description `SPACES`, and classifying the sequences it matches as comments, the default behavior is restored.

Note that this specification is ambiguous: A sequence of spaces followed by `define` could either match the `Define` pattern or the spaces alone could be classified as the comment specified by `SPACES`. The principle of the longest match guarantees that in this case the sequence will be classified as `Define` (see Section 1.1.4 [Ambiguity], page 9).

4 Literal Symbols

If the generated processor includes a parser (see Syntactic Analysis), then Eli will extract the descriptions of any literal terminal symbols from the context-free grammar defining that parser and add them to the specifications provided by type-'gla' files. For example, consider the following context-free grammar:

```
Program: Expression .
Expression: Evaluation / Binding .
Evaluation:
  Constant / BoundVariable /
  '(' Expression '+' Expression ')' /
  '(' Expression '*' Expression ')' .
Binding: 'let' BoundVariable '=' Evaluation 'in' Expression .
```

This grammar has nine terminal symbols. Two (`Constant` and `BoundVariable`) are given by identifiers, and the other seven (`(`, `)`, `+`, `*`, `let`, `=` and `in`) are given by literals.

Only the character sequences to be classified as `Constant` or `BoundVariable`, and those to be classified as comments, need be defined by type-'gla' files. Descriptions of the symbols given as literals will be automatically extracted from the grammar by Eli. Thus the lexical analyzer for this language might be described by a single type-'gla' file containing the following:

```
Constant:      PASCAL_INTEGER
BoundVariable: PASCAL_IDENTIFIER
               PASCAL_COMMENT
```

4.1 Overriding the Default Treatment of Literal Symbols

By default, a literal terminal symbol specified in a context-free grammar supplied to Eli will be recognized as though it had been specified by the appropriate regular expression. Thus the literal symbols `'+'` and `'let'` will be recognized as though the following specifications had been given by the user:

```
Plus:  $\+
Let:   $let
```

(Here Plus and Let are arbitrary identifiers describing the initial classifications of the literal symbols. No such identifiers are actually supplied by Eli, but the literal symbols are *not* initially classified as comments.)

In some situations it is useful to carry out more complex operations at the time the literal symbol is recognized. In this case, the user must do two things:

1. Mark the literal symbol as being a special case.
2. Provide a specification for the literal symbol.

As a concrete example, suppose that %% were used as a major separator in the input text and could appear either once or twice. Assume that the first occurrence is required, and the second is optional. All text following the second occurrence is to be ignored.

One approach to this problem would be to count the number of occurrences of the literal symbol %%, advancing to the end of the input text after the second. This could be done by an auxiliary scanner (see Section 1.2 [Auxiliary Scanners], page 10) that either returns a pointer to the character following the %% or a pointer to the ASCII NUL terminating the input text, and a token processor (see Section 1.3 [Token Processors], page 14) that reclassifies the second occurrence of %% as a comment. The grammar would specify only the required first occurrence of %%.

In order to mark the literal symbol %% as a special case that should not receive the default treatment, the user must supply a type-'delit' file specifying that symbol as a regular expression. The entry in the type-'delit' file also needs to define an identifier to represent the classification:

```
$%%  PercentPercent
```

Each line of a type-'delit' file consists of a regular expression and an identifier, separated by white space. The regular expression must describe a literal symbol appearing in a context-free grammar supplied to Eli. That literal symbol will not be incorporated automatically into the generated lexical analyzer; it must be specified explicitly by the user. The identifier will be given the appropriate value by an Eli-generated #define directive in file 'litcode.h'.

In our example, %% could be specified by the following line of a type-'gla' file:

```
$%%  (SkipOrNot) [CommentOrNot]
```

Initially, the separator will be classed as a comment because there is no identifier preceding the regular expression. SkipOrNot will use a state variable to decide whether or not to skip text (see Section 1.2.2 [Building scanners], page 13), while CommentOrNot will use the same state variable to decide whether or not to change the classification to PercentPercent (see Section 1.3.2 [Building Processors], page 16):

```
#include <fcntl.h>
#include "source.h"
#include "litcode.h"

static int Second = 0;

char *
SkipOrNot(char *start, int length)
{ if (!Second) return start + length;
  (void)close(finlBuf());
  initBuf("/dev/null", open("/dev/null", O_RDONLY));
  return TEXTSTART;
}

void
CommentOrNot(char *start, int length, int *class, int *intrinsic)
{ if (!Second) { Second++; *class = PercentPercent; }
}
```

The remainder of the text is skipped by closing the current input file and opening an empty file to read (see Section 'Text Input' in Library Reference Manual). Since %% is initially classified as a comment, its classification must be changed only on the first occurrence.

