

# HUGO Language Manual and Report

This manual describes HUGO, a programming language developed at the Text Editing and Photocomposition Branch of the Canadian Government Printing Office. This language has been designed to aid in the full automation of text composition and typesetting. Unlike previous programming languages, its intended users are people whose primary expertise and experience is in the area of copy styling and printing, and who are only secondarily computer programmers. With its help it is hoped that both the ability of printers to use the new computer-related technology and the ability of computers to serve printers will be improved.

As HUGO is newly developed, there is still a lot of room for improvement in its design. Any helpful comments on

the language or its implementation will be welcomed by HUGO's implementers.

## HUGO Implementation Team:

G. Latona  
S. Lucas  
S. Wilmott

Text Editing and Photocomposition Branch  
Canadian Government Printing Office  
Room 3100  
45 Blvd. Sacré Couer  
Hull, P.Q., K1A 0S7

August 20, 1980

## Table of Contents

0 Introduction	1	4 Declarations	11
0.1 Summary of the Language	1	4.1 Variable Declarations	11
0.2 Background and Development	1	4.1.1 Globals	12
0.3 Intended Use of the Language	2	4.1.2 Locals	12
0.4 Outline of the Language Description	2	4.1.3 Lists	12
0.5 Notation	3	4.1.4 List Enquires	13
1 Basic Concepts	3	4.2 Functions	14
1.1 Strings	3	4.3 User-Defined Statements	14
1.1.1 Value Contexts and Precision	3	4.4 Type Declarations	14
1.1.2 Constants	3	4.4.1 Type Attributes in Variable Declarations	14
1.2 Names	4	4.4.2 Type Built-in Functions	14
1.2.1 Scope	4	5 Programs	15
1.2.2 Variables	4	5.1 The Program	16
1.2.3 Synonyms	5	5.2 Set-up Sections	16
1.3 Program Layout	5	5.2.1 Program Initialization and Termination	16
2 Expressions	5	5.2.2 Special Program Control	16
2.1 Computational Expressions	5	5.3 Segments	16
2.1.1 Order of Execution in Expressions	5	5.4 The Stop Statement	17
2.1.2 Elements of Computational Expressions	6	6 Characters	17
2.1.3 Computational Operations	6	6.1 Setting Characters	17
2.2 Boolean Expressions	6	6.1.1 Types of Characters	17
2.2.1 Boolean Operations	6	6.1.2 The Set Statement	18
2.2.2 Lexical Comparison Operators	7	6.1.3 User-codes	18
2.2.3 Numeric Comparison Operators	7	6.1.4 Case Shifting	18
2.3 Built-in Functions	7	6.2 Character Attributes	18
2.3.1 Computational Built-in Functions	8	6.2.1 Font and Typesize	18
2.3.2 Boolean Built-in Functions	8	6.2.2 Boldface and Italic Type	20
2.3.3 Named String Constants	8	6.2.3 Slanting and Repositioning Type	20
3 Statements	8	6.2.4 Small Capitals	20
3.1 Assignment and Add Statements	8	6.3 Quads	20
3.2 Conditional Statements	9	6.4 Character Enquiries	20
3.3 Repetitive Statements	9	6.4.1 Sizing Operators	21
3.4 Simple Input and Output	9	6.4.2 Character Functions	21
3.4.1 Input	9	7 Lines	21
3.4.2 Output	10	7.1 Basic Line Makeup	22
3.4.3 The Get Function	10	7.2 Line Justification	23
3.4.4 The Put Statement	10	7.3 Variable Space in Lines	23
3.5 Then Statements	10	7.3.1 Space Bands	23
		7.3.2 Space-Filling and Leading	23
		7.4 Explicit Line Breaking Control	24
		7.5 Hyphenation	24
		7.6 Indentation	24
		7.7 Line Enquiries	25

8 Paragraphs	25	10 Ruling and Underlining	37
8.1 Character, Line and Paragraph Makeup Attributes	25	10.1 Underlining	37
8.1.1 Default Attributes	25	10.1.1 Basic Underlining	37
8.1.2 Saving and Restoring Attributes	26	10.1.2 Side-lining	38
8.2 Paragraph Organization	26	10.2 Rule-Filling	38
8.2.1 Sub-Paragraphs	26	10.3 Figure Rules	39
8.2.2 Paragraph Breaking	26	10.4 Ruling in the Page Layout	39
8.2.3 Interline Spacing	27	10.5 Ruling in the Page Body	39
8.2.4 Paragraph Enquiries	27	10.6 Rule Enquiries	39
8.3 Types of Paragraphs	27	11 Text Input	41
8.3.1 Text Paragraphs	27	11.1 Automatic Text Editing	41
8.3.2 Carried Heads	28	11.2 The Start of the Text	42
8.3.3 Reference Heads	28	11.3 The End of the Text	42
8.4 Tables	28	11.4 Patterns	42
8.4.1 Table Definition	29	11.4.1 Pattern Expressions	42
8.4.2 Tabular Text	29	11.4.2 Group Patterns	43
8.4.3 Tabular Entry Alignment	29	11.4.3 Position Patterns	43
8.4.4 Tabular Enquiries	29	11.4.4 Scanning Patterns	43
8.5 Footnotes	29	11.4.5 Examining the Next Line	44
8.5.1 Defining Footnotes	30	11.4.6 List Patterns	51
8.5.2 Positioning Footnotes	30	11.4.7 Pattern Parameters	52
8.6 Sidenotes	30	11.5 The Rescan Statement	53
8.7 Explanatory Notes	31	12 General Enquiries	54
8.8 Division Lists	31	12.1 Statistics	54
9 Pages	31	12.2 Time and Date	55
9.1 Page Layouts	31	Appendix A - Implementation Dependent Features	56
9.1.1 The Page Dimensions	31	A.1 Special Conversion Functions	56
9.1.2 Positioning Text in a Page	32	A.2 Device-Dependent Character Setting	57
9.1.3 Sequence of Pages	32	A.3 Documents	57
9.2 The Page Body	32	A.4 Job Identification and Accounting	58
9.2.1 Entering and Leaving the Page Body	32	A.5 Cpu-Time	59
9.2.2 Column Justification	32	A.6 Error Handling	62
9.2.3 Space in the Page Body	33	A.7 Source Line Numbers	62
9.2.4 Galley-Form Page Bodies	34	Appendix B - Language Summary	62
9.2.5 Explanatory Note Bodies	34	B.1 Syntax	62
9.3 Page Makeup	34	B.2 List of Synonyms	63
9.3.1 Single Stream Makeup	34	B.3 Operator Priorities	63
9.3.2 Multiple Stream Makeup	34	B.4 Default Attributes	64
9.3.3 Page Column Positioning	35	Appendix C - Using the TEPB Implementation of HUGO	64
9.4 Page Indexing	35	C.1 Invoking the HUGO System	64
9.5 Page Makeup Enquiries	35	C.2 Compiler Directives	64
9.5.1 Enquiries About the Page	37	C.3 Run-Time Directives	64
9.5.2 Areas	37	C.4 Submitting Jobs Through ATS	64
9.5.3 Standard Page Sizes	37	C.5 Stand-Alone Bilingual Merge Program	64
		C.6 XPORT Format Files	64
		C.7 Other Utilities	64

C.8 Updating the Source Library . . . . .	58	D.3.2 Standard Fonts (fonts 1 to 19 and 26 to 29) . . . . .	64
Appendix D - Font and Character Access . . . . .	59	D.3.3 Pi font 20 . . . . .	64
D.1 Fonts . . . . .	59	D.3.4 Pi font 21 . . . . .	64
D.2 Input Character Sequences . . . . .	62	D.3.5 Pi font 22 . . . . .	64
D.2.1 Standard Input Sequences . . . . .	62	D.3.6 Pi font 23 . . . . .	64
D.2.2 "Pi" Input Sequences . . . . .	62	D.3.7 Greek Pi (font 24) . . . . .	64
D.2.3 Greek Characters . . . . .	62	D.3.8 Bedford Monowidth (font 25) . . . . .	64
D.2.4 Bedford Characters . . . . .	63	Appendix E - Sample Program . . . . .	65
D.2.5 Accessing Other Characters . . . . .	63	Index . . . . .	70
D.3 Internal Character Codes . . . . .	64		
D.3.1 Perma Utility (font 0) . . . . .	64		

## 0 Introduction

This report defines the HUGO programming language. All other descriptions and all implementations of the language should attempt to conform to this definition. Any deviation from this report should be considered an error. On the other hand, any ambiguity or incompleteness in this report can itself be considered an error.

HUGO is a name and not an acronym. It should be applied only to the language and the software which implements it. It should not be applied to any application of the language or to any input coding system used with the language.

It is intended that this report be supplemented by a User's Manual, (which this is not,) which will be better suited to those just learning the language or the casual user. And for convenience, there is a Quick Reference booklet which summarizes all the features of the language and the fonts and characters available.

This report corresponds to version 1.9 of the language.

### 0.1 Summary of the Language

A HUGO program describes to the HUGO system how a job is to be phototypeset.

A HUGO program is fed to a system program called the *compiler*. The output of the compiler is a machine readable description of the task to be performed. This description, together with the input text for the job is then fed to a program called the *interpreter*. The interpreter produces a magnetic tape which in turn will cause a phototypesetting machine to produce camera ready copy.

A HUGO program itself contains descriptions of input formats, of page layouts, of table layouts, of type characteristics and of special processing for a job.

### 0.2 Background and Development

The Text Editing and Processing Branch of the Canadian Government Printing Office has been developing and supporting text processing software since the late 1960's. Since Hansard, the verbatim transcript of the proceedings of the House of Commons, was first processed in 1971, the composition of most parliamentary papers has been computerized. Continual software development has been necessary to cope with the ever growing needs of the legislature and of other branches within the government. HUGO is the latest product of this development.

The software package currently used by the Printing Office, a text composition system called TPS, originally consisted of a mixture of general purpose utilities, such as the hyphenation and justification program, and of special purpose utilities, such as the various page formatting programs. Later, with the aid of progressively more powerful parameterization mechanisms, software was developed that would process a large variety of input and page formats. However, with greater power came greater complexity, for the software's users as well as for its maintainers. The user who needed the full power of the system was forced to develop skills much like those

of a computer programmer in order to be able to create the sequences of commands required by the programs.

HUGO is a new programming language, containing procedural and non-procedural elements. It is derived from existing programming languages and existing typesetting systems, and it is adapted to the text composition application. It has been designed to allow application-knowledgeable and oriented personnel to develop composition programs appropriate to their individual work. To this end, the language has been simplified so that no programming skills are needed beyond those required to parameterize the currently used "generalized" software. The language design emphasises:

- (a) a non-mathematical notation,
- (b) absence of storage allocation and data representation declarations,
- (c) a uniform, "structured" set of control facilities, and
- (d) a powerful method of describing input formats.

Design and development of HUGO commenced in February 1978 and by October 1978 a version of the system was working and in limited use. Since then use of the language has steadily increased, and the language has been enhanced to the state described in this manual. It is not expected that any further major enhancements will be needed, although there is room for many minor improvements. The number of fonts and characters available can be increased. And the predefined sequences used to input characters can be replaced by a system better suited to video-screen input devices.

The current implementation of HUGO runs on an IBM370/148 computer running under OS/VS1 and producing output for an APS-4 phototypesetter. The HUGO compiler and interpreter are written in the PL/I programming language. HUGO can be made to run on other computers and phototypesetters either by modifying the current implementation or by re-implementing it on another computer system. At the moment, however, there are no plans to distribute the current implementation of HUGO beyond the installation at which it has been developed or to transfer it to any other configuration of computers or phototypesetters.

### 0.3 Intended Use of the Language

HUGO makes no assumptions about the format of its input data. The input can consist of text files, with or without embedded commands, or fixed-field data, as is usually produced by programs written in more traditional languages such as COBOL or FORTRAN. The power of HUGO's text editing facilities and the great flexibility of its composition facilities allow HUGO programs to be written to process data intended for a wide variety of other systems.

This is not, however, the only or even the best use that can be made of HUGO's power. With the advent of the concept of generic text coding and textual data bases, it has been recognized that designing textual input conventions is an important and non-trivial task. HUGO's power is intended primarily to give the greatest freedom

in designing these new conventions and to make them easy to implement.

Because HUGO can be used to implement input conventions that are, in effect, text formatting "languages", each HUGO program should be able to process a number of different jobs. This is a more than adequate compensation for the fact that although HUGO programs are generally easier to read than those in most other text formatting languages, more effort is generally required to set up a HUGO program.

There is one area in which HUGO is very limited when compared to other text formatting systems. HUGO programs are intended to be run in a so-called "batch" environment: once a HUGO program and its data have been submitted to the computer, no further human intervention is allowed. This mode of working is the only one available in the environment in which HUGO was developed. And it is the most desirable way for many jobs which, although possibly quite complex, may be run over and over again without change to anything but the input data. HUGO programs can be written which allow for all but the most unexpected situations that can appear, and treat them correctly. However, there are many jobs, especially those which are quite complex yet short, for which a more interactive system may be of greater use.

HUGO is different from many other "batch" text composition systems in that it processes the text in one pass, and not in two or more "phases" each of which reprocess the text from the start. This means that even the most catastrophic error (by the user or computer) part way through a job will leave the text set before the error was encountered in a form that can be run on the phototypesetter.

#### 0.4 Outline of the Language Description

A HUGO program is primarily a means of describing a complex format for a phototypesetting job. However, by emphasizing that HUGO is really a computer programming language, this report attempts to provide a precise definition of what HUGO looks like and of what it does.

The report consists of short chapters, each of which describes some major feature of the language. The first five chapters describe those parts of the language which make it a computer programming language. The chapters from number six on describe the photocomposition aspects of the language. A number of appendices summarize various aspects of the language.

#### 0.5 Notation

The HUGO language is described by a combination of English narrative and modified Backus-Naur form

(BNF). BNF describes syntax by the use of production rules each of which gives the expansion of a syntactic construction whose name is enclosed in angle brackets (< and >). For example:

<sentence> → <noun> are <adjective>

means that a sentence is a noun followed by the symbol are, followed by an adjective. If this example were complete, production rules would also exist for noun and adjective.

Symbols in the HUGO language are represented in a distinct typeface from that of the text to distinguish them from both the narrative and the names of syntactic constructs.

BNF includes a notation for allowing a syntactic construct to produce alternate consequences:

<noun> → elephants | mice

It also allows a part of a production to be repeated an arbitrary number of times, including zero times:

<adjective> → { very } big

means that an adjective can be big, very big, very very big among others.

Finally BNF allows an optional construct:

<noun> → [ pink ] elephants

means that a noun can be either elephants or pink elephants.

The following rule illustrates a construct which is a consequence of the above definitions but deserves special mention. It is used in this report to describe expressions and similar grammatical entities.

<noun phrase> → <noun> |  
<noun phrase> or <noun>

means, of course, that a noun phrase is a sequence of one or more nouns separated by the symbol or. It is used rather than the { } notation to indicate, firstly, that the order of evaluation is from left to right, as all but the last noun in a sequence have to be reduced to a noun phrase before the last noun can be included in a noun phrase. Secondly, it indicates that if there are other definitions of a noun phrase, then these other forms can be combined with the one defined above. So noun phrases could be made up of not only ors and nouns.

Every BNF grammar has a starting symbol, the thing which the whole grammar describes. In the case of this report's grammar, the starting symbol is, of course, <HUGO program>.

# 1 Basic Concepts

## 1.1 Strings

The basic unit manipulated by a HUGO program is a string. A string is a sequence of graphic characters and may serve to represent either itself or a numeric value. A string may be used as a photoset title, as a page number, as a point size to indicate to the HUGO system the size of text required, or for a variety of other purposes. The exact interpretation of a string is dependent on its context.

Each string has a length, that is, the number of characters in the string. Within a string, any character can be referred to by its character number, which is its position in the string. For example the first character in any string is character number 1.

### 1.1.1 Value Contexts and Precision

A numeric value, a string which represents a number, will only be used with at most three decimal places of accuracy, though the value itself may contain more. In certain contexts, it may be used as a whole number. And in contexts where a numeric value is being interpreted as a measurement or point size, the value is assumed to represent a number of points, and only as much precision is used as is required by the typesetting hardware. When fewer decimal places than exists in a value are required, the extra places are just dropped.

HUGO does not treat strings and numbers as distinct data types except where such an interpretation is demanded by their context. In this report, wherever a reference is made to a number, what is meant is a string whose contents conform to the syntax of numeric constants.

### 1.1.2 Constants

Values in a HUGO program are provided by evaluating expressions. The simplest form of expression is the constant. The graphic values of strings used as numbers must conform to the syntax of numeric constants.

<constant> →

<quoted constant> |  
<numeric constant>

<quoted constant> →

' { <character> } ' |  
' { <character> } ' |  
" { <character> } " |  
< { <character> } >

<character> → any graphic character other than that which terminates the enclosing quoted constant

<numeric constant> →

[ + | - ] <digit> { <digit> }  
[ . { <digit> } ] |  
[ + | - ] . <digit> { <digit> }

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Note that the syntax of a character is given in a non-formal manner, as it is entirely dependent on the device used to input the HUGO program. Also note that a numeric constant may or may not contain a sign, and

may or may not contain a decimal point, but must contain at least one digit.

Examples of constants:

'the COW jumped over the MOON.'  
<en français>  
'0.5'  
0.5  
-123  
.1234 / only .123 will be used  
/ in calculations  
<George's string> / French quotes needed  
/ to enclose English  
'> / And vica-versa

Examples of unacceptable constants:

ABC / A non-numeric string must be  
/ enclosed in quotation marks.  
<ABC' / Quotes must be matched.  
7.3.4 / Unquoted numbers must  
/ be well formed.

## 1.2 Names

Identifiers are used to name user variables, layouts, carried heads and other objects in a HUGO program.

Normally the same name can be used for two things of different types, such as a layout and a carried head, without confusion. The situation is a bit more complex for variables, which are discussed later. It is safest not to give two things the same name.

<identifier> →

<letter> { <letter> | <digit> | - }

<letter> →

A	B	C	D	E	F	G	H
I	J	K	L	M	N	O	P
Q	R	S	T	U	V	W	X
Y	Z	À	Á	Ê	Ë	È	É
Ï	Ï	Ô	Ù	Û	Ç		

Note that identifiers must contain at least one letter. Both upper- and lower-case letters may be used, but the compiler treats them both as the same letter. Also, hyphens in identifiers are ignored by the compiler. So page-number is treated the same as Pagenumber. This equivalence applies only to identifiers and not to strings.

A HUGO program may use any equivalent representation of those names defined by the language as well as of those names defined in the program. In the grammar of this report a consistent representation is used for each name to improve readability.

Examples of identifiers:

ABC  
SUM  
Page1  
ma-différence

Examples of unacceptable identifiers:

lpage / Identifier must start with a letter.  
A,B / Identifiers must contain only  
/ letters, digits or hyphens.  
'ABC' / Identifiers are not strings.

### 1.2.1 Scope

Every name has scope. The scope of a name is the part of the program in which that name can be used.

Different types of objects may have the same name with no conflict. Different objects of the same type must have unique names. The only exception to this rule occurs in the case of local and global variables, as described in the chapter on Declarations.

### 1.2.2 Variables

Variables are used to save values away for later use in expressions. Their values may be changed during the running of the HUGO program.

```
<variable> →
  <function name> |
  <global variable> |
  <local variable> |
  <list variable>

<list name> → <identifier>
<function name> → <identifier>
<global variable> → <identifier>
<local variable> → <identifier>
```

The use, definition and scope of the different types of variables are defined elsewhere in this report.

### 1.2.3 Synonyms

The keywords in the HUGO language can have system-defined synonyms which may be freely used in their place. For example, the semicolon character is synonymous with the symbol then.

Synonyms serve three purposes. They can be shorthand notation for longer symbols as in the case of the semicolon above. They can be used to produce program statements more nearly conforming to natural language usage, as in the case of inch and inches. Or they can represent the components of the language in a form

more familiar to the user. So the language is not limited to the terms familiar to anglophone computer programmers.

The synonyms available in the language are listed in an appendix to this report.

### 1.3 Program Layout

A HUGO program exists as computer-readable text divided into a sequence of lines. The elements of the program may be placed on these lines in any manner so long as the order specified by the language's syntax is preserved. A further restriction is that each element of the language, that is each identifier or constant, must be coded within the confines of a single line.

Non-executable comments may be included in the program text by placing the slash character ("/") on any text line. The slash and anything following it on the line is ignored by the HUGO system and is therefore strictly commentary. Slash characters appearing in string constants represent themselves and constitute the only exception to the above rule.

To avoid ambiguity, space characters must be placed between any successive pair of program elements that have the syntax of identifiers. A numeric constant must also be separated from a preceding identifier by one or more spaces, but numeric constants need not be immediately followed by a space. Any number of spaces may be placed between any two successive program elements. Text lines need not contain any program element, they may be entirely blank or contain only commentary.

Instructions to the HUGO compiler, such as whether or not the compiler is to list its input, can be included in the compiler input. This is done by placing the appropriate directive to the compiler on a line by itself, preceded by a hyphen. A list of the available compiler directives is, of course, implementation dependent, and will be described in the documentation of HUGO's implementation.



## 2 Expressions

There are two types of expressions. Computational expressions, usually referred to as just "expressions", produce string or numeric values. Boolean expressions specify a condition to be tested and used as a control to some conditional program logic.

### 2.1 Computational Expressions

An expression is used to produce a value. It produces this value in a number of ways:

- (a) Fetching of constant or variable values,
- (b) Calculation of values by the use of arithmetic or other operations, and
- (c) Calculation of values by the invocation of user-defined functions.

The grammar shows how these ways of producing a value can be combined.

```

<expression> → <subexpression> |
               <expression> cat <subexpression>

<subexpression> → <term> |
                 <subexpression> of <term> |
                 <subexpression> start-at <term>

<term> → <subterm> |
        <term> plus <subterm> |
        <term> minus <subterm>

<subterm> → <factor> |
           <subterm> mul-by <factor> |
           <subterm> div-by <factor> |
           <subterm> modulo <factor>

<factor> → <subfactor> |
          <factor> index <subfactor>

<subfactor> →
  <variable> |
  <constant> |
  <function call> |
  <built-in function call> |
  <subscription> |
  <factor> percent |
  ( <expression> )

<function call> →
  <function name> |
  <function name>
    ( <expression> { , <expression> } )

<subscription> →
  <subfactor>
    ( <expression> [ , <expression> ] )

```

Further definitions of operators are provided elsewhere in this report. For example, there are operators to convert numbers from measurements in picas, inches and centimetres to the unit of measurement required for the parameterization of the phototypesetting process.

Examples of expressions:

```

/ Assume in the following that
/ A has a value of 'ABC'
/ I has a value of 3
/ J has a value of 1.7
/ and F is a function defined by the user.
i mul-by 4 / result value is 12
(j plus 2) div-by i / result value is 1.233

```

```

a cat lc(a) / result value is 'ABCabc'
a(1,2) / result value is 'AB'
A start-at 2 / result value is 'BC'
a cat J / result value is 'ABC1.7'
Time-of-day / result value changes with time
/ TIME-OF-DAY is a built-in function.
trim(*xy*z***) / result value is 'xy*z'
/ where 'trim' is defined in section

```

4.2

### 2.1.1 Order of Execution in Expressions

The order of execution of operations is determined by three factors:

- (a) The priority of an operation is given by its place in the syntax. Those appearing in subfactors are evaluated first, then those in factors, those in subterms, those in terms, those in subexpressions, and finally those in expressions.
- (b) Operations at the same priority level are evaluated left-to-right.
- (c) Parentheses can be used to override the above order of evaluation.

### 2.1.2 Elements of Computational Expressions

The meanings of function calls and of built-in function calls are defined elsewhere in this report.

A subscription produces a substring of the factor, starting at the character number specified by the first expression, and of a length specified by the second expression. If the second expression is omitted, a length of 1 is used.

A variable gives its current value as its result.

A constant has its own value.

### 2.1.3 Computational Operations

Operations in expressions are evaluated as follows.

- (a) cat: concatenates the two strings on either side together.
- (b) of: produces a string which is the second argument repeated the number of times indicated by the first argument.
- (c) start-at: produces the substring of the first argument starting at the character position specified in the second argument and ending at the end of the first argument.
- (d) plus: adds its two arguments together.
- (e) minus: subtracts its second argument from its first.
- (f) mul-by: multiplies its two arguments together.
- (g) div-by: divides its second argument into its first.
- (h) modulo: produces the remainder after dividing the second argument into the first a whole number of times.
- (i) index: gives the character number of the first location where the second argument is matched within the first argument.
- (j) percent: divides the argument by 100.

Those arguments of the above operators used to indicate positions or repetition counts or are used in arithmetic computations must, of course be numbers.

## 2.2 Boolean Expressions

A Boolean expression is syntactically much like a computational expression, but it operates by combining tests rather than string or arithmetic values. The order of execution is determined as for computational expressions.

A Boolean expression either succeeds or fails.

```
<Boolean expression> → <Boolean term> |
    <Boolean expression> or <Boolean term>
```

```
<Boolean term> → <Boolean factor> |
    <Boolean term> and <Boolean factor>
```

```
<Boolean factor> →
    <Boolean built-in function call> |
    ( <Boolean expression> )
```

Examples of Boolean expressions:

```
/ Assume the values for A, I and J
/ given in chapter 2.1.
a is 'ABC' / succeeds
i eq 03 / succeeds
i is 3 / succeeds
i is 03 / fails
    / because 3 is the same as .03
    / when compared as numbers,
    / but not when compared as strings.
j verify digit-string / succeeds
j verify digit-string / fails
    / because J has a dot in it.
i odd and i gt 10 / fails
    / because although i is odd,
    / it isn't greater than 10
```

Examples of unacceptable Boolean expressions:

```
i gt 0 and lt 5
    / should be: i gt 0 and i lt 5
a odd / because A isn't a number.
```

### 2.2.1 Boolean Operations

Boolean operations allow the results of multiple comparisons and tests to be combined, and are evaluated as follows:

- or: succeeds if either of its arguments succeeds.
- and: succeeds only if both of its arguments succeed.

### 2.2.2 Lexical Comparison Operators

The lexical comparison operators impose an arbitrary ordering on strings. This ordering can be used to uniquely determine for any pair of strings which is the "lesser" and which is the "greater". The ordering also specifies that two strings are equal only if they are of the same length and correspond on a character-by-character basis.

The ordering used is dependant on the collating sequence of the character set on the computer being used to run HUGO programs. However, most such sequences sort the digits in increasing numeric order, the upper-case letters in increasing alphabetical order, and also sort the lower-case letters in increasing alphabetical order.

```
<Boolean factor> →
    <expression> lex-lt <expression> |
    <expression> lex-le <expression> |
    <expression> lex-gt <expression> |
```

```
<expression> lex-ge <expression> |
<expression> is <expression> |
<expression> isnt <expression> |
<expression> verify <expression>
```

These operators compare their two arguments and return success or failure depending on their relative position in the lexical ordering scheme:

- lex-lt: returns success if the first string is "less than" the second;
- lex-le: returns success if the first string is "less than" the second or if they are equal;
- lex-gt: returns success if the first string is "greater than" the second;
- lex-ge: returns success if the first string is "greater than" the second, or if they are equal;
- is: returns success if the two strings are exactly equal; and
- isnt: returns success if the two strings are not equal.
- verify: succeeds only if all the characters in the first argument are contained in the second argument.

### 2.2.3 Numeric Comparison Operators

The numeric comparison operators treat their arguments as numbers and allow the usual arithmetic tests to be performed on them.

```
<Boolean factor> →
    <expression> eq <expression> |
    <expression> ne <expression> |
    <expression> lt <expression> |
    <expression> le <expression> |
    <expression> gt <expression> |
    <expression> ge <expression> |
    <expression> even |
    <expression> odd
```

- eq: succeeds if the two arguments have the same value.
- ne: succeeds if the two arguments have different values.
- lt: succeeds if the first argument is less than the second argument.
- le: succeeds if the first argument is either less than or equal in value to the second.
- gt: succeeds if the first argument is greater than the second.
- ge: succeeds if the first argument is either greater than or equal to the second.
- even: succeeds if the whole number part of the argument is an even number.
- odd: succeeds if the whole number part of the argument is an odd number.

