CRAY RESEARCH, INC.

CRAY-1®
COMPUTER SYSTEMS

SKOL
REFERENCE MANUAL

SR-0033

CRAY RESEARCH, INC.

CRAY-1®
COMPUTER SYSTEMS

SKOL
REFERENCE MANUAL

SR-0033

Copyright© 1976, 1977, 1978, 1979, 1980, 1981 by CRAY RESEARCH, INC.
This manual or parts thereof may not be reproduced in any form without permission of CRAY RESEARCH, INC.

RECORD OF REVISION CRAY RESEARCH, INC. PUBLICATION NUMBER SR-0033

Each time this manual is revised and reprinted, all changes issued against the previous version in the form of change packets are incorporated into the new version and the new version is assigned an alphabetic level. Between reprints, changes may be issued against the current version in the form of change packets. Each change packet is assigned a numeric designator, starting with 01 for the first change packet of each revision level.

Every page changed by a reprint or by a change packet has the revision level and change packet number in the lower righthand corner. Changes to part of a page are noted by a change bar along the margin of the page. A change bar in the margin opposite the page number indicates that the entire page is new; a dot in the same place indicates that information has been moved from one page to another, but has not otherwise changed.

Requests for copies of Cray Research, Inc. publications and comments about these publications should be directed to:

CRAY RESEARCH, INC.,
1440 Northland Drive,
Mendota Heights, Minnesota 55120

Revision	Description
	September, 1980 - Original printing.
01	June, 1981 - This change packet brings the manual into agreement with version 1.10 of the operating system. Sections 2, 5, and 10 are reorganized. Major changes include the following additions: the SKOL EQUIVALENCE statement, the TEMPLATE...ENDTEMPLATE data structure, the NOTRACE statement, local pointer declarations, recursion stack accessing, BUFFER IN, BUFFER OUT, UNIT, BUFFER_IN LENGTH statements, direct access I/O, CFT compiler directives, and structured text. Miscellaneous technical and editorial changes are also included.

PREFACE

SKOL is a macro-based programming language that is translated into FORTRAN code, which is then compiled by a FORTRAN compiler. SKOL offers the user a more structured approach to programming than is possible with a language such as FORTRAN. The wide variety of structures available in SKOL eliminate the need for a GO TO statement and increase the user's options.

This publication describes the features of the SKOL language. Because of SKOL's close association with FORTRAN, this publication assumes that the reader knows FORTRAN. Detailed information about CRAY-1 FORTRAN is presented in the CRAY-1 FORTRAN (CFT) Reference Manual (publication SR-0009).

Understanding the SKOL macro translator is not necessary for writing programs in SKOL but is helpful in supplying new macros that extend the SKOL language.

FOREWORD

SKOL is a high-level programming language developed by Charles T. Zahn at the Stanford Linear Accelerator Center (SLAC) in Stanford, Cal. It was first documented by Zahn in a reference manual dated December, 1976.[1] SKOL is the descendant of earlier versions of the language, SGOL (Structured alGOrithmic Language) and MINISGOL. Frank D. Huber of Cray Research, Inc. extended and refined the language.

SKOL is also the name of the CRAY-1 FORTRAN (CFT) program that serves as the macro translator for the SKOL language. The SKOL macro translator is the descendant of MORTRAN2[2] and MORTRAN,[3] which were developed by A. James Cook, L. J. Shustek, and J. E. Zolnowsky at SLAC. It translates the SKOL-language program into FORTRAN code under the control of a set of string-processing macro instructions. The SKOL language is defined entirely by the set of macros contained in a file named SKOLTXT. Therefore the SKOL macro translator may be thought of as a table-driven program, with SKOLTXT as the table.

Cook envisioned FORTRAN as potentially a universal assembly language, which could be used as a base language for many higher level--and possibly special purpose--languages. Each language would be defined by a unique set of macro definitions, as Cook saw it, and all would retain the attributes of portability and extensibility because of their relationship to FORTRAN.

Zahn's creation of SKOL was essentially an experiment to prove that good diagnostic messages could be produced by a macro-based translator. Zahn did not change Cook's MORTRAN2 macro processor except to enlarge its storage area. In adapting MORTRAN2 to make SKOL a Cray Research product, Huber added many capabilities to the macro processor, roughly doubling its size.

