

UniPlus+®

System V, Release 2

Volume 6

Programming Tools Guide



Copyright © 1985 by UniSoft Systems

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, translated, transcribed or transmitted in any form or by any means manual, electronic, electro-magnetic, optical, or otherwise, without explicit written permission from UniSoft Systems.

While UniSoft Systems has endeavored to exercise care in the preparation of this guide, it nevertheless makes no warranties of any kind with regard to the documentation contained herein, including no warranty of merchantability or fitness for a particular purpose. In no event shall UniSoft be liable for incidental or consequential damages in connection with or arising out of the furnishing, performance, or use of any of this documentation.

This guide was edited and enhanced by Mary Ann Finnerty of UniSoft Systems.

UniPlus⁺ and UniSoft are registered trademarks of UniSoft Systems.
UNIX is a trademark of AT&T Bell Laboratories.

CONTENTS

Chapter 1	AWK
Chapter 2	SED
Chapter 3	LEX
Chapter 4	YACC
Chapter 5	BC
Chapter 6	DC
Chapter 7	M4
Chapter 8	MAKE
Chapter 9	AUGMAKE
Chapter 10	SCCS
Chapter 11	CURSES
Chapter 12	UUCP

1

2

3

Chapter 1: AWK

CONTENTS

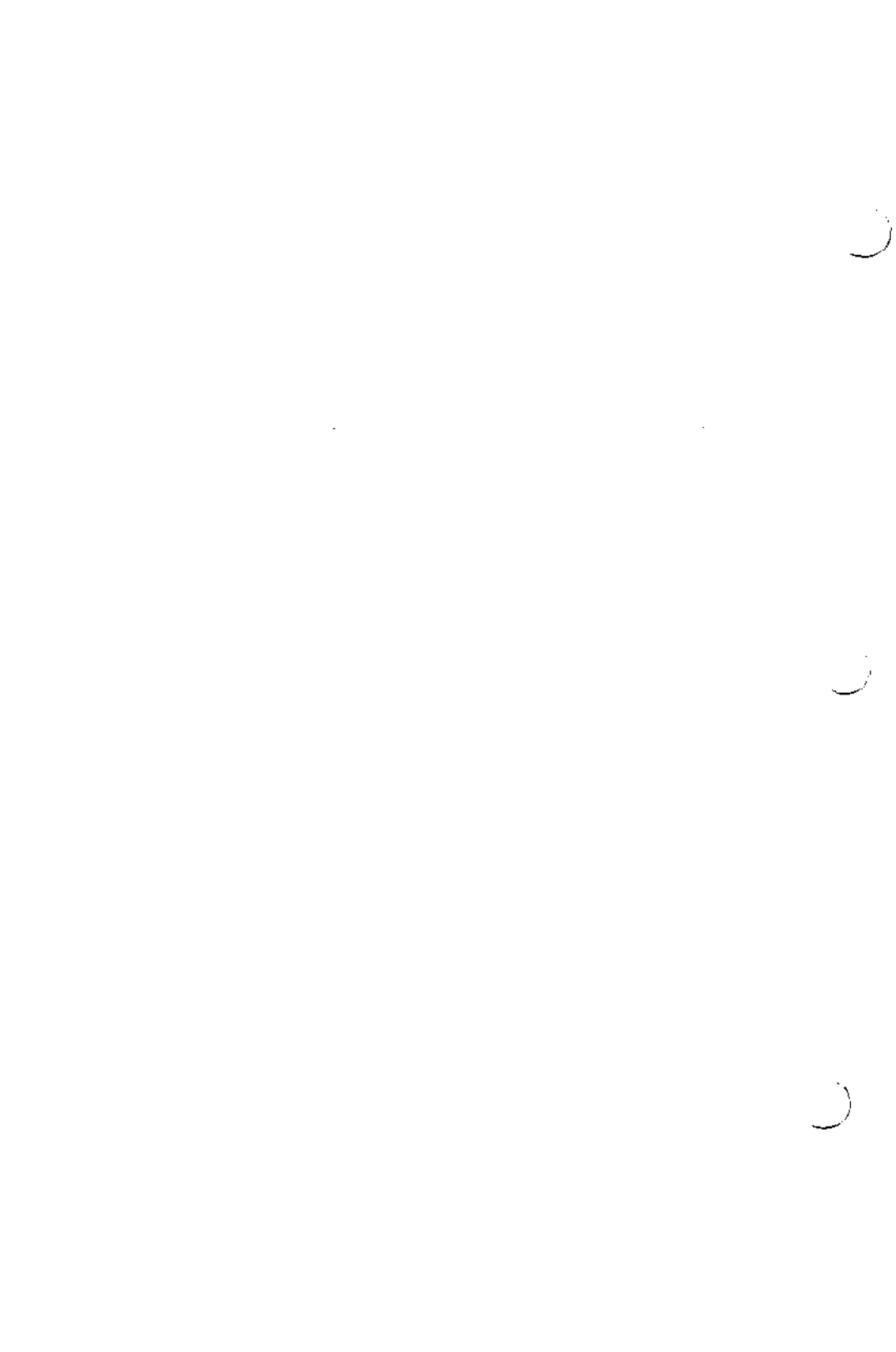
1. Introduction	1
2. General Structure	3
3. BEGIN and END Statements	4
4. Comments	7
5. Keywords	8
6. Identifiers	9
6.1 Variables	9
6.1.1 Initialization	11
6.1.2 Type	11
6.1.3 Incremented Variables	13
6.1.4 Special Variables	13
6.2 Arrays	14
6.2.1 Multidimensional Arrays	16
7. Operators	17
7.1 Binary Operations	19
7.2 Unary Operations	20
7.3 Regular Expressions	20
7.4 Relational Expressions	23
7.5 Assignment Expressions	25
7.6 Variables, Expressions and Assignments	27
8. Input Records and Fields	29
8.1 Tokens	30
8.2 Fields	31
8.3 Field Separator	32
8.4 Record Separator	33
8.5 Output Separators	34
8.6 Multiline Records	35
8.7 Ranges	36
9. Numeric Constants	37

10. String Constants	38
10.1 String Concatenation	39
11. Functions	41
11.1 Summary of Built-in Functions	44
12. Direct Command-line Usage	48
12.1 Cooperation with the Shell	52
13. Using Command Files	54
14. Control Flow Statements	56
14.1 if-else	56
14.2 while Statement	56
14.3 for Statement	57
14.4 break, continue and next Statements	60
15. Report Generation	61
15.1 Output to Printer or Terminal	62
15.2 Output to Files	68
15.3 Output to Pipes	70

LIST OF FIGURES

Figure 1.1. Keywords	8
Figure 1.2. Assignment Operators	17
Figure 1.3. Arithmetic Operators	18
Figure 1.4. Relational Operators	18
Figure 1.5. Logical Operators	18
Figure 1.6. Regular Expression Pattern Matching Operators	19
Figure 1.7. Binary Operators	19
Figure 1.8. Unary Operators	20
Figure 1.9. Numeric Constants	37
Figure 1.10. String Constants	38

Figure 1.11. Arithmetic Functions	41
Figure 1.12. String Functions	42
Figure 1.13. AWK printf Escape Characters	67



Chapter 1

AWK —

A PROGRAMMING LANGUAGE

1. Introduction

The following are examples of common **awk** applications:

- Report writing — especially those using information gathered from record-oriented data bases
- Pattern matching
- Data manipulation
- Information retrieval

An **awk** program is a sequence of statements of the form

```
pattern { action }  
pattern { action }  
...
```

An **awk** program is typically run using an input file or set of input files. Each line in a file is considered a **record** with each **field** being distinguished by white-space (default).

The basic operation of **awk** is to scan a set of input lines, in order, one at a time. In each line, **awk** searches for the **pattern** described in the **awk** program. If that pattern is found in the input line, a corresponding *action* is performed.

Each statement of an **awk** program is executed for a given input line. When all the patterns are tested, the next input line is fetched, and the **awk** program is once again executed from the beginning.

AWK

Either the **pattern** or the *action* may be omitted from an **awk** program, but not both:

- If there is no *action* for a **pattern**, the matching line is simply printed.
- If there is no *pattern* for an **action**, then the action is performed for every input line.

A NULL **awk** program does nothing.

Since **patterns** and *actions* are both optional, *actions* are enclosed in braces to distinguish them from **patterns**.

The **patterns** follow the regular expression syntax used in the UNIX editors (e.g., **sed**, **ed**, etc.)

The following command line instructs **awk** to print every line in **inputfile** which contains an **x**:

```
% awk '/x/ {print}' inputfile
```

2. General Structure

An **awk** program has the following structure:

1. A **< BEGIN >** section
This section is run before any input lines are read.
2. A **< record >** or main section
This section is *data driven*, since it is the section that is run over and over for each **record** (separate line) of input.
3. An **< END >** section
This section is run **AFTER** all the data files are processed.

Values are assigned to variables from the command line. The **< BEGIN >** section is run before these assignments are made.

The words **BEGIN** and **END** are actually **patterns** recognized by **awk**.

No declarations are necessary to formally distinguish an **awk** variable containing a character value from one containing an integer value. The **awk** processor considers the value of the variable as inferred by its use.

3. BEGIN and END Statements

The special pattern, **BEGIN**, matches the beginning of the input before the first record is read.

The pattern, **END**, matches the end of the input after the last line is processed.

BEGIN and **END** provide a way to gain control before and after processing for initialization and wrapping up.

One popular use of **BEGIN** is to place column headings on the output.

The following display is the contents of the text file **countries**, separated by tabs, which will be used in most of the examples in this document: (The headings are provided here for clarity, but are NOT contained in the example file, **countries**.)*

COUNTRY	AREA†	POP.‡	CONTINENT
Russia	8650	262	Asia
Canada	3852	24	North America
China	3692	866	Asia
USA	3615	219	North America
Brazil	3286	116	South America
Australia	68	14	Australia
India	1269	637	Asia
Argentina	72	26	South America
Sudan	968	19	Africa
Algeria	920	18	Africa

* The figures are from 1978.

† AREA is given in thousands of square miles.

‡ POPULATION is population given in millions.

For the first example, the following commands are contained in an executable file called **report**: (The symbol $\textcircled{\text{O}}$ is used here to represent the tab character, which would otherwise not be clearly distinguished from space.)

```
awk 'BEGIN { FS="␣" ;
      print "Country", "Area", "Population", "Continent"
    } {print}' countries
```

By executing **report**, then, the following output results:

```
% report
Country Area Population Continent
Russia 8650 262 Asia
Canada 3852 24 North America
China 3692 866 Asia
USA 3615 219 North America
Brazil 3286 116 South America
Australia 68 14 Australia
India 1269 637 Asia
Argentina 72 26 South America
Sudan 968 19 Africa
Algeria 920 18 Africa
%
```

The columns are not well aligned, but a **printf** statement could be used to improve the appearance of the output. The **printf** statement in **awk** has the same format designators and syntax rules as those used in the C library version of **printf**. (See the section entitled, "Output to Printer or Terminal" for more information on using **printf** statements in **awk**.)

One reason to explicitly set the value of the variable **FS** (Field Separator), is that spaces sometimes separate groups of words that should be considered within the same field. In such a case, it would be more efficient to declare **FS** to be equal to tab or some other character. In fact, if **FS** were not set to tab, the first record in **countries** would have four fields, but the second record would have five — **North** would be the contents of the

AWK

fourth field and **America** the contents of the fifth.

If **BEGIN** is present, it is the first pattern.

If used, **END** is the last pattern.

4. Comments

Comments in **awk** programs begin with the character **#** and terminate at the end of the line with a newline character.

For example:

```
print { NR, $1, $2, $3 } # print the record number and  
                        # the first three fields
```

A comment can be appended to the end of any line of an **awk** program.

AWK

5. Keywords

KEYWORDS			
BEGIN	break	int	string
END	close	length	substr
FILENAME	continue	log	while
FS	exit	next	
NF	exp	number	
NR	for	print	
OFMT	getline	printf	
OFS	if	split	
ORS	in	sprintf	
RS	index	sqrt	

Figure 1.1. Keywords

6. Identifiers

Identifiers in **awk** serve to denote **variables** and **arrays**.

An identifier is a sequence of letters, digits, and underscores, beginning with a letter or an underscore.

Uppercase and lowercase letters are different.

6.1 Variables

A **variable** is one of the following:

- **Identifier**
- **Identifier[Expression]**
- **\$Term**

The numeric value of any uninitialized **variable** is **0**, and the string value is the empty string.

An **identifier** by itself is a simple variable.

A **variable** of the form **identifier[expression]** represents an element of an associative array named by **identifier**. The string value of **expression** is used as the index into the array. The preferred value of **identifier** or **identifier (expression)** is determined by context.

The built-in variable, **\$0** has both the string and numeric value of the current input record.

If the current input record represents a number, then the numeric value of **\$0** is the number and the string value is the literal string.

AWK

The preferred value of **\$0** is string unless the current input record is a number.

The **\$0** cannot be changed by assignment.

The variables **\$1**, **\$2**, . . . refer to fields 1, 2, . . . of the current input record.

The string and numeric value of **\$i** for $1 \leq i \leq \text{NF}$ are those of the *i*-th field of the current input record.

As with **\$0**, if the *i*th field represents a number, then the numeric value of **\$i** is the number and the string value is the literal string.

The preferred value of **\$i** is string unless the *i*-th field is a number.

The **\$i** is changed by assignment. The **\$0** is then changed accordingly.

In general, **\$term** refers to the input record if **term** has the numeric value 0 and to field **i** if the greatest integer in the numeric value of **term** is **i**.

If $i < 0$ or if $i \geq 100$, then accessing **\$i** causes **awk** to produce an error diagnostic.

If $\text{NF} < i < 100$, then **\$i** behaves like an uninitialized variable.

Accessing **\$i** for $i > \text{NF}$ does not change the value of **NF**.

6.1.1 Initialization

By default, variables are initialized to the NULL string, which has a numerical value of 0. (This eliminates the need for most initialization of variables in BEGIN sections.)

The following commands are used to print the name and population of the most highly populated country listed in the example input file, **countries**. These commands are contained in a command file.

```
awk 'maxpop<$3 {maxpop = $3; country = $1}
END {print country,maxpop}' countries
```

6.1.2 Type

Variables take on numeric or string values according to context.

The following **awk** command is used to print the sum of the population figures in the file **countries**. In this example, the variable **pop** is presumed to be a number:

```
% awk '{pop += $3} END {print "TOTAL:", pop}' countries
TOTAL: 2201
```

The following **awk** command is used to print the name and area of each country in the file **countries**. In this example, the variable **country** is presumed to be a string, and the commands are assumed to be contained in an executable file called **report**:

AWK

```
% cat report
awk '{ country = $1; size = $2 ;
printf "NAME: %-10s AREA %4d", country, size }' countries
% report
NAME:  Russia      AREA:  8650
NAME:  Canada      AREA:  3852
NAME:  China       AREA:  3692
NAME:  USA         AREA:  3615
NAME:  Brazil      AREA:  3286
NAME:  Australia   AREA:   68
NAME:  India       AREA:  1269
NAME:  Argentina   AREA:   72
NAME:  Sudan       AREA:  968
NAME:  Algeria     AREA:  920
%
```

In the following example, the value-type of the variable **maxpop** depends on the type of data found in the third field (\$3). (Evaluations of this type are done at run-time.)

```
awk 'maxpop < $3 {maxpop=$3}
END {print "Largest Population:", maxpop}' countries
```

Each variable (including fields) is potentially a string or a number OR BOTH at any time.

An arithmetic expression is of the type **number**.

A concatenation of strings is of type **string**.

If both operands in a comparison are numeric, the comparison is made numerically. Otherwise, operands are coerced to strings, if necessary, and the comparison is made on strings.

The following **awk** segment may be used to force a type conversion on the variable **expr** — from type **string** to type **number**.

expr += 0

The following **awk** segment may be used to force a type conversion on the variable **expr** — from type **number** to type **string**. (This can be expressed as achieving the type conversion by causing the value of **expr** to be “string-concatenated” with the **NULL** string.)

expr = ""

6.1.3 Incremented Variables

An incremented variable has one of the forms

- **++var**
- **--var**
- **var++**
- **var--**

The **++var** has the value **var + 1** and has the effect of **var = var + 1**.

The **--var** has the value **var - 1** and has the effect of **var = var - 1**.

The **var++** has the same value as **var** and has the effect of **var = var + 1**.

The **var--** has the same value as **var** and has the effect of **var = var - 1**.

The preferred value of an incremented variable is numeric.

6.1.4 Special Variables

NR Number of the current record.

NF Number of fields in the current record.

AWK

FS	Input field separator, by default it is set to a blank or tab.
RS	Input record separator, by default it is set to the newline character.
\$i	The <i>i-th</i> input field of the current record.
\$0	The entire current input record.
OFS	Output field separator, by default it is set to a blank.
ORS	Output record separator, by default it is set to the newline character.
OFMT	The format for printing numbers, with the print statement, by default is "%.6g".
FILENAME	The name of the input file currently being read. This is useful because awk commands are typically of the form

% awk -f program file1 file2 file3 ...

6.2 Arrays

Array elements in **awk** are not declared, they must only be mentioned to be used.

Subscripts may have any non-NULL value, including non-numeric strings.

The following **awk** segment is provided as an example of a conventional numeric subscript, and illustrates assigning the value of the current input record to the **NR-th** element of an array called **record**. (**NR** is a special **awk** variable which, holds the value of the current record number.)

record[NR] = \$0

The following commands are used to fill an array called **names** with the names of each country in the file **countries**. This is

done by assigning the value of the first field of each input record (country name) to the *NR-th* element of an array called **names**. Since **NR** holds the value of the current input record, the value of **names[1]** would be the first field in the first input record (**names[1]=Russia**), the value of **names[2]** would be the first field in the second input record (**names[2]=Canada**), etc. (The command file, **justnames**, is assumed to be executable.)

```
% cat justnames
awk '{ names[NR] = $1 }
END { for ( i in names ) {print "NAME:", names[i] } }' countries
% justnames
NAME: Canada
NAME: China
NAME: USA
NAME: Brazil
NAME: Australia
NAME: India
NAME: Argentina
NAME: Sudan
NAME: Algeria
NAME: Russia
```

Arrays are also indexed by non-numeric values. This capability is similar to the associative memory of Snobol tables. The following commands, contained in the file **awkfile**, are used to illustrate such array indexing. (First the contents of **awkfile** are displayed, then the file is executed and the result is printed.):

```
% cat awkfile
awk ' /Asia/ { popl["Asia"]+= $3 }
END { print "Asia=", popl["Asia"] }' countries
% awkfile
Asia=1765
%
```


AWK

Any expression can be used as a subscript in an array reference. The following command line is used to illustrate using the string value of a field to index an array called **area**.

```
% awk '{area[$1]=$2} END {print area["India"]}' countries
1269
%
```

6.2.1 Multidimensional Arrays

Multidimensional arrays may be created by using two levels of subscripts. For example,

```
for ( i = 1; i <= 10; i++)
    for ( j = 1; j <= 10; j++)
        multi["", j] = . . .
```

7. Operators

An **awk** program has assignment, arithmetic, relational, and logical operators similar to those in the C programming language.

An **awk** program also has regular expression pattern matching operators similar to those in the UNIX operating system program **egrep** and **lex**.

ASSIGNMENT OPERATORS		
Symbol	Usage	Description
=	assignment	$X = Y$
+=	plus-equals	$X += Y$ is similar to $X = X + Y$
-=	minus-equals	$X -= Y$ is similar to $X = X - Y$
*=	times-equals	$X *= Y$ is similar to $X = X * Y$
/=	divide-equals	$X /= Y$ is similar to $X = X / Y$
%=	mod-equals	$X \% = Y$ is similar to $X = X \% Y$
++	prefix and postfix increments	$++X$ and $X++$ are similar to $X = X + 1$
--	prefix and postfix decrements	$--X$ and $X--$ are similar to $X = X - 1$

Figure 1.2. Assignment Operators

ARITHMETIC OPERATORS	
Symbol	Description
+	unary and binary plus
-	unary and binary minus
*	multiplication
/	division
%	modulus
(...)	grouping

Figure 1.3. Arithmetic Operators

RELATIONAL OPERATORS	
Symbol	Description
<	less than
<=	less than or equal to
=	equal to
!=	not equal to
>=	greater than or equal to
>	greater than

Figure 1.4. Relational Operators

LOGICAL OPERATORS	
Symbol	Description
&&	and
	or
!	not

Figure 1.5. Logical Operators

REGULAR EXPRESSION PATTERN MATCHING OPERATORS	
Symbol	Description
<code>~</code>	matches
<code>!~</code>	does not match

Figure 1.6. Regular Expression Pattern Matching Operators

7.1 Binary Operations

The following binary arithmetic operators are recognized by **awk**:

BINARY OPERATORS	
<code>+</code>	Addition
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>%</code>	Modulus

Figure 1.7. Binary Operators

The binary operator is applied to the numeric value of the left operand and the right operand and the result is the usual numeric value.

The preferred value of binary terms is the numeric result of the binary operation on the left operand but the numeric value can be interpreted as a string value (see **Numeric Constants**).

The operators `*`, `/`, and `%` have higher precedence than `+` and `-`.

All operators are left associative.

AWK

All arithmetic is done in floating point.

7.2 Unary Operations

The following unary operators are recognized by awk:

UNARY OPERATORS
+ (positive)
- (negative)

Figure 1.8. Unary Operators

A unary operator is applied to the numeric value of a term.

The preferred value of a unary term is the numeric value of the result, however, the result can be interpreted as a string value.

Unary + and - have higher precedence than *, /, and %.

7.3 Regular Expressions

The awk program can be used to search files and find patterns matching regular expressions.

A pattern in front of an action acts as a selector that determines if the action is to be executed.

A variety of expressions are used as patterns:

- Regular Expressions
- Arithmetic Relational Expressions
- String-Valued Expressions
- Combinations of Regular Expressions, Arithmetic Relational and String-Valued Expressions

The simplest such regular expression is a literal string of characters enclosed in slashes.

The following examples use the text file **countries** (used in previous examples) as input. The command line instructs **awk** to search **countries** for all lines (input records) containing the string **Asia** in any field. (Since a pattern is given without a corresponding action, ALL fields of each input record containing the pattern **Asia** (in any field), will be printed.)

```
% awk '/Asia/' countries
```

The following command line is used to print all lines (input records) beginning with either **A**, **B**, or **C**. The brackets (**[** and **]**) are used to mean "any one of the characters enclosed." (Brackets are most often used to enclose a range of characters, e.g., **A-Z** or **0-9**, etc.) The caret (**^**) represents the beginning of the line — otherwise, the following command would find all lines containing either an **A**, **B** or **C** anywhere on the line.

```
% awk '/^[ABC]/' countries
```

The following command line is used to print all lines (input records) ending with **ia**. The dollar sign (**\$**) represents the end of the line.

```
% awk '/ia$/' countries
```

The following command line is used to print all lines (input records) which contain either **N** or **S**. The pipe (**|**) separates the alternatives.

```
% awk '/N|S/' countries
```

The following command line is used to print all lines (input records) which contain a **u**, followed by **one or more** **s**'s, followed by an **i**. The plus (**+**) is used to indicate one or more occurrences of the preceding character.

```
% awk '/us+i/' countries
```

The following command line is used to print all lines (input records) which contain a **da** followed by **zero or one** **n**'s. The

AWK

question mark (?) is used to indicate zero or one occurrence of the preceding character.

```
% awk '/dan?/' countries
```

The following command line is used to print all lines (input records) which contain an s, followed by **any single character**, followed by an a.

```
% awk '/s.a/' countries
```

The following command line is used to print all lines (input records) which contain an s, followed by **zero or more i's**, followed by an a.

```
% awk '/si*a/' countries
```

NOTE: The special meaning of metacharacters — such as ^ (beginning of line), \$ (end of line), . (any single character), * (zero or more characters), etc. — can be “escaped” by preceding these characters with a backslash (\).

The following command line instructs **awk** to search all input records for an s, followed by a period (.), followed by an a.

```
% awk '/s\.a/' countries
```

Regular expressions, as used with the UNIX editors, are understood by **awk**.

The following command line is used to print the first field of all lines (input records) where the string contained in the first field ends with the characters ia.

```
% awk '$1 ~ /ia$/ {print $1}' countries
```

The following operators may be used to combine patterns so that more specific subsets may be retrieved:

- || (OR)
- && (AND)
- ! (NOT)
- Parentheses (ASSOCIATION)

The following command line is used to print all lines (input records) for which the third field (population) is **greater than or equal to 100**, and for which the second field (area) is **greater than or equal to 3000**.

```
% awk '$2 >= 3000 && $3 >= 100' countries
```

The following command line is used to print all lines (input records) for which the value of the fourth field is **either Asia or Africa**.

```
% awk '$4 == "Asia" || $4 == "Africa"' countries
```

The following command line is also used to print all lines (input records) for which the value of the fourth field is **either Asia or Africa**, but a comparison to a regular expression is used instead of the OR operator (||).

```
% awk '$1 ~ /^(Asia|Africa)$/' countries
```

Both && and || guarantee that their operands are evaluated from left to right.

With both the && and || operators, evaluation stops as soon as truth or falsehood is determined.

7.4 Relational Expressions

It is possible to perform comparisons on the fields of a file and base output, or the control flow of an awk program, on the result of the comparison.

AWK

The conditions tested are those using the relational operators. (i.e., >, <, >=, <=, == and !=.)

In relational tests, if both operands are numeric, a numerical comparison is made. Otherwise, the operands are compared as strings.

The following examples all use the text file **countries** (used in previous examples) as their input file.

The following command line is used to list the countries whose population is greater than 100 million. Note that since a pattern is given without an action, each record in the input file which matches the pattern is printed.

```
% awk '$3 > 100' countries
Russia 8650 262 Asia
China 3692 866 Asia
USA 3615 219 North America
Brazil 3286 116 South America
India 1269 637 Asia
```

The following command line is used to list just the names of the countries (\$1) on the Asian continent.

```
% awk '$4 == "Asia" {print $1}' countries
Russia
China
India
```

The following command line is used to list the countries whose names begin with a letter greater than or equal to the letter S (i.e., S, T, ..., Z):

```
% awk '$1 >= "S" ' countries
USA 3615 219 North America
Sudan 968 19 Africa
```

The following command line is used to list the countries whose name is the same as that of the continent on which it resides:

```
% awk '$1 == $4' countries
Australia 68 14 Australia
```

7.5 Assignment Expressions

An assignment expression takes the following form:

variable assignment-operator expression

The preferred value of **variable** is the same as that of **expression**, and **assignment-operator** is one of those listed in Figure 1.2.

In an expression of the form:

variable = expression

the numeric and string value of **var** becomes those of **expression**.

An expression of the form:

variable operator = expression

is equivalent to

variable = variable operator expression

where **operator** is one of:

+ - * / %

The **assignment operators** are right associative and have the lowest precedence of any operator. For example, **a += b *= c-2** is equivalent to the following sequence of assignments:

AWK

```
b = b * (c-2)
a = a+b
```

The following commands are used to print the total population of and the number of countries in the continent of Asia. To do this, the variable **pop** is incremented by the value found in the third field (population) of every input record (input line) containing the pattern **Asia**. Also, if the pattern is matched, the variable **n** is incremented by 1. After all the input records have been searched, the **END** statement is executed and the quoted strings are printed, "as is," and the values of the variables (found outside the double quotes) are substituted for the variable name and printed.

```
awk '/Asia/ {pop+= $3; ++n}
END {print "Total",pop,"Number of countries",n}' countries
```

The following operators are recognized by **awk** and **C**:

ASSIGNMENT OPERATORS RECOGNIZED BY AWK AND C	
++	Increment; May be prefix (increment BEFORE operation), or postfix (increment AFTER operation).
--	Decrement; May be prefix (decrement BEFORE operation), or postfix (decrement AFTER operation).
--=	Subtract right operand from left operand and place the result in the left operand.
/=	Divide the left operand by the right operand and place the result in the left operand.
%=	Divide the left operand by the right operand and place the remainder in the left operand.
*=	Multiply the left operand by the right operand and place the result in the left operand.
+=	Add the left operand to the right operand and put the result in the left operand.

NOTE: The operation:

$$x += y$$

provides the same solution as the operation:

$$x = x + y$$

However, $x += y$ is shorter and runs faster.

ALSO: The operation:

$$++x$$

provides the same solution as the operation:

$$x = x + 1$$

However, $++x$ is shorter and runs faster.

7.6 Variables, Expressions and Assignments

It is possible to use **awk** to perform an arithmetic operation (or series of such operations) and store the result in a variable. Although variables store values as character strings, **awk** allows an almost transparent type conversion.

The following command line is used to print only the name of each country (first field) and its population density (per square mile). The population density is found by dividing the country's population (per million persons) by its area (per thousand square miles).

AWK

```
% awk '{print $1, (1000000 * $3)/($2 * 1000)}' countries
Russia 30.289
Canada 6.23053
China 234.561
USA 60.5809
Brazil 35.3013
Australia 205.882
India 501.97
Argentina 361.111
Sudan 19.6281
Algeria 19.5652
%
```

The following commands are also used to print the name of each country (first field) and its population density (per square mile), but a `printf` statement is used to force the output into a more precisely-aligned table. (Note that the commands here, because they are too long to fit easily on a command line, are assumed to be contained in an executable file.)

```
awk '{printf "%10s %6.1f\n", $1,
(1000000 * $3)/($2 * 1000)}' countries
```

The output using the `printf` statement looks like the following:

Russia	30.3
Canada	6.2
China	234.6
USA	60.6
Brazil	35.3
Australia	205.9
India	502.0
Argentina	361.1
Sudan	19.6
Algeria	19.6

Arithmetic is done internally in floating point.

Awk recognizes the standard arithmetic operators, i.e. +, -, *, / and %.

8. Input Records and Fields

The **awk** program reads its input one record at a time unless explicitly instructed to do otherwise.

By default, a record is a sequence of characters ending with a newline character or with an end of file. It is possible to change the character used to define the end of a record to something other than newline or end of file. This is done by assigning a new character to the special variable **RS** (Record Separator).

Once read, the record is assigned to the variable **\$0**, and the **awk** program then splits the record into **fields**.

By default, a **field** is a string of characters separated by blanks or tabs. It is possible to change the character used to define the end of a field to something other than whitespace by assigning a new character to the special variable **FS** (Field Separator).

Using the input file called **countries** (used in previous examples) the first record is the following:

```
Russia 8650 262 Asia
```

When this record is read by **awk**, it is assigned to the variable **\$0**. For example, the following **awk** command:

```
% awk '{ print $0 }' countries
```

Will print the entire record of each record read by **awk**.

To have only the name of the country printed, the following **awk** command would do the trick:

AWK

```
% awk '{ print $1 }' countries
```

The first field of the present record is referred to as **\$1** by the **awk** program, the second by **\$2**, the third by **\$3**, and so forth.

The variables **\$1**, **\$2**, etc. are particularly useful when it is necessary to print the fields of a record in a different order than that found in the original file.

To produce a list of countries with the population in the first column and the name of the country in the second, using the same **countries** file as input, the following command line could be used:

```
% awk '{print $4, $1, $3}' countries
```

8.1 Tokens

The token **\$0** is a special variable whose value is that of the current input record.

The tokens **\$1**, **\$2...** are special variables whose values are those of the first field (**\$1**), the second field (**\$2**), etc., of the current input record.

The keyword **NF** (Number of Fields) is a special variable whose value is the number of fields in the current input records.

\$NF holds the value of the last field of the current input records.

The fields of each record are numbered sequentially and the total number of fields can vary from record to record.

None of these variables is defined in the action associated with a BEGIN or END pattern, where there is no current input record.

The keyword NR (Number of Records) is a variable whose value is the number of input records read so far.

The first input record read is 1.

Tokens in **awk** are usually separated by non-NULL sequences of blank, tabs, and newlines, or by other punctuation symbols such as commas and semicolons.

- **Braces** ({ }) surround actions.
- **Slashes** (/ /) surround regular expression patterns.
- **Double quotes** (" ") surround strings.

8.2 Fields

Fields are referenced by the current value of NF prepended with a \$ (i.e., the first field is referenced by \$1, the second field by \$2, etc.).

These special, “built-in” variables may be used in arithmetic and string operations, assigned values and are always initialized, by default, to the NULL string.

The following command line is used to print all lines (input records), after first converting the value of the second field (area) from “per thousand square miles” to “per million square miles.” (The conversion from “area per thousand square miles” to “area per million square miles” is achieved by dividing the value of the second field by 1,000.)

```
% awk '{ $2 /= 1000; print }' countries
```


AWK

The following command line is used to print all lines (input records), after re-assigning the fourth field to represent the population density (per square miles). (The population density is calculated by multiplying the value of the third field (population per million) by 1,000,000 and dividing by the value of the second field (area per thousand square miles) multiplied by 1,000.

```
% awk '{ $4 = (1000000 * $3)/(1000 * $2); print }' countries
```

The following command line is used to print all lines (input records) after assigning the string "United States" to the first field of the record(s) containing the pattern "USA."

```
% awk '/USA/ { $1 = "United States" ; print }' countries
```

Fields are accessed by expressions.

For example, because fields are numbered sequentially, if the total number of fields of a given record is equal to some number *n*, then the *n*-th field is the last field in the record.

Therefore, to print the **next-to-the-last** field of a given record, use the variable for the total number of fields, **\$NF**, in an expression **\$(NF-1)** which results in a value 1 less than the value of the total number of fields -- the next-to-the-last-field of the record.

8.3 Field Separator

The keyword **FS** (Field Separator) is a variable indicating the current field separator, which is initially assigned the value of any non-NULL sequence of blanks and tabs.

The keyword **FS** may be changed to any single character by using either an assignment statement, or the command-line argument **-F**.

On the command line, `-F\t` makes tab the field separator.

If the field separator is NOT a blank, then there is a field in the record on each side of the separator. For instance, if the field separator is `|`, the record `1XXX1` has three fields. The first and last field are NULL.

If the field separator is a space, or if it is not explicitly assigned, then fields are assumed, by default, to be separated by white space. In this case, the record `1XXX1` would have only one field.

8.4 Record Separator

The keyword **RS** (Record Separator) is a variable whose value is the current record separator, which is initially set to newline.

Keyword **RS** is changed to any character by using an assignment statement in the *action* portion of an **awk** program.

For example, if **inputfile** contained the following: (Note that the character `@` is used in the following examples to make the tab character visible. When using this command, an actual tab should be used, not the `\t` notation used in the C programming language, or any other such representation).

```
one@two@three@four@five
```

Then the following command could be used to print each item in **inputfile**, preceded by its record number, i.e. its sequential position in **inputfile**.

AWK

```
% awk 'BEGIN { RS="@" } { print NR, $0 }' inputfile
1 one
2 two
3 three
4 four
5 five

%
```

8.5 Output Separators

The value of **OFS** (Output Field Separator) is the output field separator. The **OFS** separates the fields of each record output by an **awk** action, and is set to space (blank) by default.

There is also an Output Record Separator **ORS**, which separates each of the records output by an **awk** action, and is set to new-line by default.

The values of both **OFS** and **ORS** may be re-assigned.

Using the input file **countries** (used in previous examples), the following commands illustrate assigning a different value to **ORS** (Output Record Separator).

```
% awk 'BEGIN { ORS = " ...\\n" } {print NR, $0 }' countries
1 Russia 8650 262 Asia ...
2 Canada 3852 24 North America ...
3 China 3692 866 Asia ...
4 USA 3615 219 North America ...
5 Brazil 3286 116 South America ...
6 Australia 68 14 Australia ...
7 India 1269 637 Asia ...
8 Argentina 72 26 South America ...
9 Sudan 968 19 Africa ...
10 Algeria 920 18 Africa ...

%
```

Note that the FS (Field Separator) was NOT re-assigned the value of tab, and therefore was, by default, space. Since the commands we used did not concern fields, and since ORS is not effected by the number of fields in a given record, the following example is provided to illustrate assigning a value to the OFS (Output Field Separator). In the first command line, FS will use its default value of space. In the second command line, FS will be re-assigned the value of tab.

```
% awk 'BEGIN { OFS = " ... " } { print NR, $0 }' countries
```

```
1 ... Russia 8650 262 Asia
2 ... Canada 3852 24 North America
3 ... China 3692 866 Asia
4 ... USA 3615 219 North America
5 ... Brazil 3286 116 South America
6 ... Australia 68 14 Australia
7 ... India 1269 637 Asia
8 ... Argentina 72 26 South America
9 ... Sudan 968 19 Africa
10 ... Algeria 920 18 Africa
```

```
% awk -F\t 'BEGIN {OFS=" ... "}{print NR,$1,$2}' countries
```

```
1 ... Russia ... 8650
2 ... Canada ... 3852
3 ... China ... 3692
4 ... USA ... 3615
5 ... Brazil ... 3286
6 ... Australia ... 68
7 ... India ... 1269
8 ... Argentina ... 72
9 ... Sudan ... 968
10 ... Algeria ... 920
```

The OFS appears wherever a comma appears in an **awk** print statement.

8.6 Multiline Records

The assignment **RS=""** (no white space between the double quotes) explicitly assigns the Record Separator the value of an

AWK

empty line while implicitly assigning the Field Separator (FS) the value of any non-NULL sequence consisting of blanks, tabs, and possibly a newline.

With this setting, none of the first NF fields of any record are NULL.

8.7 Ranges

The following command line is used to print all lines (input records) found between the first occurrence of **Canada**, and the first occurrence of **Brazil**, including the lines (input records) which contain these delimiting patterns.

```
% awk '/Canada/,/Brazil/' countries
```

The following command line is used to print all the fields of lines (input records) 2 through 5, inclusive.

```
% awk 'NR == 2, NR == 5' countries
```

The following command line is used to print all lines (input records) found between the first occurrence of **Canada**, and the first occurrence of **Africa** – **IN THE FOURTH FIELD (\$4)**, including the lines (input records) which contain these delimiting patterns.

```
% awk '/Canada/, $4 == "Africa"' countries
```

Normally, pattern ranges occur **OUTSIDE** the curly brackets ({}) which delimit an **awk action** segment. To check pattern ranges **INSIDE** an **awk action** use a control flow statement, such as an **if** or **while** statement.

9. Numeric Constants

A Numeric Constant is either:

- **A Decimal Constant**

A decimal constant is a non-NULL sequence of digits containing at most one decimal point as in 12, 12., 1.2, and .12.

OR

- **A Floating Constant**

A floating constant is a decimal constant followed by e or E followed by an optional + or - sign followed by a non-NULL sequence of digits as in 12e3, 1.2e3, 1.2e-3, and 1.2E+3.

The maximum size and precision of a numeric constant are machine dependent.

Numeric values are stored as floating point numbers.

Both the numeric and string value of a numeric constant is the decimal number represented by the constant.

The preferred value is the numeric value.

NUMERIC CONSTANTS		
Numeric Constant	Numeric Value	String Value
0	0	0
1	1	1
.5	0.5	.5
.5e2	50	50

Figure 1.9. Numeric Constants

10. String Constants

A string constant is a sequence of zero or more characters surrounded by double quotes as in "", "a", "ab", and "12".

Only the following two characters need to be escaped when put in a string:

- A double quote is put in a string by preceding it with the backslash character (\).
- A newline is put in a string by using \n in its place.

Strings can be (almost) any length.

The numeric value of a string constant is 0 unless the string is a numeric constant enclosed in double quotes. In this case, the numeric value is the number represented.

The preferred value of a string constant is its string value.

The string value of a string constant is always the string itself.

STRING CONSTANTS		
String Constant	Numeric Value	String Value
" "	0	empty space
"a"	0	a
"XYZ"	0	XYZ
"0"	0	0
"1"	1	1
".5"	0.5	.5
".5e2"	0.5	.5e2

Figure 1.10. String Constants

10.1 String Concatenation

Strings are concatenated as in the following example:

```
% awk '{ x = "hello" x = x " , world" print x }'  
hello, world
```

The following command line is used to print the value of the variable `s`, which is assigned the value of the first field (country name) of every input record which has the pattern `A` in any field, each country's name being separated by a space(" ").

```
% awk '/A/ { s = s " " $1 } END { print s }' countries  
Australia Argentina Algeria
```

Variables, string expressions, and numeric expressions may appear in concatenations.

Numeric expressions are treated as strings in concatenations.

In the first of the following example, the string value of the terms are concatenated. (The file `countries`, used in previous examples, is used as the input file). In the second example, the command line is changed to include a comma between the terms which causes the Output Field Separator (OFS) to be inserted, preventing concatenation: (As was previously mentioned, the default value of the Output Field Separator is space.)

```
% awk -F\t 'NR==1,NR==3 { print $2 $3 }' countries  
8650262  
385224  
3692866
```

```
% awk -F\t 'NR==1,NR==3 { print $2, $3 }' countries  
8650 262  
3852 24  
3692 866
```


AWK

The preferred value of the resulting expression is a string value that can be interpreted as a numeric value.

Concatenation of terms has lower precedence than binary $+$ and $-$. For example, $1+2\ 3+4$ has the string (and numeric) value 37, since $1+2$ equals 3 and $3+4$ equals 7 and the concatenation of the terms is then 37.

11. Functions

The **awk** program has a number of built-in functions that perform common arithmetic and string operations.

ARITHMETIC FUNCTIONS
cos(expression)
exp(expression)
int(expression)
log(expression)
sin(expression)
sqrt(expression)

Figure 1.11. Arithmetic Functions

These functions (**cos**, **exp**, **int**, **log**, **sin** and **sqrt**) compute the cosine, exponential, integer part, natural logarithm, sin and square root, respectively, of the numeric value of **expression**.

If the **(expression)** is omitted, then the function is applied to **\$0**.

The preferred value of an arithmetic function is numeric.

STRING FUNCTIONS
getline()
index(string1, string2)
length(string)
split(string, array, field separator)
split(string, array)
sprintf(format, expression(s))
substr(string, position)
substr(string, position, length)

Figure 1.12. String Functions

The function **getline** causes the next input record to replace the current record. It returns 1 if there is a next input record or a 0 if there is no next input record. (The value of NR is updated.)

The function **index(string1,string2)** takes the string values of **string1** and **string2** and returns the first position of where **string2** occurs in **string1**, and returns 0 if **string2** is not present in **string1**.

index("abc", "bc") = 2

index("abc", "ac") = 0

The function **length** without an argument returns the number of characters in the current input record.

The function **length(string)** returns the number of characters in the string value of **string**. For example:

length("abc") = 3

length(17) = 2

The function **split(string, array, field separator)** splits the string value of **string** into fields that are then stored in **array**

using the string value of **field separator** as the field separator.

The function **split(string, array, field separator)** returns the number of fields found in **string**.

If the function **split(string, array)** is used, the current value of **FS** is used to indicate the field separator.

The following example uses **cat** to show the contents of **inputfile** and then uses **awk** and its **split** function to print the last field and the number representing that field in **inputfile**.

```
% cat inputfile
one@two@three@four@five
% awk -F\t '{ n=split($0, a); print a[n], n }' inputfile
five 5
```

Because **n** represents the number of fields in **\$0**, and the array, **a**, the notation **a[n]** represents the value contained in the **n-th** field of **a**. Alternately, the notation **a[\$NF]** could be used, since the **n** is equal to the number of fields in a given record.

Also, since the **-F** command-line option was used to assign the value **\t** (tab) as the Field Separator (**FS**), **split** understood that character as the field delimiter, instead of the **FS** default — space.

The function **sprintf(format, expression(s))** produces the value of **expression(s)** in the format specified by the string value of **format**.

The format control conventions in the function **sprintf(format, expression(s))** are those of the **printf** statement in the C programming language library.

AWK

The function **substr(string1, position)** returns the substring of **string1** beginning at **position**.

The function **substr(string, position, length)** returns the substring of **string** that begins at **position** and is **length** characters long.

If **position+length** is greater than the length of **string**, then **substr(string, position, length)** is equivalent to **substr(string, position)**.

```
substr("abc", 2, 1) = "b"  
substr("abc", 2, 2) = "bc"  
substr("abc", 2, 3) = "bc"  
substr("abc", 3, 1) = "c"
```

Positions (**position**) less than 1 are taken as 1.

A negative or zero **length** produces a NULL result.

The preferred value of **sprintf** and **substr** is string.

The preferred value of the remaining string functions is numeric.

11.1 Summary of Built-in Functions

The following is a summary of the functions (both mathematical and string) built-in the **awk** program:

- cos** This function returns the cosine of its argument.
- exp** This function returns the exponential of its argument.
- getline** The function **getline** immediately reads the next input record.
Fields **NR** and **\$0** are all set but control is left at exactly the same spot in the **awk** program.
The **getline** function returns 0 for the end-of-file.

The **getline** function returns a 1 for a normal record.

index The function **index(string1,string2)** returns the left-most position in **string1** where the **string2** occurs. If **string1** does not contain **string2**, **index(string1,string2)** returns zero.

int This function returns the integer part of its argument — truncating toward 0.

length The function used to compute the length of a string of characters.

Without an argument, **length** is assumed to mean **length(\$0)**, i.e., the length of the entire input record. In general, **length(x)** returns the length of **x** as a STRING.

The following example is used to print all input records in the file **countries**, preceding each by its length.

```
% awk '{print length, $0 }' countries
```

The next example is used to determine and print the name of the country having the longest name in the file **countries**.

```
awk 'length($1) > max {max=length($1); name=$1}
END {print name}' countries
```

log This function returns the natural logarithm of its argument.

sin This function returns the sine of its argument.

split The function **split(motto, words, dogear)** assigns the fields of the string **motto** to successive elements of the array called **words**. If the **dogear** argument is present, its first character is used as the string field separator. Otherwise, the current value of **FS** (field separator) is used as the string field separator.

The number of elements found is returned as the value of **split**.

AWK

```
awk 'BEGIN {split("Country Area Population", titles)}
{print titles[1], $1, titles[2], $2, titles[3], $3}' countries
```

sprintf The **sprintf** function uses the same formatting conventions as does **printf**, but is used to assign an expression or its result to a variable instead of sending the results to **stdout** (Standard Output).

In the following **awk** program, the value of the string produced by formatting the values of the first and second fields (**\$1** and **\$2**) is assigned to the variable **x**. The variable **x** may then be used in subsequent computations.

```
% awk '{x = sprintf("%10s %6d ", $1, $2 ); print x}' countries
```

Russia	8650
Canada	3852
China	3692
USA	3615
Brazil	3286
Australia	68
India	1269
Argentina	72
Sudan	968
Algeria	920

sqrt This function returns the square root of its argument.

substr The substring function, **substr(words, start, max)**, returns the substring of **words**, which begins at position **start**, and which is at most **max** characters long. If the third argument (**max**, in the example), is omitted, the substring returned will begin at the position provided in the second argument (**start**), and end at the end of the string provided in the first argument (**words**).

The following example is used to print each input record, with the names of the countries abbreviated to 3 letters.

```
% awk '{ $1 = substr($1, 1, 3); print }' countries
```

The value returned by the function `substr(123456789,3,4)`, is 3456 — the 4-character string beginning at the 3-rd position in 123456789.

12. Direct Command-line Usage

One way to use the **awk** program is to feed the **pattern-action** statements to **awk** for processing directly on the command line. This method is primarily used when just a few **awk** **pattern-action** statements are to be executed (a line or two). For example:

awk 'pattern-action statements' files

Here **files** is an optional list of input files on which the **pattern-action** statements are to be run.

Note that there are single quotes around the **pattern-action** statements in order for the shell to accept the entire string as the first argument to **awk**.

If no input files are specified, **awk** takes input from the standard input (**stdin**).

You can explicitly declare that input is to come from standard input by using **-** (the hyphen) in place of, or as one of **files**. For example:

awk 'pattern-action statements' file1 file2 -

In this example, **awk** takes input from **file1**, **file2** and standard input (**-**), and then processes the **pattern-action** statements on **file1**, **file2** and finally on input received from standard input.

It is possible to assign values to variables from within an **awk** program. Because you do not declare types of variables, a variable is created simply by referring to it. For example:

x=5

This statement in an **awk** program assigns the value **5** to the variable **x**.

It is also possible to assign values to variables from the command line. This provides another way to supply input values to **awk** programs. For example:

```
awk '{print x }' x=5 -
```

will print the value **5** on the standard output.

The minus sign at the end is necessary in this instance to indicate explicitly that input is coming from **stdin** so that **x=5** is not misinterpreted as the name of a file on which **print x** should be executed.

If the input were to come from a file named **text**, the command would then be:

```
awk '{print x}' file
```

It is NOT possible to assign values to variables used in the **BEGIN** section in this way (i.e., **OFS**, **FS**, **ORS**, **RS**...).

If it is necessary to change the record separator (**RS**) and/or the field separator (**FS**), it can be done on the command line, using the following syntax:

```
awk -f cmd.awk RS=":" text
```

In this example, the pattern-action statements to be executed on the text file **text** are contained in a command file **cmd.awk**. For the sake of the example, the re-assignment of the record separator (**RS**) was done on the command line instead of in **cmd.awk**.

AWK

In this example, the record separator (**RS**) is assigned the value : (colon). This new value, then, replaces the default **RS** value of newline.

There are two ways to change the field separator (**FS**) from the command line. One is to use the following structure:

```
awk -f cmd.awk FS=":" text
```

In this example, the field separator (**FS**) is assigned the value : (colon). This new value, then, replaces the default **RS** value of space (or tab).

The other way to change the field separator (**FS**) from the command line is to use the **-F** option. For example:

```
awk -F: -f cmd.awk text
```

In this example, the pattern-action statements to be executed on the text file **text** are contained in a command file **cmd.awk**.

The field separator (**FS**) is assigned the value : (colon). This new value, then, replaces the default **FS** value of space (or tab).

If the field separator (**FS**) is explicitly assigned the value of tab, using either the command-line option (**awk -F\t**), or an assignment statement (**FS="\t"**), where **\t** represents a literal tab, blanks are NOT recognized as separating fields. For example:

```
% cat onlytabs
awk -F\t '/North America/ { n = split( $0, a );
for ( i = 1 ; i <= n ; i++ )
{ print "Field #" i, "is", a[i] }
print "" }' countries
% onlytabs
Field #1 is Canada
Field #2 is 3852
Field #3 is 24
Field #4 is North America

Field #1 is USA
Field #2 is 3615
Field #3 is 219
Field #4 is North America
```

```
%
```

However, if the field separator (FS) is explicitly assigned the value of space using either the command-line option (**awk -F" "**), or an assignment statement (**FS=" "**), tabs continue to be recognized by **awk** as field separators. For example:

AWK

```
% cat tabandspace
awk -F" " '/South America/ { n = split( $0, a );
for ( i = 1 ; i <= n ; i++ )
{ print "Field #" i, "is", a[i] }
print "" }' countries
% tabandspace
Field #1 is Brazil
Field #2 is 3286
Field #3 is 116
Field #4 is South
Field #5 is America

Field #1 is Argentina
Field #2 is 72
Field #3 is 26
Field #4 is South
Field #5 is America

%
```

12.1 Cooperation with the Shell

Normally, an `awk` program is either contained in a file or enclosed within single quotes.

The `awk` program uses many of the same characters that the shell does, such as `$` and the double quote.

Surrounding the `awk` instructions with single quotes (`'...'`), ensures that the shell passes the instructions to `awk` intact.

The following example is used to print the *n*-th field of an input file, where *n* is a parameter whose value is determined at run-time. (The ways in which *n* may be assigned a value will be discussed following the example.)

field n

The example above is equivalent to the command:

```
awk '{print $n}'
```

The variable **n** may be assigned a value in a number of ways. The following example is used to illustrate one of these ways:

```
awk '{print $ $1}'
```

Spaces are critical in the example above, since there is only one argument, even though there are two sets of quotes.

The **\$1** is OUTSIDE the quotes, and is "substituted" properly when **field** is invoked.

The next example also illustrates a way in which **n** may be assigned a value. This method exploits on the fact that the shell substitutes **\$** parameters within double quotes.

```
% awk "{print $1}"
```

Here the trick is to protect the **\$** with a ****; the **\$1** is again replaced by the number when **field** is invoked.

13. Using Command Files

If the number of **awk** pattern-action statements is more than one or two, it is usually more convenient (and efficient) to put the pattern-action statements into a file.

For example, suppose the following commands were contained in a file called **commandfile**

```
% cat commandfile
BEGIN {print "Country Names" }
{ print $1 }
%
```

Then the following command line would run those commands on the input file **countries** (used in previous examples).

```
% awk -f commandfile countries
```

Standard input could be used as input to **commandfile** by using a dash (-) in its place:

```
% awk -F\t -f commandfile -
```

Recall that the word **BEGIN** is a special pattern indicating that the action following in braces is run before any data is read. Also, that the **-F\t** is necessary because the fields in the file **countries** are separated by tabs.

Another way to use an **awk** is to create a script. The following example illustrates just such a use.

```
% cat printnames
awk -F\t 'BEGIN { print "Country Names:" }
{ print $1 }' countries
% chmod +x printnames
% printnames
Country Names:
Russia
Canada
China
USA
Brazil
Australia
India
Argentina
Sudan
Algeria
%
```


14. Control Flow Statements

14.1 if-else

The if statement has the following general structure, which is the same as that used in the C Programming Language:

```
awk 'if ( $4 == "Asia" ) { print $0 }
    else {print $1, "isn't in Asia" }' countries
```

The else segment is optional.

Several statements enclosed in curly brackets ({}) are treated as a single statement.

The following **awk** program finds the country with the largest population (among those listed in the file **countries**), while demonstrating the use of if statements.

```
awk '{ if (maxpop < $3)
      {
        maxpop=$3
        country=$1
      }
    }
END { print country, maxpop }' countries
```

14.2 while Statement

There is also a while statement in **awk** with the following general format:

```
% awk '{while ( $1 == $4 ) { print $0 } }' countries
```

The following example illustrates the Euclidean algorithm for finding the greatest common divisor of the second and third fields of each record (line) in the input file. (Even though there is no need to find the greatest common divisor of the area and population fields in **countries**, that file is used as input

for consistency.)

```
% cat getgcd
awk -F\t '{ printf "GCD of %4d and %3d ", $2, $3
             while ( $2 != $3 )
             {
               if ( $2 > $3 ) $2 = $2 - $3
               else $3 = $3 - $2
             }
             printf "is %2d\n", $2 }' countries
```

```
% getgcd
GCD of 8650 and 262 is 2
GCD of 3852 and 24 is 12
GCD of 3692 and 866 is 2
GCD of 3615 and 219 is 3
GCD of 3286 and 116 is 2
GCD of 68 and 14 is 2
GCD of 1269 and 637 is 1
GCD of 72 and 26 is 2
GCD of 968 and 19 is 1
GCD of 920 and 18 is 2
```

14.3 for Statement

The **for** statement in **awk** is like that provided by the C programming language.

The following example illustrates the usage of the **awk** for-loop by reading the fields of each of the first three records into an array, **a**, and then using array notation to represent each field in the print statement. The commands are contained in an executable file, **fielder**.

AWK

```
% cat fielder
awk -F\t 'NR==1,NR==3 { n=split($0, a);
               for ( i=1; i<= n; i++ )
               { print "Field #" i, "is", a[i] }
               print "" }' countries

% fielder
Field #1 is Russia
Field #2 is 8650
Field #3 is 262
Field #4 is Asia

Field #1 is Canada
Field #2 is 3852
Field #3 is 24
Field #4 is North America

Field #1 is China
Field #2 is 3692
Field #3 is 866
Field #4 is Asia

%
```

The following is an alternate form of the **for** statement used to access the elements of an associative array:

```
% cat otherone
awk -F\t 'NR==1,NR==3 { n=split($0, a);
               for ( i in a )
               { print "Field #" i, "is", a[i] }
               print "" }' countries
```

Chaos will result if **i** is altered or if new variables are added within the loop.

The following **awk** program, **recorder**, is used to print the current record number (the value of the built-in variable, **NR**), followed by the entire record itself for all input records in the

file countries.

```
% cat recorder
awk -F\t '{ array[NR] = $0 }
END { for ( i in array )
{ print i, array[i] } }' countries | sort -n
% recorder
1 Russia 8650 262 Asia
2 Canada 3852 24 North America
3 China 3692 866 Asia
4 USA 3615 219 North America
5 Brazil 3286 116 South America
6 Australia 68 14 Australia
7 India 1269 637 Asia
8 Argentina 72 26 South America
9 Sudan 968 19 Africa
10 Algeria 920 18 Africa
```

The following **awk** program, **popucont**, is used to illustrate using strings as array indices for the purpose of adding the populations of each continent as listed in the file **countries**.

```
% cat popucont
awk 'BEGIN { FS="␣" }
      { population[$4] += $3 }
END { for ( i in population )
{ printf "%-13s %4d0, i, population[i] } }' countries
% popucont
South America    142
Africa           37
Asia             1765
Australia        14
North America    243
```

In the above example, the total population is computed for one continent at a time.

The **condition** part of an **if**, **while** or **for** statement can include relational operators (i.e., **<**, **<=**, **>**, **>=**, **==**, **!=**).

AWK

The **condition** part of an **if**, **while** or **for** statement can also include regular expressions and the operators **~** and **!~**.

The **condition** part of an **if**, **while** or **for** statement can also include logical operators (i.e., **||**, **&&**, **!**).

The **condition** part of an **if**, **while** or **for** statement can also include parentheses for grouping.

14.4 **break**, **continue** and **next** Statements

The **break** statement, when it occurs within a **while** or **for** loop, causes an immediate exit from the loop.

The **continue** statement, when it occurs within a **while** or **for** loop, invokes the next iteration of the loop.

The **next** statement causes both an immediate skip to the next record, and a return to the beginning of the program's pattern-action statements. In this way, all the the pattern-matching statements will be performed for each record.

It is important to note the difference between **getline** and **next**. The **getline** instruction does not cause a skip to the beginning of the program's pattern-matching statements.

If an **exit** statement occurs in the **BEGIN** section of an **awk** program, the program stops executing and the **END** section is not executed (if there is one).

If an **exit** statement occurs in the main body of the program, no further records are read and the **END** section is executed.

If an **exit** statement occurs in the **END** section of an **awk** program, execution terminates at that point.

15. Report Generation

The control statements (i.e., if, else, for, while, etc.) are especially useful when **awk** is used as a report generator.

The following example is used to calculate the population for each continent listed in the file **countries**.

```
awk -F\t '{ total[$4] += $3}
END { for ( j in total )
print j, "has a population of", total[j] }' countries
```

The example above produces the following output:

```
South America has a population of 142
Africa has a population of 37
Asia has a population of 1765
Australia has a population of 14
North America has a population of 243
```

The following example is used to print the countries which contribute to the continent population total.

```
awk -F\t '{ total[$4] += $3
            country[NR] = $1
            pop[NR] = $3
            continent[NR] = $4 }
END { for ( j in total )
      if ( j != done )
      {
        print j, total[j]
        for ( i=1 ; i <= NR ; i++ )
        {
          if ( continent[i] == j )
            {print "      " country[i], pop[i]}
        } done=j } }' countries
```

The example above produces the following output:

AWK

```
South America 142
    Brazil 116
    Argentina 26
Africa 37
    Sudan 19
    Algeria 18
Asia 1765
    Russia 262
    China 866
    India 637
Australia 14
    Australia 14
North America 243
    Canada 24
    USA 219
```

15.1 Output to Printer or Terminal

An action may have no pattern; in which case the action is executed for all lines. For example:

```
% awk '{print}' countries
```

This example is one of the simplest actions performed by **awk**. It prints each line of the input to the output.

To print only the first and third fields of the file **countries** (used in previous examples), use the following command line:

```
% awk '{ print $1, $3 }' countries
```

This example prints the first field (name of the country) followed by the third field (population) and ignores all other fields.

The use of a semicolon at the end of a single statement in **awk** programs is optional. Both

```
% awk '{print $1 }' countries
```

and

```
% awk '{print $1; }' countries
```

are considered the same to **awk**.

To put two **awk** statements on the same line of an **awk** script, it is necessary to separate them with a semicolon. For example:

```
% awk '{x=5; print x }' countries
```

Parentheses are also optional with the **print** statement. For example:

```
% awk '{ print $3, $2 }' countries
```

and

```
% awk '{ print ($3, $2 ) }' countries
```

are considered equivalent by **awk**.

Items separated by a comma in a print statement are separated by the current output field separator (**OFS**). Even if the input fields are separated by tabs, the default output field separator (**OFS**) is space. Therefore, when printed, each field would be separated by a space, not by a tab, unless the **OFS** is assigned a different value. For example, using the text file **countries** (used in previous examples) for the input file, and using the following pattern-action statements contained in a command instruction file called **cmd.awk**:

AWK

```
% cat cmd.awk
BEGIN { OFS=" has a population of " }
{ print $1, $3 }
%
```

the command line:

```
% awk -F\t -f cmd.awk countries
```

will produce the following output:

```
Russia has a population of 262
Canada has a population of 24
China has a population of 866
USA has a population of 219
Brazil has a population of 116
Australia has a population of 14
India has a population of 637
Argentina has a population of 26
Sudan has a population of 19
Algeria has a population of 18
```

It is also possible to use the **print** command to print literal strings. For example:

```
% awk '{print "hello, world" }'
hello, world
%
```

In addition to the **RS** and **FS** variables previously discussed, **awk** provides other useful variables. Among them are **NR** and **NF**.

