FORTRAN Lexical Analysis Specification

W. M. Waite

July 7, 1998

Abstract

This document describes the lexical analysis problem for FORTRAN. It is a part of an Eli specification from which compilers for FORTRAN 77 and FORTRAN 90 can be generated. Generation of the lexical analyzer is controlled by a FunnelWeb macro. The following selects the code for FORTRAN 77 analysis (`0` selects the code for FORTRAN 90 analysis):

Fortran77[1] ≡
`1`This macro is invoked in definitions 3, 26, 28, 30, 33, 37, 38, 53, 54, 59, 61, 63, 73, 82, 86, 87, 88, 95, 96, 97, and 104.

The generated FORTRAN 90 compiler uses a command line option to decide whether to accept fixed- or variable-format source text.

Only the lexical analysis task (scanning and computation of intrinsic attributes) is covered in this document. Because of the ad-hoc nature of basic symbol definition in FORTRAN, a mixture of declarative and operational specifications is necessary. The declarative specifications are regular expressions, and are used to describe some of the character strings the FORTRAN scanner must recognize. Descriptions for other strings are extracted automatically by Eli from the context-free grammar defining FORTRAN's phrase structure. Those descriptions need not be repeated here.

This specification was developed while the author was a visiting researcher at the GMD in Berlin, and was originally published as Arbeitspapiere der GMD 816 in January of 1994.

Contents

1 Eli Library Modules Used **4**
1.1 The Command Line Processing Module . . . 4
1.2 The Error Reporting Module . . . 4
1.3 The Source Text Input Module . . . 5
1.4 The Memory Object Management Module . . . 5
1.5 The Character String Storage Module . . . 6

2 The Generated Scanner Module **7**
2.1 Establish a Scan Pointer . . . 7
2.2 Set the Coordinates of a Token . . . 8
2.3 Set the Extent of a Token . . . 8
2.4 Define an Auxiliary Scanner . . . 9
2.5 Define a Token Processor . . . 10
2.6 Deal With an Unacceptable Token . . . 11

3 Units of Text **12**
3.1 The Coordinate Map . . . 13
3.1.1 Mark a line change . . . 13
3.1.2 Set Token Coordinates . . . 14
3.2 Operations on Source Text Lines . . . 15
3.2.1 Predicates classifying lines . . . 16
3.2.2 Positioning operations . . . 18
3.2.3 Operations that extract information from a line . . . 21
3.3 Statement Buffer Construction . . . 24
3.3.1 Load fixed-format text . . . 25
3.3.2 Load variable-format text . . . 26
3.4 Character Sequence Normalization . . . 29
3.4.1 FORTRAN Character Conversion Table . . . 30
3.4.2 Normalization for Fixed-Format Input . . . 30
3.4.3 Normalization for Variable-Format Input . . . 31

4 Token Classification **31**
4.1 Parser Resolution of Token Classification . . . 32
4.2 Assignment Statement Recognition . . . 33
4.2.1 Case analysis for assignment . . . 34
4.2.2 Check the Remainder of a Logical IF . . . 35

5 Controlling the Scanner **36**

6 Identifiers and Keywords **37**
6.1 Distinguishing Identifiers From Keywords 38
6.2 Variable-Format Keyword Test 39
6.3 Fixed-Format Keyword Test 40
6.4 Keyword Recognition in Fixed-Format Text 41
6.5 Keywords in I/O Statements 42

7 Denotations **43**
7.1 Integer Denotations 44
7.2 Floating-Point Denotations 45
7.3 String Denotations 46
7.4 Hollerith Denotations 48
7.5 Operator Denotations 49

8 Special Problems **50**
8.1 Format Descriptors 50
8.2 Concatenation Operator 51
8.3 Array Constructor Brackets 52
8.4 Letter Ranges for IMPLICIT Statements 53

9 Specification Files **55**
9.1 scan.clp 55
9.2 scanops.h 55
9.3 scan.c 55
9.4 scan.gla 57
9.5 scan.delit 57

1 Eli Library Modules Used

The scanner uses library modules to report errors, obtain source code, manage variable-sized data structures, store character information, and guarantee that only one copy of a string is stored.

Eli Library Modules Used[2] ≡

The Error Reporting Module[4]
The Source Text Input Module[5]
The Memory Object Management Module[6]
The Character String Storage Module[7]
The Unique Identifier Module[14]

This macro is invoked in definition 104.

This section briefly describes the facilities of these modules that are used in the scanner. For a complete specification of each module, consult the Eli Library Reference Manual.

1.1 The Command Line Processing Module

The command line processing module recognizes the command line options and sets appropriate information for the generated processor. At a minimum, the generated processor must be able to access information from an input file. Additionally, the `-f` command line option is used to specify that the input to a FORTRAN 90 processor must obey the fixed-format rules.

The Command Line Processing Module[3] ≡

```
InputFile input "File to be processed";
#if !Fortran77[1]
FixedFormat "-f" boolean "Select fixed input format";
#endif
```

This macro is invoked in definition 102.

This specification says that a positional parameter is to be recognized, and its string attached as a property of the known key `InputFile`. The value of that property will be taken as the processor's input file. If no parameter is specified, standard input will be used.

1.2 The Error Reporting Module

The error reporting module defines the coordinate system and handles output of error reports from all components of an Eli-generated compiler.

The Error Reporting Module[4] ≡

```
#include "err.h"
/* Exported entities used in the FORTRAN scanner:
 *   POSITION   (type):         Source text coordinates
 *   ERROR      (constant):     Severity indicating output cannot be run
 *   curpos     (variable):     Storage for token coordinates
 *   message    (operation):    Report an error
 ***/
```

This macro is invoked in definition 2.

When the scanner constructs a token, it places the coordinates of the first character of the string represented by that token into `curpos`. Errors detected by the scanner itself are reported via the `message` operation. Any of these errors is sufficient to prevent the compiler from producing executable code, and hence they are reported with severity `ERROR`.

1.3 The Source Text Input Module

The source text input module provides access to the source text as a sequence of lines. It guarantees that if the first character of a line is in memory then all of the characters of that line, including the terminating newline character, are in contiguous memory locations. The newline terminating the last line in memory is followed by a `NUL` character, and thus the sequence of lines constitutes a C string.

The Source Text Input Module[5] ≡

```
#include "source.h"
/* Exported entities used in the FORTRAN scanner:
 *   TEXTSTART  (variable):     Pointer to a line of input text
 *   LineNum    (variable):     Index of the current text buffer line
 *   refillBuf  (operation):    Refill the text buffer from the source file
 ***/
```

This macro is invoked in definition 2.

`TEXTSTART` initially points to the first character of the first line of the source text, and `LineNum` initially has the value 1. It is the responsibility of the source module's client to maintain the values of `TEXTSTART` and `LineNum` so that they satisfy some appropriate condition. When all of the lines in the buffer have been examined, `refillBuf` can be invoked to obtain more text (if there is more in the file). On return from `refillBuf`, `TEXTSTART` contains a pointer to the new text or (if no more text is available) a pointer to a null string.

1.4 The Memory Object Management Module

Eli uses a dynamic storage mechanism called an *Obstack* to store data structures whose size cannot be accurately predicted ahead of time. Data is added to the Obstack via a sequence of "growth" operations, and the growth terminated by an invocation of `obstack_finish`:

The Memory Object Management Module[6] ≡

```
#include "obstack.h"
/* Exported entities used in the FORTRAN scanner:
 *   ObstackP          (type):         Variable-size storage area
 *   obstack_init      (operation):    Initialize using defaults
 *   obstack_begin     (operation):    Initialize using specified values
 *   obstack_grow      (operation):    Add data to the current contiguous area
 *   obstack_1grow     (operation):    Add one character to the current string
 *   obstack_finish    (operation):    Complete the current growth
 *   obstack_free      (operation):    Cut back the storage in use
 ***/
```

This macro is invoked in definition 2.

All of the information added between one invocation of `obstack_finish` and the next is guaranteed to be stored contiguously, and the pointer returned by `obstack_finish` points to the beginning of that contiguous storage area.

Obstack storage is allocated in a stack-like fashion: If `obstack_free` is applied to a pointer returned by `obstack_finish`, all of the storage allocated after the *previous* `obstack_finish` is freed. Subsequent growth will re-use that storage.

1.5 The Character String Storage Module

The character string storage module provides both temporary and permanent storage for character strings.

The Character String Storage Module[7] ≡

```
#include "csm.h"
/* Exported entities used in the FORTRAN scanner:
 *   NoStr      (constant):     Non-existent string
 *   Csm_obstk  (constant):     Dynamic string storage facility
 *   CsmStrPtr  (variable):     Pointer to string in dynamic string storage
 *   stostr     (operation):    Make a stored string permanent
 ***/
```

This macro is invoked in definition 2.

`NoStr` represents a non-existent string (in contrast to the empty string "", which certainly exists). It is used as a value of a pointer to a string when there is no string to point to. `Csm_obstk` represents the module's dynamic string storage facility in all Obstack operations. This facility can be used to store arbitrary strings, temporarily or permanently. When a string is to be stored permanently, the result of the `obstack_finish` operation that defined it should be stored in `CsmStrPtr` and then `stostr` invoked on `CsmStrPtr`. Any string pointer, including `CsmStrPtr`, can be used to describe strings that are stored temporarily. Storage for temporary strings must be freed explicitly via the `obstack_free` operation.

2 The Generated Scanner Module

The scanner module generated by Eli recognizes sequences of input characters and classifies them according to criteria embodied in the specifications. It reports the coordinates (line and column) of the beginning of the character sequence, and associates an integer value with the sequence.

The Generated Scanner Module[8] ≡

```
#include "gla.h"
/* Exported entities used in the FORTRAN scanner support routines
 *   NORETURN   (constant):     Classification of an ignored character sequence
 *   TokenStart (variable):     Pointer to the current character sequence
 *   TokenEnd   (variable):     Pointer to the first unprocessed character
 *   ResetScan  (variable):     True if TokenEnd is invalid
 ***/
```

This macro is invoked in definition 104.

The specifications from which the scanner is generated can describe character sequences that are to be ignored by the remainder of the generated program. For example, comments are usually ignored when translating a programming language, yet they must be recognized by the scanner. All such sequences are given the classification code `NORETURN` by the scanner, and their presence is not reported to the routine that invoked the scanner. `NORETURN` is exported by the scanner module to make it available to user-defined modules that override some of the internal operations of the scanner, as described below.

The scanner module has no internal storage for text. Instead, `TokenEnd` is used to specify the text to be scanned. If it is valid when the scanner is invoked, `TokenEnd` points to a sequence of characters stored in contiguous memory locations, the last of which contains a zero byte. The value of `ResetScan` on invocation of the scanner determines whether `TokenEnd` is valid.

`TokenStart` is set by the scanner to point to the first character of the current character sequence.

Several of the internal operations of an Eli-generated scanner can be replaced by user-defined versions. These changes allow the user to specify certain aspects of the scanner's behavior operationally, while specifying others declaratively. The remainder of this section describes the operations that can be replaced, and how they relate to the rest of the translation task.