## 2.3 Built-in Functions

HUGO provides a number of predefined functions which may be used in a program. These functions may be used so long as the user doesn't define a global variable or function name using the same identifier as the built-in function name. Each built-in function available is described in this report together with the feature of the language to which the built-in function is related. A built-in function may or may not have arguments.

The scope of any built-in function used within a program is the whole program.

### 2.3.1 Computational Built-in Functions

These built-in functions are associated primarily with the control functions of the language.

```
<built-in function call> →
length ( <expression> ) |
floor ( <expression> ) |
lc ( <expression> ) |
uc ( <expression> )
```

length has one argument, a string, and produces a number, the length of that string, as its result.

floor has one argument, a number, and returns that number with the fractional part deleted as its result.

lc has a string as its argument and returns that string with all its upper-case letters converted to lower-case.

uc has a string as its argument and returns that string with all its lower-case letters converted to upper-case.

### 2.3.2 Boolean Built-in Functions

Boolean built-in functions return conditions as results: they succeed or fail. They are used primarily to test the state of the HUGO system and of the user's program.

```
<Boolean built-in function call> →
not ( <Boolean expression> )
```

not has a condition, a Boolean expression, as its argument. not succeeds only if its argument fails.

### 2.3.3 Named String Constants

Some built-in functions serve as convenient representations of commonly used values which might otherwise be difficult to express.

```
<built-in function call> →
tab | bks |
char-string | digit-string | alpha-string |
lc-alpha-string | uc-alpha-string
```

The functions bks and tab return the command delimiter, the backspace character and the tab character respectively.

char-string returns a string containing all possible input characters. digit-string, alpha-string, lc-alpha-string and uc-alpha-string return a string containing all digits, all letters (both upper- and lower-case), all lower-case letters, and all upper-case letters respectively.

## 3 Statements

The purpose of HUGO programs is to *do* something. Expressions provide the objects with which, or to which, things are done, but statements tell the computer what to do.

The statements associated with major features of the language are described with those features. The statements defined in this chapter are those which perform general control functions.

```
<statement> →
  <assignment statement> |
  <add statement> |
  <conditional statement> |
  <repetitive statement> |
  <output statement> |
  <put statement> |
  <user statement> |
  <then statement> |
  <character format statement> |
  <paragraph start statement> |
  <paragraph format statement> |
  <table definition statement> |
  <page format statement> |
  <rescan statement> |
  <stop statement>
```

Examples of statements:

```
  / This loop deletes all leading
  / asterisks from the variable "string"
  / by setting up a counter "i" and with
  / a loop counting it up to
  / the first non-asterisk.
  / And then taking the end of the string
  / starting at where the counter points.
assign l to i
loop
exit-if i gt length(string)
  / If the string is all asterisks we don't
  / want to run off the end of it.
exit-if string(i) isnt '*'
  add l to i
end-loop
assign string start-at i to string
```

### 3.1 Assignment and Add Statements

An assignment statement is used to change the value of a variable to that given in an expression.

```
<assignment statement> →
  assign <expression> to <variable>
```

An add statement is used to increment the value of a variable by that given in an expression.

```
<add statement> →
  add <expression> to <variable>
```

### 3.2 Conditional Statements

Conditional statements provide alternative paths of execution through a HUGO program. They contain within them other statements, possibly including other conditional statements, which may or may not be executed depending on the success or failure of a given Boolean expression.

```
<conditional statement> →
  if <Boolean expression>
    { <statement> }
  { else-if <Boolean expression>
    { <statement> } }
  [ else
    { <statement> } ]
  end-if
```

The conditional statement consists of a sequence of Boolean expressions, each with a group of statements following it. The Boolean expressions are evaluated in order until one returns success. Then the following group of statements is executed. And then execution resumes following the end-if.

If no Boolean expression of the conditional statement returns success, then the group of statements following the else is executed, if it exists. If the else is absent, no action is taken. In either case execution resumes following the end-if.

### 3.3 Repetitive Statements

Repetitive statements allow a given group of statements to be executed repeatedly. For example, all the characters of a string may be examined for some condition by repeating a group of statements for each character.

```
<repetitive statement> →
  loop
    { <statement> |
      exit-if <Boolean expression> }
  end-loop
```

The components of a repetitive statement are executed repetitiously until a component exit-if is encountered with a Boolean expression that yields success.

### 3.4 Simple Input and Output

Input and output for a HUGO program are normally provided by the text editing and photocomposition features of the language. However, a simpler form of input and output is provided, closer to that of the usual procedural programming language. It may be used for dynamically parameterizing the HUGO program, for providing the user with messages as to the state of the program's progress, for creating a contents listing or index of the job, or for producing statistics which can be further processed by an accounting system.

Examples of input and output statements:

```
output time-of-day
  / prints the time of execution
  / on the line printer
put 'INDEX',index-word
  / puts the current value of
  / "index-word" on the system
  / file called "INDEX"
assign input to record
  / method of inputting from the
  / user's parameter file.
  / "input" is a built-in function
```

#### 3.4.1 Input

Two built-in functions are provided for input.

<built-in function call> → input

<Boolean built-in function call> → eof

input returns a string containing the characters of the next line of the parameter file for the job. All trailing blanks and carriage return characters are deleted from the resultant string.

eof returns a Boolean result. It succeeds if, and only if the last input executed found no more lines in the parameter file. In this case, the call to input would have returned a zero-length result, as if it had read a blank line.

### 3.4.2 Output

<output statement> → output <expression>

The expression is placed on a line on the message listing of the program.

### 3.4.3 The Get Function

<built-in function call> → get ( <expression> )

get has an argument indicating the name of the file from which a line of text is to be returned. The file names available are determined by the manner in which the HUGO program is invoked from its environment. The

lines returned by get do not have any trailing characters removed, as is the case with input.

When there are no more lines to be read from the specified file, the get function returns a zero-length string. Because blank lines in text files must otherwise have at least a blank character or a carriage return character on them, a zero-length line uniquely determines the end-of-file condition.

### 3.4.4 The Put Statement

<put statement> →  
put <expression> , <expression>

The first argument of the put statement specifies the name of the file on which the second argument is to be placed as a line of text. The expression is extended with blanks to satisfy the requirements of the file onto which it is put. The file names available are determined, as with the get function, externally to HUGO.

### 3.5 Then Statements

The then statement does absolutely nothing. It can be used to terminate or separate other statements to increase the readability of the program.

<then statement> → then

## 4 Declarations

Most types of names must be declared. That is, the HUGO system has to be told what they are. Immediately following are definitions of the declarations for variable and function names and for user-defined statements.

```
<declaration section> →
  <global declaration section> |
  <list declaration section> |
  <function declaration section> |
  <user declaration section>
```

### 4.1 Variable Declarations

Variable declarations are used to declare global, local and list variables.

#### 4.1.1 Globals

Global variables are used to save away values. They have the scope of the rest of the program following the variable declaration which contains them.

```
<global declaration section> →
  global <global variable declaration>
  { , <global variable declaration> }

<global variable declaration> →
  <global variable> [ <type> ] [ entry ]
  [ starts-as <expression> ] |
  <global variable> [ <type> ] [ entry ]
  literally <expression>
```

The expression following the starts-as or literally, if present, provides an initial value for the variable. This initial value is calculated and given to the variable only once at the start of program execution. No variable may be used in an expression unless it is first given a value by a starts-as, literally, an assignment statement or by being a parameter (of a function or define).

When declared with a literally specification, a variable may not have its value changed by an assignment or add statement or by a to-global pattern parameter.

The "type" and entry options in the global variable declaration are described in sections on type declaration and segments, respectively.

#### 4.1.2 Locals

Local variables are used to save away values. They have the scope of an individual program section following the variable declaration which contains them. Local variables constitute an exception to the rule that names that can appear in expressions must be unique within a program. The declaration of a local variable in a program section temporarily suspends the scope of any global variable or function name using the same identifier.

When a program section is entered, its local variables are created, and must be given new values before any use is made of them.

```
<local declaration group> →
  local <local variable declaration>
  { , <local variable declaration> }
```

```
<local variable declaration> →
  <local variable> [ <type> ] [ static ]
  [ starts-as <expression> ] |
  <local variable> [ <type> ]
  literally <expression>
```

The effect of the starts-as or literally specification is the same as with global variables, except that local variables are given their initial values each time they are created. As with global variables, local variables declared with the literally specification cannot have their values changed by an assignment or add statement.

The effect of the optional type declaration is described later in this chapter.

The effect of the optional static specification is to keep the associated local variable from being created anew every time the program section to which it belongs is entered. The local variable is created once, when the HUGO program is started. If it is declared with a starts-as to give it an initial value, the initial value is evaluated only once at the start of the program and only assigned to the local variable at that time. Thus, because static local variables keep their values between successive uses of a program section, they can be used like global variables to record the past history of a HUGO program. Unlike globals, however, their scope is the program section to which they belong and so may be used to improve the organization of a HUGO program.

Another important use of static local variables is in a user-defined function or statement or in an if-you-find section that contains a table definition statement. Because most of the tool entries of a table definition are usually used after the program section in which it is defined has completed execution, local variables can normally not be used to define these entries, as local variables normally cease to exist on exit from a program section. static local variables, however, continue to exist throughout the execution of the whole HUGO program and so may be freely used in table definitions.

#### 4.1.3 Lists

```
<list declaration section> →
  list <list name declaration>
  { , <list name declaration> }

<list name declaration> →
  <list name> [ <type> ] [ entry ]

<list variable> → <list name> ( <expression> )
```

A list is a group of elements, each of which can be used as a global variable. Any element of a list can be used, assigned to, and added to as if it were a global variable. A list variable is used to access an element of a list, and is specified by providing a list name together with an expression which is evaluated to provide a key which identifies the required element. Each unique key accesses a different element of the list.

A list initially has no elements. A new element within a list can be created by simply assigning a value to it. As with a global or local variable, an element of a list cannot be used or added to unless it has first been created by assigning a value to it.

As with global variables, the optional type and entry declarations are described elsewhere. Also as with globals, the scope of a list is the entire HUGO program following its declaration.

A special use of lists is described in the chapter on Text Input.

#### 4.1.4 List Enquires

To allow the more convenient manipulation of lists the following operators and built-in functions have been provided.

```
<factor> →
  <factor> th-elm <list name>
```

```
<Boolean factor> →
  <expression> is-in <list name> |
  <expression> isnt-in <list name>
```

```
<built-in function call> → list-size ( <list name> )
```

The list-size function returns the number of distinct elements in the list named by its argument.

The th-elm operator can be used to access the keys used to identify the elements of the list named in its second argument. The elements are arbitrarily numbered from one to the number of elements in the list, and the first argument of the th-elm operator is used to indicate by this number the key required. This operator is most useful for sequencing through all the elements of a list. Because creating new elements in a list may change the arbitrary numbering of the existing elements, care should be taken in the use of this operator. It is an error to attempt to access a key outside the range indicated for this operator.

The is-in Boolean operator succeeds if the first argument is the key of some already existing element of the list named in the second argument, and fails otherwise. The isnt-in operator succeeds/fails if is-in would have failed/succeeded.

Examples of variable declarations and their use:

```
/ These declarations are of the simple sort
/ most commonly found in HUGO programs.
/ with no type, ENTRY or STATIC parts.
Global page-number starts-as 1,
  counter / not given an initial value.
List L
Initially
Local element starts-as 1
/ Using the counter called ELEMENT,
/ this loop gives values to nine
/ elements of the list called L.
/ These values are not very useful, as
/ they are just their keys enclosed
/ in asterisks.
/ So, for example, the value output by
/ the OUTPUT statement at the end
/ of this example is '*4*'.
Loop
  Assign '*' cat element cat '*'
    to L(element)
  Add 1 to element
  Exit-if element ge 10
End-loop
Output L(4)
```

## 4.2 Functions

A user-defined function is a HUGO program segment which defines how a value is to be produced. It may or may not have values passed to it through a parameter list, and it must return a single value. This value can be accessed by a function call in an expression.

```
<function declaration section> →
  function <function name>
  [ <type> ] [ entry ] [ static ]
  [ starts-as <expression> ]
  [ with <parameter declaration>
    { , <parameter declaration> } ]
  { <local declaration group> }
  { <statement> }
```

```
<parameter declaration> →
  <local variable> [ <type> ] [ static ]
```

```
<function call> →
  <function name> |
  <function name>
  ( <expression> { , <expression> } )
```

The local variables in the parameter declarations of the function are called the function's parameters. The expressions in the function call are assigned, one at a time, to the function's parameters when the function call is executed. The number of expressions and of parameters must be the same. The parameter declarations constitute the parameters' declaration as local variables. The use of the static option in a parameter declaration means that, as with other local variables, the declared variable will retain its value even after execution of the function is terminated. This is especially useful when table definitions appear within the function. Note, however, that a new call to a function will destroy the old values of its parameters.

The function name has two uses. Within the function it is used as a local variable. The last value assigned to it within the function becomes the resulting value of the function. A function must return some value.

Outside of the function, in a scope extending from the end of the function declaration to the end of the program, the function name represents the function and may be used within function calls.

Note that the above definition prevents a function from calling itself and from calling another function which calls the first function.

The starts-as expression which optionally follows the declaration of the function name can be used to initialize the value of the function name at each use of the function, and may, if desired, provide the function's result value. If the static option is used with the function name, the local variable created to represent the function's result will behave like any other local variable. The effect of the type option on this local variable is described later in this chapter.

The entry option changes the scope of the name and is described together with segments.

#### 4 Declarations

Example of a function declaration section:

```

/ This function deletes all trailing
/ asterisks from its argument.
function trim with string
local i starts-as length(string)
loop
exit-if i le 0
/ i.e. if string is all asterisks.
exit-if string(i) ne '*'
assign i minus 1 to i
end-loop
assign string(1,i) to trim

```

#### 4.3 User-Defined Statements

User-defined statements allow the user to extend the HUGO language with statements of his own design. The user-defined statements are declared in a similar fashion to functions, except that they perform a sequence of actions without returning a result.

```

<user declaration section> →
define <user statement name>
[ entry ]
[ with <parameter declaration>
  { , <parameter declaration> } ]
{ <local declaration group> }
{ <statement> }

```

```

<user statement> →
<user statement name> |
<user statement name>
<expression> { , <expression> }

```

```

<user statement name> → <identifier>

```

The meaning of parameter declarations, and the parameter passing mechanism of user declarations is the same as for functions except that the expressions, if any, in the call (user statement) are not surrounded by an extra set of parentheses.

The entry option is defined in the section of the report on segments.

The scope of the user statement name declared is the whole program following the end of the user declaration section.

Example of a user declaration section:

```

define superior with text
save a
baseline -0.6 ex
setsize 60% ex
set text
restore a
/ superior 'superior' will set superior

```

#### 4.4 Type Declarations

Numbers in HUGO are just special cases of strings. There is no requirement to provide any special declarations for numbers, and their use is always implied by their context, such as when they are added together. However, a particular implementation of HUGO may very well use more than one internal representation for strings, depending on how they are used, in order to improve the speed of HUGO programs when they are run on a computer. The best representation will usually be chosen automatically, without any need for HUGO programs to provide the HUGO system with any data type declarations.

The average user of HUGO should not be too concerned with the speed of his programs, except that he should attempt to use good programming techniques, such as those grouped under the name "Structured Programming". But in programs which are expected to receive a lot of use, the HUGO user may wish to help the system by indicating the way in which variables are used. This is done by including type attributes in global, local, function, parameter and list declarations and by using the string and number built-in functions.

#### 4.4.1 Type Attributes in Variable Declarations

```

<type> → string | number

```

The string attribute indicates that the declared variable is used primarily in non-numeric contexts such as the argument of set, output, cat or is. The HUGO system will generally be required to do less work to process the variable in these cases. The variable can still be used in numeric contexts, but this may require more work on the part of the HUGO system.

The number attribute indicates that the declared variable is used primarily in numeric contexts such as arithmetic computations, repetition counts, substring arguments, and size specifications. A similar effect on the speed of execution results to that from using the string attribute. However, although numbers can be stored in strings, variables declared with the number attribute can only be used to store strings which conform to syntax of numeric constants. Furthermore, strings assigned to number variables may be stored as an equivalent numeric value. For example, leading, non-significant zeros may be removed from the value. And only three decimal places may be retained in the number.

There is one reason for using type attributes other than improving the execution speed of a HUGO program. This reason is that the use of a type attribute in a variable declaration may help a person reading the program to better understand the function of the variable, and so may serve a documentary purpose.

The use of a type attribute in a list variable declaration applies to the values of each of the elements of the list.

Example using type declarations:

```

/ This function expects a number between
/ 1 and 3999 as its argument and returns
/ the number in roman numerals.
/ It uses type declarations as it is
/ designed to be used by a large number
/ of HUGO programs and it is worthwhile
/ to have it run efficiently.
Function Roman-Numeral string with N number
Local p number starts-as 3.
power-of-10 number starts-as 100.
digit number / to hold each digit.
Assign
floor(N div 1000 modulo 10) of 'm'
to Roman-Numeral
/ Set up the thousands.
/ This loop sets up the hundreds,
/ tens and ones.
/ Note that 1 is i, 2 is ii, 3 is iii
/ 4 is iv,
/ 5 is v, 6 is vi, 7 is vii, 8 is viii
/ and 9 is ix.
Loop

```



```

Assign
  floor(N div-by power-of-10
    modulo 10)
  to digit
If digit lt 4
  Assign Roman-Numeral cat
    digit of 'ixc'(p) to Roman-Numeral
Else-if digit eq 4
  Assign Roman-Numeral cat
    'ixc'(p) cat 'vld'(p)
    to Roman-Numeral
Else-if digit lt 9
  Assign
    Roman-Numeral cat 'vld'(p) cat
    digit minus 5 of 'ixc'(p)
    to Roman-Numeral
Else / if digit eq 9
  Assign Roman-Numeral cat
    'ixc'(p) cat 'xcm'(p)
    to Roman-Numeral
End-if
Exit-if p le 1

```

```

Add -1 to p
Assign power-of-10 div-by 10
  to power-of-10
End-loop

```

#### 4.4.2 Type Built-in Functions

```

<built-in function call> →
  string ( <expression> ) |
  number ( <expression> )

```

A type built-in function is used to indicate to the HUGO system that its argument should be converted to the representation most suited to the given type. These functions are most useful when assigning a value to a variable whose type has not been given in its declaration. In this case some improvement in the execution speed of later uses of the variable may result, though not as much as if the variable had been declared with the appropriate type attribute, in which case the use of these built-in functions is not required.

## 5 Programs

### 5.1 The Program

A HUGO program is a complete specification of the process which will phototypeset a given form of input in a given format. The program consists of a sequence of program sections, each of which specifies some component of the overall process.

```
<HUGO program> →
  { <program section> | <segment> }
  end-hugo

<program section> →
  <set-up section> |
  <declaration section> |
  <if-you-find section> |
  <user-code section> |
  <first-layout section> |
  <layout section>
```

An example of a complete HUGO program is included in an appendix.

### 5.2 Set-up Sections

These sections are intended to be used for purposes related to typesetting other than typesetting itself.

```
<set-up section> →
  <initially section> |
  <finally section> |
  <at-start section> |
  <at-end section> |
  <at-job-start section> |
  <at-job-end section> |
  <at-page-start section> |
  <at-page-end section>
```

#### 5.2.1 Program Initialization and Termination

The initially section contains all the set-up statements that are to precede the execution of the first layout. The primary purpose of the initially section is to establish a set of default conditions for the job. More than one initially section can be coded, in which case each one is executed successively in the order in which it appears in the HUGO program.

The finally section is similar to the initially section except that it is executed after the last layout of the job has been completed. Note that this means that a finally section cannot be used to specify any part of the format of the last page of the photoset output. The primary purpose of the finally section is the gathering, recording and display of statistics and accounting information for the job.

```
<initially section> →
  initially
  { <local declaration group> }
  { <statement> }

<finally section> →
  finally
  { <local declaration group> }
  { <statement> }
```

### 5.2.2 Special Program Control

In addition to the requirements of an individual typesetting job, a HUGO program may have to satisfy the requirements of its users in terms of messages printed on the message listing, the recording of information for accounting purposes, and the imposition of local restrictions on the typesetting job. This may need to be done on a per-job or per-page basis.

To fulfill these requirements, HUGO program fragments can be "packaged" for later recall and inclusion into each HUGO program. These fragments or packages can be used together with the simple input and output facility, with the segment facility, and with the source library include feature provided by the HUGO compiler. Four sections are provided in HUGO specifically for this purpose:

```
<at-job-start section> →
  at-job-start
  { <local declaration group> }
  { <statement> }

<at-job-end section> →
  at-job-end
  { <local declaration group> }
  { <statement> }

<at-page-start section> →
  at-page-start
  { <local declaration group> }
  { <statement> }

<at-page-end section> →
  at-page-end
  { <local declaration group> }
  { <statement> }
```

The at-job-start sections are executed before any other part of the HUGO program, even before the initially sections. The at-job-end sections are executed after any other part of the HUGO program, including the finally sections. The at-page-start sections are executed immediately before and the at-page-end sections immediately after the execution of each layout in a job.

If more than one of the above kinds of section appears in a HUGO program, the sections are executed at the appropriate time in the order in which they appear.

### 5.3 Segments

```
<segment> →
  segment
  { <program section> }
  end-segment
```

A segment is a group of declarations and program sections that logically belong together. For example, a particular HUGO program may be written in which two user-defined statements use a global variable in common. The user-defined statements and the global variable can be defined together in a segment. Grouping the related elements together, and making the global variable inaccessible outside the segment will produce a program that is easier to read and to keep free of programming errors.

Global variables, lists, functions and user-defined statements declared within a segment will not be usable outside that segment unless they have the optional entry attribute included in their declaration. Objects declared in a segment without the entry attribute can be used freely within that segment, but only there.

The scope of global variables, lists, functions and user-defined statements declared within a segment but without the entry attribute is the rest of the segment following their declarations. The names of such objects declared without the entry attribute can be reused outside their segment in other global variables, lists, functions and user-defined statements. But these other declarations have a scope which excludes that of the variable within the segment so that no conflict results. The scope of global variables, lists, functions and user-defined

statements declared inside a segment and with the entry attribute is the same as if they were declared outside a segment.

It is an error to use the entry attribute in a declaration outside of a segment.

#### 5.4 The Stop Statement

<stop statement> → stop

The stop statement terminates the execution of the HUGO program without any further statements being executed, and without the current layout, if any, being completed. It may cause the current line or page of text to be lost and should only be used when the HUGO program encounters a severe error in its processing.

## 6 Characters

### 6.1 Setting Characters

It is in the typeset characters and the way they are represented on input that the HUGO system is most dependent on the input and output devices attached to the computer on which HUGO is running. Characters can be keyed either as single keystrokes or as sequences of keystrokes. Characters can also be generated by the HUGO program and typeset using the set statement.

#### 6.1.1 Types of Characters

A single keyed character of text will normally produce a single character of composed text. But due to the limitations of most keyboards and computer character sets, sequences of keyed characters may need to be interpreted as single entities. And a character or sequence of characters may represent an action other than setting a certain character: it may represent a spaceband, "tab" or backspace action or it may have a meaning defined in the HUGO program.

Whenever a character is being set by a set statement or from the input text the characters following it are examined to find the longest sequence of characters starting with the character being set and with a defined meaning. When found, this longest sequence is set as a single entity. A backspace character, when not part of a defined sequence, will cause the characters resulting from the input characters or sequences either side of the backspace to be set centered over each other. The width of the result will be that of the character or sequence preceding the backspace.

#### 6.1.2 The Set Statement

The set statement allows the explicit setting of text.

```
<character format statement> →
  set <expression>
```

The characters in the evaluated expression are set one by one, and for each the defined action is taken. The characters are set without further processing by any if-you-find. Any predefined input sequence of characters must be entirely within one set statement to be recognized as a single entity. Otherwise its components will be set as individual characters.

Any text not matched by an if-you-find will be automatically set by HUGO. Predefined input sequences will be found and processed according to their predefinitions so long as all component characters of the sequence are contiguous in the text without any if-you-finds being actioned between the elements.

Examples of set statements and predefined input sequences:

```
set 'A' / output will be A
set '`a' / output will be à
set 'a<' / output will be à
set 'é' / output will be é
set '<f' / output will be <
      / (predefined meaning)
set 'O<x' / output will be @
set 'a<b<c' / output will be b
```

### 6.1.3 User-codes

The user-code section defines the action associated with an individual character or with a specific sequence of input characters. These actions can be either redefinitions of the graphical representation of characters or sequences, or they can be input forms of various commands, especially the mode-change commands such as bold and italic. They can also, of course, be more complex combinations of these actions.

```
<user-code section> →
  user-code <expression> [ only ]
  { , <expression> [ only ] }
  { <local declaration group> }
  { <statement> }
```

The expressions are evaluated during the execution of the at-job-start sections of the HUGO program, but the statements are only executed whenever one of the defined characters or input sequences are encountered in a set statement. The expressions must evaluate to a string whose associated action is to be defined.

The user-code section is entered when an attempt is made to set a certain user-code-defined character or combination. It allows HUGO's predefined action for that character or combination to be overridden. The user-code is used only after the text editing facility has processed the input text through the if-you-finds.

If a user-code section specifies one or more expressions to be defined, then each such character or combination is given the same meaning. This is to allow more than one "spelling" of a code to be defined easily. When the only option is specified together with an expression, then that character or combination is not only given a new definition, but any previous definition of a longer character sequence which has the given character or sequence as its start loses its previous definition. So it can be specified that a user-code is to be recognized by HUGO even when part of a longer sequence.

Examples of user-code sections:

```
user-code ';'
  set '.'
user-code "A" : bks : "T"
  bold
  set '@'
  roman
user-code '--'
  set '-'
user-code < /*
  if italic roman else italic end-if
  / Paired slashes will put the
  / enclosed text into italic.
```

#### 6.1.4 Case Shifting

Under certain circumstances the case in which alphabetic characters are entered into the system may not be that in which they are to be typeset.

```
<character format statement> →
  shift-up |
  shift-down |
  no-shift
```

shift-up causes all alphabetic characters to be set in upper case until otherwise specified. shift-down causes them to be set in lower case. no-shift causes them to be set in the case in which they are entered.

Examples of case shifting:

```
shift-up
set 'abcABC1' / output will be ABCABC1
no-shift
set 'abcABC1' / output will be abcABC1
```

## 6.2 Character Attributes

Character attributes determine the size, shape and style of typeset characters. Those attributes available depend on the fonts currently available at a particular installation. However, HUGO will substitute an approximation for those attributes not available. If the typesize asked for is larger than any available, then the largest available will be substituted. If there is no italic font available in the selected family of typefaces, then a roman font will be substituted.

### 6.2.1 Font and Typesize

Font family and typesize can be selected in one command, or the typesize can be changed without affecting the current font.

```
<character format statement> →
times <expression> [ , <expression> ] |
modern <expression> [ , <expression> ] |
excelsior <expression> [ , <expression> ] |
helvetica <expression> [ , <expression> ] |
old-helvetica <expression> [ , <expression> ] |
perma <expression> [ , <expression> ] |
bedford <expression> [ , <expression> ] |
setsize <expression> [ , <expression> ] |
font <expression> , <expression>
[ , <expression> ]
```

The two expressions select a typesize for all but the font statement. When the second, optional, expression is omitted it is taken to be the same as the first. The first expression is the required setheight and the second, the setwidth.

The name of the statement selects the font in the case of times, modern, excelsior, helvetica, old-helvetica, perma and bedford. In these cases the roman medium member of the indicated family is selected. The setsize statement leaves the font unchanged. The font statement uses the first expression to select the font and the second and third expressions to select the typesize. The font numbers available depend on the phototypesetting equipment used.

Examples of change of font and setsize:

```
helvetica 12
set 'abcABC1' / output will be abcABC1
modern 1em
set 'abcABC1' / output will be abcABC1
excelsior 10 by 8
set 'abcABC1' / output will be abcABC1
times lex, 1em
set 'abcABC1' / output will be abcABC1
```

### 6.2.2 Boldface and Italic Type

Members of a font family can be selected with further character format statements.

```
<character format statement> →
roman |
italic |
bold |
medium
```

roman means "not italic", medium means "not bold", so the four combinations roman medium, italic medium, bold roman and bold italic are all meaningful. Note that if the current font is bold roman then the roman statement alone will effect no change and that the italic statement alone will change the font to the bold italic member of the current font family.