- **NF**
This variable is an integer which represents the number of fields in a given record.

and ...

- **NR**

This variable is an integer which represents the number of the given record — in relationship to all the records in a given file.

For example:

```
% awk '{print NR, NF, $0}' countries
```

prints each record number (NR), and the number of fields in each record (NF), followed by the entire record itself (\$0).

Using the text file **countries** (used in previous examples) as the input file, the following **awk** command line:

```
awk -F\t '{ print NR, NF, $0 }' countries
```

will produce the following output:

```
1 4 Russia 8650 262 Asia
2 4 Canada 3852 24 North America
3 4 China 3692 866 Asia
4 4 USA 3615 219 North America
5 4 Brazil 3286 116 South America
6 4 Australia 68 14 Australia
7 4 India 1269 637 Asia
8 4 Argentina 72 26 South America
9 4 Sudan 968 19 Africa
10 4 Algeria 920 18 Africa
```

Notice that the columns are not perfectly aligned due to the fact that the **OFS** variable was not changed from its default space and the names of the 10 countries are not the same length. There are several ways to “fine tune” output format. One of which is the output field separator variable (**OFS**) discussed previously.

AWK

The **print** command, without any arguments, prints each input record.

To print an empty line, use **print** with the argument "" (no space inbetween the double quotes).

The **awk** program also recognizes **printf** format indicators to enable more exact output instructions. The **print** command uses the default **printf** format **%6g** for each variable printed. It is possible to change this default format with a **printf** format statement just as you would use **printf** in a C program. For example:

```
awk '{ printf "%10s %6d %6d \n", $1, $2, $3 }' countries
```

The **printf** statement in this example instructs **awk** to print **\$1** as a right-justified string, 10 characters long (**%10s**); and to print **\$2** and **\$3** as 6-digit numbers. Using this **printf** statement, then, will align the columns and produce the following table:

Russia	8650	262
Canada	3852	24
China	3692	866
USA	3615	219
Brazil	3286	116
Australia	68	14
India	1269	637
Argentina	72	26
Sudan	968	19
Algeria	920	18

With **printf** no output field separator (**OFS**) or newline is produced automatically. This is why the **printf** format statement used in the example above contained an explicit newline at the end of the format specifiers (**\n**).

Other escape characters used with **printf** in a C program are also understood when used in a **awk** program. These escape characters are:

AWK printf ESCAPE CHARACTERS	
\n	Newline
\t	Tab
\b	Backspace
\r	Carriage Return

Figure 1.13. AWK printf Escape Characters

When a pattern is specified, but no action is associated with that pattern, the entire record **\$0** is printed.

The value of the variable **OFS** (output field separator) is printed wherever the comma appears in a print statment. For example, using the following pattern-action statements in a file called **cmd.awk**:

```
BEGIN { FS="␣" ; OFS="*" }
{print $1, $2, $3 }
```

giving the command:

```
awk -f cmd.awk countries
```

produces the following output:

AWK

```
Russia*8650*262
Canada*3852*24
China*3692*866
USA*3615*219
Brazil*3286*116
Australia*68*14
India*1269*637
Argentina*72*26
Sudan*968*19
Algeria*920*18
```

To get a literal comma on the output, you can either insert it in the **print** statement:

```
% awk '{print $1",", $2",", $3 }' countries
```

or you can assign a different value to the variable **OFS** (output field separator) in the **BEGIN** section:

```
% awk 'BEGIN {OFS=", "} {print $1, $2, $3 }' countries
```

Both of these last two ways of printing a comma produce the following output:

```
Russia, 8650, 262
Canada, 3852, 24
China, 3692, 866
USA, 3615, 219
Brazil, 3286, 116
Australia, 68, 14
India, 1269, 637
Argentina, 72, 26
Sudan, 968, 19
Algeria, 920, 18
```

15.2 Output to Files

The UNIX operating system shell allows you to redirect standard output to a file. The **awk** program also lets you direct output to many different files from within your **awk** program.

To direct output into a file after a **print** or a **printf** statement, use a statement of the general form:

```
% awk '{ print $1 > "NAMES" }' countries
```

The file names **MUST** be quoted (e.g., "NAMES"). Without quotes, the file names are treated as uninitialized variables and all output then goes to the same file.

If **> "NAMES"** is replaced by **>> "NAMES"**, output is appended to FILE.

The number of files that may be written in this way is presently limited to 10.

Using the text file **countries** (used in previous examples) as input, and placing the following pattern-action statements in a file called **cmd.awk**:

```
{ if ($4 == "Asia") print > "ASIA"
  if ($4 == "Europe") print > "EUROPE"
  if ($4 == "North") print > "N_AMERICA"
  if ($4 == "South") print > "S_AMERICA"
  if ($4 == "Australia") print > "AUSTRALIA"
  if ($4 == "Africa") print > "AFRICA"
}
```

the following command:

```
awk -f cmd.awk countries
```

will automatically place the information regarding **Asia** in a file called **ASIA**, etc.

Notice that the field separator was not changed in the command line above - nor was it changed in the command file **cmd.awk**, therefore the fields are assumed to be separated by spaces, and

AWK

the fourth field of the third record will be equal to **North**. Had the field separator been assigned the value of **tab**, the fourth field of the third record would be equal to **North America**.

15.3 Output to Pipes

In addition to redirecting output to a file, it is possible to direct output to a pipe. For example,

```
% awk '{ print $1 | "sort" }' countries
Algeria
Argentina
Australia
Brazil
Canada
China
India
Russia
Sudan
USA
```

This example command line will take the first field of each input record, sort these fields, and then print them on the standard output (**stdout**):

Another example of using a pipe for output is the following idiom which guarantees that its output goes to your terminal:

```
% awk '{ print $1 | "cat -u > /dev/tty" }' countries
```

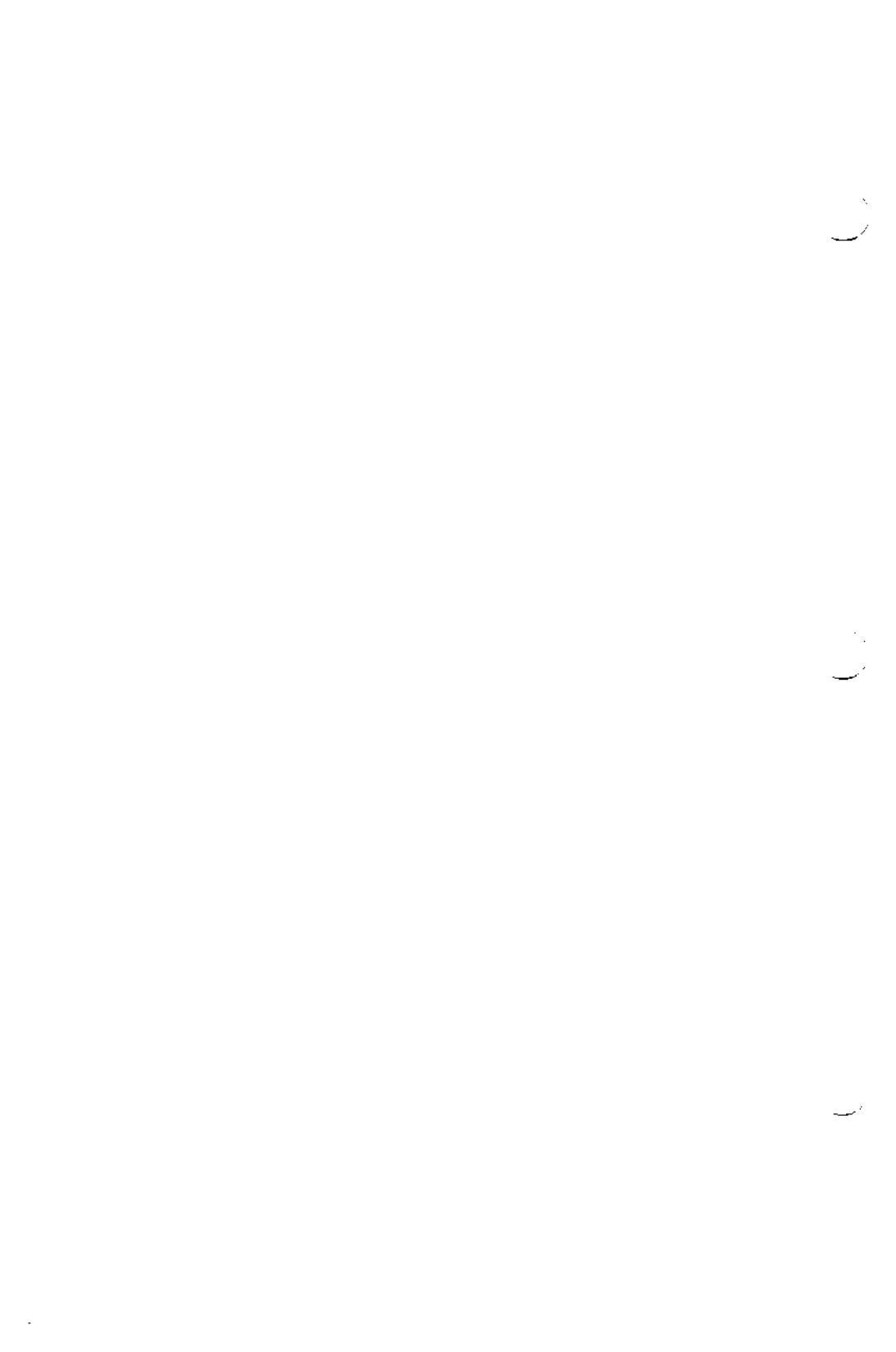
Only one output statement to a pipe is permitted within an **awk** program.

In all output statements involving redirection of output, the files or pipes are identified by their names but they are created and opened only once in the entire run.

Chapter 2
SED: A Stream Editor

CONTENTS

1. Introduction	1
2. Overall Operation	2
2.1 Command Line Options	2
2.2 Usage	3
2.3 Editing Command Syntax	4
2.4 Command Application Order	5
2.5 Pattern Space	5
3. Addressing	6
3.1 Line Number Addresses	6
3.2 Context Addresses	7
3.3 Examples	9
4. SED Editing Commands	11
4.1 Line-Oriented Commands	11
4.2 The Substitute Command	15
4.3 Input/Output Commands	18
4.4 Multiple Input Line Commands	20
4.5 Hold and Get Commands	21
4.6 Flow of Control Commands	23
4.7 Miscellaneous Commands	25



Chapter 2

SED: A Stream Editor

1. Introduction

The UniPlus⁺ stream editor (**sed**) is a noninteractive text editor. It is especially useful for:

- Editing large files that cannot be contained in a buffer. The size of a file to be edited with **sed** is limited only by the amount of secondary storage, which must be able to hold both the input and output files at the same time. Only a few lines of the current input file are in physical (volatile) memory at one time, and no temporary files (buffers) are used.
- Performing complicated editing sequences on any size file. The **sed** editor is most commonly used in shell scripts, where complicated editing actions can be stored, edited, and applied to the input file(s) as a command.
- Efficiently performing multiple global editing commands in one pass. The **sed** program running from a command file is faster and more efficient than an interactive editor like **ex**, even when **ex** is also running from a command file. However, **sed** does not provide certain commands provided by an interactive editor. For example, **sed** does not provide relative addressing. Because it operates on one line at a time, **sed** cannot move backward or forward relative to the current line in a file. In addition, **sed** does not inform you about the effects of your commands, or allow you to immediately undo them.

SED

2. Overall Operation

By default, **sed** copies standard input to standard output, performing zero or more editing actions on each line before writing it to the output. Editing actions are specified by **sed** editing commands, described below. The lines to be affected by these commands are specified by addresses, either context addresses or line numbers.

Input files are never modified; the changes are scrolled on standard output. If output is redirected to a file then the new file contains the modifications created by your editing actions.

2.1 Command Line Options

The **sed** editor is invoked via the following prototype command line:

```
sed [-n][-e script][-f sfile][file(s)]
```

The options are as follows:

- n** (no-copy). Copy only those lines explicitly specified either by **p** (print) commands or **p** arguments after **s** (substitute) commands.
- e** (expression). The *script* argument is an 'expression' (inline **sed** command(s) using the syntax of regular expressions and enclosed in single or double quotes) to be run on the input stream. There may be more than one **-e** (with its corresponding expression) on a command line. If the newlines are escaped, there may be more than one line in an expression.
- f** (source file). The *sfile* argument is a file containing **sed** commands — one to a line.

2.2 Usage

Editing commands are provided on the **sed** command line. They can either be embedded inline (with the **-e** option) or enclosed in a file and provided as an argument to the **-f** option. The following are examples of **sed** usage.

```
% sort input.file | sed -e 's/.dc/.dec./' -e 's/.3b/.u3b./'
```

(Sorts the contents of *input.file*, and performs substitutions on the first instance of **.dc** and **.3b** in each line; the results are written to standard output.)

```
% sed -e 's/\.dc/.dec./\
s/\.3b\./.u3b./' input.file
```

(Performs the same substitutions as previous command on *input.file*; the results are written to standard output.)

```
% sed -e 's/,/ /g' input.file > output.file
```

(Every comma in *input.file* is replaced with a space; the modified file is contained in *output.file*.)

If you contain the following **sed** commands in a file named *cmd.file*:

```
s/\.dc/.dec./g
s/\.3b\./.u3b./
s/,/ /g
```

Then you can use the following syntax:

SED

```
% sed -f cmd.file input.file > output.file
```

(Performs substitutions on *input.file*, the modified file is contained in *output.file*.)

You can also use the syntax::

```
% sed -n -f cmd.file input.file
```

(Performs substitutions on *input.file*, only the modified lines are written to standard output.)

Before any input file is opened, all editing commands are compiled in the order encountered (which is also the order in which they are attempted at execution time), into a form that will be moderately efficient during the execution phase (when the commands are actually applied to lines of the input file).

2.3 Editing Command Syntax

The general editing command syntax is:

```
[address1, address2] command [arguments]
```

The square brackets indicate that the enclosed argument is optional, although arguments may be required or optional according to the command given. Addresses may be line number(s) or contextual; one or both addresses may be omitted. Any number of blanks or tabs may separate addresses from the command, and blanks and tab characters at the beginning of lines are ignored.

2.4 Command Application Order

Commands are applied one at a time, in the order encountered; the input to each command is the output of all preceding commands.

This default linear ordering can be changed by the **t** (test substitution) and **b** (branch) flow-of-control commands. When the order of application is changed by these commands, it is still true that the input line to any command is the output of any previously applied commands.

2.5 Pattern Space

The range of pattern matches is called the pattern space. Ordinarily, pattern space is one line of the input text, but more than one line can be read into the pattern space by using the **next** command (**n**).

3. Addressing

Input file lines to be affected by your editing commands are specified by addresses. These addresses can be either line numbers, or context addresses. If no address is present the command is applied to every line in the input file.

Multiple commands can be applied to a single address (or address pair) by grouping commands with braces in the following format:

```
address(es) {  
    command  
    command  
}
```

3.1 Line Number Addresses

A line number is a decimal integer (greater than or equal to 1) that is incremented (by an internal counter) as each line is read from the input. A line number address matches the input line that causes the internal counter to equal the address line number. As a special case, the \$ character matches the last line of the last input file.

■ **Note:** The line counter runs cumulatively through multiple input files. It is not reset when a new consecutive input file is opened.

Commands can be preceded by 0, 1, or 2 addresses. The maximum number of allowed addresses (less than or equal to 2) is given with each command description below. It is an error when a command has more addresses than the maximum allowed.

If a command has **zero** addresses it is applied to every line in the input.

If a command has **one** address it is applied to all lines that match that address.

If a command has **two** addresses separated by a comma, it is applied to the first line that matches the first address and to all subsequent lines until (and including) the first line which matches the second address. An attempt is made on subsequent lines to again match the first address, and the process is repeated.

3.2 Context Addresses

A context address is a regular expression enclosed in slashes (/pattern/). Sed recognizes regular expressions that have the following construction:

- [1] An ordinary character is a regular expression and matches that character.
- [2] A circumflex (^) at the beginning of a regular expression matches the null character at the beginning of a line.
- [3] A dollar sign (\$) at the end of a regular expression matches the null character at the end of a line.
- [4] The \n character matches an embedded newline character but not the newline character at the end of the pattern space. (That is, newlines get embedded by using the next command n.)
- [5] A period (.) matches any character except the terminal newline character of the pattern space.
- [6] A regular expression followed by an asterisk (*) matches any number (including zero) of adjacent occurrences of the regular expression it follows.
- [7] A string of characters in square brackets ([]) matches any character in the string and no others. If, however, the first character of the string is a circumflex (^), the regular

expression matches any character except the characters in the string and the terminal newline character of the pattern space. The circumflex is the only metacharacter recognized within the square brackets. If '[' needs to be in the set of square brackets, it should be the first nonmetacharacter. For example:

[...]	Includes '['
[^...]	Does not include '['

- [8] A concatenation of regular expressions is a regular expression which matches the concatenation of strings matched by the components of the regular expression.
- [9] A regular expression between the sequences '\(' and '\)' is identical in effect to the unadorned regular expression, but has side effects which are described under the substitute command (s command) below.
- [10] The expression '\d' (where d is a digit, 0 through 9) refers to the string of characters found earlier in the same pattern by an expression using the '\(' and '\)' construction. The '\(' and '\)' sequences perform just as those in the other UniPlus⁺ editors, and are used to establish 'fields' or sections in a line of text (or all lines of text) in a file.

For example, suppose a file contained the following list of names:

```
Dick Powell
William Powell
Eleanor Powell
Jane Powell
```

The following expression will reverse the order of the names, placing a comma and a space between each first name and last name:

```
s/\([A-Z].*\) \([A-Z].*\)/\2, \1
```

For example, the following expression matches a line beginning with two repeated occurrences of the same string:

```
^3(.*)\1
```

- [11] A null regular expression standing alone (e.g., //) is equivalent to the previous compiled regular expression.
- [12] Special characters (**^** , **\$** , ***** , **[** , **** , **/**), when used as literal characters, must be preceded by a backslash (****).
- [13] For a context address to match, the whole pattern within the input address must match some portion of the pattern space.

3.3 Examples

The following shows a sample input text and the output of a **sed** command using line number addressing.

Create a text file named *input.file* that contains the following lines. (Except where noted, all examples in this chapter use the following text.)

```
In Xanadu did Kubla Khan
A stately pleasure dome decree:
Where Alph, the sacred river, ran
Through caverns measureless to man
Down to a sunless sea.
```

The command:

```
% sed -e '2q' input.file
```

copies the first two lines of the input and quits. The output on your screen will be:

SED

In Xanadu did Kubla Khan
A stately pleasure dome decree:

On the same input text, the following indicates the matches resulting from several **sed** commands using context addressing.

/an/	-- matches lines 1, 3, and 4
/an.*an/	-- matches line 1
/^an/	-- matches no lines
/./	-- matches all lines
/\./	-- matches line 5
/r*an/	-- matches lines 1, 3, and 4 (number = 0)
/\ (an\).*1/	-- matches line 1.

4. SED Editing Commands

Sed commands consist of a single alphabetic character. In the following command summaries, the maximum number of allowable addresses are shown in parentheses preceding the single character command name. Possible arguments are enclosed in angle brackets (<>), and a description of each command is given. Parentheses and angle brackets are not part of the command syntax.

4.1 Line-Oriented Commands

The commands in this section apply to whole lines of the input.

Command	Description
(2)d	(delete). The d command deletes from the file (does not write to the output) those lines matched by its addresses. It also has the side effect that no further commands are attempted on the corpse of a deleted line. As soon as the d command is executed, a new line is read from the input, and the list of editing commands is restarted from the beginning on the new line.
(2)n	(next). The n command reads the next line from the input, replacing the current line, and the current line is written to the output. The list of editing commands is continued following the n command.

SED

(1)a\
<text>

(append). The **a** command causes the text argument (<text>) to be written to the output after the line matched by its address.

The **a** command is inherently multiline; **a** must appear at the end of a line, and <text> may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (\) immediately preceding the newline character.

The <text> is terminated by the first newline character not immediately preceded by a backslash. Once an **a** command is successfully executed, text will be written to the output regardless of what later commands do to the line which triggered it. Even if that line is deleted, text will still be written to the output.

The <text> is not scanned for address matches, and no editing commands are attempted on it. The **a** command does not cause a change in the line number counter.

(1)i\
<text>

(insert). The **i** command causes the text argument (<text>) to be written to the output before the line matched by its address.

The **i** command is inherently multiline; **i** must appear at the end of a line, and `<text>` may contain any number of lines. To preserve the one-command-to-a-line fiction, interior newline characters must be hidden by a backslash character (`\`) immediately preceding the newline character. The `<text>` is terminated by the first newline character not immediately preceded by a backslash.

Once an **i** command is successfully executed, text will be written to the output regardless of what later commands do to the line which triggered it. Even if that line is deleted, text will still be written to the output.

The `<text>` is not scanned for address matches, and no editing commands are attempted on it. The **i** command does not cause a change in the line number counter.

(2)**c**\
`<text>`

(change). The **c** command deletes lines selected by its addresses and replaces them with the lines in the text argument (`<text>`).

Like **a** and **i**, **c** must be followed by a newline character hidden by a backslash; interior newline characters in `<text>` must be hidden by backslashes. The **c** command may have two addresses, and therefore select a range of lines. If it does, all lines in the range are deleted, but only one copy of text is written to the output, not one copy per line deleted.

SED

As with **a** and **i**, **<text>** is not scanned for address matches, and no editing commands are attempted on it. It does not change the line number counter.

After a line has been deleted by a **c** command, no further commands are attempted on the corpse. If text is appended after a line by **a** or **r** commands and the line is subsequently changed, the text inserted by the **c** command will be placed before the text of the **a** or **r** commands (the **r** command is described later).

Leading blanks and tabs disappear from text inserted in the output by the **a**, **i**, and **c** commands. To get leading blanks and tabs into the output, precede the first desired blank or tab by a backslash. The backslash will not appear in the output.

The following example shows line-oriented **sed** commands used on the standard input example shown in Section 3.3.

If your *cmd.file* contains the lines:

```
n
a\
XXXX
d
```

The command:

```
% sed -f cmd.file input.file > output.file
```

produces an *output.file* that contains the following lines:

```
In Xanadu did Kubla Khan
XXXX
Where Alph, the sacred river, ran
XXXX
Down to a sunless sea.
```

4.2 The Substitute Command

The substitute command uses the following syntax:

(2)**s** <pattern> <replacement> <flags>

The **s** command replaces the part of a line selected by <pattern> with <replacement>. It can be read 'substitute for pattern, replacement'. The command arguments are described as follows:

- <pattern> The pattern argument is a regular expression, like the patterns in context addresses. The only difference between <pattern> and a context address is that the context address must be delimited by slash (/) characters; <pattern> may be delimited by any character other than space or newline. By default, only the first string matched by <pattern> is replaced unless the **g** flag (below) is invoked.
- <replacement> The replacement argument begins immediately after the second delimiting character of <pattern> and must be followed immediately by another instance of the delimiting character (thus there are exactly three instances of the delimiting character). The

<replacement> is not a pattern, and the characters which are special in patterns do not have special meaning in <replacement>. Instead, the following other characters are special:

& is replaced by the string matched by <pattern>.

\d is replaced by substring **d** (**d** is a single digit), matched by parts of <pattern>, and enclosed in '\(' and '\)'. If nested substrings occur in <pattern>, substring **d** is determined by counting opening delimiters '\('). As in patterns, special characters may be made literal characters by preceding them with backslash (\).

<flags>

The flags argument may contain the following:

g (global). Substitute <replacement> for all nonoverlapping instances of <pattern> in the line. After a successful substitution, the scan for the next instance of <pattern> begins just after the end of the inserted characters. Characters put into the line from <replacement> are not rescanned.

p (print). Print the line if a successful replacement was done. The **p** flag causes the line to be written to the output if and only if a substitution was actually made by the **s** command. If several **s** commands, each

followed by a **p** flag, successfully substitute in the same input line, multiple copies of the line will be written to the output — one for each successful substitution. (write to file). Write the line to a file if a successful replacement was done. A single space must separate **w** and **<file>**. The **w** flag causes lines which are actually substituted by the **s** command to be written to a file named by **<file>**. If **<file>** exists before **sed** is run, it is overwritten; if not, it is created. The possibilities of multiple, somewhat different copies of one input line being written are the same as for **p**. A maximum of ten different file names may be mentioned after **w** flags and **w** commands.

The command:

```
% cat input.file | sed -e 's/to/by/w changes'
```

produces a file named *changes* that contains only the lines that were changed:

Through caverns measureless by man
Down by a sunless sea.

If the no-copy option is in effect (using the **-n** option on the **sed** command line), then the same effect can be achieved with the command:

SED

% **sed -n -e 's/to/by/p'** *input.file* > *changes*

If your command file contains the line:

s/[.,;?:]/*P&*/gp

Then the command:

% **sed -n -f cmd.file** *input.file*

produces the output:

A stately pleasure dome decree*P.*
Where Alph*P,* the sacred river*P,* ran
Down to a sunless sea*P.*

If the **g** flag is not used, the substitution only takes effect on the first instance of the pattern in a given line. For example, the command:

% **sed -n -e '/X/s/an/AN/p'** *input.file*

causes the substitution to occur only on the first instance of 'an':

In XANadu did Kubla Khan

4.3 Input/Output Commands

Note: Since there is a limit to the number of files that can be opened simultaneously, care should be taken that no more than ten files be mentioned in **w** commands or flags. That number is reduced by one if any **r** commands are present (only one read

file is opened at a time).

Command	Description
(2)p	(print). The <i>print</i> command writes addressed lines to the standard output file. They are written at the time the <i>p</i> command is encountered regardless of what succeeding editing commands may do to the lines.
(2)w <file>	(write to file). The <i>write</i> command writes addressed lines to the file named by <file>. Exactly one space must separate the <i>w</i> and <file>. If the file previously existed, it is overwritten; if not, it is created. The lines are written exactly as they exist when the write command is encountered for each line regardless of what subsequent editing commands may do to them. A maximum of ten different files may be mentioned in write commands and <i>w</i> flags after <i>s</i> commands combined.
(1)r <file>	(read from file). The <i>read</i> command reads the contents of <file> and appends them after the line matched by the address. Exactly one space must separate the <i>r</i> and <file>. The file is read and appended regardless of what subsequent editing commands do to the line which matched its address.

SED

If **r** and **a** commands are executed on the same line, the text from **a** commands and **r** commands is written to the output in the order that the commands are executed.

If a file mentioned by an **r** command cannot be opened, it is considered a null file, not an error, and no diagnostic is given.

If a file *note1* has the following contents:

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan and founder of the Mongol dynasty in China.

then the command:

```
% sed -e '/Kubla/r note1' input.file
```

produces:

In Xanadu did Kubla Khan

Note: Kubla Khan (more properly Kublai Khan; 1216-1294) was the grandson and most eminent successor of Genghiz (Chingiz) Khan and founder of the Mongol dynasty in China.

A stately pleasure dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

4.4 Multiple Input Line Commands

The following three commands, all spelled with capital letters, deal with pattern spaces containing embedded newline

characters. They are intended principally to provide pattern matches across lines in the input. The **P** and **D** commands are equivalent to their lowercase counterparts if there are no embedded newline characters in the pattern space.

Command	Description
(2)N	Append the next input line to the current line in the pattern space. The two input lines are separated by an embedded newline character. Pattern matches may extend across embedded newline characters.
(2)D	Delete first part of the pattern space. Delete up to and including the first newline character in the current pattern space. If the pattern space becomes empty (the only newline character was the terminal newline character), read another line from the input. In any case, begin the list of editing commands again from the beginning.
(2)P	Print first part of the pattern space. Print up to and including the first newline character in the pattern space.

4.5 Hold and Get Commands

The following commands save and retrieve part of the input for possible later use.

SED

Command	Description
(2)h	Hold pattern space. The h command copies contents of the pattern space into a hold area, destroying previous contents of the hold area.
(2)H	Hold pattern space. The H command appends contents of the pattern space to contents of the hold area. Former and new contents are separated by a newline character.
(2)g	Get contents of hold area. The g command copies contents of the hold area into the pattern space destroying previous contents.
(2)G	Get contents of hold area. The G command appends contents of the hold area to contents of the pattern space. Former and new contents are separated by a newline character.
(2)x	Exchange. The x command interchanges contents of the pattern space and the hold area.

For example, if your **sed** command file contains the commands:

```
1h
1s/ did.*/ /
1x
G
s/\n/ :/
```

When applied to *input.file*, this produces:

In Xanadu did Kubla Khan :In Xanadu
 A stately pleasure dome decree: :In Xanadu
 Where Alph, the sacred river, ran :In Xanadu
 Through caverns measureless to man :In Xanadu
 Down to a sunless sea. :In Xanadu

4.6 Flow of Control Commands

These commands do no editing on the input lines but control the application of commands to the lines selected by the address part.

Command	Description
(2)!	(don't). The don't command causes the next command (written on the same line) to be applied to those input lines not selected by the address part.
(2){	(grouping). The grouping command causes the next set of commands to be applied (or not applied) as a block to the input lines selected by the addresses of the grouping command. The first of the commands under control of the grouping may appear on the same line as the { or on the next line. The group of commands is terminated by a matching } standing on a line by itself. Groups can be nested.
(0):<label>	(place label). The label command marks a place in the list of editing commands which may be referred to by b and t commands.

The <label> argument may be any sequence of eight or fewer characters. If two different colon commands have identical labels, a compile time diagnostic will be generated; and no execution attempted.

(2)b<label>

(branch to label). The branch command causes the sequence of editing commands being applied to the current input line to be restarted immediately after the place where a colon command with the same <label> was encountered.

If no colon command with the same label can be found after all editing commands have been compiled, a compile time diagnostic is produced; and no execution is attempted.

A b command with no <label> is a branch to the end of the list of editing commands. Whatever should be done with the current input line is done, and another input line is read. The list of editing commands is restarted from the beginning on the new line.

(2)t<label>

(test substitutions). The t command tests whether any successful substitutions have been made on the current input line; if so, it branches to <label>; if not, it does nothing.

The flag which indicates that a successful substitution has been executed is reset by reading a new input line and executing a t command.

4.7 Miscellaneous Commands

Command	Description
(1)=	The = command writes the line number of the line matched by its address. to standard output.
(1)q	The q command causes the current line to be written to the output (if it should be), any appended or read text to be written, and execution to be terminated.

1

2

3

Chapter 3: LEX

CONTENTS

1. Introduction	1
2. Syntax	5
3. Character Set	7
3.1 Character Classes	7
3.2 Arbitrary Characters	9
3.3 Operators	9
4. Compilation	12
5. Definitions	13
5.1 Repetitions and Definitions	15
6. Rules	16
6.1 Regular Expressions	16
6.2 Optional Expressions	16
6.3 Repeated Expressions	16
6.4 Alternation and Grouping	17
6.5 Context Sensitivity	17
6.5.1 Left Context Sensitivity	19
6.6 Ambiguous Rules	23
7. Actions	25
7.1 yymore() and yyless()	27
7.2 input(), output(c) and unput(c)	29
7.3 yywrap()	30
7.4 REJECT	31
8. Examples	34
9. Summary	36
10. LEX and YACC	39

LIST OF FIGURES

Figure 3.1. Overview of lex	2
Figure 3.2. Lex With Yacc	39

Chapter 3

LEX -

A LEXICAL ANALYZER GENERATOR

1. Introduction

The **lex** is a program generator that produces a program in a general purpose language that recognizes regular expressions. It is designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching.

The **lex** program generator source is a table of regular expressions and corresponding program fragments. The table is translated to a program that reads an input stream, copies the input stream to an output stream, and partitions the input into strings that match the given expressions. As each such string is recognized, the corresponding program fragment is executed.

The recognition of the regular expressions is performed by a deterministic finite automaton generated by **lex**. The program fragments are executed in the order in which the corresponding regular expressions occur in the input stream.

The **lex** written code is not a complete language, but rather a generator representing a new language feature which can be added to different programming languages, called "host languages." For example, one higher-level language is used for recognizing patterns, while a more general purpose language is used for action statements.

Just as general purpose languages can produce code to run on different computer hardware, **lex** can write code in different host languages. The host language is used for the output code generated by **lex** and the program fragments which comprise

LEX

the **lex** source program.

Compatible run-time libraries for the different host languages are provided, making **lex** adaptable to many environments and users.

At present, the only supported host language is the C language, although FORTRAN (in the form of RATFOR) has been available in the past.

The **lex** generator exists on the UNIXTM operating system, but the codes generated by **lex** may be taken anywhere the appropriate compilers exist.

The program generated by **lex** is called **yylex**. The **yylex** program recognizes expressions in an input stream and performs the specified actions for each expression as it is detected. See Figure 3.1.

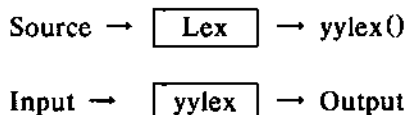


Figure 3.1. Overview of **lex**

For example:

```
%%  
[ \t] + $ ;
```

UNIX is a trademark of AT&T Bell Laboratories.

This example lex source program is all that is required to generate a program to delete all blanks or tabs at the ends of the input lines. The %% delimiter is a lex convention to mark the beginning of the "rules" — the pattern matching expressions. The rule itself, `| \t|+$;`, matches one or more instances of the characters blank and tab. The brackets enclose the character class consisting of blank and tab; the + indicates "one or more instance of the previous character(s) or character class" and the \$ indicates "end of line." No "action" is specified, so the `yylex()` program, (generated by lex), ignores these characters. Everything else is copied.

Consider this next example:

```
%%
| \t|+$ ;
| \t|+ ;
```

The coded instructions in `yylex` scan for both rules at once. Once a string of blanks or tabs is recognized, `yylex` determines if the string is followed by a newline character. If it is, then the first rule has been matched so the the corresponding action is performed — `yylex` does not copy the string to output. The second rule matches strings of one or more blanks and tabs not already satisfying the first rule, and causes `yylex` to replace a string of one or more blanks and tabs with a space.

The lex program generator can be used alone for simple transformations or for analysis and statistics gathering on a lexical level. The lex generator can also be used with a parser generator to perform the lexical analysis phase.

In `yylex`, the program generated by lex, the actions to be performed as each regular expression is found are gathered as cases of a switch. The automaton interpreter directs the control flow. It is possible to insert either declarations or additional statements in the routine containing the actions and to add sub-routines outside this action routine, should you need to do so.

LEX

The lex program generator is not limited to one character look-ahead. For example, if there are two rules, one looking for "ab" and another for "abcdefg" and the input stream is "abcdefh," lex recognizes "ab" and leaves the input pointer just before "cdefh."

2. Syntax

The general format of **lex** is:

```
{definitions}
%%
{rules}
%%
{user subroutines}
```

where the definitions and the user subroutines are often omitted. The first **%%** is required to mark the beginning of the rules, but the second **%%** is optional. The absolute minimum **lex** program is:

```
%%
```

This **lex** source would generate a program that copies the input to output unchanged.

In the outline of **lex** programs shown above, the rules consist of two parts:

1. A left column with regular expressions
2. A right column with actions and program fragments to be executed when the expressions in the left column are recognized.

For example:

```
integer printf("found keyword INT");
```

The example rule above gives the instructions to look for the string "integer," and, when found, output the statement "found keyword INT." In this example, since the host procedural language is C, the C language library function **printf** is used to print the string.

LEX

The end of the expression is indicated by the first blank or tab character. If the action is a single C language expression, it can just be given in the right column, as illustrated in the example. If the action is compound or requires more than one line, it should be enclosed in braces.

Consider the following example:

colour	printf("color");
mechanise	printf("mechanize");
petrol	printf("gas");

This lex source segment could be used to generate a program to change a number of words from British to American spelling. Actually, these rules would have to be changed somewhat to be useful. This is because were the word "petroleum" in the input stream, the program generated by this segment would cause it to be changed to "gaseum."

3. Character Set

The programs generated by **lex** handle character I/O only through the routines **input()**, **output()**, and **unput()**. The character representation provided in these routines is accepted by **lex** and used to return values in **yytext()**. (See the section entitled "**input()**, **output()** and **unput()**," in this Chapter for more information on these routines.)

For internal use, a character is represented as a small integer. If the standard library is used, a character's value is equal to the integer value of the bit pattern representing the character on the host computer, i.e., the character "A" has the value \101 (octal) in ASCII.

Of course, you need not use the integer value of a character to access the value. The character "a" is represented in the same form as the character constant 'a'. If this interpretation is changed by providing I/O routines that translate the characters, **lex** must be given a translation table that is in the "definitions" section of the source, and this translation table must be bracketed by lines containing only **%T**. The translation table, then, contains lines of the form:

```
%T
{integer} {character string}
%T
```

which indicate the value associated with each character.

3.1 Character Classes

Classes of characters can be specified using the operator pair "[and]". For example, the construction **[label]** matches a single character which may be "a," "b" or "c."

Within square brackets, most operator meanings are ignored. Only three characters are special:

LEX

1. \
2. -
3. ^

The “-” character indicates “ranges.” For example,

[a-z0-9<>_]

indicates the character class containing all the lowercase letters (a to z), digits (0 through 9), angle brackets (< and >) and the underline character (_).

Using “-” between any pair of characters which are not both uppercase letters, both lowercase letters, or both digits is sometimes acceptable to **lex**, but this is implementation dependent. Therefore, if such a range is declared, **lex** will issue a warning message. One reason for this is that [0-z] in ASCII is many more characters than is in EBCDIC.

If it is desired to include the character “-” in a character class, it should either be first or last within “[].” For example:

[-+0-9]

matches ALL digits, (0 through 9) and the two symbols - and +.

If the ^ operator appears as the first character after the left bracket, **lex** will ignore the characters within the brackets, therefore matching all characters EXCEPT those within the designated character class range. If an operation is to be performed on recognition of a string expressed using this construction, it will be done on strings OTHER THAN those within the brackets. For example:

[^abc]

matches all characters EXCEPT "a," "b" or "c" including all special and control characters. Also:

`[^a-zA-Z]`

matches any character that is NOT a letter — (neither in the range a through z nor A through Z).

The `\` character provides the escapes within character class brackets. For example:

`[a-z*]`

matches all lower case letters (a through z) and the character `"*"`.

3.2 Arbitrary Characters

The operator `"."` instructs `lex` to match ANY character — except newline.

All characters and ranges can be designated using the octal representations of those characters. This method, however, is difficult to read and, most likely, unportable. Nonetheless, the following character class range:

`[\40-\176]`

matches all printable ASCII characters from octal 40 (blank) to octal 176 (tilde — `~`).

3.3 Operators

The operator characters are

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

and if they are to be used as text characters, an appropriate "escape" should be used, i.e., to get the character `"\"`, you must escape its significance as an "operator" and can do so

LEX

easily with another backslash, "\\." (For more information on "escaping," refer to the *UniPlus⁺ User Guide*, chapter on the C Shell, under the heading "Quoting Mechanisms.")

The quotation mark operator, `"`, indicates that whatever characters follow — up to a second `"` character — are to be taken as text characters without any "magic" meaning or operator significance. The quotation mark, then, is another way to "escape" the special meaning of a character. Thus:

`xyz++"`

matches the string `"xyz++"` wherever it appears. Of course, it is harmless, though unnecessary, to quote an ordinary text character, so the expression:

`"xyz++"`

is equivalent to the one which only quoted the `++`. However, by quoting every character being used as a text character, the user can avoid remembering the list of current operator characters, and is safe should further extensions to `lex` lengthen the list.

Another use of the quoting mechanism is to get a blank into an expression. Normally, as explained above, blanks or tabs end a rule. Any blank character not contained within `[]` MUST be quoted.

Several C language "escapes" with `\` are recognized:

<code>\n</code>	newline
<code>\t</code>	tab
<code>\b</code>	backspace
<code>\\</code>	backslash

Since newline is illegal in an expression, \n must be used.

LEX

4. Compilation

The following are the two steps involved in compiling a **lex** source file:

1. The **lex** source must be turned into a generated program in the “host general purpose language.” The generated program is in a file named **lex.yy.c**.
2. Then that program must be compiled and loaded, usually with a library of **lex** subroutines. The I/O library is defined in terms of the C language standard library. On the UNIX operating system, the library is accessed by the loader flag **-ll**. In this case, an appropriate set of commands is

```
lex inputfile  
cc lex.yy.c -ll
```

The resulting program is placed in the file *a.out* for later execution.

(To use **lex** with **yacc**, see the section entitled “LEX and YACC.”) Although the default **lex** I/O routines use the C language standard library, **lex** routines such as **input**, **output** and **unput** do not. Therefore, if your own versions of these routines are given, the library is avoided.

5. Definitions

The basic format of a **lex** source is:

```
{definitions}  
%%  
{rules}  
%%  
{user routines}
```

In addition to the **rules**, there are options to define variables. Variables can go either in the **definitions** section or in the **rules** section.

Remember **lex** is generating the **rules** into a program, and any source NOT intercepted by **lex** is copied into the program generated. Also:

1. Any line not part of a **lex** rule or action and that begins with a blank or tab is copied into the **lex** generated program.
2. Any line not part of a **lex** rule or action, that begins with a blank or tab and is found PRIOR to the first %% delimiter is "external" to any function in the code.
3. Any line not part of a **lex** rule or action that begins with a blank or tab and is found IMMEDIATELY AFTER the first %% , appears in an appropriate place for declarations in the function written by **lex** which contains the actions. This material MUST look like program fragments and should precede the first **lex** rule.
4. Lines that begin with a blank or tab, and that contain a comment, are passed through to the generated program. This can be used to include comments in either the **lex** source or the generated code. The comments should follow the host language convention.
5. Anything included BETWEEN lines containing only %{ and %} is copied to output. The delimiters are discarded.

LEX

This format permits entering text like preprocessor statements that must begin in column 1 or copying lines that do not look like programs.

6. Anything AFTER THE THIRD %% delimiter, regardless of formats, etc., is copied to output AFTER the lex output.

Definitions intended for lex are given before the first %% delimiter. Any line in this section not contained between %{ and %} and beginning in column 1 is assumed to define lex substitution strings. The format of such lines is

name translation

This facility enables the string given as "translation" to be associated with the "name." The "name" and "translation" MUST be separated by at least one blank or tab, and the "name" MUST begin with a letter. The "translation" can be called by the {name} syntax in a rule. Using {D} for the digits and {E} for an exponent field. For example:

D	[0-9]
E	[DEde][--+]?{D}+
%%	
{D}+	printf("integer");
{D}+"."{D}*({E})?	
{D}*."{D}+({E})?	
{D}+{E}	printf("real");

This example abbreviates rules to recognize numbers. The first two rules for real numbers both require a decimal point and contain an optional exponent field. The first requires at least one digit before the decimal point ({D}+"."{D}*({E})?), and the second requires at least one digit after the decimal point ({D}*."{D}+({E})?). To correctly handle the the FORTRAN expression "35.EQ.I," which does NOT contain a real number, a context-sensitive rule such as:

```
[0-9]+/".EQ    printf("integer");
```

could be used, in addition to the normal rule for integers.

The "definitions" section may also contain other commands including the selection of a host language, a character set table, a list of start conditions or adjustments to the default size of arrays within lex itself for larger source programs.

5.1 Repetitions and Definitions

The operators {} specify EITHER:

- repetitions (if they enclose numbers), OR
- definition expansion (if they enclose a name).

For example:

```
{digit}
```

looks for a predefined string named "digit" and inserts it at that point in the expression. The definitions are given in the first part of the lex input — BEFORE the rules.

The expression:

```
a{1,5}
```

looks for 1 to 5 occurrences of "a."

An initial % is not an ordinary character, but has a special meaning to lex as the the separator for source program segments.

6. Rules

6.1 Regular Expressions

The regular expressions in *lex* function just as do those in the UNIX Operating System text editors (i.e., *vi*, *ed*, etc.). A *regular expression* specifies a set of strings to be matched. It contains "text characters," which match characters in the input stream, and "operator characters," which, together with those "text characters," express a string which is to be recognized before the action in the right hand column takes place.

Letters of the alphabet and digits are always text characters. For example:

integer

matches the string "integer" wherever it appears, and the expression:

a57D

looks for the string "a57D."

6.2 Optional Expressions

The operator **?** indicates an optional element of an expression. Thus:

ab?c

matches either "ac" or "abc".

6.3 Repeated Expressions

Repetitions of classes are indicated by the operators ***** and **+**. The expression:

a*

matches zero (0) or more consecutive "a" characters. The expression:

a+

matches one (1) or more instances of "a" characters. The expression:

[a-z]+

matches ALL strings of lowercase letters. The expression:

[A-Za-z][A-Za-z0-9]*

matches ALL alphanumeric strings which have a leading alphabetic character. This is a typical expression for recognizing identifiers in computer languages.

6.4 Alternation and Grouping

The operator | indicates *alternation*. For example:

(abcd)

matches either "ab" OR "cd." The parentheses are used here for grouping, only. They are not required in such a simple and clear-cut example, but are often used for clarity or to force correct interpretation of more complex expressions. For example:

(abcd+)?(ef)*

matches the strings as "abefef," "efefef," "cdef" or "cddd;" but NOT "abc," "abcd" or "abcdef."

6.5 Context Sensitivity

The **lex** program recognizes a small amount of surrounding context. The two simplest operators for this are ^ and \$.

LEX

As in the UNIX Operating System text editors, if the first character of an expression is `^`, the expression is matched only if found as the FIRST character on a line — either after a newline character or at the beginning of the input stream.

NOTE: Do not confuse the use of the `^` operator within brackets (`[]`) which instructs `lex` to match any character EXCEPT those designated character class range.

If you want to `lex` to find occurrences of a particular range of characters — but only if they occur as the first character on a line, you must use the `^` operator on the OUTSIDE of the brackets. For example, the expression:

`^[0-9]`

matches lines whose first character is a digit, 0 through 9; while the expression:

`^[^0-9]`

matches lines whose first character is NOT a digit 0 through 9.

The operator `$`, is only matched at the end of a line — immediately followed by newline. This operator is a special case of the `/` operator character which indicates “*trailing context*.” The expression

`ab/cd`

matches the string “`ab`” ONLY if followed by “`cd`.” Therefore, the expression:

`ab$`

could be expressed:

`ab/\n`

That is, the use of the `$` operator could be interpreted as an instruction to match the character(s) ONLY when followed by a

newline.

"Left context" is handled in lex by "start conditions." If a rule is only to be executed when the lex automaton interpreter is in "start condition" x , the rule should be prefixed by the angle bracket operator characters:

$\langle x \rangle$

If we considered "being at the beginning of a line" to be start condition "ONE," then the \wedge operator would be equivalent to

$\langle ONE \rangle$

See the sections entitled "Left Context Sensitivity," "Examples" and "Summary" for further explanation and illustration of "start conditions."

6.5.1 Left Context Sensitivity

Sometimes it is desirable to have several sets of lexical rules to be applied at different times in the input. For example, a compiler preprocessor might distinguish preprocessor statements and analyze them differently from ordinary statements. This requires "sensitivity" to prior context. There are several ways of handling such occurrences. For example, the \wedge operator is a "prior context operator" because it must recognize the immediately preceding left context in order to discern if a character appears at the beginning of a line, just as the $\$$ operator must recognize the immediately following right context in order to discern if a character appears at the end of a line.

Adjacent left context could be extended to produce a facility similar to that for adjacent right context, but it is unlikely to be as useful since often the relevant left context appeared some time earlier such as at the beginning of a line.

This section describes three ways of dealing with different environments:

LEX

1. A simple use of flags (when only a few rules change from one environment to another),
2. use of "start conditions" on rules, and
3. the possibility of making multiple lexical analyzers all run together.

In each case, there are rules that recognize the need to change the environment in which the following input text is analyzed and a parameter is set to reflect the change.

The simplest way of doing this is by use of a flag explicitly tested by the user's action code. If done in this way, `lex` is not involved at all. It may be more convenient, however, to have `lex` "remember" the flags as "start conditions" on the rules. Any rule may be associated with a "start condition." That rule, then, would only be recognized when `lex` is in that start condition. The current start condition may be changed at any time.

If the sets of rules for the various environments are very different, clarity may be best achieved by writing several distinct lexical analyzers and switching from one to another as desired.

Consider the following example which is written to generate a program to perform the following actions:

- Copy the input to the output.
- Change the word "magic" to "first" on every line which begins with the letter "a."
- Change "magic" to "second" on every line which began with the letter "b."
- Change "magic" to "third" on every line which began with the letter "c."

All other words and all other lines are left unchanged.

These rules are so simple that the easiest way to do this job is with a "flag." For example:

```
int flag.
%%
^a    {flag = 'a'; ECHO;}
^b    {flag = 'b'; ECHO;}
^c    {flag = 'c'; ECHO;}
\n    {flag = 0 ; ECHO;}
magic {
    switch (flag)
    {
        case 'a': printf("first"); break;
        case 'b': printf("second"); break;
        case 'c': printf("third"); break;
        default: ECHO; break;
    }
}
```

To handle the same problem using "start conditions," begin by introducing each start condition to lex in the "definitions" section with a line reading:

```
%Start  name1 name2 ...
```

where the conditions, (name1, name2, etc.), may be named in any order. The word "Start" may be abbreviated to "s" or "S."

Then, to reference the conditions use angle brackets (< >) brackets:

```
<name1>expression
```

The rule illustrated above will ONLY be recognized when lex is in the "start condition" name1. To enter that "start

LEX

condition," execute the following action statement:

```
BEGIN name1;
```

The action statement:

```
BEGIN 0;
```

resets the initial condition of the lex automaton interpreter.

A rule may be active in several start conditions.

```
<name1,name2,name3> expression
```

is a legal expression.

Any rule NOT beginning with the <> prefix operator is ALWAYS active.

The following example illustrates the use of "start conditions:"

```
%START AA BB CC
%%
^a      {ECHO; BEGIN AA;}
^b      {ECHO; BEGIN BB;}
^c      {ECHO; BEGIN CC;}
\n      {ECHO; BEGIN 0;}
<AA> magic  printf("first");
<BB> magic  printf("second");
<CC> magic  printf("third");
```

Obviously, the above is a re-write of the previous example — the problem-solving logic is exactly the same. However, in this case, lex, has been instructed to do the work instead of the host language code.

6.6 Ambiguous Rules

The **lex** program can handle ambiguous specifications. When more than one expression can match the current input, **lex** chooses as follows:

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

For example, using the following rules:

```
integer  keyword action ...;
[a-z]+  identifier action ...;
```

if the input were "integers," **lex** would interpret the input as an identifier because "[a-z]+" matches all eight characters (including the final "s"), while "integer" matches only seven characters.

If the input were "integer," both rules would match the seven characters. In that case, **lex** would select the keyword rule because it was given first. If the input were anything shorter, (e.g., "int"), the input would not match the expression "integer." It would, however, match the "[a-z]+" expression, so the identifier interpretation would be used.

The principle of preferring the longest match makes rules containing expressions like `.*` dangerous. For example:

```
'.*'
```

appears to instruct **lex** to find a match for a string in single quotes. However, it is an instruction for the program to read far ahead looking for a distant single quote. For example, if the above expression were given the following input:

```
'first' quoted string here, 'second' here
```

LEX

the expression would match almost the entire input line:

'first' quoted string here, 'second'

which is most likely NOT the desired result. A better rule to match strings within single quotes might be:

'[^\n]*'

which, given the same input, will match **"'first'."**

The consequences of errors like this are greatly lessened by the fact that the dot (.) operator does NOT match newline. Expressions like .* stop on the current line.

NOTE: Do NOT try to defeat the protection of . not matching the newline character with expressions such as: **[^\n]+** or an equivalent because the program generated by **lex** will then try to read the entire input file causing internal buffer overflows.

7. Actions

When an expression written as above is matched, **lex** executes the corresponding "action." The following describes the features of **lex** used to write "actions."

NOTE: The default "action" for **lex** copies input to output, and is performed on ALL strings not otherwise matched. Therefore, a rule which merely copies, can be omitted. If you want to absorb the entire input, WITHOUT producing any output, you **MUST** provide rules to match everything. (When **lex** is being used with **yacc**, this is the normal situation.) In other words, by default, a character combination in input, which was omitted from the rules, will be printed on the output.

One of the simplest things that can be done is to ignore the input. To accomplish this, use a semicolon ";" (semicolon is the C language "NULL statement") as the action.

The following rule:

```
[ \t\n] ;
```

causes the spacing characters, (i.e., blank, tab and newline), to be ignored since it gives the NULL statement as its associated action.

The action character, |, represents the instruction to use the action designated for the next rule for the current rule as well. For example:

```
" " |
"\t" |
"\n" ;
```

LEX

This example instructs **lex** to ignore the spacing characters — as did the previous example. The first line gives the rule “match “ ” characters” and instructs the program to perform the action indicated for the next rule. Then, the second line gives the rule “match “\t” characters” and instructs the program to perform the action indicated for the next rule. Finally, the third line gives the rule “match “\n” characters,” and gives the action “;” — the NULL statement. Therefore, the action for all three rules is the NULL statement.

In more complex actions, you may often want to know the actual text that matched a regular expression. The **lex** program leaves this text in an external character array. Consider the following example:

```
[a-z]+  printf("%s", yytext);
```

This example illustrates a way of accessing the characters matching a regular expression. Using this example, the **rule** given is to find the strings matching the regular expression “[a-z]+” and the **action** is to print those strings in the character array **yytext[]** using the C language function **printf**.

The **printf** function accepts a format argument and data to be printed. Still using this example, the format is “%s” (print string). The % character indicates data conversion, and s indicates data type string — in this case, the character array, **yytext[]**. This places the matched string on the output.

The action of printing the strings matching the regular expressions is so common, that it may be written simply as “ECHO.” For example:

```
[a-z]+  ECHO;
```

This example accomplishes the same action as the previous example using the **printf** statement.

Even though the default action is to copy input to output, the ECHO facility is included explicitly to provide a more “discriminating” copy function. For example, a rule that matches `read`, will normally match ALL instances of `read`, even those contained in other words (`bread`, `treadmill`, etc.). To avoid this, a rule of the form “[a-z]+” is needed.

The `lex` routine `yyleng` is used to facilitate counting both the number of words in the input, and the number of characters in those words. Consequently, this routine enables access to the last instance of a string matched, or even the last character of that string.

Consider the following example:

```
[a-zA-Z]+ {words++; chars += yyleng;}
```

This instruction takes the strings which match the regular expression `[a-zA-Z]+` and accumulates the number of characters in these strings in `chars`. Then, the following action instruction:

```
yytext[yyleng-1]
```

could be used to access the last character in the string matched.

7.1 `yymore()` and `yless()`

Occasionally, a `lex` action may decide that a rule has not recognized the correct span of characters. Two routines are provided to aid with this situation:

1. `yymore()` This routine instructs `lex` to tack the next input expression recognized on to the end of this input. Normally, the next input string would overwrite the current entry in `yytext`.
2. `yless(n)` This routine instructs `lex` to retain in `yytext` ONLY *n* number of those characters resulting from the current expression. Further

LEX

characters previously matched are returned to the input. This provides the same sort of look ahead offered by the / operator, though in a very different form.

Consider a language that defines a string as a set of characters between quotation marks ("), and provides that to include a the " character in a string, that character MUST be preceded by a \. The regular expression which matches that is somewhat confusing, so that it might be preferable to write the following:

```
\["~]* {
    if (yytext[yylen-1] == "\\")
        yymore();
    else
        ... normal user processing
}
```

The above lex segment will, when it finds the string:

"abc\"def",

first match the five characters "abc\; then call the `yymore()` routine which will cause the next part of the string "def to be tacked on the end of the input. Note that the final quote terminating the string should be picked up in the code labeled "normal processing."

The function `yyless()` might be used to reprocess text. For example:

```
==-[a-zA-Z] {
    printf("Operator (==) ambiguous\n");
    yyless(yylen-1);
    ... action for == ...
}
```

This lex segment will print a message, treat the operator as "==" and return the letter found after the operator to the

input stream. However, you might want to treat this syntax as “=-a.” In that case:

```

==-[a-zA-Z] {
    printf("Operator (==) ambiguous\n");
    yyless(yylen-2);
    ... action for = ...
}

```

This lex segment will print a message, treat the operator as “=” and return “-a ” to the input stream.

NOTE: It is possible to avoid the misinterpretation of operators by re-writing the regular expression. Using the same example: To indicate that the operator is “=-,” use the following rule:

```
==/[A-Za-z]
```

to indicate that the operator is “=,” use the following rule:

```
=/[A-Za-z]
```

No backup is required in the rule action. It is not necessary to recognize the whole identifier to observe the ambiguity. However, the possibility of “=-3” makes

```
==/[^\t\n]
```

a still better rule.

7.2 input(), output(c) and unput(c)

In addition to these routines, lex also permits access to the I/O routines it uses. The following are provided as lex macro definitions:

1. **input()** This routine returns the next input character.

LEX

2. **output(c)** This routine writes the character "c" on the output.
3. **unput(c)** This routine pushes the character "c" back onto the input stream to be read later by **input()**.

These routines are provided by default, but you can override them by providing your own versions. However, since these routines define the relationship between external files and internal characters, they must all be retained and/or modified consistently.

These routines may be redefined to cause input or output to be transmitted to or from other programs or internal memory. The character set used **MUST** be consistent in all routines and a value of zero returned by **input()** **MUST** mean end-of-file.

The relationship between **unput** and **input** must be retained or the **lex** look-ahead will not work. The **lex** program does **NOT** look-ahead unless explicitly instructed to do so, either by rules ending in **+**, *****, **?** or **\$**, or by those containing a **/**. Look-ahead is necessary to match an expression that is a prefix of another expression. The standard **lex** library imposes a 100-character limit on backup.

7.3 yywrap()

Another **lex** library routine that you may sometimes want to redefine is **yywrap**. This routine is called whenever **lex** reaches an end-of-file.

If **yywrap** returns a 1, which it does by default, **lex** continues with the normal wrap-up on end of input.

It is sometimes convenient to arrange for input to continue from a new source. In this case, **yywrap** could be redefined to arrange for new input and returns 0. This would then instruct

lex to continue processing.

This routine provides a convenient way to print tables, summaries, etc. at the end of a program.

NOTE: It is NOT possible to write a normal rule that recognizes end-of-file. The only access to this condition is through **yywrap**. In fact, unless a private version of **input()** is supplied, a file containing NULLs cannot be handled since a value of 0 returned by **input** is taken to be end-of-file by **yywrap**.

7.4 REJECT

Note that **lex** is normally partitioning the input stream, NOT searching for all possible matches of each expression. This means that each character is accounted for once and only once. Consider the following example:

```

she    s++;
he     h++;
\n     |
      ;

```

The first rule matches all occurrences of the string "she" and the action increments "s" for each one found. The second matches all occurrences of the string "he" and its action increments "h" for each one found. The last two rules match new-line and everything else and takes the action of ignoring them. Normally, **lex** would not recognize the instances of "he" included in "she," since once it has passed a "she" those characters are gone. To override this default, the action **REJECT** could be used to instruct **lex** to "go do the next alternative." **REJECT** causes the rule AFTER the current rule to be executed. The position of the input pointer is adjusted accordingly.

Suppose you want to count the instances of "he" included in "she." To do that, use the following rules:

LEX

```
she    {s++; REJECT;}
he     {h++; REJECT;}
\n     |
.       ;
```

In this example, after counting each expression, it is “rejected” (whenever appropriate), and the other expression is evaluated. In this example, since “he” does not include “she” the REJECT action on “he” could be eliminated. In other cases, it is not possible to state which input characters are in both classes.

Consider the following two rules:

```
a[bc]+ { ... ; REJECT;}
a[cd]+ { ... ; REJECT;}
```

- If the input to the rules above were “ab,” only the first rule would match.
- If the input to these same rules were “ad,” only the second would match.
- If the the input were “accb,” the first rule would match four characters, and then the second rule would match three characters.
- If the input were “accd,” however, the second rule would match four characters, and then the first rule would match three characters.

In general, REJECT is useful whenever the purpose of lex is to detect all examples of some items in the input for which the instances of these items may overlap or include one another, instead of lex’s usual purpose of partitioning the input stream.

Suppose you want a digram of some input. Normally, the digrams overlap, that is, the word “the” is considered to contain both “th” and “he.” Assuming a 2-dimensional array

named `digram[]` to be incremented, an appropriate `lex` procedure would be:

```
%%
[a-z][a-z] {digram[yytext[0]][yytext[1]]++; REJECT;}
.\n      ;
```

In this example, `REJECT` is used to pick up a letter pair beginning at **EVERY** character, rather than at every other character.