File 'fcntl.h' defines open and O_RDONLY, 'source.h' defines initBuf, finlBuf and TEXTSTART, and 'litcode.h' defines PercentPercent.

4.2 Using Literal Symbols to Represent Other Things

In some cases the phrase structure of a language depends upon layout cues rather than visible character sequences. For example, indentation is used in Occam2 to indicate block structure: If the first non-blank character of a line is indented further than the first non-blank character of the line preceding it, then the new line begins a new block. If the first non-blank character of a line is not indented as far as the first non-blank character of the line preceding it, then the old line ends one or more blocks depending on the difference in indentation. If the first non-blank characters of two successive lines are indented by the same amount, then the lines simply contain adjacent statements of the same block.

Layout cues can be represented by literal symbols in the context-free grammar that describes the phrase structure. The processing needed to recognize the layout cues can then be described in any convenient manner, and the sequence of white space characters implementing those cues can be classified as the appropriate literal symbol.

Suppose that the beginning of a block is represented in the Occam2 grammar by the literal symbol `'{'`, the statement separator by `';'`, and the end of a block by `'}'`. In the input text, blocks and statement separators are defined by layout cues as described above. A type-'`delit`' file marks the literal symbols as requiring special recognition and associates an identifier with each:

```
$\{  Initiate
$;  Separate
$\}  Terminate
```

Indentation can be specified as white space following a new line:

```
$\n[\t\040]*  [OccamIndent]
```

The token processor `OccamIndent` would carry out all of the processing necessary to determine the meaning of the indentation. This processing is complex, involving interactions with several other components of the generated lexical analyzer (see Section 6.4 [An Example of Interface Usage], page 42). It constitutes an operational definition of the meaning of indentation in Occam2.

5 Case Insensitivity

The default behavior of an Eli-generated lexical analyzer is to treat each ASCII character as an entity distinct from all other ASCII characters. This behavior is inappropriate for applications that do not distinguish upper-case letters from lower-case letters in certain contexts. For example, a Pascal compiler ignores the case of letters in identifiers and keywords, but distinguishes them in strings. Thus the Pascal identifiers `MyId`, `MYID` and `myid` are identical but the strings `'MyString'`, `'MYSTRING'` and `'mystring'` are different.

Case insensitivity is reflected in the identity of character sequences. In other words, the character sequences `MyId`, `MYID` and `myid` are considered to be identical character sequences if and only if the generated processor is insensitive to the case of letters. Two character sequences are identical as far as the remainder of the processor is concerned if they have the same classification and their values are equal (see Chapter 1 [Specifications], page 3). Since the classification and value are determined by the token processor, it is the token processor that must implement case insensitivity.

Two conditions must be met if a processor is to be insensitive to case:

1. A token processor that maintains a table of character sequences in which all letters are of one case must be available.
2. The specification of each case-insensitive character sequence must invoke such a token processor.

5.1 A Case-Insensitive Token Processor

The token processor `mkidn` maintains a table of character sequences and provides the same classification and value for identical character sequences. Normally, `mkidn` treats upper-case letters and lower-case letters as different characters. This behavior is controlled by an exported variable, `dofold` (see Section 'Unique Identifier Management' in Library Reference Manual): When `dofold=0` character sequences are entered into the table as they are specified to `mkidn`; otherwise all letters in the sequence are converted to upper case before the sequence is entered into the table.

Although the value of `dofold` could be altered on the basis of context by user-defined code, it is normally constant throughout the processor's execution. To generate a processor in which `dofold=1`, specify the parameter `+fold` in the request (see Section 'fold – Make the Processor Case-Insensitive' in Products and Parameters Reference Manual). If this parameter is not specified in the request, Eli will produce a processor with `dofold=0`.

The value set by `mkidn` is the (unique) index of the transformed character sequence in the table. Thus if that value is used to retrieve the sequence at a later time, the result will be the original sequence with all lower-case letters replaced by their upper-case equivalents.