Other attributes that could be added to HUGO are, for example, ultra and condensed.

Examples of bold face and italic:

```
times 8 point
set 'abcABC1' / output will be abcABC1
bold
set 'abcABC1' / output will be abcABC1
italic
set 'abcABC1' / output will be abcABC1
medium
set 'abcABC1' / output will be abcABC1
/ change in font cancels any
/ bold face and italic
helvetica 8 pt
set 'abcABC1' / output will be abcABC1
```

### 6.2.3 Slanting and Repositioning Type

```
<character format statement> →
slant <expression> |
baseline <expression> |
fudge <expression>
```

The slant statement allows characters to be slanted somewhat in the manner of italic type. The expression is the amount of slant in degrees, positive angles slanting the character to the right, negative to the left.

The baseline statement vertically offsets characters from the zero baseline. The zero baseline is that which would be in effect if no baseline statement had been executed. The baseline statement always measures its offset from this zero baseline rather than the baseline currently in use. It can be used to set inferior and superior type when used in conjunction with the setsize statement. A positive value in the expression sets type below the zero baseline and a negative value sets type above the zero baseline.

The fudge statement horizontally offsets characters which are centred over previous characters in the text stream. It can be used to correct misplaced accents. The argument is the distance by which the overstriking character is displaced to the right of its normal position. This "normal position" is determined by the design of the characters supplied on the photosetter and can be further adjusted by tables internal to the HUGO implementation which are used to select the best accent positions for a specific installation.

Setting the slant, baseline or fudge to zero resets it to its normal value.

Examples of using slant, baseline and fudge:

```
slant 12
set 'abcABC1' / output will be abcABC1
slant 0
set 'abcABC1' / output will be abcABC1
set 'abc'
  baseline 20% ex
  set 'ABC'
  baseline 0
  set '1' / output will be abcABC1
set 'é' / output will be é
fudge -5% em
  set 'é' / output will be é
fudge 1 em
  set 'é' / output will be e
```

### 6.2.4 Small Capitals

```
<character format statement> →
  small-caps |
  normal-case
```

When the small-caps statement is encountered, HUGO starts setting all lower-case letters as capital letters of a smaller size. This form of text is useful in certain types of headings, and is rarely used in running text. The normal-case statement restores the normal mode of setting lower-case letters.

Example of small capitals:

```
times 10 pt small-caps set 'Small Capitals'
  / output will be SMALL CAPITALS
```

### 6.3 Quads

```
<character format statement> →
  quad <expression> |
  quad-to <expression>
```

The quad statement allows the user to set a fixed-size horizontal space, called a quad space, in a line of text. The expression specifies the size of the space. This size may be a negative number.

A quad space produced by the quad statement can be backspaced over a character or can have a character backspaced over it. Because the first item in a backspace combination determines the combination's width, a quad backspaced over a text character has no effect, whereas a character backspaced over a quad takes on the width of the quad space.

The quad-to statement creates a quad space of sufficient size to ensure that the next text set after it on the same line is at the distance from the left-hand end of the line specified by the given expression. Any position in the line can be specified, even if it has already had text set at that position. A quad-to prevents any spacebands preceding it on the same line from being expanded by justification. On the other hand, quad-out, quad-with and quad-with-rule statements preceding the quad-to are actioned to the extent that the quad-to places extra space in the text line.

quad-to can be used to help "decimal align" numbers in tables. The number should have a quad-out preceding it to position the part of the number before the decimal point adjacent to the decimal point. Then the part of the number preceding the decimal point should be set. A

quad-to should be executed to position the decimal point, following which the decimal point and the rest of the number should be set.

Quad spaces of both sorts may cause any upcoming text to be set to the left of the normal start of the line by their specifying appropriate negative values.

### 6.4 Character Enquiries

A number of built-in functions and operators are provided to allow the HUGO programmer to enquire as to various aspects of the current state of the system and to express sizes in a convenient form. Statements which parameterize the composition usually require sizes of some sort. Where sizes are required, they must be expressed in points. The various operators in section 6.4.1 provide the necessary conversions to express sizes in other units.

Some built-in functions are provided to ease access to various characters which may otherwise be difficult to express in a HUGO program.

#### 6.4.1 Sizing Operators

All expressions that provide a height, a width or some other size are considered by HUGO to be in points. Sizing operators allow the use of other units of measurement.

```
<subfactor> →
  <subfactor> ems |
  <subfactor> ens |
  <subfactor> thins |
  <subfactor> ex |
  <subfactor> picas |
  <subfactor> points |
  <subfactor> inches |
  <subfactor> cm |
  <subfactor> mm
```

```
<factor> →
  <factor> pp <subfactor>
```

These operators convert their subfactor argument into points from the unit of measurement indicated by the operator. The units of measurement are:

- (a) ems: the current setwidth,
- (b) ens: half the current setwidth,
- (c) thins: a quarter of the current setwidth,
- (d) ex: the current setheight,
- (e) picas: 12 points,
- (f) points: 1 point,
- (g) inches: 72.29 points,
- (h) cm: 28.46 points,
- (i) mm: 2.846 points.
- (j) pp: the first argument is picas and the second, points.

Examples of using sizing operators:

```
Width 20 picas
Times 9 on 10 / i.e points
Layout page1 8.5 inches, 11 inches
  / 8 1/2 by 11 inch page
Body 22 cm / deep
```

### 6.4.2 Character Functions

<built-in function call> →

baseline |  
font |  
fudge |  
slant |  
width-of ( <expression> )

<Boolean built-in function call> →

bedford |  
bold |  
excelsior |  
helvetica |  
italic |  
medium |  
modern |  
normal-case |  
old-helvetica |  
perma |  
roman |  
small-caps |  
times

The functions bedford, bold, excelsior, helvetica, italic, medium, modern, normal-case, old-helvetica, perma, roman, small-caps and times test the current typesetting environment. They succeed only if the character attribute to which they refer is currently in effect.

The functions baseline, fudge and slant have no arguments and return the currently specified baseline offset (in points), fudge value (in points) and character slant (in degrees) respectively.

The font function returns the number of the font currently in effect. The numbers returned are those used by the font statement. The bold and italic attributes may or may not affect the font number, depending on the phototypesetting equipment used.

The width-of function has one argument, a string of characters and predefined keying combinations, and returns the width in points that those characters will set in the current font and setsize.

Example of using character status functions:

```
if italic roman else italic end-if
/ As in example in chapter 6.1.3
```

## 7 Lines

### 7.1 Basic Line Makeup

Characters when set are made up into lines. A line is terminated either explicitly or implicitly by the HUGO program, or by overflowing the maximum width specified for lines. When a line overflow condition causes breaking, the system chooses where to make the break based on its judgment of suitable breaking points. If it breaks a word on a syllable boundary it will insert a hyphen. If it breaks on a spaceband it will delete that spaceband. If it can find no suitable place to break it will pack the line as tight as possible and break without hyphenating.

Line makeup can be made very simple, by just specifying a suitable line width and explicitly terminating lines where appropriate, or it may be quite elaborate, using the various options described in this chapter and the various forms of paragraph makeup in chapter 8.

```
<line format statement> →
  width <expression> |
  finish-line |
  turn-over
```

The width statement specifies the maximum line width on which characters can be set.

The finish-line statement explicitly terminates a line of text and justifies the line as if it were the last line of a paragraph. Many other statements and conditions also terminate text lines implicitly, as does line width overflow. Examples of such statements are the at statement, any paragraph start statement, and the end of a layout.

The turn-over statement explicitly terminates a line of text, but justifies the line as if it were an intermediate line of a paragraph.

### 7.2 Line Justification

Lines of text can be set in a number of ways. In printed work it is usual to set straight text justified over the line width and headings centred or flushed to the left margin.

```
<line format statement> →
  centre |
  flush-left |
  flush-right |
  force-justify |
  justify |
  justify-centre |
  justify-right
```

The methods of line setting indicated are:

- (a) centre: Centre the text on the line width.
- (b) flush-left: Place the text lined up on the left end of the line and space out the right side.
- (c) flush-right: Place the text lined up on the right end of the line and space out the left side.
- (d) force-justify: Justify the text on the full line width by expanding the spacebands and fillers in the line.
- (e) justify: Justify all lines of a sub-paragraph but the last. Flush-left the last line. See chapter 8 for the definition of sub-paragraphs.
- (f) justify-centre: Justify all lines of a sub-paragraph but the last. Centre the last line.

- (g) justify-right: Justify all lines of a sub-paragraph but the last. Flush-right the last line.

Examples of paragraph justification:

This paragraph is set flush-right on all its lines. The normal paragraphs in this manual are set using the default of justify. The chapter headings are centred. And the examples and syntax rules are set using flush-left.

### 7.3 Variable Space in Lines

For a line to be justified, there must be places where extra space or text can be inserted.

#### 7.3.1 Space Bands

Space bands can be introduced into the text either by setting the space character or by use of the explicit spaceband and vari-space statement.

```
<line format statement> →
  spaceband |
  vari-space |
  sb-ratio <expression> |
  sb-step <expression>
```

A spaceband has some minimum width. Justification may increase this width by equally distributing all the extra space in a line amongst the spacebands. The sb-ratio allows the user to change the minimum size of a spaceband. Its expression argument specifies the spaceband size as a proportion of the setwidth when the spaceband is set. It is usually used in flush-left text where the normal spaceband is too narrow or with the Bedford monowidth font where it is best to have the spaceband the same width as a character.

The sb-step allows the user to change the increment of space that is used for justification. The extra space added to a spaceband will always be an exact multiple of the size specified by its expression argument. sb-step can be used when it is desirable to justify text set using a monowidth font, in which case it is used to ensure that any justification space added is in a multiple of the character width, so simulating the same operation on a typewriter or a computer's printer.

It should be noted that a spaceband is explicitly not a character, but rather the space between characters. It establishes a condition by which space is placed between characters. This is why a spaceband does not create space when it falls on an output line boundary; it is not "between" any thing. And it is also why more than one spaceband in a particular position will act exactly as one spaceband; the condition exists whether it is specified once or many times.

A vari-space is similar to a spaceband except that it cannot be used as a breaking point.

#### 7.3.2 Space-Filling and Leadering

Space and leader-filling can be used to provide line justification with a point at which all extra space is to be placed. If a space or leader-fill point is placed in a line the spacebands are ignored by line justification. The minimum space occupied by a fill point is zero.



```

<line format statement> →
quad-out |
quad-with <expression> [ , <expression> ] |
flush-left-with <expression>
        [ , <expression> ] |
justify-with <expression>
        [ , <expression> ]

```

quad-out provides a place at which all remaining space may be placed.

quad-with provides text and a fill-point at which that text can be inserted. The first expression must be a character or a predefined keying combination other than one defined by a user-code section. This character is repeated sufficient times to fill the fill-point. It may be used zero times. The second expression, if it appears, specifies additional space to be placed before each occurrence of the leader character, and is used to space out the leaders. Any space that is left over after leading is placed in front of the leading string. Using a space character as a leader has the same effect as a quad-out.

flush-left-with and justify-with allow leading to be done at the end of a paragraph without any special codes being inserted in the text. These statements place a quad-with in front of each explicit finish-line or other condition which causes a line termination to occur except for text overflowing a line. The flush-left-with causes all other lines to be set flush-left. The justify-with statement causes all other lines to be justified.

A particularly useful place for these last two statements is in a tabular format, where one or more columns can be leader-filled. To aid in the convenient setting of tables, blank tabular columns of text that are under the control of one of these two statements will be filled with leaders in the described manner, but only if some previous column of the tabular chunk has some text in it. This means that the user can leader-across blank entries of a table and yet have leading blank entries in a line left blank.

This, right here is an example of quad-out.

And this is an example of quad-with '`<`', 1 thin at the end of a paragraph (which is the same as using justify-with at the start of the paragraph) . . . . .

#### 7.4 Explicit Line Breaking Control

These statements allow explicit control of where breaks can be made in set text lines. They do not force line breaking but control the determination of where a break is made when line width overflow requires it.

```

<line format statement> →
break |
no-break |
break-here |
discretionary <expression>

```

no-break specifies that the following text is not to be broken at any place except where the text overfills the line width. This allows phrases with spacebands and hyphens in them to be kept on a single line. Normal determination of valid breaks is resumed by break, which is, of course, the default condition. The basic meaning of no-break is "don't break at all", but to make it more useful HUGO will not overflow the line width.

The user can produce lines of arbitrary size by use of the width statement, and force extra text into a line by use of the sb-ratio and setwidth statements.

break-here indicates a valid break in a line. It can be used to allow the HUGO system to place breaks where its normal rules would not allow.

discretionary indicates a valid place for line breaking in the same manner as break-here. In addition, it provides a character to be placed at the end of the line should this place be used as a break. It is normally used to explicitly indicate valid hyphenation points in words.

Example using discretionary:

```

Set «HU»
Discretionary '.'
Set «GO»
/ will cause the set word HUGO to
/ be hyphenated as HU-GO if so
/ needed to break a line.

```

#### 7.5 Hyphenation

Hyphenation provides an aesthetically pleasing method of breaking text lines. Considerable control can be maintained over hyphenation.

```

<line format statement> →
hyph |
nohyph |
english |
french |
hyph-margins <expression>
        [ , <expression> ] |
ladder <expression> |
hyphen <expression> |
hyph-fill <expression> |
min-word <expression> |
stick-hyphens |
no-stick-hyphens
(a) hyph: enables hyphenation,
(b) nohyph: disables hyphenation,
(c) english: enables hyphenation, and selects the english hyphenation algorithm,
(d) french: enables hyphenation, and selects the french hyphenation algorithm,
(e) hyph-margins: specifies the minimum number of letters to be left at the left and right-hand ends of a word, respectively, when it is hyphenated,
(f) ladder: specifies the maximum number of successive lines that can be hyphenated,
(g) hyphen: specifies the character to be used as the hyphen,
(h) hyph-fill: specifies a fraction, in the range from zero to one, and indicates that lines which can be filled to that proportion of their line width without hyphenating the last word are not to have their last word hyphenated,
(i) min-word: specifies that words shorter than the length given in the expression are not to be subject to hyphenation,
(j) stick-hyphens: specifies that hyphenated words cannot be split across two page columns or two pages, and
(k) no-stick-hyphens: specifies that the line containing the first part of a hyphenated word can be left at the bottom of a column.

```

### 7.6 Indentation

The left and right text margins can be adjusted by the use of quad spaces and by changing the line width. Further, and more automatic control is provided by the indentation statements.

```
<line format statement> →
  left-indent <indentation margins> |
  right-indent <indentation margins>
```

```
<indentation margins> →
  <expression> [ for <expression> ]
  [ , <expression> ] |
  <expression> for-all
```

The left-indent and right-indent statements reduce the line width by the amounts specified in the indentation margins on the left and right-hand sides respectively.

The first expression in the indentation margins specifies the indentation for the first line of a sub-paragraph. If the for option is specified then the expression following it indicates for how many lines this indentation is to be used. If the for-all option is specified then this indentation is to be used for all the lines of the sub-paragraph. In the absence of either option it is used for only one line.

The expression following the comma, if present, specifies the amount of indentation for those lines of the sub-paragraph for which the first indent is not used. If it is absent, these lines have an indentation of zero.

As an example this paragraph is indented to the left and right with the following commands:

```
Left-indent 2 picas for-all
Right-indent 2 picas, 3 picas
```

### 7.7 Line Enquiries

```
<built-in function call> →
  hyphen |
  hyph-fill |
  left-for |
```

```
left-indent ( <expression> ) |
right-indent ( <expression> ) |
sb-ratio |
sb-step |
width
```

```
<Boolean built-in function call> →
```

```
break |
english |
french |
hyph
```

- (a) hyphen: Returns the current hyphen character.
- (b) hyph-fill: Returns the argument of the currently effective hyph-fill statement (as a fraction of 1).
- (c) left-for: Returns the argument of the for option of the currently effective left-indent.
- (d) left-indent: Returns, in points, the left indentation to be applied to the line indicated by its argument of a sub-paragraph.
- (e) right-for: Returns the argument of the for option of the currently effective right-indent.
- (f) right-indent: Returns in points, the right indentation to be applied to the line indicated by its argument of a sub-paragraph.
- (g) sb-ratio: Returns the current sb-ratio statement argument.
- (h) sb-step: Returns the current sb-step statement argument.
- (i) width: Returns the currently specified line width.
- (j) break: Succeeds if the break statement is currently in effect.
- (k) english: Succeeds if English hyphenation is currently in effect.
- (l) french: Succeeds if French hyphenation is currently in effect.
- (m) hyph: Succeeds if either form of hyphenation is currently in effect.

## 8 Paragraphs

In the same way that characters are formed into words, which are then used to fill lines, so lines are formed into paragraphs, which are then used to fill columns on a page. Control can be maintained over whether or not paragraphs are broken across column or page boundaries. Paragraph-starts also reset various conditions and affect the way in which space is placed between lines.

Each type of paragraph is started by a paragraph-start statement. The leave option (section 8.1.1) on a paragraph-start statement controls whether or not the character and line make-up attributes are to be reset to their default values. The chunk, division and sub-paragraph statements control how paragraphs are to be sub-divided.

Paragraph-start statements may appear anywhere in the HUGO program, but they are designed to be used in the page body.

Each type of paragraph has restrictions as to whether or not it may appear at the top or bottom of a page column. The HUGO system will place text in the page body so as to satisfy these restrictions. Even paragraph types such as continued-head and reference-head paragraphs, which are not set in the page body at the place in which they first appear must satisfy these restrictions. In these cases the restriction is satisfied by HUGO moving the first appearance of these paragraphs to the same column as the text which immediately follows them.

### 8.1 Character, Line and Paragraph Makeup Attributes

The statements which establish character, line and paragraph attributes, which can be saved away and restored at any time, are:

baseline	justify-with
bedford	ladder
block	leading
bold	left-indent
bottom-align	medium
bottom-previous	min-para
break	min-word
centre	modern
centre-align	no-break
drop-align	no-escape-char
english	nohyph
escape-char	no-para-space
excelsior	normal-case
flush-left	no-shift
flush-left-with	no-stick-hyphens
flush-right	no-suffix-space
font	old-helvetica
force-justify	on
french	para-space
fudge	perma
helvetica	right-indent
hyph	roman
hyphen	sb-ratio
hyph-fill	sb-step
hyph-margins	setsize
italic	shift-down
justify	shift-up
justify-centre	slant
justify-right	small-caps

stick-hyphens  
suffix-space  
times  
top-align

top-previous  
unblock  
widow  
width

#### 8.1.1 Default Attributes

At the end of executing the initially sections the HUGO system saves away the character and line makeup attributes. Unless the leave option is coded on a paragraph-start statement those defaults will be restored at the beginning of all paragraphs. This allows the user to specify the default attributes for a job and means that for each paragraph only the way in which it differs from the defaults needs to be specified. These defaults are also restored at the start of each layout section and at the start of the finally sections.

In the absence of any specification in the HUGO program, the system provides defaults which are listed in Appendix B.

#### 8.1.2 Saving and Restoring Attributes

When two or more very different sets of attributes apply to different paragraphs, or when a temporary change in attributes needs to be made and then canceled, it may be desirable to save away and restore sets of attributes.

```
<paragraph format statement> →
  save <attribute set name>
  restore <attribute set name> |
  save-defaults |
  restore-defaults
```

```
<attribute set name> → <identifier>
```

The save statement saves the currently active set of character and line attributes under the given name. The restore statement resets all the character and line attributes to those at the time of the last save with the same name.

The save-defaults statement replaces the attributes stored at the end of the initially sections with the current set of attributes. So the defaults restored at the start of each paragraph, for example, will be changed. The restore-defaults restores the default attributes in the same manner as a paragraph start.

Separate copies of the default attributes are kept for the layout sections and for the page body. In multi-stream makeup, a separate copy is kept for each column. So by using the save-defaults statement a separate set of attributes can be set up for each column.

The scope of an attribute set name is the whole HUGO program.

Example of saving and restoring the environment:

```
save my-environment
times 8pt
set 'now in times'
restore my-environment
```

#### 8.2 Paragraph Organization

The type of a paragraph determines where it will be set. It does not, however, determine the internal organization

of the paragraph: how it may be broken across column or page boundaries, and whether it is to be set in a tabular format.

If a paragraph is divided into sub-paragraphs, as described later in this chapter, then the character and line attributes are saved away at the first sub-para, chunk or division statement. These attributes are then restored at the start of each following sub-paragraph within the paragraph. So, default attributes that apply only within one paragraph or table can be specified just once in much the same way that the initially sections establish defaults for the job.

### 8.2.1 Sub-Paragraphs

The most important use of sub-paragraphs is in the breaking of a paragraph into the entries of a table or division list, whose features are described later in this chapter. Another use is the manipulation of character and line makeup attributes and of line indentation.

```
<paragraph format statement> →
  sub-para [ leave ]
```

A sub-para statement describes the simplest form of sub-paragraph. Like a paragraph start statement, it starts a new line of text and resets indentation. The other two types of sub-paragraphs are started by the chunk statement for table entries and division for parts of division lists.

If the leave option is present, the attributes are retained from the previous sub-paragraph. Otherwise the attributes specified for the paragraph are restored. As described above, the paragraph attributes are saved away at the first sub-para, chunk or division statement encountered in a paragraph. They may or may not be the same as the default attributes saved at the end of the initially section or by the save-defaults statement.

The main use of a sub-para is for indenting parts of a text paragraph. All types of sub-paragraphs may be mixed in any combination within a paragraph. A useful, specialized, use of this feature is the ability to set tables, complete with headings and explanatory material, within footnotes.

### 8.2.2 Paragraph Breaking

It may be required that any given paragraph be kept together within a page column or a paragraph may be allowed to be split amongst two or more columns. If a paragraph is split it may be required that a certain minimum number of text lines be kept together at the start and end of the paragraph.

```
<paragraph format statement> →
  block |
  unblock |
  min-para <expression> |
  widow <expression> [ , <expression> ]
```

```
<Boolean built-in function call> → block
```

block specifies that a paragraph may not be split; and unblock specifies that a paragraph may be split. These attributes may be applied to individual sub-paras as well as to paragraphs.

The block function returns success if the corresponding attribute is currently in effect.

If the block attribute is not in effect, then the argument of the effective min-para statement determines the minimum number of text lines that have to be in a paragraph before it can be split. In addition, the two arguments of the widow statement determine the minimum number of text lines that must be kept together at the start and end, respectively, of a paragraph. If the widow statement has only one argument, then it is used for both values.

Hyphenation can also control paragraph splitting, as described later in this chapter.

### 8.2.3 Interline Spacing

The space between lines can be varied based on the set-size or the type of paragraph.

```
<paragraph format statement> →
  on <expression> [ , <expression> ] |
  leading <expression>
    [ , <expression> ] |
  para-space <expression>
    [ , <expression> ] |
  suffix-space <expression>
    [ , <expression> ] |
  no-para-space |
  no-suffix-space |
  space-block <expression>
```

The on statement is used in conjunction with a character format statement that changes the setsize. When used, it immediately follows the setsize statement. The first expression specifies the distance by which the zero baselines of text lines are to be separated. Issuing a new set-size statement does not affect this leading unless it is also followed by an on statement. The second expression specifies the proportion by which the interline spacing can be increased to achieve vertical justification of columns. The number for each line is used in proportion to the sum of all the proportions of all the lines in the column. The value used for a normal text line, if not specified in the on statement, is one one-thousandth the value given in the first argument expressed in points. This means that little or no extra space can be put between normal text lines unless there is no other place to put the space.

The leading statement allows the on statement to be overridden for just one line. The default value for the second argument is the value given for the first argument. This statement is especially useful for controlling the vertical space between sub-paragraphs or chunks within a paragraph.

The para-space statement specifies the space to be put before the first line of a paragraph. The distance between the zero baseline of the last line of a paragraph and that of the first line of the next is the value in the first expression of the para-space statement plus the set height of the first line of the new paragraph. The second expression specifies the proportion this space can be increased by and if not present, defaults to the value given in the first argument.

The suffix-space statement is similar to the para-space statement except that it overrides the para-space values for the next paragraph and so specifies the space to be placed following the current paragraph. It is useful for head paragraphs and is almost always used with

carried-head and continued-head paragraphs, in which case the suffix-space values get carried along with the head.

In the absence of effective para-space and suffix-space statements, the space preceding the first line of a paragraph is determined by any currently effective on statement. The no-para-space and no-suffix-space statement cancel the effect of any current para-space statement respectively.

The space-block creates a vertical space of the size specified by its argument. Its primary use is for reserving space in text columns for illustrations and for vertically positioning headings on title pages.

Interline spacing is never used preceding the first, or following the last line of a column.

#### 8.2.4 Paragraph Enquiries

```
<built-in function call> →
  para-space |
  suffix-space
```

The para-space and suffix-space functions return the amount of space, in points, applicable to the attributes they name. If no-para-space is currently in effect, then para-space returns the space normally placed between two successive text lines, that is, the difference between the current set height and the current leading specified by the on statement. If no-suffix-space is currently in effect, then zero is returned by suffix-space. Note that these values are not necessarily those used on the current paragraph, as other factors also determine the space around lines placed in the page body.

### 8.3 Types of Paragraphs

It may be required that any given paragraph be kept together within a page column or a paragraph may be allowed to be split amongst two or more columns. If a paragraph is split it may be required that a certain minimum number of text lines be kept together at the start and end of the paragraph.

#### 8.3.1 Text Paragraphs

```
<paragraph start statement> →
  head [ leave ] |
  tag [ leave ] |
  inter-para [ leave ] |
  cut-line [ leave ] |
  text [ leave ] |
  display-head [ leave ]
```

The types of paragraphs listed above differ in the way in which they can be placed within a column of text. The rules are:

- head: the paragraph may not appear at the bottom of a column;
- tag: the paragraph may not appear at the top of a column;
- inter-para: the paragraph may not appear at either the top or bottom of a column;
- cut-line: the paragraph may not be broken, and if it appears at the top or the bottom of a column it is deleted from the output text;
- text: the paragraph is free to appear anywhere in a column.

- display-head: the paragraph will be placed at the top of the next column, before the carried heads, if it does not fit completely in the current text column. If the paragraph is moved, the text of the next paragraph following it will still be fit into the current column, so this type of paragraph is useful for correctly positioning certain types of illustrations and tables.

#### 8.3.2 Carried Heads

Carried heads allow for the setting of running heads in the page body.

```
<paragraph start statement> →
  carried-head <carried head name>
  [ leave ] |
  continued-head <carried head name>
  [ leave ]
```

```
<paragraph format statement> →
  kill-head <carried head name> |
  kill-all-heads
```

```
<carried head name> → <identifier>
```

carried-head paragraphs cannot be broken across column or page boundaries nor may they appear at the bottom of a column. A carried-head paragraph appears at the top of every column after its definition until it is replaced by another carried head with the same carried head name, until it is deleted by a kill-head statement using the same carried head name, or until a kill-all-heads statement does just that. More than one carried-head paragraph may be carried from column to column so long as they have different carried head names.

continued-head paragraphs are carried heads and are identical to carried-head paragraphs except that they are not set at the point in the text where they first appear. They are set at the top of every column thereafter until deleted or replaced in the same manner as a carried-head paragraph.

The kill-head statement deletes any carried head which is currently being carried with its carried head name. It will also delete any carried heads defined since the carried head of its own name. If there is no carried head with the kill-head statement's carried head name then the statement has no effect.

The kill-all-heads statement deletes all carried heads not yet otherwise deleted. Both the kill-head and kill-all-heads statements should be placed at the end of a paragraph.

Examples of paragraph start statements:

```
carried-head main
carried-head sub
continued-head sub2
```

#### 8.3.3 Reference Heads

Reference heads are running heads set outside the page body which can be used to implement such features as dictionary headings, subject titles and other updated indications of the contents of the page.

```
<paragraph start statement> →
  reference-head <reference head name>
  [ leave ]
```

```
<page format statement> →
  use-first <reference head name> |
  use-last <reference head name>
<reference head name> → <identifier>
```

reference-head paragraphs are not set in the page body. They are set in the layout when explicitly called for by a use-first or use-last statement. The at statement described in section 9.1.2 is used to position reference heads in the layout.