The action `REJECT` does not rescan the input. Instead, it “remembers” the results of the previous scan. Therefore, if a program instructs `lex` to find a rule with trailing context and execute `REJECT`, `unput` **CANNOT** have been called to change the characters forthcoming from the input stream. This is the only restriction on the user’s ability to manipulate the not-yet-processed input.

LEX

8. Examples

For the sake of example, consider copying an input file while adding three to every positive number divisible by seven. A suitable **lex** source program follows:

```
%%
int k;
[0-9]+ {
    k = atoi(yytext);
    if (k%7 == 0)
        printf("%d", k+3);
    else
        printf("%d",k);
}
```

The rule “[0-9]+” recognizes strings of digits, 0 through 9; **atoi()** converts the digits to binary and stores the result in “k.” The operator % (remainder) is used to check whether “k” is divisible by seven; if it is, “k” is incremented by three as it is written out. It may be objected that this program alters such input items as “49.63” or “X7.” Furthermore, it increments the absolute value of all negative numbers divisible by seven. To avoid this, add a few more rules after the active one, for example:

```
%%
int k;
-?[0-9]+ {
    k = atoi(yytext);
    printf("%d", k%7 == 0 ? k+3 : k);
}
-?[0-9.]+ ECHO;
```

Numerical strings containing a dot (.), or preceded by a letter, will be picked up by one of the last two rules and not changed. The “if-else” has been replaced by a C language conditional expression to save space. The expression “a ? b : c” is evaluated as “if a then b else c.”

The following is an example using `lex` for statistics gathering. This program "histograms" the lengths of words. (A word is defined here as a string of letters).

```
int lengs[100];
%%
[a-z]+      lengs[yyval]++;
.           |
\n          ;
%%
yywrap( ) {
    int i;
    printf("Length No. words\n");
    for(i=0; i<100; i++)
        if (lengs[i] > 0)
            printf("%5d%10d\n",i,lengs[i]);
    return(1);
}
```

In the above example, the histogram is accumulated, but no output is generated until, at the end of the input, it prints the table. The final statement, ("return(1);"), indicates that `lex` is to perform wrap-up. If `yywrap` returns zero (false), it implies that further input is available and the program is to continue reading and processing. Remember, providing a `yywrap` that never returns true causes an infinite loop.

LEX

9. Summary

The general form of a lex source file is

```
{definitions}  
%%  
{rules}  
%%  
{user subroutines}
```

The “definitions” section contains a combination of the following:

1. Definitions in the form “name space translation.”
2. Included code in the form “space code.”
3. Included code in the form:

```
%{  
code  
%}
```

4. Start conditions given in the form:

```
%S name1 name2 ...
```

5. Character set tables in the form:

```
%T  
number space character-string  
...  
%T
```

6. Changes to internal array sizes in the form:

```
%x nnn
```

where “nnn” is a decimal integer representing an array size and “x” selects the parameter as follows:

LETTER	PARAMETER
p	positions
n	states
e	tree nodes
a	transitions
k	packed character classes
o	output array size

Lines in the "rules" section have the form:

expression action

where the "action" may be continued on succeeding lines by using braces to delimit it.

Regular expressions in `lex` use the following operators:

LEX

REGULAR EXPRESSION OPERATORS	
x	the character "x".
"x"	an "x", even if x is an operator.
\x	an "x", even if x is an operator.
[xy]	the character x or y.
[x-z]	the characters x, y, or z.
[^x]	any character but x.
.	any character but newline.
^x	an x at the beginning of a line.
<y>x	an x when lex is in start condition y.
x\$	an x at the end of a line.
x?	an optional x.
x*	0 or more instances of x.
x+	1 or more instances of x.
x y	an x OR a y.
(x)	an x.
x/y	an x but only if followed by y.
{xx}	expands to xx definition in lex definition section.
x{m,n}	m through n occurrences of x.

10. LEX and YACC

It is particularly easy to interface **lex** and **yacc**. The **lex** program recognizes only regular expressions; **yacc** writes parsers that accept a large class of context free grammars but requires a lower level analyzer to recognize input tokens. Thus, a combination of **lex** and **yacc** is often appropriate. When used as a preprocessor for a later parser generator, **lex** is used to partition the input stream; and the parser generator assigns structure to the resulting pieces. The flow of control in such a case is shown in Figure 3.2. Additional programs, written by other generators or by hand, can be added easily to programs written by **lex**. You will realize that the name **yylex** is what **yacc** expects its lexical analyzer to be named, so that the use of this name by **lex** simplifies interfacing.

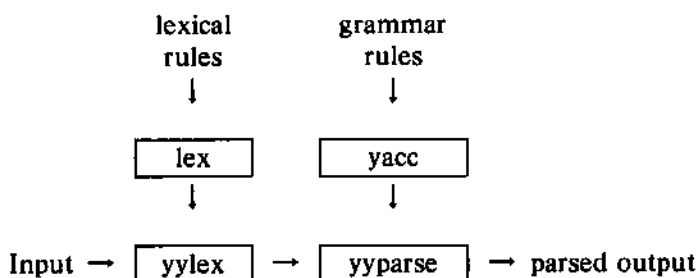


Figure 3.2. Lex With Yacc

To use **lex** with **yacc**, observe that **lex** writes a program named **yylex()** which is the name required by **yacc** for its analyzer. Normally, the default main program on the **lex** library calls the **yylex()** routine, but if **yacc** is loaded and its main program is used, **yacc** calls **yylex()**. In this case, each **lex** rule ends with

```
return(token);
```

where the appropriate token value is returned. An easy way to get access to **yacc**'s names for tokens is to compile the **lex** output file as part of the **yacc** output file by placing the line

LEX

```
# include "lex.yy.c"
```

in the last section of `yacc` input. If the grammar is to be named "good" and the lexical rules are to be named "better," the UNIX Operating System command sequence could be

```
yacc good  
lex better  
cc y.tab.c -ly -ll
```

The `yacc` library, (`-ly`), should be loaded before the `lex` library to obtain a main program that invokes the `yacc` parser. The generations of `lex` and `yacc` programs can be done in either order.

Chapter 4: YACC

CONTENTS

1. Introduction	1
2. Basic Specifications	6
3. Actions	11
4. Lexical Analysis	16
5. Parser Operation	19
6. Ambiguity and Conflicts	27
7. Precedence	35
8. Error Handling	40
9. The yacc Environment	44
10. Input Style	47
11. Left Recursion	48
12. Lexical Considerations	50
13. Reserved Words	52
14. Simulating Error and Accept in Actions	53
15. Accessing Values in Enclosing Rules	54
16. Arbitrary Value Types	56
17. Appendix 4.1	60
18. Appendix 4.2	65
19. Appendix 4.3	69
20. Appendix 4.4	80

LIST OF FIGURES

Figure 4.1. C Language Escapes Recognized by Yacc	8
--	---

Chapter 4

YACC –

YET ANOTHER COMPILER COMPILER

1. Introduction

The yacc program provides a general tool for imposing structure on the input to a computer program.

The first step in using the yacc program is to create a specification of the input process, which includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input.

The yacc program then generates a function to control the input process. This function, called a *parser*, calls the user-supplied low-level input routine (the *lexical analyzer*) to pick up the basic items (called *tokens*) from the input stream.

Tokens are organized according to the input structure rules, called *grammar rules*. When one of these rules has been recognized, then user code supplied for this rule (i.e., an *action*), is invoked.

Actions have the ability to return values and make use of the values of other actions.

The yacc program is written in a portable dialect of the C language, and the actions and output subroutine are in the C language as well. Moreover, many of the syntactic conventions of yacc follow the C language.

The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a

YACC

name. For example, one grammar rule might be

```
date : month_name day ',' year ;
```

where **date**, **month_name**, **day** and **year** represent structures of interest in the input process; presumably, **month_name**, **day** and **year** are defined elsewhere.

The comma (',') is enclosed in single quotes. This implies that the comma is to appear literally in the input.

The colon and semicolon merely serve as punctuation in the rule and have no significance in controlling the input.

With proper definitions, the following input might be matched by the rule given above:

```
July 4, 1776
```

An important part of the input process is carried out by the lexical analyzer. This user routine reads the input stream, recognizes the lower-level structures, and communicates these tokens to the parser.

For historical reasons, a structure recognized by the lexical analyzer is called a **terminal symbol**, while the structure recognized by the parser is called a **nonterminal symbol**. To avoid confusion, terminal symbols will usually be referred to as **tokens**.

There is considerable leeway in deciding whether to recognize structures using the lexical analyzer or grammar rules. For example, the following rules might be used in the above example:

```

month_name : 'J' 'a' 'n' ;
month_name : 'F' 'e' 'b' ;

```

...

```

month_name : 'D' 'e' 'c' ;

```

The lexical analyzer only needs to recognize individual letters, and **month name** is a nonterminal symbol.

Such low-level rules tend to waste time and space and may complicate the specification beyond the ability of yacc to deal with it.

Usually, the lexical analyzer recognizes the month names and returns an indication that a **month name** is seen. In this case, **month name** is a token.

Literal characters such as a comma must also be passed through the lexical analyzer and are also considered tokens.

Specification files are very flexible. If the following rule were added to the above example, entering 7/4/1776 would then be equivalent to July 4, 1776 on input:

```

date : month '/' day '/' year ;

```

In most cases, this new rule could be "slipped in" to a working system with minimal effort and little danger of disrupting existing input.

The input being read may not conform to the specifications. These input errors are detected as early as is theoretically possible with a left-to-right scan. Thus, not only is the chance of reading and computing with bad input data substantially reduced, but the bad data can usually be quickly found.

YACC

Error handling, provided as part of the input specifications, permits the reentry of bad data or the continuation of the input process after skipping over the bad data.

In some cases, yacc fails to produce a parser when given a set of specifications. For example, the specifications may be self-contradictory, or they may require a more powerful recognition mechanism than that available to yacc. The former cases represent design errors; the latter cases can often be corrected by making the lexical analyzer more powerful or by rewriting some of the grammar rules.

While yacc cannot handle all possible specifications, its power compares favorably with similar systems. Moreover, the constructions which are difficult for yacc to handle are also frequently difficult for human beings to handle.

Some users have reported that the discipline of formulating valid yacc specifications for their input revealed errors of conception or design early in the program development.

The yacc program has been extensively used in numerous practical applications, including lint, the Portable C Compiler, and a system for typesetting mathematics.

The remainder of this document describes the following subjects as they relate to yacc

- Basic process of preparing a yacc specification
- Parser operation
- Handling ambiguities
- Handling operator precedences in arithmetic expressions
- Error detection and recovery
- The operating environment and special features of the parsers yacc produces
- Suggestions to improve the style and efficiency of

- the specifications
- Advanced topics

In addition, there are four appendices:

- Appendix 4.1** This appendix contains a brief example.
- Appendix 4.2** This appendix contains a summary of the yacc input syntax.
- Appendix 4.3** This appendix contains an example using some of the more advanced features of yacc, and
- Appendix 4.4** This appendix contains a description of the mechanisms and syntax which, though no longer actively supported, are provided for historical continuity with older versions of yacc.

YACC

2. Basic Specifications

Names refer to either tokens or nonterminal symbols.

The yacc program requires token names to be declared as such. In addition, it is often desirable to include the lexical analyzer as part of the specification file.

It may be useful to include other programs as well.

Every specification file consists of three sections:

- The **declarations**
- The **grammar rules**
- The **programs**

These sections are separated by double percent (%%) marks. (The percent symbol is generally used in yacc specifications as an escape character.)

The following is a syntactic description of a yacc specification file:

```
declarations
%%
rules
%%
programs
```

The declaration section may be empty, and, if the programs section is omitted, the second %% mark may also be omitted.

The smallest legal yacc specification is therefore:

```
%%
rules
```

since the other two sections may be omitted.

Blanks, tabs, and newlines are ignored, but they may not appear in names or multicharacter reserved symbols.

Comments may appear wherever a name is legal. They are enclosed in `/* ... */`, as in C language.

The rules section is made up of one or more grammar rules. A grammar rule has the following form:

A : BODY ;

In this example, **A** represents a nonterminal name, and **BODY** represents a sequence of zero or more names and literals.

The colon and the semicolon are yacc punctuation.

Names may be of arbitrary length and may be made up of letters, dots, underscores, and noninitial digits.

Uppercase and lowercase letters are distinct. The names used in the body of a grammar rule may represent tokens or nonterminal symbols.

A literal consists of a character enclosed in single quotes (`'`).

As in C language, the backslash (`\`) is an escape character within literals, and all the C language escapes are recognized.

C LANGUAGE ESCAPES RECOGNIZED BY YACC	
<code>'\n'</code>	newline
<code>'\r'</code>	return
<code>'\"'</code>	single quote (')
<code>'\\'</code>	backslash (\)
<code>'\t'</code>	tab
<code>'\b'</code>	backspace
<code>'\f'</code>	form feed
<code>'\xxx'</code>	xxx in octal

Figure 4.1. C Language Escapes Recognized by Yacc

For a number of technical reasons, the NULL character (`'\0'` or `0`) should never be used in grammar rules.

If there are several grammar rules with the same left-hand side, the vertical bar `|` can be used to avoid rewriting the left-hand side.

The semicolon at the end of a rule can be dropped before a vertical bar. Thus the following grammar rules:

```
A : B C D ;
A : E F ;
A : G ;
```

can be given to yacc using the vertical bar:

```
A : B C D
   | E F
   | G
   ;
```

It is not necessary that all grammar rules with the same left side appear together in the grammar rules section although it makes

the input much more readable and easier to change.

If a nonterminal symbol matches the empty string, this can be indicated by the following:

```
empty : ;
```

Names representing tokens must be declared in the declarations section. For example:

```
%token  name1  name2 ...
```

Every name not defined in the declarations section is assumed to represent a nonterminal symbol.

Every nonterminal symbol must appear on the left side of at least one rule.

Of all the nonterminal symbols, the **start symbol** has particular importance.

The parser is designed to recognize the start symbol. Thus, this symbol represents the largest, most general structure described by the grammar rules.

By default, the start symbol is taken to be the left-hand side of the first grammar rule in the rules section.

It is possible and desirable to declare the start symbol explicitly in the declarations section using the **%start** keyword. For example:

```
%start  symbol
```

The end of the input to the parser is signaled by a special token, called the **end-marker**.

YACC

If the tokens up to but not including the end-marker form a structure that matches the start symbol, the parser function returns to its caller after the end-marker is seen and accepts the input.

If the end-marker is seen in any other context, it is an error.

It is the job of the user-supplied lexical analyzer to return the end-marker when appropriate.

Usually the end-marker represents some reasonably obvious I/O status, such as "end-of-file" or "end-of-record".

3. Actions

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process.

These actions may return values and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens if desired.

An action is an arbitrary C language statement and as such can do input and output, call subprograms, and alter external vectors and variables.

An action is specified by one or more statements enclosed in curly braces ({} and {}). For example:

```
A      : '(' B ')'
      {
        hello( 1, "abc" );
      }
```

and the following is an example of grammar rules with actions:

```
XXX    :   YYY   ZZZ
      {
        printf("a message\n");
        flag = 25;
      }
```

To facilitate easy communication between the actions and the parser, the action statements are altered slightly.

The dollar sign symbol (\$) is used as a signal to yacc in this context.

YACC

To return a value, the action normally sets the pseudo-variable **\$\$** to some value.

The following action does nothing but return the value of one:

```
{ $$ = 1; }
```

To obtain the values returned by previous actions and the lexical analyzer, the action may use the pseudo-variables **\$1**, **\$2**, ..., which refer to the values returned by the components of the right side of a rule, reading from left to right.

For example, if the rule is:

```
A : B C D ;
```

then **\$2** has the value returned by C, and **\$3** the value returned by D.

With the following rule, the value returned by this rule is usually the value of the **expr** in parentheses:

```
expr : '(' expr ')'
{
    $$ = $2 ;
}
```

By default, the value of a rule is the value of the first element in it (**\$1**).

Grammar rules of the following form frequently need not have an explicit action:

```
A : B ;
```

In the examples above, all the actions came at the end of rules. Sometimes, it is desirable to get control before a rule is fully

parsed. The yacc permits an action to be written in the middle of a rule as well as at the end.

This rule is assumed to return a value accessible through the usual \$ mechanism by the actions to the right of it.

In turn, it may access the values returned by the symbols to its left. For example, in the following rule x is set to 1 and y is set to the value returned by C:

```

A      : B
      {
        $$ = 1;
      }
      C
      {
        x = $2;
        y = $3;
      }
      ;

```

Actions that do not terminate a rule are actually handled by yacc by manufacturing a new nonterminal symbol name and a new rule matching this name to the empty string.

The interior action is the action triggered off by recognizing this added rule.

The yacc program actually treats the above example as if it had been written like the following: (\$ACT is an empty action.)

YACC

```
$ACT : /* empty */
{
    $$ = 1;
}
;

A : B $ACT C
{
    x = $2;
    y = $3;
}
;
```

In many applications, output is not done directly by the actions.

A data structure, such as a parse tree, is constructed in memory and transformations are applied to it before output is generated.

Parse trees are particularly easy to construct given routines to build and maintain the tree structure desired.

In the following example, the C function **node** creates a node with label **L** and descendants **n1** and **n2** and returns the index of the newly created node:

```
node( L, n1, n2 )
```

Then a parse tree is built by supplying the actions following in the yacc specification file:

```
expr : expr '+' expr
{
    $$ = node( '+', $1, $3 );
}
```

The user may define other variables to be used by the actions.

Declarations and definitions can appear in the declarations section enclosed in the marks `%{` and `%}`. These declarations and definitions have global scope, so they are known to the action statements and the lexical analyzer. For example:

```
%{    int variable = 0;    %}
```

could be placed in the declarations section making **variable** accessible to all of the actions.

The **yacc** parser uses only names beginning with **yy**. The user should avoid such names.

In these examples, all the values are integers. A discussion of values of other types is found in the part **Advanced Topics**.

4. Lexical Analysis

The user must supply a lexical analyzer to read the input stream and communicate tokens (with values, if desired) to the parser.

The lexical analyzer is an integer-valued function called `yylex`. The function returns an integer, the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable `yylval`.

The parser and the lexical analyzer must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc` or the user. In either case, the `#define` mechanism of C language is used to allow the lexical analyzer to return these numbers symbolically. For example, suppose that the token name `DIGIT` has been defined in the declarations section of the `yacc` specification file. The relevant portion of the lexical analyzer might look like the following:

```

yylex0
{
    extern int yylval;
    int c;
    ...
    c = getchar();
    ...
    switch( c )
    {
        ...
        case '0':
        case '1':
        ...
        case '9':
            yylval = c-'0';
            return( DIGIT );
        ...
    }
    ...
}

```

The intent is to return a token number of DIGIT and a value equal to the numerical value of the digit.

Provided that the lexical analyzer code is placed in the programs section of the specification file, the identifier DIGIT is defined as the token number associated with the token DIGIT.

This mechanism leads to clear, easily modified lexical analyzers. The only pitfall to avoid is using any token names in the grammar that are reserved or significant in C language or the parser. For example, the use of token names `if` or `while` will almost certainly cause severe difficulties when the lexical analyzer is compiled.

The token name `error` is reserved for error handling and should not be used naively.

YACC

As mentioned above, the token numbers may be chosen by `yacc` or the user. In the default situation, the numbers are chosen by `yacc`. The default token number for a literal character is the numerical value of the character in the local character set.

Other names are assigned token numbers starting at 257.

To assign a token number to a token (including literals), the first appearance of the token name or literal in the declarations section can be immediately followed by a nonnegative integer. This integer is taken to be the token number of the name or literal. Names and literals not defined by this mechanism retain their default definition.

It is important that all token numbers be distinct.

For historical reasons, the end-marker must have token number 0 or negative. This token number cannot be redefined by the user. Thus, all lexical analyzers should be prepared to return 0 or a negative number as a token upon reaching the end of their input.

A very useful tool for constructing lexical analyzers is the `lex` program. These lexical analyzers are designed to work in close harmony with `yacc` parsers. The specifications for these lexical analyzers use regular expressions instead of grammar rules.

`Lex` can be easily used to produce quite complicated lexical analyzers, but there remain some languages (such as `FORTRAN`) which do not fit any theoretical framework and whose lexical analyzers must be crafted by hand.

5. Parser Operation

The **yacc** program turns the specification file into a C language program, which parses the input according to the specification given.

The algorithm used to go from the specification to the parser is complex and will not be discussed here. The parser itself, however, is relatively simple and understanding how it works will make treatment of error recovery and ambiguities much more comprehensible.

The parser produced by **yacc** consists of a finite state machine with a stack.

The parser is also capable of reading and remembering the next input token (called the **look-ahead** token).

The **current state** is always the one on the top of the stack. The states of the finite state machine are given small integer labels.

Initially, the machine is in state 0 (the stack contains only state 0) and no look-ahead token has been read.

The machine has only four actions available—**shift**, **reduce**, **accept** and **error**.

A step of the parser is done as follows:

1. Based on its current state, the parser decides if it needs a look-ahead token to choose the action to be taken. If it needs one and does not have one, it calls **yylex** to obtain the next token.

YACC

2. Using the current state and the look-ahead token if needed, the parser decides on its next action and carries it out. This may result in states being pushed onto the stack or popped off of the stack and in the look-ahead token being processed or left alone.

The **shift** action is the most common action the parser takes.

Whenever a shift action is taken, there is always a look-ahead token. In the following example, in state 56, if the look-ahead token is IF, the current state (56) is pushed down on the stack, and state 34 becomes the current state (on the top of the stack):

IF shift 34

The look-ahead token is cleared.

The **reduce** action keeps the stack from growing without bounds. Reduce actions are appropriate when the parser has seen the right-hand side of a grammar rule and is prepared to announce that it has seen an instance of the rule replacing the right-hand side by the left-hand side.

It may be necessary to consult the look-ahead token to decide whether to reduce or not (usually it is not necessary). In fact, the default action (represented by a dot) is often a reduce action.

Reduce actions are associated with individual grammar rules. Grammar rules are also given small integer numbers, and this leads to some confusion. For example, in the following display, the action refers to grammar rule 18:

. reduce 18

While in the following example, the action refers to state 34:

IF shift 34

Suppose the following rule is being reduced:

A : x y z ;

The reduce action depends on the left-hand symbol (A in this case) and the number of symbols on the right-hand side (three in this case).

To reduce, first pop off the top three states from the stack. (In general, the number of states popped equals the number of symbols on the right side of the rule.) In effect, these states were the ones put on the stack while recognizing x, y, and z and no longer serve any useful purpose.

After popping these states, a state is uncovered which was the state the parser was in before beginning to process the rule.

Using this uncovered state and the symbol on the left side of the rule, perform what is in effect a shift of A. A new state is obtained, pushed onto the stack, and parsing continues.

There are significant differences between the processing of the left-hand symbol and an ordinary shift of a token, however, so this action is called a **goto** action. In particular, the look-ahead token is cleared by a shift but is not affected by a goto. In any case, the uncovered state contains an entry such as the following which causes state 20 to be pushed onto the stack and become the current state:

A goto 20

In effect, the reduce action "turns back the clock" in the parse popping the states off the stack to go back to the state where

YACC

the right-hand side of the rule was first seen.

The parser then behaves as if it had seen the left side at that time. If the right-hand side of the rule is empty, no states are popped off of the stacks.

The uncovered state is in fact the current state.

The reduce action is also important in the treatment of user-supplied actions and values. When a rule is reduced, the code supplied with the rule is executed before the stack is adjusted. In addition to the stack holding the states, another stack running in parallel with it holds the values returned from the lexical analyzer and the actions.

When a shift takes place, the external variable `yyval` is copied onto the value stack.

After the return from the user code, the reduction is carried out.

When the `goto` action is done, the external variable `yyval` is copied onto the value stack.

The pseudo-variables `$1`, `$2`, etc., refer to the value stack.

The other two parser actions are conceptually much simpler.

The `accept` action indicates that the entire input has been seen and that it matches the specification. This action appears only when the look-ahead token is the end-marker and indicates that the parser has successfully done its job.

The **error** action, on the other hand, represents a place where the parser can no longer continue parsing according to the specification. The input tokens it has seen (together with the look-ahead token) cannot be followed by anything that would result in a legal input.

The parser reports an error and attempts to recover the situation and resume parsing. The error recovery (as opposed to the detection of error) will be discussed later.

Consider the following example as a yacc specification:

```
%token DING DONG DELL
%%
rhyme : sound place
      ;
sound : DING DONG
      ;
place : DELL
      ;
```

When yacc is invoked with the **-v** option, a file called **y.output** is produced with a human-readable description of the parser.

The following example is the **y.output** file corresponding to the above grammar (with some statistics stripped off the end) where the actions for each state are specified and there is a description of the parsing rules being processed in each state:

YACC

```
state 0
    $accept : _rhyme $end
    DING shift 3
    . error
    rhyme goto 1
    sound goto 2

state 1
    $accept : rhyme $end
    $end accept
    . error

state 2
    rhyme : sound_place
    DELL shift 5
    . error
    place goto 4

state 3
    sound : DING_DONG
    DONG shift 6
    . error

state 4
    rhyme : sound place_ (1)
    . reduce 1

state 5
    place : DELL_ (3)
    . reduce 3

state 6
    sound : DING DONG_ (2)
    . reduce 2
```

The `_` character is used to indicate what has been seen and what is yet to come in each rule.

The following input can be used to track the operations of the parser:

DING DONG DELL

Initially, the current state is state 0.

The parser needs to refer to the input in order to decide between the actions available in state 0, so the first token (**DING**) is read and becomes the look-ahead token.

The action in state 0 on **DING** is **shift 3**. State 3 is pushed onto the stack, and the look-ahead token is cleared.

State 3 becomes the current state.

The next token (**DONG**) is read and becomes the look-ahead token.

The action in state 3 on the token **DONG** is **shift 6**. State 6 is pushed onto the stack, and the look-ahead is cleared.

The stack now contains 0, 3, and 6. In state 6, without even consulting the look-ahead, the parser reduces by the following, which is rule 2:

sound : DING DONG

Two states, 6 and 3, are popped off of the stack uncovering state 0.

Consulting the description of state 0 (looking for a goto on **sound**), the following is obtained:

sound goto 2

State 2 is pushed onto the stack and becomes the current state.

In state 2, the next token (**DELL**) must be read.

YACC

The action is **shift 5**, so state 5 is pushed onto the stack, which now has 0, 2, and 5 on it, and the look-ahead token is cleared.

In state 5, the only action is to reduce by rule 3. This has one symbol on the right-hand side, so one state, 5, is popped off, and state 2 is uncovered.

The goto in state 2 on **place** (the left side of rule 3) is state 4.

Now, the stack contains 0, 2, and 4.

In state 4, the only action is to reduce by rule 1. There are two symbols on the right, so the top two states are popped off, uncovering state 0 again.

In state 0, there is a goto on **rhyme** causing the parser to enter state 1.

In state 1, the input is read and the end-marker is obtained indicated by **\$end** in the **y.output** file.

The action in state 1 (when the end-marker is seen) successfully ends the parse.

The reader is urged to consider how the parser works when confronted with such incorrect strings as **DING DONG DONG, DING DONG, DING DONG DELL DELL**, etc. A few minutes spent with this and other simple examples is repaid when problems arise in more complicated contexts.

6. Ambiguity and Conflicts

A set of grammar rules is ambiguous if there is some input string that can be structured in two or more different ways.

For example, the following grammar rule is a natural way of expressing the fact that one way of forming an arithmetic expression is to put two other expressions together with a minus sign between them:

```
expr : expr '-' expr
```

Unfortunately, this grammar rule does not completely specify the way that all complex inputs should be structured. For example, if the input is

```
expr - expr - expr
```

the rule allows this input to be structured as either

```
( expr - expr ) - expr
```

or as

```
expr - ( expr - expr )
```

(The first is called **left association**, the second **right association**.)

The **yacc** program detects such ambiguities when it is attempting to build the parser.

Consider the problem that confronts the parser when provided with the following input:

```
expr - expr - expr
```

YACC

When the parser has read the second **expr**, the input seen matches the right side of the grammar rule above:

expr - expr

The parser could reduce the input by applying this rule. After applying the rule, the input is reduced to **expr** (the left side of the rule).

The parser would then read the final part of the input (displayed in the following example) and again reduce:

- expr

The effect of this is to take the left associative interpretation.

Alternatively, if the parser sees the following:

expr - expr

it could defer the immediate application of the rule and continue reading the input until it sees the following:

expr - expr - expr

It could then apply the rule to the rightmost three symbols reducing them to **expr** which results in the following being left:

expr - expr

Now the rule can be reduced once more. The effect is to take the right associative interpretation. The parser can do one of two legal things, a shift or a reduction. It has no way of deciding between them. This is called a **shift/reduce conflict**.

It may also happen that the parser has a choice of two legal reductions. This is called a **reduce/reduce conflict**.

Note that there are never any **shift/shift** conflicts.

When there are shift/reduce or reduce/reduce conflicts, **yacc** still produces a parser. It does this by selecting one of the valid steps wherever it has a choice.

A rule describing the choice to make in a given situation is called a **disambiguating rule**.

The **yacc** program invokes two disambiguating rules by default:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the *earlier* grammar rule (in the input sequence).

Rule 1 implies that reductions are deferred in favor of shifts when there is a choice.

Rule 2 gives the user rather crude control over the behavior of the parser in this situation, but reduce/reduce conflicts should be avoided when possible.

Conflicts may arise because of mistakes in input or logic or because the grammar rules (while consistent) require a more complex parser than **yacc** can construct.

The use of actions within rules can also cause conflicts if the action must be done before the parser can be sure which rule is being recognized. In these cases, the application of disambiguating rules is inappropriate and leads to an incorrect parser.

For this reason, **yacc** always reports the number of shift/reduce and reduce/reduce conflicts resolved by Rule 1 and Rule 2.

YACC

In general, whenever it is possible to apply disambiguating rules to produce a correct parser, it is also possible to rewrite the grammar rules so that the same inputs are read but there are no conflicts.

For this reason, most previous parser generators have considered conflicts to be fatal errors.

Our experience has suggested that this rewriting is somewhat unnatural and produces slower parsers. Thus, yacc will produce parsers even in the presence of conflicts.

As an example of the power of disambiguating rules, consider

```
stat  :  IF '(' cond ')' stat
      |  IF '(' cond ')' stat ELSE stat
      ;
```

which is a fragment from a programming language involving an "if-then-else" statement.

In these rules, IF and ELSE are tokens, cond is a nonterminal symbol describing conditional (logical) expressions, and stat is a nonterminal symbol describing statements.

The first rule will be called the "simple-if" rule and the second the "if-else" rule.

These two rules form an ambiguous construction since input of the following form can be structured according to these rules in two ways:

```
IF ( C1 ) IF ( C2 ) S1 ELSE S2
```

The input can be structured either as in the following example, or in the subsequent example which is the one given in most

programming languages having this construct:

```

IF ( C1 )
{
    IF ( C2 )
        S1
}
ELSE
    S2

```

or:

```

IF ( C1 )
{
    IF ( C2 )
        S1
    ELSE
        S2
}

```

Each **ELSE** is associated with the last preceding "un-ELSE'd" **IF**.

In the following example, consider the situation where the parser has seen the IF-ELSE construct and is looking at the ELSE.

```

IF ( C1 ) IF ( C2 ) S1

```

It can immediately reduce by the simple-if rule to get

```

IF ( C1 ) stat

```

and then read the remaining input

```

ELSE S2

```

and reduce by the if-else rule.

YACC

This leads to the first of the above groupings of the input.

On the other hand, the "ELSE" may be shifted, "S2" read, and then the right-hand portion can be reduced by the if-else rule to get the following which can be reduced by the simple-if rule:

```
IF ( C1 ) stat
```

This leads to the second of the above groupings of the input which is usually desired.

Once again, the parser can do two valid things—there is a shift/reduce conflict. The application of disambiguating Rule 1 tells the parser to shift in this case, which leads to the desired grouping.

This shift/reduce conflict arises only when there is a particular current input symbol, ELSE, and particular inputs, such as have already been seen:

```
IF ( C1 ) IF ( C2 ) S1
```

In general, there may be many conflicts, and each one will be associated with an input symbol and a set of previously read inputs.

The previously read inputs are characterized by the state of the parser.

The conflict messages of yacc are best understood by examining the verbose (-v) option output file. For example, the output corresponding to the above conflict state might be

**23: shift/reduce conflict (shift 45, reduce 18) on ELSE
state 23**

```

stat : IF ( cond ) stat_      (18)
stat : IF ( cond ) stat_ELSE stat
ELSE      shift 45
          reduce 18

```

where the first line describes the conflict—giving the state and the input symbol.

The ordinary state description gives the grammar rules active in the state and the parser actions.

Recall that the underline marks the portion of the grammar rules which has been seen. Thus in the example, in state 23 the parser has seen input corresponding to **IF (cond) stat** and the two grammar rules shown, are active at this time.

The parser can do two possible things:

- If the input symbol is **ELSE**, it is possible to shift into state 45. State 45 will have, as part of its description, the following line:

```
stat : IF ( cond ) stat ELSE_stat
```

since the **ELSE** will have been shifted in this state. In state 23, the alternative action [describing a dot (.)] is to be done if the input symbol is not mentioned explicitly in the actions.

- If the input symbol is not **ELSE**, the parser reduces to

```
stat : IF '(' cond ')' stat
```

by grammar rule 18.

YACC

Once again, notice that the numbers following **shift** commands refer to other "states," while the numbers following **reduce** commands refer to "grammar rule numbers."

In the **y.output** file, the rule numbers are printed after those rules which can be reduced.

In most one states, there is a **reduce** action possible as the default command.

The user who encounters unexpected shift/reduce conflicts will probably want to look at the verbose output to decide whether the default actions are appropriate.

7. Precedence

There is one common situation where the rules given above for resolving conflicts are not sufficient. This is in the parsing of arithmetic expressions.

Most of the commonly used constructions for arithmetic expressions can be naturally described by the notion of precedence levels for operators, together with information about left or right associativity.

It turns out that ambiguous grammars with appropriate disambiguating rules can be used to create parsers that are faster and easier to write than parsers constructed from unambiguous grammars.

The basic notion is to write grammar rules of the following two forms for all binary and unary operators desired:

expr : expr OP expr

and

expr : UNARY expr

This creates a very ambiguous grammar with many parsing conflicts.

As disambiguating rules, the user specifies the precedence or binding strength of all the operators and the associativity of the binary operators.

This information is sufficient to allow yacc to resolve the parsing conflicts in accordance with these rules and construct a parser that realizes the desired precedences and associativities.

YACC

The precedences and associativities are attached to tokens in the declarations section. This is done by a series of lines beginning with one of the following yacc keywords: **%left**, **%right** or **%nonassoc**, followed by a list of tokens.

All of the tokens on the same line are assumed to have the same precedence level and associativity; the lines are listed in order of increasing precedence or binding strength. For example:

```
%left '+' '-'  
%left '*' '/'
```

describes the precedence and associativity of the four arithmetic operators.

Plus and minus are left associative and have lower precedence than star and slash, which are also left associative.

The keyword **%right** is used to describe right associative operators, and the keyword **%nonassoc** is used to describe operators, like the operator **.LT.** in FORTRAN, that may not associate with themselves. For example, the following is illegal in FORTRAN and such an operator would be described with the keyword **%nonassoc** in yacc:

```
A .LT. B .LT. C
```

As an example of the behavior of these declarations, the following description might be used to structure the subsequent input:

```

%right '='
%left '+' '-'
%left '*' '/'

%%

expr : expr '=' expr
     | expr '+' expr
     | expr '-' expr
     | expr '*' expr
     | expr '/' expr
     | NAME
     ;

```

The following is the input to be structured by the above description in order to perform the correct precedence of operators:

a = b = c*d - e - f*g

The result of the structuring is as follows:

a = (b = ((c*d) - e) - (f*g)))

When this mechanism is used, unary operators **MUST**, in general, be given a precedence.

Sometimes a unary operator and a binary operator have the same symbolic representation but different precedences.

An example is unary and binary minus (**-**). Unary minus may be given the same strength as multiplication, or even higher, while binary minus has a lower strength than multiplication.

The keyword, **%prec**, changes the precedence level associated with a particular grammar rule.

YACC

The keyword **%prec** appears immediately after the body of the grammar rule, before the action or closing semicolon, and is followed by a token name or literal.

The keyword **%prec** causes the precedence of the grammar rule to become that of the following token name or literal. For example, the following rules might be used to give unary minus the same precedence as multiplication:

```
%left '+' '-'
%left '*' '/'

%%

expr : expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | '-' expr %prec '*'
    | NAME
    ;
```

A token declared by **%left**, **%right**, and **%nonassoc** need not be, but may be, declared by **%token** as well.

The precedences and associativities are used by **yacc** to resolve parsing conflicts. They give rise to disambiguating rules. Formally, the rules work as follows:

1. The precedences and associativities are recorded for those tokens and literals that have them.
2. A precedence and associativity is associated with each grammar rule. It is the precedence and associativity of the last token or literal in the body of the rule. If the **%prec** construction is used, it overrides this default. Some grammar rules may have no precedence and associativity associated with them.

3. When there is a reduce/reduce conflict or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two disambiguating rules given at the beginning of the section are used, and the conflicts are reported.
4. If there is a shift/reduce conflict and both the grammar rule and the input character have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociating implies error.

Conflicts resolved by precedence are not counted in the number of shift/reduce and reduce/reduce conflicts reported by yacc. This means that mistakes in the specification of precedences may disguise errors in the input grammar. It is a good idea to be sparing with precedences and use them in an essentially "cookbook" fashion until some experience has been gained.

The `y.output` file is very useful in deciding whether the parser is actually doing what was intended.

8. Error Handling

Error handling is an extremely difficult area, and many of the problems are semantic ones. When an error is found, for example, it may be necessary to reclaim parse tree storage, delete or alter symbol table entries, and, typically, set switches to avoid generating any further output.

It is seldom acceptable to stop all processing when an error is found. It is more useful to continue scanning the input to find further syntax errors. This leads to the problem of getting the parser "restarted" after an error.

A general class of algorithms to do this involves discarding a number of tokens from the input string and attempting to adjust the parser so that input can continue.

To allow the user some control over this process, yacc provides a simple, but reasonably general feature. The token name "error" is reserved for error handling. This name can be used in grammar rules. In effect, it suggests places where errors are expected and recovery might take place.

The parser pops its stack until it enters a state where the token "error" is legal. It then behaves as if the token "error" were the current look-ahead token and performs the action encountered.

The look-ahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a cascade of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted.

If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

As an example, a rule of the following form means that on a syntax error the parser attempts to skip over the statement in which the error is seen:

```
stat : error
```

More precisely, the parser scans ahead, looking for three tokens that might legally follow a statement, and start processing at the first of these. If the beginnings of statements are not sufficiently distinctive, it may make a false start in the middle of a statement and end up reporting a second error where there is in fact no error.

Actions may be used with these special error rules. These actions might attempt to reinitialize tables, reclaim symbol table space, etc.

Error rules such as the above are very general but difficult to control. Rules such as the following are somewhat easier. Here, when there is an error, the parser attempts to skip over the statement but does so by skipping to the next semicolon:

```
stat : error ';' ;
```

All tokens after the error and before the next semicolon cannot be shifted and are discarded. When the semicolon is seen, this rule will be reduced and any "cleanup" action associated with it performed.

Another form of error rule arises in interactive applications where it may be desirable to permit a line to be reentered after an error. The following example is one way to do this:

YACC

```
input : error '\n'
      {
          printf( "Reenter last line: " );
      }
      input
      {
          $$ = $4;
      }
      ;
```

There is one potential difficulty with this approach. The parser must correctly process three input tokens before it admits that it has correctly resynchronized after the error. If the reentered line contains an error in the first two tokens, the parser deletes the offending tokens and gives no message. This is clearly unacceptable.

For this reason, there is a mechanism that can force the parser to believe that error recovery has been accomplished. The following statement in an action resets the parser to its normal mode:

```
yyerrok ;
```

The last example can be rewritten, somewhat more usefully, as the following:

```
input : error '\n'
      {
          yyerrok;
          printf( "Reenter last line: " );
      }
      input
      {
          $$ = $4;
      }
      ;
```

As previously mentioned, the token seen immediately after the "error" symbol is the input token at which the error was discovered. Sometimes, this is inappropriate. For example, an error recovery action might take upon itself the job of finding the correct place to resume input. In this case, the previous look-ahead token must be cleared. The following statement in an action will have this effect:

```
yyclearin ;
```

For example, suppose the action after error were to call some sophisticated resynchronization routine (supplied by the user) that attempted to advance the input to the beginning of the next valid statement. After this routine is called, the next token returned by **yylex** is presumably the first token in a legal statement. The old illegal token must be discarded and the error state reset. A rule similar to the one following could perform this:

```
stat      : error  
{  
    resynch();  
    yyerrok ;  
    yyclearin;  
}  
;
```

These mechanisms are admittedly crude but do allow for a simple, fairly effective recovery of the parser from many errors. Also, the user can get control to deal with the error actions required by other portions of the program.

9. The yacc Environment

When the user inputs a specification to **yacc**, the output is a file of C language programs, called **y.tab.c** on most systems. (Due to local file system conventions, the names may differ from installation to installation.)

The function produced by **yacc** is an integer valued function called **yyparse()**. When it is called, it in turn repeatedly calls **yylex()**, the lexical analyzer supplied by the user (see **Lexical Analysis**), to obtain input tokens.

Eventually, an error is detected, **yyparse()** returns the value 1, and no error recovery is possible, or the lexical analyzer returns the end-marker token and the parser accepts. In this case, **yyparse()** returns the value 0.

The user must provide a certain amount of environment for this parser in order to obtain a working program. For example, as with every C language program, a program called **main()** must be defined that eventually calls **yyparse()**.

A routine called **yyerror()** prints a message when a syntax error is detected.

These two routines (**main()** and **yyerror()**) must be supplied in one form or another by the user.

To ease the initial effort of using **yacc**, a library has been provided with default versions of **main()** and **yyerror()**. The name of this library is system dependent. On many systems, the library is accessed by a **-ly** argument to the loader.

The following source code examples show the triviality of these routines:

```

main()
{
    return ( yyparse() );
}

```

and

```

# include <stdio.h>

yyerror(s)
char *s;
{
    fprintf( stderr, "%s\n", s );
}

```

The argument to `yyerror()` is a string containing an error message, usually the string "syntax error."

The average application wants to do better than this. Ordinarily, the program should keep track of the input line number and print it along with the message when a syntax error is detected.

The external integer variable `yychar` contains the look-ahead token number at the time the error was detected. This may be of some interest in giving better diagnostics.

Since the `main()` program is probably supplied by the user (to read arguments, etc.), the `yacc` library is useful only in small projects or in the earliest stages of larger ones.

The external integer variable `yydebug` is normally set to 0. If it is set to a nonzero value, the parser will output a verbose description of its actions including a discussion of the input symbols read and what the parser actions are.

YACC

Depending on the operating environment, it may be possible to set **yydebug** by using a debugging system.

10. Input Style

It is difficult to provide rules with substantial actions and still have a readable specification file. The following are a few style hints:

1. Use all uppercase letters for token names and all lowercase letters for nonterminal names.
2. Put grammar rules and actions on separate lines. This allows either to be changed without an automatic need to change the other.
3. Put all rules with the same left-hand side together. Put the left-hand side in only once and let all following rules begin with a vertical bar.
4. Put a semicolon only after the last rule with a given left-hand side and put the semicolon on a separate line. This allows new rules to be easily added.
5. Indent rule bodies by two tab stops and action bodies by three tab stops.

The example in Appendix 4.1 is written following this style, as are the examples in this section (where space permits).

The user must make up his own mind about these stylistic questions.

The central problem, however, is to make the rules visible through the morass of action code.

11. Left Recursion

The algorithm used by the yacc parser encourages so called "left recursive" grammar rules. Rules of the following form match this algorithm:

```
name      :  name rest_of_rule  ;
```

Rules such as the two following frequently arise when writing specifications of sequences and lists. In each of these cases, the first rule will be reduced for the first item only; and the second rule will be reduced for the second and all succeeding items:

```
list      :  item  
          |  list ',' item  
          ;
```

and

```
seq       :  item  
          |  seq item  
          ;
```

With right recursive rules, such as the following, the parser is a bit bigger; and the items are seen and reduced from right to left:

```
seq       :  item  
          |  item seq  
          ;
```

More seriously, an internal stack in the parser is in danger of overflowing if a very long sequence is read. The user should use left recursion wherever reasonable.

It is worth considering if a sequence with zero elements has any meaning, and if so, consider writing the sequence specification as in the following, using an empty rule:

```
seq  : /* empty */  
      | seq item  
      ;
```

Once again, the first rule would always be reduced exactly once before the first item was read, and then the second rule would be reduced once for each item read.

Permitting empty sequences often leads to increased generality. However, conflicts might arise if yacc is asked to decide which empty sequence it has seen when it hasn't seen enough to know!

12. Lexical Considerations

Some lexical decisions depend on context.

For example, the lexical analyzer might want to delete blanks normally but not within quoted strings, or names might be entered into a symbol table in declarations but not in expressions.

One way of handling this situation is to create a global flag that is examined by the lexical analyzer and set by actions. For example, the following example specifies a program that consists of zero or more declarations followed by zero or more statements. The flag **dflag** is 0 when reading statements and 1 when reading declarations, except for the first token in the first statement. This token must be seen by the parser before it can tell that the declaration section has ended and the statements have begun. In many cases, this single token exception does not affect the lexical scan.

```

%{
    int dflag;
%}
... other declarations ...

%%

prog      :   decls  stats
           ;

decls     :   /* empty */
           {
               dflag = 1;
           }
           |   decls  declaration
           ;

stats     :   /* empty */
           {
               dflag = 0;
           }
           |   stats  statement
           ;

... other rules ...

```

This kind of "back-door" approach can be elaborated to a noxious degree. Nevertheless, it represents a way of doing some things that are difficult if not impossible to do otherwise.

13. Reserved Words

Some programming languages permit you to use words like `if`, which are normally reserved as label or variable names, provided that such use does not conflict with the legal use of these names in the programming language. This is extremely hard to do in the framework of `yacc`. It is difficult to pass information to the lexical analyzer telling it "this instance of `if` is a keyword and that instance is a variable". The user can make a stab at it using the mechanism described in the last subsection, but it is difficult.

A number of ways of making this easier are under advisement. Until then, it is better that the keywords be **reserved**, i.e., forbidden for use as variable names.

14. Simulating Error and Accept in Actions

The parsing actions of error and accept can be simulated in an action by use of macros **YYACCEPT** and **YYERROR**.

The **YYACCEPT** macro causes **yyparse()** to return the value 0.

YYERROR causes the parser to behave as if the current input symbol had been a syntax error. The function **yyerror()** is called, and error recovery takes place.

These mechanisms can be used to simulate parsers with multiple end-markers or context sensitive syntax checking.

15. Accessing Values in Enclosing Rules

An action may refer to values returned by actions to the left of the current rule. The mechanism is simply the same as with ordinary actions, a dollar sign followed by a digit.

```

sent      :  adj noun verb adj noun
          {
            look at the sentence ...
          }
          ;
adj       :  THE
          {
            $$ = THE;
          }
          |  YOUNG
          {
            $$ = YOUNG;
          }
          ...
          ;
noun      :  DOG
          {
            $$ = DOG;
          }
          |  CRONE
          {
            if( $0 == YOUNG )
            {
              printf( "what?\n" );
            }
            $$ = CRONE;
          }
          ;
          ...

```

In this case, the digit may be 0 or negative.

In the action following the word **CRONE**, a check is made that the preceding token shifted was not **YOUNG**. Obviously, this is only possible when a great deal is known about what might precede the symbol **noun** in the input.

There is also a distinctly unstructured flavor about this. Nevertheless, at times this mechanism prevents a great deal of trouble especially when a few combinations are to be excluded from an otherwise regular structure.

16. Arbitrary Value Types

By default, the values returned by actions and the lexical analyzer are integers.

The yacc program can also support values of other types including structures.

The yacc program keeps track of the types and inserts appropriate union member names so that the resulting parser is strictly type checked.

The yacc value stack is declared to be a **union** of the various types of values desired.

The user declares the union and associates union member names to each token and nonterminal symbol having a value.

When the value is referenced through a **\$\$** or **\$n** construction, yacc will automatically insert the appropriate union name so that no unwanted conversions take place.

Type checking commands such as lint is far more silent.

There are three mechanisms used to provide for this typing.

- First, there is a way of defining the union. This must be done by the user since other programs, notably the lexical analyzer, must know about the union member names.
- Second, there is a way of associating a union member name with tokens and nonterminals.
- Third, there is a mechanism for describing the type of those few values where yacc cannot easily determine the type.

To declare the union, the user includes the following in the declaration section:

```
%union
{
    body of union ...
}
```

This declares the yacc value stack and the external variables `yylval` and `yyval` to have type equal to this union.

If yacc was invoked with the `-d` option, the union declaration is copied onto the `y.tab.h` file.

Alternatively, the union may be declared in a header file, and a `typedef` used to define the variable `YYSTYPE` to represent this union.

Thus, the header file might have said the following, instead:

```
typedef union
{
    body of union ...
}
YYSTYPE;
```

The header file must be included in the declarations section by use of `%{` and `%}`.

Once `YYSTYPE` is defined, the union member names must be associated with the various terminal and nonterminal names.

The following construction is used to indicate a union member name.

```
< name >
```


YACC

If this follows one of the keywords **%token**, **%left**, **%right**, and **%nonassoc**, the union member name is associated with the tokens listed.

For example, the following causes any reference to values returned by these two tokens to be tagged with the union member name **optype**:

```
%left  <optype>  '+'  '-'
```

Another keyword, **%type**, is used to associate union member names with nonterminals. For example, the following may be used to associate the union member **nodetype** with the nonterminal symbols **expr** and **stat**.

```
%type  <nodetype>  expr  stat
```

There remains a couple of cases where these mechanisms are insufficient. If there is an action within a rule, the value returned by this action has no *a priori* type.

Similarly, reference to left context values (such as **\$0**) leaves **yacc** with no easy way of knowing the type. In this case, a type can be imposed on the reference by inserting a union member name between **<** and **>** immediately after the first **\$**, as in the following example:

```
rule      :  aaa  
          {  
            $<intval>$ = 3;  
          }  
          bbb  
          {  
            fun( $<intval>2, $<other>0 );  
          }  
;
```

This syntax has little to recommend it, but the situation arises rarely.

A sample specification is given in Appendix 4.3.

The facilities in this subsection are not triggered until they are used. In particular, the use of **%type** will turn on these mechanisms. When they are used, there is a fairly strict level of checking.

For example, use of **\$n** or **\$\$** to refer to something with no defined type is diagnosed.

If these facilities are not triggered, the **yacc** value stack is used to hold **int**'s, as was true historically.

17. Appendix 4.1

This section contains an example which gives the complete yacc applications for a small desk calculator.

The calculator has 26 registers labeled **a** through **z** and accepts arithmetic expressions made up of the following operators:

Arithmetic Operators	
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus (Remainder)
&	Binary AND
	Binary OR
=	Assignment

If an expression at the top level is an assignment, the value is printed. Otherwise, the expression is printed.

As in C language, an integer that begins with 0 (zero) is assumed to be octal. Otherwise, it is assumed to be decimal.

As an example of a yacc specification, the desk calculator does a reasonable job of showing how precedence and ambiguities are used and demonstrates simple recovery.

The major oversimplifications are that the lexical analyzer is much simpler for most applications, and the output is produced immediately line by line.

Note the way that decimal and octal integers are read in by grammar rules. This job is probably better done by the lexical analyzer.

```

%{
# includes<stdio.h>
# includes<ctype.h>

int regs[26];
int base;

%}
%start list
%token DIGIT LETTER
%left '|'
%left '&'
%left '+' '-'
%left '*' '/' '%'
%left UMINUS /* precedence for unary minus */
%%
/* beginning of rule section */

list : /* empty */
    | list stat '\n'
    | list error '\n'
    {
        yyerrorok;
    }
;

stat : expr
    {
        printf( "%dn", $1 );
    }
    | LETTER '=' expr
    {
        regs[$1] = $3
    }
;

```

YACC

```
expr : '(' expr ')'
    {
        $$ = $2;
    }
    | expr '+' expr
    {
        $$ = $1 + $3;
    }
    | expr '-' expr
    {
        $$ = $1 - $3;
    }
    | expr '*' expr
    {
        $$ = $1 * $3;
    }
    | expr '/' expr
    {
        $$ = $1/$3;
    }
    | exp '%' expr
    {
        $$ = $1 % $3;
    }
    | expr '&' expr
    {
        $$ = $1 & $3;
    }
    | expr '|' expr
    {
        $$ = $1 | $3;
    }
    | '-' expr %prec UMINUS
    {
        $$ = - $2;
    }
    | LETTER
    {
        $$ = reg[$1];
    }
```

```
    }  
    | number  
    ;  
number : DIGIT  
    {  
        $$ = $1; base = ($1==0) ? 8 ; 10;  
    }  
    | number DIGIT  
    {  
        $$ = base * $1 + $2  
    }  
    ;
```

YACC

```
%% /* start of program */
/*
 * lexical analysis routine
 * return LETTER for lowercase letter
 * (i.e., yylval = 0 through 25)
 * returns DIGIT for digit
 * (i.e., yylval = 0 through 9)
 * all other characters are returned immediately
 *
 */

yylex( )
{
    int c;
    while (c=getchar( ) ) == ' ' /* skip blanks */
        ;
    if( islower( c ))
    {
        yylval = c - 'a';
        return( LETTER );
    }
    if( isdigit( c ))
    {
        yylval = c - '0';
        return( DIGIT );
    }
    return( c );
}
```

18. Appendix 4.2

This appendix has a description of the yacc input syntax as a yacc specification.

Context dependencies, etc. are not considered.

The yacc input specification language is most naturally specified as an LR(2) grammar. The sticky part comes when an identifier is seen in a rule immediately following an action. If this identifier is followed by a colon, it is the start of the next rule. Otherwise, it is a continuation of the current rule which just happens to have an action embedded in it.

As implemented, the lexical analyzer looks ahead after seeing an identifier and decides whether the next token (skipping blanks, newlines, and comments, etc.) is a colon. If so, it returns the token **C_IDENTIFIER**. Otherwise, it returns **IDENTIFIER**.

Literals (quoted strings) are also returned as **IDENTIFIERS** but never as part of **C_IDENTIFIERS**.

YACC INPUT GRAMMAR		
BASIC ENTRIES		
%token	IDENTIFIER	Includes identifiers and literals.
%token	C_IDENTIFIER	Identifier (but not literal) followed by a colon.
%token	NUMBER	Zero through nine.

YACC

YACC INPUT GRAMMAR		
RESERVED WORDS		
%token	LEFT	%left => LEFT
%token	RIGHT	%right => RIGHT
%token	NONASSOC	%nonassoc => NONASSOC
%token	TOKEN	%token => TOKEN
%token	PREC	%prec => PREC
%token	TYPE	%type => TYPE
%token	START	%start => START
%token	UNION	%union => UNION
%token	MARK	That is, the %% mark.
%token	LCURL	That is, the %{ mark.
%token	RCURL	That is, the %} mark.

/* ASCII character literals stand for themselves */

%token spec

%%

spec : defs MARK rules tail

;

tail : MARK

{

In this action, eat up the rest of the file

}

/* empty: the second MARK is optional */

;

defs : /* empty */

defs def

;

defs : START IDENTIFIER

UNION

{

Copy union definition to output

}

LCURL

{

Copy C code to output file

```

RCURL
}
| ndefs rword tag nlist
;
rword : TOKEN
      | LEFT
      | RIGHT
      | NONASSOC
      | TYPE
      ;
tag : /* empty: union tag is optional */
    | '<' IDENTIFIER '>'
    ;
nlist : nmno
      | nlist nmno
      | nlist ',' nmno
      ;
nmno : IDENTIFIER /* Note: literal illegal with %type */
      | IDENTIFIER NUMBER /* Note: illegal with %type */
      ;

```

YACC

/* rule section */

```
rule      : C_IDENTIFIER rbody proc
          | rules rule
          ;

rule      : C_IDENTIFIER rbody prec
          | '[' rbody prec
          ;

rbody     : /* empty */
          | rbody IDENTIFIER
          | rbody act
          ;

act       : '{'
          {
            Copy action translate $$'s etc.
          }
          ;

prec      : /* empty */
          | PREC IDENTIFIER
          | PREC IDENTIFIER act
          | prec';'
          ;
```

19. Appendix 4.3

This appendix gives an example of a grammar using some of the advanced features.

The desk calculator example in Appendix 4.1 is modified to provide a desk calculator that does floating point interval arithmetic.

The calculator understands floating point constants, as well as the arithmetic operations $+$, $-$, $*$, $/$, unary $-$, and the letters a through z .

The calculator also understands intervals written as is the following example, where X is less than or equal to Y :

(X,Y)

There are 26 interval valued variables A through Z that may also be used.

The usage is similar to that in Appendix 4.1. That is, assignments return no value and print nothing while expressions print the (floating or interval) value.

Intervals are represented by a structure consisting of the left and right endpoint values stored as doubles. This structure is given a type name, `INTERVAL`, by using `typedef`.

The yacc value stack can also contain floating point scalars and integers which are used to index into the arrays holding the variable values.

The entire strategy depends strongly on being able to assign structures and unions in C language. In fact, many of the

YACC

actions call functions that return structures as well.

Note the use of **YYERROR** to handle error conditions — division by an interval containing 0 and an interval presented in the wrong order. The error recovery mechanism of yacc is used to throw away the rest of the offending line.

In addition to the mixing of types on the value stack, this grammar also demonstrates an interesting use of syntax to keep track of the type (for example, scalar or interval) of intermediate expressions.

Scalars can be automatically promoted to an interval if the context demands an interval value. This causes a large number of conflicts when the grammar is run through yacc—18 Shift/Reduce and 26 Reduce/Reduce. The problem can be seen by looking at the following input lines:

2.5 + (3.5 - 4.)

and

2.5 + (3.5,4)

Notice that the 2.5 is to be used in an interval value expression in the second example, but this fact is not known until the comma is read. By this time, 2.5 is finished, and the parser cannot go back and change its mind.

More generally, it might be necessary to look ahead an arbitrary number of tokens to decide whether to convert a scalar to an interval.

This problem is evaded by having two rules for each binary interval valued operator — one when the left operand is a scalar and one when the left operand is an interval. In the second case, the right operand must be an interval, so the conversion will be applied automatically.

Despite this evasion, there are still many cases where the conversion may be applied or not, leading to the above conflicts. They are resolved by listing the rules that yield scalars first in the specification file. In this way, the conflict will be resolved in the direction of keeping scalar valued expressions scalar valued until they are forced to become intervals.

This way of handling multiple types is very instructive but not very general. If there were many kinds of expression types instead of just two, the number of rules needed would increase dramatically and the conflicts even more dramatically. Thus, while this example is instructive, it is better practice in a more normal programming language environment to keep the type information as part of the value and not as part of the grammar.

Finally, a word about the lexical analysis. The only unusual feature is the treatment of floating point constants.

The C language library routine `atof()` is used to do the actual conversion from a character string to a double precision value.

If the lexical analyzer detects an error, it responds by returning a token that is illegal in the grammar provoking a syntax error in the parser and thence error recovery.