2.1 Establish a Scan Pointer

The position of the scanner in the input text is defined by `TokenEnd`, a character pointer exported by the scanner module. `TokenEnd` points to the next character to be examined by the scanner. When the scanner is invoked, it checks the content of its exported variable `ResetScan`. If `ResetScan` is nonzero, the scanner sets it to zero and uses the macro `SCANPTR` to obtain a non-null value for `TokenEnd`:

Establish a Scan Pointer[9] ≡

```
/* Establish a scan pointer
 *   If no further text is available then on exit-
 *     TokenEnd points to a null string
 *   Otherwise on exit-
 *     TokenEnd points to a string that is guaranteed to contain
 *       a newline character
 */
#define SCANPTR
```
This macro is invoked in definition 57.

In an Eli-generated processor, the source text module is initialized before the scanner is invoked for the first time. Thus the initial invocation of `SCANPTR` can assume that the exit condition of the source buffer initialization operation holds, provided that the user has not supplied any additional operations to invalidate that condition.

`ResetScan` may be set nonzero by any user-supplied procedure, causing `SCANPTR` to be invoked at the beginning of the next invocation of the scanner.

`SCANPTR` normally sets `TokenEnd` to the value of `TEXTSTART`, a string pointer exported by the source module. `ResetScan` is not normally set by any component of the generated compiler, so `SCANPTR` is only executed on the first scanner invocation.

2.2 Set the Coordinates of a Token

The scanner establishes coordinates for each token that it recognizes. These coordinates are used primarily for associating reports with appropriate positions in the source text. `SETCOORD` is a macro that places the coordinates of the current text position into the variable `curpos`, exported by the error reporting module.

Set the Coordinates of a Token[10] ≡

```
/* Set the coordinates of the current token
 *   On entry-
 *     p=index of the current position in the current source line
 *   On exit-
 *     curpos=coordinates of the current position
 */
#define SETCOORD(p)
```
This macro is invoked in definition 21.

`SETCOORD` normally sets the line coordinate to the value of `LineNum`, an integer variable exported by the error reporting module, and sets the column coordinate to the value of its argument `p`.

2.3 Set the Extent of a Token

When monitoring is being used, the scanner must indicate the coordinates of the character following the token, as well as the coordinates of the first character. These coordinates are used for associating extents of input text with tokens. `SETENDCOORD` is a macro that places the ending coordinates of the current token into the variable `curpos`. The appropriate fields of the curpos structure are only available when monitoring support is requested or if `RIGHTCOORD` macro is defined.

Set the Extent of a Token[11] ≡

```
/* Set the coordinates of the end of the current token
 *   On entry-
 *     p=index of the current position in the current source line
 *   On exit-
 *     endpos=coordinates of the current position
 */
#define SETENDCOORD(p)
```
This macro is invoked in definition 21.

`SETENDCOORD` normally sets the line coordinate to the value of `LineNum`, an integer variable exported by the error reporting module, and sets the column coordinate to the value of its argument `p`.

2.4 Define an Auxiliary Scanner

In the specification of a pattern defining a basic symbol, the user can nominate an auxiliary scanner. It will be invoked after the scanner has recognized the pattern described by the regular expression (the declarative part of a basic symbol specification). All auxiliary scanners obey the same interface:

Define an Auxiliary Scanner[12](◇1) ≡

```
char *
#ifdef PROTO_OK
◇1(char *start, int length)
#else
◇1(start, length)
char *start; int length;
#endif
/* Standard interface for an auxiliary scanner
 *   On entry-
 *     start points to the first character of the scanned string
 *     length=length of the scanned string
 *   On exit-
 *     The function returns a pointer to the first character
 *       beyond the scanned string
 ***/
```

This macro is invoked in definitions 55 and 83.

Normally, an auxiliary scanner changes the length of the character sequence matched by the pattern. This allows the user to specify operationally patterns that are very tedious to describe with regular expressions.

Eli nominates the auxiliary scanner `auxNUL` for the end-of-text pattern, which the user is not allowed to specify via a regular expression. On entry to `auxNUL`, `start` points to a zero byte and `length=0`. A default routine, which simply returns the value of `start`, will be used if this routine is left unspecified.

If `auxNUL` returns a pointer to a non-null string, the generated scanner will immediately scan that string, considering it to be a continuation of the string being scanned when the end-of-text pattern was recognized. In this case the scanner will not return any indication that the end-of-text pattern was recognized.

2.5 Define a Token Processor

In the specification of a pattern defining a basic symbol, the user can nominate a token processor. It will be invoked after the scanner has recognized the pattern described by the regular expression, and after any specified auxiliary scanner has been invoked. All token processors obey the same interface:

Define a Token Processor[13](◇1) ≡

```
void
#ifdef PROTO_OK
◇1(char *start, int length, int *klass, int *intrinsic)
#else
◇1(start, length, klass, intrinsic)
char *start; int length, *klass; int *intrinsic;
#endif
/* Standard interface for a processor
 *   On entry-
 *     start points to the first character of the scanned string
 *     length=length of the scanned string
 *     klass points to a location containing the initial classification
 *     intrinsic points to a location to receive the intrinsic attribute
 *   On exit-
 *     klass points to a location containing the final classification
 *     intrinsic points to a location containing the intrinsic attribute value
 *       (if relevant)
 ***/
```

This macro is invoked in definitions 60, 63, 65, 67, 70, 77, 81, 83, 85, 87, 91, 94, 97, and 100.

Normally, a token processor calculates a value for the intrinsic attribute from the characters of the scanned string. It may also change the classification of the scanned string and/or the value of `TokenEnd`.

Eli nominates the token processor `EndOfText` for the end-of-text pattern, which the user is not allowed to specify via a regular expression. On entry to `EndOfText`, `start` points to a zero byte, `length=0`, and `klass` points to a location containing the classification code for the end of the text. A default routine, which simply returns, will be used if this routine is left unspecified. `EndOfText` will not be invoked if `auxNUL` returns a pointer to a non-null string.

The token processor mkidn is used to guarantee that only one copy of a specific string appears in the character storage module. This token processor obeys the standard interface given above. It is part of the unique identifier management module, whose interface is the file idn.h.

The Unique Identifier Module[14] ≡

```
#include "idn.h"
```

This macro is invoked in definition 2.

2.6 Deal With an Unacceptable Token

The parser determines the phrase structure of the sequence of tokens supplied by the scanner. It builds this phrase structure from left to right, examining one token beyond the currently-accepted sequence. If that token is a valid continuation of the sequence, then it is accepted and the sequence extended. Because of the technique used to create the parser, a symbol will only be accepted if it is a valid continuation of the current sequence.

It might be that a particular sequence of characters could be interpreted as any of several different tokens. The scanner must make a choice among these possibilities, returning that choice to the parser, but it does not have information about whether this choice would be acceptable.

When the parser finds a token unacceptable, it invokes the routine Reparatur with a description of the unacceptable token. Reparatur may choose to alter the token and request that the parser decide whether the altered token is acceptable:

Deal With an Unacceptable Token[15] ≡

```
int
#ifdef PROTO_OK
Reparatur(POSITION *coord, int *klass, int *intrinsic)
#else
Reparatur(coord, klass, intrinsic)
POSITION *coord; int *klass, *intrinsic;
#endif
/* Repair a syntax error by changing the lookahead token
 *   On entry-
 *     coord points to the coordinates of the lookahead token
 *     klass points to the classification of the lookahead token
 *     intrinsic points to the intrinsic attribute of the lookahead token
 *   If the lookahead token has been changed then on exit-
 *     Reparatur=1
 *     coord, klass and intrinsic reflect the change
 *   Else on exit-
 *     Reparatur=0
 *     coord, klass and intrinsic are unchanged
```

```
***/
```

This macro is invoked in definition 50.

A default routine, which returns `0` without modifying the token, will be used if `Reparatur` is left unspecified.

3 Units of Text

The source text input module provided by the Eli library guarantees that a complete line (including its terminating newline character) is stored contiguously in memory. By default, an Eli-generated translator therefore treats one line as a "unit of text": Scanning operations are normally restricted to one line, and unless special actions are taken no token may cross a line boundary.

This view is inappropriate for FORTRAN. The presence of continuation lines means that a single FORTRAN statement can be spread over an arbitrary number of lines, with any token broken between lines at arbitrary points. In FORTRAN 90, it is also possible to write a number of statements on the same line.

The statement is the natural unit of text for a FORTRAN scanner to store contiguously in memory: Basic symbols cannot span statement boundaries. Recognition of certain constructs involves extensive lookahead, but lookahead beyond the end of a statement is never required.

A classic structure clash like the one between the source module's lines and the scanner's statements is solved by using a buffer that contains integral numbers of *both* kinds of object. The buffer is filled by operations on the largest object that is a component of each of the clashing objects.

The FORTRAN structure clash is solved by filling a buffer with the statements from a sequence of lines. This buffer is terminated at the first point where the end of a statement coincides with the end of a line. Characters are the largest objects common to both lines and statements, and therefore the buffer is filled by operations on characters.

The statement buffer is implemented as an Obstack containing a single string. Source text operations define an abstract data type used to conceal the details of access to input lines. Information needed for precise error reporting is stored in a coordinate map:

Units of Text[16] ≡

```
static Obstack Statement;
static char *Stmt = NoStr;
```

The Coordinate Map[17]

Operations on Source Text Lines[23]
Statement Buffer Construction[37]

This macro is invoked in definition 104.

3.1 The Coordinate Map

The source text coordinates (line number and character position) are a part of the information that the scanner is required to provide for a token. Thus, although the scanner deals with a buffer reflecting statement structure rather than line structure, information about the source text position of a character must be obtainable. That information is stored in an array that is created dynamically as the statement buffer is filled:

The Coordinate Map[17] ≡

```
typedef struct { int IndexInStmt, LineIndex, Offset; } MapElement;
static MapElement *Map, Current;
static Obstack MapData;
```

Coordinate-setting routine[22]

This macro is invoked in definition 16.

The array has one element for each point in the statement where the coordinates of the character do not have the same line number as the coordinates of its predecessor within that statement. The coordinates of characters following that point, up to the next such point, can be determined from their distance from that point and the map element for that point.

There is also an element for the first character position of the statement buffer because it has no predecessor, and an element for the character position beyond the end of the statement buffer to provide an upper limit for the lookup operation.

`IndexInStmt` is the index of the element's character position in the statement buffer. `LineIndex` is the line number of the character at that position, and (`IndexInStmt - Offset`) is its column number. `Map` points to the completed array for the statement buffer, while `Current` is used in constructing the map elements.

3.1.1 Mark a line change

Map elements marking line changes are created as follows:

Mark a line change[18](◇1) ≡

```
{ Current.LineIndex = LineNum;
  Current.Offset = Current.IndexInStmt - ((◇1) - TEXTSTART + 1);
  obstack_grow(&MapData, &Current, sizeof(Current)); }
```

This macro is invoked in definitions 38, 39, 42, and 44.

`LineNum` is a variable exported by the error reporting module. Its value is initially 1, and it is neither set nor examined by any operation of the error module or the source module.

`TEXTSTART` is a variable exported by the source module. Each source module operation that delivers text sets `TEXTSTART` to point to the start of the text it has delivered, but otherwise `TEXTSTART` is neither set nor examined by the source module.

The operations on source text lines guarantee that `LineNum` and `TEXTSTART` satisfy the following condition at appropriate points:

Invariant for the source text coordinate system[19] ≡

```
 * LineNum=index of the current source line in the entire text
 * TEXTSTART points to the first character of the current source line
```

This macro is invoked in definitions 29 and 34.

`Current.Offset` is maintained by the statement buffer construction operations. For fixed-format text, updates are fixed by the definition of FORTRAN. When constructing a statement buffer from variable-format text the following operation is used:

Add a character to the statement buffer[20](◇1) ≡

```
obstack_1grow(&Statement, ◇1); Current.IndexInStmt++;
```

This macro is invoked in definitions 38, 40, and 44.