5.2 Making Literal Symbols Case Insensitive

Since literal symbols are recognized exactly as they stand in the grammar, they are case sensitive by definition. For example, if a grammar for Pascal contains the literal symbol `'begin'` then the generated processor will recognize only the character sequence `begin` as an instance of that literal symbol. This behavior could be changed by redefining the literal symbol as a nonliteral symbol (say) `BEGIN`, and providing the following specification in a type-'`gla`' file:

```
BEGIN:  $[Bb][Ee][Gg][Ii][Nn]  [mkidn]
```

If the number of literal symbols to be treated as case-insensitive is large, this is a very tedious and error-prone approach. It also distorts the grammar by converting literal terminal symbols to non-literal terminal symbols.

To solve this problem, Eli allows the user to specify a set of literal symbols that should be placed into the table used by `mkidn`, with their classification codes, at the time the generated lexical analyzer is loaded. If the `+fold` parameter is also specified, all lower-case letters in these symbols will be replaced by their upper-case equivalents before the symbol is placed into the table. The desired behavior is then obtained by invoking `mkidn` after recognizing the appropriate character sequence in the input text.

The set of literal symbols to be placed into the table is specified by giving a sequence of regular expressions in a type-'`gla`' file, and then deriving the `:kwd` product from that file (see Section 'kwd – Recognize Specified Literals as Identifiers' in Products and Parameters Reference Manual). The regular expressions describe the form of the literal symbols in the grammar, *not* the input character sequences to be recognized.

Suppose, for example, that a Pascal grammar specified all keywords as literal symbols made up of lower-case letters:

```
Statement:
  ...
  'while' Expression 'do' Statement /
  ...
```

A type-'gla' file describing the form these symbols take in the grammar would consist of the single line $[a-z]+. If the name of that file was 'PascalKey.gla' then the user could tell Eli to initialize mkidn's table with all of the keywords by including the following line in a type-'specs' file:

```
PascalKey.gla :kwd
```

In Pascal, keywords have the form of identifiers in the input text. Therefore the canned description PASCAL_IDENTIFIER suffices to recognize both identifiers and keywords. PASCAL_IDENTIFIER invokes mkidn to obtain the classification and value of the sequence recognized by the regular expression $[a-zA-Z][a-zA-Z0-9]*. Since mkidn's table has been initialized with the character sequences for the literal keyword symbols, and their classifications, they will be appropriately recognized.

The :kwd product and the +fold parameter are independent of one another. Thus, in order to make the generated lexical analyzer accept Pascal keywords with arbitrary case the user must both provide the :kwd specification and derive with the +fold parameter.

6 The Generated Lexical Analyzer Module

This chapter discusses the generated lexical analyzer module, its interface, and its relationship to other modules in the generated processor. An understanding of the material here is not necessary for normal use of the lexical analyzer.

There are some special circumstances in which it is necessary to change the interactions between the lexical analyzer and its environment. For example, there is a mismatch between the lexical analyzer and the source code input module of a FORTRAN 90 compiler: The unit of input text dealt with by the source code module is the line, the unit dealt with by the lexical analyzer is the statement, and there is no relationship between lines and statements. One line may contain many statements, or one statement may be spread over many lines. This mismatch problem is solved by requiring the two modules to interact via a buffer, and managing that buffer so that it contains both an integral number of lines and an integral number of statements. Because the lexical analyzer normally works directly in the source module's buffer, that solution requires a change in the relationship between the lexical analyzer and its environment.

The interaction between the lexical analyzer and its environment is governed by the following interface:

```
#include "gla.h"
/* Entities exported by the lexical analyzer module
 * NORETURN (constant) Classification of a comment
 * ResetScan (variable) Flag causing scan pointer reset
 * TokenStart (variable) Address of first classified character
 * TokenEnd (variable) Address of first unclassified character
 * StartLine (variable) Column index = (TokenEnd - StartLine)
 * glalex (operation) Classify the next character sequence
 ***/
```

There are three distinct aspects of the relationship between the lexical analyzer and its environment, and each is dealt with in one section of this chapter. First we consider how the lexical analyzer selects the character sequence to be scanned, then we see how the lexical analyzer's attention can be switched, and finally how the classification results are reported.