YACC

```
%{  
  
#include <stdio.h>  
#include <ctype.h>  
  
typedef struct interval  
{  
    double lo, hi;  
} INTERVAL;  
  
INTERVAL vmul(), vdiv();  
  
double atof();  
double dreg[26];  
INTERVAL vreg[26];  
  
%}  
  
%start line  
  
%union  
{  
    int ival;  
    double dval;  
    INTERVAL vval;  
}  
  
%token <ival> DREG VREG /*indices into dreg, vreg */  
%token <dval> CONST /* floating point constant */  
  
%type <dval> dexp /* expression */  
%type <vval> vexp /* interval expression */  
  
/* precedence information about the operators */  
  
%left '+' '-'  
%left '*' '/'  
%left UMINUS /* precedence for unary minus */
```

```

%%
lines : /* empty */
      | lines line
      ;
line  : dexp '\n'
      {
        printf( "%15.8f\n", $1 );
      }
      | vexp '\n'
      {
        printf( "(%15.8f , %15.8f )\n", $1.1o, $1.hi );
      }
      | DREG '=' '\n'
      {
        dreg[$1] = $3;
      }
      | VREG '-' vexp '\n'
      {
        vreg[$1] = $3;
      }
      | error '\n'
      {
        yyerrorok;
      }
      ;

dexp  : CONST
      | DREG
      {
        $$ = dreg[$1]
      }
      | dexp '+' dexp
      {
        $$ = $1 + $3
      }
      | dexp '-' dexp
      {
        $$ = $1 - $3
      }

```


YACC

```

| dexp '*' dexp
{
    $$ = $1 * $3
}
| dexp '/' dexp
{
    $$ = $1 / $3
}
| '-' dexp    %prec UMINUS
{
    $$ = - $2
}
| '(' dexp ')'
{
    $$ = $2
}
;

vexp : dexp
{
    $$hi = $$lo = $1;
}
| '(' dexp ',' dexp ')'
{
    $$lo = $2;
    $$hi = $4;
    If( $$lo > $$hi )
    {
        printf( "interval out of order n" );
        YYERROR;
    }
}
| VREG
{
    $$ = vreg[$1]
}
| vexp '+' vexp
{
    $$hi = $1hi + $3hi;
}

```

```

    $$lo = $1.lo + $3.lo
  }
  dexp '+' vexp
  {
    $$hi = $1 + $3.hi;
    $$lo = $1 + $3.lo
  }
  vexp '=' vexp
  {
    $$hi = $1.hi - $3.lo;
    $$lo = $1.lo - $3.hi
  }
  dexp '-' vdep
  {
    $$hi = $1 - $3.lo;
    $$lo = $1 - $3.hi
  }
  vexp '*' vexp
  {
    $$ = vmul( $1.lo,$.hi,$3 )
  }
  dexp '*' vexp
  {
    $$ = vmul( $1, $1, $3 )
  }
  vexp '/' vexp
  {
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1.lo, $1.hi, $3 )
  }
  dexp '/' vexp
  {
    if( dcheck( $3 ) ) YYERROR;
    $$ = vdiv( $1.lo, $1.hi, $3 )
  }
  '-' vexp    %prec UMINUS
  {
    $$hi = -$2.lo; $$lo = -$2.hi
  }

```

YACC

```

        | '(' vexp ')'
        }
        $$ = $2
    }
    ;

%%

# define BSZ 50 /* buffer size for floating point number */

/*
 *lexical analysis
 */

yylex( )
{
    register c;
    while ((c=getchar()) == ' ') /* skip blanks */
        ;
    if(isupper(c))
    {
        yylval.ival = c - 'A'
        return(VREG);
    }
    if(islower(c))
    {
        yylval.ival = c - 'a',
        return(DREG);
    }
}

/*
 * gobble up digits. points, exponents
 */
if(isdigit(c) | c == '.')
{
    char buf[BSZ + 1], *cp = buf;
    int dot = 0, exp = 0;

    for(;;(cp - buf) < BSZ ; ++cp,c=getchar())
    {
        *cp = c;
        if(isdigit(c))
            continue;

```

```

    if(c == '.')
    {
        if(dot++ || exp)
            return( '.' ); /* causes syntax error */
        continue;
    }
    if(c == 'e')
    {
        if( exp++ )
            return( 'e' ); /* causes syntax error */
        continue;
    }
    break;          /* end of number */
}
*cp = '\0';
if((cp - buff) >= BSZ)
    printf( "constant too long truncated\n");
else
    ungetc(c, stdin); /* push back last char read */
yylval.dval = atof(buf);
return(CONST);
}
return(c);
}

```

YACC

```
/*  
 * returns the smallest interval  
 * between a, b, c and d  
 */
```

INTERVAL hilo(a, b, c, d)

double a, b, c, d;

```
{  
    INTERVAL v;  
    if( a > b )  
    {  
        v.hi = a;  
        v.lo = b;  
    }  
    else  
    {  
        v.hi = b;  
        v.lo = a;  
    }  
    if( c > d )  
    {  
        if( c > v.hi )  
            v.hi = c;  
        if( d < v.lo )  
            v.lo = d;  
    }  
    else  
    {  
        if( d > v.hi )  
            v.hi = d;  
        if( c < v.lo )  
            v.lo = c;  
    }  
    return( v );  
}
```

```
INTERVAL vmul( a, b, v )  
double a, b;  
INTERVAL v;  
{  
    return( hilo( a*v.hi, a*v.lo, b*v.hi, b*v.lo ) );  
}
```

```
dcheck( v )  
INTERVAL v;  
{  
    if( v.hi >=0.&& v.lo <=0. )  
    {  
        printf( "divisor internal contains 0.\n" );  
        return( 1 );  
    }  
    return( 0 );  
}
```

```
INTERVAL vdiv( a, b, v )  
double a, b;  
INTERVAL v;  
{  
    return( hilo( a/v.hi, a/v.lo, b/v.hi, b/v.lo ) );  
}
```

20. Appendix 4.4

This appendix mentions synonyms and features that are supported for historical continuity but, for various reasons, are not encouraged.

1. Literals may also be delimited by double quotes.
2. Literals may be more than one character long. If all the characters are alphabetic, numeric, or `_` the type number of the literal is defined just as if the literal did not have the quotes around it.

Otherwise, it is difficult to find the value for such literal.

The use of multicharacter literals is likely to mislead those unfamiliar with `yacc` since it suggests that `yacc` is doing a job which must be actually done by the lexical analyzer.

3. Most places where `(%)` is legal, backslash (`\`) may be used. In particular, `\\` is the same as `%%`, `\left` the same as `%left`, etc.
4. There are a number of other synonyms:

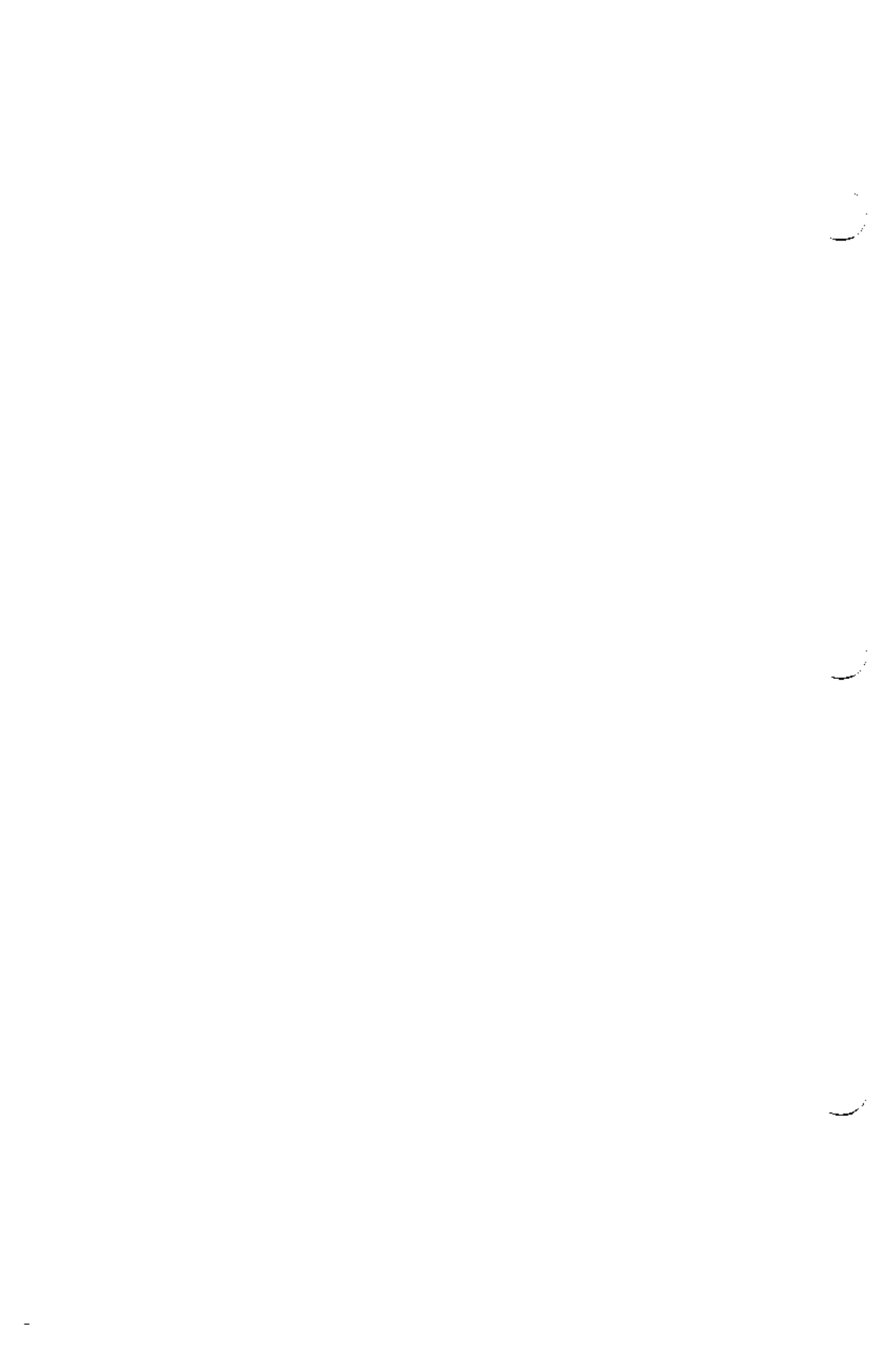
<code>%<</code>	is the same as	<code>%left</code>
<code>%></code>	is the same as	<code>%right</code>
<code>%binary</code>	is the same as	<code>%nonassoc</code>
<code>%2</code>	is the same as	<code>%nonassoc</code>
<code>%0</code>	is the same as	<code>%token</code>
<code>%term</code>	is the same as	<code>%token</code>
<code>%=</code>	is the same as	<code>%prec</code>

5. Action may also have the form

`= { ... }`

and the curly braces can be dropped if the action is a single C language statement.

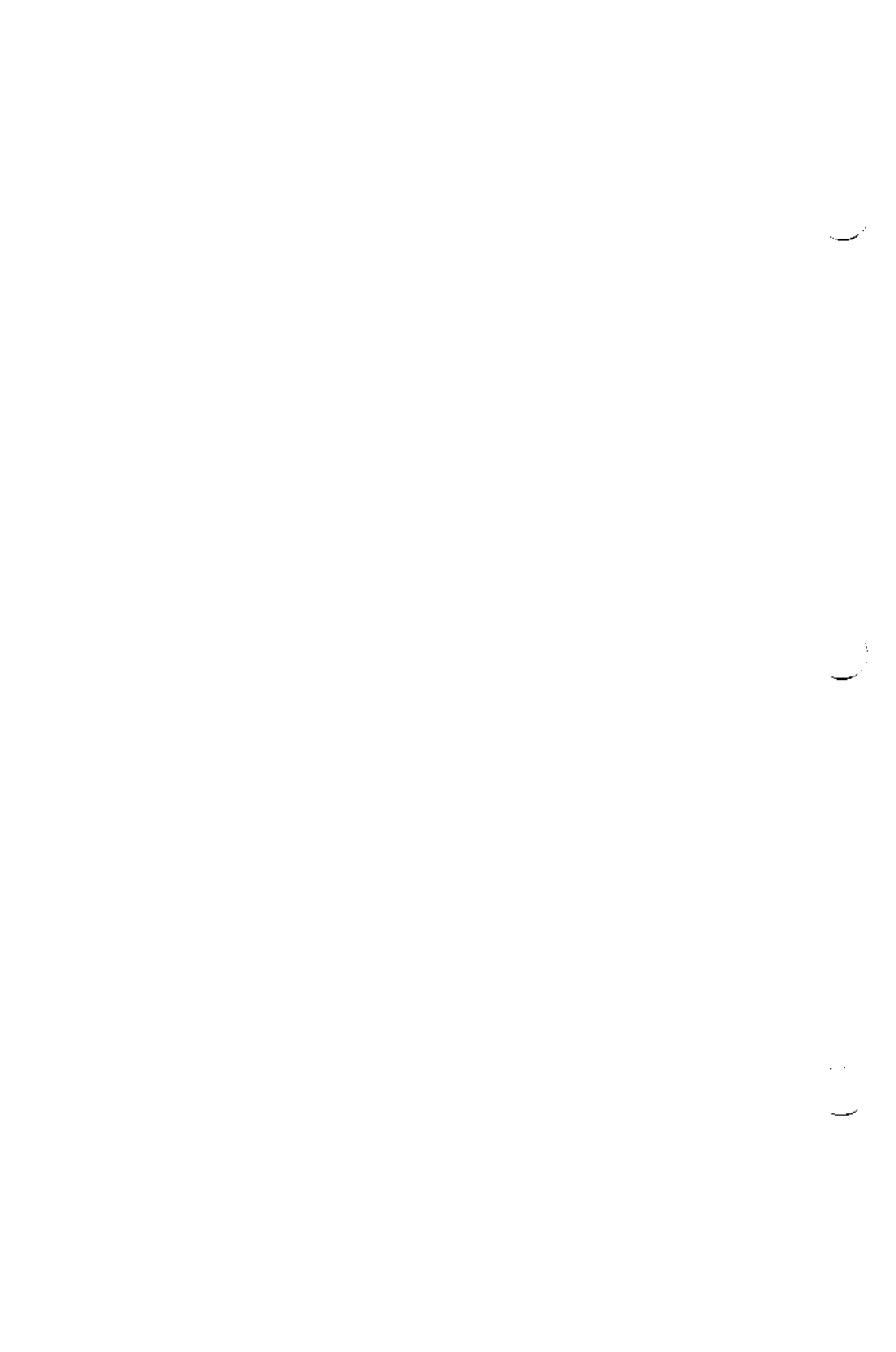
6. The C language code between %{ and %} use to be permitted at the head of the rules section as well as in the declaration section.



Chapter 5: BC

CONTENTS

1. Introduction	1
2. Usage	2
3. Syntax	4
3.1 Comments	4
3.2 Constants	4
3.3 Keywords	4
3.4 Identifiers	4
3.4.1 Functions	5
3.4.2 Arrays or Subscripted Variables	8
3.4.3 Storage Classes	8
3.5 Statements	9
3.6 Expressions	10
3.7 Assignment Statements	13
3.8 Bases — “ibase” and “obase”	16
3.9 Scaling	17
3.10 Control Statements	19
3.10.1 Relational Operators	20
3.10.2 The “if” Statement	20
3.10.3 The “while” Statement	21
3.10.4 The “for” Statement	21



Chapter 5

BC —

AN ARBITRARY PRECISION DESK CALCULATOR LANGUAGE

1. Introduction

The **bc** language and compiler were developed on the UNIXTM operating system to facilitate arbitrary precision arithmetic.

The output of the **bc** compiler is interpreted and executed by a collection of routines that can input, output and do arithmetic on infinitely large integers and on scaled fixed-point numbers.

The **bc** routines are based on a dynamic storage allocator. Overflow does not occur until all available core storage is exhausted.

The **bc** language has a complete control structure as well as immediate-mode operation. Functions can be defined and saved for later execution.

A small collection of library functions is also available, including **sin**, **cos**, **arctan**, **log**, **exponential** and **Bessel** functions of integer order.

UNIX is a trademark of AT&T Bell Laboratories.

BC

The **bc** compiler was written to utilize those routines of **DC** which are capable of doing arithmetic on integers of arbitrary size.

The **bc** compiler is not intended to provide a complete programming language, but may be effectively used to do a number of tasks. For example:

- Compile large integers
- Compute accurately to many decimal places
- Convert numbers from one base to another base

There is a scaling provision that permits the use of decimal point notation, and one made for input and output in bases other than decimal. Numbers can be converted from decimal to octal simply by setting the output base to eight. The limit on the number of digits that can be handled depends on the amount of core storage available.

2. Usage

Entering the command:

```
$ bc
```

enables you to execute **bc** commands directly.

If the calculations you need to perform are complicated, you may find it more efficient to define the functions or procedures you need in a file and give the file as an argument to **bc** on the command line using the **-f** option.

The **bc** reads and executes the argument file before accepting commands from the keyboard. This is the way **bc** loads programs and function definitions.

Bc has its own set of library functions which can be accessed with the **-l** (library) option. The **bc** library functions include:

s	sine
c	cosine
a	arctangent
l	natural logarithm
e	exponential
j(n,x)	bessel function integer order

The **bc** library option also initially sets the **scale** to 20, but this can be reset using the **scale** function call. (See the section on "Functions".)

For the immediate evaluation of simple arithmetic expressions which do not involve standard **bc** library functions or need any specially user-designed functions, simply access **bc** directly. To do a simple addition, first issue the **bc** command as demonstrated above, and then enter the calculation to be done. For example:

```
$ bc
142857 + 285714
```

bc will then respond immediately with the result: (Response from **bc** indicated with **bold**.)

```
$ bc
142857 + 285714
428571
```

Then, to exit **bc**, use either **quit** or CTRL-D, which will stop execution of a **bc** program and return your shell prompt.

NOTE: The **quit** statement is not treated as an executable statement, and so cannot be used in a function definition or in an **if**, **for** or **while** statement.

Even if you were to put this explicit addition request, or series of such requests, in a file, and use that file as an argument to

BC

bc on the command line; once it completed the calculations, **bc** would wait for further input from the terminal. Reaching the end of the command argument file will NOT instruct **bc** to exit.

3. Syntax

The syntax of **bc** is very similar to that of the C language, which should help those of you already familiar with C to begin using **bc** quickly and with a minimum of difficulty.

3.1 Comments

The characters **/*** introduce a comment which terminates with the characters ***/**.

3.2 Constants

Constants consist of arbitrarily long numbers with an optional decimal point. The hexadecimal digits A through F are also recognized as digits with values 10 through 15, respectively.

Constants are "primitive expressions."

3.3 Keywords

auto	for	length	return	while
break	ibase	obase	scale	
define	if	quit	sqrt	

3.4 Identifiers

Identifiers, or named-expressions, are places where values are stored. Therefore, named-expressions are legal on the left side of an assignment. The value of a named-expression is the value stored in the place named.

There are three kinds of identifiers:

1. Simple identifiers

2. Array, or subscripted, variables
3. Function calls

All three types are single lowercase letters, but these identifiers do not conflict. That is, a **bc** program may have a simple variable identifier named **x**, an array named **x** and a function named **x** all of which are separate and distinct.

3.4.1 Functions

The name of a function is a single lowercase letter. Function names are permitted to coincide with simple variable names. Twenty-six different defined functions are permitted in addition to the 26 variable names. The input line

```
define a(x){
```

begins the definition of a function with one argument. This line must be followed by one or more statements which make up the body of the function ending with a right brace (**}**). The general form of a function is

```
define a(x) {
    ...
    ...
    return
}
```

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the "function arguments." A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments.

Return of control from a function occurs when a **return** statement is executed, or when the end of the function is reached.

BC

The **return** statement can take either of the following two forms:

```
return  
return(x)
```

In the first case, the value returned from the function is 0; and in the second, the value returned from the function is the expression in parentheses.

Variables used in the function can be declared as *automatic* by a statement of the form

```
auto x,y,z
```

There can be only one such **auto** statement in a function, and it must be the first statement in the definition. These automatic variables are allocated space and initialized to zero on entry to the function and thrown away on return (exit). The values of any variables with the same names outside the function are not disturbed. Functions may be called recursively and the automatic variables at each level of call are protected.

The parameters named in a function definition are treated in the same way as the automatic variables of that function with the single exception that they are given a value on entry to the function. The following is an example of a function definition:

```
define a(x,y){  
    auto z  
    z = x*y  
    return(z)  
}
```

The value of this function *a*, when called, is the product of its two arguments, "x" and "y."

A function is called by the appearance of its name followed by a string of arguments which are enclosed in parentheses and separated by commas. The result is unpredictable if the wrong number of arguments is used.

Functions with no arguments are defined and called using empty parentheses.

Using the function $a(x,y)$ defined above, the following input:

`a(7,3.14)`

would send the result 21.98 to standard output. Using this same function, the following input:

`z = a(a(3,4),5)`

would send the result 60 to standard output.

The following are brief descriptions of three pre-defined bc functions:

sqrt(expression) The result is the square root of the expression. The result is truncated in the least significant decimal place. The scale of the result is the scale of the expression or the value of `scale`, whichever is larger.

length(expression) The result is the total number of significant decimal digits in the expression. The scale of the result is zero.

scale(expression) The result is the scale of the expression. The scale of the result is zero.

3.4.2 Arrays or Subscripted Variables

A single lowercase letter variable name followed by an expression in brackets is called a *subscripted variable* or an *array variable*. The variable name is called the *array name*, and the expression in brackets is called the *subscript*.

Only 1-dimensional arrays are permitted. The names of arrays are permitted to coincide with the names of simple variables and function names. Any fractional part of a subscript is discarded before use. Subscripts must be greater than or equal to 0 and less than or equal to 2047.

Subscripted variables may be used in expressions, in function calls, and in return statements. An array name may be used as an argument to a function or may be declared as automatic in a function definition by the use of empty brackets.

```
f(a[])
define f(a[])
auto a[]
```

When an array name is so used, the whole contents of the array are copied for the use of the function and thrown away on exit from the function. Array names that refer to whole arrays cannot be used in any other contexts.

3.4.3 Storage Classes

There are only two storage classes in **bc**:

1. global
2. automatic (local).

Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function are local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as

auto are allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. The **auto** arrays are specified by the array name followed by empty square brackets.

Automatic variables in **bc** do not work in exactly the same way as in C language. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until return is made from the function, reference to these names refers only to the new values.

3.5 Statements

Statements must be separated by a semicolon or newline. Except where altered by control statements, execution is sequential.

When a statement is an expression (unless the main operator is an assignment) the value of the expression is printed followed by a newline character.

Statements may be grouped together and used when one statement is expected by surrounding them with braces **{ }**.

The following statement prints the string inside the quotes.

```
"any string"
```

The **break** statement causes termination of a **for** or **while** statement.

```
auto identifier [, identifier]
```

The **auto** statement causes the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the

BC

array identifiers. Array identifiers are specified by following the array name with empty square brackets. The **auto** statement must be the first statement in a function definition.

```
define ([parameter[,parameter...]]){  
    statements}
```

The **define** statement defines a function. The parameters may be ordinary identifiers or array names. Array names must be followed by empty square brackets.

```
return  
return (expression)
```

The **return** statement causes the following:

- Termination of a function
- Popping of the auto variables on the stack
- Specifies the results of the function.

The first form is equivalent to **return**(0). The result of the function is the result of the expression in parentheses.

The **quit** statement stops execution of a **bc** program and returns control to the UNIX system software when it is first encountered. Because it is not treated as an executable statement, it cannot be used in a function definition or in an **if**, **for**, or **while** statement.

3.6 Expressions

Any term in an expression may be prefixed by a minus sign to indicate that it is a negative (the **unary** minus sign).

The value of an expression is printed unless the main operator is an assignment.

Division by zero produces an error comment.

The following is a table of the operators that can be used with bc.

OPERATOR	FUNCTION
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remaindering - (integer result truncated toward zero)
^	Exponentiation
=	Assignment Operator

More complex expressions with several operators and with parentheses are interpreted with the following precedence:

$$\begin{array}{c}
 ^ \\
 * \quad \% \quad / \\
 + \quad -
 \end{array}$$

NOTES: Contents of parentheses are evaluated BEFORE material outside the parentheses.

Exponentiations are performed from right to left, while the other operators are performed from left to right.

a^b^c and $a^{(b^c)}$ are equivalent

$a/b*c$ is equivalent to $(a/b)*c$ and NOT to $a/(b*c)$.

Following are brief descriptions of the various types of expressions understood by bc:

- expression** The result is the negative of the expression.
- ++named-expression** The named expression is incremented by one. The result is the

	value of the named expression after incrementing.
--named-expression	The named expression is decremented by one. The result is the value of the named expression after decrementing.
named-expression ++	The named expression is incremented by one. The result is the value of the named expression before incrementing.
named-expression--	The named expression is decremented by one. The result is the value of the named expression before decrementing.
	The exponentiation operator binds right to left.
expression ^ expression	The result is the first expression raised to the power of the second expression. The second expression must be an integer. If a is the scale of the left expression and b is the absolute value of the right expression, then the scale of the result is $\min(a \times b, \max(\text{scale}, a))$
	The operators $*$, $/$, and $\%$ bind left to right.
expression * expression	The result is the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result is $\min(a + b, \max(\text{scale}, a, b))$
expression / expression	The result is the quotient of the two expressions. The scale of the result

is the value of **scale**.

expression % expression The % operator produces the remainder of the division of the two expressions. More precisely, $a \% b$ is $a - a / b * b$.

The scale of the result is the sum of the scale of the divisor and the value of **scale**.

The additive operators bind left to right.

expression + expression The result is the sum of the two expressions. The scale of the result is the maximum of the scales of the expressions.

expression - expression The result is the difference of the two expressions. The scale of the result is the maximum of the scales of the expressions.

3.7 Assignment Statements

The assignment operators bind right to left.

Ordinary variables are used as internal storage registers. They have single lowercase letter names, are used to hold integer values and have an initial value of 0. The statement

$$x = x + 3$$

has the effect of increasing by three the value of the contents of register x . In this case, although the increase in value is performed, that value is not printed. To print the value of x after the assignment, either explicitly call x , as in the following:

BC

```
x=x+3  
x
```

or surround the assignment with parentheses which instructs bc to treat the statement as the value of the result of the operation. The assignment can then be used anywhere an expression can. For example:

```
(x=x+3)
```

In this example, the value of x is incremented and the resulting value is printed.

The following is an example of the use of the value of an assignment statement even when it is not parenthesized. The input line:

```
x=a[i=i+1]
```

instructs bc to increment i before using it as a subscript and then assign the resulting value to x.

Since these registers must be unique, single lowercase letter names, only 26 of these named storage registers are available.

The assignment statements work in exactly the same manner as in the C programming language. The following table lists the assignment statement constructs:

ASSIGNMENT STATEMENTS		
$x=y=z$	is the same as	$x=(y=z)$
$x=+y$	is the same as	$x=x+y$
$x=-y$	is the same as	$x=x-y$
$x=-y$	is the same as	$x=-y$
$x=*y$	is the same as	$x=x*y$
$x=/y$	is the same as	$x=x/y$
$x=\%y$	is the same as	$x=x\%y$
$x=\^y$	is the same as	$x=x\^y$
$x++$	is the same as	$(x=x+1)-1$
$x--$	is the same as	$(x=x-1)+1$
$++x$	is the same as	$x=x+1$
$--x$	is the same as	$x=x-1$

$x=y=z$	is the same as	$x=(y=z)$
$x=+y$	is the same as	$x=x+y$
$x=-y$	is the same as	$x=x-y$
$x=-y$	is the same as	$x=-y$
$x=*y$	is the same as	$x=x*y$
$x=/y$	is the same as	$x=x/y$
$x=\%y$	is the same as	$x=x\%y$
$x=\^y$	is the same as	$x=x\^y$
$x++$	is the same as	$(x=x+1)-1$
$x--$	is the same as	$(x=x-1)+1$
$++x$	is the same as	$x=x+1$
$--x$	is the same as	$x=x-1$

•NOTE: In some of these constructions, spaces are significant. There is an important difference between $x=-y$ and $x=-y$. The first replaces x by $x-y$ and the replaces x by $-y$.

3.8 Bases – “ibase” and “obase”

There are two special internal quantities:

1. **ibase** (input base)
2. **obase** (output base)

The contents of **ibase** determines the base used for interpreting the numbers input, and is initially set to 10 (decimal). To set the input base to be something else, use the “=” assignment operator. For example, the following sets the input base to base 8:

```
ibase = 8  
11
```

Assuming that the output base is 10, **bc** would interpret 11 in base 8 and return the decimal value 9. This is an easy method of doing octal to decimal conversions.

If, after having changed the value of **ibase**, you want to change the input base back to decimal, you must use:

```
ibase = 12
```

Because, having changed the input base to 8, the number 10 would be interpreted as octal, and 10 in octal is equal to 8, therefore effecting no change on **ibase**.

For dealing in hexadecimal notation, the characters A through F are permitted in numbers (regardless of what base is in effect) and are interpreted as digits having values 10 through 15, respectively. The statement

```
ibase = A
```

changes the base to decimal regardless of the current input base. No error message is given if negative and large positive numbers are assigned to **ibase**, but they have no effect. No

mechanism has been provided for the input of arbitrary numbers in bases less than 1 and greater than 16.

The content of **obase** is used as the base for output numbers, and is initially set to 10 (decimal). Assuming that **ibase** is set to 10, the following input lines:

```
obase = 16
1000
```

produce the following output line:

```
3E8
```

Thus providing a simple decimal to hexadecimal conversion facility.

Very large output bases are permitted and are sometimes useful. For example, large numbers can be output in groups of five digits by setting **obase** to 100000.

Very large numbers are split across lines with 70 characters per line. To force the continuation of a line, end it with a backslash (\). Decimal output conversion is practically instantaneous, but output of very large numbers (i.e., more than 100 digits) with other bases is rather slow. Nondecimal output conversion of a 100-digit number takes about 3 seconds.

The **ibase** and **obase** have no effect on the course of internal computation or on the evaluation of expressions. They only affect input and output conversions, respectively.

3.9 Scaling

The number of digits after the decimal point of a number is referred to as its **scale**. Numbers may have up to 99 decimal digits after the decimal point. This fractional part may be retained for use in further computations by use of the third

internal quantity — scale.

The contents of **scale** must be no greater than 99 and no less than its initial value of 0. However, appropriate scaling can be arranged when more than 99 fraction digits are required.

When two scaled numbers are combined by means of one of the arithmetic operations, the result has a scale determined by the following rules:

Addition and Subtraction The scale of the result is the larger of the scales of the two operands. In this case, there is never any truncation of the result.

Multiplication The scale of the result is never less than the maximum of the two scales of the operands and never more than the sum of the scales of the operands. Subject to those two restrictions, the scale of the result is set equal to the contents of the internal quantity **scale**.

Division The scale of a quotient is the contents of the internal quantity **scale**. The scale of a remainder is the sum of the scales of the quotient and the divisor.

Exponentiation The result of an exponentiation is scaled as if the implied multiplications were performed. An exponent must be an integer.

Square root The scale of a square root is set to the maximum of the scale of the argument and the contents of **scale**.

All of the internal operations are actually carried out in terms of integers with digits being discarded when necessary. In every case where digits are discarded, truncation — NOT rounding — is performed.

The internal quantities **scale**, **ibase** and **obase** can be used in expressions just like other variables. The input line

scale = scale + 1

increases the value of **scale** by one, and the input line

scale

causes the current value of **scale** to be printed.

The value of **scale** retains its meaning as the number of decimal digits to be retained in internal computation even when **ibase** or **obase** are not equal to 10. The internal computations are always conducted in decimal regardless of the value of **ibase** and **obase**.

3.10 Control Statements

There are three control statements available with **bc**:

1. The "if" statement
2. The "while" statement
3. The "for" statement

The **if**, **while** and **for** statements may be used to alter the flow within programs or to cause iteration. The range of each of them is a simple statement or a compound statement. A *compound statement* consists of a collection of statements enclosed in braces. Additionally, each of these control structures relies

in part on the evaluation of a *relation*.

3.10.1 Relational Operators

Unlike all other operators, the relational operators are only valid as the object of an **if** or **while** statement or inside a **for** statement.

expression < *expression*
expression > *expression*
expression <= *expression*
expression >= *expression*
expression == *expression*
expression != *expression*

The following table illustrates the six *relational operators* and their definitions:

RELATIONAL OPERATORS	
OPERATOR	DEFINITION
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

NOTE: DO NOT use "=" instead of "==" as a relational operator. Unfortunately, both of these are legal, so there will be no diagnostic message, but "=" will NOT do a comparison. The "=" operator is an Assignment Operator.

3.10.2 The "if" Statement

The **if** statement causes execution of its range **if and only if** the relation is true. Then control passes to the next statement in sequence.

3.10.3 The "while" Statement

The **while** statement causes execution of its range repeatedly as long as the relation is true. The relation is tested **BEFORE** each execution of its range; and if the relation is false, control passes to the next statement beyond the range of the **while** statement.

The following function computes values of the exponential function by summing the appropriate series without regard for possible truncation errors:

```
scale = 20
define e(x){
    auto a, b, c, d, n
    a = 1
    b = 1
    c = 1
    d = 0
    n = 1
    while(1==1){
        a = a*x
        b = b*n
        c = c + a/b
        n = n + 1
        if(c==d) return(c)
        d = c
    }
}
```

3.10.4 The "for" Statement

The typical use of a **for** statement is for controlled iteration, For example:

```
for(expression1 ; relation ; expression2) statements
```

The **for** statement begins by executing **expression1**. Then the **relation** is tested. If the relation is true, the *statements* in the range of the **for** are executed. Then **expression2** is executed. The relation is then tested, etc.

BC

Following are a few more examples using the **for** statement. The first is a control statement which, if given a positive integer, will return the factorial of that number. The numbers in **bold** are responses from **bc**, the **\$**'s are the shell prompts and everything else is user input.

```
$ bc
define f(n){
  auto i, x
  x=1
  for(i=1; i<=n; i=i+1) x=x*i
  return(x)
}
f(5)
120
f(3)
6
quit
$
```

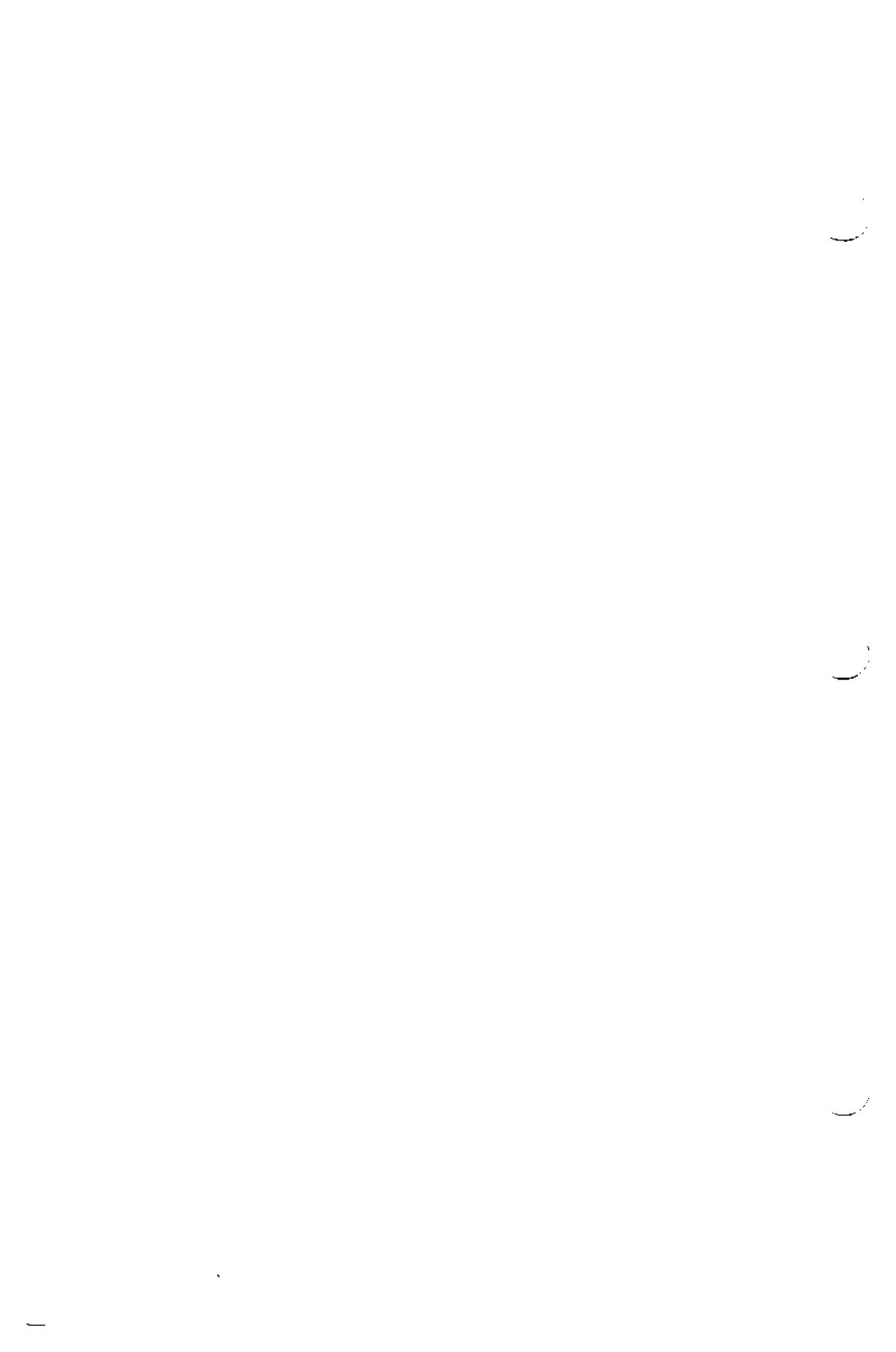
The following is the definition of a function that computes values of the binomial coefficient (m and n are assumed to be positive integers):

```
define b(n,m){
  auto x, j
  x=1
  for(j=1; j<=m; j=j+1) x=x*(n-j+1)/j
  return(x)
}
```

Chapter 6: DC

CONTENTS

1. Introduction	1
2. Representation of Numbers	3
3. The Allocator	4
4. Internal Arithmetic	6
5. Usage	7
6. Special Character Commands	8
7. Addition and Subtraction	10
8. Multiplication	11
9. Division	12
10. Remainder	13
11. Square Root	14
12. Exponentiation	15
13. Input Conversion and Base	16
14. Output Commands	17
15. Output Format and Base	18
16. Stack Commands	19
17. Subroutine Definitions and Calls	20
18. Internal Registers	21
19. Programming DC	22
20. Pushdown Registers and Arrays	24
21. Miscellaneous Commands	25
22. Design Choices	26



Chapter 6

DC -

A INTERACTIVE DESK CALCULATOR

1. Introduction

The **dc** program is an interactive desk calculator program implemented on the UNIXTM operating system to do arbitrary-precision integer arithmetic. It has provisions for manipulating scaled fixed-point numbers and for input and output in bases other than decimal.

The size of numbers that can be manipulated by **dc** is limited only by available core storage.

The **dc** program works like a stacking calculator using reverse Polish notation. Ordinarily, **dc** operates on decimal integers; but an input base, output base, and a number of fractional digits to be maintained can be specified.

A language called **bc** has been developed which accepts programs written in the familiar style of higher-level programming languages and compiles the output which is interpreted by **dc**. (See the chapter "BC - AN ARBITRARY PRECISION DESK CALCULATOR LANGUAGE" in the UniPlus⁺ Programming Tools Guide for more information.)

Some of the commands described below were designed for the compiler interface and are not easy for a human user to manipulate.

Numbers that are typed into **dc** are put on a pushdown stack. The **dc** commands work by taking the top number or two off the stack, performing the desired operation, and pushing the

DC

result on the stack. If an argument is given, input is taken from that file until its end, then it is taken from the standard input.

2. Representation of Numbers

Numbers are stored internally using a dynamic storage allocator. Numbers are kept in the form of a string of digits to the base 100 stored one digit per byte (centennial digits). The string is stored with the low-order digit at the beginning of the string.

For example, the representation of 157 is 57,1. After any arithmetic operation on a number, care is taken that all digits are in the range 0 to 99 and that the number has no leading zeros. The number zero is represented by the empty string.

Negative numbers are represented in the 100s complement notation, which is analogous to twos complement notation for binary numbers. The high-order digit of a negative number is always -1 and all other digits are in the range 0 to 99. The digit preceding the high-order -1 digit is never a 99.

The representation of -157 is 43,98,-1. This is called the **canonical form** of a number. The advantage of this kind of representation of negative numbers is ease of addition. When addition is performed digit by digit, the result is formally correct. The result need only be modified, if necessary, to put it into canonical form.

Because the largest valid digit is 99 and the byte can hold numbers twice that large, addition can be carried out and the handling of carries done later when it is convenient.

An additional byte is stored with each number beyond the high-order digit to indicate the number of assumed decimal digits after the decimal point. The representation of .001 is "1,3" (the **scale** has been **emboldened** to emphasize the fact that it is not the high-order digit. The value of this extra byte is called the **scale factor** of the number.

3. The Allocator

The **dc** program uses a dynamic string storage allocator for all of its internal storage. All reading and writing of numbers internally is through the allocator.

Associated with each string in the allocator is a 4-word header containing pointers to the beginning of the string, the end of the string, the next place to write, and the next place to read. Communication between the allocator and **dc** is via pointers to these headers.

The allocator initially has one large string on a list of free strings. All headers except the one pointing to this string are on a list of free headers.

Requests for strings are made by size. The size of the string actually supplied is the next higher power of two.

When a request for a string is made, the allocator first checks the free list to see if there is a string of the desired size. If none is found, the allocator finds the next larger free string and splits it repeatedly until it has a string of the right size.

Leftover strings are put on the free list. If there are no larger strings, the allocator tries to combine smaller free strings into larger ones. Since all strings are the result of splitting large strings, each string has a neighbor that is next to it in core and, if free, can be combined with it to make a string twice as long.

If a string of the proper length cannot be found, the allocator asks the system for more space. The amount of space on the system is the only limitation on the size and number of strings in **dc**. If the allocator runs out of headers at any time in the process of trying to allocate a string, it also asks the system for more space.

There are routines in the allocator for reading, writing, copying, rewinding, forward spacing, and backspacing strings. All string manipulation is done using these routines.

The reading and writing routines increment the read pointer or write pointer so that the characters of a string are read or written in succession by a series of read or write calls.

The write pointer is interpreted as the end of the information-containing portion of a string and a call to read beyond that point returns an end of string indication. An attempt to write beyond the end of a string causes the allocator to allocate a larger space and then copy the old string into the larger block.

4. Internal Arithmetic

All arithmetic operations are done on integers.

The operands (or operand) needed for the operation are popped from the main stack and their scale factors stripped off. Zeros are added or digits removed as necessary to get a properly scaled result from the internal arithmetic routine.

For example, if the scale of the operands is different and decimal alignment is required, as it is for addition, zeros are appended to the operand with the smaller scale. After performing the required arithmetic operation, the proper scale factor is appended to the end of the number before it is pushed on the stack.

A register called **scale** plays a part in the results of most arithmetic operations. The **scale** register limits the number of decimal places retained in arithmetic computations. The **scale** register may be set to the number on the top of the stack truncated to an integer with the **k** command.

The **K** command may be used to push the value of **scale** on the stack. The value of **scale** must be greater than or equal to 0 and less than 100. The descriptions of the individual arithmetic operations includes the exact effect of **scale** on the computations.

5. Usage

Any number of commands are permitted on a line.

Blanks and new-line characters are ignored except within numbers and in places where a register name is expected.

A number is an unbroken string of digits 0 through 9 and uppercase letters A through F (treated as digits with values 10 through 15, respectively). The number may be preceded by an underscore () to input a negative number and numbers may contain decimal points.

The value of a number is pushed onto the stack. The top two values on the stack may be added (+), subtracted (-), multiplied (*), divided (/), remaindered (%) and/or exponentiated (^) by using the appropriate operator.

The two entries are popped off the stack, and the result is pushed on the stack in their place.

The result of a division is an integer truncated toward zero.

An exponent must not have any digits after the decimal point.

6. Special Character Commands

The following is a list of special characters, and their functions in **dc**:

- [...]** Puts the bracketed character string onto the top of the stack.
- !** Interprets the rest of the line as a UNIX software command. Control returns to **dc** when the command terminates.
- ?** A line of input is taken from the input source (usually the console) and executed.
- c** All values on the stack are popped; the stack becomes empty.
- d** The top value on the stack is duplicated.
- f** All values on the stack and in registers are printed.
- i and I** The top value on the stack is popped and used as the number radix for further input.
- The command **I** pushes the value of the input base on the stack.
- k and K** The top of the stack is popped, and that value is used as a scale factor that influences the number of decimal places that are maintained during multiplication, division, and exponentiation. The scale factor must be greater than or equal to zero and less than 100.
- The **K** command may be used to push the value of scale on the stack.
- lx and Lx** The **I** command puts the contents of register **x** on top of the stack. The initial value of a new register is treated as a zero by the command **I**, but treated as an error by the command **L**.

- The **Lx** command pops the stack for register **x** and puts the result on the main stack.
- o and O** The top value on the stack is popped and used as the number radix for further output.
- The command **O** pushes the value of the output base on the stack. base is pushed onto the stack.
- p** The top value on the stack is printed. The top value remains unchanged.
- q and Q** Exits the program. If executing a string, the recursion level is popped by two. If **q** is uppercase, the top value on the stack is popped; and the string execution level is popped by that value.
- sx and Sx** The top of the main stack is popped and stored in a register named **x** (where **x** may be any character). The value of register **x** is pushed onto the stack. Register **x** is not altered.
- Sx** pushes the top value of the main stack onto the stack for the register **x**.
- v** Replaces the top element on the stack by its square root. The square root of an integer is truncated to an integer.
- x and X** The **x** command assumes the top of the stack is a string of **dc** commands, removes it from the stack, and executes it.
- The command **X** replaces the number on the top of the stack with its scale factor.
- z and Z** The value of the stack level is pushed onto the stack.
- The command **Z** replaces the top of the stack with its length.

7. Addition and Subtraction

The scales of two numbers are compared and trailing zeros are supplied to the number with the lower scale to give both numbers the same scale.

The number with the smaller scale is multiplied by 10 if the difference of the scales is odd. The scale of the result is then set to the larger of the scales of the two operands.

Subtraction is performed by negating the number to be subtracted and proceeding as in addition.

The addition is performed digit by digit from the low-order end of the number. The carries are propagated in the usual way. The resulting number is brought into **canonical form**, which may require stripping of leading zeros, or for negative numbers, replacing the high-order configuration 99, -1 by the digit -1. In any case, digits that are not in the range 0 through 99 must be brought into that range, propagating any carries or borrows that result.

8. Multiplication

The scales are removed from the two operands and saved. The operands are both made positive. Then multiplication is performed in a digit by digit manner that exactly follows the hand method of multiplying.

The first number is multiplied by each digit of the second number, beginning with its low-order digit.

The intermediate products are accumulated into a partial sum which becomes the final product.

The final product is put into the **canonical form** and its sign is computed from the signs of the original operands.

The scale of the result is set equal to the sum of the scales of the two operands. If that scale is larger than the internal register scale and also larger than both of the scales of the two operands, then the scale of the result is set equal to the largest of these three last quantities.

9. Division

The scales are removed from the two operands.

Zeros are appended, or digits are removed from the dividend to make the scale of the result of the integer division equal to the internal quantity scale.

The signs are removed and saved.

Division is performed much as it would be done by hand. The difference of the lengths of the two numbers is computed. If the divisor is longer than the dividend, zero is returned. Otherwise, the top digit of the divisor is divided into the top two digits of the dividend. The result is used as the first (high-order) digit of the quotient. If it turns out to be one unit too low, the next trial quotient is larger than 99; and this is adjusted at the end of the process.

The trial digit is multiplied by the divisor, the result subtracted from the dividend, and the process is repeated to get additional quotient digits until the remaining dividend is smaller than the divisor.

Finally, the digits of the quotient are put into the **canonical form** with propagation of carry as needed.

The sign is set from the sign of the operands.

10. Remainder

The division routine is called, and division is performed exactly as described. The quantity returned is the remains of the dividend at the end of the divide process.

Since division truncates toward zero, remainders have the same sign as the dividend.

The scale of the remainder is set to the maximum of the scale of the dividend and the scale of the quotient plus the scale of the divisor.

DC

11. Square Root

The scale is removed from the operand.

Zeros are added if necessary to make the integer result have a scale that is the larger of the internal quantity **scale** and the scale of the operand.

The method used to compute the square root is Newton's method with successive approximations by the following rule:

$$X_{n+1} = (X_n + Y/X_n)$$

The initial guess is found by taking the integer square root of the top two digits.

12. Exponentiation

Only exponents with 0 scale factor are handled.

The scale of the base is removed.

If the exponent is 0, the result is 1. If the exponent is negative, it is made positive, and the base is divided into 1.

The integer exponent is viewed as a binary number.

The base is repeatedly squared, and the result is obtained as a product of those powers of the base that correspond to the positions of the one-bits in the binary representation of the exponent.

Enough digits of the result are removed to make the scale of the result the same as if the indicated multiplication had been performed.

13. Input Conversion and Base

Numbers are converted to the internal representation as they are read in.

The scale stored with a number is simply the number of fractional digits input.

Negative numbers are indicated by preceding the number with an underscore (). The hexadecimal digits A through F correspond to the numbers 10 through 15 regardless of input base.

The **i** command can be used to change the base of the input numbers. This command pops the stack, truncates the resulting number to an integer, and uses it as the input base for all further input.

The input base (**ibase**) is initialized to 10 (decimal) but may, for example, be changed to 8 or 16 for octal or hexadecimal to decimal conversions.

No mechanism has been provided for the input of arbitrary numbers in bases less than 1 or greater than 16.

The command **I** pushes the value of the input base on the stack.

14. Output Commands

The command **p** causes the top of the stack to be printed. It DOES NOT remove the top of the stack.

All of the stack and internal registers are output by typing the command **f**.

The **o** command is used to change the output base (**obase**). This command uses the top of the stack truncated to an integer as the base for all further output.

The output base is initialized to 10 (decimal). It works correctly for any base.

The command **O** pushes the value of the output base on the stack.

15. Output Format and Base

The input and output bases only affect the interpretation of numbers on input and output. They have no effect on arithmetic computations.

Large numbers are output with 70 characters per line; a backslash (\) indicates a continued line. All choices of input and output bases work correctly, although not all are useful. A particularly useful output base is 100000, which has the effect of grouping digits in fives. Bases of 8 and 16 are used for decimal-octal or decimal-hexadecimal conversions.

16. Stack Commands

The command **c** clears the stack.

The command **d** pushes a duplicate of the number on the top of the stack onto the stack.

The command **z** pushes the stack size on the stack.

The command **X** replaces the number on the top of the stack with its scale factor.

The command **Z** replaces the top of the stack with its length.

17. Subroutine Definitions and Calls

Enclosing a string in brackets “[]” pushes the ASCII string on the stack.

The **q** command quits or (in executing a string) pops the recursion levels by two.

18. Internal Registers

Numbers or strings may be stored in internal registers or loaded on the stack from registers with the commands **s** and **l**:

- sx** The command **sx** pops the top of the stack and stores the result in register **x**. The **x** can be any character — even blank or newline, is a valid register name.
- lx** The command **lx** puts the contents of register **x** on the top of the stack. The **x** can be any character — even blank or newline, is a valid register name.

NOTE: The **l** command has no effect on the contents of register **x**. The **s** command, however, is destructive.

19. Programming DC

By combining a few of the available constructs, such as:

- The load, store, execute and print commands,
- the “[]” construction to store strings and
- the testing commands,

it is possible to program dc.

The testing commands

`<x >x =x !<x !>x !=x`

cause the top two elements of the stack to be popped and compared. Register `x` is executed if the top two elements of the stack satisfy the stated relation. Exclamation point is negation.

For example, the following expressions instruct dc to print the numbers 0 through 9:

```
[lip1+ si li10>]sa
0si lax
```

Consider the first expression in this example:

```
[lip1+ si li10>]sa
```

This first instruction makes use of the “[]” construction for storing strings. The entire expression is stored as a character string on top of the stack. Reading from left to right, this character array holds the following commands:

- load the contents of register `i` on top of the stack, and print it. (Note: Using the `print` command does not remove the top of the stack.)
- Add (+) 1 to the value found on top of the stack, and place the result on top of the stack.
- store the value currently found on top of the stack in register `i`.

- load the contents of register **i** on top of the stack, then load the number "10" onto the stack. Use the testing operator **>** on these top two stack elements and see if 10 is greater than the number which was loaded from register **i**. If 10 is greater, then execute register **a**.

This is the "control element" of the **dc** example program, since it will stop the processing of the expressions as soon as the value in register **i** is equal to 10.

Continuing to read from left to right, the character array is stored in register **a**.

The second line of the example contains, the expressions:

0si lax

The **0si** instruction clears register **i** by storing 0 in that register, thereby clobbering any previous value it may have had.

The **lax** instruction loads the contents of register **a** on top of the stack and executes it.

20. Pushdown Registers and Arrays

NOTE: These commands are designed for use by a compiler, not directly by programmers. They involve pushdown registers and arrays.

Dc can be thought of as having individual stacks for each register. These registers are operated on by the commands **S** and **L**:

- Sx** **Sx** pushes the top value of the main stack onto the stack for the register **x**.
- Lx** **Lx** pops the stack for register **x** and puts the result on the main stack.

The commands **s** and **l** also work on registers but not as push-down stacks. The command **l** does not affect the top of the register stack, but **s** destroys what was there before.

The commands to work on arrays are **:** and **;**:

- :x** The command **:x** pops the stack and uses this value as an index into the array **x**. The next element on the stack is stored at this index in **x**. An index must be greater than or equal to 0 and less than 2048.
- ;x** The command **;x** loads the main stack from the array **x**. The value on the top of the stack is the index into the array **x** of the value to be loaded.

21. Miscellaneous Commands

The command **!** interprets the rest of the line as a UNIX software command and passes it to the UNIX operating system to execute.

One other compiler command is **Q**. This command uses the top of the stack as the number of levels of recursion to skip.

22. Design Choices

The reason for the use of a dynamic storage allocator is that a general purpose program can be used for a variety of other tasks. The allocator has some value for input and for compiling (i.e., the bracket "[...]" commands) where it cannot be known in advance how long a string will be. The result is that at a modest cost in execution time:

- All considerations of string allocation and sizes of strings are removed from the remainder of the program.
- Debugging is made easier.
- The allocation method used wastes approximately 25 percent of available space.

The choice of 100 as a base for internal arithmetic seemingly has no compelling advantage. Yet the base cannot exceed 127 because of hardware limitations and at the cost of 5 percent in space debugging was made a great deal easier, and decimal output was made much faster.

The reason for a stack-type arithmetic design was to permit all **dc** commands from addition to subroutine execution to be implemented in essentially the same way. The result was a considerable degree of logical separation of the final program into modules with very little communication between modules.

The rationale for the lack of interaction between the scale and the bases is to provide an understandable means of proceeding after a change of base or scale (when numbers had already been entered).

An earlier implementation which had global notions of scale and base did not work out well. If the value of **scale** is interpreted in the current input or output base, then a change of base or scale in the midst of a computation causes great

confusion in the interpretation of the results. The current scheme has the advantage that the value of the input and output bases are only used for input and output, respectively, and they are ignored in all other operations.

The value of **scale** is not used for any essential purpose by any part of the program. It is used only to prevent the number of decimal places resulting from the arithmetic operations from growing beyond all bounds.

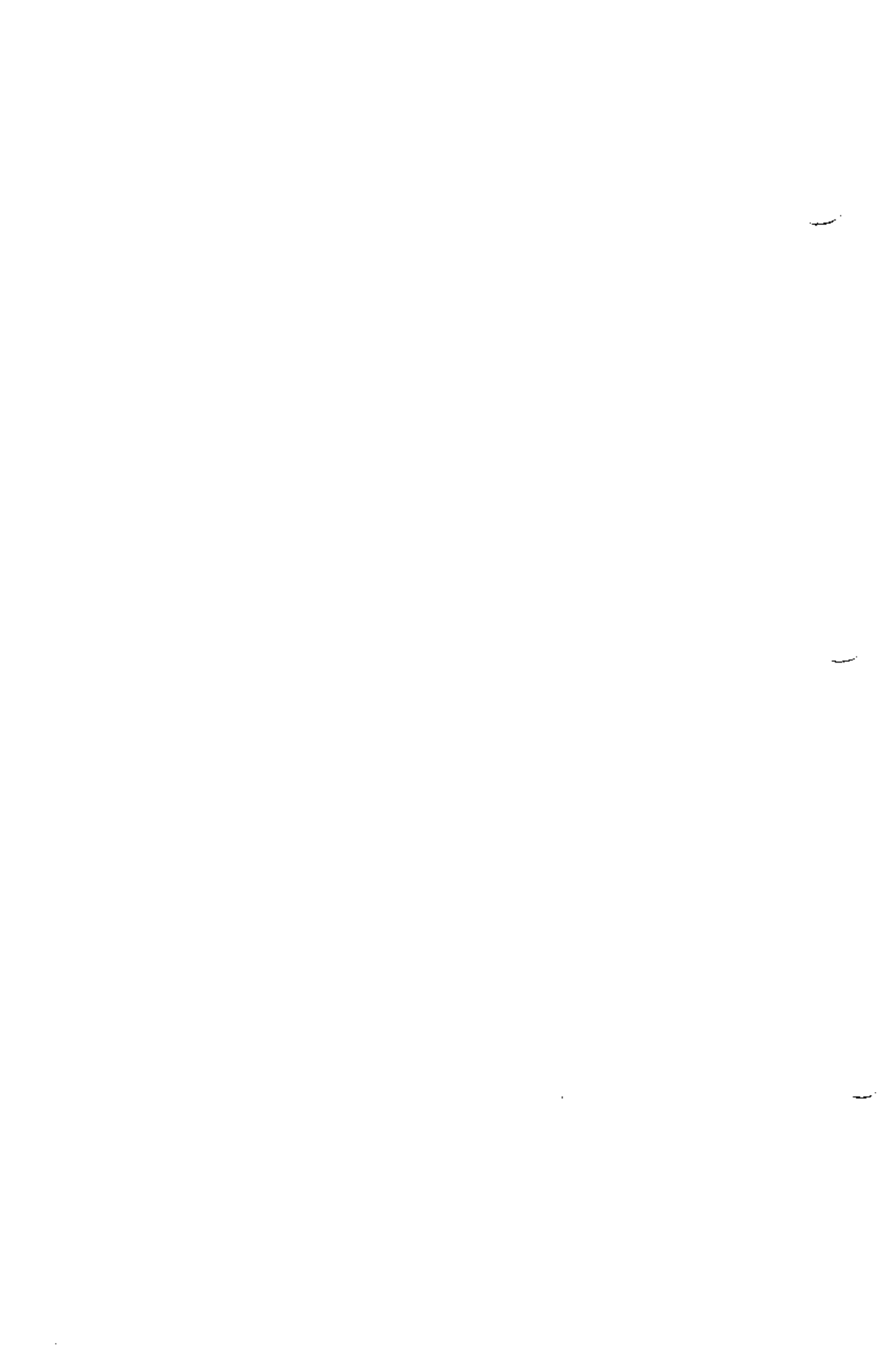
The rationale for the choices for the scales of the results of arithmetic is that in no case should any significant digits be thrown away if, on appearances, the user actually wanted them. Thus, if the user wants to add the numbers 1.5 and 3.517, it seemed reasonable to give them the result 5.017 without requiring to unnecessarily specify rather obvious requirements for precision.

On the other hand, multiplication and exponentiation produce results with many more digits than their operands. It seemed reasonable to give as a minimum the number of decimal places in the operands but not to give more than that number of digits unless the user asked for them by specifying a value for **scale**.

Square root can be handled in just the same way as multiplication.

Division gives arbitrarily many decimal places, and there is simply no way to guess how many places the user wants. In this case only, the user must specify a **scale** to get any decimal places at all.

The scale of remainder was chosen to make it possible to recreate the dividend from the quotient and remainder. This is easy to implement -- no digits are thrown away.



Chapter 7: M4 MACROS

CONTENTS

1. Introduction	1
2. Usage	2
3. Built-In Macros	2
3.1 Arithmetic Built-in Macros	4
4. Defining Macros	5
4.1 define()	5
4.2 Quoting	7
4.3 changequote()	8
4.4 undefine()	9
4.5 ifdef()	9
4.6 Arguments	10
5. File Manipulation	11
5.1 include() and sinclude()	11
5.2 divert(), undivert() and divnum	11
6. String Manipulation	12
6.1 len()	12
6.2 substr()	13
6.3 index() and translit()	13
6.4 dnl	14
6.5 ifelse()	14
7. Executing System Commands	15
7.1 syscmd() and maketemp	15
8. Printing	16
8.1 errprint()	16
8.2 dumpdef	16

1.

2.

3.

Chapter 7

M4 –

MACRO PROCESSOR

1. Introduction

The M4 macro processor is a front end for rational Fortran (Ratfor) and the C programming languages. The `#define` statement in C language and the analogous `define` in Ratfor are examples of the basic facility provided by any macro processor.

The basic operation of M4 is to read every alphanumeric token (string of letters and digits) input and determine if the token is the name of a macro. The name of the macro is replaced by its defining text, and the resulting string is pushed back onto the input to be rescanned.

Macros may be called with arguments. The arguments are collected and substituted into the right places in the defining text before the defining text is rescanned.

Besides the straightforward replacement of one string of text by another, the M4 macro processor provides the following features:

- Arguments
- Arithmetic capabilities
- File manipulation
- Conditional macro expansion
- String and substring functions

The M4 macro processor accepts user-defined macros, as well as its "built-in" macros. Both types of macros work exactly the same way except that some of the built-in macros have side effects on the state of the process.

M4 MACROS

2. Usage

At the beginning of a program, a symbolic name or symbolic constant can be defined as a particular string of characters. The M4 compiler will replace later unquoted occurrences of the symbolic name with the corresponding string.