3.1.2 Set Token Coordinates

The scanner operation to set the coordinates of a token must be defined to use the map information. In order to avoid a search in the map for each token, `CoordBase` is kept pointing to the map element defining the last line change.

Set Token Coordinates[21] ≡

```
Set the Coordinates of a Token[10] \
  { extern void TokenCoords ELI_ARG((int, POSITION *, int)); \
    TokenCoords(p, &curpos, 0); }

Set the Extent of a Token[11] \
  { extern void TokenCoords ELI_ARG((int, POSITION *, int)); \
    TokenCoords(p, &curpos, 1); }
```

This macro is invoked in definition 103.

Coordinate-setting routine[22] ≡

```
static MapElement *CoordBase;

void
#ifdef PROTO_OK
TokenCoords(int p, POSITION *pos, int right)
#else
```

```
TokenCoords(p, pos, right)
int p; POSITION *pos; int right;
#endif
{ while (p >= CoordBase[1].IndexInStmt) CoordBase++;
  if (!right) {
    pos->line = CoordBase->LineIndex;
#ifdef MONITOR
    pos->col = pos->cumcol = p - CoordBase->Offset;
#else
    pos->col = p - CoordBase->Offset;
#endif
  } else {
#ifdef RIGHTCOORD
    pos->rline = CoordBase->LineIndex;
#ifdef MONITOR
    pos->rcol = pos->rcumcol = p - CoordBase->Offset;
#else
    pos->rcol = p - CoordBase->Offset;
#endif
#endif
  }
}
```

This macro is invoked in definition 17.

The `while` loop ensures that the proper map element is being used. The change defined after the newline character marking the end of the statement guarantees that this loop will always terminate.

3.2 Operations on Source Text Lines

Three classes of operations are provided on source text lines:

Operations on Source Text Lines[23] ≡

Predicates classifying lines[24]
Positioning operations[28]
Operations that extract information from a line[33]

This macro is invoked in definition 16.

These operations can be thought of as defining an abstract data type that embodies the essential properties of FORTRAN source text: All access to the source text is handled by these operations, all of which behave properly when the source text is exhausted.

3.2.1 Predicates classifying lines

FORTRAN distinguishes three classes of line: comment, continuation and initial. The predicates given in this section provide operational definitions of the first two classes; the third consists of lines not belonging to the first two. The end of the source text is classified as an initial line.

Predicates classifying lines[24] ≡

> *Comment Lines*[26]
> *Continuation Lines*[27]

This macro is invoked in definition 23.

Both of the line classification predicates have the same interface specification:

Line classification predicate:[25](◇1) ≡

```
char *
#ifdef PROTO_OK
◇1(char *p)
#else
◇1(p)
char *p;
#endif
/* Standard interface for a line classification predicate
 *   On entry-
 *     p points to the first character of a source text line
 *   If the scanning operation is satisfied then on exit-
 *     The function returns a pointer to the first unexamined character
 *   Otherwise on exit-
 *     The function returns a null pointer
 ***/
```

This macro is invoked in definitions 26 and 27.

Comment Lines Lines with `C` or `*` in the first character position are comments in FORTRAN 77 text and in the fixed-format of FORTRAN 90, but not in the variable format. In FORTRAN 90 a line whose first nonblank characater is an exclamation mark is also a comment line, except when the exclamation mark occurs in character position 6 and the text is in fixed format. Lines that do not contain any non-blank characters are always comment lines.

The following code provides an operational description of these rules; it is satisfied if the line pointed to by `p` on entry is a comment line. If `IsComment` is satisfied, it returns a pointer to the first character of the next source text line:

Comment Lines[26] ≡

Line classification predicate:[25](‘IsComment’)

```
{ register char *q;

  if (!*p) return NoStr;
  for (q = p; *q == ' ' || *q == '\t'; q++) ;
  switch (*q) {
  case '\n':
    LineNum++; return q+1;
  default:
    return NoStr;
#if !Fortran77[1]
  case '!':
    if (FixedFormat && q == p+5) return NoStr; break;
#endif
  case 'C': case 'c': case '*':
    if (!FixedFormat || q != p) return NoStr;
  }
  do { q++; } while (*q != '\n');
  LineNum++; return q+1;
}
```

This macro is invoked in definition 24.

Continuation Lines Lines with spaces in positions 1-5 and a non-blank, non-zero character in position 6 are continuation lines in FORTRAN 77 text and in the fixed-format text of FORTRAN 90. A line whose first non-blank character is an ampersand is a continuation line in FORTRAN 90 variable format text. Finally, if the last nonblank, non-comment character of a FORTRAN 90 variable-format text line is an ampersand then the next non-comment line is a continuation line.

The last rule involves interaction among lines, and is therefore not within the competence of any line scanning operation. It is described as a part of the statement buffer construction process.

The following code provides an operational description of the first two rules; it is satisfied if the line pointed to by `p` on entry is a continuation line:

Continuation Lines[27] ≡

Line classification predicate:[25](‘IsContinue’)

```
{
  if (!*p) return NoStr;
  if (FixedFormat) {
    register int i;
    for (i = 0; i < 5; i++) if (p[i] != ' ') return NoStr;
    if (p[5] == ' ' || p[5] == '\t') return NoStr;
```

```
      if (p[5] == '0') { p[5] = ' '; return NoStr; }
      return p+6;
    } else {
      register char c;
      while ((c = *p++) == ' ' || c == '\t') ;
      return c == '&' ? p : NoStr;
    }
  }
```

This macro is invoked in definition 24.

3.2.2 Positioning operations

Certain contexts require source lines of certain classes. The positioning operations are used to guarantee that the current line satisfies a particular requirement.

Positioning operations[28] ≡

```
Advance to the next non-comment line[30]
Advance to the next initial line[31]
#if !Fortran77[1]
Advance to a new file if necessary[32]
#endif
```

This macro is invoked in definition 23.

All of the positioning operations have the same interface specification:

Line positioning operation:[29](◇1) ≡

```
void
#ifdef PROTO_OK
◇1(char *p)
#else
◇1(p)
char *p;
#endif
/* Standard interface for a line positioning operation
 *   On entry-
 *     p points to the first character of a source text line
 *     LineNum is the index of the source line pointed to by p
 *   On exit-
Invariant for the source text coordinate system[19]
 *     The line pointed to by TEXTSTART is of the desired class
 ***/
```

This macro is invoked in definitions 30, 31, and 32.

Advance to the next non-comment line Comment lines play no role whatsoever in FORTRAN statements. They must be recognized in the source text and their presence reflected in the line index, but they need not be retained.

Advance to the next non-comment line[30] ≡

```
Line positioning operation:[29]('NextNonComment')
{ char *next;
  for (;;) {
    if (!*p) {
      refillBuf(p);
      p = TEXTSTART;
#if !Fortran77[1]
      if (!*p) p = ContinuationText();
#endif
    }
    if (!(next = IsComment(p))) break;
    p = next;
  }
#if Fortran77[1]
  TEXTSTART = p;
#else
  NextIncludedLine(p);
#endif
}
```

This macro is invoked in definition 28.

If the source text buffer remains empty after an invocation of `refillBuf`, the end of the source file has been reached. In FORTRAN 90, the source file may be one named in an `INCLUDE` directive. `ContinuationText` will check for this situation and handle it properly. If the source text buffer remains empty after return from `ContinuationText` then there is no further input text of any kind.

Advance to the next initial line At the beginning of any FORTRAN text, and at every statement beginning a new line in variable-format FORTRAN 90 text, the next initial line must be found:

Advance to the next initial line[31] ≡

```
Line positioning operation:[29]('NextInitialLine')
{
  for (;;) {
    char *next;
    NextNonComment(p);
    if (!(next = IsContinue(TEXTSTART))) return;
```

```
    { POSITION e; e.line = LineNum; e.col = next - TEXTSTART + 1;
      message(ERROR, "Continuation without initial line", 0, &e);
      while (*next++ != '\n') ;
      LineNum++;
      p = next;
    }
  }
}
```

This macro is invoked in definition 28.

`NextInitialLine` returns a pointer to the next initial line, and reports an error at the first significant character position of any continuation line preceding that initial line.

Advance to a new file if necessary A FORTRAN 90 `INCLUDE` directive is not a statement. It must occur on a line by itself (possibly followed by a comment), and may not be *within* any statement. Thus the first non-comment line of any included file must be an initial line, and the first non-comment line following the `INCLUDE` directive must also be an initial line.

Advance to a new file if necessary[32] ≡

Line positioning operation:[29](`'NextIncludedLine'`)

```
{ char c, *q;

  if (!(TEXTSTART = p)) return;
  StartLine = p - 1;

  while ((c = *p++) == ' ' || c == '\t') {
    if (c == '\t') StartLine -= TABSIZE(p - StartLine);
  }

  q = "include";
  while (*q) {
    if (F77Fold[c] != *q) return;
    c = *p++; q++;
  }

  while ((c = *p++) == ' ' || c == '\t') {
    if (c == '\t') StartLine -= TABSIZE(p - StartLine);
  }

  if (c != '\'' && c != '"' ) return;
  curpos.line = LineNum; curpos.col = p - StartLine;
  p = fstr(p - 1, 1); q = CsmStrPtr;
```

```
    while ((c = *p++) == ' ' || c == '\t') {
      if (c == '\t') StartLine -= TABSIZE(p - StartLine);
    }

    if (c == '!') while (c != '\n') c = *p++;
    LineNum++;
    if (c == '\n') {
      TEXTSTART = p;
      if (!ReadingFrom(FindFile(q)))
        message(ERROR, "Cannot open include file", 0, &curpos);
      p = TEXTSTART;
    } else {
      curpos.col = p - StartLine;
        message(ERROR, "Only a comment can follow INCLUDE", 0, &curpos);
      while (c != '\n') c = *p++;
    }

    obstack_free(Csm_obstk, q);
    NextInitialLine(p);
  }
```

This macro is invoked in definition 28.

`ReadingFrom` is an operation exported by the Eli Include module.

3.2.3 Operations that extract information from a line

Source text extraction operations obtain information from the source text buffer, placing that information into one of two Obstacks. The information is transformed by expanding tabs as it is extracted. Tabs are expanded at this stage because the amount of space represented by each depends on its position in the source text line. This position is trivially available during extraction, but could only be obtained at great cost later.

The constraints on the information in a line and the subsequent processing needs are different for the fixed and variable input formats, so separate extraction operations are required:

Operations that extract information from a line[33] ≡

> *Extract fixed-format text*[35]
> `#if !`*Fortran77*[1]
> *Extract variable-format text*[36]
> `#endif`

This macro is invoked in definition 23.

No variable-format extraction operation is required when a FORTRAN 77 scanner is being generated.

Both of the extraction operations have the same interface specification:

Line extraction operation:[34](◇1) ≡

```
void
#ifdef PROTO_OK
◇1(char *p)
#else
◇1(p)
char *p;
#endif
/* Standard interface for a line extraction operation
 *   On entry-
 *     The current line is not a comment line
 *     p points to the first character to be extracted
Invariant for the source text coordinate system[19]
 *   On exit-
 *     The source text is positioned at the next non-comment line
Invariant for the source text coordinate system[19]
 ***/
```

This macro is invoked in definitions 35 and 36.