The use-first statement selects the first reference head with the specified reference head name from the preceding page body and sets it in the layout. The use-last statement selects the last such and sets it. If only one reference head appears in a page body with a given reference head name then it will be subsequently set by both the use-first and use-last statements. If no reference head with a given head name appears in a page body then the last reference head that was defined previous to that page body is used by the use-first and use-last statements. If no reference head with a given name appears in a job prior to a use-first or use-last statement using that name then no text is set.

If a use-first or use-last statement is used in a page layout before the body of the page is set then the reference heads will come from the previous page. If a use-first or use-last statement is used after the body then the reference heads will come from that body.

Example of defining and using a reference head:

```
reference-head top
  set 'Head' / in an "if-you-find"
  at 0,10 inches
  use-first top / in the "layout"
```

## 8.4 Tables

Tabular formatting is of two general types: that which places tables within the columns of a page, and that which lays out the whole of the page body in a table-like format. The former is discussed in this section. The latter is discussed under the heading of multi-stream makeup.

### 8.4.1 Table Definition

The table definition statement provides HUGO with a specification of tabular format to be used when next a chunk statement is encountered.

```
<table definition statement> →
  table { <statement> }
    { tcol <table column setup> }
  end-table
<table column setup> →
  <expression> [ , <expression> ]
  { <statement> }
```

The table column setups in the table definition statement specify, for each column in the table, the special attributes of that column. The first expression specifies the position of the tabular column with respect to the left edge of the page column within which the tabular entry resides. The positions of the tabular columns need not be in any particular order. The second expression, if present, specifies the width of the lines in the entry. The statements in a table column setup are used to specify

line justification methods, indentation and vertical alignment of the entry.

Tabular formats can be provided to HUGO from within any section of a program, and then used after the section has been exited. Any use within a table column setup of a non-static local variable or parameter whose scope has been exited and cannot be restored by a return from a function, user-defined statement or user-code is a violation of that variable's scope and as such is in error.

### 8.4.2 Tabular Text

The chunk statement starts a new line in a table, which is called a chunk. It invokes the latest table definition. The text for the entries within a tabular line are separated by tab statements.

When a chunk is started the paragraph attributes are first restored. The statements preceding the first tcol in the current table definition are executed, after which the character and line makeup attributes are saved away to be restored at the start of each entry in the chunk. This allows those attributes which all the entries have in common to be grouped together in the front of the table definition.

After these chunk attributes have been established, the table column setup for the first entry is executed when a chunk statement is used. Then, the HUGO system returns just following where the chunk statement appeared so that the text for the first entry can be gathered from the input. Each tab statement returns control to the next table column setup, which is executed in the same way as the first. There can be fewer entries in a chunk than there are table column setups, in which case the unused ones are just ignored. But there should not be more.

Each entry starts a new sub-paragraph, and so resets indentation. The sub-entry starts a sub-paragraph without terminating either an entry or a chunk.

```
<paragraph format statement> →
  tab |
  chunk [ leave ] |
  sub-entry [ leave ] |
```

The tab statement specifies that the next table column is to be entered. The contents of each table column setup in the table definition statements is evaluated each time the corresponding column is entered, so that the position, width and other attributes of an entry can be dynamically calculated.

### 8.4.3 Tabular Entry Alignment

```
<paragraph format statement> →
  top-align |
  bottom-align |
  drop-align |
  centre-align |
  top-previous |
  bottom-previous
```

top-align, bottom-align, drop-align, centre-align, top-previous and bottom-previous are intended to be used in a table column setup and specify the method of vertically aligning the entries in a table as follows:

- (a) top-align: the first line of the entry is aligned at the top of the longest entry in the current chunk,
- (b) bottom-align: the last line of the entry is aligned at the bottom of the longest entry in the current chunk,
- (c) drop-align: the baseline of the first line in the entry is lined up with the zero baseline of the last line in the previous entry,
- (d) centre-align: the entry is vertically centred on the longest entry in the current chunk,
- (e) top-previous: the first line of the entry is aligned at the top of the previous entry in the current chunk,
- (f) bottom-previous: the last line of the entry is aligned at the bottom of the previous entry in the current chunk.

top-align is assumed if no other alignment is specified.

The tabular format defined in the sample program in Appendix E is used in this report in Chapter 9.5.3 and in Appendix B.2 and B.3. The following example illustrates the establishment of bottom-align as the default for this sort of table, the overriding of that default, and the use of a variety of typestyles within a table.

```

table
  bottom-align
  tcol 1cm, 3cm
  top-align
  flush-right
  tcol 4.5cm
  flush-left-with '.', 1 thin
  tcol 8cm, 2cm
  flush-left
  setsize .7 em on 1.1 ex
end-table

```

#### 8.4.4 Tabular Enquiries

```

<built-in function call> →
  max-column |
  tabular-column |
  tabular-position

```

These functions can only be used while a table is being set. max-column returns the number of table column setups in the currently effective table definition. tabular-column returns the number of the current entry, counting the first entry in a chunk as 1. tabular-position returns the position specified in the table column setup for the current entry.

The tabular-column function only reflects the tabular makeup actions that have actually been performed. If a "tab" character has already been encountered in the input stream it will not be reflected in the tabular-count until that character has been interpreted as the "tabbing" action by the tabular composition facilities. Depending on how the text editing facilities of the language are implemented (Chapter 11), this may not get done until an if-you-find or line end is actioned from the input. On the other hand this is of no concern unless the tabular-column function is used in a pattern expression to control whether or not an if-you-find is to be selected for use. If this is the case for any particular HUGO

program, then to ensure that all "tabs" are actioned when they are encountered the following section should be placed at the end of the user's program:

```

if-you find tab tab
/ i.e. if you find a tab, do a tab

```

#### 8.5 Footnotes

Footnotes can be set in either the single- or multi-stream forms of page makeup described in chapter 9. A footnote is a line or paragraph placed at the bottom of a text column. A line which contains a reference to the footnote must also be present. HUGO ensures that the footnote and its reference appear in the same column.

HUGO does not automatically number or mark with asterisks or daggers either footnotes or their references. This can, however, be easily done by the user with the other facilities that HUGO provides and in the manner that the user desires. For example the asterisk\* in this sentence was input at the same point as the reference to the footnote below.

##### 8.5.1 Defining Footnotes

```

<paragraph start statement> →
  footnote [ leave ] |
  foot-head [ leave ]

```

```

<paragraph format statement> →
  foot-ref |
  no-foot-head

```

A footnote paragraph can be placed before or after the paragraph which contains the reference to it. The foot-ref statement is placed on the line which is to determine the column in which its footnote is to be included. No identification is used to link footnotes and their references, the first footnote in a job is associated to the first reference, and so on with the second and later pairs. If the reference to a footnote appears within the footnote paragraph itself, the footnote is associated with the first line of the next following text paragraph.

A footnote head is a line or paragraph used to separate the text of a column from its footnotes. Only one footnote head is used for each column. The one used is the last foot-head paragraph which precedes the first footnote set in the column. If no footnotes are set in a column then no footnote head will be set in that column. If the no-foot-head statement precedes any given footnote, then if that footnote is the first in its column, then no footnote head will be used.

##### 8.5.2 Positioning Footnotes

Normally footnotes appear at the bottom of the column in which they are referenced. No attempt is made to balance the number of footnotes in each column, and in page bodies which are terminated without being filled with text, footnotes are set within the shorter body. Some control is available over this normal situation.

\*Just an example of a footnote.

```
<page format statement> →
  foot-balance |
  foot-bottom  |
  foot-lineup  |
  no-foot-balance |
  no-foot-bottom |
  no-foot-lineup
```

`foot-balance` only has effect in multi-stream makeup. It causes HUGO to attempt to include the same number of footnote references in each column following the last lineup point in the text. It will therefore cause matching footnotes to appear in the same page body.

`foot-bottom` causes footnotes to always appear pushed to the bottom of the full body depth specified in the defining body statement. This may mean that extra space has to be placed before the first footnote or the footnote headings.

`foot-lineup` only has effect in multi-stream makeup. It specifies that the top of the first footnote in each column or of the footnote heading, if one is present, is to be at the same vertical position.

`no-foot-balance`, `no-foot-bottom` and `no-foot-lineup` specify that the corresponding control is to be disabled, and the normal situation restored.

### 8.6 Sidenotes

Sidenotes are entered and referenced like footnotes. Rather than the sidenotes themselves being set within the text columns, they are set in their own columns and vertically aligned with the lines which reference them.

```
<paragraph start statement> →
  sidenote [ leave ]
<paragraph format statement> → side-ref
<page format statement> →
  overfill <expression>
```

A sidenote paragraph can be placed before or after the paragraph which contains the reference to it. The `side-ref` statement is used to indicate the place where a sidenote is referenced, and functions the same way as the `foot-ref` statement.

The top of the setheight of the first line of a sidenote is aligned with top of the setheight of the line referencing it. If this would cause it to overlap other sidenotes, it is placed immediately below these other sidenotes.

Adjustments can be made to the exact vertical position of sidenotes with the `baseline` statement.

The position of the column in which sidenotes are set is determined by the `columns` or `multi-columns` statement controlling page makeup, as described in chapter 9.

Normally, if a sidenote does not fit within the current body depth, then it and its reference will be pushed to the next column or page. The `overfill` statement specifies that a sidenote may extend below the body depth without being pushed to the next column. Its argument is a

number specifying the maximum amount by which any sidenote can so overhang before the normal mechanism is invoked.

### 8.7 Explanatory Notes

Explanatory notes are like footnotes and sidenotes in that they are referenced from, and related to matter in normal text paragraphs, but they differ in that they are not set in the same page body as the text which references them. They can be set within the same page layout as their references, but are usually set on a facing page, and to aid in this, special line-up facilities are provided. The statements below are used to define and reference explanatory note paragraphs. The statements used to define where these paragraphs are set in the page layout are described in Chapter 9.2.5.

```
<paragraph start statement> →
  ex-note [ leave ] |
  ex-note-head [ leave ]
<paragraph format statement> →
  ex-note-ref |
  no-ex-note-ref
```

As with footnotes and sidenotes, an explanatory note reference within the referenced paragraph is considered by the system to be on the first line of the next normal text paragraph.

The system will attempt to set an explanatory note in the first `ex-note-body` (see Chapter 9.2.5) following the body in which it is referenced. If there is too much explanatory note text for an `ex-note-body` then the current `ex-note-head` paragraph, if any, will be used as a continuation head at the top of the next `ex-note-body` into which the note will be carried. `ex-note-heads` are only ever set as continuation heads. `no-ex-note-head` says there is no such head.

### 8.8 Division Lists

A division list is a form of table in which the entries run down the page column instead of across the column.

```
<paragraph format statement> →
  division <expression> { , <expression> }
```

The division statement, like the chunk statement, starts a group of sub-paragraphs. The division list, like a chunk of a table, should be embedded within a paragraph. The number of expressions determines in how many sub-columns the entries are to be placed. The expressions specify the horizontal position of each sub-column. The entries in a division list are divided amongst the sub-columns as evenly as possible.

The entries of a division list, like those of a table, are separated by the `tab` statement.

Division lists can be used to pack lists of short items into a text column as in the list of makeup attributes in Chapter 8.1.



## 9 Pages

Text may be typeset during the execution of the initially and finally sections of the HUGO program. This text is placed in the output line by line with no page structuring. All other text is set within the confines of the pages in the job. The main function of the page formatting facilities is to position text within pages.

### 9.1 Page Layouts

A page layout has a width and a height. The page headings and body are positioned within the layout by the HUGO program. These items must be positioned so as to allow for the proper left, right, top and bottom margins on the page.

Positions within the layout are expressed as pairs of distances from the left and top edges of the page layout.

More than one page layout may be used in a job. Different chapters may be formatted differently. Different layouts may be required for the first page of a job and for even and odd numbered pages.

Refer to the sample program in Appendix E for the page layout used to define the format of the pages in this report.

#### 9.1.1 The Page Dimensions

```
<layout section> →
  layout <layout name> <layout shape>
    { <local declaration group> }
    { <statement> }
```

```
<layout name> → <identifier>
```

```
<layout shape> →
  <expression> , <expression> |
  <expression> [ portrait | landscape ]
```

A layout section describes a page format. The layout name is used by a first-layout or next-layout statement to specify the sequence of page formats. The statements contained in the layout section specify the various headings, and the location and depth of the body of the page.

The layout shape specifies the shape and size of the page. If two expressions are given, then they are taken to be the page width and height respectively. If only one expression is given together with the portrait option then that value is taken to be the page width and the HUGO system calculates an appropriately proportioned depth in such a way that the page depth is the long dimension. If the landscape option is specified then the given value is the depth and the HUGO system calculates and appropriately proportioned width so that the page width is the long dimension. If neither portrait or landscape is specified then portrait is assumed.

The "appropriately proportioned" dimensions calculated for the portrait and landscape forms of the layout shape produce a page such that if the page is cut in half parallel to its short dimension then the result will be two pages exactly half the area of the original and of exactly the same shape. This is the rule used to produce the metric standard page sizes described in section 9.5.3.

#### 9.1.2 Positioning Text in a Page

```
<page format statement> →
  at <expression> , <expression>
```

At any given stage of the processing of a page layout, the HUGO system is at some unique position within that layout. That position is where the next item of text in the page will be set. The at statement allows the user to specify the position of the next item of text. The two expressions specify the distance from the lefthand edge and the top edge, respectively, of the page layout.

If the next line of text is a heading then the position specified in the at statement will be the lower lefthand corner of the text line, that is at the left end of the line's baseline. If further heading lines are set without intervening at statements, then they will be placed at the same horizontal position as the first but advanced by the amount given in the applicable on, para-space or suffix-space statement.

If the next text set after the at statement is the page body then the at statement specifies the location of the top lefthand corner of the body. If the first line after the page body is set without an intervening at statement, it will be advanced from the actual depth at which the body was set rather than that specified in the body statement. This allows page headings to be set tight against the page body, if required.

The elements of a page layout may be set in any order that satisfies the requirements of an individual job. The page headings, body and footings do not have to be set in that order. If a subject title is to depend on the contents of the current page it must be set after the page body even though it is positioned above the page body. Likewise subject information from the previous page must be set before the page body is set.

#### 9.1.3 Sequence of Pages

```
<first-layout section> →
  first-layout <layout name>
```

```
<page format statement> →
  next-layout <layout name>
```

The first-layout program section specifies the first page layout to be used in a job. It must be used in a job or no pages will be set.

The next-layout statement specifies the page layout to be used after the current page has been completed. If no next-layout statement appears in a layout section then the current page layout will be reused.

### 9.2 The Page Body

The text for page headings is generally specified in a HUGO program. The text for the page body and for reference headings comes from a separate file called the text file. The data in this file is processed by the if-you-find sections of the HUGO program and then made available for inclusion in the page body. When the HUGO system encounters a body statement, it collects as many lines as will fit into that page body, sets them

under the control of the various body formatting statements supplied to the system by the HUGO program, and then continues to execute the page layout.

A page body is divided into columns. There are many ways in which a body may be divided into columns, and though it is not usually done, part of a body may be set with one type of division and another part may be set with a different type. The top half of a body may be set with two columns, for example, and the bottom half with three columns. If the body format is changed part way through a page body then the text previous to the change is set over as short a depth as possible, and the remaining body depth is used to set the new format.

### 9.2.1 Entering and Leaving the Page Body

```
<page format statement> →
  body <expression> |
  finish-body [ long ] |
  finish [ long ] |
  new-body [ long ] |
  sub-body |
  normalize-page |
  finish-column |
  reserve <expression>
```

The body statement is used in a layout section to indicate the location and depth of a page body. An at statement is used to position the page body. The expression in the body statement gives the depth. The width of the body is determined by the type of column makeup used and by the width of the lines set in the body. More than one body may be set within a layout section.

The finish-body statement indicates that the text in the page body is to be set over as short a depth as possible, that execution of the page body is to be terminated, and that execution of the page layout is to resume immediately following the body statement. The new-body statement does exactly the same thing except that if the current page body does not contain any text, execution of the page body will not be terminated. A finish-body or new-body statement should appear in an if-you-find section or an at-end section.

The finish statement forces the end of the text input file. It terminates a page body in the same manner as a finish-body statement and specifies that no layout sections are to be executed after the current one. After the current layout section is completed, the finally sections will be executed, then the job is terminated.

The long option following either the finish-body, new-body or finish statement specifies that instead of the remaining text being set over as short a depth as possible, the text is to be set over the full remaining body depth, as if it were an intermediate page in a chapter.

The sub-body statement does the same thing as the new-body statement as far as setting the text in the page body over as short a depth as possible, but it leaves control in the current body if possible.

The normalize-page statement indicates to HUGO that an acceptable location at which to break a text column has been reached. HUGO will then flush as much text as possible into the current page body, and therefore allow the user to reliably enquire as to the appropriate value of

the page-count function. This method of synchronization should only be used when this information is needed to determine whether a page break is to be called for by the user, as when the text in two separate documents is being matched up.

The finish-column statement specifies that the current column of text is to be immediately terminated, independently of the decisions of the body-filling algorithms.

The reserve statement specifies that if the space given by its numeric argument is not available in the current column, then it is to be terminated immediately and a new text column started. It can be used to keep a measured amount of text following it together on the same page.

### 9.2.2 Column Justification

Normally a columns, multi-columns, finish-body or finish statement causes the existing body text to be set over the shortest possible depth before continuing its own action. They also cause the set text to be justified vertically over the depth, as does leaving a page body when it has been overfilled with text. These actions can be prevented from happening with the following commands.

```
<page format statement> →
  top-flush |
  justify-vertically |
  evenup |
  no-evenup |
  evenup-step <expression> |
  maximum-expansion <expression> |
  no-maximum-expansion
```

top-flush prevents vertical justification of columns. justify-vertically causes it to resume, and is the default.

evenup causes text to be set over the shortest possible depth whenever a columns, finish-body or finish statement is executed. no-evenup causes text to be set over the full remaining body depth and suppresses balancing the text in the columns of single stream makeup. The argument of the evenup-step statement specifies the accuracy to which column balancing is to be attempted.

The argument of maximum-expansion indicates the maximum amount of extra space that can ever be placed between two lines to achieve vertical justification. This will prevent large "gaps" being left in extreme cases, but may have the offsetting disadvantage of preventing a column of text from being fully justified. no-maximum-expansion indicates there is no maximum limit on vertical justification.

### 9.2.3 Space in the Page Body

```
<page format statement> →
  body-space <expression> [ , <expression> ]
```

The first expression in the body-space statement specifies the amount of vertical space to be placed in the page body whenever a sub-body, columns or multi-columns statement is executed. This space separates blocks of text set using different body formats within the same body.

The second expression, if present, indicates the amount of vertical space that should remain in the body after the

body space has been taken from it. If it does not, then the body is to be terminated, and the layout reentered.

#### 9.2.4 Galley-Form Page Bodies

Sometimes the power of the page body formatting facilities are not required, and only a simple method of filling a page body is required.

```
<page format statement> →
galley <expression>
```

The galley statement performs the function of the body statement except that no distinction is made when placing lines in the body as to the paragraph type of those lines, none of the page format controls have any effect, and no vertical justification is performed. This form of page body can be used to set simple page formats, or for the setting of "proof" copy text, where the traditional "galley-form" of setting text can be emulated.

When in a galley-form page body, none of the statements in Chapter 9 have any effect except those in section 9.1, finish-body and finish. The long option of these last two statements also has no effect.

#### 9.2.5 Explanatory Note Bodies

Explanatory notes (see Chapter 8.7) become "active" when the text which references them has been set in a page body. Any active explanatory notes can be set by use of an explanatory note body.

```
<page format statement> →
ex-note-body <expression> |
ex-note-columns <expression>
{ , <expression> } |
ex-note-lineup |
no-ex-note-lineup
```

```
<Boolean built-in function> →
any-ex-notes
```

An ex-note-body can be used anywhere in a page layout. HUGO attempts to set all the active explanatory notes in the ex-note-body. If there is too much text for the body, the overflow will be kept for the next ex-note-body. An ex-note-body can be placed in the same page layout as a body, in a separate layout intended to be printed as a "facing" page to the referencing text, or all explanatory notes can be gathered together at the end of a chapter or a job. The ex-note-columns statement defines the horizontal displacement of the columns of text in an ex-note-body. For single-stream makeup there can only be one such column, and in multi-stream makeup there must be a column for each column in the body from which references are made to explanatory notes.

If ex-note-lineup is specified then an attempt is made to vertically align the first line of an explanatory note with the line on which it is referenced, even though the two lines may be in different page layouts. If there is too much text in the ex-note-body, if an explanatory note does not fit in the first ex-note-body into which HUGO attempts to fit it, or if no-ex-note-lineup is specified then no such attempt is made. However, explanatory notes in multi-stream makeup which are lined-up using the lineup statement will still be vertically aligned with respect to each other.

The any-ex-notes function succeeds if and only if there are any unset active explanatory notes. It can be used, together with a conditional statement and a next-layout statement, to control sequencing between main text page layouts and layouts used to set explanatory notes. Using this function, the user can arrange to set enough pages to make sure all explanatory notes are set adjacent to the page from which they are referenced, or the user can place explanatory notes wherever there is space available on following "facing" pages.

### 9.3 Page Makeup

There are two general types of page body formatting. Single stream makeup allows a single stream of text paragraphs to be set in multiple columns so that the last line in one column is logically followed by the first line in the next. The chief use of this form of makeup is for single language publications where the line width needs to be kept short enough to be read conveniently. Multiple stream makeup allows the text input to be divided into segments, and the page column into which each segment is to be placed to be specified. It can be used for publications which have an overall tabular composition such as catalogs, and for side-by-side bilingual publications, with or without side notes.

#### 9.3.1 Single Stream Makeup

```
<page format statement> →
columns <column position>
{ , <column position> }
```

The number of column positions in the columns statement specifies how many columns are to be set in single stream makeup column format. The value of each column position specifies the horizontal position of the corresponding column with respect to the left hand margin of the page body (*not* the page layout). These positions need not be in ascending order.

When, as usually is the case, a new page body is entered without the specification of a new columns statement, the last columns statement used in the previous page remains in effect. A columns statement may also be placed in an initially section to specify the column format for the first part of, or for the whole job.

Example of a columns statement:

```
columns 0, 3.5in, 7in / assuming width 3in
```

#### 9.3.2 Multiple Stream Makeup

```
<page format statement> →
multi-columns <column position>
{ , <column position> } |
into-column <expression> |
lineup
```

The number of column positions in the multi-columns statement specifies how many columns are to be set in multiple stream column format. The value of each column position specifies the horizontal position of the corresponding column as in the columns statement.

The into-column statement specifies into which column text is to be placed until otherwise specified. The column numbers which can be specified are in the range 1 (one) to the number of columns in the current multi-columns format. The into-column statement can only be used in

multiple stream makeup. The current column should only ever be changed at a paragraph boundary.

The lineup statement specifies that the text in all columns is to be aligned at this point. Enough vertical space is placed in each column to line up the top of the next paragraph in each column.

### 9.3.3 Page Column Positioning

A column position specifies the horizontal location of a page makeup column of text within the body of the page. It may in addition indicate the horizontal position of the column into which sidenote text is to be placed.

```
<column position> → <expression>
                    [ with <expression> ]
```

The first expression gives the position of the text columns described for single- and multi-stream makeup. The second expression, if present, gives the position of the sidenote column, the filling of which is described in chapter 8. The sidenote column may be to the left or the right of the text column, but should not, of course, overlap it.

### 9.4 Page Indexing

The indexing facilities of HUGO are designed to aid in the generation of tables of contents, of book indexes, of special forms of running headings on a page, and generally, of lists of information about how a publication is set. To do indexing it is usually necessary to save away some information about text being set and to retrieve it at a later time. This can be done using global variables, lists and the put statement and get built-in function.

If the page on which indexed text appears is not to be recorded, then the above facilities together with the text editing facilities of the language are sufficient. However, as the page on which a line of text will appear is not completely determined when that text is first set, some further aid is required to get this information.

```
<page format statement> →
  index-item <expression>
  { <statement> }
  end-index
```

```
<built-in function call> →
  index-item |
  index-x |
  index-y
```

An index-item statement must be executed while a line of text is being set in the body of a page. The expression will be evaluated at that time and saved away together with the statements enclosed in the index-item. These statements are not executed immediately. When a page body that has index-items within it is completed, and "locked up" so that the position of all text in it has been determined, then these statements will be executed. They are considered to be executed within the layout but outside the page body. So the at statement can be used in the index-item to position text, but no further text can be set within the page body.

The same index-item statement may be used more than once in any given page body. Each execution is treated independently of the others and is expected to be distinguished by the value of the expression in the index-item.

The built-in functions associated with page indexing can only be used inside an index-item. They each return some information about the execution of the index-item statement with which they are associated:

```
index-item returns the value of the expression given
when the index-item statement was executed,
index-x returns the distance in points from the left-
hand edge of the page layout to the left-hand end of
the line of text associated with the index-item.
index-y return the distance in points from the top edge
of the page layout to the baseline of the associated
line of text.
```

## 9.5 Page Makeup Enquiries

### 9.5.1 Enquiries About the Page

```
<built-in function call> →
  at-y |
  columns |
  evenup-step |
  layout-depth |
  layout-width |
  max-depth |
  max-width |
  multi-column
```

- (a) at-y can only be used in the layout outside of any body. It returns the current vertical position as determined by the at statement, by setting lines and by page bodies. After a body, this function returns the vertical position of the last line actually set in that body.
- (b) columns returns the number of columns in the current page body format.
- (c) evenup-step returns the value of the currently effective evenup-step statement.
- (d) layout-depth returns the depth of the current layout.
- (e) layout-width returns the width of the current layout.
- (f) max-depth returns the maximum depth that HUGO allows for a layout. This is also, of course, the maximum allowed body depth and the greatest allowed rule height.
- (g) max-width returns the maximum width that HUGO allows for a layout. It is also the maximum allowed line width and rule width.
- (h) multi-column returns the number of the current multi-stream makeup column as determined by the latest into-column statement. It is only meaningful in multi-stream makeup.

### 9.5.2 Areas

Page makeup divides each body within a page into areas. An area is a column in single- or multi-stream makeup. If a new page format is started part way through a page body then both the old and new formats are considered to generate their own areas. A number of built-in functions and an operator have been provided to allow a HUGO program to enquire about these areas. They can only be meaningfully used after a body is set in a layout but before that layout is completed. They can be used to number lines of text on a page and to determine where page-masking rules are to be placed.

```
<built-in function call> →
  area-count |
  area-size ( <expression> ) |
  footnote-count ( <expression> ) |
  x-posn ( <expression> )
```

```
<factor> → <factor> y-posn <subfactor>
```

area-count returns the number of areas in the previous body.

area-size returns the number of text lines in the area whose number is given by its argument. The areas are numbered starting at 1 for the first area in a body, and then in increasing order down and across the body. The text lines counted are those which are visible in the given area. Carried heads are included but footnotes, sidenotes and reference heads are not.

footnote-count returns footnote heading lines in the area given by its argument.

x-posn returns the distance in points between the left-hand edge of the layout and the left-hand edge of the column which constitutes the area given by the function's argument. Note that all lines in any given area are at the same horizontal position.

y-posn returns the distance in points from the top of the page layout to the zero baseline of the line given in the first argument in the area given by the second. Lines in

an area are numbered from 1. The footnote head lines are included in this count. The first footnote line is given the number returned by the area-size function plus 1.

### 9.5.3 Standard Page Sizes

A set of built-in functions is provided to support the standard metric page sizes. These functions return the short dimension of a page with the corresponding page size, and can be used with the portrait and landscape forms of the layout section to give the proper standard page dimensions. Note that the a4 size (which is about 8 ¼ by 11 ¼ inches) is the metric replacement for the usual letter page.

```
<built-in function call> →
  a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8
```

The results of these functions are approximately:

a0 .....	84.1 cm or 33 in.
a1 .....	59.5 cm or 23 ½ in.
a2 .....	42.0 cm or 16 ½ in.
a3 .....	29.7 cm or 11 ¾ in.
a4 .....	21.0 cm or 8 ¼ in.
a5 .....	14.9 cm or 5 ¾ in.
a6 .....	10.5 cm or 4 ¼ in.
a7 .....	7.4 cm or 3 in.
a8 .....	5.2 cm or 2 in.