To run the M4 compiler, give the command:

m4 [optional files]

Each argument file is processed in order. If there are no arguments or if an argument is “—”, the standard input is read at that point.

The processed text is written on the standard output which may be captured for subsequent processing with the following input:

m4 [files] >outputfile

3. Built-In Macros

The following are the 31 M4 built-in macros:

changequote	Restores original characters or makes new quote characters the left and right brackets.
changescom	Changes left and right comment markers from the default # and newline.
deer	Returns the value of its argument decremented by 1.
define	Defines new macros.
defn	Returns the quoted definition of its argument(s).
divert	Diverts output to 1-out-of-10 diversions.
divnum	Returns the number of the currently active diversion.
dnl	Reads and discards characters up to and including the next newline.

dumpdef	Dumps the current names and definitions of items named as arguments.
errprint	Prints its arguments on the standard error file.
eval	Prints arbitrary arithmetic on integers.
ifdef	Determines if a macro is currently defined.
ifndef	Performs arbitrary conditional testing.
include	Returns the contents of the file named in the argument. A fatal error occurs if the file name cannot be accessed.
incr	Returns the value of its argument incremented by 1.
index	Returns the position where the second argument begins in the first argument of index.
len	Returns the number of characters that makes its argument.
m4exit	Causes immediate exit from M4.
m4wrap	Pushes the exit code back at final EOF.
maketemp	Facilitates making unique file names.
popdef	Removes current definition of its argument(s) exposing any previous definitions.
pushdef	Defines new macros but saves any previous definition.
shift	Returns all arguments of shift except the first argument.
sinclude	Returns the contents of the file named in the arguments. The macro remains silent and continues if the file is inaccessible.
substr	Produces substrings of strings.
syscmd	Executes the UNIX System command given in the first argument.

M4 MACROS

traceoff	Turns macro trace off.
traceon	Turns the macro trace on.
translit	Performs character transliteration.
undefine	Removes user-defined or built-in macro definitions.
undivert	Discards the diverted text.

3.1 Arithmetic Built-in Macros

The M4 provides three built-in functions for doing arithmetic on integers (only).

1. **incr**
2. **decr**
3. **eval**

The simplest is **incr** which increments its numeric argument by 1. The built-in **decr** decrements by 1. Thus to handle the common programming situation where a variable is to be defined as "one more than N ," use the following:

```
define(N, 100)
define(N1, 'incr(N)')
```

Then **N1** is defined as one more than the current value of **N**.

The more general mechanism for arithmetic is a built-in called **eval** which is capable of arbitrary arithmetic on integers. The operators in decreasing order of precedence are

```

unary + and -
** or ^ (exponentiation)
* / % (modulus)
+ -
== != < <= > >=
! (not)
& or && (logical AND)
| or || (logical OR).

```

Parentheses may be used to group operations where needed. All the operands of an expression given to `eval` must ultimately be numeric. The numeric value of a true relation (like `1>0`) is 1 and false is 0. The precision in `eval` is 32 bits under the UNIX operating system.

As a simple example, define `M` to be `2==N+1` using `eval` as follows:

```

define(N, 3)
define(M, 'eval(2==N+1)')

```

The defining text for a macro should be quoted unless the text is very simple. Quoting the defining text usually gives the desired result and is a good habit to get into.

4. Defining Macros

4.1 `define()`

The primary built-in function of M4 is `define`. `Define` is used to define new macros. The following input:

```
define(name, stuff)
```

M4 MACROS

causes the string *name* to be defined as *stuff*. All subsequent occurrences of *name* will be replaced by *stuff*. *Name* must be alphanumeric and must begin with a letter (the underscore counts as a letter). *Stuff* is any text that contains balanced parentheses.

Use of a slash may stretch *stuff* over multiple lines. The following is a typical example of the use of **define**, in which *N* is defined to be 100 and is then used in a later **if** statement:

```
define(N, 100)
```

```
...  
if (i > N)
```

The left parenthesis must immediately follow the word **define** to signal that **define** has arguments.

If a user-defined macro or built-in name is not followed immediately by "(", it is assumed to have no arguments.

Macro calls have the following general form:

```
name(arg1,arg2,...argn)
```

A macro name is only recognized as such if it appears surrounded by nonalphanumerics.

In the following example the variable *NNN* is absolutely unrelated to the defined macro *N* even though the variable contains a lot of *N*s:

```
define(N, 100)
```

```
...  
if (NNN > 100)
```

Macros may be defined in terms of other names. For example, the following example defines both *M* and *N* to be 100. If *N* is redefined and subsequently changes, *M* retains the value of 100

not *N*.

```
define(N, 100)
define(M, N)
```

The M4 macro processor expands macro names into their defining text as soon as possible. The string *N* is immediately replaced by 100. Then the string *M* is also immediately replaced by 100. The overall result is the same as using the following input in the first place:

```
define(M, 100)
```

The order of the definitions can be interchanged as follows:

```
define(M, N)
define(N, 100)
```

Now *M* is defined to be the string *N*, so when the value of *M* is requested later, the result is the value of *N* at that time (because the *M* will be replaced by *N* which will be replaced by 100).

4.2 Quoting

The more general solution to delay the expansion of the arguments of **define** by **quoting** them.

Any text surrounded by left and right single quotes is not expanded immediately but has the quotes stripped off. The value of a quoted string is the string stripped of the quotes. If the input is

```
define(N, 100)
define(M, 'N')
```

the quotes around the *N* are stripped off as the argument is being collected. The results of using quotes is to define *M* as the string *N*, not 100.

M4 MACROS

The general rule is that M4 always strips off one level of single quotes whenever it evaluates something. This is true even outside of macros.

If the word **define** is to appear in the output, the word must be quoted in the input as follows:

```
'define' = 1;
```

Another example of using quotes is redefining *N*. To redefine *N*, the evaluation must be delayed by quoting

```
define(N, 100)
...
define('N', 200)
```

In M4, it is often wise to quote the first argument of a macro. The following example will not redefine *N*:

```
define(N, 100)
...
define(N, 200)
```

The *N* in the second definition is replaced by 100. The result is equivalent to the following statement:

```
define(100, 200)
```

This statement is ignored by M4 since only things that look like names can be defined.

4.3 **changequote()**

If left and right single quotes are not convenient for some reason, the quote characters can be changed with the following built-in macro:

```
changequote(l, r)
```

The built-in **changequote** makes the new quote characters the left and right brackets. The original characters can be restored by using **changequote** without arguments as follows:

changequote

4.4 **undefine()**

There are two additional built-ins related to **define**. The **undefine** macro removes the definition of some macro or built-in as follows:

undefine('N')

The macro removes the definition of *N*. Built-ins can be removed with **undefine**, as follows:

undefine('define')

But once removed, the definition cannot be reused.

4.5 **ifdef()**

The built-in **ifdef** provides a way to determine if a macro is currently defined.

Depending on the system, a definition appropriate for the particular machine can be made as follows:

```
ifdef('pdp11', 'define(wordsize,16)')  
ifdef('u3b', 'define(wordsize,32)')
```

Remember to use the quotes.

The **ifdef** macro actually permits three arguments. If the first argument is defined, the value of **ifdef** is the second argument.

If the first argument is not defined, the value of **ifdef** is the third argument.

M4 MACROS

If there is no third argument, the value of **ifdef** is null.

If the name is undefined, the value of **ifdef** is then the third argument, as in

```
ifdef('unix', on UNIX, not on UNIX)
```

4.6 Arguments

User-defined macros may also have arguments, so different invocations can have different results. Within the replacement text for a macro (the second argument of its **define**), any occurrence of **\$n** is replaced by the **n**th argument when the macro is actually used. Thus, the following macro, **bump**, generates code to increment its argument by 1:

```
define(bump, $1 = $1 + 1)
```

The **'bump(x)'** statement is equivalent to **'x = x + 1'**.

A macro can have as many arguments as needed, but only the first nine are accessible (**\$1** through **\$9**). The macro name is **\$0** although that is less commonly used. Arguments that are not supplied are replaced by null strings, so a macro can be defined which simply concatenates its arguments like this:

```
define(cat, $1$2$3$4$5$6$7$8$9)
```

Thus, **'cat(x, y, z)'** is equivalent to **'xyz'**. Arguments **\$4** through **\$9** are null since no corresponding arguments were provided. Leading unquoted blanks, tabs or newlines that occur during argument collection are discarded. All other white space is retained. Thus:

```
define(a, b c)
```

defines **'a'** to be **'b c'**.

Arguments are separated by commas; however, when commas are within parentheses, the argument is not terminated nor separated. For example,

```
define(a, (b,c))
```

has only two arguments. The first argument is **a**. The second is literally **(b,c)**. A bare comma or parenthesis can be inserted by quoting it.

5. File Manipulation

5.1 include() and sinclude()

A new file can be included in the input at any time by the built-in function **include**. For example,

```
include(filename)
```

inserts the contents of *filename* in place of the **include** command. The contents of the file is often a set of definitions. The value of **include** (**include**'s replacement text) is the contents of the file. If needed, the contents can be captured in definitions, etc.

A fatal error occurs if the file named in **include** cannot be accessed. To get some control over this situation, the alternate form **sinclude** can be used. The built-in **sinclude** (silent include) says nothing and continues if the file named cannot be accessed.

5.2 divert(), undivert() and divnum

The output of M4 can be diverted to temporary files during processing, and the collected material can be output upon command. The M4 maintains nine of these diversions, numbered 1 through 9. If the built-in macro

```
divert(n)
```

M4 MACROS

is used, all subsequent output is put onto the end of a temporary file referred to as *u*. Diverting to this file is stopped by the **divert** or **divert(0)** command which resumes the normal output process.

Diverted text is normally output all at once at the end of processing with the diversions output in numerical order. Diversions can be brought back at any time by appending the new diversion to the current diversion. Output diverted to a stream other than 0 through 9 is discarded.

The built-in **undivert** brings back all diversions in numerical order. The built-in **undivert** with arguments brings back the selected diversions in the order given. The act of undiverting discards the diverted text (as does diverting) into a diversion whose number is not between 0 and 9, inclusive.

The value of **undivert** is *not* the diverted text. Furthermore, the diverted material is *not* rescanned for macros.

The built-in **divnum** returns the number of the currently active diversion. The current output stream is zero during normal processing.

6. String Manipulation

6.1 **len()**

The built-in **len** returns the length of the string (number of characters) that makes up its argument. Thus:

len(abcdef)

is 6, and **len((a,b))** is 5 (the parentheses and comma are counted along with a and b).

6.2 substr()

The built-in **substr** can be used to produce substrings of strings. Using input, **substr(s, i, n)** returns the substring of *s* that starts at the *i*th position (origin zero) and is *n* characters long. If *n* is omitted, the rest of the string is returned. Inputting

```
substr('now is the time',1)
```

returns the following string:

```
ow is the time.
```

If *i* or *n* are out of range, various actions occur.

6.3 index() and translit()

The built-in **index(s1, s2)** returns the index (position) in *s1* where the string *s2* occurs or -1 if it does not occur. As with **substr**, the origin for strings is 0.

The built-in **translit** performs character transliteration and has the general form

```
translit(s, f, t)
```

which modifies *s* by replacing any character found in *f* by the corresponding character of *t*. Using input

```
translit(s, aeiou, 12345)
```

replaces the vowels by the corresponding digits. If *t* is shorter than *f*, characters that do not have an entry in *t* are deleted. As a limiting case, if *t* is not present at all, characters from *f* are deleted from *s*. So

```
translit(s, aeiou)
```

would delete vowels from *s*.

M4 MACROS

6.4 `dnl`

There is a built-in macro called `dnl` that deletes all characters that follow it up to and including the next new line. The `dnl` macro is useful mainly for throwing away empty lines that otherwise tend to clutter up M4 output. Using input

```
define(N, 100)
define(M, 200)
define(L, 300)
```

results in a new line at the end of each line that is not part of the definition. The new line is copied into the output so that each `define` statement is followed by a blank line. If the built-in macro `dnl` is added to each of these lines, the newlines will disappear.

```
define(N, 100)dnl
define(M, 200)dnl
define(L, 300)dnl
```

Another method of achieving the same results is to input

```
divert(-1)
define(N, 100)
define(M, 200)
define(L, 300)
divert
```

6.5 `ifelse()`

Arbitrary conditional testing is performed via built-in `ifelse`. In the simplest form

```
ifelse(a, b, c, d)
```

compares the two strings *a* and *b*. If *a* and *b* are identical, `ifelse` returns the string *c*. Otherwise, string *d* is returned. Thus, a macro called `compare` can be defined as one which compares two strings and returns "yes" or "no" if they are the same or different as follows:

```
define(compare, 'ifelse($1, $2, yes, no)')
```

Note the quotes which prevents evaluation of **ifelse** occurring too early. If the fourth argument is missing, it is treated as empty.

The built-in **ifelse** can actually have any number of arguments and provides a limited form of multiway decision capability. In the input

```
ifelse(a, b, c, d, e, f, g)
```

if the string *a* matches the string *b*, the result is *c*. Otherwise, if *d* is the same as *e*, the result is *f*. Otherwise, the result is *g*. If the final argument is omitted, the result is null, so

```
ifelse(a, b, c)
```

is *c* if *a* matches *b*, and null otherwise.

7. Executing System Commands

7.1 **syscmd()** and **maketemp**

Any program in the local operating system can be run by using the **syscmd** built-in. For example,

```
syscmd(date)
```

on the UNIX system runs the **date** command. Normally, **syscmd** would be used to create a file for a subsequent **include**.

To facilitate making unique file names, the built-in **maketemp** is provided with specifications identical to the system function *mktemp*. The **maketemp** macro fills in a string of XXXXX in the argument with the process id of the current process.

M4 MACROS

8. Printing

8.1 `errprint()`

The built-in **errprint** writes its arguments out on the standard error file. An example would be

```
errprint('fatal error')
```

8.2 `dumpdef`

The built-in **dumpdef** is a debugging aid that dumps the current names and definitions of items named as arguments. If no arguments are given, then all current names and definitions are printed. Do not forget to quote the names.

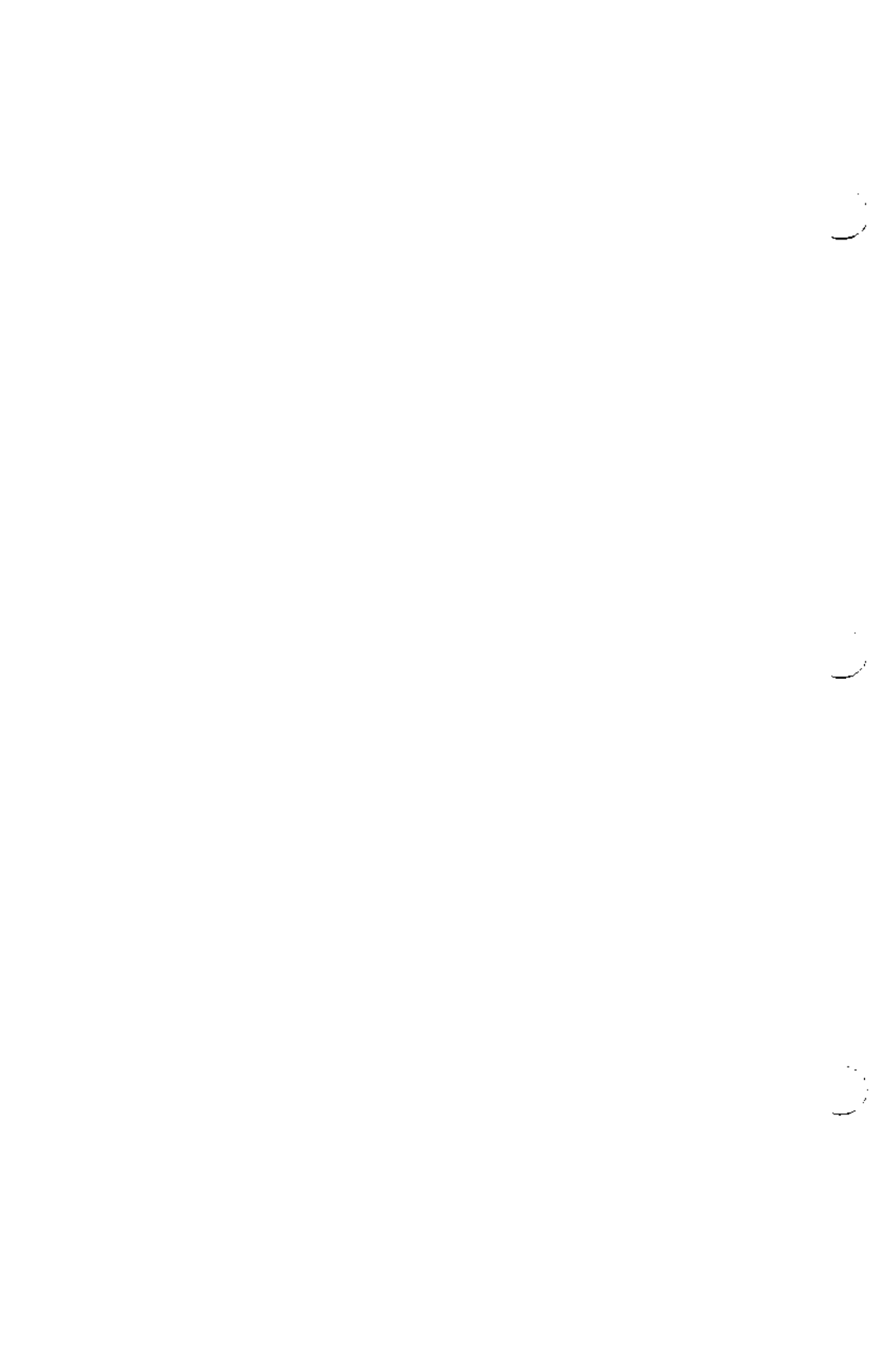
Chapter 8: MAKE

CONTENTS

1. Introduction	1
2. Basic Operation	3
3. Usage	13
4. Substitutions	18
4.1 \$@, \$?, \$< and \$*	20
4.2 .DEFAULT and .PRECIOUS	21
5. Command Usage	22
6. Suffixes and Transformation	24
7. Implicit Rules	27
8. Suggestions and Warnings	29

LIST OF FIGURES

Figure 8.1. Default MAKE Suffix List	27
--	----



Chapter 8

MAKE –

MAINTAINING COMPUTER PROGRAMS

1. Introduction

It is a common practice is to divide large programs into smaller pieces that are more manageable individually, but which create a complicated process to update and maintain.

The pieces may require several different treatments, or have to be compiled with special options, definitions and declarations. The resulting code may need further transformation by loading the code with certain libraries under control of special options.

Another activity that complicates program development is a long editing session. A programmer may lose track of the files changed and the object modules still valid especially when a change to a declaration can make a dozen other files obsolete.

The programmer must also remember to compile a routine that has been changed or that uses changed declarations.

The **make** program is a software tool that maintains, updates, and regenerates groups of computer programs.

The **make** program was written to facilitate the process of updating programs by maintaining information on the following:

- Files that are dependent upon other files
- Files that were modified recently

MAKE

- Files that need to be reprocessed or recompiled after a change in the source
- The exact sequence of operations needed to make and exercise a new version of the program

The **make** program keeps track of program file dependencies, and whenever a change is made in any part of a program, the **make** command creates the proper files simply, correctly, and with a minimum amount of effort.

The **make** program also provides a simple macro substitution facility and the ability to encapsulate commands in a single file for convenient administration.

2. Basic Operation

The basic operation of **make** is:

- Find the name of the needed target file in the description.
- Ensure that all of the files on which it depends exist and are up to date.
- Create the target file if it has not been updated since its generators were modified.

A descriptor file is used to define the graph of dependencies. The **make** program determines the work necessary to update a given program by performing a depth-first search of this list of dependencies. The descriptor file **make** expects is called a **makefile**, often easy to write and only infrequently changed.

Once the interfile dependencies and command sequences have been included in such a file, the command:

make

is often sufficient to update the appropriate files — regardless of the number of files edited since the last **make**. Obviously, then, the operation of **make** is highly dependent on the ability to find the date and time that a file was last modified.

As an example of the use of **make**, the following is the description file used to maintain the **sed** command:

MAKE

Description file for the sed command:

```
ROOT =
OL = $(ROOT)/
SL = $(ROOT)/usr/src/cmd
RDIR = $(SL)/sed
INS = :
REL = current
CSID = -r`gsid sed $(REL)`
MKSID = -r`gsid sed.mk $(REL)`
LIST = lp
INSDIR = $(OL)bin
IFLAG = -n
B10 =
CFLAGS = -O $(B10)
LDFLAGS = -s $(IFLAG)
SOURCE = sed.h sed0.c sed1.c
FILES = sed0.o sed1.o
MAKE = make
compile all: sed
    :

sed:    $(FILES)
        $(CC) $(LDFLAGS) -o sed $(FILES)
        $(INS) $(INSDIR) sed

$(FILES):: sed.h
    :

install:
    $(MAKE) -f sed.mk INS="install -f" OL=$(OL)

build:  bldmk
        get -p $(CSID) s.sed.src $(REWIRE) | ntar -d $(RDIR) -g

bldmk:  ; get -p $(MKSID) s.sed.mk > $(RDIR)/sed.mk
listing:
        pr sed.mk $(SOURCE) | $(LIST)
```

```

listmk: ; pr sed.mk | $(LIST)
edit:
    get -e -p s.sed.src | ntar -g
delta:
    ntar -p $(SOURCE) > sed.src
    delta s.sed.src
    rm -f $(SOURCE)

mkedit: ; get -e s.sed.mk
mkdelta: ; delta s.sed.mk
clean:
    rm -f $(FILES)

clobber: clean
    rm -f sed

delete: clobber
    rm -f $(SOURCE)

```

Assuming that the source for **sed** and the **makefile**, called **sed.mk**, are both in the current directory, giving the command:

```
make -n -p -d -f sed.mk > make.out
```

will cause **make** to do the following:

- Print the commands it would have executed, without actually executing them (**-n**).
- Print the complete set of macro definitions and target descriptions (**-p**).
- Print out detailed information on files and times examined (**-d**) for debugging purposes.
- Use the file **sed.mk** as the descriptor file (**-f**).

Using this example, the following is a large excerpt of the output saved in the file **make.out**:

MAKE

```
doname(sed,1)
doname(sed0.o,2)
doname(sed.h,3)
TIME(sed.h) = 469758551
doname(sed0.c,3)
TIME(sed0.c) = 469758582
TIME(sed0.o) = 475912131
doname(sed1.o,2)
doname(sed.h,3)
TIME(sed.h) = 469758551
doname(sed1.c,3)
TIME(sed1.c) = 469758590
TIME(sed1.o) = 475912301
cc -s -n -o sed sed0.o sed1.o
: /bin sed
TIME(sed) = 483750638
:
Open directories:
3: .
Macros:
? = sed
@ =
< = sed1.c
* = sed1
MAKE = make
FILES = sed0.o sed1.o
SOURCE = sed.h sed0.c sed1.c
LDFLAGS = -s $(IFLAG)
B10 =
IFLAG = -n
INSDIR = $(OL)bin
LIST = lp
MKSID = -r`gsid sed.mk $(REL)`
CSID = -r`gsid sed $(REL)`
REL = current
INS = :
RDIR = $(SL)/sed
SL = $(ROOT)/usr/src/cmd
```

```

OL = $(ROOT)/
ROOT =
LOADLIBES =
FFLAGS =
EFLAGS =
FC = f77
RFLAGS =
RC = f77
CFLAGS = -O $(B10)
PFLAGS =
PC = pc
AS = as
CC = cc
LFLAGS =
LEX = lex
YFLAGS =
YACCE = yacc -e
YACCR = yacc -r
YACC = yacc
$ = $

sed1.c done=2
sed0.c done=2
delete: done=0
    depends on: clobber
    commands:
        rm -f $(SOURCE)
clobber: done=0
    depends on: clean
    commands:
        rm -f sed
clean: done=0
    commands:
        rm -f $(FILES)
mkdelta: done=0
    commands:
        delta s.sed.mk
mkedit: done=0
    commands:

```

MAKE

```
    get -e s.sed.mk
delta: done=0
  commands:
    ntar -p $(SOURCE) > sed.src
    delta s.sed.src
    rm -f $(SOURCE)
edit: done=0
  commands:
    get -e -p s.sed.src | ntar -g
listmk: done=0
  commands:
    pr sed.mk | $(LIST)
listing: done=0
  commands:
    pr sed.mk $(SOURCE) | $(LIST)
bldmk: done=0
  commands:
    get -p $(MKSID) s.sed.mk > $(RDIR)/sed.mk
build: done=0
  depends on: bldmk
  commands:
    get -p $(CSID) s.sed.src $(REWIRE) | \
      ntar -d $(RDIR) -g
install: done=0
  commands:
    $(MAKE) -f sed.mk INS="install -f" OL=$(OL)
sed.h done=2
sed1.o: done=2
  depends on: sed.h
  commands:
    :
sed0.o: done=2
  depends on: sed.h
  commands:
    :
sed: done=2
  depends on: sed0.o sed1.o
  commands:
```

```

$(CC) $(LDFLAGS) -o sed $(FILES)
$(INS) $(INSDIR) sed
all: done=0
    depends on: sed
    commands:
        :
compile: done=2 (MAIN NAME)
    depends on: sed
    commands:
        :
.l.out: done=0
    commands:
        $(LEX) $(LFLAGS) $<
        $(CC) $(CFLAGS) lex.yy.c $(LOADLIBES) -ll -o $@
        rm lex.yy.c
.y.out: done=0
    commands:
        $(YACC) $(YFLAGS) $<
        $(CC) $(CFLAGS) y.tab.c $(LOADLIBES) -ly -o $@
        rm y.tab.c
.e.out: done=0
    commands:
        $(FC) $(EFLAGS) $(RFLAGS) $(FFLAGS) $< \
            $(LOADLIBES) -o $@
        -rm $*.o
.r.out: done=0
    commands:
        $(FC) $(EFLAGS) $(RFLAGS) $(FFLAGS) $< \
            $(LOADLIBES) -o $@
        -rm $*.o
.f.out: done=0
    commands:
        $(FC) $(EFLAGS) $(RFLAGS) $(FFLAGS) $< \
            $(LOADLIBES) -o $@
        -rm $*.o
.o.out: done=0
    commands:
        $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@

```

MAKE

```
.c.out: done=0
commands:
    $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
.s.out: done=0
commands:
    $(CC) $(CFLAGS) $< $(LOADLIBES) -o $@
.ye.e: done=0
commands:
    $(YACCE) $(YFLAGS) $<
    mv y.tab.e $@
.yr.r: done=0
commands:
    $(YACCR) $(YFLAGS) $<
    mv y.tab.r $@
.l.c: done=0
commands:
    $(LEX) $(LFLAGS) $<
    mv lex.yy.c $@
.y.c: done=0
commands:
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
.l.o: done=0
commands:
    $(LEX) $(LFLAGS) $<
    $(CC) $(CFLAGS) -c lex.yy.c
    rm lex.yy.c
    mv lex.yy.o $@
.ye.o: done=0
commands:
    $(YACCE) $(YFLAGS) $<
    $(EC) $(RFLAGS) -c y.tab.e
    rm y.tab.e
    mv y.tab.o $@
.yr.o: done=0
commands:
    $(YACCR) $(YFLAGS) $<
    $(RC) $(RFLAGS) -c y.tab.r
```

```

rm y.tab.r
mv y.tab.o $@
.y.o: done=0
commands:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.s.o: done=0
commands:
    $(AS) -o $@ $<
.f.o: done=0
commands:
    $(FC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.r.o: done=0
commands:
    $(FC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.e.o: done=0
commands:
    $(FC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.cl.o: done=0
commands:
    class -c $<
.p.o: done=0
commands:
    $(PC) $(PFLAGS) -c $<
.c.i: done=0
commands:
    $(CC) $(CFLAGS) -P $<
.c.s: done=0
commands:
    $(CC) $(CFLAGS) -S $<
.c.o: done=0
commands:
    $(CC) $(CFLAGS) -c $<
.i    done=0
.p    done=0
.cl   done=0

```

MAKE

```
.s    done=0
.l    done=0
.ye   done=0
.yr   done=0
.y    done=0
.r    done=0
.e    done=0
.f    done=0
.c    done=0
.o    done=0
.out  done=0
.SUFFIXES:  done=0
  depends on: .out .o .c .f .e .r .y
              .yr .ye .l .s .cl .p .i
```

Although none of the source files or grammars were mentioned by name in the description file, **make** found them using its suffix rules and issued the necessary commands.

The printed output could have been sent to a file by changing the definition of the **LIST** macro on the command line to the following:

```
make listing "LIST = cat >zap"
```

Or, this macro could be changed in the **makefile** to always direct the output of **make** to the file **zap**.

3. Usage

The basic operation of **make** is to accomplish the following:

- Update a target file by ensuring that all of the files on which the target file depends exist and are up to date.
- Create the target file if it has not been modified since the dependents were modified.
- Perform a depth-first search of the graph of dependencies.

To illustrate, consider a simple example in which a program named **prog** is made by compiling and loading three C language files **x.c**, **y.c** and **z.c** with the **IS** library. By convention, the output of the C language compilations will be found in files named **x.o**, **y.o** and **z.o**.

Assume that the files **x.c** and **y.c** share some declarations in a file named **defs**, but that **z.c** does not. That is, **x.c** and **y.c** have the line

```
#include "defs"
```

while **z.c** does not.

The following text describes the relationships and operations:

```
prog :    x.o y.o z.o
        cc x.o y.o z.o  -lS  -o prog
```

```
x.o y.o : defs
```

If this information were stored in a file named **makefile**, the command:

```
make
```

would perform the operations needed to recreate **prog** after any changes had been made to any of the four source files **x.c**, **y.c**, **z.c** or **defs**. Even the **-f** option is unnecessary if the

MAKE

descriptor file is named **makefile**.

The **make** program operates using the following three sources of information:

- A user-supplied description file
- File names and "last-modified" times from the file system
- Built-in rules to bridge some of the gaps

In the example, the first line:

```
prog : x.o y.o z.o
```

is understood by **make** to mean that **prog** depends on three ".o" files.

Once these object files are current, the second line:

```
prog : x.o y.o z.o
      cc x.o y.o z.o -lS -o prog
```

describes how to load them to create **prog**.

The third line

```
prog :      x.o y.o z.o
           cc x.o y.o z.o -lS -o prog
x.o y.o : defs
```

states that **x.o** and **y.o** depend on the file **defs**.

From the file system, **make** discovers that there are three ".c" files corresponding to the needed ".o" files and uses built-in information on how to generate an object from a source file (i.e., issue a "cc -c" command).

By not taking advantage of **make**'s innate knowledge, the following longer descriptive file results.

```

prog :   x.o y.o z.o
        cc x.o y.o z.o -ls -o prog

x.o :   x.c defs
        cc -c x.c

y.o :   y.c defs
        cc -c y.c

z.o :   z.c
        cc -c z.c

```

If none of the source or object files have changed since the last time **prog** was made, all of the files are current, and the command

make

announces this fact and stops.

If, however, the **defs** file has been edited, **x.c** and **y.c** (but not **z.c**) are recompiled. Then **prog** is created from the new ".o" files.

If only the file **y.c** had changed, only it is recompiled, but it is still necessary to reload **prog**.

If no target name is given on the **make** command line, the first target mentioned in the description is created. Otherwise, the specified targets are made.

The command

make x.o

would recompile **x.o** if **x.c** or **defs** had changed.

If the file exists after the commands are executed, the file's time of last modification is used in further decisions.

MAKE

If the file does not exist after the commands are executed, the current time is used in making further decisions. A useful method is to include rules with mnemonic names and commands that do not actually produce a file with that name. These entries can take advantage of **make**'s ability to generate files and substitute macros. For example: "save" might be included to copy a certain set of files, or an entry "cleanup" might be used to throw away unneeded intermediate files.

It is also possible to maintain a zero-length file purely to keep track of the time at which certain actions were performed. For example:

```
print: $(FILES)  
pr $? | $P  
touch print
```

The "print" entry prints only the files changed since the last **make print** command. A zero-length file **print** is maintained to keep track of the time of the printing, the time since the file **print** was last **touched**.

The **\$?** macro in the command line then picks up only the names of the files changed since **print** was touched.

The **make** program has a simple "macro" mechanism for substituting in **dependency** lines and command strings. Macros are defined by command arguments or description file lines with embedded equal signs.

A macro is invoked by preceding the name by a dollar sign. The name of the macro is either the single character after the dollar sign or a name inside parentheses. (Macro names longer than one character must be parenthesized.)

The following are valid macro invocations:

\$(CFLAGS)**\$2****\$(xy)****\$Z****\$(Z)**

The last two invocations are identical. A **\$\$** is a dollar sign.

The **\$***, **\$@**, **\$?**, and **\$<** are four special macros which change values during the execution of the command.

The following fragment shows assignment and use of some macros:

```
OBJECTS = x.o y.o z.o
LIBES = -IS
prog: $(OBJECTS)
cc $(OBJECTS) $(LIBES) -o prog
...
```

Using the example illustrated above, the **make** command loads the three object files with the **IS** library. The command:

```
make "LIBES= -ll -IS"
```

would load them with both the **LEX (-ll)** and the standard **(-IS)** libraries. Macro definitions on the command line override definitions in the description file.

NOTE: Remember to **QUOTE** arguments with embedded blanks in UNIX Operating System commands.

MAKE

4. Substitutions

A description file contains the following information:

- Macro definitions
- Dependency information
- Executable commands

Comments are introduced with a hash symbol (#) and all characters on the same line after a hash symbol are ignored. Blank lines and lines beginning with a hash symbol (#) are totally ignored.

If a non-comment line is too long, the line can be continued by using a backslash. If the last character of a line is a backslash, **make** replaces the backslash, the newline and all following blanks and tabs with a single blank.

A macro definition is a line containing an equal sign NOT preceded by a colon or a tab. The macro name is composed of a string of letters and digits to the left of the equal sign (trailing blanks and tabs are stripped). The macro name is assigned the value of the string of characters following the equal sign (leading blanks and tabs are stripped).

The following are valid macro definitions:

```
2 = xyz
abc = -ll -ly -ls
LIBES =
```

The last definition assigns LIBES the NULL string. A macro that is never explicitly defined has the NULL string as the macro's value.

Macro definitions may also appear on the **make** command line.

The general form of an entry in a **make** descriptor file is:

```
target1 [target2 . .] [:] [dependent1 . .] [; commands] [# . .]
[(tab) commands] [# . .]
```

...

- Items inside brackets may be omitted.
- **Targets** and **dependents** are strings of letters, digits, periods, and slashes.
- Shell metacharacters such as “*” and “?” are expanded.
- A **command** is any string of characters not including a sharp (#) except when the sharp is in quotes or not including a newline.
- **Commands** may appear either after a semicolon on a dependency line or on lines beginning with a tab immediately following a **dependency** line.
- A **dependency** line may have either a single or a double colon.
- A **target** name may appear on more than one dependency line, but all of those lines must be of the same (single or double colon) type.

For the usual SINGLE-COLON case, a command sequence may be associated with, at most, ONE **dependency** line.

If the **target** is out-of-date with any of the **dependents** on any of the lines, and a command sequence is specified (even a NULL one following a semicolon or tab), it is executed. Otherwise, a default **creation rule** may be invoked.

In the DOUBLE-COLON case, a command sequence may be associated with each dependency line. If the target is out of

MAKE

date with any of the files on a particular line, the associated commands are executed. A built-in rule may also be executed. If a **target** must be created, the sequence of commands is executed. This detailed form is of particular value in updating archive-type files.

Normally, each command line is printed and then passed to a separate invocation of the shell after substituting for macros. The printing is suppressed in the **silent mode** or if the command line begins with an @ sign. For example:

```
@size make /usr/bin/make
```

If the command line above were in a descriptor file, the printing of the command line itself would be suppressed by the @ sign, but the output of the command would be printed.

The **make** program normally stops if any command signals an error by returning a nonzero error code. Errors are ignored by the following actions:

- Use of the **-i** flags on the **make** command line
- Using the "fake" **target** name **".IGNORE"** appears in the description file
- Beginning the command string in the description file with a hyphen

Some commands return meaningless status. Because each command line is passed to a separate invocation of the shell, care must be taken with certain commands (e.g., **cd** and shell control commands) that have meaning only within a single shell process. These results are forgotten before the next line is executed.

4.1 \$@, \$?, \$< and \$*

Before issuing any command, certain internally maintained macros are set.

The **\$@** macro is set to the full **target** name of the current target. The **\$@** macro is evaluated only for explicitly named dependencies.

The **\$?** macro is set to the string of names that were found to be younger than the target. The **\$?** macro is evaluated when explicit rules from the **makefile** are evaluated.

If the command was generated by an implicit rule, the **\$<** macro is the name of the related file that caused the action.

If the command was generated by an implicit rule, the **\$*** macro is the prefix shared by the current and the dependent file names.

4.2 .DEFAULT and .PRECIOUS

If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **".DEFAULT"** are used. If there is no **".DEFAULT,"** **make** prints a message and stops.

If a file or files are assigned as dependent to **.PRECIOUS**, those files will not be removed regardless of any command to the contrary.

MAKE

5. Command Usage

The **make** command takes:

- macro definitions,
- flags,
- description file names,
- target file names as arguments in the form:

make [*flags*] [*macro definitions*] [*targets*]

The following summary of command options explains how these arguments are interpreted:

1. First, all **macro definition** arguments (arguments with embedded equal signs) are analyzed and the assignments made. Command-line macros override corresponding definitions found in the description files.
2. Next, the flag arguments are examined. The permissible flags are as follows:
 - i Ignore error codes returned by invoked commands. This mode is entered if the fake target name **“.IGNORE”** appears in the description file.
 - s Silent mode. DO NOT print command lines before executing. This mode is also entered if the **“fake” target** name **“.SILENT”** appears in the description file.
 - r DO NOT use the built-in rules.
 - n NO EXECUTE mode. Print commands, but do not execute them. Even lines beginning with an **“@”** sign are printed.
 - t Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

- q Question. The **make** command returns a zero or nonzero status code depending on whether the target file is or is not up to date.
 - p Print out the complete set of macro definitions and target descriptions.
 - d DEBUG mode. Print out detailed information on files and times examined.
 - f Description file name. The next argument is assumed to be the name of a description file. A file name of "-" denotes the standard input. If there are no "-f" arguments, the file named **makefile** or **Makefile** in the current directory is read. The contents of the description files override the built-in rules if they are present.
 - b Compatibility mode for old **makefiles**.
 - k Abandon work on the current entry, but continue work on other branches that do not depend upon that entry
 - e Cause environment variables to override assignments within **makefiles**.
 - m Print a memory map showing text, data and stack. DOES NOT operate on systems with a **getu** system call.
3. Finally, the remaining arguments are assumed to be the names of **targets** to be made, and are done in left-to-right order. If there are no remaining arguments, the first name in the description files that does not begin with a period is "**made**."

MAKE

6. Suffixes and Transformation

The **make** program does not know what file name suffixes are interesting or how to transform a file with one suffix into a file with another suffix. This information is stored in an internal table that has the form of a description file. If the **-r** flag is used, the internal table is not used.

The list of suffixes is actually the dependency list for the name **".SUFFIXES"** in the description file. The **make** program searches for a file with any of the suffixes on the list. If such a file exists and there is a transformation rule for that combination, **make** transforms a file with one suffix into a file with another suffix.

The names of the transformation rules are the concatenation of the two suffixes. For example, the name of the rule to transform a **.r** file to a **.o** file is **".r.o"**

If the rule is present and no explicit command sequence has been given in the description file, the command sequence for the rule **.r.o** is used.

If a command is generated by using one of these suffixing rules, the macro **\$*** is given the value of the "root" name of the file to be **"made"** — the "root" name is everything but the suffix. The macro **\$<** is the name of the **dependent** that caused the action.

The order of the suffix list is significant since the list is scanned from left to right. The first name formed that has BOTH a file and a rule associated with it is used.

If new names are to be appended, an entry can be added for **".SUFFIXES"** in the description file. The **dependents** are

added to the usual list.

A ".SUFFIXES" line without any dependents deletes the current list. It is necessary to clear the current list if the order of names is to be changed.

The following is an excerpt from the default rules file (rules.c):

MAKE

```
.SUFFIXES : .o .c .e .r .f .y .yr .ye .l .s
YACC = yacc
YACCR = yacc -r
YACCE = yacc -e
YFLAGS =
LEX = lex
LFLAGS =
CC = cc
AS = as -
CFLAGS =
RC = ec
RFLAGS =
EC = ec
EFLAGS =
FFlags =
.c.o :
    $(CC) $(CFLAGS) -c $<
.e.o .r.o .f.o :
    $(EC) $(RFLAGS) $(EFLAGS) $(FFLAGS) -c $<
.s.o :
    $(AS) -o $@ $<
.y.o :
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.c
    mv y.tab.o $@
.y.c :
    $(YACC) $(YFLAGS) $<
    mv y.tab.c $@
```

7. Implicit Rules

The **make** program uses a table of interesting suffixes and a set of transformation rules to supply default dependency information and implied commands.

The default suffix list is as follows:

DEFAULT SUFFIX LIST	
.o	Object file
.c	C source file
.e	EFL source file
.r	RATFOR source file
.f	FORTRAN source file
.s	Assembler source file [as (1)]
.y	YACC-C source grammar
.yr	YACC-RATFOR source grammar
.ye	YACC-EFL source grammar
.l	LEX source grammar.

Figure 8.1. Default MAKE Suffix List

If there are two paths connecting a pair of suffixes, the longer one is used **ONLY** if the intermediate file exists or is named in the description.

If the file **x.o** were needed, and a file called **x.c** was found in the description or directory, the **x.o** file would be compiled. If there were also an **x.l**, that grammar would be run through **LEX** BEFORE compiling the result.

If the file **x.o** were needed and no **x.c** was found, but an **x.l** existed, **make** would discard the intermediate C language file and use the direct link.

MAKE

It is possible to change the names of some of the compilers used in the default or the flag arguments with which they are invoked by knowing the macro names used.

The compiler names are the macros **AS**, **CC**, **RC**, **EC**, **YACC**, **YACCR**, **YACCE** and **LEX**.

The command

make CC=newcc

will cause the **newcc** command to be used instead of the usual C language compiler.

The macros **CFLAGS**, **RFLAGS**, **EFLAGS**, **YFLAGS** and **LFLAGS** may be set to cause these commands to be issued with optional flags.

For example:

make "CFLAGS=-O"

causes the C language optimizer to be used.

8. Suggestions and Warnings

The most common difficulties arise from **make**'s specific meaning of **dependency**. For example, if file **x.c** contains the line `"#include "defs",` the object file **x.o** depends on **defs**. However, the source file **x.c** does NOT depend on **defs**. If **defs** is changed, nothing is done to the file **x.c**, but file **x.o** must be recreated.

To discover what **make** would do, the **-n** option is very useful. For example:

```
make -n
```

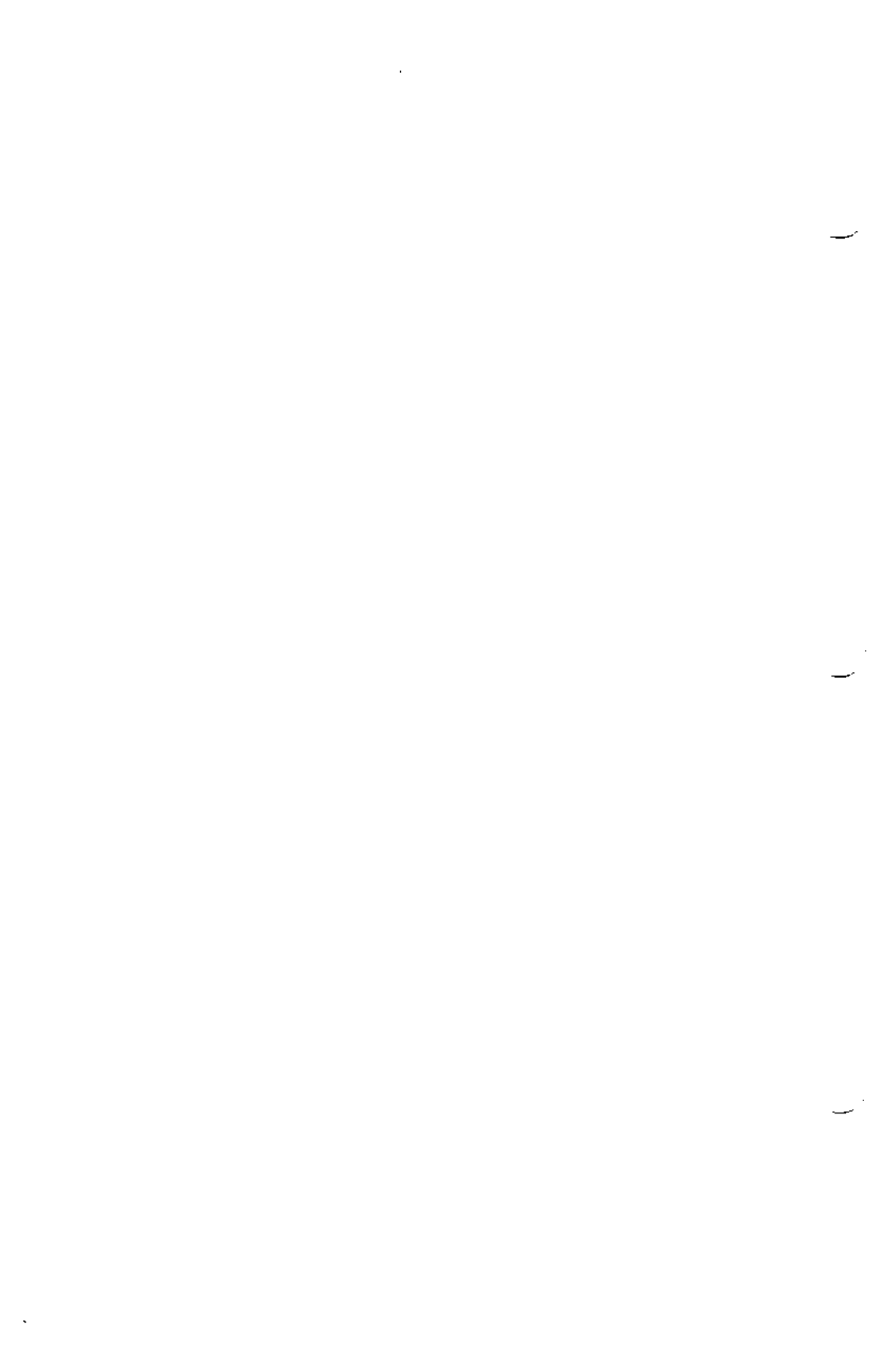
instructs **make** to print out the commands which it would issue without actually executing them.

If a change to a file is absolutely certain to be mild in character, such as adding a new definition to an include file, the **-t** (touch) option can save a lot of time. When **-t** is used, **make** updates the modification times on the affected file, but does not perform a large number of superfluous recompilations. For example:

```
make -ts
```

("touch silently") causes the relevant files to appear up to date. Obvious care is necessary since this mode of operation subverts the intention of **make** and destroys all memory of the previous relationships.

The debugging flag (**-d**) causes **make** to print out a very detailed description of what it is doing including the file times. The output is verbose and recommended ONLY as a last resort.



Chapter 9: AUGMAKE

CONTENTS

1. Introduction	1
2. Environment Variables	2
2.1 MAKEFLAGS	2
2.2 Precedence	2
3. Internal Definitions	5
4. Recursive makefiles	9
5. Shell Command Format	10
6. Archive Libraries	11
7. Example	15
8. SCCS File Names	17
8.1 SCCS Suffixes	17
8.2 SCCS Transformation Rules	17
8.3 Invisible SCCS makefiles	18
9. The NULL Suffix	19
10. Include Files	20
11. Dynamic Dependency Parameters	21
11.1 Extensions of Internally Generated Macros	22
12. Output Translations	24
13. Incompatibility with Old Version of make	26
14. The Hidden Variable	27

1

2

3

Chapter 9

MAKE – AN AUGMENTED VERSION

1. Introduction

This section describes an augmented version of the UNIXTM Operating System's **make** command. This new version of **make** is presented through brief descriptions and examples of working **makefiles**. Only the additional features are described in this chapter. See the chapter entitled "MAKE – MAINTAINING COMPUTER PROGRAMS" in the **UniPlus+ Programming Tools Guide**.

This augmented version is upward compatible with the old version.

Some justification will be given for the chosen implementation, and examples will demonstrate the additional features.

This augmented version was developed primarily due to the fact that the previous version of **make** had the following shortcomings:

- Handling of libraries was tedious.
- Handling of the Source Code Control System (SCCS) file name format was difficult or impossible.
- Environment variables were completely ignored by **make**.
- There was a general lack of ability to maintain files in a remote directory.

These shortcomings hindered large scale use of **make** as a programming tool.

AUGMAKE

This augmented version of **make** is modified to handle the above problems.

2. Environment Variables

2.1 MAKEFLAGS

A new macro, **MAKEFLAGS**, is maintained by **make**. This new macro is defined as the collection of all input flag arguments into a string (without minus signs). The new macro is **exported** and accessible to further invocations of **make**.

Command line flags and assignments in the **makefile** update **MAKEFLAGS**.

:MAKEFLAGS are read and set again when the environment settings are read by **make**.

2.2 Precedence

Environment variables are read and added to the macro definitions each time **make** executes. Precedence is a prime consideration in doing this properly. The following is the precedence of assignments:

1. Command line
2. Makefile(s)
3. Environment
4. Internal definitions from **rules.c**

When executed, **make** assigns macro definitions in the following order:

1. Read the **MAKEFLAGS** environment variable.

Each letter in **MAKEFLAGS** is processed as an input flag argument, unless the letter is one of the following:

- -f
- -p

• -r

If the MAKEFLAGS variable is NULL, or is not present, MAKEFLAGS is set to the NULL string.

2. Read and set the input flags from the command line.

The command line adds to the previous settings in the MAKEFLAGS environment variable.

3. Read macro definitions from the command line.

Any macro definitions set from the command line CAN-NOT be reset. Further assignments to these macro names are ignored.

4. Read the internal list of macro definitions.

Make reads the file **rules.c** in the source for **make**, which contains the internal list of macro definitions.

Consider the following:

If the command **make -r ...** is given, and a **makefile** includes the internally defined rules and macros of the current version of **make**, the **-r** option would have no effect. In fact, the effect would be identical to that occurring if both the **-r** option and the **include** line in the **makefile** were excluded.

The section of this document entitled **Internal Definitions** contains the complete **makefile** that represents the rules and internally defined macros of the current version of **make**. The contents of this **makefile** can be directed to standard output with the following command:

```
make -fp - < /dev/null 2>/dev/null
```

5. Read the environment.

The environment variables are treated as macro

AUGMAKE

definitions and marked as exported.

NOTE: Because **MAKEFLAGS** is NOT an internally defined variable in the file **rules.c**, this step has the effect of doing the same assignment twice. (The exception to this is when **MAKEFLAGS** is assigned on the command line.) The reason the **MAKEFLAGS** variable was read previously was to make sure that the debug flag was turned on, if necessary, before anything else was done.

:MAKEFLAGS is read and set again.

6. Read the **makefile(s)**.

The assignments in the **makefile(s)** override the environment unless the **-e** flag is used. The command line option **-e** instructs **make** to override the **makefile** assignments with the environment settings.

MAKEFLAGS override the environment if assigned. This is useful for further invocations of **make** from the current **makefile**.

There is no way to override the command line assignments.

Consider the following:

If the command **make -e ...** is given, the variables in the environment override the definitions in the **makefile** and reset the precedence of assignments to the following:

1. Command line
2. Environment
3. **Makefile(s)**
4. Internal definitions from **rules.c**

3. Internal Definitions

The following is the contents of the **makefile** representing the internally defined macros and rules:

```
# LIST OF SUFFIXES

.s .s~ .sh .sh~ .h .h~

# PRESET VARIABLES

MAKE=make
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LD=ld
LDFLAGS=
CC=cc
CFLAGS=-o
AS=as
ASFLAGS=
GET=get
GFLAGS=

# SINGLE SUFFIX RULES

.c:
    $(CC) -n -O $< -o $@

.c~:
    $(GET) $(GFLAGS) -p $< > $.c
    $(CC) -n -O $.c -o $*
    -rm -f $.c

.sh:
    cp $< @

.sh~:
    $(GET) $(GFLAGS) -p $< > .sh
    cp $* .sh $*
    -rm -f $* .sh
```


AUGMAKE

DOUBLE SUFFIX RULES

```
.c.o:
    $(CC) $(CFLAGS) -c $<

.c~.o:
    $(GET) $(CFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c

.c~.c:
    $(GET) $(GFLAGS) -p $< > $*.c

.s.o:
    $(AS) $(ASFLAGS) -o $@ $<

.s~.o:
    $(GET) $(GFLAGS) -p $< > $*.s
    $(AS) $(ASFLAGS) -o $*.o $*.s
    -rm -f $*.s

.y.o:
    $(YACC) $(YFLAGS) $<
    $(CC) $(CFLAGS) -c y.tab.c
    rm y.tab.o$@

.y~.o:
    $(GET) $(GFLAG) -p $< > $*.y
    $(YACC) $(YFLAGS) $*.y
    $(CC) $(CFLAG) -c y.tab.c
    rm -f y.tab $*.y
    mv y.tab.o $*.o
```

```

.l.o:
$(LEX) $(LFLAGS) $<
$(CC) $(CFLAGS) -c lex.yy.c
rm lex.yy.c
mv lex.yy.o $@

.l~.o:
$(GET) $(GFLAGS) -p $< > $*.l
$(LEX) $(LFLAGS) $*.l
$(CC) $(CFLAGS) -c lex.yy.c
rm -f lex.yy.c $*.l
mv lex.yy.o $*.o

.y.c:
$(YACC) $(YFLAGS) $<
mv y.tab.c $@

.y~.c:
$(GET) $(GFLAGS) -p $< > $*.y
$(YACC) $(YFLAGS) $*.y
mv -f $*.c
-rm -f $*.y

.l.c:
$(LEX) $<
mv lex.yy.c$@

.c.a:
$(CC) -c $(FLAGS) $<
ar rv $@ $*.o
rm -f $*.o

.c~.a:
$(GET) $(GFLAGS) -p $< > $*.c
$(CC) -c $(CFLAGS) $*.c
ar rv $@ $*.o

```

AUGMAKE

.s~.a:

```
$(GET) $(GFLAGS) -p $< > $*.s  
$(AS) $(ASFLAGS) -o $*.o $*.s  
ar rv $@ $*.o  
-rm -f $*.sol
```

.h~.h

```
$(GET) $(GFLAGS) -p $< > $*.h
```

4. Recursive makefiles

If the sequence **\$(MAKE)** appears anywhere in a shell command line, the command line is executed even if the **-n** flag has been set.

Since the **-n** flag is exported across invocations of **make** by the **MAKEFLAGS** variable, the only thing that actually gets executed is the **make** command itself. This feature is useful when a hierarchy of **makefile(s)** describes a set of software subsystems.

For testing purposes, **make -n ...** can be executed and everything that would have been done will be echoed to standard output, including output from lower level invocations of **make**.

AUGMAKE

5. Shell Command Format

The **make** program remembers embedded newlines and tabs in shell command sequences. If there is a **for** loop in the makefile with indentation, **make** will print it out with the indentation and backslashes.

Output can still be piped to the shell and is readable.

This is a cosmetic change — no new function is gained.

6. Archive Libraries

The augmented version of **make** has an improved interface to archive libraries. The previous version of **make** allows a user to name a member of a library in either of the following ways:

- **lib(object.o)**
- **lib((_localtime))**

Use of **lib((_localtime))** actually refers to the entry point of an object file within the library, and instructs **make** to look through the library, locate the entry point and translate the reference given to the correct object file name.

Using the old version of **make**, to maintain an archive library the following type of **makefile** is then required:

```
lib:: lib(ctime.o)
      $(CC) -c -O ctime.c
      ar rv lib ctime.o
      rm ctime.o
lib:: lib(fopen.o)
      $(CC) -c -O fopen.c
      ar rv lib fopen.o
      rm fopen.o
```

... and so on for each object ...

This method is tedious and error prone. Obviously, the command sequences for adding a C language file to a library are the same for each invocation. In most cases, the file name is the only difference each time. Therefore, the augmented version of **make** provides a rule for building libraries.

The rule for building libraries using the augmented version of **make** is the **.a** suffix rule. For example, the **.c.a** rule is the rule for:

AUGMAKE

- Compiling a C language source file
- Adding a C language source file to the library
- Removing the .o cadaver of the C language source file

The **.y.a** rule is the rule for performing the same functions on a YACC file.

The **.s.a** rule is the rule for performing the same functions on an assembler file.

The **.l.a** rule is the rule for performing the same functions on a LEX file.

The current archive rules defined internally are **.c.a**, **.c~.a**, and **.s~.a**. (See the section of this document concerning SCCS files for an explanation of the tilde (~) syntax.)

Programmers may choose to define additional rules in the **makefile(s)**.

A library is then maintained with the following shorter **makefile**:

```
lib:
    lib(ctime.o)
    @echo lib up-to-date.
```

It should be understood that it is the first parenthesis in the name of the file and not an explicit **.a** suffix which identifies the target suffix rule. For example, the actual rule **.c.a** is defined as follows:

```
.c.a:
    $(CC) -c $(CFLAGS) $<
    ar rv $@ $*.o
    rm -f $*.o
```

In the `.c.a` rule:

- `$@`** This macro is the `.a` target. (Using the previous example, this macro would be defined as `lib`.)
- `$<` and `$*`** These macros are set to the out-of-date C language file, and the file name scans the suffix. Using the previous example, these macros would be defined as `ctime.c` and `ctime`. Using this example, the `$<` macro could have been changed to `$*.c`.

When `make` sees this instruction in the `makefile` (assuming the object in the library is out of date with respect to `ctime.c`, and there is no `ctime.o` file) it translates that construction into the following sequence of operations:

1. Do `lib`.
2. To do `lib`, do each dependent of `lib`.
3. Do `lib(ctime.o)`.
4. To do `lib(ctime.o)`, do each dependent of `lib(ctime.o)`. (There are none in this example.)

To let `ctime.o` have dependencies, the following syntax is required:

```
lib(ctime.o):    $(INCDIR)/stdio.h
```

Thus, explicit references to `.o` files are unnecessary.

There is also a new macro for referencing the archive member name when this form is used. The `$$` macro is evaluated each time `$@` is evaluated. If there is no current archive member, `$$` is NULL. If an archive member exists, then `$$` evaluates to the expression between the parentheses.

5. Use internal rules to try to build `lib(ctime.o)`. (There is no explicit rule.)

AUGMAKE

Note that it is the first parenthesis in the name **lib(ctime.o)** which identifies the target suffix. This is the key. There is no explicit **.a** at the end of the **lib** library name. The parenthesis forces the **.a** suffix. In this sense, the suffix is **hard-wired** into **make**.

6. Break the name **lib(ctime.o)** up into **lib** and **ctime.o**. Define two macros, **\$@ (=lib)** and **\$* (=ctime)**.
7. Look for a rule **.X.a** and a file **\$*.X**. The first **.X** (in the **.SUFFIXES** list in **rules.c**) which fulfills these conditions is **.c** so the rule is **.c.a** and the file is **ctime.c**.
8. Set **\$<** to be **ctime.c** and execute the rule.

In fact, **make** must then do **ctime.c**. However, the search of the current directory yields no other candidates, and the search ends.

9. The library has been updated. Do the next instruction associated with the **lib:** dependency. Therefore, **make** will echo **lib up to date**.

7. Example

The following is an example **makefile** for a larger library:

```
# Example makefile
```

```
LIB=lsxlib
```

```
PR=lp
```

```
INSDIR=/rl/flopO/
```

```
INS=eval
```

```
lsx: $(LIB) low.o mch.o
    ld -x low.o mch.o $(LIB)
    mv a.out lsx
    @size lsx
```

```
# Here, $(INS) as either "." or "eval".
```

```
lsx:
    $(INS)'cp lsx $(INSDIR)lsx . .
    strip $(INSDIR)lsx . .
    ls -l $(INSDIR)lsx'
```

```
print:
    $(PR) header.slow.smch.s*.h*.c Makefile
```

```
$(LIB):
    $(LIB)(CLOCK.o)
    $(LIB)(main.o)
    $(LIB)(tty.o)
    $(LIB)(trap.o)
    $(LIB)(sysent.o)
    $(LIB)(sys2.o)
    $(LIB)(syn3.o)
    $(LIB)(syn4.o)
    $(LIB)(sys1.o)
    $(LIB)(sig.o)
    $(LIB)(flo.o)
```

AUGMAKE

```
$(LIB)(kl.o)
$(LIB)(alloc.o)
$(LIB)(nami.o)
$(LIB)(iget.o)
$(LIB)(rdwri.o)
$(LIB)(subr.o)
$(LIB)(bio.o)
$(LIB)(decfd.o)
$(LIB)(sip.o)
$(LIB)(space.o)
$(LIB)(puts.o)
@echo $(LIB) now up to date.
```

```
.s.o:
    as -o $*.o header.s $*.s
```

```
.o.a:
    ar rv $@ $<
    rm -f $<
```

```
.s.a:
    as -o $*.o header.s $*.s
    ar rv $@ $*.o
    rm -f $*.o
```

```
.PRECIOUS:
    $(LIB)
```

Note that using this **makefile**, there will not be any lingering ***.o** files. The result is a library maintained directly from the source files (or more generally from the SCCS files).