Extract fixed-format text Information on a fixed-format line extends up to, but not beyond, character position 72. A line that has fewer than 72 characters is assumed to contain spaces in the missing positions. The fixed-format extraction routine therefore pads short lines and discards the excess characters of long lines:

Extract fixed-format text[35] ≡

```
Line extraction operation:[34]('ExtractFixedLine')
{ register char c;
  char *Position0 = TEXTSTART - 1;

  while (p - Position0 <= 72) {
/* Invariant: (p - Position0) indexes the first unfilled column
 *            p points to the first unprocessed character
 */
    if ((c = *p) == '\n') { obstack_1grow(&Statement, ' '); Position0--; }
    else {
      p++;
      if (c != '\t') obstack_1grow(&Statement, c);
      else {
        register int size = TABSIZE(p - Position0);
        do obstack_1grow(&Statement, ' ');
```

```
        while (size-- && (p - Position0--) <= 72);
      }
    }
  }

  while (*p++ != '\n') ;
  LineNum++;
  NextNonComment(p);
}
```

This macro is invoked in definition 33.

Information is placed directly into the statement buffer by `ExtractFixedLine`, because the fixed format guarantees that all of the information actually belongs to the statement.

Extract variable-format text Information on a variable-format line may occupy any number of character positions up to 132 (no attempt is made to check this limit). The variable-format extraction routine therefore simply transfers the characters:

Extract variable-format text[36] ≡

```
char *CurrentLine = NoStr;

Line extraction operation:[34]('ExtractVariableLine')
{ register char c;
  char *Position0 = TEXTSTART - 1;

  if (CurrentLine) obstack_free(Csm_obstk, CurrentLine);
  if (*p) {
    while ((c = *p++) != '\n') {
      if (c != '\t') obstack_1grow(Csm_obstk, c);
      else {
        register int size = TABSIZE(p - Position0);
        Position0 -= size;
        do obstack_1grow(Csm_obstk, ' '); while (size--);
      }
    }
    obstack_1grow(Csm_obstk, '\n');
  }
  obstack_1grow(Csm_obstk, '\0');
  CurrentLine = obstack_finish(Csm_obstk);
  LineNum++;
  NextNonComment(p);
}
```

This macro is invoked in definition 33.

In the variable format it is necessary to examine lines that are adjacent in the statement, but not necessarily adjacent in source text. The content of the second line is used to determine whether it is a continuation of the first and, if so, how the two should be joined. Thus the current line is stored temporarily in the character storage module's Obstack by `ExtractVariableLine`. This allows access to the current line via `CurrentLine`, and to the next non-comment line via `TEXTSTART`.

3.3 Statement Buffer Construction

The technique used to load the statement buffer depends on the format of the source text. For FORTRAN 77, only the fixed-format buffer loading technique is relevant, but for FORTRAN 90 the technique must be selected on the basis of the command line argument. Therefore routines implementing both techniques must be incorporated when generating a FORTRAN 90 scanner:

Statement Buffer Construction[37] ≡

> *Load fixed-format text*[39]
>
> `#if !`*Fortran77*[1]
> *Load variable-format text*[40]
> `#endif`
>
> *Load the Statement Buffer*[38]

This macro is invoked in definition 16.

After the statement buffer has been loaded, the string pointed to by `Stmt` is either null (indicating no further text is available) or it contains an integral number of FORTRAN statements. The last character of the string is a newline character. On each request to load the statement buffer, the previous contents (if any) of the statement buffer and associated coordinate map are discarded and the space reused:

Load the Statement Buffer[38] ≡

```
void
LoadStmtBuffer()
{
  if (Stmt) {
    obstack_free(&Statement, (void *)Stmt);
    obstack_free(&MapData, (void *)Map);
  } else {
    obstack_init(&Statement);
    obstack_init(&MapData);
    NextInitialLine(TEXTSTART);
  }
```

```
  Current.IndexInStmt = 1;
Mark a line change[18](‘TEXTSTART’)

  if (*TEXTSTART) {
#if Fortran77[1]
    LoadFixedFormat();
#else
    if (FixedFormat) LoadFixedFormat(); else LoadVariableFormat();
#endif
Add a character to the statement buffer[20](‘'\n'’)
  }

Add a character to the statement buffer[20](‘'\0'’)
Mark a line change[18](‘TEXTSTART’)
  Stmt = (char *)obstack_finish(&Statement);
  CoordBase = Map = (MapElement *)obstack_finish(&MapData);
}
```

This macro is invoked in definition 37.

During the process of loading the statement buffer, `TEXTSTART` points to the first character of the first unused line in the source text.

3.3.1 Load fixed-format text

In fixed-format text, the statement buffer contains characters 1-72 of the initial line and characters 7-72 of each of the following continuation lines (if any):

Load fixed-format text[39] ≡

```
void
LoadFixedFormat()
{ ExtractFixedLine(TEXTSTART);
  Current.IndexInStmt += 72;

  for (;;) {
    char *next;
    if (!(next = IsContinue(TEXTSTART))) return;
Mark a line change[18](‘next’)
    ExtractFixedLine(next);
    Current.IndexInStmt += 66;
  }
}
```

This macro is invoked in definition 37.

3.3.2 Load variable-format text

In variable-format text, the amount of information in the statement buffer coming from a line depends upon the contents of that line. Thus a state must be maintained as the lines are scanned in the source buffer:

Load variable-format text[40] ≡

```
void
LoadVariableFormat()
{ register char J, *p;
  int
    JSQ = 0,            /* 1 if within a string delimited by ' */
    JDQ = 0,            /* 1 if within a string delimited by " */
    JHOLL = 0,          /* 1 if within a Hollerith constant */
    HCount = -1;        /* Possible Hollerith count */

  ExtractVariableLine(TEXTSTART); p = CurrentLine;
  while ((J = *p++) != '\n') {
Characters within a string[41]
Characters not within any string[43]
add:
Add a character to the statement buffer[20]('J')
  }
  obstack_free(Csm_obstk, CurrentLine); CurrentLine = NoStr;
  NextInitialLine(TEXTSTART);
}
```

This macro is invoked in definition 37.

`JSQ`, `JDQ`, `JHOLL` and `HCount` are state variables. Continuations are dealt with differently depending on the context (inside or outside of a string). `LoadVariableFormat` guarantees that upon exiting the `while` loop the last line processed was not continued (recall that the statement buffer is terminated upon reaching the end of a statement that is also the end of a line). Thus it is appropriate to use `NextInitialLine` to skip comments and verify that there are no spurious continuation lines.

The label `add` is the common target for adding a character to the statement buffer regardless of its context. Jumps to `add` are used in lieu of more structured control flow because the decisions about context and when to actually add a character form a tree.

Characters within a string String context is indicated by a nonzero value of one of the three state variables `JSQ`, `JDQ` or `JHOLL`. Each of the three kinds of string requires a check for completion, which depends upon the kind of string:

Characters within a string[41] ≡

```
    if (JSQ && J == '\'') { JSQ = 0; goto add; }
    if (JDQ && J == '"') { JDQ = 0; goto add; }
    if (JSQ || JDQ || JHOLL) {
      if (J != '&') {
        if (JHOLL && --HCount == 0) JHOLL = 0;
        goto add;
      }
```
Advance to the continuing character of a string[42]
```
      continue;
    }
```

This macro is invoked in definition 40.

If an ampersand appears in a string context, then it indicates a continuation if and only if the remainder of the line is blank and the next non-comment line's first nonblank character is also an ampersand. If these conditions are not met, the ampersand is simply a character of the string:

Advance to the continuing character of a string[42] ≡

```
{ register char temp, *tempp = p;
  char *next = IsContinue(TEXTSTART);
  while ((temp = *tempp++) == ' ') ;
  if (temp != '\n' || !next) goto add;
```
Mark a line change[18](`next`)
```
  ExtractVariableLine(next); p = CurrentLine;
}
```

This macro is invoked in definition 41.

Characters not within any string If the routine is not in a string context, then its behavior depends strongly on the particular character:

Characters not within any string[43] ≡

```
    switch (J) {
    case '&':
```
Advance to the continuing character of a non-string[44]
```
      continue;
    case '!': while (*p != '\n') p++; continue;
    case '\'': JSQ = 1; HCount = -1; break;
    case '"': JDQ = 1; HCount = -1; break;
    case '(': case ',': case '/': case '*': HCount = 0; break;
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      if (HCount >= 0) HCount = HCount * 10 + J - '0'; break;
```

```
case 'H': case 'h': if (HCount > 0) JHOLL = 1; break;
case ' ': if (HCount == 0) break;
default: HCount = -1;
}
```

This macro is invoked in definition 40.

An exclamation point indicates a comment that terminates the line, and either of the string quotes indicates a shift to string context. A shift to string context due to the beginning of a Hollerith constant is harder to detect.

The state variable `HCount` is used to keep track of the possibility of a Hollerith constant, and its putative length: `HCount` negative indicates that no Hollerith constant is possible, a zero value indicates that it is possible but no length has been specified, and a positive value indicates that it is possible and (if present) has the specific length. Hollerith constants can only follow four characters, and blanks are not allowed within the count portion of a Hollerith constant in the variable format.

Continuation of the line is indicated by an ampersand, which might be followed by a comment:

Advance to the continuing character of a non-string[44] ≡

```
{ char *next = IsContinue(TEXTSTART);
  while ((J = *p++) == ' ') ;
  if (J != '\n') {
    if (J != '!') {
      POSITION e;
      e.line = Current.LineIndex; e.col = p - CurrentLine - Current.Offset;
      message(ERROR, "Only a comment can follow &", 0, &e);
    }
    while (*p++ != '\n') ;
  }
  if (!next) {
Add a character to the statement buffer[20](‘' '’)
    if (HCount > 0) HCount = -1;
    next = TEXTSTART;
  }
Mark a line change[18](‘next’)
  ExtractVariableLine(next); p = CurrentLine;
}
```

This macro is invoked in definition 43.

Outside of a string context, an ampersand *must* be followed by either the end of the line or a comment.

If the continuation line begins with an ampersand, the continuation may occur within any token. Otherwise a token may not be partially on one line and partially on the next. This exception is indicated by inserting a space which, in the variable format, terminates any non-string token.

3.4 Character Sequence Normalization

Upper- and lower-case letters are equivalent in FORTRAN except when they appear in string data. Also, blanks are insignificant outside of string data in FORTRAN 77 and the fixed-format input text of FORTRAN 90. Any attempt to fold characters and remove irrelevant blanks when assembling a statement leads to rather complex decision processes and code that is difficult to follow. It is easier to deal with these problems after the character sequence for a token has been recognized.

Several different kinds of transformations may be necessary to normalize character strings in different contexts, but all of these operations follow the same basic pattern:

Character Sequence Normalization[45](◇1) ≡

```
char *
#ifdef PROTO_OK
◇1(char *start, int length, ObstackP obstk, char *table)
#else
◇1(start, length, obstk, table)
char *start; int length; ObstackP obstk; char *table;
#endif
/* Normalize a string to an obstack
 *   On entry-
 *     start points to a string to be normalized
 *     length=length of the string to be normalized
 *     obstk points to the area in which the normalized string will be stored
 *     table points to the translation table
 *   On exit-
 *     ◇1 points to the normalized string
 ***/
```

This macro is invoked in definitions 48 and 49.