6.1 Interaction Between the Lexical Analyzer and the Text

There is no internal storage for text in the lexical analyzer module. Instead, `TokenEnd` is set to point to arbitrary text storage. (Normally the pointer is to the source buffer, see Section

'Text Input' in Library Reference Manual.) The text pointed to must be an arbitrary sequence of characters, the last of which is an ASCII NUL.

At the beginning of a scan, `TokenEnd` points to the beginning of the string on which a sequence is to be classified. The lexical analyzer tests that string against its set of regular expressions, finding the longest sequence that begins with the first character and matches one of the regular expressions.

If the regular expression matched is associated with an auxiliary scanner then that auxiliary scanner is invoked with the matched sequence (see Section 1.2.2 [Building scanners], page 13). The auxiliary scanner returns a pointer to the first character that should not be considered part of the character sequence being matched, and that pointer becomes the value of `TokenEnd`. `TokenStart` is set to point to the first character of the string.

When no initial character sequence matches any of the regular expressions an error report is issued, `TokenEnd` is advanced by one position (thus discarding the first character of the string), and the process is restarted. If the string is initially empty, no attempt is made to match any regular expressions. Instead, the auxiliary scanner `auxNUL` is invoked immediately. If this auxiliary scanner returns a pointer to an empty string then the auxiliary scanner `auxEOF` is invoked immediately. Finally, if `auxEOF` returns a pointer to an empty string then the Token processor `EndOfText` is invoked immediately. (If either `auxNUL` or `auxEOF` returns a pointer to a non-empty string, scanning begins on this string as though `TokenEnd` had pointed to it initially.)

`TokenStart` addresses a sequence of length `TokenEnd-TokenStart` when a token processor is invoked (see Section 1.3 [Building Processors], page 14). Because `TokenStart` and `TokenEnd` are exported variables, the token processor may change them if that is appropriate. All memory locations below the location pointed to by `TokenStart` are undefined in the fullest sense of the word: Their contents are unknown, and they may not even exist. Memory locations beginning with the one pointed to by `TokenStart`, up to but not including the one pointed to by `TokenEnd`, are known to contain a sequence of non-NUL characters. `TokenEnd` points to a sequence of characters, the last of which is an ASCII NUL. If the token processor modifies the contents of `TokenStart` or `TokenEnd`, it must ensure that these conditions hold after the modification.

6.2 Resetting the Scan Pointer

If the exported variable `ResetScan` is non-zero when the operation `glalex` is invoked, the lexical analyzer's first action is to execute the macro `SCANPTR`. `SCANPTR` guarantees that `TokenEnd` addresses the string to be scanned. If `ResetScan` is zero when `glalex` is invoked, `TokenEnd` is assumed to address that string already. `ResetScan` is statically initialized to `1`, meaning that `SCANPTR` will be executed on the first invocation of `glalex`.

In the distributed system, `SCANPTR` sets `TokenEnd` to point to the first character of the source module's text buffer. Since this is also the first character of a line, `StartLine` must also be set (see Section 3.1 [Maintaining the Source Text Coordinates], page 25):

```
#define SCANPTR { TokenEnd = TEXTSTART; StartLine = TokenEnd - 1; }
```

See Section 'Text Input' in Library Reference Manual. This implementation can be changed by supplying a file '`scanops.h`', containing a new definition of `SCANPTR`, as one of your specification files.

`ResetScan` is set to zero after `SCANPTR` has been executed. Normally, it will never again have the value `1`. Thus `SCANPTR` will not be executed on any subsequent invocation of `glalex`. Periodic refilling of the source module's text buffer and associated re-setting of `TokenEnd` is handled by `auxNUL` when the lexical analyzer detects that the string is exhausted. More complex behavior, using `ResetScan` to force resets at arbitrary points, is always possible via token processors or other clients.

6.3 The Classification Operation

The classification operation `glalex` is invoked with a pointer to an integer variable that may be set to the value representing the classified sequence. An integer result specifying the classification is returned by `glalex`, and the coordinates of the first character of the sequence are stored in the error module's exported variable `curpos` (see Section 'Source Text Coordinates and Error Reporting' in Library Reference Manual).