8. SCCS File Names

The syntax of **make** does not directly permit prefix references. Commonly, UNIX Operating System users distinguish most types of files, by means of a suffix, so references to prefixes are seldom necessary. That is, with one important exception — SCCS files.

SCCS file names are preceded with a **.s** prefix. To allow **make** easy access to this prefix, the augmented version of **make** uses the tilde (**~**) as an identifier of SCCS files.

The expression **.c~.o** refers to the rule which transforms an SCCS C language source file into an object file.

```
.c~.o:
    $(GET) $(GFLAGS) -p $< > $*.c
    $(CC) $(CFLAGS) -c $*.c
    -rm -f $*.c
```

The tilde appended to any suffix transforms the file search into an SCCS file name search with the actual suffix named by the dot and all characters up to (but not including) the tilde (**~**).

8.1 SCCS Suffixes

The following SCCS suffixes are internally defined:

```
.c~ .y~ .s~ .sh~ .h~
```

8.2 SCCS Transformation Rules

The following rules involving SCCS transformations are internally defined:

```
.c~: .l~.o: .sh~: .y~.c: .c~.o:
.c~.a: .s~.o: .s~.a: .y~.o: .h~.h:
```

AUGMAKE

Other rules and suffixes which may prove useful can be defined using the tilde as a handle on the SCCS file name format.

8.3 Invisible SCCS makefiles

SCCS **makefiles** are invisible to **make** in that if **make** is typed and only a file named **s.makefile** exists, **make** will get, read and remove the file.

If the **-f** option is used, **make** will get, read and remove arguments and **include** files.

9. The NULL Suffix

In the UNIX Operating System source code, there are many commands which consist of a single source file. It was wasteful to maintain an object of such files for **make**. The current implementation supports single suffix rules (a NULL suffix).

For example, to maintain the program **cat**, a rule in the **makefile** of the following form is needed:

```
.c:
    $(CC) -n -O $< -o $@
```

This **.c:** rule is internally defined so no **makefile** is necessary at all. To **make** the programs **cat**, **dd**, **echo** and **date** (all single file programs), the command line would simply be:

```
make cat dd echo date
```

Giving this command instructs **make** to pass all four C language source files through the above shell command line associated with the **.c:** rule.

The following are the internally defined single suffix rules:

```
.c: .c~: .sh: .sh~:
```

Others may be added in the **makefile** by the user.

10. Include Files

If the string **include** appears as the first seven letters of a line in a **makefile** and is followed by a blank or a tab, the string following is assumed to be a file name to be read by the current invocation of **make**.

The file descriptors are stacked for reading **include** files, therefore no more than approximately 16 levels of nested **includes** are supported.

11. Dynamic Dependency Parameters

A new dependency parameter has been defined which has meaning **ONLY** on the dependency line in a **makefile**. This parameter, **\$\$@**, refers to the current **thing** to the left of the colon (which is **\$@**).

For example, consider the following:

```
cat:  $$@.c
```

In this example, the dependency is translated into the string **cat.c** at execution time. This is useful for building a large number of executable files, each of which has only one source file.

For example, the UNIX software command directory could have a **makefile** like the following:

```
CMDS = cat dd echo date cc cmp comm ar ld chown
```

```
$(CMDS):  $$@.c
          $(CC) -O $? -o $@
```

In this example, **CMDS** is defined as a subset of all the single file programs.

For multiple file programs, a directory is usually allocated and a separate **makefile** is made. For any particular file that has a peculiar compilation procedure, a specific entry must be made in the **makefile**.

The second useful dependency parameter is **\$\$(@F)**. The parameter **\$\$(@F)** is another form of **\$\$@** which represents the file name part of **\$\$@**. This parameter is also evaluated at execution time.

AUGMAKE

To illustrate the usefulness of this parameter, consider the following example. To maintain the **/usr/include** directory from a **makefile** in another directory (called **/usr/src/head** in this example), the **makefile** would look like:

```
INCDIR = /usr/include

INCLUDES = \
    $(INCDIR)/stdio.h \
    $(INCDIR)/pwd.h \
    $(INCDIR)/dir.h \
    $(INCDIR)/a.out.h

$(INCLUDES): $$(@F)
    cp $? $@
    chmod 0444 $@
```

This **makefile** would completely maintain the **/usr/include** directory whenever one of the above files in **/usr/src/head** was updated.

11.1 Extensions of Internally Generated Macros

The internally generated macros:

- **\$***
- **\$@**
- **\$<**

are useful generic terms for current targets and out-of-date relatives.

To this list have been added the following related macros:

- **\$(@D)**
- **\$(@F)**
- **\$(*D)**
- **\$(*F)**
- **\$(<D)**
- **\$(<F)**

The **D** refers to the **D**irectory part of the single letter macro, and the **F** refers to the **F**ile name part of the single letter macro. These additions are useful when building hierarchical makefiles.

For example, the following instruction uses the **D** to gain access to directory names for purposes of using the **cd** command:

```
cd $(<D); $(MAKE) $(<F)
```

Also consider the following example, which forces a complete rebuild of the operating system:

```
FRC=FRC make -f 70.mk
```

where the current directory is the top of the file system.

FRC is a convention for **FoRCing make** to completely rebuild a target starting from scratch.

12. Output Translations

Macros in shell commands can now be translated when evaluated. To accomplish this, use the following format:

```
$(macro:string1=string2)
```

When **make** encounters this construction, it first evaluates the meaning of **\$(macro)** by considering **\$(macro)** as a bunch of strings each delimited by white space (blanks or tabs). Then, for each appearance of **string1** in **\$(macro)**, **string2** is substituted. The **string1** is located when the following regular expression is matched:

```
.*<string1>[TAB|BLANK ]
```

This particular form of regular expression is used because **make** generally concerns itself with suffixes.

A more general regular expression match can be implemented if necessary.

To illustrate the usefulness of this facility, consider the following example situation:

To maintain an archive library, the out-of-date members must be accumulated and a shell script must be written to handle all the C programs. The following fragment will optimize the executions of **make** for archive libraries:

```
.SUFFIXES: .c .a  
.c.a::  
$(LIB): $(LIB)(a.o) $(LIB)(b.o) $(LIB)(c.o)  
$(CC) -c $(CFLAGS) $(?:.o=.c)  
ar rv $(LIB) $?  
rm $?
```

AUGMAKE

The translation (\$(?:.o=.c)) is added in an effort to make more general use of the wealth of information which make generates.

AUGMAKE

13. Incompatibility with Old Version of make

The only known incompatibility with the older version of **make** is seen in the following example **makefile**:

```
all:   cat dd

dd:    dd.o
      $(CC) -o $@ $?

cat:   cat.o
      $(CC) -o $@ ??
```

The old version of **make** will NOT complain that **all** does not have a rule associated with it, but the augmented version will.

The **-b** option instructs **make** to revert to the old method, so that old makefiles can be run with this augmented version of **make**.

Any other differences are unintentional.

14. The Hidden Variable

There is an interesting **hidden** variable in this version of **make** — **#!** which represents the current predecessor tree.

For example, using the following **makefile**:

```
all:    cat
        @echo cat up-to-date

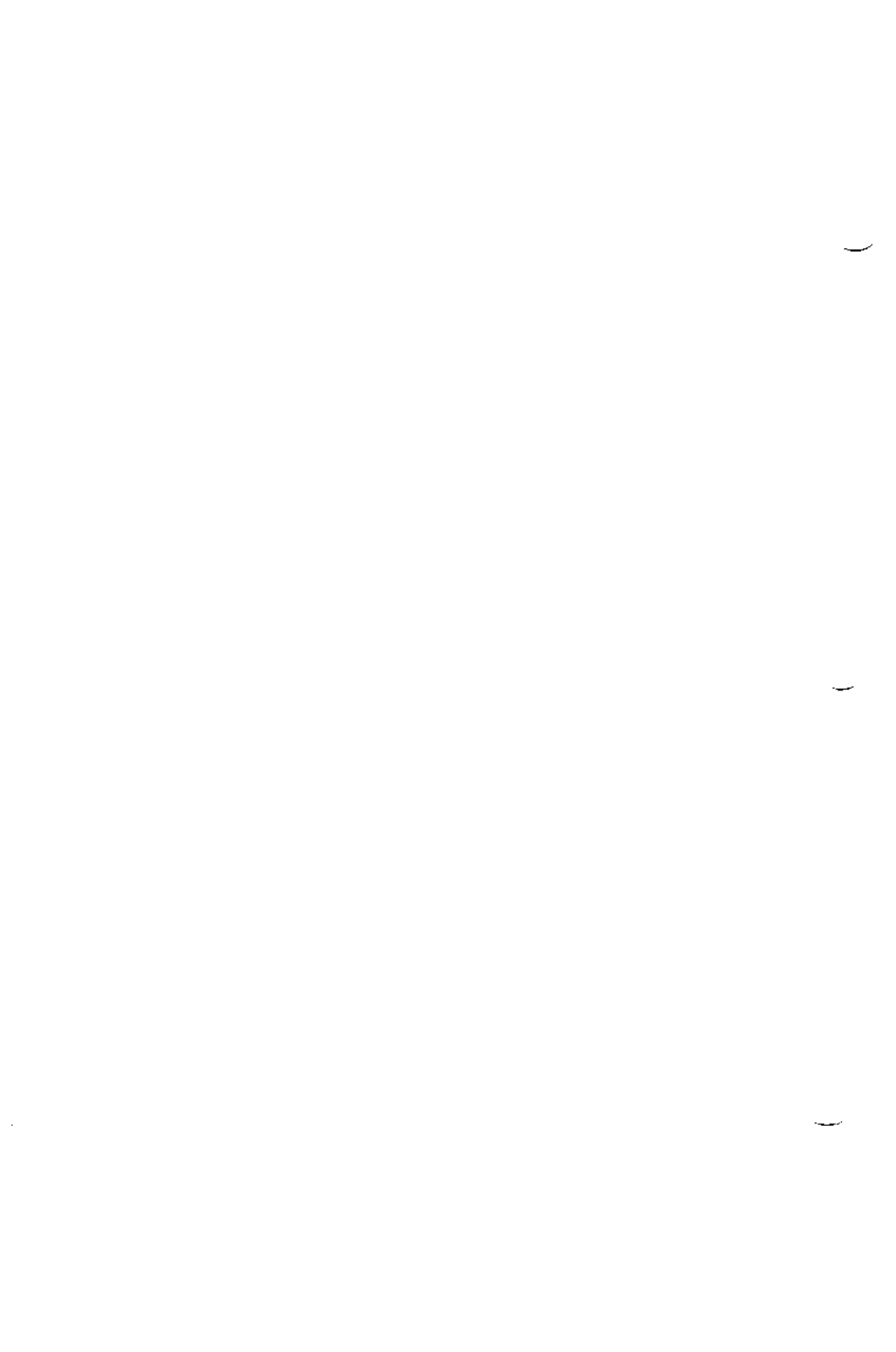
cat:    cat.c
        echo #!
```

when the **echo #!** is executed, the variable **#!** evaluates to
cat.c cat all

which is not **all** that useful! Further, it occasionally prints the following message:

#! nulled, predecessor circle

This message means that the predecessors of a file are circular. The actual evaluation of the **#!** macro was aborted, and its value set to **NULL**. Otherwise, there is no effect.



Chapter 10

SCCS: Source Code Control System

CONTENTS

1. Introduction	2
2. SCCS for Beginners	4
2.1 Terminology	4
2.2 Creating an SCCS File	4
2.3 Retrieving an SCCS File	6
2.4 Record Your Changes in a New Version	7
2.5 SIDs: More on Retrieving Versions	8
2.6 Help	9
3. SCCS Files	11
3.1 Protection	11
3.1.1 SCCS Administrator	13
3.1.2 An Interface Program	13
3.2 Formatting	16
3.3 Auditing	17
3.4 Delta Numbering	19
4. SCCS Commands Conventions	23
4.1 Command Arguments	23
4.2 Flags	24
4.3 Diagnostics	24
4.4 Temporary Files	24
5. SCCS Command Summary	28
5.1 The admin Command	30
5.1.1 Creating SCCS Files	30
5.1.2 SCCS Flags	31
5.1.3 Comments and MR Numbers	34
5.1.4 Descriptive Text	35
5.2 The cdc Command	36

5.3	The comb Command	37
5.4	The delta Command	39
5.5	The get Command	44
5.5.1	ID Keywords	45
5.5.2	Retrieving Different Versions	46
5.5.3	Retrieving Files to Make Deltas	49
5.5.4	Concurrent Edits of Different SIDs	52
5.5.5	Concurrent Edits of Same SID	58
5.5.6	Keyletters That Affect Output	58
5.6	The unget Command	62
5.7	The help Command	63
5.8	The prs Command	64
5.9	The rmdel Command	67
5.10	The sact Command	69
5.11	The sccsdiff Command	70
5.12	The val Command	71
5.13	The what Command	72

LIST OF FIGURES

Figure 10-1.	A Project-Dependent Interface Program	14
Figure 10-2.	Evolution of an SCCS File	20
Figure 10-3.	Tree Structure With Branch Deltas	21
Figure 10-4.	Extending the Branching Concept	22
Figure 10-5.	Determination of New SID (Sheet 1 of 3)	55
Figure 10-5.	Determination of New SID (Sheet 2 of 3)	56
Figure 10-5.	Determination of New SID (Sheet 3 of 3)	57

Chapter 10

SCCS: Source Code Control System

1. Introduction

The Source Code Control System (SCCS) is a collection of UniPlus⁺ software commands that controls and accounts for changes to files of text. SCCS provides facilities for:

- Storing text files
- Retrieving earlier versions of files
- Controlling update privileges to files
- Identifying the version of a retrieved file
- Recording when, where, why, and by whom each change is made to a file.

It is convenient to think of SCCS as a custodian of files. It stores the original file on disk, and whenever changes are made to the file, SCCS stores only the changes. Each set of changes is called a **delta**. Different versions of the file can then be reconstructed by applying deltas to the original version of the file.

This chapter and the relevant portions of the *UniPlus⁺ User Manual* provide a complete guide to SCCS. The following topics are covered here:

- **SCCS for Beginners:** A step-by-step tutorial that demonstrates how to create, update, and retrieve a version of an SCCS file.
- **SCCS Files:** A description of the protection mechanisms, format, auditing, and delta numbering of SCCS files. The differences between individual SCCS use and group or project SCCS use are discussed, and the role of the SCCS Administrator is introduced.

SCCS

- **SCCS Command Conventions:** A description of the conventions that generally apply to SCCS commands and the temporary files generated by SCCS commands for various purposes during their execution.
- **SCCS Command Summary:** A description of all SCCS commands and their most frequently used arguments.

Neither the implementation of SCCS nor the installation procedure for SCCS is described in this chapter.

Throughout this chapter, references to *name(1M)*, *name(7)*, or *name(8)* refers to the *name* entry in the *UniPlus+ Administrator Manual*. All other references of the form *name(n)*, (where *n* is a number 1 through 6 possibly followed by a letter) refer to the *name* entry in section *n* of the *UniPlus+ User Manual*.

The percent sign (%) at the beginning of command lines represents the default C Shell prompt and is not part of the command.

2. SCCS for Beginners

This section assumes that you know how to login to a UniPlus⁺ system and create files using a text editor. To supplement the material in this section, see the detailed SCCS command descriptions in the *UniPlus⁺ User Manual*.

2.1 Terminology

Each SCCS file is composed of one or more sets of changes applied to the null (empty) version of the file. Each set of changes is called a **delta**. Each delta normally depends on all previous deltas.

Each delta is assigned a name, called the **SID** (SCCS **I**dentification **S**tring). The SID is normally composed of two components: the **release** number and **level** number, separated by a period. The first delta (for the original file) is called 1.1, the second 1.2, the third 1.3, etc. The release number can also be changed (allowing, for example, deltas 2.1, 3.1, etc.) to indicate a major change to the file. Under special conditions, SIDs can be created with four components. See the section on "Delta Numbering" for more information.

Each delta of an SCCS file defines a particular version of the file. For example, delta 1.5 defines version 1.5 of the SCCS file. This version is obtained by applying to the null (empty) version of the file the changes that constitute deltas 1.1, 1.2, etc., up to and including delta 1.5 itself, in that order.

2.2 Creating an SCCS File

Consider an ordinary text file called *lang* that contains a list of programming languages:

SCCS

```
c
pl/i
fortran
cobol
algol
```

The following **admin** command creates an SCCS file and initializes delta 1.1 from the file *lang*:

```
% admin -i lang s.lang
```

The **-i** keyletter, together with its value *lang*, indicates that **admin** is to create a new SCCS file and 'initialize' the new (null) version with the contents of the file named *lang*. This initial version is a set of changes (delta 1.1) applied to the null SCCS file.

The *s.lang* argument is the file name for the new SCCS file.

■ All SCCS files must have names that begin with 's'.

The **admin** command returns a warning message (which may also be issued by other SCCS commands):

```
No id keywords (cm7)
```

The significance of this message is described under the **get** command in the section on "SCCS Command Summary" below. In the following examples, this warning message is not shown although it may actually be issued by the commands.

You can now remove the *lang* file from your directory:

```
% rm lang
```

since it can be easily reconstructed using the SCCS `get` command.

2.3 Retrieving an SCCS File

The *lang* file can be reconstructed using the following `get` command:

```
% get s.lang
```

This retrieves the latest version of file *s.lang* and prints the following messages:

```
1.1  
5 lines
```

These messages indicate the SID of the version retrieved, and the length of the retrieved text. The text is placed in a file (called the *g-file*) whose name is formed by deleting the 's.' prefix from the name of the SCCS file. Hence, the file *lang* is reconstructed.

When you use the `get` command with no keyletters (in the format above) the *lang* file is created with read-only mode (mode 440), and no information about the SCCS file is retained. If you want to be able to change an SCCS file and create a new version, use the `-e` keyletter on the `get` command line:

```
% get -e s.lang
```

The `-e` keyletter causes `get` to create *lang* with read-write permission (so you can edit it) and places certain information about the SCCS file in another file (called the *p-file*), which will

SCCS

be read by the **delta** command when the time comes to create a delta.

The same messages display, except the SID of the next **delta** (to be created) is also issued. For example:

```
% get -e s.lang
```

```
1.1
new delta 1.2
5 lines
```

After this command you can **vi** into the *lang* file and make changes. For example:

```
% vi lang
```

```
c
pl/i
fortran
cobol
algol
ada
pascal
```

2.4 Record Your Changes in a New Version

The command:

```
% delta s.lang
```

records the changes you made to the *lang* file within the SCCS file.

Delta prompts with:

```
comments?
```

Your response should be a description of why the changes were made. For example,

comments? **added more languages**

The **delta** command then reads the *p-file* and determines what changes were made to the file *lang*. When this process is complete the changes to *lang* are stored in *s.lang* and **delta** displays:

1.2
2 inserted
0 deleted
5 unchanged

The number 1.2 is the SID of the new delta, and the next three lines refer to the changes recorded in the *s.lang* file.

2.5 SIDs: More on Retrieving Versions

The **-r** keyletter allows you to retrieve a particular delta by specifying its SID on the **get** command line. For our previous example, the following commands are all equivalent:

```
% get s.lang  
% get -r1 s.lang  
% get -r1.2 s.lang
```

The numbers following the **-r** keyletter are SIDs.

- The first command retrieves the most recent version of the SCCS file, since no SID is specified.
- When you omit the level number of the SID (as in the second command), SCCS retrieves the most recent level number in that release. (In our previous example, the latest version in release 1, namely 1.2.)

SCCS

- The third command explicitly requests the retrieval of a particular version (in this case, also 1.2).

Whenever a major change is made to a file, the significance of that change is usually indicated by changing the release number (first component of the SID) of the delta being made. Since normal automatic numbering of deltas proceeds by incrementing the level number (second component of the SID), the user must explicitly change the release number as follows:

```
% get -e -r2 s.lang
```

Because release 2 does not yet exist, **get** retrieves the latest version *before* release 2 and changes the release number of the next delta to 2, naming it 2.1 rather than 1.3. This information is stored in the *p-file* so the next execution of the **delta** command will produce a delta with the new release number. The **get** command then outputs:

```
1.2
new delta 2.1
7 lines
```

which indicates that version 1.2 has been retrieved and that 2.1 is the version **delta** will create. Subsequent versions of the file will be created in release 2 (deltas 2.2, 2.3, etc.).

2.6 Help

The **help** command is useful whenever there is any doubt about the meaning of an SCCS message. Detailed explanations of almost all SCCS messages can be found using the **help** command and the code printed in parentheses after the message.

If you give the command:

% get abc

SCCS will print the message:

ERROR [abc]: not an SCCS file (col)

The string 'col' is a code that may be used to obtain a fuller explanation of that message using the **help** command. The command:

% help col

produces:

col:

'not an SCCS file'

A file that you think is an SCCS file
does not begin with the characters 's.'.

3. SCCS Files

This section discusses the protection mechanisms used by SCCS, the format of SCCS files, recommended procedures for auditing SCCS files, and how deltas are numbered.

3.1 Protection

SCCS provides certain protection features directly, i.e., three SCCS file flags (*release ceiling*, *release floor*, and *release lock*), and the *user list* for SCCS files.

The flags are set using the **-f** keyletter with the **admin** command (see the section on "SCCS Flags" under the **admin** command below.) The flags used for file protection are:

- f flag** This keyletter specifies a *flag* and possibly a value for the flag, to be placed in the SCCS file. Several **f** keyletters may be supplied on a single **admin** command line.
 - c cell** The highest release ('ceiling') that may be retrieved by a **get** command for editing. *ceil* is a number less than or equal to 9999. The default value is 9999.
 - f floor** The lowest release ('floor') that may be retrieved by a **get** command for editing. *floor* is a number less than 9999 and greater than 0. The default value is 1.
 - l list** A list of 'locked' releases to which deltas can no longer be made. (See *admin(1)* for the syntax of this list.) The **get -e** command on one of these locked releases fails. The character **a** in *list* is equivalent to specifying *all releases* for the named SCCS file.

SCCS files also contain a *user list* of login names and/or group IDs of users who are allowed to create deltas of that file. This list is empty by default, which means that anyone may create deltas. To add names to the list (either to allow permission or deny it) the **-a** keyletter is used with the **admin** command.

-a login A *login* name or numerical group ID. A group ID is equivalent to specifying all *login* names common to that ID. If *login* or group ID is preceded by an exclamation character ! they are denied permission to make deltas.

These features are described in more detail under the **admin** command below.

In addition to the above features, SCCS uses standard UniPlus⁺ protection mechanisms to prevent unauthorized changes to SCCS files (i.e., changes made by non-SCCS commands). These include:

1. New SCCS files created by the **admin** command are given mode 444 (read-only). It is recommended that you don't change this mode, since it prevents any direct modification of the files by non-SCCS commands. It is further recommended that the directories containing SCCS files should be mode 755, which allows only the owner of the directory to modify its contents.
2. SCCS files should be kept in directories that contain only SCCS files and any temporary files created by SCCS commands. This simplifies protection and auditing of SCCS files. The contents of directories should correspond to convenient logical groupings, e.g., subsystems of a large project.
3. SCCS commands will not process an SCCS file that is linked to another file. The commands that modify SCCS files do so by creating a copy (called the *x-file*), modifying

SCCS

it, then removing the old *s.file* and renaming the *x.file*. (Remember that all SCCS files must have names that begin with 's'.) If the old file had more than one link, this process would break the links. SCCS commands produce an error message rather than process a file that has been linked.

3.1.1 SCCS Administrator

For these protection mechanisms to be most effective, a single user should own the SCCS files and directories. If you are the only SCCS user, the real and effective user IDs are the same; and the user ID owns the directories containing SCCS files. (See `passwd(1)` in the *UniPlus⁺ User Manual*.) In this case, you can simply use SCCS directly.

However, when there are several users assigned responsibility for one SCCS file (e.g., in large software development projects), one user (i.e., one user ID) must be chosen as the 'owner' of the SCCS files. This single user will be the only one to 'administer' the SCCS files (e.g., using the **admin** command). This user is called the 'SCCS Administrator' for that project.

This means that no SCCS users other than the SCCS Administrator are able to use those commands that require write permission in the directory containing the SCCS files. Instead, a project-dependent program must be written to provide an interface to certain SCCS commands, usually the **get**, **delta**, and, if desired, **rmdel** and **cde** commands.

3.1.2 An Interface Program

An example interface program follows:

```

main (argc, argv)
int argc;
char *argv[];
    register int i;
    char cmdstr [LENGTH];
    /*
    Process file arguments (those that don't begin with '-')
    */
    for (i = 1; i < argc; i++)
        if (argv[i][0] != '-')
            argv[i] = filearg (argv[i]);
    /*
    Get 'simple name' of name used to invoke this program (strip
    off directory prefix, if any)
    */
    argv[0] = sname (argv[0]);
    /*
    Invoke actual SCCS command, passing arguments
    */
    sprintf(cmdstr, "/usr/bin/%s", argv[0]);
    execv(c,dstr,argv);

```

Figure 10-1. A Project-Dependent Interface Program

A program such as this invokes the desired SCCS command and causes it to inherit the privileges of the interface program for the duration of that command's execution. Users whose login names or group IDs are in the *user list* for that file (but who are not the owner), and who have the path to the executable interface program in their PATH variable, are given the necessary permissions only for the duration of the execution of the interface program. They can modify the SCCS files only through the use of those commands that are linked to the interface program.

The interface program must be owned by the SCCS administrator, must be executable by the new owner, and must have the

SCCS

'set user ID on execution' bit 'on'.

Links can then be created between the executable interface program and the command names. For example, if the path to the file is:

/sccs/interface.c

Then the commands:

```
% cd /sccs
```

```
% cc interface.c -o inter [args]
```

compile the program into the executable module *inter*. At this point, the command:

```
% chmod 4755 inter
```

sets the proper mode and 'set user ID on execution bit.' You can then create links from any directory with the commands:

```
% ln /sccs/inter get
```

```
% ln /sccs/inter delta
```

```
% ln /sccs/inter rmdel
```

```
% ln /sccs/inter cdc
```

- The path to the directory containing the links must then be included prior the */usr/bin* directory in the PATH variable (in the *.login* or *.profile* files of all SCCS users who need to use the desired SCCS commands). For example:

```
set path=(. /usr/new /bin /sccs /usr/bin /usr/local/bin)
```

Depending on the type of interface program you have written, the names of the links can be arbitrary (if the program can determine from them the names of the commands to be invoked), the pathname to your project can be supplied, and so on. If the pathname to your project is supplied in the interface program, the user can use the syntax:

```
% get -e s.abc
```

regardless of where he is currently located in the file system.

The project-dependent interface program, as its name implies, must be custom-built for each project.

3.2 Formatting

SCCS files are composed of lines of ASCII text arranged in six parts as follows:

Checksum	A line containing the sum of all the characters of the file (not including this checksum itself).
Delta Table	Information about each delta, such as type, SID, date and time of creation, and commentary.
User List	List of login names and/or group IDs of users who are allowed to modify the file by adding or removing deltas.
Flags	Indicators that control certain actions of SCCS commands.
Descriptive Text	Arbitrary text provided by the user; usually a summary of the contents and purpose of the file.

SCCS

Body Actual text that is being administered by SCCS, intermixed with internal SCCS control lines.

See *sccsfile(4)* for detailed information about the contents of these parts of an SCCS file. The *user list* is described in the section on "Protection" above. *Flags* and *descriptive text* are discussed in the section on "The admin Command" below. The *checksum* is described in the "Auditing" section that follows.

It is important to note that, because SCCS files are ASCII files, they can be processed by 'normal' UniPlus⁺ commands such as **vi**, **grep**, and **cat**.[†] This is very convenient when an SCCS file must be modified manually or when you simply want look at the file. However, it is important to be careful about introducing changes that will affect future deltas.

■ **Caution:** Use extreme care when modifying SCCS files with non-SCCS commands.

3.3 Auditing

On rare occasions (such as a system crash) an SCCS file or one or more 'blocks' of it can be destroyed. The SCCS commands issue an error message when a file you attempt to process does not exist; they also use the checksum stored in the SCCS file to determine whether a file has been corrupted since it was last accessed.

If an SCCS file has been corrupted (if part of its contents are lost) the only SCCS command that will process the file is the

[†] Described in Section 1 of the *UniPlus⁺ User Manual*.

admin command with the **-h** or **-z** keyletters, as described below. It is recommended that SCCS files be audited for possible corruptions on a regular basis. The simplest and fastest way to audit your SCCS files is to execute the **admin** command with the **-h** keyletter:

```
% admin -h s.file1 s.file2 ...
```

If the new checksum of any file is not equal to the checksum in the first line of that file, the message

corrupted file (co6)

is produced for that file. This process continues until all the files have been examined. The **admin -h** command can also be applied to directories:

```
% admin -h directory1 directory2 ...
```

Note that this will not automatically report whether any SCCS files are missing. As with any other directory in the UNIX file system, use the **ls** command † to list the contents of each directory and ensure that no files are missing.

If you have an SCCS file that has been extensively corrupted, the best solution is to restore the file from a backup copy. If there is only minor damage, you may be able to repair it using **vl**.

† Described in Section 1 of the *UniPlus* † *User Manual*.

SCCS

After you have repaired the file with a text editor, use the command:

```
% admin -z s.file
```

This command recomputes the checksum so that it agrees the actual contents of the file. After this command is executed on a file, any corruption that existed in the file will no longer be detectable by the **admin -h** command.

3.4 Delta Numbering

It is convenient to think of the deltas applied to an SCCS file as a tree structure where the root is the initial version of the file. Deltas are named with numbers that contain exactly two components; the **release** number and **level** number, separated by a period:

release.level

The root delta is normally named 1.1 and successor deltas are named 1.2, 1.3, etc., by successively incrementing the level number. This is performed automatically by SCCS whenever a delta is made.

You can also change the release number (using the **-r** keyletter) to indicate a major change to the file. When you have done this, the release number also applies to all successor deltas unless specifically changed again.

The evolution of a particular file may be represented as in Figure 10-2.

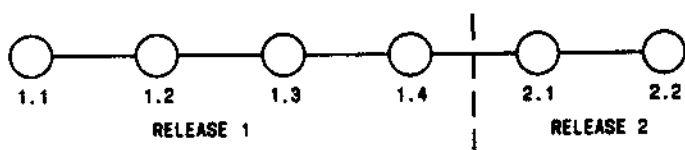


Figure 10-2. Evolution of an SCCS File

Figure 10-2 represents the normal sequential development of an SCCS file. Changes that are part of any given delta are dependent upon *all* preceding deltas. This linear progression of file versions is sometimes called the 'trunk' of the SCCS tree for that file.

However, there can be more complex situations (where changes in a given delta are *not* dependent on all previous deltas) that require a branching in the tree. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible.

To illustrate the branching concept, consider a program that is being used at Version 1.3. Development work on Release 2 of that program is already in progress. Release 2 may already have some deltas (as shown in Figure 10-2).

Assume that a user reports a problem in Version 1.3 that requires changes only to Version 1.3, without affecting subsequent deltas (i.e., in Figure 10-2, deltas 1.4, 2.1, 2.2, etc.). This requires a 'branch' from the previous linear ordering.

SCCS

The new delta's name consists of four components; the **release** and **level** numbers (as in the 'trunk' delta) plus a **branch** number and **sequence** number. Each number is separated by a period.

release.level.branch.sequence

The names of 'branch' deltas contain exactly four components, and thus a branch delta can always be identified as such from its name. The numbers are assigned sequentially as shown in Figure 10-3. The delta number 1.3.1.2 identifies the second delta of the first branch derived from delta 1.3.

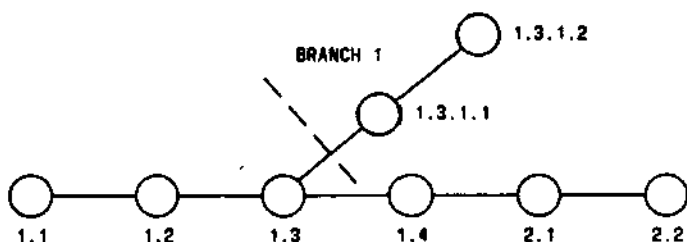


Figure 10-3. Tree Structure With Branch Deltas

The concept of branching can be applied to any delta in the tree. However, although the 'trunk' delta may be identified from the 'branch' delta's name, (by the release and level numbers) it is not possible to determine the entire path leading from the ancestor 'trunk' delta to the 'branch' delta. This is because the branch number is assigned chronologically, *in order of creation of the branch*, independent of its location relative to the ancestor delta.

For example, if delta 1.3 has one branch emanating from it, all deltas on that branch will be named 1.3.1.n. If a delta on this branch then has another branch emanating from it, all deltas on the new branch will be named 1.3.2.n (see Figure 10-4). The only information that can be derived from the name of delta 1.3.2.2 is that it is chronologically the second delta on the chronologically second branch whose trunk ancestor is delta 1.3.

In particular, it is *not* possible to determine from the name of delta 1.3.2.2 all the deltas between it and ancestor 1.3.

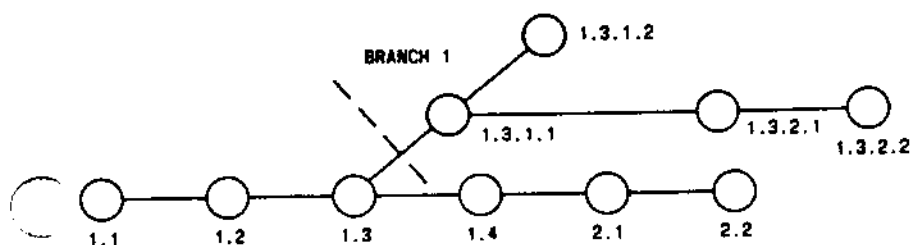


Figure 10-4. Extending the Branching Concept

It is obvious that the concept of branching deltas allows the generation of arbitrarily complex tree structures. Although this capability has been provided for certain specialized uses, it is strongly recommended that the SCCS tree be kept as simple as possible. Comprehending the tree structure becomes very difficult as it becomes more complex.

4. SCCS Commands Conventions

This section discusses the conventions and rules that apply to SCCS commands. Except where noted otherwise, these conventions apply to all SCCS commands. A list of the temporary files generated by various commands (and referred to in the Command Summary) is also provided.

4.1 Command Arguments

SCCS commands accept two types of arguments:

- Keyletters
- File arguments

Keyletters begin with a minus sign (-), followed by a lowercase alphabetic character, which is in some cases followed by a value. Keyletters control the execution of the command to which they are supplied. All keyletters specified for a given command apply to all file arguments of that command.

Keyletters are processed before any file arguments, with the result that the placement of keyletters is arbitrary (i.e., keyletters may be interspersed with file arguments). File arguments, however, are processed left to right. Somewhat different argument conventions apply to the **help**, **what**, **sccsdiff**, and **val** commands.

File arguments (names of files and/or directories) specify the file(s) that the given SCCS command is to process. Naming a directory is equivalent to naming all the SCCS files within the directory. Non-SCCS files in the named directories are silently ignored. In general, file arguments may not begin with a minus sign, but if the name '-' (a single minus sign) is specified as an argument to a command, the command reads the standard input (until end-of-file) and takes each line as the name of an

SCCS file to be processed. This feature is often used in pipelines with, for example, the **find** or **ls** commands.[†]

4.2 Flags

Certain actions of SCCS commands can be controlled by flags that appear in SCCS files. Some of these flags are discussed in the section on "SCCS Flags" below. For a complete description of flags, see *admin(1)* in the *UniPlus⁺ User Manual*.

4.3 Diagnostics

The distinction between the real user and the effective user (see *passwd(1)* and the section on "Protection" above) of a directory or file affects certain SCCS commands. For the present, we assume that the real and effective user are the same.

SCCS commands produce diagnostics (on the diagnostic output) of the form:

ERROR [*filename*]: message text (code)

The code in parentheses may be used as an argument to the **help** command to obtain a further explanation of the diagnostic message. Detection of a fatal error during the processing of a file causes the SCCS command to stop processing that file and to proceed with the next file, in order, if more than one file has been named.

4.4 Temporary Files

Several SCCS commands generate temporary files and file copies during the process of creating, retrieving, and updating

[†] Described in Section 1 of the *UniPlus⁺ User Manual*.

SCCS

SCCS files. These temporary files are normally named by stripping off the 's.' prefix of the SCCS file name and replacing it with another single alphabetic character. The **g-file** is named by simply deleting the 's.' prefix. Thus, if the SCCS file is named 's.abc' the **g-file** will be named 'abc' and the **p-file** will be named 'p.abc'.

These files are as follows:

g-file This is the text file created by a **get** command. It contains a particular stripping version of an SCCS file, and its name is formed by stripping off the s. prefix from the SCCS file.

The **g-file** is created in the current directory and is owned by the real user. The mode assigned to the **g-file** depends on how the **get** command is invoked. The version it contains also depends on how the **get** command is invoked. The default version is the most recent 'trunk' delta (i.e., excluding branches).

d-file When you invoke a **get** command, SCCS creates its own temporary copy of the **g-file** by performing an internal **get** at the SID specified in the **p-file** entry. This temporary copy is called the **d-file**.

When you record your changes in a new version, the **delta** command compares the **d-file** to the **g-file** (using the **diff** command). † The differences between the **g-file** and the **d-file** are the changes that constitute the delta.

† Described in Section 1 of the *UniPlus User Manual*.

p-file When the `get -e` command creates a *g-file* with read-write permission (so you can edit it), it places certain information about the SCCS file (i.e., the SID of the retrieved version, the SID to be given to the new delta when it is created, and the login name of the user executing `get`) in another new file called the *p-file*.

When you record your changes in a new version, the `delta` command reads the *p-file* for the SID and the login name of the user creating the new delta.

When the new delta has been made, the *p-file* is updated by removing the relevant entry. If there is only one entry in the *p-file*, then the *p-file* itself is removed.

q-file Updates to the *p-file* are made to a temporary copy, the *q-file*, whose use is similar to the use of the *x-file*.

x-file All SCCS commands that modify an SCCS file do so by writing a temporary copy, called the *x-file* (to ensure that the SCCS file is not damaged if processing terminates abnormally.) When processing is complete, the old SCCS file is removed and the *x-file* is renamed (with the *s.* prefix) to be the SCCS file.

The *x-file* is created in the directory containing the SCCS file, given the same mode as the SCCS file, and owned by the effective user.

z-file To prevent simultaneous updates to an SCCS file, commands that modify SCCS files create a 'lock-file' called the *z-file*. This file exists only for the duration of the execution of the command that creates it. The *z-file* contains the process number of the command that creates it. While the *z-file* exists, it indicates to other commands that the SCCS file is being updated. SCCS commands that modify SCCS files will not process a file if the corresponding *z-file* exists.

The *z-file* is created with read-only mode (mode 444) (possibly modified by the user's `umask`), in the

SCCS

directory containing the SCCS file. It is owned by the effective user.

l-file The `get -l` command creates an *l-file* containing a table showing the deltas used in constructing a particular version of the SCCS file. This file is created in the current directory with mode 444 (read-only) and is owned by the real user.

In general, users can ignore *q-files*, *x-files* and *z-files*, although they can be useful in the event of system crashes or similar situations.

5. SCCS Command Summary

This section describes the major features of all the SCCS commands. Detailed descriptions of the commands and of all their arguments are given in the *UniPlus⁺ User Manual*. The discussion below covers only the more common arguments of the various SCCS commands. The relevant manual pages should be consulted for further information.

The SCCS commands are as follows:

admin	Creates SCCS files and applies changes to parameters of SCCS files.
cdc	Changes the commentary associated with a delta.
comb	Combines two or more consecutive deltas of an SCCS file into a single delta; often reduces the size of the SCCS file.
delta	Applies changes (deltas) to the text of SCCS files, i.e., creates new versions.
get	Retrieves versions of SCCS files.
unget	'Undoes' a get -e command if invoked before the new delta is created.
help	Prints explanations of diagnostic messages.
prs	Prints portions of an SCCS file in user specified format.
rmDEL	Removes a delta from an SCCS file; allows the removal of deltas that were created by mistake.
sact	Accounts for SCCS files in the process of being changed.
sccsdiff	Shows the differences between any two versions of an SCCS file.

SCCS

- val** Validates an SCCS file.
- what** Searches any UNIX system file(s) for all occurrences of a special pattern and prints out what follows it; is useful in finding identifying information inserted by the **get** command.

5.1 The **admin** Command

The **admin** command creates new SCCS files or (using keyletters) changes parameters of existing ones. See *admin(1)* in the *UniPlus⁺ User Manual* for a complete list of keyletters and flags to the **admin** command.

5.1.1 Creating SCCS Files

SCCS files are created in read-only mode (444) and are owned by the effective user (see the section on "Protection" above). Only a user with write permission in a directory containing SCCS files can use the **admin** command on a file in that directory.

An SCCS file is created with the command:

```
% admin -i $\textit{first}$   $\textit{s.abc}$ 
```

where *first* (the value of the **-i** keyletter) is a file from which the text of the initial delta of the SCCS file $\textit{s.abc}$ is to be taken.

If you omit the value of the **-i** keyletter, **admin** reads the standard input for the text of the initial delta. Thus, the command:

```
% admin -i  $\textit{s.abc}$  <  $\textit{first}$ 
```

is also valid. Only one SCCS file may be created at a time using the **-i** keyletter.

If the text of the initial delta does not contain ID keywords, the message:

```
    No id keywords (cm7)
```

SCCS

is issued as a warning. See the section on "ID Keywords" under the **get** command for more information.

If the invocation of the **admin** command sets the *i* flag (using the **-f** keyletter described in the section on "SCCS Flags" below) the above message is treated as a fatal error and the SCCS file is not created.

When an SCCS file is created, the release number assigned to its first delta is normally '1', and its level number is always '1'. Thus, the first delta of an SCCS file is normally '1.1'.

The **-r** keyletter to the **admin** command is used to specify a different release number for the initial delta. Because it is only meaningful in creating the first delta (with **admin**), its use is only permitted with the **-i** keyletter. The command:

```
% admin -ifirst -r3 s.abc
```

indicates that the first delta should be named '3.1' rather than '1.1'.

5.1.2 SCCS Flags

The flags of an SCCS file are used to direct certain actions of the various commands. See *admin(1)* in the *UniPlus+ User Manual* for a description of all the flags.

When you create an SCCS file, its flags are either initialized by the **-f** keyletter (which you supply on the command line), or assigned default values if no keyletters are supplied. The flags of an SCCS file are initialized or changed using the **-f** keyletter, and deleted using the **-d** keyletter.

The **-f** keyletter is used to set a flag and, possibly, to set its value.

For example, the command:

```
% admin -ifirst -fi -fmmodname s.abc
```

sets the **i** flag and the **m** (module name) flag. The **i** flag specifies that the warning message (stating that there are no ID keywords contained in the SCCS file) should be treated as an error. The value *modname* specified for the **m** flag is the value that the **get** command will use to replace the **sccs2** ID keyword. (When the **get** command retrieves a delta, it creates a text file called *g-file*, whose name is formed by stripping off the 's.' prefix on the SCCS file name. In the absence of the **m** flag, the name of the *g-file* is used as the replacement for the **sccs2** ID keyword.)

Note that several **-f** keyletters may be supplied on a single invocation of **admin** and that **-f** keyletters may be supplied whether the command is creating a new SCCS file or processing an existing one.

The **-d** keyletter is used to delete a flag from an SCCS file and may only be specified when processing an existing file. For example, the command:

```
% admin -dm s.abc
```

removes the **m** flag from the SCCS file. Several **-d** keyletters may be supplied on a single invocation of **admin** and may be intermixed with **-f** keyletters.

SCCS

SCCS files also contain a *user list* of login names and/or group IDs of users who are allowed to create deltas of that file. This is an important file parameter that is checked by several SCCS commands to ensure that the delta is authorized.

This list is empty by default, which means that anyone may create deltas. The **-a** keyletter is used to specify users who are given permission or denied permission to create deltas. Specifying a group ID is the equivalent of naming all login names common to that group ID. For example, the command:

```
% admin -a vz -a ram -a 1234 s.abc
```

gives permission to create deltas to the login names *vz* and *ram* and the group ID *1234*. The command:

```
% admin -a! vz s.abc
```

denies permission to create deltas to the login name *vz*. You can use the **-a** keyletter whether **admin** is creating a new SCCS file or processing an existing one, and it can appear several times on a command line.

Similarly, the **-e** keyletter is used to remove (erase) login names or group IDs from the list. For example:

```
% admin -e vz s.abc
```

removes the login name *vz* from the user list of the SCCS file '*s.abc*'.

5.1.3 Comments and MR Numbers

When an SCCS file is created, you may choose to insert comments stating your reasons for creating the file.

In a controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc., collectively called MRs. The creation of an SCCS file may sometimes be the direct result of an MR. They can be recorded by number in a delta via the **-m** keyletter, which can be supplied on the **admin** (or the **delta**) command line.

The **-y** keyletter can also be used to supply comments on the command line rather than through the standard input.

If comments (**-y** keyletter) are omitted, a comment line of the form:

date and time created YY/MM/DD HH:MM:SS by logname

is automatically generated.

If you want to supply an MR number (using the **-m** keyletter), the **v** flag must also be set (using the **-f** keyletter described below), as in the command:

```
% admin -i first -m mrlst -fv s.abc
```

The **v** flag simply causes the **delta** command to prompt for MR numbers as the reason for creating a delta. (See **sccsfile(4)** in the *UniPlus⁺ User Manual*.) Note that the **-y** and **-m** keyletters are only effective if a new SCCS file is being created.

SCCS

5.1.4 Descriptive Text

The portion of the SCCS file reserved for descriptive text can be initialized or changed using the **-t** keyletter. The descriptive text is intended as a summary of the contents and purpose of the SCCS file.

To insert descriptive text in a file you are creating, the **-t** keyletter is followed by the name of a file from which the descriptive text is to be taken. For example, when an SCCS file is being created, the command:

```
% admin -i first -t desc s.abc
```

specifies that the descriptive text is to be taken from a file named *desc*.

When processing an *existing* SCCS file, the **-t** keyletter specifies that the descriptive text (if any) currently in the file is to be replaced with the text in the named file. Thus:

```
% admin -t desc s.abc
```

specifies that the descriptive text of the SCCS file is to be replaced by the contents of *desc*. If you omit the file name after the **-t** keyletter as in:

```
% admin -t s.abc
```

the descriptive text currently in the SCCS file is removed.

5.2 The `cdc` Command

The `cdc` command changes the comments or MR numbers that were supplied when that delta was created. It is invoked as follows:

```
% cdc -r 3.4 s.abc
```

This specifies that you want to change the comments of delta 3.4 of the SCCS file `s.abc`.

The `cdc` command also allows you to delete selected MR numbers associated with the specified delta by preceding the selected MR numbers by the exclamation character '!'.

`Cdc` prompts for new comments and MR numbers:

```
% cdc -r 3.4 s.abc
```

```
MRs? mrlist !mrlist
```

```
comments? deleted wrong MR number and inserted  
correct MR number
```

The new MR number(s) in the first *mrlist* are inserted, and the old MR number(s) (preceded by the exclamation character) are deleted. The old comments are kept and preceded by a line indicating that they have been changed (i.e., superseded). The new comments are entered ahead of this comment line. The 'inserted' comment line records the login name of the user executing `cdc` and the time of its execution.

5.3 The **comb** Command

The **comb** command generates a shell script (see *csb(1)* in the *UniPlus+ User Manual*), which is written to standard output. When executed, the script attempts to reconstruct the named SCCS files to save space by discarding deltas that are no longer useful and combining other specified deltas.

- It is not recommended that **comb** be used as a matter of routine; its use should be restricted to a very small number of times in the life of an SCCS file. It is recommended that **comb** be run with the **-s** keyletter (in addition to any other keyletters desired) before any actual reconstructions.

In the absence of any keyletters, **comb** preserves only 'leaf' (most recent) deltas and the minimum number of ancestor deltas necessary to preserve the 'shape' of the SCCS file tree. The effect of this is to eliminate middle deltas on the trunk and on all branches of the tree. Thus, in Figure 10-4, deltas 1.2, 1.3.2.1, 1.4, and 2.1 would be eliminated.

Some of the keyletters are summarized as follows:

- p The **-p** keyletter specifies the oldest delta that is to be preserved in the reconstruction. All older deltas are discarded.
- c The **-c** keyletter specifies a list (see *get(1)* in the *UniPlus+ User Manual* for the syntax of this list) of deltas to be preserved. All other deltas are discarded.
- s The **-s** keyletter causes the generation of a shell script that, when run, produces only a report summarizing the percentage space (if any) to be saved by reconstructing each named SCCS file. It is recommended that **comb** be run with this keyletter (in addition to any others desired) before any actual reconstructions.

It should be noted that the shell script generated by **comb** is not guaranteed to save space. In fact, it is possible for the reconstructed file to be larger than the original. Note, too, that the shape of the SCCS file tree may be altered by the reconstruction process.

SCCS

5.4 The **delta** Command

When a version of an SCCS file is retrieved using the **get -e** command, it is copied into a text file called the *g-file*, which can then be edited.

The **delta** command is used to incorporate the changes made to a *g-file* into the corresponding SCCS file, i.e., to create a delta (a new version of the SCCS file).

Execution of the **delta** command requires the existence of a *p-file*. The **delta** command examines the *p-file* to verify the presence of an entry containing the user's login name and a valid SID for the next delta. If the user's login name is not found, an error message results, because the user who retrieved the *g-file* must be the same user who creates the delta. If the login name of the user appears in more than one entry in the *p-file*, the same user has executed more than one **get -e** on the same SCCS file. In this case the **-r** keyletter must then be used with **delta** to specify the SID that uniquely identifies the *p-file* entry. This entry is the one used to obtain the SID of the delta to be created.

The **delta** command also performs the same permission checks performed by **get -e**. If all checks are successful, **delta** determines what has been changed in the *g-file* (using the **diff** command on the new version and its own temporary copy of the *g-file* as it was before editing). This temporary copy of the *g-file* is called the *d-file* (its name is formed by replacing the 's.' of the SCCS file name with 'd.'). It is obtained by performing an internal **get** on the SID specified in the *p-file* entry.

In practice, the most common use of **delta** is:

```
% delta s.abc
```

which prompts on the standard output (but only if it is a terminal):

comments?

to which the user replies with a description of why the delta is being made. Your response may be up to 512 characters long if you escape all newlines with a backslash (\). The response is terminated by an unhidden newline character.

In a controlled environment, it is expected that deltas are created only as a result of some trouble report, change request, trouble ticket, etc., collectively called MRs. It is desirable (or necessary) to record such MR number(s) within each delta. If the SCCS file has a **v** flag set, **delta** first prompts with:

MRs? (Modification Requests)

on the standard output. (Again, this prompt is printed only if the standard output is a terminal.) The standard input is then read for MR numbers, separated by blanks and/or tabs. Your response may be up to 512 characters long if you escape all newlines with a backslash (\). The response is terminated by an unhidden newline character.

The **-y** and/or **-m** keyletters on the **delta** command line may be used to supply comments and MR numbers, respectively, instead of supplying these through the standard input. The format of the **delta** command is then:

```
% delta -ydescriptive comment -mmrlist s.abc
```

In this case, the corresponding prompts are not printed, and the standard input is not read. The **-m** keyletter is allowed only if the SCCS file has a **v** flag. These keyletters are useful when **delta** is executed from within a shell script (see *csk*(1) in the

SCCS

UniPlus⁺ User Manual.

The comments and/or MR numbers, whether prompted for by **delta** or supplied via keyletters, are recorded as part of the entry for the delta being created, and apply to all SCCS files processed by the same invocation of **delta**. This implies that (if **delta** is invoked with more than one file argument and the first file named has a **v** flag) all files named must have the **v** flag. Similarly, if the first file named does not have this flag, then none of the files named may have it. Any file that does not conform to these rules is not processed.

When processing is complete, the SID of the created delta (obtained from the **p-file** entry) and the counts of lines inserted, deleted, and left unchanged by the delta, are written to the standard output.

Thus, a typical output might be:

```
1.4
14 inserted
7 deleted
345 unchanged
```

It is possible that the counts of lines reported as inserted, deleted, or unchanged by **delta** do not agree with the user's perception of the changes applied to the **g-file**. The reason for this is that there usually are a number of ways to describe a set of changes, especially if lines are moved around in the **g-file**, and **delta** is likely to find a description that differs from the user's perception. However, the total number of lines of the new delta (the number inserted plus the number left unchanged) should agree with the number of lines in the edited **g-file**.

If (in the process of making a delta) **delta** finds no ID keywords in the edited *g-file*, the message:

No id keywords (cm7)

is issued after the prompts for comments but before any other output. This indicates that any ID keywords that may have existed in the SCCS file have been replaced by their values or deleted during the editing process. This could be caused by:

- Creating a delta from a *g-file* that was created by a **get** command without the **-e** keyletter (ID keywords are replaced by **get** in that case).
- Accidentally deleting or changing the ID keywords while you are editing the *g-file*.
- The file had no ID keywords to begin with.

In any case, it is left up to the user to determine what to do about it. The delta is created whether or not ID keywords are present, unless there is an **l** flag in the SCCS file indicating that this should be treated as a fatal error. In this last case, the delta is not created until the ID keywords are inserted in the *g-file* and the **delta** command is executed again.

After the processing of an SCCS file is complete, the corresponding *p-file* entry is removed from the *p-file*. All updates to the *p-file* are made to a temporary copy called the *q-file*. If there is only one entry in the *p-file*, then the *p-file* itself is removed.

Once processing of the corresponding SCCS file is complete, **delta** also removes the edited *g-file* unless the **-n** keyletter is specified. Thus:

SCCS

% **delta** -n s.abc

keeps the *g-file* upon completion of processing.

The -s keyletter suppresses all output that is normally directed to the standard output except for the prompts 'comments?' and 'MRs?'. Use of the -s keyletter together with the -y keyletter (and possibly, the -m keyletter) causes **delta** neither to read standard input nor to write to standard output.

The differences between the *g-file* and the *d-file* (see above), are the changes that constitute the delta. These may be printed on the standard output by using the -p keyletter. The format of this output is similar to that produced by **diff**.

5.5 The get Command

The **get** command creates a text file that contains a particular version of an SCCS file. The particular version is retrieved by beginning with the initial version and then applying deltas, in order, until the desired version is obtained. The created file is called the *g-file*.

The *g-file* name is formed by removing the 's.' from the SCCS file name. The *g-file* is created in the current directory and is owned by the real user. The mode assigned to the *g-file* depends on how the **get** command is invoked.

The simplest invocation of **get** is:

```
% get s.abc
```

which retrieves the latest "ancestor" (version) of the SCCS file tree (i.e., excluding branches) and produces (for example) on the standard output:

```
1.3
67 lines
No id keywords (cm7)
```

This indicates that:

1. Version 1.3 of file *s.abc* was retrieved (1.3 is the latest delta).
2. This version has 67 lines of text.
3. No ID keywords were substituted in the file.

The generated *g-file* (file 'abc') is given mode 444 (read-only). This particular way of invoking **get** is intended to produce *g-files* only for inspection, compilation, etc. It is not intended for

SCCS

editing (i.e., not for making deltas).

When several file arguments (or directory-name arguments) are given on the `get` command line, similar information is displayed for each file processed, but the SCCS file name precedes it. For example, the command:

```
% get s.abc s.def
```

produces:

```
s.abc:  
1.3  
67 lines  
No id keywords (cm7)
```

```
s.def:  
1.7  
85 lines  
No id keywords (cm7)
```

5.5.1 ID Keywords

In generating a *g-file* to be used for compilation, it is useful to record the date and time of creation, the version retrieved, the module's name, etc., within the *g-file*. This information appears in a load module when one is eventually created.

SCCS provides a mechanism for recording such information about deltas automatically, using Identification (ID) keywords. Identification (ID) keywords can appear anywhere in the generated file, and will be replaced by appropriate values according to the definitions of these ID keywords.

The format of an ID keyword is an uppercase letter enclosed by percent signs (%). When these appear in the generated SCCS

file they are replaced by appropriate values according to the definitions of these ID keywords.

For example:

`%I%`

is defined as the ID keyword that is replaced by the SID of the retrieved version of a file. Similarly:

`%H%`

is defined as the ID keyword for the current date (in the form 'mm/dd/yy').

When no ID keywords are substituted by `get`, the following message is issued:

No id keywords (cm7)

This message is normally treated as a warning by `get`, although the presence of the `I` flag in the SCCS file causes it to be treated as an error.

For a complete list of the approximately 20 ID keywords, see `get(1)` in the *UniPlus⁺ User Manual*.

5.5.2 Retrieving Different Versions

Normally, the default version of an SCCS file is the most recent delta of the highest-numbered release on the basic 'trunk' of the SCCS file tree (exclusive of branches). However, if the SCCS file being processed has a `d` (default SID) flag, the SID specified as the value of this flag is used as a default. The default SID is interpreted in exactly the same way as the value supplied with the `-r` keyletter of `get`.

SCCS

Various keyletters allow the retrieval of other than the default version of an SCCS file. When these keyletters are used on the **get** command line, the **d** (default SID) flag (if any) is ignored.

The **-r** keyletter is used to specify which SID you want to retrieve. For example:

```
% get -r 1.3 s.abc
```

retrieves version 1.3 of file *s.abc* and produces (for example) on the standard output:

```
1.3  
64 lines
```

A branch delta can be retrieved similarly:

```
% get -r 1.5.2.3 s.abc
```

which produces (for example) on the standard output:

```
1.5.2.3  
234 lines
```

When a 2- or 4-component SID is specified as a value for the **-r** keyletter (as above) and the particular version does not exist in the SCCS file, an error message results.

If you omit the level number:

```
% get -r 3 s.abc
```

the highest level number (most recent delta) within the given release will be retrieved, if the given release exists. Thus, the

above command might output:

```
3.7  
213 lines
```

If the given release does not exist, `get` retrieves the most recent 'trunk' delta (not in a branch) with the highest level number within the highest-numbered existing release that is lower than the release you specify. For example, assuming release 9 does not exist in file `s.abc` and that release 7 is actually the highest-numbered release below 9, the command:

```
% get -r 9 s.abc
```

might produce:

```
7.6  
420 lines
```

which indicates that delta 7.6 is the latest version of file `s.abc` that is not in a branch and that is below release 9. Similarly, omission of the sequence number, as in:

```
% get -r 4.3.2 s.abc
```

results in the retrieval of the branch delta with the highest sequence number on the given branch if it exists. (If the given branch does not exist, an error message results.) This might result in the following output:

```
4.3.2.8  
89 lines
```

The `-t` keyletter is used to retrieve the latest (top) version in a particular release (i.e., when no `-r` keyletter is supplied or when its value is simply a release number). The latest version is

SCCS

defined as that delta which was produced most recently, independent of its location on the SCCS file tree. Thus, if the most recent delta in release 3 is 3.5,

```
% get -r 3 -t s.abc
```

might produce:

```
3.5  
59 lines
```

However, if branch delta 3.2.1.5 were the latest delta (created after delta 3.5), the same command might produce

```
3.2.1.5  
46 lines
```

5.5.3 Retrieving Files to Make Deltas

When you specify the **-e** keyletter to the **get** command, the resulting **g-file** has read-write permission and can be edited to make a new delta. Consequently, the use of the **-e** keyletter is restricted.

The presence of the **-e** keyletter causes **get** to check the following SCCS protection devices (see the section on "Protection" above).

1. The login name or group ID of the user executing **get** must be on the *user list* (a list of login names and/or group IDs of users allowed to make deltas). A null (empty) user list behaves as if it contained all possible login names.
2. The release of the version being retrieved must be greater than or equal to the **f** flag (*floor*) and lower than or equal to the **c** flag (*ceiling*). These flags are specified in the SCCS file; they have default values of 1 and 9999,

respectively.

3. The release is not locked against editing. The 'lock' is specified as a flag in the SCCS file.
4. Whether or not multiple concurrent edits are allowed for the SCCS file as specified by the *j* flag in the SCCS file.

Failure of any of the first three conditions causes the processing of the corresponding SCCS file to terminate.

If the above checks succeed, *get -e* creates a *g-file* in the current directory with mode 644 (readable by everyone, writable only by the owner). This is possibly modified by the user's *umask*. The *g-file* is owned by the real user. If a writable *g-file* already exists, *get* terminates with an error. This is to prevent inadvertent destruction of a *g-file* that already exists and is being edited for the purpose of making a delta.