## 10 Ruling and Underlining

The ruling facilities can be used to draw lines around, under, and through text. They can be used in association with the character, line, paragraph and page makeup facilities of HUGO, depending on what features are to be used to control the size, shape and position of the line.

In some types of ruling, such as underlining, the lines are drawn with only one orientation, in this case, horizontal. However, with the more general line drawing facilities, both horizontal and vertical lines can be drawn. In these cases the HUGO facilities allow the user to draw a solid rectangle. This rectangle can be drawn wide and of short height to produce a horizontal line, or it can be drawn narrow and high to produce a vertical line. The "length" and "thickness" of a line have to be interpreted accordingly.

### 10.1 Underlining

Underlining is used in conjunction with character makeup and can be used to place lines under individual characters, under words or under phrases. Extensive underlining can be avoided by the use of side-lining, which summarizes a group of underlined lines by placing a vertical line beside those lines.

#### 10.1.1 Basic Underlining

```
<character format statement> →
  underline |
  no-underline |
  underline-specs <expression> , <expression>
```

underline turns underlining on and no-underline turns it off. It can be turned on and off at any point in a line. Not only text, but quad spaces, spacebands, and leader-fill can be underlined.

underline-specs specifies the thickness and position of the line to be used for underlining. The first expression specifies the distance between the text lines' zero baseline and the top edge of the underline. The second specifies its thickness. These values are interpreted in proportion to the current setsize and will be modified by the system to reflect any change in setsize. They should therefore be expressed using the ex operator.

As an example, part of this line is underlined just by using the underline statement to turn it on and the no-underline statement to turn it off.

#### 10.1.2 Side-lining

```
<paragraph format statement> →
  side-line <expression> , <expression>
  [ , <expression> , <expression> ] |
  side-line-vary <expression> , <expression>
  [ , <expression> , <expression> ] |
  no-side-line
```

The side-line and side-line-vary statements both indicate that side-lining is to be used and they specify the position and width of the side-line. When enabled a sideline will be drawn beside a text line whose text would otherwise have been entirely underlined. Successive side-lined

lines will have their side-lines connected to produce a single line. The four expressions in the side-line and side-line-vary statement indicate:

- The horizontal distance between the left edge of the side-line and the left-hand end of the text line. A negative number places the line to the left of the text line, and a positive number greater than the line width places it to the right of the text line.
- The width of the side-line.
- The distance above the zero baseline of a side-lined text line where the side-line is to start. If not given, the setheight of the first character set on the line is used.
- The distance below the zero baseline of a side-lined text line that the side-line is to end. If not given, the side-line ends on the baseline.

The two forms of side-line differ in that side-line-vary adjusts the position of the top of the side-line by the amount of extra space placed before the first line side-lined. This allows successive, but separate, side-lines to be connected and viewed as a single line. This adjustment is not performed by side-line.

no-side-line indicates that underlines are to be left under all fully underlined lines.

### 10.2 Rule-Filling

Rule-filling provides the same sort of line expansion as the leader-filling described in chapter 7 with the quad-with statement, except that the line is expanded using a solid line with no extra space inserted at the front or within the line as might be the case when filling with a dash character. When present in a text line, a space-, leader- or rule-filling point is used to place all the extra space of the line.

```
<line format statement> →
  quad-with-rule |
  flush-left-with-rule |
  justify-with-rule |
  rule-specs <expression> , <expression>
```

quad-with-rule, flush-left-with-rule, and justify-with-rule function exactly as do quad-with, flush-left-with and justify-with respectively, except that instead of a character, a rule is used for filling.

rule-specs specifies the thickness and position of the line to be used for rule-filling. The specification is interpreted in the same way as that for underlining. The position measurement is taken from the text's zero baseline to the top of the line, and is usually a negative number which places the line above the text baseline.


For example, Quad-with-rule does \_\_\_\_\_this.

### 10.3 Figure Rules

A figure is a character explicitly constructed with straight lines and rectangles. It fits in a text line as does any other character and can be back-spaced over or be back-spaced onto another character or figure.

```
<character format statement> →
  figure <expression>
    { fig-rule <expression> , <expression> ,
      <expression> , <expression> }
  end-figure
```

The expression following the keyword figure specifies the width of the character to be created. Each solid rectangle is drawn by a fig-rule. Horizontal lines are drawn as rectangles which are wider than they are high. Vertical lines are drawn as rectangles which are higher than they are wide. The four expressions in each fig-rule specify the horizontal position, vertical position, width and height respectively of a rectangle. The positions are measured from the zero baseline and left-hand end of the width specified for the figure to the top left-hand corner of the rectangle. Positive positions are measured to the right and down from this point. fig-rules may overlap the edges of their figure.

As an example, this  was created using the following figure definition:

```
figure 20 points
  fig-rule 0, -5 pt, 5 pt, 5 pt
  fig-rule 5 pt, -3 pt, 10 pt, 1 pt
  fig-rule 15 pt, -5 pt, 5 pt, 5 pt
end-figure
```

#### 10.4 Ruling in the Page Layout

```
<page format statement> →
  rule <expression> , <expression>
```

This statement draws a solid rectangle of the width given in its first argument and of the height given in its second argument. The rectangle's top left-hand corner is placed at the current position in the page layout. Therefore, this command is usually immediately preceded by an at statement.

The rule statement is used to draw lines at fixed positions and of fixed sizes in the page layout. So, for example, it can be used to draw lines under the page headings, or a box around the page body.

#### 10.5 Ruling in the Page Body

```
<paragraph format statement> →
  rule <expression> , <expression> ,
    <expression> , <expression> |
  vertical-rule <expression> , <expression> ,
    <expression> , <expression> |
  vertical-rule-vary <expression> , <expression> ,
    <expression> , <expression> |
  end-para-rule <expression> , <expression> ,
    <expression> , <expression>
```

These statements are intended to be used in the text columns in a page body, primarily to draw lines between and around paragraphs and table entries. All the statements are normally placed at the start of a paragraph, because the rules are positioned using the zero baselines of the first and last lines of the current paragraph.

The first two arguments of rule indicate the position of the top left-hand corner of the solid rectangle it draws in relation to the left-hand end of the zero baseline of the current line of the paragraph. They are the distance of that corner to the right, and its distance below the baseline, respectively. A negative distance "below" will place

the rectangle above the baseline. The third and fourth arguments indicate the width and height, respectively, of the rectangle.

The end-para-rule statement acts just the same as the rule statement, except that the rectangle is positioned with respect to the left-hand end of the zero baseline of the last line of the current paragraph, rather than that of the current line. To allow for the breaking of paragraphs across columns, the end-para-rule is also used at the bottom of any column whose last line is one of the lines in the current paragraph. In this case the last line from the paragraph in the column is used to determine the position of the rule. No similar facility is provided for automatically placing rules at the top of any column into which a paragraph is broken as more flexibility in the specification of heading material is usually required. This situation can be dealt with by the use of rules in carried-heads and continued-heads.

The vertical-rule and vertical-rule-vary statements act just the same as the side-line and side-line-vary statements, respectively, except that the lines are not drawn to replace underlines. Instead, the top of the vertical rule is positioned with respect to the left-hand end of the current zero baseline and the bottom of the rule with respect to the left-hand end of the zero baseline of the last line of the paragraph.

Note that the rule, vertical-rule and end-para-rule statements can be used to draw the top, side and bottom lines, respectively, of a box around a paragraph. In addition, the position of a rule and of the top of a vertical-rule are determined by the current line's position. So if they are used on an intermediate line of a paragraph, the lines they draw will be, or will start within a paragraph. This is especially useful in tables.

When a paragraph is split across text columns or pages, the vertical rules will be split accordingly. Also the end-para-rules are drawn at the bottom of each part of the split paragraph. If a carried head is defined to carry a rule to the top of the next column when splitting a paragraph, then paragraphs can have boxes drawn around them whether they are split or not.

For example, the box around this sentence was created by the statements listed below.

```
rule -2 pt, -1 ex, width plus 4 pt, 1 pt
vertical-rule -3 pt, 1 pt, -1 ex, 3 pt
vertical-rule width plus 2 pt, 1 pt,
-1 ex, 3 pt
end-para-rule -2 pt, 2 pt,
width plus 4 pt, 1 pt
```

#### 10.6 Rule Enquiries

The four built-in functions below return the current effect of the underline-specs and rule-specs statements. The heights and positions of the line they generate are dependent on the current setheight.

```
<built-in function call> →
  underline-posn |
  underline-height |
  rule-posn |
  rule-height
```

The four functions return the underline position and thickness, and the filling rule position and thickness

respectively. The values returned are those defined as

the arguments of the underline-specs and rule-specs statements.



## 11 Text Input

The setting of continuous text in the body of a page is the primary purpose of a typesetting system. The text for the body constitutes the bulk of the input for any job. HUGO provides an automatic text-editing capability for the processing of the many varieties of conditions which can occur in this text. It is the responsibility of the writer of the HUGO program to specify those attributes of the text which identify different types of paragraphs, the format of tables and the conditions under which typeset changes.

The one feature of HUGO that probably most distinguishes it from other text composition systems is the entire absence of system-defined commands in the textual input. If the nature of a specific job requires input commands then they must be defined through the text editing facility.

### 11.1 Automatic Text Editing

An if-you-find section specifies a condition in the input text and its corresponding action.

```
<if-you-find section> →
  if-you-find <pattern expression>
  { <local declaration group> }
  { <statement> }
```

The pattern expression specifies the condition in the input text to be acted on by this section. At each position in the input, the if-you-finds are looked at in order of their appearance in the HUGO program. The first one to successfully compare is used by the system. The statements within it are executed. The text in the input which was matched by the if-you-find is stepped over and not processed further. The succeeding input character is the next one scanned by the if-you-finds.

Any character of the input unmatched by any if-you-find is set in the output by a system generated set statement.

### 11.2 The Start of the Text

```
<at-start section> →
  at-start
  { <local declaration group> }
  { <statement> }
```

The at-start section specifies a sequence of actions to be performed when HUGO attempts to fill the first page body in a job with text. The statements within the at-start section are executed just before any input is read or any if-you-finds are executed.

The at-start section is primarily used to set headings within the first page body and to set default values for running headings.

### 11.3 The End of the Text

```
<at-end section> →
  at-end
  { <local declaration group> }
  { <statement> }
```

The at-end section specifies a sequence of actions to be performed when the end of the input text is encountered. Any text set during execution of the at-end section will be set within the page body of the last page(s) in the job.

Note that the final layout section will be completed and the finally sections executed after the execution of the at-end section. The action taken at the end of the input text in the absence of any at-end section is that of the finish statement.

### 11.4 Patterns

A pattern expression is like a Boolean expression except that it includes tests of the input text stream and performs the other text matching functions required by an if-you-find. However, its chief characteristic is still that it either succeeds or fails.

```
<pattern expression> → <pattern subexpression>
  [ to-local <local variable> |
    to-global <global variable> |
    to-global <list variable> ]

<pattern subexpression> → <pattern term> |
  <pattern subexpression> or <pattern term>

<pattern term> → <pattern factor> |
  <pattern term> and <pattern factor>

<pattern factor> →
  <pattern built-in function call> |
  ( <pattern expression> ) |
  <Boolean factor> |
  <expression> |
  <list name>
```

The built-in functions given above are only meaningful in pattern expressions and cannot be used elsewhere. They are defined in sections 11.4.2, 11.4.3, 11.4.4 and 11.4.5.

Examples of if-you-find sections:

```
if-you-find '...'
  set '-'
if-you-find s1
  spaceband
  / puts a spaceband at the
  / start of each input line
if-you-find
  '[quad,' and (without(')) to-local q)
  and ']'
  quad q points
  / this defines a "quad" command that
  / can be placed in the input
```

#### 11.4.1 Pattern Expressions

The or and and operators in a pattern can be interpreted as: A or B means "find either A or B in the input stream"; A and B means "find A followed by B in the input stream".

A pattern which is a Boolean subfactor succeeds or fails as the Boolean subfactor succeeds or fails. It is considered to match the input stream if it succeeds but uses up no text characters in doing so. A pattern factor which is an expression is considered to succeed only if the character string which is the result of the expression matches character-for-character with the corresponding number of characters in the input stream.

### 11.4.2 Group Patterns

Group patterns match with any member of a class of strings. The class to be used can either be specified by the user or by HUGO.

There are two types of group patterns: those which match only with strings of a given length, and those which match with a string of any length. Both types of group patterns can be defined to only match with characters selected from a given set (for example, the digits). The fixed-length group pattern fails if either it finds a character not in the given set or there are not sufficient characters left in the current input line for the comparison to be performed. The variable-length group patterns never fail, and at worst match with the zero-length string.

```
<pattern factor> →
  <expression> alphas |
  <expression> chars |
  <expression> digits |
  <expression> lc-alphas |
  <expression> uc-alphas |
  <expression>
    within <expression> |
  <expression>
    without <expression>
```

Fixed-length group patterns are specified by the operators:

- alphas: which will match with the given number of alphabetic characters;
- chars: which will match with any string of the given length;
- digits: which will match with the given number of digits (0,1,2,3,4,5,6,7,8,9);
- lc-alphas: which will match with the given number of lower-case alphabetic characters;
- uc-alphas: which will match with the given number of upper-case alphabetic characters;
- within: which will match with any string of the length specified by its first argument, made up only of characters selected from its second argument; and
- without: which will match with any string of the length specified by its first argument, made up only of characters *not* contained in its second argument.

```
<pattern built-in function call> →
  alphas |
  chars |
  digits |
  lc-alphas |
  uc-alphas |
  within ( <expression> ) |
  without ( <expression> )
```

Variable-length group patterns are specified by the built-in functions with the same names as the above operators. The functions `alphas`, `chars`, `digits`, `lc-alphas` and `uc-alphas` have no arguments, and the functions `within` and `without` have one argument, corresponding to the second argument of the related operator. They will match with the longest possible string of the kind of characters they represent. Note that the `chars` function

matches with all the remaining text of the current input text line.

### 11.4.3 Position Patterns

```
<pattern built-in function call> →
  el |
  lpos ( <expression> ) |
  rpos ( <expression> ) |
  sl |
  to-lpos ( <expression> ) |
  to-rpos ( <expression> )
```

The `lpos` function has one argument which specifies a position within the input text line. The very start of a line is considered position 1 and each character position to the right increases the count by 1. So the argument of `lpos` can be considered the position of the next character, measured from the left of the input text line. The `lpos` function succeeds only if it is called when the pattern matching process is at that position in the line.

The `rpos` function has one argument which specifies a position within the input text line. The very end of a line is considered position 0 and each character position to the left increases the count by 1. So the argument of `rpos` can be considered the position of the next character, measured from the right of the input text line. The `rpos` function succeeds only if it is called when the pattern matching process is at that position in the line.

`sl` succeeds only if it attempts to match at the start of an input line. It produces exactly the same effect as `lpos(1)`.

`el` succeeds only if it attempts to match at the very end of an input line. It produces exactly the same effect as `rpos(0)`. `el` "uses up" the end-of-line condition, so that if it (or for that matter `rpos(0)`) is part of a successful pattern match on a line, then no other if-you-finds will match following it on the same line.

`to-lpos` will match with all the text in a line up until it reaches the position from the left-hand end of the line specified by its argument. It will fail if either that position has already been passed or if the line isn't long enough to get to the specified position.

`to-rpos` will match similarly with all the text up to the specified position, but in this case measured from the right-hand end of the line. As in the case of `to-lpos`, it will fail if it cannot get to that position.

### 11.4.4 Scanning Patterns

Scanning patterns allow the construction of more complex searches than those allowed by the use of basic patterns. Scanning patterns have patterns as their arguments and modify the normal operation of those patterns in a variety of ways.

```
<pattern built-in function call> →
  followed-by ( <pattern expression> ) |
  many ( <pattern expression> ) |
  not-followed-by ( <pattern expression> ) |
  optional ( <pattern expression> ) |
  up-to ( <pattern expression> )
```

`followed-by` succeeds if the pattern which is its argument succeeds, but `followed-by` does not actually "use up" any text and is considered to match with zero characters. It is useful for checking the right-hand context of

an item being picked up by a pattern, yet leaving it in the input stream for processing by other if-you-finds.

not-followed-by is similar to followed-by in its action, but as its name implies, succeeds only if the pattern which is its argument fails, and fails otherwise.

many will use the pattern which is its argument as many times as possible. It is therefore a generalization of the group patterns. For example, digits means the same thing as many(1 digits). Like the group patterns, many will succeed even if its pattern does not match, in which case it will match with zero characters.

up-to is the generalization of the without function, much as many is the generalization of the within function. It matches with everything up to but not including the part of the input line matched by its argument. It is, however, unlike without in that if it is unable to find the pattern it will fail.

optional matches with the string matched by its argument. However, if that match fails, optional does not fail but rather matches with zero characters.

Example of using many:

```
If-You-Find '.' and many('..')
/ This will match with any string of periods
/ separated by dashes, whereas
If-You-Find '.' and within('..')
/ will match with a period followed by any
/ number of periods and dashes
/ in any order.
```

#### 11.4.5 Examining the Next Line

A limited search of the text line following the line under consideration by an if-you-find can be made using the following functions.

```
<pattern built-in function call> →
  next-line ( <pattern expression> ) |
  no-next-line
```

next-line examines the start of the next line for the pattern given as its argument. It will fail if either the pattern fails or if there is, in fact, no next line. The latter case means that the line being examined by the if-you-find is the last line of the input.

Another way of determining if the current line is the last line of the input is to use the no-next-line function, which succeeds if there is no next line. It will match with zero characters.

Neither the next-line nor the no-next-line functions can be used in the pattern which is the argument of next-line. Therefore, no more than two lines can be examined at one time without the text of those lines being saved away, for example, stored in global variables.

#### 11.4.6 List Patterns

A pattern factor which is a list name instructs HUGO to compare the next part of the current input line with the key of every element in the named list. If comparison finds character-by-character equivalence, then the pattern factor succeeds, and matches with the part of the input line successfully compared with. If two or more keys in a list can compare successfully, then one of them is arbitrarily selected by HUGO.

Note that the key, rather than the value of each list element is compared with. This way the value of the matched element may be used to determine the action taken by the HUGO program.

As an example, using the list defined in the example in Chapter 4.1.4:

```
if-you-find L
/ will match with any digit from 1 to 9.
```

#### 11.4.7 Pattern Parameters

The to-local option of a pattern expression allows text matched in the input stream to be processed by the HUGO program. The local variable specified in the to-local is given the value of the actual input string with which the pattern expression matches. The appearance of the local variable in the to-local option constitutes its declaration as a local variable of the if-you-find. The same local variable may be used more than once in a pattern, but the uses must be separated by ors in such a way that for any match, only one value is given to the local variable.

to-global operates in the same way as to-local, except that the variable which is to be given a value must be either a global variable or an element of a list. Unlike to-local, to-global does not constitute a declaration of the given variable. The global variable or list must have been previously declared as such in the HUGO program, although the global variable or list element need not have been previously given a value.

The local variable declared in a to-local may not be used to produce the value given to it while the pattern is being processed. This is because the local variable isn't actually created until the pattern has succeeded, and the associated program section entered. to-global, on the other hand, does its job immediately on the pattern which precedes it matching. So its value can be used later in the same pattern, and the value of the global variable or list element is changed whether or not the pattern finally succeeds.

Care should be taken in using pattern parameters embedded within other pattern elements. For example:

```
if-you-find '$' and
  (optional(1 alpha) to-local x)
/ will assign the alphabetic character found
/ to 'x' if one was found, and the null
/ string otherwise. However
if-you-find '$'
  and optional(1 alpha to-local x)
/ will assign a value to 'x' only if the
/ alphabetic character is found. If one is
/ not found, then any use of 'x' within
/ the if-you-find section will be in error.
```

#### 11.5 The Rescan Statement

```
<rescan statement> → rescan <expression>
```

The rescan statement allows the HUGO program to generate a piece of text to be processed by the if-you-finds. This text can be picked up by one if-you-find, and then rescanned by another. Or it can be entirely the result of the computations of the program.

The effect of the rescan statement is that its expression argument is replaced in the current input record at the

place currently advanced to by the if-you-finds. When next resumed, the searching of the if-you-finds will recommence at the start of the rescanned string. The insertion of the rescanned string in the input record does not affect the text already there, and in effect, lengthens

the record. So the interpretations of the lpos and rpos builtin-functions will be affected: in particular, the lpos values of all the remaining characters in the line will be increased by the length of the rescanned string.

## 12 General Enquiries

This chapter contains descriptions of those parts of the HUGO language which do not affect the function of HUGO programs, but which allow the user to check on the status of the HUGO system. They may be used to typeset or print headings for a composition job or to implement an accounting package to be used with HUGO. Other facilities, more dependent on the specific implementation of HUGO, are described in Appendix A.

### 12.1 Statistics

<built-in function call> →  
 char-count |  
 char-in-page |  
 chars-input |  
 film-length |  
 line |  
 line-count |  
 page-count |  
 total-set

All these functions return some indication of the amount of work done by the HUGO system. They indicate:

- (a) char-count: the total number of characters set,
- (b) char-in-page: the total number of characters set in the current layout,
- (c) chars-input: the total number of characters read from the text file,
- (d) film-length: the total amount of linear output medium used (in points).
- (e) line: the total number of lines read from the text file,
- (f) line-count: the total number of lines set,
- (g) page-count: the total number of layouts started,
- (h) total-set: the total width of all lines set (in points),

### 12.2 Time and Date

<built-in function call> →  
 time-of-day |  
 seconds |  
 date |  
 julian-date

These functions return:

- (a) time-of-day: a printable representation of the current time using the 24-hour clock, to the nearest second. For example: 00:18:58,
- (b) seconds: the time of day represented as the number of seconds since midnight accurate to a thousandth of a second,
- (c) date: a printable representation of the current date. For example: 21 AUG 80
- (d) julian-date: the date represented as a five-digit number, the first two digits of which are the last two digits of the year and the last three of which are the number of the day in the year, with January 1 as 001 and December 31 as 365 (or 366 in leap years). For example: 80234

## Appendix A - Implementation Dependent Features

The features of HUGO described in this appendix are very much dependent on the IBM/370 implementation of the language which produces output to drive the APS/4 phototypesetter. They may be required to have substantially different meanings on other computers or may even be meaningless there. These features are provided because interaction between a HUGO program and non-standard elements in its environment may sometimes be necessary, but they should not be used otherwise.

### A.1 Special Conversion Functions

It is usually of no concern to the user of HUGO how string and numeric values are stored inside the computer. But when a file is being used that was produced by another computer program, or when a file is put to be used by another program, the requirements of that other program may necessitate the use of special conversions. These conversions convert back and forth between the representations used by HUGO and those used by the other programs. The other representation is stored by HUGO in a string of characters which should only be used for input or output.

The binary and packed data types described in this section are a feature of the IBM/370 computer and may or may not exist on other computers.

```
<subexpression> →
  <subexpression>
  bin <term> |
  <subexpression>
  pack <term>
```

```
<subfactor> → <subfactor> ebcdic
```

```
<built-in function call> →
  unbin ( <expression> ) |
  unpack ( <expression> ) |
  attn
```

The bin operator converts the number specified by the second argument into binary format. The first argument specifies the number of characters that that binary representation is to occupy.

The pack operator converts its second argument to packed decimal format. The length of the result is given by the first argument. Leading zeros will be placed in the argument to force a specific length.

The unbin function has one argument, which is assumed to be a string in binary format, and which is converted back into a number. The unpack similarly converts a string back from packed format.

The ebcdic operator gives the character whose EBCDIC number is the argument. Each character has an EBCDIC number whose value is determined by the implementation and which allows it to be represented in a program via the ebcdic command even if the character is not on the input device.

The attn function returns the character normally used to represent the "break" character on the input device used to produce text input for HUGO.

### A.2 Device-Dependent Character Setting

Sometimes it may be necessary to use the character codes of the typesetting device on which the output of a HUGO program is set rather than those of the computer on which the HUGO program is run. In addition, it may occasionally be desirable to issue commands in the language of the typesetting device.

```
<character format statement> →
  escape-char <expression> |
  no-escape-char |
  set-absolute
  <expression> [ , <expression> ]
```

```
<built-in function call> → escape-char
```

The escape-char statement has as its argument a single character which is to become the current "escape character". When an attempt is made to set the escape character then the character following the escape character is set without conversion from the computers' character set to that of the typesetting device. The no-escape-char statement specifies that there is no escape character.

The escape-char built-in function returns the current escape character if there is one, and the zero-length string otherwise.

The set-absolute statement sets all its first argument without conversion from the computers' character set. It may contain commands in the phototypesetters' command language. The second argument, if present, is used as the width of the first argument when set. This width is used for line filling and justification.

### A.3 Documents

The file used to input text for processing by the if-you-find sections can be divided into sub-files called documents. The following built-in functions allow the HUGO program to gather information about these documents, about the input and output files and about the HUGO job.

```
<built-in function call> →
  input-volume |
  output-volume |
  document-line
```

The input-volume function returns the name or number of the device used as the text file. Under OS/VS1 this is the volume serial number of the input device. But when the "DOC" run-time directive is specified to divide the input into documents, this function returns the identification of the current document being read.

The output-volume function returns the name or number of the device on which text is set by a HUGO job. Under OS/VS1, this is the volume serial number of the tape used to drive the phototypesetter.

The document-line function returns the count of lines read in the current document if the "DOC" run-time directive is used. Otherwise it returns the same value as the line built-in function.

**A.4 Job Identification and Accounting**

```

<statement> →
  account
    <expression> [ , <expression> ]
<built-in function call> →
  job |
  job-name

```

The account statement places the data contained in its first argument on the accounting file attached to the HUGO job and flags the record as being of the type specified in the second argument. If the second argument is not present, then the record type is chosen by the HUGO system. The format of the accounting information, and the available record types, are determined by the accounting system being used.

The job function returns the job identification information provided by the "JOB" run-time directive. If "NOJOB" is in effect, this function returns the zero-length string.

The job-number function returns the name or number of the HUGO job. This value is determined by the environment in which HUGO is running. When running under OS/VS1 it is the job name on the JCL JOB card.

**A.5 Cpu-Time**

```

<built-in function call> → cpu-time

```

If the information is available to HUGO, this function returns the number of seconds of computer time the HUGO job has used. This number should be accurate to one thousandth of a second. If not available, this function may either return zero, or a counted value indicating in some way how much work the HUGO job has done.

**A.6 Error Handling**

If the HUGO system encounters a user-caused error at run-time, then the at-job-end is entered immediately.

This allows statistics to be gathered for all jobs, whether they run to normal completion or not. If no at-job-end section is in the HUGO program, or if the error is encountered in the at-job-end section itself, then the user's program is terminated immediately, without further processing of any kind.

```

<statement> →
  error <expression> |
  warning <expression>

```

```

<Boolean built-in function call> → error

```

The error statement causes HUGO to take the action it normally performs when a serious user error is encountered. The expression argument is used as the text of the error message generated. The error statement will terminate normal program execution and either stop the program immediately, or cause the at-job-end section to be entered.

The warning statement is similar to the error statement except that, as with other non-terminal user errors, just a message is issued, and normal program execution is not terminated.

The error function returns success if the at-job-end section has been entered due to a user error. It will return failure if the at-job-end was entered normally, and will also return failure if used in any other section of a HUGO program.

**A.7 Source Line Numbers**

```

<built-in function call> → source-line

```

Each line in a HUGO source program is given a unique number, which may or may not have an alphabetical suffix, and which is used in the source listing produced by the HUGO compiler. source-line returns the number of the line in the HUGO program from which it is called. It may be used to identify user-generated error messages.

## Appendix B - Language Summary

### B.1 Syntax

This appendix gathers together all the BNF rules in the body of the report.

First, though, here is a short summary of BNF itself:

- <thing> → means "a thing is ..."
- <this> <that> means "this followed by that"
- <this> | <that> means "either this or that"
- { <this> } means "zero or more of this"
- [ <this> ] means "an optional this" (zero or one of this).