The routine uses each character of the string pointed to by its first argument as an index into the table pointed to by the fourth argument. Depending on the content of the indexed element, a value may be added to the Obstack pointed to by the third argument. (This pattern is similar to that of the "translate and test" instructions found on machines beginning in the 1960's.)

Character translation code[46] ≡

FORTRAN Character Conversion Table[47]
IMPLICIT Character Conversion Table[101]
Normalization for Fixed-Format Input[48]
Normalization for Variable-Format Input[49]

This macro is invoked in definition 104.

3.4.1 FORTRAN Character Conversion Table

One table can be used by most of the normalization operations needed for the FORTRAN scanner. It specifies that all lower-case letters are to be replaced by their upper-case equivalents, and that spaces are to be treated specially. All other characters should remain unchanged:

FORTRAN Character Conversion Table[47] ≡

```
static char F77Fold[] = {
   0 ,  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,
   8 ,  9 ,  10,  11,  12,  13,  14,  15,
   16,  17,  18,  19,  20,  21,  22,  23,
   24,  25,  26,  27,  28,  29,  30,  31,
   0 , '!', '"', '#', '$', '%', '&', '\'',       /* Skip spaces */
  '(', ')', '*', '+', ',', '-', '.', '/',
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', ':', ';', '<', '=', '>', '?',
  '@','a', 'b', 'c', 'd', 'e', 'f', 'g',       /* Change upper to lower */
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '[', '\\',']', '^', '_',
  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', 127
};
```

This macro is invoked in definition 46.

The zero entries in the table indicate special treatment. Although the null character indicates special treatment, that entry is irrelevant because the string to be normalized will never contain a null in this application. There is nothing in the interface to preclude appearance of null characters, and they might be present in other applications of character sequence translation. A table for such an application would have an appropriate entry.

3.4.2 Normalization for Fixed-Format Input

Spaces are ignored in FORTRAN 77 and fixed-format FORTRAN 90 input. The normalization routine therefore ignores a character whose table entry is zero:

Normalization for Fixed-Format Input[48] ≡

Character Sequence Normalization[45](`NormalizeFixed`)

```
{
  register char temp;
```

```
    while (length-- > 0) {
      if (temp = table[*start++]) obstack_1grow(obstk, temp);
    }
    obstack_1grow(obstk, '\0');
    return (char *)obstack_finish(obstk);
  }
```

This macro is invoked in definition 46.

3.4.3 Normalization for Variable-Format Input

Spaces are terminators in variable-format FORTRAN 90 input. The normalization routine therefore stops at a character whose table entry is zero:

Normalization for Variable-Format Input[49] ≡

```
  Character Sequence Normalization[45]('NormalizeVariable')
  {
    register char temp;

    while (length-- > 0 && (temp = table[*start++])) obstack_1grow(obstk, temp);
    obstack_1grow(obstk, '\0');
    return (char *)obstack_finish(obstk);
  }
```

This macro is invoked in definition 46.

4 Token Classification

The scanner must classify every token as an instance of some particular terminal symbol of the FORTRAN grammar. In some cases the classification is determined completely by the characters making up the token, but in other cases it is not. For example, the token "if" might be either an identifier or a keyword. The fact that blanks are ignored in the fixed input format means that "3 E1" might be a floating-point constant or it might be an integer followed by an identifier.

Additional context must be used when the characters of the token do not uniquely determine its classification. If no more than one of the possible classifications is allowable at every point in a parse, then the parser can provide sufficient context. Some cases that the parser cannot resolve can be resolved on the basis of whether the statement being processed is or is not an assignment. In a few cases, more specialized processing is needed to classify the token.

4.1 Parser Resolution of Token Classification

When the parsing context is sufficient to determine which of several classifications is appropriate, the scanner simply chooses one possible class for the token. If tokens of that class are not allowed in the given context, the parser will invoke the `Reparatur` routine. This routine then chooses another possible class. Effectively, the parser forces the scanner to step through the possible classifications until one is found that works in the current context or all have been exhausted.

Resolution of any classification problem is the task of the token processors. One token processor is nominated in the specification of the pattern defining the token. This token processor chooses one possible class, saves information defining a token of another class, and nominates a processor to be applied to that token. `Reparatur` uses the saved information to invoke that processor if the current token is unacceptable to the parser:

Parser Resolution of Token Classification[50] ≡

> *State describing the next possible classification*[51]
>
> `int CurrentClass = NORETURN;`
>
> *Deal With an Unacceptable Token*[15]

```
{
  if (CurrentClass != *klass) return 0;
  CurrentClass = NORETURN;
  *klass = NewClass;
  TokenEnd = NewEnd;
  Processor(TokenStart, TokenEnd - TokenStart, klass, intrinsic);
  return 1;
}
```

This macro is invoked in definition 104.

If the current token has no alternative, `Reparatur` terminates immediately with an indication that the parser should report an error. Otherwise it notes that the alternative has been used, establishes the entry conditions for a token processor, and invokes the processor nominated to handle this alternative.

Three values, are required to characterize an alternative classification:

State describing the next possible classification[51] ≡

```
int NewClass;   /* Code for the alternative classification */
char *NewEnd;   /* Pointer to the first character beyond the token */
void  (*Processor) ELI_ARG((char *, int, int *, int *));
```

This macro is invoked in definition 50.

When specifying an alternative classification, a token processor must note the presence of an alternative and establish these three values:

Define the next possible interpretation[52]($\diamond$3) $\equiv$

```
/* klass points to the classification being returned by this processor */
  { CurrentClass = *klass; NewClass = ◇1; NewEnd = ◇2; Processor = ◇3; }
```

This macro is invoked in definitions 64, 66, 70, 81, 85, 91, 94, 97, and 100.

Note that at the token processor has decided upon the classification it will return to the parser at the time it defines the alternative.

4.2 Assignment Statement Recognition

Only the left context of a token is available to the parser. Thus, for example, the parser cannot decide whether `if` is a keyword or an identifier when the character sequence "`if (`" occurs at the beginning of a statement. In this case the decision can only be made on the basis of characters following the matching "`)`" character. Most such decisions boil down to a simple rule: if the statement is an assignment then the token should be classified as an identifier; otherwise the token should be classified as a keyword.

Recognizing an assignment statement is tedious, but relatively straightforward:

Assignment Statement Recognition[53] $\equiv$

```
int
#ifdef PROTO_OK
IsAssignment(char *p)
#else
IsAssignment(p)
char *p;
#endif
/* Check for an assignment statement
 *   If the string pointed to by p is an assignment statement then on exit-
 *     IsAssignment=1
 *   Otherwise on exit-
 *     IsAssignment=0
 ***/
{
  register char J;
  char JSQ = 0, JDQ = 0, ISW = 0, JEQ = 0;
  int Level = 0, JHOLL = 0;
#if !Fortran77[1]
  char JCOLON = 0;
#endif

  if (!*p) return 0;
  while ((J = *p++) != '\n') {
```

```
    if (J == ' ') continue;
    if (JSQ) { if (J == '\'') JSQ = 0; continue; }
    if (JDQ) { if (J == '"') JDQ = 0; continue; }
Case analysis for assignment[54]
    if (ISW) {
Remember this point for a possible assignment test[56]('p - 1')
      return JEQ;
    }
  }
  return JEQ;
}
```

This macro is invoked in definition 104.

`JSQ` and `JDQ` indicate whether the current position is within a string (in which case the actual character is ignored). `JEQ` is nonzero only after an equals sign not enclosed by parentheses has been seen. `ISW` is nonzero only if the current character is the one following a right parenthesis that is itself not enclosed in parentheses, and no unparenthesized equals sign appears to the left of the current character.

The basic idea is that an assignment statement is characterized by an equals sign that is not contained in parentheses. Of the non-assignment statements, only the DO statement has an equals sign that is not contained in parentheses. But the DO statement also has a comma that is not contained in parentheses, while the assignment does not, so the two are easily distinguished.

If `ISW=1` after analyzing the current character, that character determines whether the statement is or is not an assignment: An equals sign means an assignment, and any other character means some other kind of statement. That distinction is expressed by `JEQ`. Note, however, that only a part of the statement has been examined. If the statement is a logical IF statement then the remainder constitutes a statement in its own right and classification of tokens within that statement will depend on whether it is an assignment. Thus it is necessary to remember the position of the current character (`p - 1`) as the point at which to begin a new assignment test if one becomes necessary.

4.2.1 Case analysis for assignment

Individual characters must be examined in order to maintain the state variables and recognize situations that allow the procedure to terminate before reaching the end of the statement:

Case analysis for assignment[54] ≡

```
#if !Fortran77[1]
    if (J != ':') JCOLON = 0;
#endif
    switch (J) {
    case '\'': JSQ = 1; JHOLL = 0; break;
    case '"': JDQ = 1; JHOLL = 0; break;
```

```
    case '0': case '1': case '2': case '3': case '4':
    case '5': case '6': case '7': case '8': case '9':
      if (JHOLL) JHOLL++; break;
    case 'h': case 'H': if (JHOLL > 1) return 0; JHOLL = 0; break;
#if !Fortran77[1]
    case ':': if (JCOLON) return 0; JCOLON = 1; JHOLL = 0; break;
    case ';': return JEQ;
#endif
    case '/': case '*': if (Level == 0 && !JEQ) return 0; JHOLL = 1; break;
    case ',': if (Level == 0) return 0; JHOLL = 1; break;
    case '(': Level++; ISW = 0; JHOLL = 1; continue;
    case ')': JHOLL = 0; if (--Level) break; ISW = !JEQ; continue;
    case '=': if (Level != 0 && !JEQ) return 0; JEQ = 1;
    default: JHOLL = 0;
    }
```

This macro is invoked in definition 53.

Hollerith constants cannot occur in assignments, so the recognition of a Hollerith constant results in immediate termination without the need to examine the constant itself. `JHOLL` controls the recognition of a Hollerith constant: It is set to `1` when a character that might precede a Hollerith constant is seen, and incremented for each digit occurring in a context where a Hollerith constant might be expected. `JHOLL` is set to `0` after seeing any character that could not precede a Hollerith constant or be part of its count.

FORTRAN 90's double colon cannot occur in assignments, so the recognition of a double colon results in immediate termination. `JCOLON` controls the recognition of a double colon: It is set to `1` when a colon is seen and set to `0` by any other character except a space (the case analysis code is not executed for spaces).

FORTRAN 90 also introduces semicolon as a statement terminator, so that character ends the analysis.

4.2.2 Check the Remainder of a Logical IF

A FORTRAN logical IF statement consists of a parenthesized logical expression followed by a statement. When the assignment test is applied to a logical IF statement, it decides that statement is not an assignment when it reaches the first non-blank character after the parenthesized logical expression. This character is the first character of the statement controlled by the logical expression, and is the point at which the assignment test must be re-applied.

In order to re-apply the assignment test, normal scanning must be interrupted when the scanner reaches the appropriate point in the input text. This can be guaranteed by using the null character as a "breakpoint": Save the first character of the statement controlled by the logical expression, replacing it in the input text with a null character. When the scanner recognizes the null character (which denotes "end of text"), it invokes the auxiliary scanner `auxNUL`. The default `auxNUL` procedure simply returns the location of the null character. By supplying a different version of `auxNUL`, however, this behavior can be changed:

Check the Remainder of a Logical IF[55] $\equiv$

Define an Auxiliary Scanner[12](‘auxNUL’)

```
{
  if (NewScanMark) {
    *start = NewScanMark; NewScanMark = '\0';
    Assignment = IsAssignment(start);
  }
  return start;
}
```

This macro is invoked in definition 104.