Any ID keywords appearing in the *g-file* are not substituted by *get* (when the *-e* keyletter is specified) because the generated *g-file* is subsequently used to create another delta. Replacement of ID keywords causes them to be permanently changed within the SCCS file. In view of this, *get* does not need to check for the presence of ID keywords within the *g-file*, so the message

No id keywords (cm7)

is never output when *get* is invoked with the *-e* keyletter.

In addition, the *-e* keyletter causes the creation (or updating) of a *p-file* which is used to pass information to the *delta* command.

SCCS

The following is an example of the use of the `-e` keyletter:

```
% get -e s.abc
```

which produces (for example) on the standard output:

```
1.3  
new delta 1.4  
67 lines
```

The `-r` and/or `-t` keyletters can be used together with the `-e` keyletter to specify a particular version to be retrieved for editing.

When you want to change the release number of a new delta, specify the new release number on the command line. For example, if the most recent delta of a particular SCCS file is 1.5 and you want to create delta 2.1, use the command:

```
% get -e -r2 s.lang
```

Because release 2 does not yet exist, `get` retrieves the latest version *before* release 2 and changes the release number of the next delta to 2, naming it 2.1 rather than 1.6. This information is stored in the *p-file* so the next execution of the `delta` command will produce a delta with the new release number. The `get` command then outputs (for example):

```
1.2  
new delta 2.1  
7 lines
```

Subsequent versions of the file will be created in release 2 (deltas 2.2, 2.3, etc.).

The **-i** and **-x** keyletters can be used to specify a list of deltas to be *included* and *excluded* by **get**. (See **get(1)** in the *UniPlus+ User Manual* for the syntax of this list.) *Including* a delta means forcing the changes that constitute the particular delta to be included in the retrieved version. This is useful if you want to apply the same changes to more than one version of the SCCS file. *Excluding* a delta means forcing it not to be applied. This may be used to undo (in the version of the SCCS file to be created) the effects of a previous delta.

Whenever deltas are included or excluded, **get** checks for possible interference between those deltas and deltas that are normally used in retrieving the particular version of the SCCS file. Two deltas can interfere, for example, when each one changes the same line of the retrieved *g-file*. Any interference is indicated by a warning that shows the range of lines within the retrieved *g-file* in which the problem may exist. The user is expected to examine the *g-file* to determine whether a problem actually exists, and to do whatever is necessary (e.g., edit the file).

Warning: The **-i** and **-x** keyletters should be used with extreme care.

The **-k** keyletter facilitates regeneration of a *g-file* that may have been accidentally removed or ruined after a **get -e** command, or to simply generate a *g-file* in which the replacement of ID keywords has been suppressed.

A *g-file* generated by the **-k** keyletter is identical to one produced by **get -e**, except that no processing related to the *p-file* takes place.

5.5.4 Concurrent Edits of Different SIDs

The ability to retrieve different versions of an SCCS file allows a number of deltas to be 'in progress' at the same time. This

SCCS

means that a number of **get -e** commands can be executed on the same file. However, unless multiple concurrent edits are explicitly allowed, (see the section on "Concurrent Edits of Same SID" below) no two **get -e** executions can retrieve the same version of an SCCS file.

The **p-file** is created by the **get** command (with **-e**) and is named by replacing the 's.' in the SCCS file name with 'p.'. It is created in the directory containing the SCCS file, given mode 644 (readable by everyone, writable only by the owner), and owned by the effective user.

The **p-file** contains the following information for each delta that is still 'in progress':

- The SID of the retrieved version.
- The SID to be given to the new delta when it is created.
- The login name of the real user executing **get**.

The first execution of **get -e** causes the creation of the **p-file** for the corresponding SCCS file. Subsequent executions only update the **p-file** with a line containing the above information. Before updating, however, **get** checks to assure that no entry (already in the **p-file**) specifies that the SID (of the version to be retrieved) is already retrieved (unless multiple concurrent edits are allowed. See the section on "Concurrent Edits of Same SID" below.)

If both checks succeed, the user is informed that other deltas are in progress and processing continues. If either check fails, an error message results. It is important to note that the various executions of **get** should be carried out from different directories. Otherwise, only the first execution succeeds since subsequent executions would attempt to overwrite a writable **g-file**, which is an SCCS error condition. In practice, such

multiple executions are performed by different users so that this problem does not arise since each user normally has a different working directory. (See the section on "Protection" above for a discussion about how different users are permitted to use SCCS commands on the same files.)

Figure 10-5 shows examples of the version of an SCCS file retrieved by **get**, as well as the SID of the version to be eventually created by **delta**, as a function of the SID specified to **get**.

SID SPECIFIED*	-b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
none†	no	R default to mR	mR.mL	mR.(mL+1)
none‡	yes	R default to mR	mR.mL	mR.mL.(mB+1)
R	no	R > mR	mR.mL	R.1§
R	no	R == mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R == mR	mR.mL	mR.mL.(mB+1).1
R	—	R < mR		
R	—	R < mR and does not exist	hR.mL**	hR.mL.(mB+1).1
R	—	Trunk successor in release > R and R exists	R.mL	R.mL.(mB+1).1

See footnotes on sheet 3 of 3.

Figure 10-5. Determination of New SID (Sheet 1 of 3)

SID SPECIFIED*	-b KEY-LETTER USED†	OTHER CONDITIONS	SID RETRIEVED	SID OF DELTA TO BE CREATED
R.L.	no	No trunk successor	R.L.	R.(L+1)
R.L.	yes	No trunks successor	R.L.	R.L.(mB+1).1
R.L.	—	Trunk in release > = R	R.L.	R.L.(mS+1).1
R.L.b	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	no	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	—	Branch successor	R.L.B.S	R.L.(mB+1).1

See footnotes on sheet 3 of 3.

Figure 10-5. Determination of New SID (Sheet 2 of 3)

Footnotes:

* **R**, **L**, **B**, and **S** are release, level, branch, and sequence components of the SID. **m** means **maximum**. Thus, for example, **R.mL** means 'the maximum level number within release R'; **R.L.(mB+1).1** means 'the first sequence number on the (maximum branch number plus 1) of level L within release R'.

Also note that if the SID specified is of the form **R.L**, **R.L.B**, or **R.L.B.S**, each of the specified components must exist.

† The **-b** keyletter is effective only if the **b** flag is present in the file (see *admin(1)*). In this state, an entry of **-i** means *irrelevant*.

‡ This case applies if the **d** (default SID) flag is not present in the file. If the **d** flag is present in the file, the SID obtained from the **d** flag is interrupted as if it had been specified on the command line. Thus, one of the other cases in this figure applies.

§ This case is used to force the creation of the first delta in the new release.

** **hR** is the highest existing release that is lower than the specified, nonexistent, release **R**.

Figure 10-5. Determination of New SID (Sheet 3 of 3)

5.5.5 Concurrent Edits of Same SID

Under normal conditions, **get -e** commands are not permitted to occur concurrently on the same SID. That is, **delta** must be executed before another **get -e** is executed on the same SID.

However, if the **j** flag is set in the SCCS file, multiple concurrent edits (two or more successive executions of **get -e** on the same SID) are allowed.

Thus, the command:

```
% get -e s.abc
    1.1
    new delta 1.2
    5 lines
```

may be immediately followed by:

```
% get -e s.abc
    1.1
    new delta 1.1.1.1
    5 lines
```

without an intervening execution of **delta**. In this case, a **delta** command corresponding to the first **get** produces delta 1.2 (assuming 1.1 is the latest (most recent) delta), and the **delta** command corresponding to the second **get** produces delta 1.1.1.1.

5.5.6 Keyletters That Affect Output

See *get(1)* in the *UniPlus⁺ User Manual* for a full description of **get** keyletters.

SCCS

When you specify the **-p** keyletter to the **get** command, the retrieved text is written on standard output rather than a **g-file**. In this case, all output normally directed to the standard output (such as the SID of the version retrieved and the number of lines retrieved) is directed instead to the standard error output.

The **-p** keyletter is used, for example, to create **g-files** with arbitrary names:

```
% get -p s.abc > filename
```

The **-s** keyletter suppresses all output that is normally directed to the standard output (the SID of the retrieved version, the number of lines retrieved, etc., are not written). This does not affect messages to the standard error output.

This keyletter is used to prevent nondiagnostic messages from appearing on the user's terminal, and is often used in conjunction with the **-p** keyletter to 'pipe' the output of **get**. For example:

```
% get -p -s s.abc | nroff
```

The **-g** suppresses the actual retrieval of the text of a version of the SCCS file. This can be used in a number of ways, for example, to verify the existence of a particular SID in an SCCS file:

```
% get -g -r 4.3 s.abc
```

This outputs the given SID if it exists in the SCCS file, or generates an error message if it does not exist.

The **-g** keyletter is also used to regenerate a *p-file* that has been accidentally destroyed.

```
% get -e -g s.abc
```

The **-l** keyletter creates an *l-file* (named by replacing the 's.' of the SCCS file name with 'l.'). This file is created in the current directory with mode 444 (read-only) and is owned by the real user. It contains a table showing the deltas used in constructing a particular version of the SCCS file. (See *get(1)* in the *UniPlus+ User Manual* for a description of the format of this table)

For example, the command:

```
% get -r 2.3 -l s.abc
```

generates an *l-file* that shows the deltas applied to retrieve version 2.3 of the SCCS file.

Specifying a value of **p** with the **-l** keyletter:

```
% get -lp -r 2.3 s.abc
```

causes the generated output to be written to the standard output rather than to the *l-file*.

You can use the **-g** keyletter with the **-l** keyletter to suppress the actual retrieval of the text.

SCCS

The **-m** keyletter is used to identify the changes applied to an SCCS file, line by line. When you specify this keyletter to the **get** command, each line of the generated **g-file** is preceded by the SID of the delta that caused that line to be inserted. The SID is separated from the text of the line by a tab character.

The **-n** keyletter causes each line of the generated **g-file** to be preceded by the value of the **%M%** ID keyword (the module name) and a tab character. The **-n** keyletter is most often used in a pipeline with **grep**.† For example, the following command searches the latest version of each SCCS file in a directory for all lines that match a given pattern:

```
% get -p -n -s directory | grep pattern
```

If both the **-m** and **-n** keyletters are specified, each line of the generated **g-file** is preceded by the value of the **scs1** ID keyword and a tab (caused by the **-n** keyletter) and shown in the format produced by the **-m** keyletter.

Because the contents of the **g-file** are modified when you use the **-m** and/or **-n** keyletters, this **g-file** cannot be used for creating a delta, and neither **-m** nor **-n** can be used with the **-e** keyletter.

† Described in Section 1 of the *UniPlus ' User Manual*.

5.6 The unget Command

The **unget** command 'undoes' a **get -e** command if it is invoked before the **delta** command is executed. There are three keyletters that can be used with **unget**:

- rSID** Uniquely identifies the delta that is no longer intended (included in the **p-file**). This is only necessary if two or more **get -e** commands of the same SCCS file are in progress.
- s** Suppresses the display of the intended SID of the delta on standard output.
- n** Retains the **g-file** in the current directory instead of removing it.

For example, the command:

```
% get -e s.abc
```

can be followed by:

```
% unget s.abc
```

even if the *abc* file has been edited. Invoking **unget** before **delta** will cause the last version to be unchanged.

5.7 The help Command

The **help** command prints explanations of SCCS commands and of messages that these commands may print. Arguments to **help**, zero or more of which may be supplied, are simply the names of SCCS commands or the code numbers that appear in parentheses after SCCS messages. If no argument is given, **help** prompts for one. The **help** command has no concept of keyletter arguments or file arguments. Explanatory information related to an argument, if it exists, is printed on the standard output. If no information is found, an error message is printed. Note that each argument is processed independently, and an error resulting from one argument will not terminate the processing of the other arguments.

Explanatory information related to a command is a synopsis of the command. For example, the command:

```
% help ge5 rmdel
```

produces:

```
ge5:  
'nonexistent sid'  
The specified sid does not exist in the  
given file.  
Check for typos.  
  
rmdel:  
rmdel -rSID name ...
```

5.8 The prs Command

The **prs** command is used to print on the standard output all or part(s) of an SCCS file in a format specified by the user. The specific format is called the output *data specification*; it is supplied via the **-d** keyletter on the **prs** command line.

The data specification is a string consisting of SCCS file data keywords (not to be confused with **get** ID keywords). These keywords can (optionally) be interspersed with text.

Data keywords specify which parts of an SCCS file are to be retrieved and output. All parts of an SCCS file (see *sccsfile(4)*) have an associated data keyword. Data keywords are strings (usually an uppercase character, two uppercase characters, or an upper- and lowercase character) enclosed by colons. For example:

:I:

is defined as the data keyword that is replaced by the SID of a specified delta. Similarly, **:F:** is defined as the data keyword for the SCCS file name currently being processed, and **:C:** is defined as the comment line associated with a specified delta. For a complete list of the data keywords, see *prs(1)* in the *UniPlus⁺ User Manual*.

There is no limit to the number of times a data keyword can appear in a data specification. For example, the command:

```
% prs -d":I: this is the top delta for :F: :I:" s.abc
```

may produce on the standard output:

```
2.1 this is the top delta for s.abc 2.1
```


SCCS

Information can be obtained from a single delta by specifying the SID of that delta using the **-r** keyletter. For example:

```
% prs -d":F:::I: comment line is: :C:" -r 1.4 s.abc
```

may produce the following output:

```
s.abc: 1.4 comment line is: THIS IS A COMMENT
```

If the **-r** keyletter is not specified, the value of the SID defaults to the most recently created delta.

In addition, information from a range of deltas may be obtained by specifying the **-e** or **-l** keyletters.

The **-e** keyletter substitutes data keywords for the SID designated by the **-r** keyletter and all earlier deltas.

The **-l** keyletter substitutes data keywords for the SID designated by the **-r** keyletter and all later deltas. Thus, the command:

```
% prs -d :I: -r 1.4 -e s.abc
```

may output:

```
1.4  
1.3  
1.2.1.1  
1.2  
1.1
```

and the command:

% prs -d :l: -r 1.4 -l s.abc

may produce:

3.3
3.2
3.1
2.2.1.1
2.2
2.1
1.4

Substitution of data keywords for all deltas of the SCCS file may be obtained by specifying both the -e and -l keyletters.

SCCS

5.9 The **rmidel** Command

The **rmidel** command allows you to remove a delta from an SCCS file. It should normally only be used when incorrect global changes were made a part of the delta to be removed.

The delta to be removed must be a 'leaf' delta, that is, it must be the most recently created delta on its 'branch' or on the 'trunk' of the SCCS file tree. In Figure 10-4, only deltas 1.3.1.2, 1.3.2.2, and 2.2 can be removed; once they are removed, then deltas 1.3.2.1 and 2.1 can be removed, etc.

To be allowed to remove a delta, the effective user must have write permission in the directory containing the SCCS file. In addition, the real user must either be the one who created the delta being removed or be the owner of the SCCS file and its directory.

The **-r** keyletter, which is mandatory, is used to specify the complete SID of the delta to be removed (i.e., it must have two components for a 'trunk' delta and four components for a 'branch' delta). The command:

```
% rmidel -r 2.3 s.abc
```

specifies the removal of delta 2.3 of the SCCS file. Before removing the delta, **rmidel** checks that the release number (R) of the given SID satisfies the relation:

$$\text{floor} \leq R \leq \text{ceiling}$$

The **rmidel** command also checks that the SID specified is not that of a version for which a **get -e** is 'in progress', i.e., has been executed and whose associated delta has not yet been made.

In addition, the login name or group ID of the user must appear in the file's *user list* (or the *user list* must be empty). Also, the release specified cannot be locked against editing. (See the section on "Protection" for an explanation of these checks; and *admin(1)* in the *UniPlus+ User Manual*.)

If these conditions are not satisfied, processing is terminated, and the delta is not removed. After the specified delta has been removed, its type indicator in the 'delta table' of the SCCS file is changed from **D** ('delta') to **R** ('removed').

SCCS

5.10 The **sact** Command

The **sact** command accounts for the SCCS files that are in the process of being changed (i.e., that have been retrieved by a **get -e** command and have not been recorded by a **delta** command yet). There are five fields reported for each named file:

- Field 1 Specifies the SID of the existing SCCS file being changed.
- Field 2 Specifies the SID of the new delta to be created.
- Field 3 Contains the login name of the user who executed the **get -e** command.
- Field 4 Contains the date that the **get -e** command was executed.
- Field 5 Contains the time that the **get -e** command was executed.

The command:

```
% sact s.abc
```

produces a display such as:

```
1.2 1.3 john 85/06/20 16:15:15
```

5.11 The **sccsdiff** Command

The **sccsdiff** command determines (and prints on the standard output) the differences between two specified versions of one or more SCCS files. The versions to be compared are specified using the **-r** keyletter, in the same format used for the **get** command.

The two versions must be specified as the first two arguments to this command in the order they were created, i.e., the older version is specified first. Any following keyletters are interpreted as arguments to the **pr** command (which actually prints the differences)[†] and must appear before any file names. The SCCS files to be processed are named last. Directory names and a name of '-' (a single minus sign) are not acceptable to **sccsdiff**.

The differences are printed in the form generated by **diff(1)**. The following is an example of invoking **sccsdiff**:

```
% sccsdiff -r 3.4 -r 5.6 s.abc
```

[†] Described in Section 1 of the *UniPlus User Manual*.

5.12 The val Command

The **val** command is used to determine if a file is an SCCS file meeting the characteristics specified by an optional list of keyletter arguments. Any characteristics not met are considered errors.

The **val** command checks for the existence of a particular delta when the SID for that delta is explicitly specified via the **-r** keyletter. The string following the **-y** or **-m** keyletter is used to check the value set by the **t** or **m** flag, respectively (see *admin(1)* in the *UniPlus⁺ User Manual* for a description of the flags).

The **val** command treats the special argument **'-'** differently from other SCCS commands. This argument allows **val** to read the argument list from the standard input as opposed to obtaining it from the command line. The standard input is read until end of file.

This capability allows for one invocation of **val** with different values for the keyletter and file arguments. For example,

```
% val -
-y c -m abc s.abc
-m xyz -y pl1 s.xyz
(EOF)
```

first checks if the *s.abc* file has a value *c* for its 'type' flag and value *abc* for the 'module name' flag. Once processing of the first file is completed, **val** then processes the remaining files, in this case, *s.xyz*, to determine if they meet the characteristics specified by the keyletter arguments associated with them.

The **val** command returns an 8-bit code; each bit set indicates the occurrence of a specific error (see *val(1)* for a description of possible errors and the codes). The appropriate diagnostic is also printed unless suppressed by the **-s** keyletter. A return code of '0' indicates all named files met the characteristics specified.

5.13 The **what** Command

The **what** command is used to find identifying information within any UNIX system file whose name is given as an argument to **what**. Directory names and a name of '-' (a single minus sign) are not treated specially as they are by other SCCS commands and no keyletters are accepted by the command.

The **what** command searches the given file(s) for all occurrences of the string '@(#)', which is the replacement for the @(#) ID keyword (see *get(1)* in the *UniPlus⁺ User Manual*), and prints (on the standard output) the balance following that string until the first double quote ("), greater than (>), backslash (\), newline, or (nonprinting) NUL character. For example, if the SCCS file *s.prog.c* (a C language program) contains the following line:

```
char id[] = "@(#)scs2:5.1";
```

The command:

```
% get -r 3.4 s.prog.c
```

is executed, the resulting *g-file* is compiled to produce *prog.o* and *a.out*. Then the command:

```
% what prog.c prog.o a.out
```

produces:

SCCS

```
prog.c:  
  prog.c:3.4  
prog.o:  
  prog.c:3.4  
a.out:  
  prog.c:3.4
```

The string searched for by **what** does not need to be inserted in the SCCS file via an ID keyword of **get**; it can be inserted in any convenient way.

Chapter 11: CURSES

CONTENTS

1. Introduction	1
2. General Usage	2
2.1 Example Program — SCATTER	4
3. Initialization Routines	7
4. Structure Routines	9
5. Mini-Curses	11
6. Option-Setting Routines	14
7. Output Routines	18
7.1 Delays	24
8. Window-Manipulation Routines	26
8.1 Multiple Windows	29
8.2 Example Program — WINDOW	31
9. Terminal Mode-Setting Routines	34
9.1 TTY Mode Functions	37
9.2 Video Attributes	39
9.3 Special Keys	43
9.4 Scrolling Region	44
9.5 Highlighting	45
9.6 Example Program — HIGHLIGHT	47
10. Multiple Terminals	50
10.1 Current Terminal	53
10.2 Example Program — TWO	54
10.3 Additional Terminals	59
11. Low Level Terminfo Usage	60
11.1 Terminfo Level Routines	62
11.2 Example Program — TERMHL	68
12. Input	74
12.1 Example Program — SHOW	78

13. Portability	82
13.1 Portability Functions	82
14. Example Program — EDITOR	84

LIST OF FIGURES

Figure 11.1. Framework of a Curses Program	2
Figure 11.2. Use of Attributes	46
Figure 11.3. Using Multiple Terminals for Output	54
Figure 11.4. Terminfo level framework	61
Figure 11.5. Function Keys Returned by getch() (1 of 2)	77
Figure 11.6. Function Keys Returned by getch() (2 of 2)	78

Chapter 11

CURSES

1. Introduction

This chapter is an introduction to **curses(3X)**. It is intended for the programmer who must write a screen-oriented program using the **curses** package. This chapter documents each **curses** function, supplies several examples and is intended as a reference.

Some information is also included on **terminfo(4)**, since the **curses** program uses **terminfo** to set its parameters for the correct terminal screen type.

CURSES

2. General Usage

A **curses** program follows the framework shown in the following figure (Figure 11.1):

```
#include <curses.h>

...
    initscr(); /* Initialization */

    cbreak(); /* Various optional mode settings */
    nonl();
    noecho();

...
    while (!done) { /* Main body of program */
        ...
        /* Sample calls to draw on screen */
        move(row, col);
        addch(ch);
       printw("Formatted print with value %d\n", value);
        ...
        /* Flush output */
        refresh();
        ...
    }

    endwin(); /* Clean up */
    exit(0);
```

Figure 11.1. Framework of a Curses Program

The first routine called in this example is **initscr()**. A **curses** program ALWAYS begins with a call to this routine.

The section which is commented “/* optional mode settings */” contains some, not all, of the possible mode settings.

The mode settings given here are:

- **cbreak()**
This setting enables characters typed by the user to become immediately available to the program.
- **nonl()**
This setting which disables the translation of `<return>` to "newline" on input, thus enabling **curses** to make better use of the "linefeed" capability, maximizing the speed of cursor motion.
- **noecho()**
This setting turns off the immediate echoing of characters as they are typed.

For the complete list of mode settings, see the section of this document entitled, **Terminal Mode Setting**.

It is important to note that in spite of the ensuing functions **addch()** and **printw()**, which work much like **putchar()** and **printf()**, no output is actually displayed on the screen until the **refresh** routine is called.

The reason for this is that **move()**, **addch()** and the other routines controlling movement and drawing on a screen, send their first output to the **stdscr** (standard screen) data structure. This data structure, called a **window**, buffers the curses output until **refresh()** is called.

The **curses** program keeps track of what is on the physical screen as well as what is being saved in **stdscr**. When **refresh** is called, the two **windows** are compared and the physical screen is made to look like the one created in **stdscr** using the fewest possible characters. This function, which takes into account the capabilities of the terminal as well as the similarities of the screens, is called **cursor optimization**, and is the source for the name of the **curses** package.

CURSES

Two important facts to note are that, first, due to the hardware scrolling terminals, writing to the lower righthand corner character position is NOT possible with **curses**.

Secondly, some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not.

The variables **LINES** and **COLS** are defined by **initscr()** which initializes the current screen size with information gathered from **terminfo**. When writing a **curses** program, making use of these variables will help make the code more generally useful.

2.1 Example Program – SCATTER

Following is the first example program, “**SCATTER**,” which reads a file and prints the characters of that file on the screen in random order.

```

/*
 * SCATTER. This program takes the first
 * 23 lines from the standard
 * input and displays them on the
 * CRT screen, in a random manner.
 */

#include < curses.h>

#define MAXLINES 120
#define MAXCOLS 160
char s[MAXLINES][MAXCOLS]; /* Screen Array */

main()
{
    register int row=0,col=0;
    register char c;
    int char_count=0;
    long t;
    char buf[BUFSIZ];

    initscr();
    for(row=0;row<MAXLINES;row++)
        for(col=0;col<MAXCOLS;col++)
            s[row][col]=' ';

    row = 0;
    /* Read screen in */
    while( (c=getchar()) != EOF && row < LINES ) {
        if(c != '\n') {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')
                char_count++;
        } else {
            col=0;
            row++;
        }
    }
}

```


CURSES

```
time(&t);    /* Seed the random number generator */
srand((int)(t&0177777L));

while(char_count) {
    row=rand() % LINES;
    col=(rand() >> 2) % COLS;
    if(s[row][col] != ' ')
    {
        move(row, col);
        addch(s[row][col]);
        s[row][col]=EOF;
        char_count--;
        refresh();
    }
}
endwin();
exit(0);
}
```

3. Initialization Routines

These functions are called when initializing a program:

initscr()

- Almost always the first function call in a **curses** program.
- Determines the terminal type and initializes **curses** data structures.
- Arranges that the first call to **refresh** will clear the screen.

endwin()

- Should be called before exiting a **curses** program.
- Restores tty modes, moves the cursor to the lower left corner of the screen, resets the terminal into the proper non-visual mode, and tears down all appropriate data structures.

newterm(a,b)

- Should be used instead of **initscr** in **curses** programs which address more than one terminal.
- Takes 2 arguments:
 - a The first argument is a string representing the **type of terminal**.
 - b The second argument is a **file descriptor** for output to the terminal.
- The **file descriptor** should be open for reading AND writing to receive input from the terminal.
- Should be called once for each terminal.
- Returns a pointer to a **SCREEN** data structure as a reference to that terminal.
- Should call **endwin** for each terminal.
- If an error occurs, the **NULL** value is returned.

CURSES

set_term(a)

- Used to switch to a different terminal.
- Takes 1 argument:
 - **a** Pointer to a SCREEN data structure which represents the “new” terminal. This new terminal then becomes the **current terminal**.
- Returns the PREVIOUS “current terminal;” all other calls affect ONLY the current terminal.

longname()

- This function returns a pointer to a static area containing a verbose description of the current terminal.
- It is defined only AFTER a call to either
 - **initscr,**
 - **newterm,** or
 - **setupterm.**

4. Structure Routines

All programs using **curses** should include the file `<curses.h>`. This file defines several **curses** functions as macros, and defines several global variables as well as the datatype "WINDOW."

There are two WINDOW constants **stdscr** (the standard screen) and **curscr** (the current screen).

Integer constants **LINES** and **COLS** are defined and contain the size of the screen.

Constants **TRUE** and **FALSE** are defined, with values 1 and 0, respectively.

Additional constants which are values returned from most **curses** functions are **ERR** and **OK**. **OK** is returned if the function could be properly completed, and **ERR** is returned if there was some error, such as moving the cursor outside of a window.

The include file `<curses.h>` automatically includes `<stdio.h>` and an appropriate tty driver interface file, currently either `<sgtty.h>` or `<termio.h>`.

NOTE: Including `<stdio.h>` again by including it explicitly is harmless but wasteful. However, including `<sgtty.h>` again by including it explicitly will usually result in a fatal error.

A program using **curses** should include the loader option `-lcurses` in the makefile. This is true for **BOTH** the **terminfo** level and the **curses** level.

The compilation flag `-DMINICURSES` can be included if you restrict your program to a small subset of **curses** concerned

CURSES

primarily with screen output and optimization. The routines possible with mini-curses are listed in the following section appropriately entitled **Mini-Curses**.

5. Mini-Curses

Curses copies from the current window to an internal screen image for every call to **refresh**.

If the programmer is only interested in screen output optimization, and does not want the windowing or input functions, an interface to the lower level routines is available. This will make the program somewhat smaller and faster.

The interface is a subset of full **curses**, so that conversion between the levels is not necessary to switch from mini-curses to full **curses**.

The following functions of **curses** and **terminfo** are available to the user of minicurses:

CURSES FUNCTIONS AVAILABLE WITH MINI-CURSES		
addch(ch)	addstr(str)	attroff(at)
attron(at)	attrset(at)	clear()
erase()	initscr	move(y,x)
mvaddch(y,x,ch)	mvaddstr(y,x,str)	newterm
refresh()	standend()	standout()

The following functions of **curses** and **terminfo** are NOT available to the user of minicurses:

CURSES

CURSES FUNCTIONS NOT AVAILABLE WITH MINI-CURSES			
box	clrtobot	clrtoeol	delch
deleteln	delwin	getch	getstr
inch	insch	insertln	longname
makenew	mvdelch	mvgetch	mvgetstr
mvinch	mvinsch	mvprintw	mvscanw
mvwaddch	mvwaddstr	mvwdelch	mvwgetch
mvwgetstr	mvwin	mvwinch	mvwinsch
mvwprintw	mvwscanw	newwin	overlay
overwrite	printw	putp	scanw
scroll	setscrreg	subwin	touchwin
vidattr	waddch	waddstr	wclear
wclrtobot	wclrtoeol	wdelch	wdeleteln
werase	wgetch	wgetstr	winsch
winsertrl	wmove	wprintw	wrefresh
wscanw	wsetscrreg		

The subset mainly requires the programmer to avoid use of more than the one window **stdscr**. Thus, all functions beginning with “w” are generally undefined.

Certain high level functions that are convenient but not essential are also not available, including **printw** and **scanw**.

Also, the input routine **getch** CANNOT be used with mini-curses.

Features implemented at a low level, such as use of hardware insert/delete line and video attributes, are available in both versions.

Mode setting routines such as **crmode** and **noecho** are allowed.

CURSES

To access MINI-CURSES, add **-DMINICURSES** to the CFLAGS in the makefile.

If routines are requested that are not in the subset, the loader will print error messages such as

Undefined:

m_getch

m_waddch

to tell you that the routines **getch** and **waddch** were used but are not available in the subset.

Since the preprocessor is involved in the implementation of mini-curses, the entire program must be recompiled when changing from one version to the other.

CURSES

6. Option-Setting Routines

These functions set options within **curses**. In each case, **window** is the WINDOW affected, and **flag** is a boolean flag with value **TRUE** or **FALSE** indicating whether to enable or disable the option.

All options are initially **FALSE**.

It is not necessary to turn these options off before calling **endwin**.

clearok(window,flag)

- If set, the next call to **wrefresh** with this window will clear the screen and redraw the entire screen.
- If **window** is **curser**, the next call to **wrefresh** with any WINDOW will cause the screen to be cleared.
- This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

idlok(win,bf)

- If **ENABLED**, **curses** will consider using the hardware insert/delete line feature of terminals so equipped.
- If **DISABLED**, **curses** will never use this feature.
- The insert/delete **CHARACTER** feature is always considered.
- If insert/delete line **CANNOT** be used, **curses** will redraw the changed portions of all lines that do not match the desired line.

keypad(window,flag)

- This option “enables” the terminal keypad, if the keypad has both the facility to transmit (keypad

- “on”) and work locally (keypad “off”).
- If **ENABLED**, when a function key is used, **getch** will return a single value representing that function key.
- If **DISABLED**, **curses** will NOT treat function keys specially.

leaveok(window,flag1)

- Causes the cursor to be left wherever the update leaves it.
- Useful for applications where the cursor is not used, since it reduces the need for cursor motions.
- When **ENABLED**, (if possible) the cursor is made invisible.

meta(window,flag)

- If **ENABLED**, characters returned by **getch** are transmitted with ALL 8 bits, instead of stripping the highest bit.
- The value **OK** is returned if successful; the value **ERR** is returned if either the terminal or system cannot receive 8-bit input.
- Useful for extending the non-text command set in applications where the terminal has a meta shift key.
- **Curses** takes whatever measures are necessary to arrange for 8-bit input. On some versions of the UNIX Operating System, raw mode will be used. On others, the character size will be set to 8, parity checking disabled, and stripping of the 8th bit turned off.
- Note that 8-bit input is a fragile mode.
- Many programs and networks **ONLY** pass 7 bits.
- If **ANY** link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

CURSES

nodelay(window,flag)

- Causes **getch** to be a **non-blocking call**.
- If no input is ready, **getch** will return **-1**.
- If **DISABLED**, **getch** will “hang” until a key is pressed.

intrflush(window,flag)

- If **ENABLED**, when an interrupt key (**interrupt**, **quit**, **suspend**) is used, all output in the **tty driver queue** is “flushed” giving faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen.
- If **DISABLED**, prevents the “flush.”
- **DEFAULT** is for **intrflush** to be **ENABLED**.
- Depends on support in the underlying teletype driver.

typeahead(fd)

- Sets the file descriptor for **typeahead** check.
- Takes one argument:
fd This flag is the “file descriptor” returned from **open** or **fileno**. **fd** is an integer.
- Setting **fd** is **-1** **DISABLEs** **typeahead** check.
- The **DEFAULT** for **fd** is **0**. (0 is the standard input file descriptor, just as 1 is the standard output file descriptor and 2 refers to standard error.)
- **Typeahead** is checked independently for each screen, and for multiple interactive terminals it should be set to the appropriate input for each screen.
- A call to **typeahead** affects **ONLY** the current screen.

scrollok(window,flag)

- Controls what happens when the cursor of a **WINDOW** is moved “off the edge” (i.e., by reaching a

newline on the bottom line or by having finished typing the last character on the last line.)

- If **DISABLED**, the cursor is left on the bottom line.
- If **ENABLED**, **wrefresh** is called on the **WINDOW**, causing the **physical terminal** and **WINDOW** to be “scrolled up” one line.
- The **idlok** function **MUST** also be called to get the “physical” scrolling effect on the terminal.

setscrreg(top,bottom)

wsetscrreg(window,top,bottom)

- Both functions are used to set a “software scrolling region” in a **WINDOW** (either **curscr** or **stdscr**).
- The arguments are:

window Specifies the **WINDOW** affected by this call to **wsetscrreg**.

top The line number of the top margin of the scrolling region.

Line 0 is the top line of the **WINDOW**.

bottom The line number of the bottom margin of the scrolling region.

- If **ENABLED WITH scrollok**, an attempt to move off the bottom margin line will cause all lines in the “scrolling region” to scroll up one line.
- This has nothing to do with use of a terminal’s “physical scrolling region capability” — **ONLY** the **text** of the window is scrolled.
- If **idlok** is **ENABLED AND** the terminal has either a “scrolling region” or “insert/delete line” capability, the **curses** output routines will use one of those capabilities.

CURSES

7. Output Routines

refresh()

wrefresh(win)

- Must be called to get ANY OUTPUT on the terminal,
- The **wrefresh** function is the "window" version of **refresh** and copies the specified WINDOW, (**win**), to the **physical terminal screen**.
- The **refresh** function copies the default WINDOW (**stdscr**) to the **physical terminal screen**.
- The physical cursor of the terminal is left at the location of the WINDOW's cursor — UNLESS **leaveok** is ENABLED.

doupdate()

wnoutrefresh(window)

- Used to allow multiple updates with more efficiency than **wrefresh**.
- The **wrefresh** function works by first copying the specified WINDOW to the virtual screen (what the programmer wants to appear on the screen) with a call to (**wnoutrefresh**). Then the (**doupdate**) routine is called to update the screen.

By using the **wnoutrefresh** function for each window, INSTEAD OF **wrefresh**, it is possible to call the **doupdate** routine only once with probably fewer total characters transmitted.

prefresh(pad,pminr,pminc,sminr,sminc,smaxr,smaxc)

pnoutrefresh(pad,pminr,pminc,sminr,sminc,smaxr,smaxc)

- These routines are analogous to **wrefresh** and **wnoutrefresh** except that they involved PADS, instead of WINDOWS.
- The following additional parameters are needed to

indicate what part of the PAD and screen are involved:

pmnr Specifies the UPPER MOST ROW of the rectangle PAD to be displayed.

Together with **pmnc**, used to specify the UPPER LEFT CORNER of the rectangle PAD to be displayed.

pmnc Specifies the LEFT MOST COLUMN of the rectangle PAD to be displayed.

sminr Specifies the UPPER MOST ROW (on the screen) of the rectangle in which the PAD will be displayed.

Together with **smnc**, **smaxr** and **smaxc** used to specify the edges (on the screen) of the rectangle in which the PAD will be displayed.

smnc Specifies the LEFT MOST COLUMN (on the screen) of the rectangle in which the PAD will be displayed.

smaxr Specifies the BOTTOM ROW (on the screen) of the rectangle in which the PAD will be displayed.

smaxc Specifies the RIGHT MOST COLUMN (on the screen) of the rectangle in which the PAD will be displayed.

- The LOWER RIGHT CORNER of the rectangle PAD to be displayed is calculated from the screen coordinates, since both the PAD and the rectangle in which it will be displayed MUST be the same size.
- Both rectangles must be entirely contained within their respective structures.

The following routines are used to "draw" text on WINDOWS. In all cases, the following conventions apply:

- If not explicitly stated, the WINDOW affected is **stdscr**.

CURSES

- **y** and **x** are the row and column, respectively.
- The UPPER LEFT CORNER is ALWAYS (0,0).
- Functions beginning with **mv** imply a call to **move** BEFORE the call to the other function.

move(y, x)

wmove(window, y, x)

- Used to move the cursor in the specified WINDOW (**window** or **stdscr**) to the location specified by the **x** and **y** parameters.
- The physical terminal cursor is NOT moved UNTIL **refresh** is called.
- The position specified is relative to the UPPER LEFT CORNER of the SCREEN. If the UPPER LEFT CORNER of the specified WINDOW is NOT the same as the UPPER LEFT CORNER of the SCREEN, The SCREEN coordinates of that corner of the window must be passed to **move** instead of (0,0).

addch(ch)

waddch(window, ch)

mvaddch(y, x, ch)

mvwaddch(window, y, x, h)

- The character **ch** is put in the WINDOW at the current cursor position and the cursor position is advanced.
- If **ch** is a tab, newline or backspace, the cursor will be moved appropriately in the WINDOW.
- If **ch** is a control character OTHER THAN tab, newline or backspace, for example CTRL-u, it will appear on the screen preceded by a "^" (in this case, as ^U).
- At the right margin, an automatic newline is performed.
- If **scrollok** is ENABLED, when the cursor reaches

the bottom of the "scrolling region," the scrolling region will scroll up one line.

- The parameter **ch** is actually an **INTEGER**, NOT A **CHARACTER**.
- **Video attributes** can be combined with a character by "OR-ing" them into the parameter thereby **SETTING THOSE ATTRIBUTES**.

The intent here is that text, including **attributes**, can be copied from one place to another with both **insch** and **addch**.)

addstr(str)
waddstr(window,str)
mvaddstr(y,x,str)
mvwaddstr(window,y,x,str)

- Used to write all the characters of a NULL-terminated character string, **str**, on the given **WINDOW**.

erase()
werase(window)

- Used to copy blanks to **EVERY** position in the **WINDOW**.

clear()
wclear(window)

- These functions are analogous to **erase** and **werase** but they also call **clearok** so that the screen will be cleared on the next call to **refresh** that **WINDOW**.

clrtoobot()
wclrtoobot(window)

- The **clrtoobot** routine causes all lines **BELOW** the cursor in the **WINDOW** to be erased.

CURSES

- The **clrtoeol()** routine also causes the rest of the current line to the right of the cursor to be erased.

clrtoeol()

wclrtoeol(window)

- These functions cause the rest of the current line to the right of the cursor to be erased.

delch()

wdelch(window)

mvdelch(y,x)

mvwdelch(window,y,x)

- The character under the cursor in the window is deleted.
- All characters to the right on the same line are moved to the left one position.
- This DOES NOT USE the hardware “delete character” feature.

deleteln()

wdeleteln(window)

- The line under the cursor in the window is deleted.
- All lines below the current line are moved up one line.
- The bottom line of the window is cleared.
- This DOES NOT USE the hardware “delete line” feature.

insch(c)

winsch(window,c)

mvinsch(y,x,c)

mvwinsch(window,y,x,c)

- The character, **c**, is inserted **BEFORE** the character under the cursor.
- All characters to the right are moved one space to

the right, possibly losing the rightmost character on the line.

- This DOES NOT USE the hardware “insert character” feature.

insertln()

wininsertln(window)

- A blank line is inserted ABOVE the current line.
- The bottom line is lost.
- This DOES NOT USE the hardware “insert line” feature.

printw(fmt,args)

wprintw(window,fmt,args)

mvprintw(y,x,fmt,args)

mvwprintw(window, y, x, fmt, args)

- These functions correspond to **printf**. The characters which would be output by **printf** are instead output by the function **waddch** on the given WINDOW.

box(window, vert, hor)

- A box is drawn around the edge of the WINDOW.
- The parameters **vert** and **hor** are the characters used in drawing the box.

scroll(window)

- The WINDOW is “scrolled up” one line.

This involves moving the lines in the WINDOW data structure.

- As an optimization, if the WINDOW is **stdscr** and the “scrolling regions” is the entire WINDOW, the physical screen will be “scrolled” at the same time.

CURSES

7.1 Delays

The following functions are highly UNPORTABLE, but are often needed by programs that use **curses** (especially real-time response programs).

Some of the following functions require modification before they are prepared to run on a given operating system.

In ALL cases, the routine will compile and return an error status if the requested action is not possible.

draino(ms)

- The program is suspended until the output queue has "drained" enough to complete in the time specified by the argument **ms**.

The argument **ms** is expected to be some number of milliseconds

For example, **draino(50)**, at 1200 baud, would pause until there are no more than 6 characters in the output queue, because it would take 50 milliseconds to output the additional 6 characters.

- The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen.
- If the operating system DOES NOT support the **ioctl**s (I/O controls) needed to implement **draino**, the value **ERR** is returned.
- If the operating system DOES support the **ioctl**s (I/O controls) needed to implement **draino**, the value **OK** is returned.

napms(ms)

- This function suspends the program for **ms** milliseconds.

CURSES

- This function is similar to **sleep** but has higher resolution.
- The resolution actually provided will vary with the facilities available in the operating system, and often a change to the operating system will be necessary to produce the best results.

If resolution of at least .1 second is not possible, the routine will round to the next higher second, call **sleep**, and return ERR. (Otherwise, the value OK is returned.)

- Often the resolution provided is 1/60th second.

CURSES

8. Window-Manipulation Routines

newwin(num_lines, num_cols, beg_row, beg_col)

- Creates a new WINDOW with the number of lines and columns specified in the first two parameters.
- The UPPER LEFT CORNER of the WINDOW is at row **beg_row** and column **beg_col**.
- If either **num_lines** or **num_cols** is given as ZERO, **curses** uses, by DEFAULT, the result of the expression **LINES - beg_row** and **COLS - beg_col**.
- Using **newwin(0,0,0,0)** creates a new full-screen WINDOW.

newpad(num_lines, num_cols)

- Creates a new PAD data structure.
- A PAD is like a window -- BUT it is NOT RESTRICTED by the screen size, AND it is NOT ASSOCIATED with a particular part of the screen.
- PADS can be used when a large WINDOW is needed, and only a part of the WINDOW will be on the screen at one time.
- There are NO AUTOMATIC refreshes of PADS (e.g. from scrolling or echoing of input).
- PAD is NOT a LEGAL argument to **refresh** -- use **prefresh** or **pnoutrefresh** instead.
- Just as "window versions" of functions must have an extra parameter to specify the specific WINDOW to be affected, the PAD routines require additional parameters to:
 - Specify the part of the PAD to be displayed
 - Specify the location on the screen to be used for display

subwin(orig, num_lines, num_cols, x, y)

- Create a new WINDOW within another window.

The original WINDOW is given to **subwin** as its first parameter, **orig**.

- The number of lines and columns of the "sub-WINDOW" are given in the second and third parameters, **num_lines** rows by **num_cols** columns.
- The WINDOW is located at row **x** and column **y** — relative to the screen, not **orig**.
- The new sub-WINDOW is made in the WINDOW **orig**, and changes made to one WINDOW affect BOTH WINDOWS.
- It is often necessary to call **touchwin** before calling **wrefresh** when using **subwin**.

delwin(window)

- Used to delete the specified WINDOW (**window**), and free any memory associated with it.
- In the case of overlapping WINDOWS, sub-WINDOWS should be deleted BEFORE the deleting the main WINDOW.

mvwin(window, br, bc)

- Used to move a specified WINDOW (**window**) so that the UPPER LEFT CORNER is at row **br** and column **bc**.
- If the move would cause the WINDOW to be OFF THE SCREEN, it is an ERROR, and the WINDOW is NOT moved.

touchwin(window)

- Used to "throw away" optimization information concerning which parts of the WINDOW have been "touched."
- Sometimes necessary when using overlapping WINDOWS, since a change to one WINDOW will affect the other WINDOW but that change will NOT be reflected in the record of "which lines have been

CURSES

changed" in the other WINDOW.

overlay(window1, window2)

overwrite(window1, window2)

- Used to "overlay" **window1** on top of **window2**. All text in **window1** is copied into **window2**.
- The **overlay** is NONDESTRUCTIVE (blanks are not copied).
- The **overwrite** is DESTRUCTIVE.

8.1 Multiple Windows

A window is a data structure representing all or part of the CRT screen. It has room for:

- A two dimensional array of characters and attributes for each character to a total of 16 bits per character (7 for text and 9 for attributes)
- A cursor
- A set of current attributes
- A number of flags.

Curses provides two full screen windows:

1. **stdscr**
This full screen window is the standard screen buffer.
2. **curscr**
This full screen window is the current screen which represents the physical terminal screen.

It is important to understand that a window is only a data structure. Therefore, use of more than one window does not imply use of more than one terminal, nor does it involve more than one process.

A window is merely an object which can be copied to all or part of the terminal screen. The current implementation of **curses** does not allow windows which are bigger than the screen.

The programmer can create additional windows with the function **newwin()** which returns a pointer to a newly created window and takes four parameters:

lines	The length of the screen
cols	The width of the screen

CURSES

begin_row The left most side of the screen

begin_col The top of the screen

The upper left corner of the window will be at screen position (**begin_row**, **begin_col**).

All operations that affect **stdscr** have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a "w" on the front of the **stdscr** function, and the window name is added as the first parameter.

For example, the function **waddch(mywin, c)** would write the character **c** to window **mywin**, and the function **wrefresh(mywin)** would flush the contents of a window (**mywin**) to the screen.

Windows are useful for maintaining several different screen images, and alternating the user among them.

It is possible to subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more recently refreshed window.

In all cases, the "non-w" version of the function calls the "w" version of the function, using **stdscr** as the additional argument.

For example, a call to **addch(c)** results in a call to **waddch(stdscr, c)**

A set of "move" functions are also provided for most of the common functions so that a call to **move** is made right before the call to the other function.

For example, **mvaddch(row, col, c)** is one function which actually performs two function calls:

1. The first to **move(row, col)**
2. The second to **addch(c)**

There are “move” functions for the “window” functions as well. For example, **mvwaddch(row, col, mywin, c)** causes first the move to **row, col** and then the character **c** to be written to the window **mywin**.

8.2 Example Program — WINDOW

The program “WINDOW” is an example of the use of multiple windows.

- The main display is kept in **stdscr**.
- When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen.
- A call to **wrefresh** on the temporary window created, causes the window to be **WRITTEN OVER stdscr** on the screen.
- Calling **refresh** on **stdscr** results in the original window being **REDRAWN** on the screen.
- The function **touchwin** is called before writing out an overlapping window. This is necessary to defeat an optimization in **curses**.

If you have trouble refreshing a new window which overlaps an old window, it may be necessary to call **touchwin** on the new window to get it completely written out.

Following is the fourth example program, “WINDOW:”

CURSES

```
/* WINDOW. This program is an example of
 * the use of multiple windows
 */

#include < curses.h>

WINDOW *cmdwin;

main()
{
    int i, c;
    char buf[120];

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0); /* top 3 lines */
    for (i=0; i<LINES; i++)
        mvprintw(i, 0, "This is line %d of stdscr", i);

    for (;;) {
        refresh();
        c = getch();
        switch (c) {
            case 'c': /* Enter command from keyboard */
                werase(cmdwin);
                wprintw(cmdwin, "Enter command:");
                wmove(cmdwin, 2, 0);
                for (i=0; i<COLS; i++)
                    waddch(cmdwin, '-');
                wmove(cmdwin, 1, 0);
                touchwin(cmdwin);
                wrefresh(cmdwin);
                wgetstr(cmdwin, buf);
                touchwin(stdscr);
                /*
                 * The command is now in buf.
                */
            }
```

```
        * It should be processed here.  
        */  
        break;  
case 'q':  
    endwin();  
    exit(0);  
}  
}  
}
```

CURSES

9. Terminal Mode-Setting Routines

The following functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called:

cbreak()

nocbreak()

- Calling the **cbreak** function puts the terminal setting IN **CBREAK** mode.
- Calling the **nocbreak** function takes the terminal setting OUT OF **CBREAK** mode.
- If **CBREAK** mode is SET, characters typed are immediately available to the program.
- If **NOCBREAK** mode is SET, the teletype driver buffers characters typed UNTIL newline is typed.
- **INTERRUPT** and **FLOW CONTROL** characters are NOT AFFECTED by this mode.
- **DEFAULT**, initially, is **CBREAK** mode is NOT set. Therefore, most interactive **curses** programs call **cbreak** to set **CBREAK** mode.

echo()

noecho()

- Control whether characters are “echoed” as typed.
- **DEFAULT**, initially, is that **ECHO** is SET and, therefore, characters typed are echoed by the teletype driver.
- To control echoing to one area of the screen, or not to echo at all, use the function **noecho()** to TURN OFF **ECHO**.

nl()

nonl()

- Control whether “newline” is translated into “carriage return” and “linefeed” on output, and

whether "return" is translated into "newline" on input.

- **DEFAULT**, initially, **NEWLINE** and **RETURN** ARE TRANSLATED.
- If **nonl** is called, **TRANSLATIONS ARE DISABLED**. This is often used to make better use of the linefeed capability to obtain faster cursor motion.

raw()

noraw()

- The **raw()** function **SETS RAW mode**.
- The **noraw()** function **UNSETS RAW mode**.
- **RAW mode** is similar to **cbreak** mode in that characters typed are immediately available to the program. However, in **RAW mode**, the **INTERRUPT** and **CONTROL FLOW** characters **ARE AFFECTED** by **RAW mode**.
- The **INTERRUPT** and **CONTROL FLOW** characters (**interrupt**, **quit** and **suspend**) are **UNINTERPRETED** in **RAW mode** and **DO NOT** generate a signal, but pass directly to the program.

(Just as when a terminal is set in **RAW mode**, giving the command "**reset**" followed by a **<RETURN>** will result in an error messaged "**reset^M Command not found.**" The **<LINE FEED>** key or a **CTRL-I** must be used instead of the **<RETURN>** key to transmit the **reset** command.)

- **RAW mode** causes 8-bit input and output.

resetty()

savetty()

- The **resetty** function restores the state of the **tty** modes.
- The **savetty** function saves the state of the **tty**

CURSES

modes.

- The **savetty** function saves the current state in a buffer,
- The **resetty** function restores the state to what it was at the last call to **savetty**.

9.1 TTY Mode Functions

In addition to the save/restore routines **savetty()** and **resetty()**, standard routines are available for going into and out of normal tty mode. These routines are:

resetterm()

- This routine puts the terminal back in the mode it was in when **curses** was started.
- The **endwin** routine automatically calls **resetterm**.
- The routine to handle CTRL-z (on other systems that have process control) also uses **resetterm**. Programmers should use this routine before and after shell escapes, and also if they write their own routine to handle CTRL-z. This routine is also available at the **terminfo** level.

fixterm()

- This routine undoes the effects of **resetterm**, i.e., restores the "current **curses** mode."
- The routine to handle CTRL-z (on other systems that have process control) also uses **fixterm**. Programmers should use this routine before and after shell escapes, and also if they write their own routine to handle CTRL-z. This routine is also available at the **terminfo** level.

saveterm()

- This routine saves the current state to be used by **fixterm()**.

nodelay(a, b)

- **nodelay** takes two arguments:
 - a** First argument is the window to be affected.
 - b** Second argument is either **TRUE** (meaning

CURSES

“ENABLE”) or FALSE (meaning “DISABLE”).

- A terminal is put in **nodelay mode** with the following call:

nodelay(stdscr, TRUE)

- While in this mode, any call to **getch** will return **-1** if there is nothing waiting to be read immediately. This is useful for writing programs requiring “real time” behavior where the users watch action on the screen and press a key when they want something to happen.
- For example, the cursor can be moving across the screen, in real time. When it reaches a certain point, the user can press an arrow key to change direction at that point.

9.2 Video Attributes

Video attributes can be displayed in any combination on terminals with this capability. They are treated as an extension of the standout capability, which is still present.

Each character position on the screen has 16 bits of information associated with it. Seven of these bits are the character to be displayed, leaving separate bits for nine video attributes. These bits are used for **standout**, **underline**, **reverse video**, **blink**, **dim**, **bold**, **blank**, **protect**, and **alternate character set**.

The attribute **standout** is taken to be whatever highlighting works best on the terminal, and should be used by any program that does not need specific or combined attributes.

Underlining, **reverse video**, **blink**, **dim**, and **bold** are the usual video attributes.

Blank means that the character is displayed as a space, for security reasons. Protected and alternate character set depend on the particular terminal. The use of these last three bits is subject to change and not recommended.

Note also that not all terminals implement all attributes — in particular, no current terminal implements both **dim** and **bold**.

The routines to use these attributes include

attrset(attrs)	wattrset(win, attrs)
attron(attrs)	wattron(win, attrs)
attroff(attrs)	wattroff(win, attrs)
standout()	wstandout(win)
standend()	wstandend(win)

Attributes, if given, can be any combination of **A_STANDOUT**, **A_UNDERLINE**, **A_REVERSE**, **A_BLINK**, **A_DIM**, **A_BOLD**, **A_INVIS**, **A_PROTECT**, and

CURSES

A_ALTCHARSET.

These constants, defined in **curses.h**, can be combined with the **C | (OR)** operator to get multiple attributes.

The **attrset** routine sets the current attributes to the given **attrs**.

The **attron** routine turns on the given **attrs** in addition to any attributes that are already on.

The **attroff** routine turns off the given attributes, without affecting any others.

standout and **standend** are equivalent to **attron(A_STANDOUT)** and **attroff(A_NORMAL)**.

If the particular terminal does not have the particular attribute or combination requested, **curses** will attempt to use some other attribute in its place.

If the terminal has no highlighting at all, all attributes will be ignored.

attroff(at)
wattroff(window, attrs)
attron(at)
wattron(window, attrs)
attrset(at)
wattrset(window, attrs)
standout()
standend()
wstandout(window)
wstandend(window)

- These functions set the **current attributes** of the

specified WINDOW, (either **stdscr** or **window**).

- The **current attributes** can be any combination of **A_STANDOUT**, **A_REVERSE**, **A_BOLD**, **A_DIM**, **A_BLINK** and **A_UNDERLINE**.
- These constants are defined in `< curses.h >` and can be combined with the C “|” (OR) operator.
- The **current attributes** of a WINDOW are applied to all characters that are written into the WINDOW with **waddch**.
- **Attributes** are a property of the character, and move with the character through any scrolling and insert/delete line/character operations.
- To the extent possible on the particular terminal, the **attributes** will be displayed as the graphic rendition of characters put on the screen.
- The function **attrset(at)** SETS the **current attributes** of the given WINDOW to **at**.
- The function **attroff(at)** TURNS OFF the specified attributes without affecting any other attributes.
- The function **attron(at)** TURNS ON the named attributes without affecting any others.
- The function **standout** is equivalent to **attron(A_STANDOUT)**.
- The function **standend** is equivalent to **attrset(0)**, that is, it TURNS OFF ALL attributes.

beep()
flash()

- These functions are used to produce “signals.”

The function **beep** invokes the audible alarm on the terminal, if possible. If the terminal is not capable of producing an audible alarm, this function will invoke a “flash” (or visible bell), if that is possible.

The function **flash** invokes a visible bell, or “flash,” on the screen. If the terminal is not capable of producing a visible bell, this function will invoke a

CURSES

- “beep” (audible bell), if that is possible.
- If neither an audible nor visual signal is possible, nothing will happen.
- Nearly all terminals have an audible signal (bell or beep) but only some can flash the screen.

9.3 Special Keys

Many terminals have special keys, such as arrow keys, keys to erase the screen, insert or delete text, and keys intended for user functions. The particular sequences these keys send differ from terminal to terminal.

Curses allows the programmer to handle these keys. For example, a program using special keys should turn on the keypad by calling the following routine at initialization:

```
keypad(stdscr, TRUE)
```

This will cause special characters to be passed through to the program by the function **getch**.

These keys have values starting at 0401, so they should not be stored in a **char** variable, as significant bits will be lost.

A program using special keys should avoid using the **escape** key, since most sequences start with escape, creating an ambiguity.

Curses will set a one second alarm to deal with this ambiguity, which will cause delayed response to the escape key. It is a good idea to avoid escape in any case, since there is eventually pressure for nearly any screen oriented program to accept arrow key input.

CURSES

9.4 Scrolling Region

There is a programmer-accessible scrolling region.

Normally, the scrolling region is set to the entire window, but the following calls set the scrolling region for **stdscr** or the given window to any combination of top and bottom margins:

```
setscrreg(top, bot)  
wsetscrreg(win, top, bot)
```

When scrolling past the bottom margin of the scrolling region, the lines in the region will move up one line, destroying the top line of the region. If scrolling has been enabled with **scrollok**, scrolling will take place only within that window.

Note that the scrolling region is a software feature, and only causes a window data structure to scroll. This may or may not translate to use of the hardware scrolling region feature of a terminal, or insert/delete line.

9.5 Highlighting

The function **addch** always draws two things on a window:

1. The character itself
2. Its **attribute**

In addition to the character itself, a set of **attributes** is associated with the character. These attributes cover various forms of highlighting of the character. For example, the character can be put in **reverse video**, **bold**, or be **underlined**.

A window always has a set of **current attributes** associated with it. The current attributes are associated with each character as it is written to the window.

The current attributes can be changed with a call to **attrset(attrs)**.

The names of the attributes are

ATTRIBUTES	
A_BOLD	A_REVERSE
A_DIM	A_STANDOUT
A_INVIS	A_UNDERLINE

Consider the following figure as an example of the use of both the function **attrset()** and the attribute **A_BOLD**:

CURSES

```
printw("A word in ");  
attrset(A_BOLD);  
printw("boldface");  
attrset(0);  
printw(" really stands out.\n");  
...  
refresh();
```

Figure 11.2. Use of Attributes

The output produced by this code, assuming a screen with the capability to output **bold** characters, would be:

A word in **boldface** really stands out.

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

Another attribute is called **A_STANDOUT**. This attribute is used to make text attract the attention of the user. The particular hardware attribute used for standout varies from terminal to terminal, and is chosen to be the most visually pleasing attribute the terminal has. **A_STANDOUT** is typically implemented as reverse video or bold.

Many programs don't really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the **A_STANDOUT** attribute is recommended.

Two convenient functions, **standout()** and **standend()** turn the **A_STANDOUT** attribute on and off, respectively.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use:

attrset(A_BLINK|A_BOLD)

Individual attributes can be turned on and off with **attron** and **attroff** without affecting other attributes.

In order to determine how to update the screen, **curses** must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to **refresh**, and to know the cursor position and screen contents at all times.

9.6 Example Program – HIGHLIGHT

The next example program is called “**HIGHLIGHT**” which uses attributes. The program takes a text file as input and allows embedded escape sequences to control attributes.

HIGHLIGHT comes about as close to being a “filter” as is possible with **curses**. It is not a true filter, however, because **curses** must “take over” the CRT screen.

In this example program,

- **\U** turns on underlining
- **\B** turns on bold
- **\N** restores normal text
- **scrollok** allows the terminal to scroll — should the file be longer than one screen (i.e., when an attempt is made to draw past the bottom of the screen, **curses** will automatically scroll the terminal up a line and call **refresh**.)

Following is the third example program, “**HIGHLIGHT:**”

CURSES

```
/*
 * HIGHLIGHT: a program to turn \U, \B, and
 * \N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;

    if (argc != 2) {
        fprintf(stderr, "Usage: highlight file\n");
        exit(1);
    }

    fd = fopen(argv[1], "r");
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\\') {
            c2 = getc(fd);
            switch (c2) {
                case 'B':
                    attrset(A_BOLD);
                    continue;
                case 'U':
```

```
        attrset(A_UNDERLINE);
        continue;
    case 'N':
        attrset(0);
        continue;
    }
    addch(c);
    addch(c2);
}
else
    addch(c);
}
fclose(fd);
refresh();
endwin();
exit(0);
}
```

CURSES

10. Multiple Terminals

The **curses** package can produce output on more than one terminal at once. This is useful for single process programs that access a common database, such as multi-player games. However, such programs are plagued with difficult problems, and **curses** does not solve all of them.