#### The Grammar:

- <add statement> →  
add <expression> to <variable>
- <assignment statement> →  
assign <expression> to <variable>
- <at-end section> →  
at-end  
{ <local declaration group> }  
{ <statement> }
- <at-job-end section> →  
at-job-end  
{ <local declaration group> }  
{ <statement> }
- <at-job-start section> →  
at-job-start  
{ <local declaration group> }  
{ <statement> }
- <at-page-end section> →  
at-page-end  
{ <local declaration group> }  
{ <statement> }
- <at-page-start section> →  
at-page-start  
{ <local declaration group> }  
{ <statement> }
- <at-start section> →  
at-start  
{ <local declaration group> }  
{ <statement> }
- <attribute set name> → <identifier>
- <Boolean built-in function> →  
any-ex-notes
- <Boolean built-in function call> →  
not ( <Boolean expression> )
- <Boolean built-in function call> → eof
- <Boolean built-in function call> →  
bedford |  
bold |  
excelsior |  
helvetica |  
italic |  
medium |  
modern |  
normal-case |  
old-helvetica |

perma |  
roman |  
small-caps |  
times

- <Boolean built-in function call> →  
break |  
english |  
french |  
hyph
- <Boolean built-in function call> → block
- <Boolean built-in function call> → error
- <Boolean expression> → <Boolean term> |  
<Boolean expression> or <Boolean term>
- <Boolean factor> →  
<Boolean built-in function call> |  
( <Boolean expression> )
- <Boolean factor> →  
<expression> lex-lt <expression> |  
<expression> lex-le <expression> |  
<expression> lex-gt <expression> |  
<expression> lex-ge <expression> |  
<expression> is <expression> |  
<expression> isnt <expression> |  
<expression> verify <expression>
- <Boolean factor> →  
<expression> eq <expression> |  
<expression> ne <expression> |  
<expression> lt <expression> |  
<expression> le <expression> |  
<expression> gt <expression> |  
<expression> ge <expression> |  
<expression> even |  
<expression> odd
- <Boolean factor> →  
<expression> is-in <list name> |  
<expression> isnt-in <list name>
- <Boolean term> → <Boolean factor> |  
<Boolean term> and <Boolean factor>
- <built-in function call> →  
length ( <expression> ) |  
floor ( <expression> ) |  
lc ( <expression> ) |  
uc ( <expression> )
- <built-in function call> →  
tab | bks |  
char-string | digit-string | alpha-string |  
lc-alpha-string | uc-alpha-string
- <built-in function call> → input
- <built-in function call> → get ( <expression> )
- <built-in function call> → list-size ( <list name> )
- <built-in function call> →  
string ( <expression> ) |  
number ( <expression> )



- <built-in function call> →  
 baseline |  
 font |  
 fudge |  
 slant |  
 width-of ( <expression> )
- <built-in function call> →  
 hyphen |  
 hyph-fill |  
 left-for |  
 left-indent ( <expression> ) |  
 right-indent ( <expression> ) |  
 sb-ratio |  
 sb-step |  
 width
- <built-in function call> →  
 para-space |  
 suffix-space
- <built-in function call> →  
 max-column |  
 tabular-column |  
 tabular-position
- <built-in function call> →  
 index-item |  
 index-x |  
 index-y
- <built-in function call> →  
 at-y |  
 columns |  
 evenup-step |  
 layout-depth |  
 layout-width |  
 max-depth |  
 max-width |  
 multi-column
- <built-in function call> →  
 area-count |  
 area-size ( <expression> ) |  
 footnote-count ( <expression> ) |  
 x-posn ( <expression> )
- <built-in function call> →  
 a0 | a1 | a2 | a3 | a4 | a5 | a6 | a7 | a8
- <built-in function call> →  
 underline-posn |  
 underline-height |  
 rule-posn |  
 rule-height
- <built-in function call> →  
 char-count |  
 char-in-page |  
 chars-input |  
 film-length |  
 line |  
 line-count |  
 page-count |  
 total-set
- <built-in function call> →  
 time-of-day |  
 seconds |  
 date |  
 julian-date
- <built-in function call> →  
 unbin ( <expression> ) |  
 unpack ( <expression> ) |  
 attn
- <built-in function call> → escape-char
- <built-in function call> →  
 input-volume |  
 output-volume |  
 document-line
- <built-in function call> →  
 job |  
 job-name
- <built-in function call> → cpu-time
- <built-in function call> → source-line
- <carried head name> → <identifier>
- <character> → any graphic character other than that which terminates the enclosing quoted constant
- <character format statement> →  
 set <expression>
- <character format statement> →  
 shift-up |  
 shift-down |  
 no-shift
- <character format statement> →  
 times <expression> [ , <expression> ] |  
 modern <expression> [ , <expression> ] |  
 excelsior <expression> [ , <expression> ] |  
 helvetica <expression> [ , <expression> ] |  
 old-helvetica <expression> [ , <expression> ] |  
 perma <expression> [ , <expression> ] |  
 bedford <expression> [ , <expression> ] |  
 setsize <expression> [ , <expression> ] |  
 font <expression> , <expression> [ , <expression> ]
- <character format statement> →  
 roman |  
 italic |  
 bold |  
 medium
- <character format statement> →  
 slant <expression> |  
 baseline <expression> |  
 fudge <expression>
- <character format statement> →  
 small-caps |  
 normal-case
- <character format statement> →  
 quad <expression> |  
 quad-to <expression>
- <character format statement> →  
 underline |  
 no-underline |  
 underline-specs <expression> , <expression>

```

<character format statement> →
  figure <expression>
    { fig-rule <expression> , <expression> ,
      <expression> , <expression> }
  end-figure

<character format statement> →
  escape-char <expression> |
  no-escape-char |
  set-absolute
  <expression> [ , <expression> ]

<column position> → <expression>
  [ with <expression> ]

<conditional statement> →
  if <Boolean expression>
    { <statement> }
  { else-if <Boolean expression>
    { <statement> } }
  [ else
    { <statement> } ]
  end-if

<constant> →
  <quoted constant> |
  <numeric constant>

<declaration section> →
  <global declaration section> |
  <list declaration section> |
  <function declaration section> |
  <user declaration section>

<digit> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<expression> → <subexpression> |
  <expression> cat <subexpression>

<factor> → <subfactor> |
  <factor> index <subfactor>

<factor> →
  <factor> th-elem <list name>

<factor> →
  <factor> pp <subfactor>

<factor> → <factor> y-posn <subfactor>

<finally section> →
  finally
  { <local declaration group> }
  { <statement> }

<first-layout section> →
  first-layout <layout name>

<function call> →
  <function name> |
  <function name>
  ( <expression> { , <expression> } )

<function call> →
  <function name> |
  <function name>
  ( <expression> { , <expression> } )

<function declaration section> →
  function <function name>
  [ <type> ] [ entry ] [ static ]
  [ starts-as <expression> ]
  [ with <parameter declaration>
    { , <parameter declaration> } ]
  { <local declaration group> }
  { <statement> }

<function name> → <identifier>

<global declaration section> →
  global <global variable declaration>
  { , <global variable declaration> }

<global variable> → <identifier>

<global variable declaration> →
  <global variable> [ <type> ] [ entry ]
  [ starts-as <expression> ] |
  <global variable> [ <type> ] [ entry ]
  literally <expression>

<HUGO program> →
  { <program section> | <segment> }
  end-hugo

<identifier> →
  <letter> { <letter> | <digit> | - }

<if-you-find section> →
  if-you-find <pattern expression>
  { <local declaration group> }
  - { <statement> }

<indentation margins> →
  <expression> [ for <expression> ]
  [ , <expression> ] |
  <expression> for-all

<initially section> →
  initially
  { <local declaration group> }
  { <statement> }

<layout name> → <identifier>

<layout section> →
  layout <layout name> <layout shape>
  { <local declaration group> }
  { <statement> }

<layout shape> →
  <expression> , <expression> |
  <expression> [ portrait | landscape ]

<letter> →
  A | B | C | D | E | F | G | H |
  I | J | K | L | M | N | O | P |
  Q | R | S | T | U | V | W | X |
  Y | Z | À | Á | Ê | Ë | È | É |
  Î | Ï | Ò | Ó | Ô | Õ | Ç

<line format statement> →
  width <expression> |
  finish-line |
  turn-over

<line format statement> →
  centre |
  flush-left |
  flush-right |
  force-justify |
  justify |
  justify-centre |
  justify-right

```

```

<line format statement> →
  spaceband |
  vari-space |
  sb-ratio <expression> |
  sb-step <expression>
<line format statement> →
  quad-out |
  quad-with <expression> [ , <expression> ] |
  flush-left-with <expression>
    [ , <expression> ] |
  justify-with <expression>
    [ , <expression> ]
<line format statement> →
  break |
  no-break |
  break-here |
  discretionary <expression>
<line format statement> →
  hyph |
  nohyph |
  english |
  french |
  hyph-margins <expression>
    [ , <expression> ] |
  ladder <expression> |
  hyphen <expression> |
  hyph-fill <expression> |
  min-word <expression> |
  stick-hyphens |
  no-stick-hyphens
<line format statement> →
  left-indent <indentation margins> |
  right-indent <indentation margins>
<line format statement> →
  quad-with-rule |
  flush-left-with-rule |
  justify-with-rule |
  rule-specs <expression> , <expression>
<list declaration section> →
  list <list name declaration>
    { , <list name declaration> }
<list name> → <identifier>
<list name declaration> →
  <list name> [ <type> ] [ entry ]
<list variable> → <list name> ( <expression> )
<local declaration group> →
  local <local variable declaration>
    { , <local variable declaration> }
<local variable> → <identifier>
<local variable declaration> →
  <local variable> [ <type> ] [ static ]
    { starts-as <expression> } |
  <local variable> [ <type> ]
    literally <expression>
<numeric constant> →
  [ + | - ] <digit> { <digit> }
  [ . { <digit> } ] |
  [ + | - ] . <digit> { <digit> }
<output statement> → output <expression>
<page format statement> →
  use-first <reference head name> |
  use-last <reference head name>
<page format statement> →
  foot-balance |
  foot-bottom |
  foot-lineup |
  no-foot-balance |
  no-foot-bottom |
  no-foot-lineup
<page format statement> →
  overfill <expression>
<page format statement> →
  at <expression> , <expression>
<page format statement> →
  next-layout <layout name>
<page format statement> →
  body <expression> |
  finish-body [ long ] |
  finish [ long ] |
  new-body [ long ] |
  sub-body |
  normalize-page |
  finish-column |
  reserve <expression>
<page format statement> →
  top-flush |
  justify-vertically |
  evenup |
  no-evenup |
  evenup-step <expression> |
  maximum-expansion <expression> |
  no-maximum-expansion
<page format statement> →
  body-space <expression> [ , <expression> ]
<page format statement> →
  galley <expression>
<page format statement> →
  columns <column position>
    { , <column position> }
<page format statement> →
  multi-columns <column position>
    { , <column position> } |
  into-column <expression> |
  lineup
<page format statement> →
  index-item <expression>
    { <statement> }
  end-index
<page format statement> →
  rule <expression> , <expression>
<page format statement> →
  ex-note-body <expression> |
  ex-note-columns <expression>
    { , <expression> } |
  ex-note-lineup |
  no-ex-note-lineup

```

- <paragraph format statement> →  
 save <attribute set name>  
 restore <attribute set name> |  
 save-defaults |  
 restore-defaults
- <paragraph format statement> →  
 sub-para [ leave ]
- <paragraph format statement> →  
 block |  
 unblock |  
 min-para <expression> |  
 widow <expression> [ , <expression> ]
- <paragraph format statement> →  
 on <expression> [ , <expression> ] |  
 leading <expression>  
 [ , <expression> ] |  
 para-space <expression>  
 [ , <expression> ] |  
 suffix-space <expression>  
 [ , <expression> ] |  
 no-para-space |  
 no-suffix-space |  
 space-block <expression>
- <paragraph format statement> →  
 kill-head <carried head name> |  
 kill-all-heads
- <paragraph format statement> →  
 tab |  
 chunk [ leave ] |  
 sub-entry [ leave ] |
- <paragraph format statement> →  
 top-align |  
 bottom-align |  
 drop-align |  
 centre-align |  
 top-previous |  
 bottom-previous
- <paragraph format statement> →  
 foot-ref |  
 no-foot-head
- <paragraph format statement> → side-ref
- <paragraph format statement> →  
 ex-note-ref |  
 no-ex-note-ref
- <paragraph format statement> →  
 division <expression> { , <expression> }
- <paragraph format statement> →  
 side-line <expression> , <expression>  
 [ , <expression> , <expression> ] |  
 side-line-vary <expression> , <expression>  
 [ , <expression> , <expression> ] |  
 no-side-line
- <paragraph format statement> →  
 rule <expression> , <expression> ,  
 <expression> , <expression> |  
 vertical-rule <expression> , <expression> ,  
 <expression> , <expression> |
- vertical-rule-vary <expression> , <expression> ,  
 <expression> , <expression> |  
 end-para-rule <expression> , <expression> ,  
 <expression> , <expression>
- <paragraph start statement> →  
 head [ leave ] |  
 tag [ leave ] |  
 inter-para [ leave ] |  
 cut-line [ leave ] |  
 text [ leave ] |  
 display-head [ leave ]
- <paragraph start statement> →  
 carried-head <carried head name>  
 [ leave ] |  
 continued-head <carried head name>  
 [ leave ]
- <paragraph start statement> →  
 reference-head <reference head name>  
 [ leave ]
- <paragraph start statement> →  
 footnote [ leave ] |  
 foot-head [ leave ]
- <paragraph start statement> →  
 sidenote [ leave ]
- <paragraph start statement> →  
 ex-note [ leave ] |  
 ex-note-head [ leave ]
- <parameter declaration> →  
 <local variable> [ <type> ] [ static ]
- <pattern built-in function call> →  
 alphas |  
 chars |  
 digits |  
 lc-alphas |  
 uc-alphas |  
 within ( <expression> ) |  
 without ( <expression> )
- <pattern built-in function call> →  
 el |  
 lpos ( <expression> ) |  
 rpos ( <expression> ) |  
 sl |  
 to-lpos ( <expression> ) |  
 to-rpos ( <expression> )
- <pattern built-in function call> →  
 followed-by ( <pattern expression> ) |  
 many ( <pattern expression> ) |  
 not-followed-by ( <pattern expression> ) |  
 optional ( <pattern expression> ) |  
 up-to ( <pattern expression> )
- <pattern built-in function call> →  
 next-line ( <pattern expression> ) |  
 no-next-line
- <pattern expression> → <pattern subexpression>  
 [ to-local <local variable> |  
 to-global <global variable> |  
 to-global <list variable> ]

```

<pattern factor> →
  <pattern built-in function call> |
  ( <pattern expression> ) |
  <Boolean factor> |
  <expression> |
  <list name>

<pattern factor> →
  <expression> alphas |
  <expression> chars |
  <expression> digits |
  <expression> lc-alphas |
  <expression> uc-alphas |
  <expression>
    within <expression> |
  <expression>
    without <expression>

<pattern subexpression> → <pattern term> |
  <pattern subexpression> or <pattern term>

<pattern term> → <pattern factor> |
  <pattern term> and <pattern factor>

<program section> →
  <set-up section> |
  <declaration section> |
  <if-you-find section> |
  <user-code section> |
  <first-layout section> |
  <layout section>

<put statement> →
  put <expression> , <expression>

<quoted constant> →
  '{ <character> }' |
  '{ <character> }' |
  "{ <character> }" |
  « { <character> } »

<reference head name> → <identifier>

<repetitive statement> →
  loop
    { <statement> |
      exit-if <Boolean expression> }
  end-loop

<rescan statement> → rescan <expression>

<segment> →
  segment
    { <program section> }
  end-segment

<set-up section> →
  <initially section> |
  <finally section> |
  <at-start section> |
  <at-end section> |
  <at-job-start section> |
  <at-job-end section> |
  <at-page-start section> |
  <at-page-end section>

<statement> →
  <assignment statement> |
  <add statement> |
  <conditional statement> |
  <repetitive statement> |
  <output statement> |
  <put statement> |
  <user statement> |
  <then statement> |
  <character format statement> |
  <paragraph start statement> |
  <paragraph format statement> |
  <table definition statement> |
  <page format statement> |
  <rescan statement> |
  <stop statement>

<statement> →
  account
    <expression> [ , <expression> ]

<statement> →
  error <expression> |
  warning <expression>

<stop statement> → stop

<subexpression> → <term> |
  <subexpression> of <term> |
  <subexpression> start-at <term>

<subexpression> →
  <subexpression>
    bin <term> |
  <subexpression>
    pack <term>

<subfactor> →
  <variable> |
  <constant> |
  <function call> |
  <built-in function call> |
  <subscription> |
  <factor> percent |
  ( <expression> )

<subfactor> →
  <subfactor> ems |
  <subfactor> ens |
  <subfactor> thins |
  <subfactor> ex |
  <subfactor> picas |
  <subfactor> points |
  <subfactor> inches |
  <subfactor> cm |
  <subfactor> mm

<subfactor> → <subfactor> ebcidic

<subscription> →
  <subfactor>
    ( <expression> [ , <expression> ] )

<subterm> → <factor> |
  <subterm> mul-by <factor> |
  <subterm> div-by <factor> |
  <subterm> modulo <factor>

<table column setup> →
  <expression> [ , <expression> ]
  { <statement> }

<table definition statement> →
  table { <statement> }
  { tool <table column setup> }
  end-table

```

<term> → <subterm> |  
    <term> plus <subterm> |  
    <term> minus <subterm>

<then statement> → then

<type> → string | number

<user-code section> →  
    user-code <expression> [ only ]  
    { , <expression> [ only ] }  
    { <local declaration group> }  
    { <statement> }

<user declaration section> →  
    define <user statement name>  
    [ entry ]  
    [ with <parameter declaration>  
      { , <parameter declaration> } ]  
    { <local declaration group> }  
    { <statement> }

<user statement> →  
    <user statement name> |  
    <user statement name>  
    <expression> { , <expression> }

<user statement name> → <identifier>

<variable> →  
    <function name> |  
    <global variable> |  
    <local variable> |  
    <list variable>

**B.2 List of Synonyms**

The symbols in the first column may be used to replace those in the second wherever they appear in the grammar.

& (ampersand) .....	and	en .....	ens
:(colon) .....	cat	ex-note-column .....	ex-note-columns
*(asterisk) .....	of	inch .....	inches
!(exclamation mark) .....	or	in .....	inches
% (percent) .....	percent	indent .....	left-indent
@ (at sign) .....	start-at	iuf .....	if-you-find
;(semicolon) .....	then	justify-center .....	justify-centre
alpha .....	alphas	lc-alpha .....	lc-alphas
by .....	, (comma)	lex-eq .....	is
called .....	to-local	lex-ne .....	isnt
center .....	centre	mul .....	mul-by
center-align .....	centre-align	p .....	picas
char .....	chars	pica .....	picas
column .....	columns	point .....	points
digit .....	digits	pt .....	points
div .....	div-by	pts .....	points
em .....	ems	t .....	tcol
		thin .....	thins
		uc-alpha .....	uc-alphas
		x .....	mul-by

**B.3 Operator Priorities**

The order of execution of the various types of operators in HUGO is determined by the language's grammar. The following is a summary of that order.

Each operator has a priority level which determines the order in which it is executed. Higher priority operators are evaluated first. For example, in:

a plus b mul-by c minus d

the mul-by operator is evaluated before the other two because of its higher priority. Operators of the same priority level are evaluated in left-to-right order. So in the example, plus is evaluated before minus. Parentheses can, of course, be used to change this order.

The priority levels of the operators are:

to-local	0
to-global	0
or	1
and	2
alphas	3
chars	3
digits	3
eq	3
even	3
ge	3
gt	3
is	3
is-in	3
isnt	3
isnt-in	3
lc-alphas	3
le	3
lex-ge	3
lex-gt	3

lex-le	3
lex-lt	3
lt	3
ne	3
odd	3
uc-alphas	3
verify	3
within	3
without	3
cat	4
bin	5
of	5
pack	5
start-at	5
minus	6
plus	6
div-by	7
modulo	7
mul-by	7
index	8
pp	8
th-elem	8
y-posn	8
cm	9
ebcdic	9
ems	9
ens	9
ex	9
inches	9
mm	9
percent	9
picas	9
points	9
thins	9



**B.4 Default Attributes**

Unless otherwise specified in a HUGO program, the character, line and paragraph makeup attributes resulting from the execution of the following statements apply.

```

column 0
baseline 0
body-space 1 pica, 0
break
english / hyphenation
evenup
evenup-step 1 pica
ex-note-lineup
fudge 0
hyphen '.'
hyph-fill 100%
hyph-margins 3, 3
justify
justify-vertically
ladder 3
left-indent 0 for-all
min-para 3
min-word 5
no-escape-char
no-foot-balance
no-foot-bottom

```

```

no-foot-head
no-foot-lineup
no-maximum-expansion
no-para-space
no-shift
no-sideline
no-suffix-space
no-underline
overfill 2 picas
right-indent 0 for-all
rule-specs 5% ex, -30% ex
sb-ratio 27%
sb-step 0.1 points
slant 0
stick-hyphens
times 10 points on 11 points
    roman
    medium
    normal-case
top-align
unblock
underline-specs 5% ex, 15% ex
widow 2, 2
width 40 picas

```

## Appendix C - Using the TEPB Implementation of HUGO

The current implementation of HUGO runs on an IBM/370 computer operating under OS/VS1. HUGO jobs must be described to the operating system by the Job Control Language (JCL). This appendix describes how to submit HUGO jobs, the options available to aid in running HUGO jobs, and a number of utility programs to help both in this process and in the manipulation of text prepared for running through HUGO. These facilities are not part of HUGO itself but are part of the support provided to allow HUGO to be used conveniently.

### C.1 Invoking the HUGO System

Three catalogued procedures are available for running HUGO on the IBM/370 computer. The first is used when the HUGO program and the input data can be included "in-stream" with the data. The JCL is:

```
// EXEC HUGO1
    HUGO program source
//GO.SYSIN DD *
    input data
//
```

The second catalogued procedure is used to allow greater flexibility in the input of the HUGO program and its data. The program source should be placed as a single document on an archive tape followed by the input data split into as many documents as required. In this case the JCL is:

```
// EXEC XHUGO1,V=tape#
//
```

The third procedure is used to run Bilingual jobs. The program source should be placed as a single document on an archive tape, followed by the MERGE description of the input codes used (see the description of the MERGE program), followed by the English and French input data split into documents as required. The JCL to be used is:

```
// EXEC BHUGO1,V=tape#
//
```

The tape used for XHUGO1 or BHUGO1 should be one of the machine room work tapes.

### C.2 Compiler Directives

The HUGO compiler reads a HUGO program called the *source program*. It generates a form of this program suitable for execution by the HUGO run-time system, which is called the *object program*. In addition, the compiler produces a variety of listings suitable for printing on a high-speed line printer.

Compiler directives control not the object program produced by the compiler, but rather the printed listings and other products of the compiler. Some directives can be specified in the PARM field of the JCL card, some can be specified in the source of the HUGO program and some can be specified in both places.

The directives specified in the PARM field must be preceded by a slash character and separated by commas. No additional space characters are permitted in the PARM field, and all the information must be entered

using upper-case letters. Those directives that must be given a value such as "SIZE" and "INCLUDE" must be separated from their values by an equals sign, as in "SIZE=10000".

The directives specified in the source of the HUGO program must each be placed on a line by themselves, with the directive preceded by a hyphen character. Any number of space characters may be placed before or after the hyphen character. Any directive to be given a value, such as "INCLUDE" must be separated from its value by one or more space characters, as in "- INCLUDE TEXT". Either upper- or lower-case letters may be used to enter these directives.

The following is a list of available directives. All can be used in either the PARM field or program source unless otherwise indicated. All directives which enable a condition or a listing can be disabled by prefixing the directive with "NO".

**AUTOTAB** causes the compiler to automatically generate code for the user-code section "user-code tab" without it needing to be included in the program source. It can only be specified in the PARM field.

**DUMP** instructs the compiler to list the object program it outputs in numeric form.

**GOMAP** instructs the compiler to include a record of the line numbers in the object program so that they may be used to generate error messages at run-time.

**INCLUDE** has as its value the name of a source library member, the text of which is to be included in the input to the compiler. The source library is described later.

**ISOURCE** instructs the compiler to list the contents of all source library members included by directives in the source program.

**LEVEL** has as its value the lowest level of error message to be generated. Messages are grouped into levels to minimize the number of error messages generated by a single error in the source program. This directive can only appear in the PARM field.

**LEX** instructs the compiler to list each symbol it reads from the input together with the lexical class the compiler has assigned to it. It can only be specified in the PARM field.

**LIST** instructs the compiler to list the object program that it generates, in symbolic form.

**MAP** instructs the compiler to list the correspondence between the source lines and the code it generates. This is the table output by the GOMAP directive.

**MAXERR** has a value which is the number of errors the compiler can encounter in a source program before it terminates compilation. This directive can only be given in the PARM field.

**OBJNUM** instructs the compiler to print the location in the generated object program corresponding to each line of source program if SOURCE is enabled. This number is similar to that listed by the MAP directive.

**PAGE** instructs the compiler to list the next line of the source program on a new page. This directive is only meaningful when the SOURCE directive is

enabled, and can only be placed in the source program.

PROTECT instructs the compiler to allow the put, account and stop statements, the get built-in function and the at-job-start, at-job-end, at-page-start and at-page-end sections only in source program text taken from the source library member included by a PARM field directive and to flag them as errors if they come from any other source. This directive can only come from the PARM field.

PSOURCE instructs the compiler to list the program source lines included from a source library member by a directive in the PARM field. This directive can only come from the PARM field.

SIZE has a value which is the size in bytes of the largest object program the compiler can generate. The value can also be specified in thousands of bytes by suffixing the number by the letter "K". This directive can only appear in the PARM field.

SOURCE instructs the compiler to list the text of the input source program.

TABLES instructs the compiler to list the size of the various components in the object program it generates and also all variables, user-defined statements and built-in functions used in the program.

Unless otherwise specified the compiler uses these directives:

```
AUTOTAB NODUMP GOMAP
ISOURCE LEVEL=4 NOLEX
NOLIST NOMAP MAXERR=9999
NOOBJNUM NOPROTECT NOPSOURCE
SIZE=40K SOURCE NOTABLES
```

Any source program line which the compiler considers to contain an error is listed in spite of NOSOURCE, NOISOURCE or NOPSOURCE being in effect.

When using one of the procedures described in section C.1, the PARM field for the compiler must be referred to on the EXEC card either as PARM.HUGO, or as COPT, in which case the slash preceding the directives must be omitted.

### C.3 Run-Time Directives

The HUGO run-time system reads in an object program generated by the HUGO compiler and executes it. Run-Time directives allow certain aspects of this execution to be controlled or listed. The directives can only be entered in the PARM field in the JCL used to invoke HUGO. Directives which take a value must, as with the compiler directives, be separated from their value by an equals sign, and this list of directives may not contain any space characters.

A list of directives follows. As with the compiler, those directives which enable conditions or listings can be prefixed with "NO" to disable them.

CMDT asks for a count to be kept of the number of times each different command in the pseudo-machine-language processed by the interpreter is executed, and for a listing of these counts to be produced, sorted in order of frequency.

DOC indicates that the document descriptor lines generated by XPORT are to be used to divide the text

input into documents. In this case the text following the equals sign in these lines is returned as the value of the input-volume built-in function, and the number returned by the document-line function is made to count the text lines within each document starting at 1. When used to divide documents, the document descriptor lines are not made available to the if-you-find sections of the HUGO program, nor are they typeset.

FILM has a value which is the maximum length in metres of output medium to be produced by a HUGO program. The program is terminated if this length is exceeded.

GLIST instructs the interpreter to produce a short listing each time space is recovered from the area in which HUGO strings are stored.

HEAP has a value which indicates how many elements are to be allocated to all lists and index-items.

JOB may or may not have a value. If present, the value is used as the result of the job built-in function. If JOB is specified without a value, then the first line of SYSIN is examined. It is expected to start with "\*\*JOB", with JOB spelt with upper- or lower-case letters, and with one-or-more spaces after JOB. The remaining text on the line is used as the result of the job function and this line is deleted from the input.

LINET may have zero or two values, separated by a comma. This directive instructs HUGO to list each line of output text in the form used by the phototypesetter as it is created. If given, the two values give a range of *input* line numbers outside of which this listing is not to be produced.

LIST instructs HUGO to list the results of page makeup each time a text body is "locked up".