Note that this strategy could be extended to a variety of different kinds of breakpoint simply by making `auxNUL` do more complex testing to determine the kind of breakpoint it had reached.

Setting the breakpoint in this simple case is straightforward:

Remember this point for a possible assignment test[56](◇1) ≡

```
{ NewScanMark = *(◇1); *(◇1) = '\0'; }
```

This macro is invoked in definition 53.

5 Controlling the Scanner

The generated scanner will process the characters of the statement buffer, extracting sequences that constitute basic symbols, classifying them, arranging for their intrinsic attribute values to be computed, and then passing them to the parser. It operates on the statement buffer, a character string whose last character is a newline (`\n`). The statement buffer contains no comments, and no tab characters.

Because the scanner does not operate directly on the text in the buffer provided by the source module, the standard initialization operation must be overridden:

Initialize the scanner[57] ≡

Establish a Scan Pointer[9] `\`
```
{ extern void NewStmtBuffer(); NewStmtBuffer(); }
```

This macro is invoked in definition 103.

`NewStmtBuffer` must invoke the statement buffer construction operation and establish the values of `TokenEnd` and `StartLine` for the scanner. Because classification of identifiers and keywords depends upon whether the scanner is processing an assignment, `NewStmtBuffer` must also use `IsAssignment` to classify the first statement:

Create a statement buffer and prepare to scan it[58] ≡

```
void
```

```
NewStmtBuffer()
{ LoadStmtBuffer();
  TokenEnd = Stmt; StartLine = Stmt - 1;
  Assignment = IsAssignment(TokenEnd);
}
```

This macro is invoked in definition 104.

The statement buffer may contain an arbitrary number of statements. Those statements will be separated by semicolons, and the last may or may not be terminated by a semicolon. Any sequence of semicolons has the same effect as a single semicolon, and any sequence of semicolons followed by a newline has the same effect as a newline alone. Either a semicolon or a newline constitutes an end-of-statement token. These rules are embodied in the following specification:

End-of-statement marker[59] ≡

```
#if Fortran77[1]
xEOS:   $\n      [EndOfStmt]
#else
xEOS:   $\n|;(\040*;)*(\040*\n)?         [EndOfStmt]
#endif
```

This macro is invoked in definition 105.

It consists of a terminal name (`xEOS`), a regular expression (introduced by `$` and terminated by white space) that describes the allowable character sequences, and the name of a token processor (`EndOfStmt`) nominated to be invoked after the pattern is recognized.

`EndOfStmt` must decide whether the sequence ended with a newline (in which case it must arrange for the statement buffer to be refilled) or not (in which case it must merely classify the next statement in the buffer):

End-of-statement token processor[60] ≡

```
Define a Token Processor[13](‘EndOfStmt’)
{ if (start[length-1] == '\n') ResetScan = 1;
  else Assignment = IsAssignment(TokenEnd);
}
```

This macro is invoked in definition 104.

Recall than if `ResetScan` is `1` when the scanner is entered, the value of `TokenEnd` is invalid and must be re-established. That will cause the scanner initialization operation to be executed, invoking `NewStmtBuffer` as described above.

6 Identifiers and Keywords

The form of an identifier in FORTRAN 77 and in the FORTRAN 90 fixed format can be defined by the following specification:

Identifiers and Keywords[61] ≡

```
#if Fortran77[1]
xIdent: $[a-zA-Z](\040*[a-zA-Z0-9])*    [keycheck]
#else
xIdent: $[a-zA-Z](\040*[a-zA-Z0-9_])*   [keycheck]
#endif
```

This macro is invoked in definition 105.

Keywords also satisfy this definition, and keywords are not reserved. Thus any identifier or keyword will be classified as an `xIdent` by the scanner generated from this specification, and the distinction must be made by `keycheck`. Even when the symbol could be a keyword, that keyword might not be acceptable in the current context. If the parser rejects the keyword, the symbol must be re-classified as an identifier. Re-classification is also carried out by a token processor, and two such processors are needed for FORTRAN because there are different re-classification requirements for the two input formats. Finally, keyword recognition in the fixed format requires the ability to recognize some prefix of the character sequence scanned, because the fact that spaces are ignored means that the keyword can be run together with an identifier that follows it. Thus four distinct routines are needed to process identifiers and keywords:

Token processors for identifiers and keywords[62] ≡

Keyword Recognition in Fixed-Format Text[68]
Re-Classify a Fixed-Format Keyword as an Identifier[67]
Re-Classify a Variable-Format Keyword as an Identifier[65]
Distinguishing Identifiers From Keywords[63]

This macro is invoked in definition 104.

6.1 Distinguishing Identifiers From Keywords

A character sequence that satisfies the definition of an identifier or keyword must first be normalized by converting all letters to one case and, in the case of fixed-format input, removing all spaces. If the sequence occurs within an assignment statement, then it is always interpreted as an identifier. Otherwise it must be checked against the set of possible keywords, a process that is dependent on the input format being used:

Distinguishing Identifiers From Keywords[63] ≡

```
Define a Token Processor[13]('keycheck')
{ int k;

#if Fortran77[1]
  CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
#else
  if (FixedFormat)
```

```
    CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
  else {
    CsmStrPtr = NormalizeVariable(start, length, Csm_obstk, F77Fold);
    TokenEnd = start + strlen(CsmStrPtr);
  }
#endif

  if (Assignment) {
    int dummy = xIdent;
    mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
    return;
  }

#if !Fortran77[1]
Variable-Format Keyword Test[64]
#endif

Fixed-Format Keyword Test[66]
```

This macro is invoked in definition 62.

A FORTRAN 77 compiler will not contain a copy of the routine `NormalizeVariable`, and therefore the call must not be generated in that case. Similarly, there is no need to include the variable-format keyword test in a FORTRAN 77 compiler.

In the variable-format case, the normalization process determines the length of the sequence. Thus `TokenEnd` can be set immediately after normalization. The fixed-format case is more difficult, because spaces can occur within the sequence and the end is not known until after the keyword test has been completed.

6.2 Variable-Format Keyword Test

The keyword test is straightforward in the variable-format case. Keywords are pre-loaded into the string table, and the normalized string is known to contain exactly the characters of the identifier or keyword. Therefore `mkidn` can be used to look the string up and return its classification and intrinsic attribute value. If the classification is not that of an identifier, the sequence is reported as a keyword. Of course this classification is incorrect if the keyword is not acceptable in the current context, so the interpretation of the sequence as an identifier must be noted.

Variable-Format Keyword Test[64] ≡

```
  if (!FixedFormat) {
    mkidn(CsmStrPtr, TokenEnd - start, klass, intrinsic);
    if (*klass != xIdent)
Define the next possible interpretation[52](‘xIdent’,‘ TokenEnd’,‘ nobody’)
    return;
  }
```

This macro is invoked in definition 63.

If the parser rejects the keyword, the only action necessary is to re-classify the sequence as an identifier; the intrinsic attribute remains unchanged. But this re-classification has already been done by the time the token processor `nobody` is invoked, so that token processor does nothing.

Re-Classify a Variable-Format Keyword as an Identifier[65] ≡

```
Define a Token Processor[13](‘nobody’)
{ }
```

This macro is invoked in definition 62.

6.3 Fixed-Format Keyword Test

In fixed-format input, a keyword may be a prefix of the character sequence extracted. Thus it is not sufficient to simply use `mkidn` to look up the normalized character sequence in the string table. `Keyword` is the routine that performs this lookup, resulting in a table index that defines the keyword found (if no keyword is found, `Keyword` returns 0). If the index of the keyword is `k` then `KeyTable[k].keycode` is the keyword's classification, and `KeyTable[k].length` is its length.

Fixed-Format Keyword Test[66] ≡

```
  if ((k = Keyword(CsmStrPtr)) >= 0) {
    int n;

    obstack_free(Csm_obstk, CsmStrPtr);

    *klass = KeyTable[k].keycode;
Define the next possible interpretation[52](‘xIdent’,‘ start + length’,‘ mkfidn’)

    TokenEnd = start;
    for (n = 0; n < KeyTable[k].length; n++) while (*TokenEnd++ == ' ') ;
  } else {
    int dummy = xIdent;
    mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
    return;
  }
}
```

This macro is invoked in definition 63.

Once the keyword has been identified, the normalized string can be discarded by invoking `obstack_free`. The keyword's classification is obtained from the table, and if the parser rejects that keyword then the sequence must be presented as an identifier.

Because the keyword may be any prefix of the character sequence, `TokenEnd` must be reset by advancing over the non-blank characters of the keyword.

If the parser rejects the keyword then the scanner must recover the complete character sequence, normalize it, and use `mkidn` to obtain the corresponding intrinsic attribute value.

Re-Classify a Fixed-Format Keyword as an Identifier[67] ≡

```
Define a Token Processor[13](‘mkfidn’)
{ int dummy = *klass;
  CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
  mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
}
```

This macro is invoked in definition 62.

In this case the classification of the token is known to be “identifier”, so the value set by `mkidn` must be ignored. If the particular character sequence has not been seen previously, however, its classification must be set. This behavior is obtained through the use of `dummy`, which is used to communicate the classification to `mkidn` and receive the updated classification from `mkidn`.

6.4 Keyword Recognition in Fixed-Format Text

`Keyword` searches a table of normalized keywords for the longest element that is a prefix of a normalized string. This table is constructed from the FORTRAN grammar by a separate tool. Each element of the table is a structure with three components: the normalized keyword (`keychars`), the length of that keyword (`length`), and its classification (`keycode`). The elements are sorted alphabetically, so that a linear search from the end of the table will find the longest match.

Keyword Recognition in Fixed-Format Text[68] ≡

```
typedef struct {                /* Definition of a keyword */
  char *keychars;                  /* Character form */
  int keycode;                     /* Syntax code */
  int length;                      /* Length of the keyword string */
} Keywd;

#include "keywds.h"

/**/
int
#ifdef PROTO_OK
Keyword(char *c)
#else
Keyword(c)
```

```
char *c;
#endif
/* Get the classification code for a keyword
 *   On entry-
 *     c points to a normalized identifier string
 *   If c has a keyword prefix then on exit-
 *     Keyword=Syntax code of the keyword
 *   Otherwise on exit-
 *     Keyword=-1
 **/
{
  int i;

  for (i = MAXKWD; i >= 0; i--) {
    register char *p = c, *q =  KeyTable[i].keychars;
    register int different = 0;

    while (*p && *q && !different) different = *p++ - *q++;

    if (!different) {
      if (!*q) return i;
    } else if (different > 0 && p == c+1) return -1;
  }

  return -1;
}
```

This macro is invoked in definition 62.

File `keywds.h` contains an initialized declaration of array `KeyTable`, the generated keyword table. It also declares `MAXKWD`, the index of the last table element.

6.5 Keywords in I/O Statements

`ERR` is classified as an identifier in `READ (5,ERR)` and as a keyword in `READ (5,ERR=100)`. The classification of such sequences of letters in I/O statements is determined by the following character: If it is `=` then the sequence of letters is a keyword, otherwise the sequence of letters is an identifier. In effect, the `=` becomes a part of the sequence:

Keywords in I/O Statements[69] ≡

```
        $[a-zA-Z][a-zA-Z\040]*=          [mkiokw]
```

This macro is invoked in definition 105.