The added capabilities make the Cray Research version of SKOL somewhat less portable than Zahn's original language. The present product is designed to translate source code into FORTRAN statements that are compatible with the CFT compiler. The same code may not be compatible with another compiler unless modifications are made to the SKOL macro translator. A listing of the translator includes comments indicating machine-dependent areas and suggesting modifications that may be necessary to adapt the translator and the SKOLTXT macro file to another machine.

CONTENTS

PREFACE iii

FOREWORD v

1. INTRODUCTION 1-1

OVERVIEW 1-1
Control structures 1-1
Data structures 1-3
Other features 1-3
Extensibility 1-4
DIAGNOSTIC MESSAGES 1-4
FORMAL SYNTAX REPRESENTATION 1-5

2. SYNTAX 2-1

CHARACTER SET 2-1
LITERAL STRINGS 2-2
IDENTIFIERS USED IN SKOL 2-3
STATEMENT LENGTH 2-3
STATEMENT TERMINATORS 2-4
COMMENTS 2-4
VARIABLE DECLARATIONS 2-5
EQUIVALENCE STATEMENT 2-6
BASED VARIABLES 2-6
EXTERNAL DECLARATIONS 2-7
IMPLICIT VARIABLE DECLARATION 2-8
DATA STATEMENT 2-9
RELATIONAL OPERATORS 2-9
LOGICAL OPERATORS 2-10
ODD AND EVEN 2-11
BOOLEAN 2-11

3. PROGRAM CONTROL STRUCTURES 3-1

IF...ELSEIF...ELSE...ENDIF 3-1
IF FIRST TIME 3-2
UNTIL...THEN...ENDUNTIL 3-3
UNTIL...ENDUNTIL 3-5

3. PROGRAM CONTROL STRUCTURES (continued)
REPEAT...ENDREPEAT 3-6
LOOP...WHILE...ENDLOOP 3-7
DO...ENDDO 3-8
FOR...ENDFOR 3-10
INCR, DECR 3-11

4. USER-DEFINED TYPES AND THE SCALAR CASE STRUCTURE 4-1

TYPE DECLARATION 4-1
NUMERIC TYPES 4-4
SCALAR CASE STRUCTURE (WHEN...ENDWHEN) 4-4
IN 4-6
FIRST, LAST 4-7

5. COMPLEX DATA STRUCTURES 5-1

PARTIAL-WORD VARIABLES 5-1
TEMPLATES 5-3
TABLES 5-5
TABLE POINTERS 5-7
RECORD TYPES 5-11
RECORD POINTERS 5-13
MAKEAVAIL statement 5-15
NEW statement 5-15
FREE statement 5-16
POINTER USAGE. 5-16
WITH...ENDWITH 5-17
LINK...ENDLINK 5-18

6. TEXT MANIPULATION. 6-1

CONSTANT 6-1
PARAMETER. 6-2
MACRO 6-3
DEFINE 6-5
Positional parameters 6-6
Special handling for blanks 6-7
Accidental recursion 6-9
TRACE 6-9
NOTRACE 6-12
VALIDATE 6-13
WHEN 6-14
UNTIL 6-14
FOR 6-15
RECUR_UNDER, RECUR_OVER 6-15
LOOP, CALL 6-15

6. TEXT MANIPULATION (continued)
CONDITIONAL COMPILATION 6-15
A series of conditional statements 6-16
Structured text 6-18
A single conditional statement 6-18
ASSERT 6-19

7. STRINGS 7-1

DELETE 7-2
REPLACE 7-3
INSERT 7-4
SPECIAL NOTATION 7-4
CATENATE 7-5
STRING ASSIGNMENT STATEMENT 7-6
MOVE 7-6
STRING COMPARISON EXTENSION 7-7

8. PROGRAM MODULES 8-1

THE MAIN SEGMENT 8-2
SUBROUTINES 8-3
FUNCTIONS 8-4
ENTRY STATEMENT 8-4
RETURN STATEMENT 8-6
BLOCKDATA SEGMENTS 8-7
ROUTINES 8-7
PROCESSES AND COROUTINES 8-8
RECURSIVE ROUTINES 8-12
LOCAL POINTER DECLARATIONS 8-15
ACCESSING THE RECURSION STACK 8-16