RECSIZE has a value which is the maximum length of a logical record that can be processed by the if-you-find sections. The strings inserted into the input by the rescans statement increase the size of the input records and so may affect the use of this directive.

TRACE may have zero or two values, separated by a comma. This directive instructs HUGO to list each command in the object program as it is executed. If given, the two values give a range of locations in the object program for which tracing is to be done. Otherwise, the whole program is traced.

TRACEIN has a value which is the number of the first input text line on which tracing is to start.

TRACEL has a value which is the number of commands to be executed before tracing is to be started.

TRIMCR instructs HUGO to remove carriage return characters from the end of records before they are processed by the if-you-find sections. NOTRIMCR should always be specified when non-text fields are input to HUGO.

VECS has a value which tells HUGO the maximum number of text lines, headings, and index-items that can appear in the body of a layout. Note that each line in each entry of each table chunk and each line in each division entry is counted as a text line.

The values used unless otherwise specified are selected from:

```
NOCMDT NODOC FILM=2 NOGLIST
HEAP=500 NOJOB NOLINET NOLIST
RECSIZE=1000 NOTRACE TRACEL=0
TRIMCR VECS=1000
```

When using one of the procedures described in section C.1, the PARM field of the run-time step must be referred to as either PARM.GO, or NOPT, in which case the slash preceding the directives must be omitted.

#### C.4 Submitting Jobs Through ATS

The JCL for running a HUGO job or one of the utilities described in this appendix can be submitted to OS from punch cards. However, when the HUGO program and input data have been prepared on ATS, the JCL can be submitted from the terminal. Only a job card has to be added to the JCL given above, and there are only two differences from submitting by cards:

The job name must be of the form Z000nnn where "ooo" is the operator number submitting the job and "nnn" is a three digit number chosen by the submitter to separate his individual jobs from each other. And the job card must be immediately preceded by a line containing

```
Ⓢt jzzzzzzz:dummy
```

The character set used at the terminal is the same as on the keypunch except that: (a) lower case will be converted to upper case, (b) overstruck characters are reduced to their last character, and (c) some characters display differently (in particular, the c cedilla character must be used to represent an equals sign). A sample job card is:

```
//Z0186300 JOB (Q9,T148),HUGO,
// MSGLEVEL=(1,1),MSGCLASS=P,TIME=2
```

Note that the print class P is normally used with jobs on the 148.

To submit a job an operator needs access to the z0 queue and to an intermediate disk file. Assuming operator 186 wishes to submit the job in document "J300", the following sequence of commands would be used:

```
Ⓢx:z0;J300
Ⓢo:disk;z0;a11;186
```

Jobs can be deleted from the z0 queue before being output (by "o") using the queue delete command:

```
Ⓢd:z0;186;J300
```

This command deletes the job J300 belonging to operator 186 from the z0 queue. After the "Ⓢo" command has been executed, only the OS operator can delete a job.

The contents of an ATS queue can be listed. To list the contents of the queue "z0", before it has been emptied by the "Ⓢo" command, the following command can be executed:

```
Ⓢl:z0
```

Jobs submitted by operators can receive data directly back from the computer. Assuming that operator 186 has access to the file called "da001" and that job J300 contains the following JCL card which is used for writing:

```
//SYSPRINT DD DSN=ATS.SPOOL1,DISP=SHR
```

then executing the rewind command:

```
ⓈREW;da001
```

before the job is submitted, and the input command after the job has completed running:

```
Ⓢ~1;da001;po009;z0;a11;186
```

will return the data written on the SYSPRINT card as a message. Any print data set from a job can be returned this way, using any "da" data set. Note that the IBM names for the files are the same as the ATS names except that "da" is replaced by "ATS.SPOOL" and that leading zeroes are removed from the file number.

The ATS procedure facility can be used to substantially reduce the amount of typing required to submit a job.

#### C.5 Stand-Alone Bilingual Merge Program

A program called MERGE exists which takes as input an XPORT-format file and interleaves the French and English paragraphs on that file to produce a single output file, which is likewise in XPORT-format. This program can be used to re-arrange Bilingual text in a form suitable for use with the multi-stream page makeup capability of HUGO. The MERGE program is unable to provide all the error recovery options necessary for such a function, as the text editing system used to prepare input for HUGO is, so this utility is only intended to be used in the absence of other ways of merging text. The user of the program can define those codes which are to be used as interleaving points, those which mark the start of English and French text and those which are to be inserted in the output to record where interleaving was done.

The input to MERGE must consist of, in order:

1. a description of the codes to be used,
2. English text, preceded by a special code which indicates it is such, and
3. the corresponding French text, likewise preceded by an identifying code.
4. Optionally, further pairs of groups of English and French text.

The description, English text and French text occupy the one file. The program uses all the lines up to the first line on which a code is actually used as the description. Each code is defined on a separate line. A code definition line consists of a code class identifier, followed by one or more blanks, followed by the code in question. The code class identifier specifies what function that code will perform. The different identifiers are:

- "1" Tells the program to insert this code in front of English line-up groups.
- "2" Tells the program to insert this code in front of French line-up groups.
- "3" Tells the program to insert this code in front of English line-up groups that have no corresponding French.
- "4" Tells the program to insert this code in front of French line-up groups that have no corresponding English.
- "E" This code appears at the head of the English input text.
- "F" This code appears at the head of the French input text.
- "L" This code marks "line-up" paragraphs in the

input that are expected to have a corresponding paragraph in the other language.

"A" This code marks "line-up" paragraphs expected to have no corresponding paragraph.

"S" This code marks a "sidenote" paragraph. Side notes precede other types of paragraphs, take on the characteristics of those other paragraphs, and are grouped with them for interleaving purposes.

"T" This code marks a "tag" paragraph which is to be grouped with any preceding line-up paragraph for interleaving. Codes not described in the definitions are assumed to be "T" types, and so this code will not normally be used.

0, 1, 2, 3, 4, E and F - type paragraphs must be defined exactly once in the description. Many or no L, A, S and T - type paragraphs may be defined. The program inserts codes defined by 1, 2, 3 and 4 - types. It deletes codes defined by 0, E and F - types. L, A, S and T - types are left alone.

All codes used with MERGE must start with the same character. The normal character used is the asterisk (\*). This may be changed by the following description line:

```
0 */
```

This means that the code delimiter character is to be changed from the asterisk to the slash character.

The JCL for running MERGE without HUGO processing is:

```
// EXEC PGM=MERGE
//STEPLIB DD DSN=TPS.Q.PGM,DISP=SHR
//SYSUT1 DD input file description
//SYSUT2 DD output file description
//SYSUT3 DD SPACE=(CYL,(1,1)),UNIT=SYSDA.
// DCB=(RECFM=VB,LRECL=256,BLKSIZE=6200)
```

### C.6 XPORT Format Files

Three programs exist which support the XPORT format of text files. These programs are available on the TEPB IBM 370/148.

The XPORT format file is simply the data placed on an IBM standard-labeled variable-blocked file with a carriage-return character placed at the end of each line of text. It is the preferred form of input to the current implementation of HUGO.

(In the following,  $\epsilon$  represents the equals sign on the keypunch

XPORT converts an archive tape to an XPORT format file. To run it, code:

```
// EXEC XPORT,V=archive tape number
```

This procedure creates a temporary file called '&&TEXT' on which it places its output. This name can be changed, for example to '&&DATA', by adding the field FN='&&DATA' to the EXEC card. Alternatively, the SYSUT1 DD card can be overridden by a description of the desired output file.

If the field OPT='SPLIT' is added to the EXEC card, and a SYSUT2 DD card describing a second output file is added to the JCL, then the first document on the input archive tape will be put on the SYSUT1 file and the rest of the input will be placed on the SYSUT2 file.

The output file descriptions must either contain their own DCB information (as is done in the procedure for the &&TEXT file) or specify already existing files, as this information is not provided by the program. For compatibility with the physical standards for XPORT format tapes, when tape output is used from XPORT, only one output file should be used, and the SYSUT1 DD card should be coded:

```
//SYSUT1 DD DSN=file name.
// UNIT=3400,DISP=(,KEEP).
// DCB=(RECFM=VB,BLKSIZE=2048,LRECL=235,DEN=3)
```

The other options available with XPORT, which can be coded in the same way as the SPLIT option, are:

TRIMCR, which places records on the output file with no trailing carriage return character. This option is normally not used with HUGO, and converts otherwise empty lines of text into lines containing a single space character. This option is most useful in conjunction with FIX.

FIX, which forces all the records output from XPORT to be of a certain length. It forces this length by truncating longer records and by padding shorter records with space characters. The required length is indicated by placing a  $\epsilon$  after FIX, and following that by the length, as in 'FIX=80'. This option is usually used in conjunction with the TRIMCR option.

DOC, which causes the document name and operator information from the input to be placed in the output for the use of HUGO. The lines containing this information are prefixed by an equals sign. For example, the document 'Doc', belonging to operator 185, submitted by operator 186, and read from archive tape 200999, would be prefixed in the output by the line '=Doc:185;186;200999'. The use of this option by the HUGO run-time system is described with the HUGO run-time directives.

MINDY converts an XPORT format file to an archive tape. To run it, code:

```
// EXEC MINDY,V=archive tape number,
// DOC=document description
```

The document description is mandatory and must be in one of two formats:

```
'document name:operator number'
```

or

```
'document name:operator number;document size'
```

In the first form a single document is placed on the output archive tape with the specified name. In the second, the input file is split into documents of the specified size. The given document name has a sequence number appended to it to generate unique names. In both forms the given operator number is used in generating the archive tape.

The input for MINDY is normally taken from the temporary file '&&TEXT'. This file can be overridden in the same manner as for XPORT.

PUTXPT lists an XPORT format file. To run it code:

```
// EXEC PUTXPT
```

Here, again, the input for printing is taken from the temporary file '&&TEXT', which can be overridden as in XPORT.

### C.7 Other Utilities

Another two utilities which can be used for listing, although not specifically designed for use with XPORT files, are LIST23 and TAPEEDIT. The former is used for listing a file in hexadecimal format, and is invoked by:

```
// EXEC LIST23, FN='file name', PARM=HGD
```

The file name can be any temporary or permanent disk file, and if the number of input records printed needs to be restricted, the letter "L" followed by the *five-digit* number of records can be appended to the PARM field. For example, if only two hundred records are required, the field can be coded "PARM=HGDL00200".

TAPEEDIT is used to print the whole of a magnetic tape, including the labels. It is invoked by:

```
// EXEC TAPEEDIT, V=tape volume serial number
```

The number of records printed may be restricted by specifying the last file and block required. For example, if the last record required is the fifth record in the second file then, assuming that tape 200999 is to be listed, the following JCL would be used:

```
// EXEC TAPEEDIT, Vc200999, EBc5, EFc2
```

### C.8 Updating the Source Library

A source library member may be added to or replaced on the standard HUGO source library by the following procedure:

First the text of the member must be placed on an archive tape, and then XPORT must be used to copy it from there to the library. If the archive tape used is number 200999 and the member name is GEORGE, for example, then the JCL is:

```
// EXEC XPORT, V=200999
//SYSUT1 DD DSN=TPS.HUGOLIB(GEORGE), DISP=OLD
```

This places the member on the testing library. Once tested to satisfaction, it can be copied onto the library used in production, using the following JCL:

```
// EXEC PGM=IEBGENER
//SYSIN DD DUMMY
//SYSPRINT DD SYSOUT=P
//SYSUT1 DD DSN=TPS.HUGOLIB(GEORGE), DISP=SHR
//SYSUT2 DD DSN=TPS.HUGOPROD(GEORGE), DISP=OLD
```

## Appendix D - Font and Character Access

### D.1 Fonts

These are all the fonts and characters currently available for use with the TEPB implementation of HUGO. Fonts 0 and 25 (Perma and Bedford) are available in any size from 5 point to 12.5 point and any combination of height and width within these ranges. All other fonts are available in any size from 5 point to 25 point. But if the set height is between 5 points and 9.9 points, the set width is restricted to be between 5 points and 12.5 points. And if the set height is between 12.6 points and 25 points, the set width is restricted to be between 12.5 points and 25 points. Any setsize specified in a HUGO program outside these ranges will be adjusted by HUGO to the closest approximation within these ranges.

#### Perma (font 0)

This paragraph is set in the Perma utility font which is normally used for setting comments and for line numbering on "proof" copies of composed text. Because of its non-standard size, it is not normally appropriate to use the "Pi" fonts in conjunction with this font. The characters available with this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Times (font 1)

This paragraph is set in the Times font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Times Italic (font 2)

This paragraph is set in the Times Italic font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Times Bold (font 3)

This paragraph is set in the Times Bold font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Times Bold Italic (font 4)

This paragraph is set in the Times Bold Italic font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Modern (font 5)

This paragraph is set in the Modern font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Modern Italic (font 6)

This paragraph is set in the Modern Italic font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Modern Bold (font 7)

This paragraph is set in the Modern Bold font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Modern Bold Italic (font 8)

This paragraph is set in the Modern Bold Italic font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ

#### Excelsior (font 9)

This paragraph is set in the Excelsior font which can be used together with the "Pi" fonts to set most of the characters that a user may need. The characters available in this font are:

ABCDEFGHIJKLMN**OP**QRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$c&:;, "?!%\*( )/ -—|œæEÆ  
 ¼½¾⅞⅓⅔⅕⅖⅗⅘⅙⅚⅛⅜⅝⅞ⅠⅡⅢ





Pi font 22
The following characters are those available in "Pi" font 22, which may be accessed while in one of the usual text fonts:

•†‡\$%&'@#δ∅}∅⊕†‡%oll

Pi font 23
The following characters are those available in "Pi" font 23, which may be accessed while in one of the usual text fonts:

o" \* ★ ◻ / ∅ ∅ ← → ↑ ↓ ……

Pi font 24
The following characters are those available in the Greek "Pi" font, which may be accessed while in one of the usual text fonts:

ΑΒΓΔΕΖΗΘΙΚΑΜΝΞΟΠΡΣΤΤΦΧΨΩ
αβγδεζηθικλμνξοπρσττυφχψωφ

Bedford (font 25)
This paragraph is set in the Bedford mono-width font. This font is usually used with an "sb-ratio" of 60% and an "sb-step" of "60% em" in order to give the effect of type-written or computer-printed text. It is not normally used with the "Pi" fonts. The characters available in this font are:

ABCDEFGHIJKLMN0PQRSTUVWXYZ
abcdefghijklmnpqrstuvwxy
1234567890\$& ; : . , ' ? ! % \* ( ) / - - { } < > +
= | " " - \_ # @ ç Ç « » " " " " [ ] ▶ ◀ ◄ — / - / ~ ' "
éèèâ

TIMES SMALL-CAPS (FONT 26)
THIS PARAGRAPH IS SET IN THE TIMES SMALL-CAPS FONT WHICH CAN BE USED TOGETHER WITH THE "PI" FONTS TO SET MOST OF THE CHARACTERS THAT A USER MAY NEED. THE CHARACTERS AVAILABLE IN THIS FONT ARE:
ABCDEFGHIJKLMN0PQRSTUVWXYZ
ABCDEFGHIJKLMN0PQRSTUVWXYZ
1234567890\$& ; : . , ' ? ! % \* ( ) / - - œ æ Ç Æ
1/8 3/8 1/2 3/4 1/2 3/4 1/2 3/4 Ç « » [ ] † ‡

MODERN SMALL-CAPS (FONT 27)
THIS PARAGRAPH IS SET IN THE MODERN SMALL-CAPS FONT WHICH CAN BE USED TOGETHER WITH THE "PI" FONTS TO SET MOST OF THE CHARACTERS THAT A USER MAY NEED. THE CHARACTERS AVAILABLE IN THIS FONT ARE:
ABCDEFGHIJKLMN0PQRSTUVWXYZ
ABCDEFGHIJKLMN0PQRSTUVWXYZ
1234567890\$& ; : . , ' ? ! % \* ( ) / - - œ æ Ç Æ
1/8 3/8 1/2 3/4 1/2 3/4 1/2 3/4 Ç « » [ ] † ‡

HELVETICA SMALL-CAPS (FONT 28)
THIS PARAGRAPH IS SET IN THE HELVETICA SMALL-CAPS FONT WHICH CAN BE USED TOGETHER WITH THE "PI" FONTS TO SET MOST OF THE CHARACTERS THAT A USER MAY NEED. THE CHARACTERS AVAILABLE IN THIS FONT ARE:
ABCDEFGHIJKLMN0PQRSTUVWXYZ
ABCDEFGHIJKLMN0PQRSTUVWXYZ
1234567890\$& ; : . , ' ? ! % \* ( ) / - - œ æ Ç Æ
1/8 3/8 1/2 3/4 1/2 3/4 1/2 3/4 Ç « » [ ] † ‡

OLD-HELVETICA SMALL-CAPS (FONT 29)
THIS PARAGRAPH IS SET IN THE OLD-HELVETICA SMALL-CAPS FONT WHICH CAN BE USED TOGETHER WITH THE "PI" FONTS TO SET MOST OF THE CHARACTERS THAT A USER MAY NEED. THE CHARACTERS AVAILABLE IN THIS FONT ARE:
ABCDEFGHIJKLMN0PQRSTUVWXYZ
ABCDEFGHIJKLMN0PQRSTUVWXYZ
1234567890\$& ; : . , ' ? ! % \* ( ) / - - œ æ Ç Æ
1/8 3/8 1/2 3/4 1/2 3/4 1/2 3/4 Ç « » [ ] † ‡

### D.2 Input Character Sequences

The following sections describe the predefined character sequences that when set will be interpreted as single actions (see Chapter 6.1). They access characters in four font groups: the "standard" fonts, the "Pi" fonts, the Greek font, and the Bedford font. Standard characters include those accessing such actions as spacing and "tabbing".

At the current time, all defined input sequences consist of either a single character or three characters, the second one of which is the backspace character.

#### D.2.1 Standard Input Sequences

These input sequences can be used in any font which contains the photocomposed characters they represent. The following characters appear on input the same as they do on output:

ABCDEFGHIJKLMNOPQRSTUVWXYZ  
 abcdefghijklmnopqrstuvwxyz  
 1234567890\$%&:;.,?!%\*()-/çÇ□[]

The sequences below on the left when input will produce the characters or actions on the right:

—	—
-<e	-
o<e	æ
a<e	œ
O<E	Œ
A<E	Æ
1<8	1/8
3<8	3/8
5<8	5/8
7<8	7/8
1<4	1/4
1<2	1/2
3<4	3/4
1<3	1/3
2<3	2/3
L<ç	£
C<ç	Ç
(<f	«
)<f	»
(<s	[
)<s	]
d</	†
D</	‡
.<	.
m<s	em space
n<s	en space
t<s	thin space

#### D.2.2 "Pi" Input Sequences

These input sequences can be used when in any font, and will access the appropriate characters from one of the fonts 20, 21, 22 or 23.

m<b	×
d<b	+
p<p	+
t<a	—
l<e	∞
g<e	∞
L<E	∞

G<E	∞
l<t	∞
g<t	∞
p<-	#
e<q	≡
E<Q	≡
n<e	∞
s<r	∞
o<c	∞
O<C	∞
o<r	∞
O<R	∞
o<t	∞
o<o	∞
O<O	∞
p<b	∞
p<d	∞
p<D	∞
p<a	∞
p<m	∞
p<f	∞
p<P	∞
@	@
#	#
m<d	∞
p<	∞
p<(	∞
p<)	∞
o<p	∞
p<t	∞
p<T	∞
%<0	%
p<l	∞
p<L	∞
p<o	∞
p<q	∞
p<Q	∞
p<S	∞
p<*	∞
p<x	∞
p<X	∞
p</	∞
O</	∞
O</	∞
o</	∞
p<w	∞
p<e	∞
p<r	∞
p<n	∞
p<s	∞
t<	∞
n<	∞
T<	∞
N<	∞

#### D.2.3 Greek Characters

These input sequences can be used when in any font, and will access the appropriate characters from the Greek "Pi" font 24.

A<'	Α
B<'	Β
G<'	Γ
D<'	Δ

E	'	.....	E
Z	'	.....	Z
H	'	.....	H
Q	'	.....	Θ
I	'	.....	I
K	'	.....	K
L	'	.....	Λ
M	'	.....	M
N	'	.....	N
X	'	.....	Ξ
O	'	.....	O
P	'	.....	Π
R	'	.....	P
S	'	.....	Σ
T	'	.....	T
U	'	.....	Υ
F	'	.....	Φ
C	'	.....	Χ
Y	'	.....	Ψ
W	'	.....	Ω
a	'	.....	α
b	'	.....	β
g	'	.....	γ
d	'	.....	δ
e	'	.....	ε
z	'	.....	ζ
h	'	.....	η
q	'	.....	θ
i	'	.....	ι
k	'	.....	κ
l	'	.....	λ
m	'	.....	μ
n	'	.....	ν
x	'	.....	ξ
o	'	.....	ο
p	'	.....	π
r	'	.....	ρ
s	'	.....	σ
v	'	.....	ς
t	'	.....	τ
u	'	.....	υ
f	'	.....	φ
c	'	.....	χ
y	'	.....	ψ
w	'	.....	ω
j	'	.....	Ϝ

**D.2.4 Bedford Characters**

These input sequences can be used when in any font, and will access the appropriate characters from the Bedford monowidth font 25. Note that some of these characters are similar to characters in other fonts. However, they are considered distinct characters by HUGO, with their own input sequences.

b<	(	.....	{
b<	)	.....	}
b<	l	.....	<

b<	g	.....	>
b<	p	.....	+
b<	e	.....	#
b<	o	.....	—
b<	O	.....	ε
b<	C	.....	ς
b<	N	.....	ι
b<	u	.....	ι
b<	L	.....	Π
b<	n	.....	#
b<	a	.....	@
b<	b	.....	fixed space
b<	T	.....	▼
b<	B	.....	▲
b<	X	.....	●
b<	E	.....	—
b<	S	.....	/
b<	V	.....	∧
b<	I	.....	∩
b<	t	.....	∩
b<	q	.....	∩
b<	Q	.....	∩
b<	'	.....	ε
b<	'	.....	ε
b<	'	.....	ε
b<	A	.....	à
b<	s	.....	back-space

**D.2.5 Accessing Other Characters**

Some characters in the fonts listed in this appendix cannot be accessed by any of the above input sequences. These other characters can be accessed by use of the internal codes listed in section D.3 together with either the set-absolute or escape-char facilities. Or they can be accessed by setting the following input sequences which, although they cannot be directly input on most input devices, can be generated in a HUGO program by use of the ebcdic or bin operators. The sequences all consist of a single input character, whose "ebcdic" number is given here in lieu of the input character itself.

98	.....	˘ (lower case)
99	.....	˙ (upper case)
100	.....	˚ (lower case)
101	.....	˛ (upper case)
102	.....	˜ (lower case)
103	.....	˝ (upper case)
104	.....	˝ (lower case)
105	.....	˝ (upper case)
112	.....	˝ (lower case)
113	.....	˝ (upper case)
114	.....	˝ (lower case)
115	.....	˝ (upper case)
116	.....	˝ (lower case)
117	.....	˝ (upper case)
118	.....	˝ (lower case)
119	.....	˝ (upper case)
120	.....	1

**D.3 Internal Character Codes**

The following tables detail the internal character representations used on the phototypesetter at TEPB. Each character has a numeric code in the range 1 to 128. The code for any character can be found by summing the numbers at the front of the row and at the top of the column in which the character appears.

Blank or missing entries usually indicate that there is no character in that position. The three exceptions to this rule are:

- character 90 in font 0 is a fixed-width space of 60% em;
- character 96 in font 25 is a fixed-width space of 60% em; and
- character 128 in font 25 is a fixed-width space of -60% em (and so can be used as a "backspace" in the monowidth font).

Most fonts have the same internal layout, which is detailed in section D.3.2. The other sections detail the non-standard layouts.

**D.3.1 Perma Utility (font 0)**

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	a	b	c
30	d	e	f	g	h	i	j	k	l	m
40	n	o	p	q	r	s	t	u	v	w
50	x	y	z	1	2	3	4	5	6	7
60	8	9	0	\$	c	&	:	;	.	,
70	'	'	?	!	%	*	(	)	/	-
80	-	-	ff	fi	fl	fm	=	+	=	
90										

**D.3.2 Standard Fonts (fonts 1 to 19 and 26 to 29)**

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	a	b	c
30	d	e	f	g	h	i	j	k	l	m
40	n	o	p	q	r	s	t	u	v	w
50	x	y	z	1	2	3	4	5	6	7
60	8	9	0	\$	¢	&	:	;	.	,
70	'	'	?	!	%	*	(	)	/	-
80	—	-	1	α	æ	Æ	1/8	3/8	3/4	
90	7/8	1/4	1/2	3/4	1/8	3/8	3/4	3/8	3/4	3/8
100	»	-	-	-	-	-	-	-	-	-
110	]	†	‡							[

**D.3.3 Pi font 20**

	0	1	2	3	4	5	6	7	8	9
0		x	+	+	-	≤	≥	≤	≥	<
10	>	±	=	≡	≠	√				

**D.3.4 Pi font 21**

	0	1	2	3	4	5	6	7	8	9
0		©	©	®	®	-	-	.	.	.
10	-	-	-	Δ	o	o				

**D.3.5 Pi font 22**

	0	1	2	3	4	5	6	7	8	9
0		•	†	‡	\$	δ	♀	¶	@	#
10	∂	©	{	}	⊖	⊕	†	‡	%	/
20	/									

**D.3.6 Pi font 23**

	0	1	2	3	4	5	6	7	8	9
0		°	'	"	*	*	□	■	/	∅
10	∅	∅	←	→	›	↑	↓	.	..	.
20	..									