There are three points at which these interactions can be altered:

1. Setting coordinate values
2. Deciding on a continuation after a classification
3. Returning a classification

All of these alterations are made by supplying macro definitions in a specification file called '`scanops.h`'. The remainder of this section defines the macro interfaces and gives the default implementations.

6.3.1 Setting coordinate values

The coordinates of the first character of a sequence are set by the macro `SETCOORD`. Its default implementation uses the standard coordinate invariant (see Section 3.1 [Maintaining the Source Text Coordinates], page 25):

```
/* Set the coordinates of the current token
 *   On entry-
 *     LineNum=index of the current line in the entire source text
 *     p=index of the current column in the entire source line
 *   On exit-
 *     curpos has been updated to contain the current position as its
 *     left coordinate
 */
#define SETCOORD(p) { LineOf(curpos) = LineNum; ColOf(curpos) = (p); }
```

When execution monitoring (see Section 'Monitoring' in Monitoring) is in effect, more care must be taken. In addition to the above, `SETCOORD` must also set the cumulative column position, which is the column position within the overall input stream (as opposed to just the current input file). Ordinarily the two column positions will be the same, so the default implementation of `SETCOORD` for monitoring is:

```
#define SETCOORD(p) { LineOf(curpos) = LineNum; \
      ColOf(curpos) = CumColOf(curpos) = (p); }
```

When monitoring, it is also necessary to set the coordinates of the first character beyond the sequence. This is handled by the macro `SETENDCOORD`:

```
/* Set the coordinates of the end of the current token
 *   On entry-
 *     LineNum=index of the current line in the entire source text
 *     p=index of the current column in the entire source line
 *   On exit-
 *     curpos has been updated to contain the current position as its
 *     right coordinate
 */
#ifndef SETENDCOORD
#define SETENDCOORD(p) { RLineOf(curpos) = LineNum; \
 RColOf(curpos) = RCumColOf(curpos) = (p); }
#endif
```

6.3.2 Deciding on a continuation after a classification

Classification is complete after the regular expression has been matched, any specified auxiliary scanner invoked, and any specified token processor invoked. At this point, one of three distinct actions is possible:

`RETURN v` Terminate the invocation of `glalex`, returning the value `v` as the classification.

`goto rescan`
: Start a new scan at the character addressed by `TokenEnd`, without changing the coordinate value.

`continue` Start a new scan at the character addressed by `TokenEnd`, resetting the coordinates to the coordinates of that character.

`WRAPUP` is the macro responsible for deciding among these possibilities. When it is executed, `TokenEnd` addresses the first character beyond the classified sequence and `extcode` holds the classification code. Here is the default implementation:

```
#define WRAPUP { if (extcode != NORETURN) RETURN extcode; }
```

If `WRAPUP` does not transfer control, the result is the `continue` action. Thus the default implementation of `WRAPUP` terminates the invocation of `glalex` if the current character sequence is not classified as a comment (`extcode != NORETURN`), and starts a new scan at the next character if the current character sequence is classified as a comment.

If execution monitoring is in effect, the classification event must be reported in addition to selecting a continuation:

```
#define WRAPUPMONITOR { \
  if (extcode != NORETURN) { \
    char save = *TokenEnd; \
    *TokenEnd = '\0'; \
    generate_token("token", LineOf(curpos), ColOf(curpos), \
    CumColOf(curpos), RLineOf(curpos), RColOf(curpos), \
    RCumColOf(curpos), TokenStart, TokenEnd - TokenStart, \
    *v, extcode); \
    *TokenEnd = save; \
  } \
}
```

`WRAPUPMONITOR` is invoked instead of `WRAPUP` if execution monitoring is in effect.

6.3.3 Returning a classification

Once the decision has been made to terminate the glalex operation and report the classification, it is possible to carry out arbitrary operations in addition to returning the classification code. For example, execution monitoring requires that this event be reported. Here is the default implementation:

```
#ifdef MONITOR
#define RETURN(v) { generate_leave("lexical"); return v; }
#else
#define RETURN(v) { return v; }
#endif
```