Note here that no classification is specified for the sequence (the specification has no name). Such sequences are given the distinguished classification `NORETURN` by the scanner, and unless this classification is changed their presence will not be reported to the parser. The character sequences that are valid keywords are pre-loaded into the identifier table, with their classification, so they can be recognized by an invocation of `mkidn`:

Make a Keyword that is Terminated by =[70] ≡

```
Define a Token Processor[13](‘mkiokw’)
{ if (!Assignment) {
    CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
    mkidn(CsmStrPtr, strlen(CsmStrPtr), klass, intrinsic);
    if (*klass != NORETURN) {
Define the next possible interpretation[52](
    ‘xIdent’,‘ start + length - 1’,‘ mkfidn’)
      return;
    }
  }
  TokenEnd = start + length - 1; *klass = xIdent;
  keycheck(start, length - 1, klass, intrinsic);
}
```

This macro is invoked in definition 104.

I/O statement keywords obviously cannot appear in assignment statements, and if `mkidn` does not change the `NORETURN` classification then the sequence is not a valid keyword. When the sequence is determined not to be a keyword, the `=` character is stripped off and `keycheck` invoked to complete the classification. Even when the sequence is a valid keyword, however, it may not be appearing in an appropriate context. Therefore the processor must prepare for the possibility that the parser will reject this classification.

7 Denotations

A denotation represents a constant value, and that value can be determined from the text of the basic symbol. FORTRAN provides denotations for integers, floating-point values, strings and operators. The strings can be delimited by quotes or specified by giving their length:

Denotations[71] ≡

Integer Denotations[73]
Floating-Point Denotations[78]
String Denotations[82]
Hollerith Denotations[84]
Operator Denotations[86]

This macro is invoked in definition 105.

Denotations are defined in this section to contain spaces, even though FORTRAN 90 variable-format input permits spaces only in strings. The reason is that the generated scanner must be able to handle either the fixed or the variable format.

Each denotation is represented internally by an intrinsic attribute, whose meaning depends on the particular denotation. A token processor is therefore nominated to compute the intrinsic attribute value for each denotation:

Token processors for denotations[72] ≡

> *Make an Integer Value*[77]
> *Make a Floating Point Value*[81]
> *Make a String Value*[83]
> *Make a Hollerith Value*[85]
> *Make an Operator Denotation*[87]

This macro is invoked in definition 104.

The nominated processor may also perform other duties, as noted in its description.

7.1 Integer Denotations

Integer denotations are described by sequences of digits. The digits may be decimal, binary, octal or hexadecimal:

Integer Denotations[73] ≡

```
xIcon:  $Dig[74]\040*(Op[75]|Efmt[76])? [mkfint]
#if !Fortran77[1]
xBcon:  $B('[01]+'|\"[01]+\")
xOcon:  $O('[0-7]+'|\"[0-7]+\")
xZcon:  $Z('[0-9a-fA-F]+'|\"[0-9a-fA-F]+\")
#endif
```

This macro is invoked in definition 71.

Decimal digits are useful in describing other denotations. To reduce the amount of space occupied by these descriptions, it is useful to define `Dig` as a shorthand notation for any sequence of digits and spaces, the first of which is a digit:

Dig[74] ≡

```
[0-9](\040*[0-9])*
```

This macro is invoked in definitions 73, 76, 78, 79, 80, 84, 88, 89, and 90.

An integer followed by a dot or the letter `e` (or `E`) can be falsely recognized as a floating-point number. It is not possible to distinguish these cases syntactically, since floating-point denotations and integer denotations are often acceptable in the same context. Therefore the scanner must look far ahead, recognizing the construct beginning with either the dot or the `e` for what it is, in order to decide that the digit sequence is really an integer. One such construct is a dot-delimited operator, abbreviated by `Op`, and the other is an E-format descriptor, abbreviated by `Efmt`:

Op[75] ≡

`\.\040*[a-zA-Z][a-zA-Z\040]*\.` This macro is invoked in definitions 73 and 86.

Efmt[76] ≡
`(E|e)\040*`*Dig*[74]`\040*\.\040*`*Dig*[74] This macro is invoked in definition 73.

An integer denotation is represented internally by an intrinsic attribute that gives the value of the integer:

Make an Integer Value[77] ≡

```
Define a Token Processor[13]('mkfint')
{
  *intrinsic = 0;
  while (length-- > 0) {
    register int v = *start - '0';

    if (v >= 0 && v < 10) *intrinsic = *intrinsic * 10 + v;
    else if (*start != ' ') { TokenEnd = start; return; }
    start++;
  }
#ifdef MONITOR
  while (TokenEnd[-1] == ' ') TokenEnd--;
#endif
}
```

This macro is invoked in definition 72.

The token processor `mkfint` also sets `TokenEnd` to point to the first character that is neither a digit nor a space. This character would be the dot or letter `e`, and hence the first character of the following token.

If monitoring is enabled, `TokenEnd` is backed up over any sequence of spaces following the integer, so that a lexical monitor won't try to lump these characters with the integer.

7.2 Floating-Point Denotations

Floating-point denotations are described by sequences of digits in conjunction with either a decimal point or an exponent:

Floating-Point Denotations[78] ≡

```
xRcon:  $Dig[74]Exp[79]('e|E')|Sig[80](Exp[79]('e|E'))?          [mkfloat]
xDcon:  $Dig[74]Exp[79]('d|D')|Sig[80](Exp[79]('d|D'))?          [mkfloat]
```

This macro is invoked in definition 71.

Double-precision values are indicated by the letter `d` (or `D`) as the exponent marker.

Exponents are described by the following shorthand:

Exp[79](◇1) ≡
  `(◇1)\040*(\+|\-)?\040*`*Dig*[74] This macro is invoked in definition 78.

A significand is a sequence of digits containing a decimal point. There may be digits before and/or after the point:

Sig[80] ≡
  `(`*Dig*[74]`\040*\.(\040*[0-9])*|\.\040*`*Dig*[74]`)` This macro is invoked in definition 78.

A floating-point denotation is represented internally by the index of a string in the character storage:

Make a Floating Point Value[81] ≡

```
Define a Token Processor[13](‘mkfloat’)
{ int dummy = xRcon;

  CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
  mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
  if (*start == '.') return;
  while (length-- > 0) {
    register int temp = *start;
    if (temp >= '0' && temp <= '9' || temp == ' ') start++;
  }
Define the next possible interpretation[52](‘xIcon’,‘ start’,‘ mkfint’)
}
```

This macro is invoked in definition 72.

The token processor `mkfloat` also establishes the initial digit string (if one exists) as a possible token if the floating-point denotation is unacceptable at this point in the parse.

7.3 String Denotations

String denotations are described by quoted sequences of characters. The apostrophe is the only quote allowed in FORTRAN 77, while either an apostrophe or a quotation mark is allowed in FORTRAN 90:

String Denotations[82] ≡

```
#if Fortran77[1]
xScon:  $'      (fstr)  [mkfstr]
#else
xScon:  $['\"]  (fstr)  [mkfstr]
#endif
```

This macro is invoked in definition 71.

The auxiliary scanner `fstr` extracts the body of the string from the statement, replacing each doubled internal quote by a single quote. It stores this body in the space provided by the character storage module. The token processor `mkfstr` then uses `mkidn` to obtain a unique index for the string:

Make a String Value[83] ≡

```
Define an Auxiliary Scanner[12](‘fstr’)
/* Additional postcondition-
 *   CsmStrPtr points to the transformed string
 ***/
{ register char temp, quote;

  quote = *start++;
  for (;;) {
    if ((temp = *start++) == '\n') {
      message(ERROR, "Closing quote missing", 0, &curpos);
      start--;
      break;
    }
    if (temp == quote) {
      if (*start != quote) break;
      start++;
    }
    obstack_1grow(Csm_obstk, temp);
  }
  obstack_1grow(Csm_obstk, '\0');
  CsmStrPtr = (char *)obstack_finish(Csm_obstk);

  return start;
}

Define a Token Processor[13](‘mkfstr’)
{ int dummy = xScon;

  mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
}
```

This macro is invoked in definition 72.

It would have been possible to combine `fstr` and `mkfstr` into a single routine, but this was not done because `fstr` by itself is useful for extracting the file name of an `INCLUDE` directive.

7.4 Hollerith Denotations

Hollerith denotations are represented by a sequence of digits followed by the letter `h` (or `H`) followed by a sequence of characters. The length of the character sequence following the letter `h` is the value of the sequence of digits interpreted as a decimal integer. Only the digit sequence and the letter `h` are described declaratively:

Hollerith Denotations[84] ≡

```
xHcon:  $Dig[74](H|h)   [mkholl]
```

This macro is invoked in definition 71.

A Hollerith denotation is represented internally by the index of a string in the character storage. The token processor evaluates the length of the string that should follow the letter `h` and then collects it. It also establishes the sequence of digits preceding the letter `h` as an integer, in case the Hollerith denotation is syntactically unacceptable:

Make a Hollerith Value[85] ≡

```
Define a Token Processor[13]('mkholl')
{ register char temp, *p = start;
  register int count = 0, digits = length - 1;
  int dummy = xScon;

  while (digits--) {
    register int v = *p++ - '0';
    if (v >= 0) count = count * 10 + v;
  }
  *intrinsic = count;

  p++;
  while (count > 0 && (temp = *p++) != '\n') {
    obstack_1grow(Csm_obstk, temp);
    count--;
  }
  obstack_1grow(Csm_obstk, '\0');
  CsmStrPtr = (char *)obstack_finish(Csm_obstk);

  if (count) {  /* Return the integer preceding the H */
    obstack_free(Csm_obstk, CsmStrPtr);
    *klass = xIcon; TokenEnd = start + length - 1;
    return;
  }

  mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
  *klass = xScon;
```

Define the next possible interpretation[52](
  ‘xIcon’,‘ start + length - 1’,‘ mkfint’)

```
  TokenEnd = p;
}
```

This macro is invoked in definition 72.

Collection of the string is terminated by either exhaustion of the count or arrival at the end of the available text (indicated by `\n`). Note that arrival at the end of the text is not an error, but simply indicates that the construct should not be regarded as a Hollerith denotation. Thus `mkholl` simply falls back and returns the integer count. The value of the intrinsic attribute has already been set (before using the count to extract characters), and the character string extracted is discarded.

7.5 Operator Denotations

Operator denotations are described by sequences of letters bounded by dots:

Operator Denotations[86] ≡

```
#if Fortran77[1]
        $Op[75]         [mkfopr]
#else
xDop:   $Op[75]         [mkfopr]
#endif
```

This macro is invoked in definition 71.

An operator denotation is represented internally by the index of its normalized string (including the bounding dots) in the character storage:

Make an Operator Denotation[87] ≡

```
Define a Token Processor[13](‘mkfopr’)
{
#if Fortran77[1]
  CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
#else
  if (FixedFormat)
    CsmStrPtr = NormalizeFixed(start, length, Csm_obstk, F77Fold);
  else {
    CsmStrPtr = NormalizeVariable(start, length, Csm_obstk, F77Fold);
    if (length != strlen(CsmStrPtr)) {
      message(ERROR,"Space within an operator",0,&curpos);
      obstack_free(Csm_obstk, CsmStrPtr);
```

```
      *intrinsic = 0;
      return;
    }
  }
#endif
  mkidn(CsmStrPtr, strlen(CsmStrPtr), klass, intrinsic);
}
```

This macro is invoked in definition 72.

In the FORTRAN 90 variable format, an operator denotation cannot contain spaces. Thus if the normalized version of the string is not the same length as the original then the original contained a space and an error must be reported. The normalized string is discarded in this case.