**D.3.7 Greek Pi (font 24)**

	0	1	2	3	4	5	6	7	8	9
0		Α	Β	Γ	Δ	Ε	Ζ	Η	Θ	Ι
10	Κ	Λ	Μ	Ν	Ξ	Ο	Π	Ρ	Σ	Τ
20	Υ	Φ	Χ	Ψ	Ω	α	β	γ	δ	ε
30	ζ	η	θ	ι	κ	λ	μ	ν	ξ	ο
40	π	ρ	σ	ς	τ	υ	φ	χ	ψ	ω
50	ϕ									

**D.3.8 Bedford Monowidth (font 25)**

	0	1	2	3	4	5	6	7	8	9
0		A	B	C	D	E	F	G	H	I
10	J	K	L	M	N	O	P	Q	R	S
20	T	U	V	W	X	Y	Z	a	b	c
30	d	e	f	g	h	i	j	k	l	m
40	n	o	p	q	r	s	t	u	v	w
50	x	y	z	1	2	3	4	5	6	7
60	8	9	0	\$	¢	&	:	;	.	,
70	'	'	?	!	%	*	(	)	/	-
80	-	{	1	}	<	>	+	=		"
90	"	-	-	Π	#	@	.	ç	Ç	«
100	»	-	-	-	-	-	-	-	-	[
110	]	►	◄	●	—	/	-	!	~	'
120	"	é	è	ê	à					

## Appendix E - Sample Program

```

/ This program was used to typeset this manual in the format in which
you are now reading it.
- include ROMAN
  / to include the function defined in Chapter 4.4.1

function count starts-as 0 with a, b
  local p
  loop
    assign a index b to p
    exit-if p gt length(a)
    add 1 to count
    assign a start-at p plus 1 to a
  end-loop

global group1 starts-as 'N'
global in-preface-page starts-as 'Y'
global page-number starts-as 0
global preface-page-number starts-as 0

list toc
global toc-size starts-as 0
define enter-toc with head, level
  index-item level * 'm<s' : head
  add 1 to toc-size
  assign index-item : '*' : page-number to toc(toc-size)
end-index
at-end
  local n starts-as 1, h, p, l
  assign 'Y' to in-preface-page
  finish-body
  column 0
  head width 6.5 in
  setsize 15 on 18 bold
  centre nohyph
  set 'Table of Contents'
  columns 0, 3.5 in
  loop
    exit-if n gt toc-size
    assign toc(n) to h
    assign 0 to l
    assign 1 to p
    loop
      exit-if p plus 2 gt length(h)
      exit-if h(p,3) isnt 'm<s'
      add 1 to l
      add 3 to p
    end-loop
    if l gt 0
      tag
    else
      text
    end-if
    justify-with '.4.', 1 en
    assign h index '*' to p
    left-indent 0, (1 plus 2) ems
    right-indent 2 ems for-all
    set h(1,p minus 1)
    tag
    no-para-space
    setsize 1 em on 0
    flush-right
    set h start-at p plus 1
    add 1 to n
  end-loop

iuf '<*>'
  set '*'
iuf sl & '*p'

```

```

text
iuf sl & '*pc '
text
para-space 2 pt
iuf '*pe '
text block
para-space 12 pt
iuf sl & '*h0 ' & (chars called head)
column 0
reference-head chapter
quad 1 em
head setsize 25 on 30 bold
width 6.5 in
centre nohyph
set head
columns 0, 3.5 in
iuf sl & '*h1 ' & (chars called head)
assign 'N' to in-preface-page
finish-body
column 0
reference-head chapter
quad 1 em
reference-head chapter
width 6.5 in
italic flush-left
set head
enter-toc head, 0
head
setsize 15 on 18 bold
width 6.5 in
centre nohyph
set head
columns 0, 3.5 in
iuf sl & '*h2 ' & (chars called head)
finish-body
enter-toc head, 1
head bold
nohyph
set head
iuf sl & '*h3 ' & (chars called head)
enter-toc head, count(head(1,head index ' ' minus 1),'')
head para-space 1 ex bold
nohyph
set head
iuf sl & '*h4 '
head para-space 1 ex bold
nohyph
iuf sl & '*preface'
text para-space 2 ex
iuf sl & '*s1'
text no-para-space
iuf sl & '*li ' & within(' ') & '- '
sub-para
indent 1 em, 2 ems
set '-'
quad-to left-indent(2)
iuf sl & '*li ' &
within(' ') & ((' & without(')) & ')' called a) & ''
sub-para
indent 1 em, 3 ems
set a
quad-to left-indent(2)
iuf sl & '*li '
sub-para
indent 1 em, 2 ems
iuf sl & '*st1'
text
table
t 0, 1.4 in
t 0, 3 in drop-align indent 1.5 in, .5 in quad-out
end-table
iuf sl & '*st2'

```

```

text
table width 1.4 in
  t 0 flush-left-with '.<.', 1 thin
  t 1.6 in flush-left
end-table
iuf sl & '*st3'
text block para-space 1 ex setsize 10 on 12
table width .25 in centre
  t 0, .4 in flush-right times 1 em
  t .5 in
  t .75 in
  t 1 in
  t 1.25 in
  t 1.5 in
  t 1.75 in
  t 2 in
  t 2.25 in
  t 2.5 in
  t 2.75 in
end-table
iuf sl & '*til '
chunk
iuf sl & '*ti' & digits & ' '
tab
iuf sl & ('*el' ! '*et' & digits)
iuf sl & ('*sg1' ! '*sg3')
text
widow 4 flush-left
sb-ratio 40% nohyph
assign 'Y' to group1
iuf sl & '*sg2'
tag
widow 4 flush-left
font 25, 7.5 on 9
sb-ratio width-of('m') div-by 1 em nohyph
assign 'Y' to group1
iuf sl & '*sg4'
text block
widow 4 flush-left
width 1.4 in
indent 0, 1 em
division 0, 1.6 in
assign 'T' to group1
iuf sl & '*sg5'
column 1 in
text width 4.5 in
widow 4 flush-left
font 25, 7.5 on 9
sb-ratio width-of('m') div-by 1 em no-hyph
assign 'Y' to group1
iuf sl & '*eg' & digits
assign 'N' to group1
iuf sl & within(' ') called s
if group1 is 'Y'
  sub-para
  quad length(s) ens
else-if group1 is 'T'
  tab
else
  spaceband
end-if
iuf '*hi0.'
hyph times 1 em break
iuf '*hi2.'
italic medium
iuf '*hil.' ! '*hi3.'
font 17, 1 em
nohyph no-break

iuf '*null'
iuf '*fudge' & (within('-0123456789.')) called f)

```

```

        fudge f em
iuf '*font' & (1 digit & digits called f)
        & ',' & (1 digit & within(digit-string:'.') called sh)
        & ',' & (1 digit & within(digit-string:'.') called sw)
        save it
        font f,sh,sw
iuf '*restore'
        restore it
iuf '*slant' & (1 digit & digits called s)
        slant s
iuf '*baseline' & (within('-0123456789.') called b) & '*'
        baseline b ex
iuf '*time'
        set time-of-day
iuf '*date'
        set date
iuf '*julian'
        set julian-date
iuf '*e' & (digits called n) & '*'
        set n ebcdic
iuf '*jr' flush-right
iuf '*qw' & (without(', ') called c) & ',' & (digits called n) & '*'
        quad-with c, n% ems
iuf '*in' & (digits called m) & ',' & (digits called n) & ',' &
        (digits called o) & ',' & (digits called p) & '*'
        left-indent m pt, n pt
        right-indent o pt, p pt
iuf '*fmr' foot-ref
iuf '*fnd'
        foot-note
        para-space 2 ex
        setsize 8 on 9
iuf '*un' underline
iuf '*nu' no-underline
iuf '*fl' finish-line
iuf '*qr' quad-with-rule
iuf '*HH'
        figure 20 pt
        fig-rule 0, -5 pt, 5 pt, 5 pt
        fig-rule 5 pt, -3 pt, 10 pt, 1 pt
        fig-rule 15 pt, -5 pt, 5 pt, 5 pt
        end-figure
iuf '*box'
        rule -2 pt, -1 ex, width plus 4 pt, 1 pt
        vertical-rule -3 pt, 1 pt, -1 ex, 3 pt
        vertical-rule width plus 2 pt, 1 pt, -1 ex, 3 pt
        end-para-rule -2 pt, 2 pt, width plus 4 pt, 1 pt
iuf '*charlist' & (digits called m) & ',' & (digits called n) & '*'
        set ('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz':
        <1234567890$&:;.,'?!%*()->:
        106ebcdic:120ebcdic:81ebcdic:82ebcdic:83ebcdic:84ebcdic:
        70ebcdic:71ebcdic:72ebcdic:73ebcdic:85ebcdic:86ebcdic:
        87ebcdic:88ebcdic:89ebcdic:121ebcdic:'çç«»':
        112ebcdic:113ebcdic:114ebcdic:115ebcdic:116ebcdic:117ebcdic:
        118ebcdic:119ebcdic:'[]':
        32ebcdic:33ebcdic:34ebcdic:35ebcdic:36ebcdic:37ebcdic:
        38ebcdic:39ebcdic:40ebcdic:48ebcdic:49ebcdic:50ebcdic:
        51ebcdic:52ebcdic:53ebcdic:54ebcdic:55ebcdic:56ebcdic)
        (m,n)
iuf '*sn' & (digits called n) & '*'
        assign n minus 1 to page-number
iuf '* ' & without(' .') called it
        output it : ' on line ' : line
        save status
        font 19, 1 em
        set it
        restore status
initially
        width 3 in
        para-space 5 pt
        times 9 on 10
        columns 0, 3.5 in

```



```
body-space 1 pica
first-layout page
layout page 8.5 in by 11 in
width 8.5 in italic
at 0, 0 quad-with-rule set ' cut here'
width 6.5 in
at 1 in, .5 in flush-right
if in-preface-page is 'Y'
  add 1 to preface-page-number
  set roman-numeral(preface-page-number)
else
  add 1 to page-number
  roman set page-number italic
end-if
at 1 in, lin body 9 in
at 1 in, .5 in use-first chapter
width 8.5 in
at 0, 10.75 in set line
end-HUGO
```

## Index

- account A.4; C.2  
 add 3.1  
 <add statement> 3.1; 3  
 alpha B.1  
 alphas 11.4.2; B.1  
 alpha-string 2.3.3  
 and 2.2, 11.4; 2.2.1, 11.4.1, B.1  
 any-ex-notes 9.2.5  
 area-count 9.5.2  
 area-size 9.5.2  
 assign 3.1  
 <assignment statement> 3.1; 3  
 at 9.1.2; 7.1, 8.3.3, 9.2.1, 9.4, 9.5.1, 10.4  
 at-end 11.3  
 <at-end section> 11.3; 5.2  
 at-job-end 5.2.2; A.6, C.2  
 <at-job-end section> 5.2.2; 5.2  
 at-job-start 5.2.2; C.2  
 <at-job-start section> 5.2.2; 5.2  
 at-page-end 5.2.2; C.2  
 <at-page-end section> 5.2.2; 5.2  
 at-page-start 5.2.2; C.2  
 <at-page-start section> 5.2.2; 5.2  
 at-start 11.2  
 <at-start section> 11.2; 5.2  
 attn A.1  
 <attribute set name> 8.1.2  
 at-y 9.5.1  
 a0 9.5.3  
 a1 9.5.3  
 a2 9.5.3  
 a3 9.5.3  
 a4 9.5.3  
 a5 9.5.3  
 a6 9.5.3  
 a7 9.5.3  
 a8 9.5.3  
 baseline statement: 6.2.3; builtin: 6.4.2; 8.6  
 bedford statement: 6.2.1; builtin: 6.4.2  
 bin A.1; B.1, D.2.5  
 bks 2.3.3  
 block 8.2.2  
 body 9.2.1; 8.5.2, 8.7, 9.1.2, 9.2, 9.2.4, 9.2.5, 9.5.1  
 body-space 9.2.3  
 bold statement: 6.2.2; builtin: 6.4.2; 6.1.3  
 <Boolean built-in function> 9.2.5  
 <Boolean built-in function call> 2.3.2, 3.4.1, 6.4.2,  
 7.7, 8.2.2, A.6; 2.2  
 <Boolean expression> 2.2; 2.3.2, 3.2, 3.3  
 <Boolean factor> 2.2, 2.2.2, 2.2.3, 4.1.4; 11.4  
 <Boolean term> 2.2  
 bottom-align 8.4.3  
 bottom-previous 8.4.3  
 break statement: 7.4; builtin: 7.7  
 break-here 7.4  
 <built-in function call> 2.3.1, 2.3.3, 3.4.1, 3.4.3,  
 4.1.4, 4.4.2, 6.4.2, 7.7, 8.2.4, 8.4.4, 9.4, 9.5.1,  
 9.5.2, 9.5.3, 10.6, 12.1, 12.2, A.1, A.2, A.3, A.4,  
 A.5, A.7; 2.1  
 by B.1  
 called B.1  
 carried-head 8.3.2; 8.2.3, 10.5  
 <carried head name> 8.3.2  
 cat 2.1; 2.1.3, 4.4.1, B.1  
 center B.1  
 center-align B.1  
 centre 7.2; B.1  
 centre-align 8.4.3; B.1  
 char B.1  
 <character> 1.1.2  
 <character format statement> 6.1.2, 6.1.4, 6.2.1,  
 6.2.2, 6.2.3, 6.2.4, 6.3, 10.1.1, 10.3, A.2; 3  
 char-count 12.1  
 char-in-page 12.1  
 chars 11.4.2; B.1  
 chars-input 12.1  
 char-string 2.3.3  
 chunk 8.4.2; 8, 8.2, 8.2.1, 8.4.1, 8.8  
 cm 6.4.1; B.1  
 column B.1  
 <column position> 9.3.3; 9.3.1, 9.3.2  
 columns statement: 9.3.1; builtin: 9.5.1; 8.6, 9.2.2,  
 9.2.3, 9.3.2, B.1  
 <conditional statement> 3.2; 3  
 <constant> 1.1.2; 2.1  
 continued-head 8.3.2; 8, 8.2.3, 10.5  
 cpu-time A.5  
 cut-line 8.3.1  
 date 12.2  
 <declaration section> 4; 5.1  
 define 4.3; 4.1.1  
 <digit> 1.1.2; 1.2  
 digit B.1  
 digits 11.4.2; 11.4.4, B.1  
 digit-string 2.3.3  
 discretionary 7.4  
 display-head 8.3.1  
 div B.1  
 div-by 2.1; 2.1.3, B.1  
 division 8.8; 8, 8.2, 8.2.1  
 document-line A.3; C.3  
 drop-align 8.4.3  
 ebcdic A.1; B.1, D.2.5  
 ei 11.4.3  
 else 3.2  
 else-if 3.2  
 em B.1  
 ems 6.4.1; B.1  
 en B.1  
 end-figure 10.3  
 end-hugo 5.1  
 end-if 3.2  
 end-index 9.4  
 end-loop 3.3  
 end-para-rule 10.5  
 end-segment 5.3  
 end-table 8.4.1  
 english statement: 7.5; builtin: 7.7  
 ens 6.4.1; B.1  
 entry 4.1.1, 4.1.3, 4.2, 4.3; 5.3  
 eof 3.4.1  
 eq 2.2.3; B.1  
 error A.6  
 escape-char A.2; D.2.5

- even 2.2.3; B.1  
 evenup 9.2.2  
 evenup-step statement: 9.2.2; builtin: 9.5.1  
 ex 6.4.1; 10.1.1, B.1  
 excelsior statement: 6.2.1; builtin: 6.4.2  
 exit-if 3.3  
 ex-note 8.7  
 ex-note-body 9.2.5; 8.7  
 ex-note-column B.1  
 ex-note-columns 9.2.5; B.1  
 ex-note-head 8.7  
 ex-note-lineup 9.2.5  
 ex-note-ref 8.7  
 <expression> 2.1; 2.2.2, 2.2.3, 2.3.1, 3.1, 3.4.2, 3.4.3, 3.4.4, 4.1.1, 4.1.2, 4.1.3, 4.1.4, 4.2, 4.3, 4.4.2, 6.1.2, 6.1.3, 6.2.1, 6.2.3, 6.3, 6.4.2, 7.1, 7.3.1, 7.3.2, 7.4, 7.5, 7.6, 7.7, 8.2.2, 8.2.3, 8.4.1, 8.6, 8.8, 9.1.1, 9.1.2, 9.2.1, 9.2.2, 9.2.3, 9.2.4, 9.2.5, 9.3.2, 9.3.3, 9.4, 9.5.2, 10.1.1, 10.1.2, 10.2, 10.3, 10.4, 10.5, 11.4, 11.4.2, 11.4.3, 11.5, A.1, A.2, A.4, A.6  
 <factor> 2.1, 4.1.4, 6.4.1, 9.5.2  
 fig-rule 10.3  
 figure 10.3  
 film-length 12.1  
 finally 5.2.1  
 <finally section> 5.2.1; 5.2  
 finish 9.2.1; 9.2.2, 9.2.4, 11.3  
 finish-body 9.2.1; 9.2.2, 9.2.4  
 finish-column 9.2.1  
 finish-line 7.1; 7.3.2  
 first-layout 9.1.3; 9.1.1  
 <first-layout section> 9.1.3; 5.1  
 floor 2.3.1  
 flush-left 7.2; 7.3.1  
 flush-left-with 7.3.2; 10.2  
 flush-left-with-rule 10.2  
 flush-right 7.2  
 followed-by 11.4.4  
 font statement: 6.2.1; builtin: 6.4.2  
 foot-balance 8.5.2  
 foot-bottom 8.5.2  
 foot-head 8.5.1  
 foot-lineup 8.5.2  
 footnote 8.5.1  
 footnote-count 9.5.2  
 foot-ref 8.5.1; 8.6  
 for 7.6; 7.7  
 for-all 7.6  
 force-justify 7.2  
 french statement: 7.5; builtin: 7.7  
 fudge statement: 6.2.3; builtin: 6.4.2  
 function 4.2; 4.1.1, 8.4.1  
 <function call> 2.1, 4.2  
 <function declaration section> 4.2; 4  
 <function name> 1.2.2; 2.1, 4.2  
 galley 9.2.4  
 ge 2.2.3; B.1  
 get 3.4.3; 3.4.4, 9.4, C.2  
 global 4.1.1  
 <global declaration section> 4.1.1; 4  
 <global variable> 1.2.2; 4.1.1, 11.4  
 <global variable declaration> 4.1.1  
 gt 2.2.3; B.1  
 head 8.3.1; 8.2.3  
 helvetica statement: 6.2.1; builtin: 6.4.2  
 <HUGO program> 5.1; 0.5  
 hyph statement: 7.5; builtin: 7.7  
 hyphen statement: 7.5; builtin: 7.7  
 hyph-fill statement: 7.5; builtin: 7.7  
 hyph-margins 7.5  
 <identifier> 1.2; 1.2.2, 4.3, 8.1.2, 8.3.2, 8.3.3, 9.1.1  
 if 3.2  
 if-you-find 11.1; 6.1.2, 6.1.3, 8.4.4, 11.2, 11.4, 11.4.3, 11.4.4, 11.4.5, 11.4.7, 11.5, 12.1, B.1  
 <if-you-find section> 11.1; 5.1  
 in B.1  
 inch 1.2.3, B.1  
 inches 6.4.1; 1.2.3, B.1  
 indent B.1  
 <indentation margins> 7.6  
 index 2.1; 2.1.3, B.1  
 index-item 9.4; C.3  
 index-x 9.4  
 index-y 9.4  
 initially 5.2.1; 8.2  
 <initially section> 5.2.1; 5.2  
 input 3.4.1; 3.4.3  
 input-volume A.3; C.3  
 inter-para 8.3.1  
 into-column 9.3.2; 9.5.1  
 is 2.2.2; 4.4.1, B.1  
 is-in 4.1.4; B.1  
 isnt 2.2.2; B.1  
 isnt-in 4.1.4; B.1  
 italic statement: 6.2.2; builtin: 6.4.2; 6.1.3  
 iuf B.1  
 job A.4; C.3  
 job-name A.4  
 job-number A.4  
 julian-date 12.2  
 justify 7.2  
 justify-center B.1  
 justify-centre 7.2; B.1  
 justify-right 7.2  
 justify-vertically 9.2.2  
 justify-with 7.3.2; 10.2  
 justify-with-rule 10.2  
 kill-all-heads 8.3.2  
 kill-head 8.3.2  
 ladder 7.5  
 landscape 9.1.1; 9.5.3  
 layout 9.1.1  
 layout-depth 9.5.1  
 <layout name> 9.1.1; 9.1.3  
 <layout section> 9.1.1; 5.1  
 <layout shape> 9.1.1  
 layout-width 9.5.1  
 lc 2.3.1  
 lc-alpha B.1  
 lc-alphas 11.4.2; B.1  
 lc-alpha-string 2.3.3  
 le 2.2.3; B.1  
 leading 8.2.3  
 leave 8.2.1, 8.3.1, 8.3.2, 8.3.3, 8.4.2, 8.5.1, 8.6, 8.7; 8, 8.1.1  
 left-for 7.7  
 left-indent statement: 7.6; builtin: 7.7; B.1  
 length 2.3.1  
 <letter> 1.2

## Index

- lex-eq B.1
- lex-ge 2.2.2; B.1
- lex-gt 2.2.2; B.1
- lex-le 2.2.2; B.1
- lex-lt 2.2.2; B.1
- lex-ne B.1
- line 12.1; A.3
- line-count 12.1
- <line format statement> 7.1, 7.2, 7.3.1, 7.3.2, 7.4, 7.5, 7.6, 10.2
- line-up 9.3.2; 9.2.5
- list 4.1.3; C.3
- <list declaration section> 4.1.3; 4
- <list name> 1.2.2; 4.1.3, 4.1.4, 11.4
- <list name declaration> 4.1.3
- list-size 4.1.4
- <list variable> 4.1.3; 1.2.2, 11.4
- literally 4.1.1, 4.1.2
- local 4.1.2
- <local declaration group> 4.1.2; 4.2, 4.3, 5.2.1, 5.2.2, 6.1.3, 9.1.1, 11.1, 11.2, 11.3
- <local variable> 1.2.2; 4.1.2, 4.2, 11.4
- <local variable declaration> 4.1.2
- long 9.2.1; 9.2.4
- loop 3.3
- lpos 11.4.3
- lt 2.2.3; B.1
- it 11.4.4
- many 8.4.4
- max-column 9.5.1
- max-depth 9.2.2
- maximum-expansion 9.5.1
- max-width statement: 6.2.2; builtin: 6.4.2
- medium 8.2.2
- min-para 2.1; 2.1.3, B.1
- minus 7.5
- min-word 6.4.1; B.1
- mm statement: 6.2.1; builtin: 6.4.2
- modern 2.1; 2.1.3, B.1
- modulo B.1
- mul 2.1; 2.1.3, B.1
- mul-by 9.5.1
- multi-column 9.3.2; 8.6, 9.2.2, 9.2.3
- multi-columns 2.2.3; B.1
- ne 9.2.1
- new-body 9.1.3; 9.1.1, 9.2.5
- next-layout 11.4.5
- next-line 7.4
- no-break A.2
- no-escape-char 9.2.2
- no-evenup 8.7
- no-ex-note-head 9.2.5
- no-ex-note-lineup 8.7
- no-ex-note-ref 8.5.2
- no-foot-balance 8.5.2
- no-foot-bottom 8.5.1
- no-foot-head 8.5.2
- no-foot-lineup 7.5
- nohyph 9.2.2
- no-maximum-expansion 11.4.5
- no-next-line 8.2.3; 8.2.4
- no-para-space statement: 6.2.4; builtin: 6.4.2
- normal-case 9.2.1
- normalize-page 6.1.4
- no-shift 10.1.2
- no-side-line B.1
- no-stick-hyphens 8.2.3; 8.2.4
- no-suffix-space 2.3.2
- not 11.4.4
- not-followed-by 10.1.1
- no-underline 1.1.2
- number statement: 4.4.1; builtin: 4.4.2; 4.4
- <numeric constant> 2.2.3; B.1
- odd 2.1; 2.1.3, B.1
- of statement: 6.2.1; builtin: 6.4.2
- oid-helvetica 8.2.3; 8.2.4, 9.1.2
- on 6.1.3
- only 11.4.4
- optional 2.2, 11.4; 2.2.1, 11.4.1, 11.4.7, B.1
- or 3.4.2; 4.4.1
- output 3.4.2; 3
- <output statement> A.3
- output-volume 8.6
- overflow B.1
- p A.1; B.1
- pack 12.1; 9.2.1
- page-count 8.3.3, 8.5.2, 8.6, 9.1.2, 9.1.3, 9.2.1, 9.2.2, 9.2.3, 9.2.4, 9.3.1, 9.3.2, 9.4, 10.4; 3
- <page format statement> 9.2.5
- <page format statement> 8.1.2, 8.2.1, 8.2.2, 8.2.3, 8.3.2, 8.4.2, 8.4.3, 8.5.1, 8.6, 8.7, 8.8, 10.1.2, 10.5; 3
- <paragraph start statement> 8.3.1, 8.3.2, 8.3.3, 8.5.1, 8.6; 3
- <paragraph start statement> 8.7
- <parameter declaration> 4.2; 4.3
- para-space statement: 8.2.3; builtin: 8.2.4; 9.1.2
- <pattern built-in function call> 11.4.2, 11.4.3, 11.4.4, 11.4.5; 11.4
- <pattern expression> 11.4; 11.1, 11.4.4, 11.4.5
- <pattern factor> 11.4, 11.4.2
- <pattern subexpression> 11.4
- <pattern term> 11.4
- percent 2.1; 2.1.3, B.1
- perma statement: 6.2.1; builtin: 6.4.2
- pica B.1
- picas 6.4.1; B.1
- plus 2.1; 2.1.3, B.1
- point B.1
- points 6.4.1; B.1
- portrait 9.1.1; 9.5.3
- pp 6.4.1; B.1
- <program section> 5.1; 5.3
- pt B.1
- pts B.1
- put 3.4.4; A.1, C.2
- <put statement> 3.4.4; 3
- quad 6.3
- quad-out 7.3.2; 6.3
- quad-to 6.3
- quad-with 7.3.2; 6.3, 10.2
- quad-with-rule 10.2; 6.3
- <quoted constant> 1.1.2
- reference-head 8.3.3; 8
- <reference head name> 8.3.3
- <repetitive statement> 3.3; 3
- rescan 11.5; C.3
- <rescan statement> 11.5; 3
- reserve 9.2.1

- restore 8.1.2
- restore-defaults 8.1.2
- right-for 7.7
- right-indent statement: 7.6; builtin: 7.7
- roman statement: 6.2.2; builtin: 6.4.2
- rpos 11.4.3
- rule 10.4, 10.5
- rule-height 10.6
- rule-posn 10.6
- rule-specs 10.2; 10.6
- save 8.1.2
- save-defaults 8.1.2; 8.2.1
- sb-ratio statement: 7.3.1; builtin: 7.7; 7.4
- sb-step statement: 7.3.1; builtin: 7.7
- seconds 12.2
- <segment> 5.3; 5.1
- segment 5.3
- set 6.1.2; 4.4.1, 6.1, 6.1.1
- set-absolute A.2; D.2.5
- setsize 6.2.1; 6.2.3, 8.2.3
- <set-up section> 5.2; 5.1
- setwidth 7.4
- shift-down 6.1.4
- shift-up 6.1.4
- side-line 10.1.2; 10.5
- side-line-vary 10.1.2; 10.5
- sidenote 8.6
- side-ref 8.6
- sl 11.4.3
- slant statement: 6.2.3; builtin: 6.4.2
- small-caps statement: 6.2.4; builtin: 6.4.2
- source-line A.7
- spaceband 7.3.1
- space-block 8.2.3
- start-at 2.1; 2.1.3, B.1
- starts-as 4.1.1, 4.1.2, 4.2
- <statement> 3, A.4, A.6; 3.2, 3.3, 4.2, 4.3, 5.2.1, 5.2.2, 6.1.3, 8.4.1, 9.1.1, 9.4, 11.1, 11.2, 11.3, 4.1.2, 4.2
- static 7.5
- stick-hyphens 5.4; C.2
- stop 5.4; 3
- <stop statement> 5.4; 3
- string statement: 4.4.1; builtin: 4.4.2; 4.4
- sub-body 9.2.1; 9.2.3
- sub-entry 8.4.2
- <subexpression> 2.1, A.1
- <subfactor> 2.1, 6.4.1, A.1; 9.5.2
- sub-para 8.2.1; 8, 8.2, 8.2.2
- <subscription> 2.1
- <subterm> 2.1
- suffix-space statement: 8.2.3; builtin: 8.2.4; 9.1.2
- t B.1
- tab statement: 8.4.2; builtin: 2.3.3; 8.8
- table 8.4.1
- <table column setup> 8.4.1
- <table definition statement> 8.4.1; 3
- tabular-column 8.4.4
- tabular-count 8.4.4
- tabular-position 8.4.4
- tag 8.3.1
- tccl 8.4.1; 4.1.2, 8.4.2, B.1
- <term> 2.1; A.1
- text 8.3.1
- th-elem 4.1.4; B.1
- then 3.5; 1.2.3, B.1
- <then statement> 3.5; 3
- thin 7.3.2, B.1
- thins 6.4.1; B.1
- time-of-day 12.2
- times statement: 6.2.1; builtin: 6.4.2
- to 3.1
- to-global 11.4; 4.1.1, 11.4.7, B.1
- to-local 11.4; 11.4.7, B.1
- to-lpos 11.4.3
- top-align 8.4.3
- top-flush 9.2.2
- top-previous 8.4.3
- to-rpos 11.4.3
- total-set 12.1
- turn-over 7.1
- <type> 4.4.1; 4.1.1, 4.1.2, 4.1.3, 4.2
- uc 2.3.1
- uc-alpha B.1
- uc-alphas 11.4.2; B.1
- uc-alpha-string 2.3.3
- unbin A.1
- unblock 8.2.2
- underline 10.1.1
- underline-height 10.6
- underline-posn 10.6
- underline-specs 10.1.1; 10.6
- unpack A.1
- up-to 11.4.4
- use-first 8.3.3
- use-last 8.3.3
- user-code 6.1.3
- <user-code section> 6.1.3; 5.1
- <user declaration section> 4.3; 4
- <user statement> 4.3; 3
- <user statement name> 4.3
- <variable> 1.2.2; 2.1, 3.1
- vari-space 7.3.1
- verify 2.2.2; B.1
- vertical-rule 10.5
- vertical-rule-vary 10.5
- warning A.6
- widow 8.2.2
- width statement: 7.1; builtin: 7.7; 7.4
- width-of 6.4.2
- with 4.2, 4.3, 9.3.3
- within 11.4.2; 11.4.4, B.1
- without 11.4.2; 11.4.4, B.1
- x B.1
- x-posn 9.5.2
- y-posn 9.5.2; B.1