6.4 An Example of Interface Usage

Recognition of Occam2 block structure from indentation is an example of how a token processor might use the lexical analyzer interface (see Section 4.2 [Using Literal Symbols to Represent Other Things], page 31). The token processor OccamIndent is invoked after a newline character (possibly followed by spaces and/or tabs) has been recognized:

```
#include "err.h"
#include "gla.h"
#include "source.h"
#include "litcode.h"

extern char *auxNUL();
extern char *coordAdjust();

#define MAXNEST 50
static int IndentStack[MAXNEST] = {1};
static int *Current = IndentStack;

void
OccamIndent(char *start, int length, int *class, int *intrinsic)
{ if (start[length] == '\0') {
    start = auxNUL(start, length);
    if (start[length] != '\0') { TokenEnd = start; return; };
    TokenEnd = start + length;
  }

  if (*TokenEnd == '\0' && Current == IndentStack) return;

  { char *OldStart = StartLine;
    int OldLine = LineNum, Position;
```

```
        (void)coordAdjust(start, length); Position = TokenEnd-StartLine;
        if (*Current == Position) *class = Separate;
        else if (*Current < Position) {
          *class = Initiate;
          if (Current == IndentStack + MAXNEST)
            message(DEADLY, "Nesting depth exceeded", 0, &curpos);
          *++Current = Position;
        } else {
          *class = Terminate; Current--;
          LineNum = OldLine; StartLine = OldStart; TokenEnd = start;
        }
      }
    }
```

Since the source buffer is guaranteed only to hold an integral number of lines (see Section 'Text Input' in Library Reference Manual), OccamIndent must first refill the buffer if necessary. The library routine auxNUL carries out this task, returning a pointer to the character sequence passed to it (see Section 1.2.1 [Available scanners], page 10). Remember that the character sequence may be moved in the process of refilling the buffer, and therefore it is vital to reset both start and TokenEnd after the operation.

If auxNUL is invoked and adds characters to the buffer, then those characters might be white space that should have been part of the original pattern. In this case OccamIndent can return, having set TokenEnd to point to the first character of the original sequence. Since the sequence was initially classified as a comment (because the specification did not begin with an identifier followed by a colon, see Section 4.2 [Using Literal Symbols to Represent Other Things], page 31), the overall effect will be to re-scan the newline and the text now following it.

If auxNUL is invoked but does not add characters to the buffer, then the newline originally matched is the last character of the file. TokenEnd should be set to point to the character following the newline.

When the end of the file has been reached, and no blocks remain unterminated, then the newline character has no meaning. By returning under these conditions, OccamIndent classifies the newline as a comment. Otherwise, the character sequence matched by the pattern must be interpreted on the basis of the indentation it represents.

Because a single character sequence may terminate any number of blocks, it may be necessary to interpret it as a sequence of terminators. The easiest way to do this is to keep re-scanning the same sequence, returning one terminator each time, until all of the relevant blocks have been terminated.

In order to make that possible, `OccamIndent` must save the current values of the pointer from which column indexes are determined (`StartLine`) and the cumulative line number (`LineNum`).

The pattern with which `OccamIndent` is associated will match a character sequence beginning with a newline and containing an arbitrary sequence of spaces and tabs. To determine the column index of the first character following this sequence, apply `coordAdjust` to it (see Section 1.2.1 [Available scanners], page 10). That auxiliary scanner leaves the character sequence unchanged, but re-establishes the invariant on `LineNum` and `StartLine` (see Section 3.1 [Maintaining the Source Text Coordinates], page 25). After the invariant is re-established, the column index can be computed.

`Current` points to the element of `IndentStack` containing the column index of the first character of a line belonging to the current block. (If no block has been opened, the value is 1.) When the column index of the character following the initial white space is equal to this value, that white space should be classified as a separator. Otherwise, if the column index shows an indentation then the white space should be classified as an initiator and the new column position should be pushed onto the stack. Stack overflow is a deadly error, making further processing impossible (see Section 'Source Text Coordinates and Error Reporting' in Library Reference Manual). Finally, if the column index shows an exdentation then the white space should be classified as a terminator and the column position for the terminated block deleted from the stack.

When a newline terminates a block, it must be re-scanned and interpreted in the context of the text surrounding the terminated block. Therefore in this case `StartLine` and `LineNum` are restored to the values they had before `coordAdjust` was invoked, and `TokenStart` is set to point to the newline character at the start of the sequence. Thus the next invocation of the lexical analyzer will again recognize the sequence and invoke `OccamIndent` to interpret it.

Index

Z