It is the responsibility of the program to determine the file name of each terminal line, and what kind of terminal is on each of those lines. This is a major problem, because the standard method (i.e., checking the **\$TERM** variable in the environment), does NOT work, since each process can only examine its own environment.

Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. A program wishing to take over another terminal cannot just shut off whatever program is currently running on that line (except, of course, for some applications, such as an inter-terminal communication program, or a program that takes over unused tty lines).

A typical solution requires the user logged in on each line to run a program that notifies the master program that the user is interested in joining the master program, telling it:

- The notification program's process id
- The name of the tty line
- The type of terminal being used

Then the program goes to sleep until the master program finishes.

When done, the master program wakes up the notification program, and all programs exit.

Since all information about the **current terminal** is kept in a global variable:

```
struct SCREEN *SP;
```

and although the **SCREEN** structure is "hidden" from the user, the C compiler will accept declarations of variables which are pointers. The user program should declare one **SCREEN** pointer variable for each terminal it wishes to handle.

The routine

```
struct SCREEN *SP  
newterm(type, fd)
```

will set up a new terminal of the given terminal type which puts its output on "file descriptor" **fd**.

A call to **initscr** is essentially:

```
newterm(getenv("TERM"),stdout)
```

A program wishing to use more than one terminal should use **newterm** for each terminal and save the value returned as a reference to that terminal.

To switch to a different terminal, call

```
set_term(term)
```

The old value of "SP" will be returned.

The programmer should not assign directly to "SP" because certain other global variables must also be changed.

All **curses** routines always affect the current terminal. To handle several terminals, switch to each one in turn with **set_term**, and then access it.

CURSES

- Each terminal must be set up with **newterm**, and closed down with **endwin**.

10.1 Current Terminal

The **current terminal** facility is how **curses** handles multiple terminals. All function calls always affect the **current terminal**. The master program sets up each terminal, saving a reference to the terminals in its own variables. When the master program wishes to affect a terminal, it sets the **current terminal** as desired, and then calls the ordinary **curses** routines.

References to terminals have type **struct screen**.

A new terminal is initialized by calling **newterm(type, fd)**. The parameter **type** is a character string, naming the kind of terminal being used.

The parameter **fd** is a "stdio" file descriptor to be used for input and output to the terminal.

The **newterm** function returns a screen reference to the terminal being set up. If only output is needed, the file can be open for output only.

The call to **newterm** replaces the normal call to **initscr**, which calls **newterm(getenv("TERM"), stdout)**

The call "**set_term(sp)**" changes the current terminal. The parameter **sp** is the screen reference to be made current. The function **set_term** returns a reference to the previous terminal.

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**.

Options such as **cbreak** and **noecho** must be set separately for each terminal.

CURSES

The functions **endwin** and **refresh** must be called separately for each terminal.

The following program fragment is an example of using multiple terminals as output for an "Important Message":

```
for (i=0; i<nterm; i++) {  
    set_term(terms[i]);  
    mvaddstr(0, 0, "Important message");  
    refresh();  
}
```

Figure 11.3. Using Multiple Terminals for Output

10.2 Example Program – TWO

The following sample program **TWO** does the following:

- Pages through a file, showing one page to the first terminal and the next page to the second terminal
- Waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space
- Sleeps for one second between checks

Note that each terminal had to be separately put into **nodelay** mode. Since no standard multiplexor is available in current versions of the UNIX™ Operating System, it is necessary to either invoke **wait**, or call **sleep(1)**, between each check for keyboard input.

The **TWO** program is just a simple example of two terminal **curses**. It does not handle notification, but instead, it requires the name and type of the second terminal on the command line.

The **sleep 100000** instruction must be input by the user on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the

second terminal.

The following is the fifth example program, "TWO:"

CURSES

```
/* TWO. This program is a simple example
 * of two terminal curses.
 */

#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
    int done();
    int c;

    if (argc != 4) {
        fprintf(stderr, "Usage: two othertty otherttytype inputfile\n");
        exit(1);
    }

    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "w+");
    signal(SIGINT, done); /* die gracefully */

    me = newterm(getenv("TERM"), stdout); /* initialize my tty */
    you = newterm(argv[2], fdyou); /* Initialize his terminal */

    set_term(me);          /* Set modes for my terminal */
    noecho();              /* turn off tty echo */
    cbreak();              /* enter cbreak mode */
    nonl();                /* Allow linefeed */
    nodelay(stdscr, TRUE); /* No hang on input */

    set_term(you);         /* Set modes for other terminal */
}
```

```

noecho();
cbreak();
nonl();
nodelay(stdscr, TRUE);

/* Dump first screen full on my terminal */
dump_page(me);

/* Dump second screen full on his terminal */
dump_page(you);

for (;;) {                                /* for each screen full */
    set_term(me);
    c = getch();
    if (c == 'q')                          /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(me);

    set_term(you);
    c = getch();
    if (c == 'q')                          /* wait for user to read it */
        done();
    if (c == ' ')
        dump_page(you);
    sleep(1);
}

dump_page(term)
struct screen *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line=0; line<LINES-1; line++) {
        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrbot();

```

CURSES

```
        done();
    }
    mvprintw(line, 0, "%s", linebuf);
}
standout();
mvprintw(LINES-1, 0, "--More--");
standend();
refresh();          /* sync screen */
}

/*
 * Clean up and exit.
 */
done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0);    /* to lower left corner */
    clrtoeol();         /* clear bottom line */
    refresh();          /* flush out everything */
    endwin();           /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0);    /* to lower left corner */
    clrtoeol();         /* clear bottom line */
    refresh();          /* flush out everything */
    endwin();           /* curses cleanup */

    exit(0);
}
```

10.3 Additional Terminals

Curses will work even if absolute cursor addressing is not possible, as long as the cursor can be moved from any location to any other location. It considers local motions, parameterized motions, home and carriage return.

Curses is aimed at full duplex, alphanumeric video terminals. No attempt is made to handle half-duplex, synchronous, hard copy or bitmapped terminals.

Bitmapped terminals can be handled by programming the bitmapped terminal to emulate an ordinary alphanumeric terminal. This does not take advantage of the bitmap capabilities, but it is the fundamental nature of **curses** to deal with alphanumeric terminals.

The **curses** handles terminals with the "magic cookie glitch" in their video attributes. (The term "magic cookie" means that a change in video attributes is implemented by storing a "magic cookie" in a location on the screen. This "cookie" takes up a space, preventing an exact implementation of what the programmer wanted.) Curses takes the extra space into account, and moves part of the line to the right, as necessary. In some cases, this will unavoidably result in losing text from the right hand edge of the screen. Advantage is taken of existing spaces.

CURSES

11. Low Level Termino Usage

Some programs need to use lower level primitives than those offered by **curses**. For such programs, the **terminfo** interface is offered.

This interface does not manage your CRT screen, but it does give you access to the ways in which you can manipulate the terminal.

Whenever possible, the higher level **curses** routines should be used because using them will make your program more portable to a wider class of terminals and versions of the UNIX Operating System.

Another consideration is that the glitches and misfeatures present in physical terminals are taken care of in the higher level **curses** routines. Working at the **terminfo** level, a programmer must be prepared to deal with them him/herself.

There are, however, two circumstances when it is appropriate to use **terminfo**. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second is when you are writing a filter.

A typical filter does one transformation on the input stream without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of **terminfo** is indicated.

A program writing at the **terminfo** level uses the framework shown in the following display:

```

#include < curses.h>
#include < term.h>

...
    setupterm(0, 1, 0);
    ...
    putp(clear_screen);
    ...
    reset_shell_mode();
    exit(0);

```

Figure 11.4. Terminfo level framework

Initialization is done by calling **setupterm**.

Passing the values 0, 1, and 0 to **setupterm** invokes reasonable defaults.

If **setupterm** can't figure out what kind of terminal is being used, it will print an error message and exit.

All "terminfo-level" programs should call **reset_shell_mode** before they exit.

Global variables, such as **clear_screen** and **cursor_address**, are defined by the call to **setupterm**. They can be output by using **putp** or **tputs**. (The **tputs** function allows the programmer more control).

NOTE: These strings (**clear_screen** and **cursor_address**) **SHOULD NOT** be directly output to the terminal using **printf** since they contain padding information.

A program that directly outputs strings will fail on terminals that require padding, or that use the **xon/xoff** flow control protocol.

CURSES

In the **terminfo** level, the higher level routines described previously are NOT available.

For a list of capabilities and a description of what they do, see **terminfo(4)** in the *UniPlus⁺ User Manual*, Sections 2-6.

11.1 Terminfo Level Routines

The following routines are called by low level programs that need access to the specific capabilities of **terminfo**.

A program working at this level should include both

- `< curses.h >` and
- `< term.h >`

(in that order).

After a call to **setupterm**, the **terminfo** capabilities will be available with macro names defined in `< term.h >`. See **terminfo(4)** in the *UniPlus⁺ User Manual*, Sections 2-6, for more information.

BOOLEAN-VALUED CAPABILITIES will have the value **1** if the capability is **AVAILABLE**, and the value **0** if it is not.

NUMERIC CAPABILITIES have the value **-1** if the capability is **MISSING**, and the value **0** (at least) if it is present.

STRING CAPABILITIES (both those with and without parameters) have the value **NULL** if the capability is **MISSING**, and otherwise are character pointers which point to a character string **containing** the capability.

The special character codes involving the `\` and `^` characters (e.g., `\r` notation for return and `^A` notation for the sequence "CTRL-a") are translated into the appropriate ASCII characters.

PADDING INFORMATION (of the form \$<time>) and parameter information (beginning with %) are left uninterpreted at this stage.

The routine **tputs** INTERPRETS PADDING INFORMATION.

The routine **tparm** INTERPRETS PARAMETER INFORMATION.

If the program only needs to handle one terminal, the definition **-DSINGLE** can be passed to the C compiler, resulting in **static references** to capabilities instead of **dynamic references**. This can result in smaller code, but prevents use of more than one terminal at a time.

setupterm(term, filenum, status)

- Used to initialize a terminal.
- Takes three arguments:

term The first parameter is a character string representing the **name** of the terminal being used.

filenum The second parameter is the **file descriptor** of the output terminal.

status The third parameter is a pointer to the integer representing the relative status, i.e. success or failure, of the function.

The values returned (and their interpretations) may be:

VALUE	INTERPRETATION
1	SUCCESS
0	NO SUCH TERMINAL
-1	PROBLEM IN LOCATING TERMINFO DATABASE

- If **term** is 0, curses will use the value of the

CURSES

- environment variable **\$TERM** as the first parameter.
- If **errret** is 0, **curses** will assume that NO ERROR CODE IS WANTED.
- If **errret** is 0 (DEFAULTED), and something goes wrong, **setupterm** will print an appropriate error message and exit, rather than returning.

Thus, by calling **setupterm(0, 1, 0)**, potential worries regarding initialization errors can be avoided.

- If the environment variable **\$TERMINFO** is SET to a **pathname**, the **setupterm** routine will check for a compiled **terminfo** description of the terminal under the path **\$TERMINFO**, BEFORE checking **/etc/term**.

Otherwise, only **/etc/term** is checked.

- The **setupterm** routine checks the tty driver mode bits, using **filenum**, and changes any that might prevent the correct operation of other low level routines.

For example, if the system is expanding tabs, the **setupterm** routine will REMOVE the definition of the **tab** and **backtab** functions because **curses** assumes if the hardware tabs are not used, they may not be properly set in the terminal. (Other system dependent changes, such as disabling a virtual terminal driver, may be made here.)

- As a side effect, the **setupterm** function initializes the global variable **ttytype** (an array of characters) to the value of a list of names for the terminal. This "name list" is taken from the beginning of the **terminfo** description for the terminal.
- After the call to the **setupterm** routine, the global variable **cur_term** is SET to point to the current structure of terminal capabilities.
- By calling **setupterm** for each terminal, and saving and restoring **cur_term**, it is possible for a program to use two or more terminals at once.

- The mode that turns newlines into the sequence <RETURN><LINEFEED> on output is NOT DISABLED. Therefore, programs that use either **cursor_down** or **scroll_forward** should DISABLE this mode.
- The **setupterm** routine calls **reset_prog_mode** after ANY changes it makes.

```

reset_prog_mode()
reset_shell_mode()
def_prog_mode()
def_shell_mode()

```

- All are used to change the tty modes between the two states:
 - shell** The mode they were in BEFORE the program was started
 - program** The mode needed by the program
- The **def_prog_mode** routine saves the current terminal mode as PROGRAM MODE.
- **setupterm** and **initscr** call **def_shell_mode** automatically.
- The **reset_prog_mode** routine puts the terminal into PROGRAM MODE.
- The **reset_shell_mode** routine puts the terminal into NORMAL MODE.
- A typical calling sequence is for a program to:
 - 1 call **initscr** (or **setupterm**);
 - 2 then SET the PROGRAM MODE by calling routines such as **cbreak** and **noecho**;
 - 3 then call **def_prog_mode** to save the current state;
 - 4 then, before a shell escape or CTRL-z suspension, call **reset_shell_mode**, to restore normal mode for the shell;
 - 5 then, when the program resumes, call **reset_prog_mode**.
- All programs MUST call **reset_shell_mode** BEFORE

CURSES

they exit.

- The **endwin** routine automatically calls **reset_shell_mode**.
- **NORMAL MODE** is stored in **cur_term->Ottyb**.
- **PROGRAM MODE** is stored in **cur_term->Nttyb**.
- Both **cur_term->Ottyb** and **cur_term->Nttyb** are type **sgttyb**. Currently the possible types are **struct sgttyb** and **struct termio**.
- The **def_prog_mode** routine should be called to save the current state in **Nttyb**.

vidputs(newmode, putc)

- Takes two arguments:
newmode Any combination of **attributes**, defined in **< curses.h >**.
putc A "putchar-like" function. The proper string to put the terminal in the given video mode is output.
- The previous mode is remembered by this routine.
- The result characters are passed through **putc**.

vidattr(newmode)

- The string to put the terminal in the given video mode, (**newmode**), is output to **stdout**.

tparm(instr, p1, p2, p3, p4, p5, p6, p7, p8, p9)

- Used to instantiate a parameterized string.
- The character string returned has the given parameters applied, and is suitable for **tputs**.
- Up to 9 parameters can be passed, in addition to the parameterized string.

tputs(cp, affcnt, outc)

- Processes a string capability, possibly containing padding information.

- Enough padding characters to allow a specified delay time replace the padding specification, and the resulting string is passed, one character at a time, to the routine **outc**, which expects a one character parameter. This routine often just calls **putchar**.
- This routine takes 3 arguments:
 - cp** the capability string
 - affcnt** the number of units affected by the capability,
 - outc** the routine to which the resulting string is passed.
- For example, the **affcnt** for **insert_line** is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal, but **affcnt** is also used by the padding information of some terminals as a multiplication factor.
- If the capability does not have a factor, the value 1 should be passed.

putp(str)

- Used to output a capability with NO **affcnt**.
- The string specified by **str** is output to **putchar** with an **affcnt** of 1.

delay_output(ms)

- A delay is inserted into the output stream for the number specified by the parameter **ms** in milliseconds.
- The current implementation inserts sufficient "padding" for the delay. This should NOT be used in place of a high resolution sleep, but rather for delay effects in the output.
- Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping.
- Since large numbers of pad characters can be output, it is recommended that **ms** NOT exceed 500.

CURSES

mvcur(oldrow, oldcol, newrow, newcol)

- This routine optimally moves the cursor from the position specified in the first two parameters, (**oldrow**, **oldcol**) to the position specified in the third and fourth parameters, (**newrow**, **newcol**).
- The user program is expected to keep track of the current cursor position.
- Unless a full screen image is kept, **curses** will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion.

For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over, but if **curses** does not have access to the screen image, it doesn't know what these characters are.

If available, terminal insert and delete line and character functions are considered by **curses**. Calling the following routine will **ENABLE** insert/delete line:

idlok(stdscr, TRUE);

By default, **curses** will NOT use the terminal's insert/delete line capability. This was not done for performance reasons, since there is no speed penalty involved, but because many terminal do not have this capability and if **curses** uses insert/delete line, the result on the screen can be visually annoying. Also many simple programs using **curses** do not need this capability, the default is to avoid insert/delete line. However, insert/delete character is ALWAYS considered.

11.2 Example Program — TERMHL

The next example program, "TERMHL," shows a simple use of **terminfo**. It is a version of the example program "HIGHLIGHT" but uses **terminfo** instead of **curses**.

TERMHL can be used as a filter. The following strings are used:

- To enter bold and underline mode
- To turn off all attributes

The routine **vidattr** could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it did since there are several ways to change video attribute modes.

This program was written to illustrate "typical" use of **terminfo**.

The function **tputs(cap, affcnt, outc)** applies padding information. Some capabilities contain strings like **\$<20>**, which means "pad for 20 milliseconds," but **tputs** generates enough pad characters to delay for the appropriate time.

The first parameter to **tputs**, (**cap**), is the string capability to be output.

The second parameter to **tputs**, (**affcnt**), is the number of lines affected by the capability. Some capabilities may require padding that depends on the number of lines affected.

For example, **insert_line** may have to copy all lines below the current line, and may require time proportional to the number of lines copied. By convention **affcnt** is 1 if no lines are affected. The value 1 is used, rather than 0, for safety, since **affcnt** is multiplied by the amount of time per item, and anything multiplied by 0 is 0.

The third parameter to **tputs** (i.e., **outc**), is a routine to be called with each character.

CURSES

For many simple programs, **affcnt** is always 1 and **putc** just calls **putchar**. In those cases, the routine **putp(cap)** is a convenient abbreviation. In fact, **TERMHL** could be simplified by using **putp**.

There is a special check in **TERMHL** for the **underline_char** capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. **TERMHL** keeps track of the current mode, and if the current character is supposed to be underlined, will output **underline_char**.

The following is the sixth example program "**TERMHL:**"

```

/*
 *  TERMHL. This program is a terminfo level
 *  version of the HIGHLIGHT example program.
 */

#include < curses.h>
#include < term.h>

int ulmode = 0;      /* Currently underlining */

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;
    int outch();

    if (argc > 2) {
        fprintf(stderr, "Usage: termhl [file]\n");
        exit(1);
    }

    if (argc == 2) {
        fd = fopen(argv[1], "r");
        if (fd == NULL) {
            perror(argv[1]);
            exit(2);
        }
    } else {
        fd = stdin;
    }

    setupterm(0, 1, 0);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\\') {

```

CURSES

```
        c2 = getc(fd);
        switch (c2) {
        case 'B':
            tputs(enter_bold_mode, 1, outch);
            continue;
        case 'U':
            tputs(enter_underline_mode, 1, outch);
            ulmode = 1;
            continue;
        case 'N':
            tputs(exit_attribute_mode, 1, outch);
            ulmode = 0;
            continue;
        }
        putchar(c);
        putchar(c2);
    }
    else
        putchar(c);
}
fclose(fd);
fflush(stdout);
resetterm();
exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
    outch(c);
    if (ulmode && underline_char) {
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}
```

```
/*  
 * Outchar is a function version of putchar that can be passed to  
 * tputs as a routine to call.  
 */  
outch(c)  
int c;  
{  
    putchar(c);  
}
```

CURSES

12. Input

Functions are provided for input from the keyboard. The primary function is **getch()** which waits for a character from the keyboard, and returns that character.

The **getch()** function is like **getchar** except that it goes through **curses**. Its use is recommended for programs using the **cbreak()** or **noecho()** options, since several terminal or system dependent options become available that are not possible with **getchar**.

Options available with **getch** include **keypad** which allows extra keys such as arrow keys, function keys and other special keys that transmit escape sequences to be treated as just another key. The values returned for these keys are listed in Figure 11.2.

NOTE: The values for these keys are over octal 400, so they should be stored in a variable larger than a **char**.

The option **nodelay** mode causes the value **-1** to be returned if there is no input waiting. By default, **getch** will wait until a character is typed.

Another routine available with **getch** is **getstr(str)**, which allows input of an entire line (all characters until the appearance of a newline). This routine handles "echoing" and the erase and kill characters.

getch()
wgetch(window)
mvgetch(y,x)
mvwgetch(window,y,x)

- A character is read from the terminal associated with the **WINDOW**.
- In **NODELAY** mode, if there is no input waiting,

the value `-1` is returned.

- In `DELAY` mode, the program will “hang” until the system passes text through to the program, which, depending on the setting of `cbreak`, will either be after one character, or after the first newline.
- If `KEYPAD` mode is `ENABLED`, and a function key is pressed, the code for that key will be returned instead of the “raw” characters.
- If a character is received that could be the beginning of a function key (i.e., `<ESCAPE>`), `curses` sets a 1-second timer. If the remainder of the sequence does not come in within that 1 second, the character is passed through to the program. Otherwise the function key value is returned.

getstr(str)

wgetstr(window,str)

mvgetstr(y,x,str)

mvwgetstr(window,y,x,str)

- Calls to `getch` are made until a newline is received.
- The resulting value is placed in the area pointed at by the character pointer `str`.
- `ERASE` and `KILL` characters are interpreted.

scanw(fmt,args)

wscanw(window,fmt,args)

mvscanw(y,x,fmt,args)

mvwscanw(window,y,x,fmt,args)

- This function corresponds to `scanf` in that first, `wgetstr` is called on the `WINDOW`, and then the resulting line is used as input for the scan.

getyx(window,y,x)

- The cursor position of the `WINDOW` is placed in the two integer parameters, `y` and `x`.

CURSES

Since this is a macro, no address indicator (&) is necessary.

inch()
winch(window)
mvinch(y,x)
mvwinch(window,y,x)

- The character at the current position in the named WINDOW is returned.
- If any **attributes** are set for that position, their values will be "OR-ed" into the value returned.
- The pre-defined constants **A_ATTRIBUTES** and **A_CHARTEXT** can be used with the operator "&" to extract either just the character or just the attributes.

The following function keys might be returned by **getch** if **keypad** has been enabled.

NOTE: Not all of these are currently supported, due to lack of definitions in **terminfo** or the terminal not transmitting a unique code when the key is pressed.

NAME	VALUE	KEY NAME
KEY_BREAK	0401	Break key (unreliable)
KEY_DOWN	0402	Down arrow
KEY_UP	0403	Up arrow
KEY_LEFT	0404	Left arrow
KEY_RIGHT	0405	Right arrow
KEY_HOME	0406	Home (upper left)
KEY_BACKSPACE	0407	Backspace (unreliable)
KEY_F0	0410	Function keys; 64 keys are reserved -- numbers 0410-0477
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char Enter insert mode
KEY_EIC	0514	Exit insert mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)

Figure 11.5. Function Keys Returned by getch() (1 of 2)

CURSES

NAME	VALUE	KEY NAME
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	Partial (soft) reset (unreliable)
KEY_RESET	0531	(hard) Reset (unreliable)
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down bottom (lower left)

Figure 11.6. Function Keys Returned by getch() (2 of 2)

12.1 Example Program – SHOW

The next program, “**SHOW**,” gives an example of the use of **getch**. **SHOW** pages through a file, showing one screen full each time the user presses the space bar.

By creating an input file for **SHOW** made up of 24 line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called **show scripts**.

The following is a brief description of the functions used in the example program:

- **cbreak**
This function is called so that the user can press the space bar without having to hit return.
- **noecho**
This function is called to prevent the space from echoing

in the middle of a **refresh**, messing up the screen.

- **nonl**
This function is called to enable more screen optimization.
- **idlok**
This function is called to allow insert and delete line, since many show scripts are constructed to duplicate bugs caused by that feature.
- **clrtoeol** and **clrtoebol**
These functions clear from the cursor to the end of the line and screen, respectively.

Following is the second example program, "SHOW:"

CURSES

```
/*
 * SHOW. This program pages through
 * a file, showing one screen full
 * each time the user presses the space bar.
 */

#include < curses.h>
#include < signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if(argc != 2)
    {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }
    if((fd = fopen(argv[1], "r")) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);

    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
```

```
for(line=0; line<LINES; line++)
{
    if(fgets(linebuf, sizeof linebuf, fd) == NULL)
    {
        clrtoeol();
        done();
    }
    move(line, 0);
    printw("%s", linebuf);
}
refresh();
if(getch() == 'q')
    done();
}

void
done()
{
    move(LINES-1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

CURSES

13. Portability

Several useful routines are provided to improve portability. The implementation of these routines is different from system to system, and the differences can be isolated from the user program by including them in **curses**.

Functions **erasechar()** and **killchar()** return the characters which erase one character, and kill the entire input line, respectively.

The function **baudrate()** will return the current baud rate, as an integer. For example, at 9600 baud, the integer 9600 will be returned, not the value **B9600** from **<sgtty.h>**.

The routine **flushinp()** will cause all typeahead to be thrown away.

13.1 Portability Functions

These functions DO NOT directly involve terminal dependent character output, but tend to be needed by programs that use **curses**.

Unfortunately, the implementation of these functions is highly system-dependent. However, they have been included here to provide at least a guideline for those concerned with writing portable **curses** programs.

baudrate()

- **baudrate** returns the output speed of the terminal.
- The number returned is the "integer baud rate," (for example, 9600), rather than a "table index" such as "B9600."

erasechar()

- The ERASE character chosen by the user is returned.
- This is the character typed by the user to ERASE the character just typed.

killchar()

- The LINE KILL character chosen by the user is returned.
- The KILL character is used when a decision is made to abort the current input line before it has been transmitted to the program (e.g., with a carriage return).

flushinp()

- **flushinp** throws away any typeahead not yet read by the program.

CURSES

14. Example Program – EDITOR

The following program, “**EDITOR**,” is the last example in this document. This program is a very simple screen editor, patterned after the vi editor.

EDITOR keeps its buffer in **stdscr**, just to keep the program simple. Obviously a real screen editor would keep a separate data structure.

NO provision is made for the following:

- Files of any length other than the size of the screen
- Lines longer than the width of the screen
- Control characters in the file

The following are several important points concerning this sample program:

1. The routine to write out the file illustrates the use of the **mvnch** function, which returns the character in a window at a given position.
2. The data structure used here does not have a provision for keeping track of the number of characters in a line, or the number of lines in the file, so trailing blanks are eliminated when the file is written out.
3. The program uses built-in **curses** functions:
 - **insch**
 - **delch**
 - **insertln**
 - **deleteln**

These functions behave much as the similar functions on intelligent terminals behave, inserting and deleting a character or line.

4. The command interpreter accepts ASCII characters AND special function keys.

This is important because some editors are **modeless**, using nonprinting characters for commands, so both arrow keys and ordinary ASCII characters should be handled.

On the other hand, not all terminals have arrow keys, so a program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for each special key.

5. The **addstr** function is roughly like the C language **fputs** function, which writes out a string of characters. Like **fputs**, **addstr** does NOT add a trailing newline. The **addstr** function works with strings with same way that **addch** works with characters.
6. The **mvaddstr** function is the "move" version of **addstr**, which moves to the given location in the window before writing.
7. The CTRL-L command illustrates a feature especially useful in **curses** programs.

Often some program, beyond the control of **curses**, may write something to the screen, or some line noise may mess up the screen and **curses** may not have been able to keep track of what should and should not be on the screen. The CTRL-L command clears and redraws the screen. This is done with the call to **clearok(curscr)**, which sets a flag causing the next **refresh** to first clear the screen. Then **refresh** is called to force the redraw.

8. The **flash()** function, flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and is particularly useful if the bell bothers someone within earshot of the user.
9. The routine **beep()** is similar to the **flash()** function, but is called when a real beep is desired. If for some reason the terminal is unable to beep, but is able to flash, a call

CURSES

to beep will flash the screen.

10. Another important point is that the input command is terminated by CTRL-d, NOT "<ESC>" (escape). Even though <ESC> is one of the few special keys available on every keyboard (<RETURN> and <BREAK> are the only others), use of <ESC> as a separate key introduces an ambiguity.

Most terminals use sequences of characters beginning with <ESC> ("escape sequences") to control the terminal, and some have special keys that send <ESC> sequences to the computer. If the computer receives an <ESC> from the terminal, there is no way to distinguish whether the user pushed the <ESC> key, or whether a special key was pressed.

One way to handle the ambiguity is to wait, which is what **curses** does for up to 1 second. Then, if another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key.

While this strategy works most of the time, it is not "foolproof." It is possible for the user to press <ESC>, then to type another key quickly, which causes **curses** to think a special key has been pressed.

Also, there is a one second pause until the <ESC> can be passed to the user program, resulting in slower response to the <ESC> key.

The following is the seventh and last example program, "EDITOR:"

```
/*  
 * EDITOR. A screen-oriented editor. The user  
 * interface is similar to a subset of vi.  
 * The buffer is kept in stdscr itself to simplify  
 * the program.  
 */
```

```
#include < curses.h>
```

```
#define CTRL(c) ('c' & 037)
```

```
main(argc, argv)  
char **argv;  
{  
    int i, n, l;  
    int c;  
    FILE *fd;  
  
    if (argc != 2) {  
        fprintf(stderr, "Usage: edit file\n");  
        exit(1);  
    }  
  
    fd = fopen(argv[1], "r");  
    if (fd == NULL) {  
        perror(argv[1]);  
        exit(2);  
    }  
  
    initscr();  
    cbreak();  
    nonl();  
    noecho();  
    idlok(stdscr, TRUE);  
    keypad(stdscr, TRUE);  
  
    /* Read in the file */  
    while ((c = getc(fd)) != EOF)  
        addch(c);
```

CURSES

```
    fclose(fd);

    move(0,0);
    refresh();
    edit();

    /* Write out the file */
    fd = fopen(argv[1], "w");
    for (l=0; l<23; l++) {
        n = len(l);
        for (i=0; i<n; i++)
            putc(mvinch(l, i), fd);
        putc('\n', fd);
    }
    fclose(fd);

    endwin();
    exit(0);
}

len(lineno)
int lineno;
{
    int linelen = COLS-1;

    while (linelen >=0 && mvinch(lineno, linelen) == ' ')
        linelen--;
    return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
    int c;

    for (;;) {
        move(row, col);
```

```
refresh();
c = getch();
switch (c) { /* Editor commands */

/* hjkl and arrow keys: move cursor */
/* in direction indicated */
case 'h':
case KEY_LEFT:
    if (col > 0)
        col--;
    break;

case 'j':
case KEY_DOWN:
    if (row < LINES-1)
        row++;
    break;

case 'k':
case KEY_UP:

    if (row > 0)
        row--;
    break;

case 'l':
case KEY_RIGHT:
    if (col < COLS-1)
        col++;
    break;

/* i: enter input mode */
case KEY_IC:
case 'i':
    input();
    break;

/* x: delete current character */
case KEY_DC:
```

CURSES

```
case 'x':
    delch();
    break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
    move(++row, col=0);
    insertln();
    input();
    break;

/* d: delete current line */
case KEY_DL:
case 'd':
    deleteln();
    break;

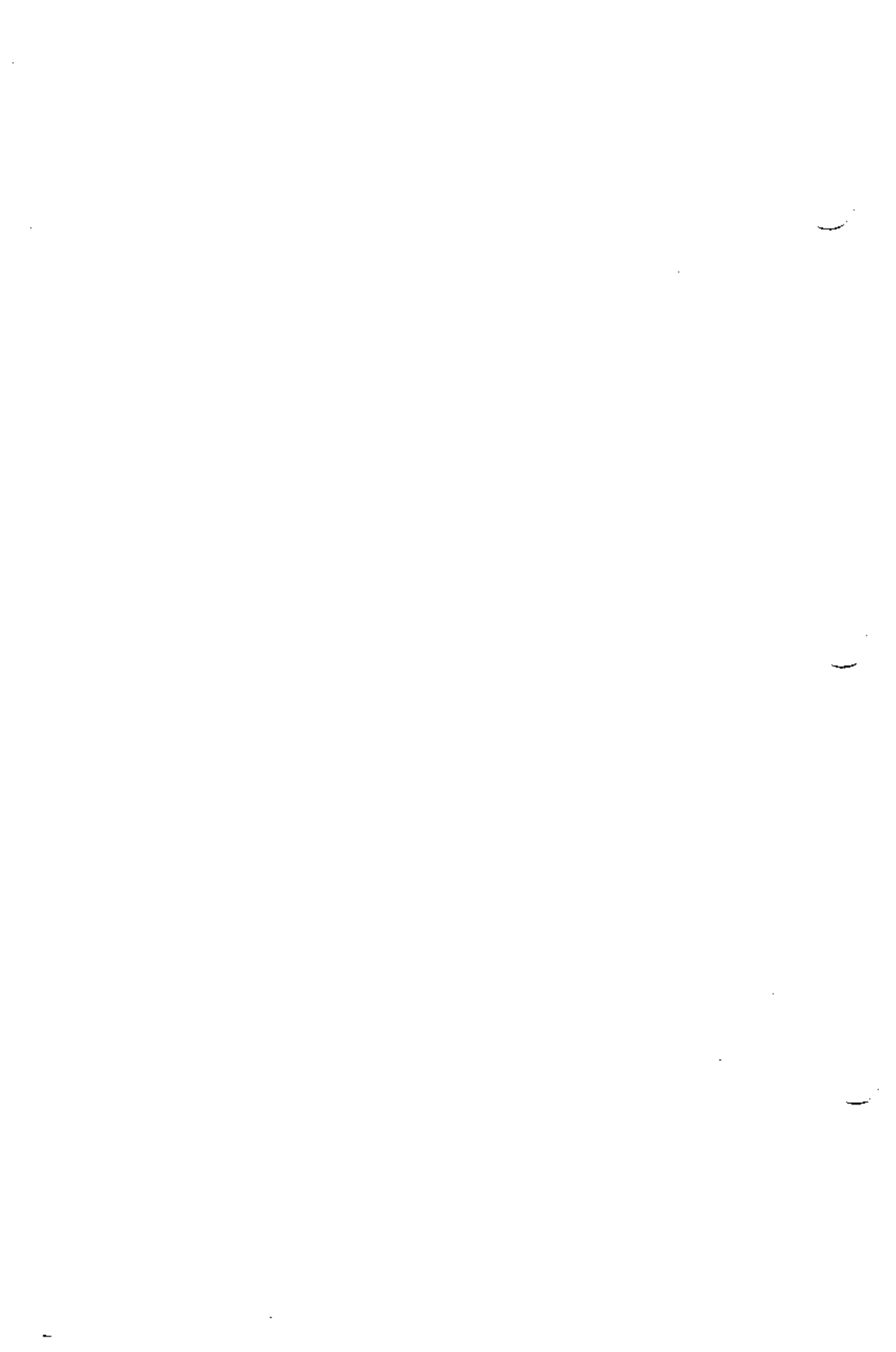
/* CTRL-L: redraw screen */
case KEY_CLEAR:
case CTRL(L):
    clearok(curscr);
    refresh();
    break;

/* w: write and quit */
case 'w':
    return;

/* q: quit without writing */
case 'q':
    endwin();
    exit(1);
default:
    flash();
    break;
}
}
```

```
/*
 * Insert mode: accept characters and insert them.
 * End with CTRL-d or EIC
 */
input()
{
    int c;

    stdout();
    mvaddstr(LINES-1, COLS-20, "INPUT MODE");
    standend();
    move(row, col);
    refresh();
    for (;;) {
        c = getch();
        if (c == CTRL(D) | c == KEY_EIC)
            break;
        insch(c);
        move(row, ++col);
        refresh();
    }
    move(LINES-1, COLS-20);
    clrtoeol();
    move(row, col);
    refresh();
}
```



Chapter 12: UUCP

CONTENTS

1. Introduction	1
2. The Uucp Network	1
2.1 Network Hardware	2
2.2 Network Topology	2
2.2.1 Hardware Topology	3
2.2.2 Software Topology	4
2.3 Forwarding	6
2.4 Security	7
2.5 Software Structure	7
2.6 Rules of the Road	7
2.6.1 Queuing	7
2.6.2 Dialing and the DDD Network	8
2.6.3 Scheduling and Polling	8
2.6.4 Retransmissions and Hysterisis	9
2.6.5 Purging and Cleanup	9
2.7 Special Places: The Public Area	9
2.8 Permissions	9
2.8.1 File Level Protection	9
2.8.2 System Level Protection	10
2.8.3 Forwarding Permissions	10
3. Network Usage	10
3.1 Name Space	10
3.1.1 Naming Conventions	11
3.2 Forwarding Syntax	12
3.3 Types of Transfers	13

3.3.1	Transmissions of Files to a Remote	13
3.3.2	Fetching Files From a Remote	13
3.3.3	Switching	13
3.3.4	Broadcasting	14
3.4	Remote Executions	14
3.5	Spooling	14
3.6	Notification	14
3.7	Tracking and Status	15
3.7.1	The Job ID	15
3.8	Job Status	16
3.9	Network Status	17
3.10	Job Control	17
3.10.1	Job Termination	17
3.10.2	Requeuing a Job	18
3.10.3	Network Names	18
4.	Utilities That Use Uucp	18
4.1	The Stockroom	18
4.2	Mail	18
4.3	Netnews	19
4.4	Uuto	19
4.5	Other Applications	19

LIST OF FIGURES

Figure 12-1.	UUCP Nodes	3
Figure 12-2.	UUCP Network Excluding One Node	5
Figure 12-3.	UUCP Network With Several Levels of Permissions	5

Chapter 12

UUCP: UNIX TO UNIX SYSTEM COPY

1. Introduction

The **uucp** network has provided a means of information exchange between UNIX systems over the direct distant dialing network for several years. This chapter provides you with the background to make use of the network.

The first half of the document discusses concepts. Understanding these basic principles helps the user make the best possible use of the **uucp** network. The second half explains the use of the user level interface to the network and provides numerous examples.

There are several major uses of the network. Some of the uses are:

- Distribution of software
- Distribution of documentation
- Personal communication (mail)
- Data transfer between closely sited machines
- Transmission of debugging dumps and data exposing bugs
- Production of hard copy output on remote printers.

2. The Uucp Network

The **uucp(1)** network is a network of UNIX systems that allows file transfer and remote execution to occur on a network of UNIX systems. The *extent* of the network is a function of both the interconnection hardware and the controlling network software. Membership in the network is tightly controlled via the software to preserve the integrity of all members of the network. You cannot use the **uucp** facility to send files to systems that are not part of the **uucp**

UUCP

network. The following parts describe the topology, services, operating rules, etc., of the network to provide a framework for discussing use of the network.

2.1 Network Hardware

The **uucp** was originally designed as a dialup network so that systems in the network could use the **DDD** network to communicate with each other. The three most common methods of connecting systems are:

1. Connecting two UNIX systems directly by cross-coupling (via a null modem) two of the computers ports. This means of connection is useful for only short distances (several hundred feet can be achieved although the RS232 standard specifies a much shorter distance) and is usually run at high speed (9600 baud). These connections run on asynchronous terminal ports.
2. Using a modem (a private line or a limited distance modem) to *directly* connect processors over a private line (using 103- or 212-type data sets).
3. Connecting a processor to another system through a modem, an automatic calling unit (ACU), and the **DDD** network. This is by far the most common interconnection method, and it makes available the largest number of connections.

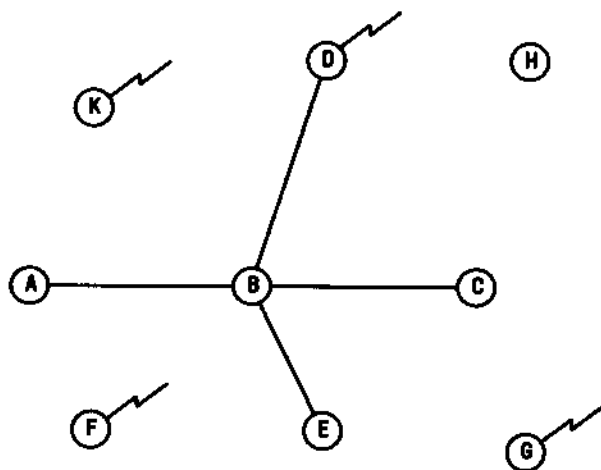
The **uucp** could be extended to use higher speed media (e.g., **HYPERchannel**, **Ethernet**, etc.), and this possibility is being explored for future UNIX system releases. Some sites already support local modifications to **uucp** to allow the use of **Datakit**, **X.25** (permanent virtual circuits), and calling through data switches.

2.2 Network Topology

A large number of connections between systems are possible via the **DDD** network. The topology of the network is determined by both the hardware connections and the software that control the network. The next two parts deal with how that topology is controlled.

2.2.1 Hardware Topology

As discussed earlier, it is possible to build a network using permanent or dial up connections. In Figure 12.1, a group of systems (A, B, C, D, and E) are shown connected via hard-wired lines. All systems are assumed to have some answer-only data sets so that remote users or systems can be connected.



LEGEND



-  - AUTOMATIC CALLING UNIT
-  - COMPUTER SYSTEM

Figure 12-1. UUCP Nodes

UUCP

A few systems have automatic calling units (K, D, F, and G) and one system (H) has no capability for calling other systems. Users should be aware that the network consists of a series of point-to-point connections (A-B, B-C, D-B, E-B) even though it appears in Figure 12.1 that A and C are directly connected through B. The following observations are made:

1. System H is isolated. It can be made part of the network by arranging for other systems to *poll* it at fixed intervals. This is an important concept to remember since transfers from systems that are *polled* do not leave the system until that system is called by a polling system.
2. Systems K, F, G, and D easily reach all other systems since they have calling units.
3. If system A (E or G) wishes to send a file to H (K, F, or G), it must first send it to D (via system B) since D is the only system with a calling unit.

2.2.2 Software Topology

The hardware capability of systems in the network defines the *maximum* number of connections in the network. The software at each node restricts the access by other systems and thereby defines the extent of the network. The systems of Figure 12.1 can be configured so that they appear as a network of systems that have equal access to each other or some restrictions can be applied. As part of the security mechanism used by *uucp*, the extent of access that other systems have can be controlled at each node. Figures 12.2 and 12.3 show how the network might appear at one node.

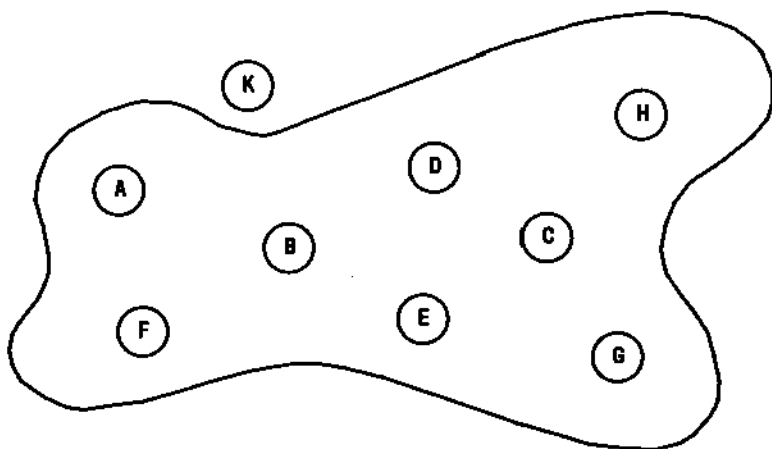


Figure 12-2. UUCP Network Excluding One Node

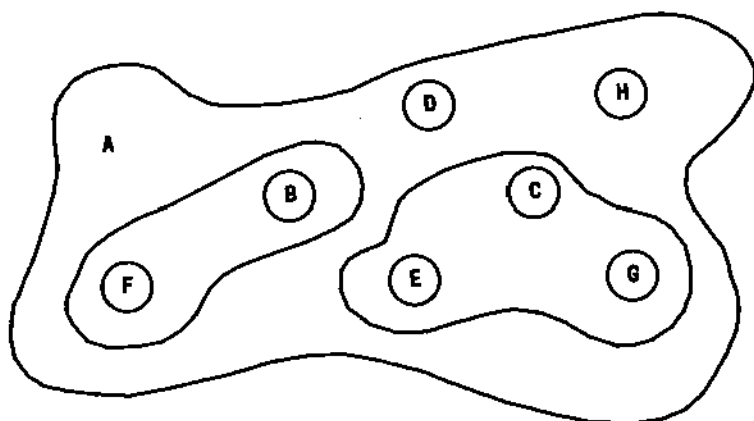


Figure 12-3. UUCP Network With Several Levels of Permissions

UUCP

Access is available from all systems in Figure 12.2, however, in Figure 12.3 some of the systems have been configured to have greater or less access privileges than others (i.e., systems C, E, and G have one set of access privileges, systems F and B have another set, etc.).

The **uucp** uses the UNIX system password mechanism coupled with a system file (*/usr/lib/uucp/L.sys*) and a file system permission file (*/usr/lib/uucp/USERFILE*) to control access between systems. The password file entries for **uucp** (usually, **luucp**, **nuucp**, **uucp**, etc.) allow only those remote systems that know the passwords for these IDs to access the local system. (Great care should be taken in revealing the password for these **uucp** logins since knowing the password allows a system to join the network.) The system file (*/usr/lib/uucp/L.sys*) defines the remote systems that a local host knows about. This file contains all information needed for a local host to contact a remote system (including system name, password, login sequence, etc.) and as such is protected from viewing by ordinary users.

In summary, while the available hardware on a network of systems determines the connectivity of the systems, the combination of *password* file entries and the **uucp** system files determine the extent of the network.

2.3 Forwarding

One of the additions to **uucp** is a limited forwarding capability whereby systems that are part of the network can forward files through intermediate nodes. For example, in Figure 12.1, it is possible to send a file between node A and C through intermediate node B. For security reasons, when forwarding, files may only be transmitted to the *public* area or fetched from the remote systems *public* area.

2.4 Security

The most critical feature of any network is the security that it provides. Users are familiar with the security that UNIX system provides in protecting files from access by other users and in accessing the system via *passwords*. In building a network of processors, the notion of security is widened because access by a wider community of users is granted. Access is granted on a *system* basis (that is, access is granted to *all* users on a remote system). This follows from the fact that the process of sending (receiving) a file to (from) another system is done via daemons that use one special user ID(s). This user ID(s) is granted (denied) access to the system via the **uucp** system file (*/usr/lib/uucp/L.sys*) and the areas that the system has access to is controlled by another file (*/usr/lib/uucp/USERFILE*). For example, access can be granted to the entire file system tree or limited to specific areas.

2.5 Software Structure

The **uucp** network is a batch network. That is, when a request is made, it is spooled for later transmission by a daemon. This is important to users because the success or failure of a command is only known at some later time via **mail(1)** notification. For most transfers, there is little trouble in transmitting files between systems, however, transmissions are occasionally delayed or fail because a remote system cannot be reached.

2.6 Rules of the Road

There are several rules by which the network runs. These rules are necessary to provide the smooth flow of data between systems and to prevent duplicate transmissions and lost jobs. The following chapters outline these rules and their influence on the network.

2.6.1 Queuing

Jobs submitted to the network are assigned a sequence number for transmission. Jobs are represented by a file (or files) in a common spool directory (*/usr/spool/uucp*). When a file transfer daemon

UUCP

(uucico) is started to transmit a job, it selects a system to contact and then transmits *all* jobs to that system. Before breaking off the conversation, any jobs to be received from that remote system are accepted. The system selected as the one to contact is randomly selected if there is work for more than one system. In releases of uucp prior to UNIX system 5.0, the first system appearing in the spool directory is selected so preference is given to the most recently spawned jobs. Uucp may be sending to or receiving from many systems simultaneously. The number of incoming requests is only limited by the number of connections on the system, and the number of outgoing transfers is limited by the number of ACUs (or direct connections).

2.6.2 Dialing and the DDD Network

In order to transfer data between processors that are not directly connected, an auto dialer is used to contact the remote system. There are several factors that can make contacting a remote system difficult.

1. All lines to the remote system may be busy. There is a mechanism within uucp that restricts contact with a remote system to certain times of the day (week) to minimize this problem.
2. The remote system may be down.
3. There may be difficulty in dialing the number (especially if a large sequence of numbers involving access through PBXs is involved). The dialing algorithm tries dialing a number *twice* and the algorithm used to dial remote systems is not perfect, particularly when intermediate dial tones are involved.

2.6.3 Scheduling and Polling

When a job is submitted to the network, an attempt to contact that system is made *immediately*. Only one conversation at a time can exist between the same two systems.

Systems that are *polled* can do nothing to force immediate transmission of data. Jobs will only be transmitted when the system is polled (hourly, daily, etc.) by a remote system.

2.6.4 Retransmissions and Hysterisis

The **uucp** network is fairly persistent in its attempt to contact remote systems to complete a transmission. To prevent **uucp** from continually calling systems that are unavailable, *hysterisis* is built into the algorithm used to contact other systems. This mechanism forces a minimum fixed delay (specifiable on a per system basis) to occur before another transmission can take place to that system.

2.6.5 Purging and Cleanup

Transfers that cannot be completed after a defined period of time (72 hours is the value that is set when the system is distributed) are deleted and the user is notified.

2.7 Special Places: The Public Area

In order to allow the transfer of files to a system for which a user does not have a login on, the *public* directory (usually kept in */usr/spool/uucppublic*) is available with general access privileges. When receiving files in the *public* area, the user should dispose of them quickly as the administrative portion of **uucp** purges this area on a regular basis.

2.8 Permissions

2.8.1 File Level Protection

In transferring files between systems, users should make sure that the destination area is writable by **uucp**. The **uucp** daemons preserve execute permission between systems and assign permission 0666 to transferred files.

UUCP

2.8.2 System Level Protection

The system administrator at each site determines the global access permissions for that processor. Thus, access between systems may be confined by the administrator to only some sections of the file system.

2.8.3 Forwarding Permissions

The forwarding feature is a recent addition to the **uucp** package. You should be aware that

1. When forwarding is attempted through a node that is running an old version of **uucp**, the transmission fails.
2. Nodes that allow forwarding can restrict the forwarding feature in several ways.
 - a. Forwarding is allowed for only certain users.
 - b. Forwarding to certain destination nodes (e.g., Australia) should be avoided.
 - c. Forwarding for selected source nodes is allowed.
3. The most important restriction is that forwarding is allowed only for files sent to or fetched from the *public* area.

3. Network Usage

The following parts discuss the user interface to the network and give examples of command usage.

3.1 Name Space

In order to reference files on remote systems, a syntax is necessary to uniquely identify a file. The notation must also have several defaults to allow the reference to be compact. Some restrictions must also be placed on pathnames to prevent security violations. For example, pathnames may not include *..* as a component because it is difficult to determine whether the reference is to a restricted area.

3.1.1 Naming Conventions

Uucp uses a special syntax to build references to files on remote systems. The basic syntax is

`system-name!pathname`

where the *system-name* is a system that uucp is aware of. The *pathname* part of the name may contain any of the following:

1. A fully qualified *pathname* such as

`mhtsa!usr/you/file`

The *pathname* may also be a directory name as in

`mhtsa!usr/you/directory`

2. The login directory on a remote may be specified by use of the `~` character. The combination `~user` references the login directory of a user on the remote system. For example,

`mhtsa!~adm/file`

would expand to

`mhtsa!usr/sys/adm/file`

if the login directory for user *adm* on the remote system is */usr/sys/adm*.

3. The *public* area is referenced by a similar use of the prefix `~/user` preceding the *pathname*. For example,

`mhtsa!~/you/file`

would expand to

`mhtsa!usr/spool/uucp/you/file`

UUCP

if */usr/spool/uucp* is used as the spool directory.

4. Pathnames not using any of the combinations or prefixes discussed above are prefixed with the current directory (or the login directory on the remote). For example,

mhtsa!file

would expand to

mhtsa!/usr/you/file

The naming convention can be used in reference to either the *source* or *destination* file names.

3.2 Forwarding Syntax

Uucp can pass files between systems via intermediate nodes. This is done via a variation of the *bang (!)* syntax that describes the path to be taken to reach that file. For example, a user on system *a* wishing to transmit a file to system *e* might specify the transfer as

uucp file b!c!d!e!~/you/file

if the user desires the request to be sent through *b*, *c*, and *d* before reaching *e*. Note that the pathname is the path that the file would take to reach node *e*. Note also that the destination *must* be specified as the *public* area. Fetching a file from another system via intermediate nodes is done similarly. For example,

uucp b!c!d!e!~/you/file x

fetches *file* from system *e* and renames it *x* on the local system. The forwarding prefix is the path *from* the local system and not the path from the remote to the local system.

3.3 Types of Transfers

Uucp has a very flexible command syntax for file transmission. The following chapters give examples of different combinations of transfers.

3.3.1 Transmissions of Files to a Remote

Any number of files can be transferred to a remote system via **uucp**. The syntax supports the *****, **?** and **[..]** metacharacters. For example,

```
uucp *.[ch] mhtsa!dir
```

transfers *all* files whose name ends in *c* or *h* to the directory *dir* in the users login directory on *mhtsa*.

3.3.2 Fetching Files From a Remote

Files can be fetched from a remote system in a similar manner. For example,

```
uucp mhtsa!*.[ch] dir
```

will fetch all files ending in *c* or *h* from the users login directory on *mhtsa* and place the copies in the subdirectory *dir* on the local system.

3.3.3 Switching

Transmission of files can be arranged in such a way that the local system effectively acts as a switch. For example,

```
uucp mhtsb!files mhtsa!filed
```

will fetch *files* from the users login directory on *mhtsb*, rename it as *filed*, and place it in the login directory on *mhtsa*.

UUCP

3.3.4 Broadcasting

Broadcast capability (that is, copying a file to many systems) is *not* supported by **uucp**, however, it can be simulated via a shell script as in

```
for i in mhtsa mhtsb mhtsd
do
    uucp file $i!broad
done
```

Unfortunately, one **uucp** command is spawned for each transmission so that it is not possible to track the transfer as a single unit.

3.4 Remote Executions

The remote execution facility allows commands to be executed remotely. For example,

```
uux '!diff mhtsa!/etc/passwd mhtsd!/etc/passwd > !pass.diff'
```

will execute the command **diff**(1) on the password file on *mhtsa* and *mhtsd* and place the result in *pass.diff*.

3.5 Spooling

To continue modifying a file while a copy is being transmitted across the network, the **-C** option should be used. This forces a *copy* of the file to be queued. The default for **uucp** is not to queue copies of the files since it is wasteful of both Central Processing Unit time and storage. For example, the following command forces the file *work* to be copied into the spool directory before it is transmitted.

```
uucp -C work mhtsa!~/you/work
```

3.6 Notification

The success or failure of a transmission is reported to users asynchronously via the **mail**(1) command. A new feature of **uucp** is to

provide notification to the user in a file (of the users choice). The choices for notification are:

1. Notification returned to the requesters system (via the `-m` option). This is useful when the requesting user is distributing files to other machines. Instead of logging onto the remote machine to read mail, mail is sent to the requester when the copy is finished.
2. A variation of the `-m` option is to force notification in a file (using the `-mfile` option where *file* is a file name). For example,

```
uucp -mans /etc/passwd mhtsb!/dev/null
```

sends the file */etc/passwd* to system *mhtsb* and place the file in the bit bucket (*/dev/null*). The status of the transfer is reported in the file *ans* as,

```
uucp job 0306 (8/20-23:08:09) (0:31:23) /etc/passwd copy succeeded
```

3. `Uux(1)` always reports the exit status of the remote execution unless notification is suppressed (via the `-n` option).

3.7 Tracking and Status

The most pervasive change to the `uucp` package is revising the internal formatting of jobs so that each invocation of `uucp` or `uux(1)` corresponds to a single job. It is now possible to associate a single job number with each command execution so that the job can be terminated or its status obtained.

3.7.1 The Job ID

The default for the `uucp` and `uux` command is *not* to print the job number for each job. This was done for compatibility with previous versions of `uucp` and to prevent the many shell scripts built around `uucp` from printing job numbers. If the following environment variable

UUCP

JOBNO = ON

is made part of the users environment and exported, **uucp** and **uux** prints the job number. Similarly, if the user wishes to turn the job numbers off, the environment variable is set as follows:

JOBNO = OFF

If you wish to force printing of job numbers without using the environment mechanism, use the **-j** option. For example,

```
uucp -j /etc/passwd mhtsb!/dev/null
uucp job 282
```

forces the job number (282) to be printed. If the **-j** option is not used, the IDs of the jobs (belonging to the user) are found by using the **uustat(1)** command. This provides the job number. For example,

```
uustat
0282 tom mhtsb 08/20-21:47 08/20-21:47 JOB IS QUEUED
0272 tom mhtsb 08/20-21:46 08/20-21:46 JOB IS QUEUED
```

shows that the user has two jobs (282 and 272) queued.

3.8 Job Status

The **uustat** command allows a user to check on one or all jobs that have been queued. The ID printed when a job is queued is used as a key to query status of the particular job. An example of a request for the status of a given job is

```
uustat -j0711

0711 tom mhtsb 07/30-02:18 07/30-02:18 JOB IS QUEUED
```

There are several status messages that may be printed for a given job; the most frequent ones are JOB IS QUEUED and JOB COMPLETED (meanings are obvious). The manual page for `uustat` lists the other status messages.

3.9 Network Status

The status of the last transfer to each system on the network is found by using the `uustat` command. For example,

```
uustat -mall
```

reports the status of the last transfer to all of the systems known to the local system. The output might appear as

mhb5c	08/10-12:35	CONVERSATION SUCCEEDED
resear	08/20-17:01	CONVERSATION SUCCEEDED
minimo	07/22-16:31	DIAL FAILED
austra	08/20-18:36	WRONG TIME TO CALL
ucbvax	08/20-20:37	LOGIN FAILED

where the status indicates the time and state of the last transfer to each system. When sending files to a system that has not been contacted recently, it is a good idea to use `uustat` to see *when* the last access occurred (because the remote system may be down or out of service).

3.10 Job Control

With the unique job ID generated for each `uucp` or `uux` command, it is possible to control jobs in the following ways.

3.10.1 Job Termination

A job that consists of transferring many files from several different systems can be terminated using the `-k` option of `uustat`. If any part of the job has left the system, then only the *remaining* parts of

UUCP

the job on the local system is terminated.

3.10.2 Requeuing a Job

The `uucp` package clears out its working area of jobs on a regular basis (usually every 72 hours) to prevent the buildup of jobs that cannot be delivered. The `-r` option is used to force the date of a job to be changed to the current date, thereby lengthening the time that `uucp` attempts to transmit the job. It should be noted that the `-r` option does not impart *immortality* to a job. Rather, it only postpones deleting the job during housekeeping functions until the next cleanup.

3.10.3 Network Names

Users may find the names of the systems on the network via the `uname(1)` command. Only the *names* of the systems in the network are printed.

4. Utilities That Use Uucp

There are several utilities that rely on `uucp` or `uux(1)` to transfer files to other systems. The following parts outline the more important of these functions. This increases awareness of the extent of the use of the network.

4.1 The Stockroom

The UNIX system *stockroom* is a facility whereby a library of source can be maintained at a central location for distribution of source or bug fixes. Access to and distribution of library entries is controlled by shell scripts that use `uucp`.

4.2 Mail

The `mail(1)` command uses `uux` to forward mail to other systems. For example, when a user types

```
mail mhmts!tom
```

the **mail** command invokes **uux** to execute **rmail** on the remote system (**rmail** is a link to the **mail** command). Forwarding mail through several systems (e.g., **mail a!b!tom**) does not use the **uucp** forwarding feature but is simulated by the **mail** command itself.

4.3 Netnews

The **netnews** software (**usenet**) that is locally supported on many systems uses **uux** in much the same way that **mail** does to broadcast *network mail* to systems subscribing to news categories.

4.4 Uuto

The **uuto(1)** command uses the **uucp** facility to send files while allowing the local system to control the file access. Suppose your login is **emsgene** and you are on system **aaaaa**. You have a friend (David) on system **bbbbbb** with a login name of **wldmc**. Also assume that both systems are networked to each other [See **uname(1)**]. To send files using **uuto**, enter the following:

```
uuto filename aaaaa!wldmc
```

where **filename** is the name of a file to be sent. The files are sent to a public directory defined in the **uucp** source. In this example, David will receive the following mail:

```
>From nuucp Tue Jan 25 11:09:55 1983
/usr/spool/uucppublic/receive/wldmc/aaaaa\
//filename from aaaaa!emsgene arrived
```

See **uuto(1)** for more details.

4.5 Other Applications

The Office Automation System (OAS) uses **uux** to transmit electronic mail between systems in a manner similar to the standard **mail** command. Some sites have replaced utilities such as **lp(1)**, with shell scripts that invoke **uux** or **uucp**. Other sites use the **uucp** network as a backup for higher speed networks (e.g., PCL, NSC

UUCP

HYPERchannel, etc.).

Colophon

Composed at UniSoft Systems Inc.
on the UniPlus⁺ Operating System
Designed by the Documentation Department
Printed in Times Roman on Sequoia Matt