8 Special Problems

Format descriptors, concatenation operators, FORTRAN 90 array constructor brackets and `IMPLICIT` statements present scanning problems that do not fit into any of the previous categories.

8.1 Format Descriptors

Format descriptors constitute a sublanguage that should be handled with a different scanner. Eli does not currently have the capability for defining multiple scanners, however, so format descriptors that take the form of (say) identifiers must be recognized as identifiers.

Format descriptors containing dots or beginning with sequences of digits need to be recognized specially:

Format Descriptors[88] ≡

```
#if Fortran77[1]
xFcon:  $[IiFfDd]D.D[89]|[EeGg]D.D[89](Efw[90])? [mkidn]
#else
xFcon:  $[IiBbOoZzFfDd]D.D[89]|(E[NnSs]?|e[NnSs]?|G|g)D.D[89](Efw[90])? [mkidn]
#endif
xPcon:  $((\+|\-)\040*)?Dig[74](P|p)    [mkfmti]
xXcon:  $Dig[74](X|x)                   [mkfmti]
```

This macro is invoked in definition 105.

Here `D.D` is shorthand for two digit sequences separated by a dot and `Efw` is shorthand for an exponent field width specification:

D.D[89] ≡

```
\040*Dig[74]\.\040*Dig[74]
```

This macro is invoked in definition 88.

Efw[90] ≡
`\040*[Ee]\040*`*Dig*[74]This macro is invoked in definition 88.

An `xFcon` cannot be anything but a format descriptor, so `mkidn` is used to enter the character sequence into permanent character storage and set the intrinsic attribute to index that entry. Either of the other descriptors could be an integer followed by an identifier or keyword. In addition to entering the string into the character storage, we define `xIcon` as an alternate interpretation of the token should the parse fail given the initial one:

Make a Format Descriptor[91] ≡

```
Define a Token Processor[13](‘mkfmti’)
{ register char c = *start;
  mkidn(start, length, klass, intrinsic);
  if (c != '+' && c != '-')
Define the next possible interpretation[52](
    ‘xIcon’,‘ start + length - 1’,‘ mkfint’)
}
```

This macro is invoked in definition 104.

8.2 Concatenation Operator

FORTRAN defines the sequence "//" as the concatenation operator. Unfortunately this sequence cannot be recognized as two successive slashes, because the first slash could be a division operator in a general expression. The sequence "//" can also appear in a format, where it represents two successive slashes. Therefore the scanner must recognize both the concatenation operator and two successive slashes. Token processors to must be used to set up a possible backtrack and to accept only the first slash if the concatenation operator is not acceptable to the parser:

Concatenation Operator[92] ≡

```
$\/\040*\/    [mkconc]
$\/           [mkslsh]
```

This macro is invoked in definition 105.

Literals in the grammar do not have names, and therefore we have no names to use in the specifications that nominate token processors for those literals. One solution would be to replace the literal in the grammar by non-literal terminals (thus naming them), and using the non-literal terminals in the specification above. That solution would reduce the documentation value of the grammar, however. A preferable solution is to supply an additional specification that simply associates names with the literals:

Literal recognized when dealing with the concatenation operator[93] ≡

```
$\/     Slash
```

```
$\/\/   Concat
```

This macro is invoked in definition 106.

This name can then be used in the normal way by the token processors:

Token processors for the concatenation operator[94] ≡

Define a Token Processor[13](‘`mkslsh`’)
`{ *klass = Slash; }`

Define a Token Processor[13](‘`mkconc`’)
`{ *klass = Concat;`
Define the next possible interpretation[52](‘`Slash`’,‘ `start + 1`’,‘ `mkslsh`’) `}`

This macro is invoked in definition 104.

8.3 Array Constructor Brackets

FORTRAN 90 defines the sequences “`(/`” and “`/)`” as array constructor brackets. Unfortunately, these sequences can also appear in formats. Therefore the scanner must recognize both the brackets and their components, and associate token processors with them to split the array brackets when they are recognized in the context of a format:

Array Constructor Brackets[95] ≡

```
#if !Fortran77[1]
  $\(\040*\/    [mklabr]
  $\/\040*\)    [mkrabr]
  $\)           [mkrpar]
#endif
  $\(           [mklpar]
```

This macro is invoked in definition 105.

We need to associate names with the literals:

Literals recognized when dealing with array constructor brackets[96] ≡

```
#if !Fortran77[1]
$\(\/   LeftAcBracket
$\/\)   RightAcBracket
$\)     RightParen
#endif
$\(     LeftParen
```

This macro is invoked in definition 106.

These names can then be used by the token processors:

Token processors for array constructor brackets[97] ≡

```
Define a Token Processor[13](‘mklpar’)
{ *klass = LeftParen; }

#if !Fortran77[1]
Define a Token Processor[13](‘mkrpar’)
{ *klass = RightParen; }

Define a Token Processor[13](‘mklabr’)
{ *klass = LeftAcBracket;
Define the next possible interpretation[52](‘LeftParen’,‘ start + 1’,‘ mklpar’) }

Define a Token Processor[13](‘mkrabr’)
{ *klass = RightAcBracket;
Define the next possible interpretation[52](‘Slash’,‘ start + 1’,‘ mkslsh’) }
#endif
```

This macro is invoked in definition 104.

8.4 Letter Ranges for IMPLICIT Statements

The description of a type has been extended in FORTRAN 90 to allow a “kind” specification. That specification follows the type keyword, and could be any expression. Unfortunately, this leads to a requirement for a very long lookahead in order to classify a character sequence as the letter range of an `IMPLICIT` statement.

Consider the FORTRAN 90 `IMPLICIT` statement `IMPLICIT INTEGER (A-Z) (I-N)`. Here the character sequence `(A-Z)` is an expression defining the kind of integer values and the character sequence `(I-N)` is the letter range. The distinguishing property is that the letter range is followed by the end of the statement, whereas the expression is not. Clearly this distinction requires looking beyond a putative letter range to see whether the next character is a comma, semicolon or newline. Note that it is not possible to make the decision with any smaller context than the entire letter sequence plus the following character:

Letter Ranges for IMPLICIT Statements[98] ≡

```
xImpl:  $\(Range[99](,Range[99])*\)\040*(,|;|\n)        [mkimpl]
```

This macro is invoked in definition 105.

Range[99] ≡

```
\040*[a-zA-Z]\040*(-\040*[a-zA-Z]\040*)?
```

This macro is invoked in definition 98.

It is clear that most sequences classified by the generated scanner as `xImpl` will not, in fact, represent letter ranges. The token processor `mkimpl` must therefore make provision for reclassifying the first character of the sequence as a left parenthesis:

Letter Sequences in IMPLICIT Statements[100] ≡

```
Define a Token Processor[13]('mkimpl')
{ int dummy;

  CsmStrPtr = NormalizeFixed(start, length - 1, Csm_obstk, FoldIntrinsic);
  TokenEnd = start + length - 1;
  dummy = xIdent; mkidn(CsmStrPtr, strlen(CsmStrPtr), &dummy, intrinsic);
Define the next possible interpretation[52]('LeftParen',' start + 1',' mklpar')
}
```

This macro is invoked in definition 104.

Note that `TokenEnd` is set to begin the next scan with the comma, semicolon or newline that followed the letter range and that this character is not part of the string being normalized.

The intrinsic attribute established for the letter range should be the normalized sequence without the enclosing parentheses, and that is accomplished by using the following translation table:

IMPLICIT Character Conversion Table[101] ≡

```
static char FoldIntrinsic[] = {
   0 ,  1 ,  2 ,  3 ,  4 ,  5 ,  6 ,  7 ,
   8 ,  9 ,  10,  11,  12,  13,  14,  15,
   16,  17,  18,  19,  20,  21,  22,  23,
   24,  25,  26,  27,  28,  29,  30,  31,
   0 , '!', '"', '#', '$', '%', '&', '\'',       /* Skip spaces */
   0 ,  0 , '*', '+', ',', '-', '.', '/',
  '0', '1', '2', '3', '4', '5', '6', '7',
  '8', '9', ':', ';', '<', '=', '>', '?',
  '@','a', 'b', 'c', 'd', 'e', 'f', 'g',       /* Change upper to lower */
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '[', '\\',']', '^', '_',
  '`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
  'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
  'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
  'x', 'y', 'z', '{', '|', '}', '~', 127
};
```

This macro is invoked in definition 46.

Here, in addition to the entry for the space character, the entries for the two parentheses are zero. Thus these characters will be skipped when normalizing the sequence. Only the letters, dashes and internal commas remain.

9 Specification Files

The specifications appearing in this document are organized into files according to the language in which they are written. Those files are generated by Eli from the file that contains the document, without further action on the part of the user. This section describes the purpose of each of the generated files.

9.1 scan.clp

This file provides the information needed by the command line processor to generate an appropriate command line module.

scan.clp[102] ≡

The Command Line Processing Module[3]

This macro is attached to a product file.

9.2 scanops.h

This file is included by the scanner frame. It overrides the default definitions for the scanner initialization and coordinate-setting macros.

scanops.h[103] ≡

```
#include "eliproto.h"
```

Initialize the scanner[57]
Set Token Coordinates[21]

This macro is attached to a product file.

9.3 scan.c

This file contains the code implementing the structure clash resolution, the auxiliary scanners, and the token processors.

scan.c[104] ≡

```
static char RCSid[] =
  "$Id: Scan.fw,v 1.22 1998/07/07 20:40:43 waite Exp $";

#include <string.h>
```

```
#include "eliproto.h"

#if Fortran77[1]
#define FixedFormat 1
#endif

#if !Fortran77[1]
#include "Include.h"
#include "clp.h"
#include "CmdLineIncl.h"
#endif

Eli Library Modules Used[2]
The Generated Scanner Module[8]

#include "termcode.h"
#include "litcode.h"
#include "tabsize.h"

Character translation code[46]

#if !Fortran77[1]
extern void NextIncludedLine ELI_ARG((char *));
extern char *fstr ELI_ARG((char *start, int length));
#endif

Units of Text[16]

static int Assignment = 0;      /* Nonzero if the statement is an assignment */
static char NewScanMark = '\0'; /* Trigger for restarting the scanner */

Assignment Statement Recognition[53]
Check the Remainder of a Logical IF[55]

Parser Resolution of Token Classification[50]

Create a statement buffer and prepare to scan it[58]

Token processors for identifiers and keywords[62]
Token processors for denotations[72]
Make a Keyword that is Terminated by =[70]
Make a Format Descriptor[91]
Token processors for the concatenation operator[94]
Token processors for array constructor brackets[97]
```

Letter Sequences in IMPLICIT Statements[100]

End-of-statement token processor[60]

This macro is attached to a product file.

9.4 scan.gla

This file contains the declarative specifications of the character strings to be recognized in the input text.

scan.gla[105] ≡

Identifiers and Keywords[61]
Keywords in I/O Statements[69]

Denotations[71]

Format Descriptors[88]

Concatenation Operator[92]

Array Constructor Brackets[95]

Letter Ranges for IMPLICIT Statements[98]
End-of-statement marker[59]

This macro is attached to a product file.

9.5 scan.delit

This file describes literals that should not automatically be used as patterns for the scanner.

scan.delit[106] ≡

Literal recognized when dealing with the concatenation operator[93]
Literals recognized when dealing with array constructor brackets[96]

This macro is attached to a product file.