9. INPUT/OUTPUT 9-1

OUTPUT 9-1
INPUT 9-5
REMARK 9-6
FORMAT 9-7
READ, WRITE 9-7
DECODE, ENCODE 9-8
LOG 9-9
DATA ITERATION (THE FOR CLAUSE) 9-9
END=..., ERR=... 9-11
PAGE, SKIP 9-12
BUFFER IN, BUFFER OUT, UNIT, BUFFER_IN_LENGTH 9-13
DIRECT-ACCESS I/O 9-13

10. USING SKOL ON THE CRAY-1 10-1

SKOL AND CFT CONTROL STATEMENTS 10-1
SKOL, FORTRAN, AND CAL LISTINGS 10-5
CFT COMPILER DIRECTIVES 10-6
ACCESSING SKOLTXT 10-7
TRANSLATOR DIRECTIVES 10-7
TRANSLATING FORTRAN TO SKOL 10-12
INDENTATION 10-12
MACRO TRANSLATION ERROR MESSAGES 10-13

APPENDIX SECTION

A. THE SKOL MACRO TRANSLATOR A-1

INITIALIZER A-1
INPUT PROCESSOR A-3
MACRO TRANSLATOR A-4
OUTPUT PROCESSOR A-4
MACRO DEFINITION FORMAT A-5
SIMPLE TEXT SUBSTITUTION A-6
PARAMETERS IN MACROS A-8
NUMBER GENERATORS A-11
STACKS FOR NUMBERS AND SYMBOLS A-11
MACRO DEFINITIONS IN REPLACEMENT TEXT A-14
IMPORTANCE OF ORDER IN MACRO CREATION A-15
IMPLEMENTATION OF PASCAL SCALAR TYPES AND CASE STATEMENT A-15

B. THE CHAR DATA TYPE B-1

ORD, CHR B-3
VALUE B-3
READSTRING, WRITESTRING B-4
THE C FORMAT B-5

C. SKOL KEYWORDS C-1

D. STRUCTURED TEXT D-1

E. ERROR MESSAGES E-1

F. SAMPLE PROGRAM F-1

G. BIBLIOGRAPHY G-1

INDEX

TABLES

2-1 FORTRAN and SKOL relational operators 2-10
9-1 Carriage control specifiers 9-2
10-1 Translator directives 10-7
10-2 Nesting level where indentation ceases 10-13
A-1 Character correspondences A-2
A-2 Input line establishing the macro definition format A-3
A-3 Operators in replacement parts of macros A-24

INTRODUCTION 1

SKOL was chosen from among the many available FORTRAN preprocessors because of the completeness and excellent design of its control structures, resulting in superior readability for its programs. Other advantages are its Pascal-like records and enumerated types, and its inherent extensibility. Because it is a FORTRAN preprocessor, and because most valid FORTRAN statements are passed through unchanged by the macro translator, the advantages of the underlying language (efficiency, an extensive subroutine library, and hooks to the operating system) are still available to the SKOL user. In addition, a SKOL program can include segments of FORTRAN code, and a FORTRAN program can include segments of SKOL code.

OVERVIEW

SKOL offers the user a more structured approach to programming than is possible with FORTRAN and consequently a more readable program that is easier to maintain. The following major features of the language exemplify the structured programming philosophy inherent in SKOL.

CONTROL STRUCTURES

SKOL is rich in its variety of control structures. Although some of them are redundant, this redundancy serves a purpose. When the simplest applicable control structure is used wherever possible, it gives the reader early notice as to what level of complexity to expect.

SKOL control structures begin and end with matching keywords that define their scope. The macro translator makes sure they are nested properly and provides automatic indentation to simplify coding and modification. Many of these structures, which are summarized below and described in detail elsewhere, will be familiar to people acquainted with other high-level languages.

- The structures of the major program segments (MAIN...ENDMAIN, FUNCTION...ENDFUNCTION, SUBROUTINE...ENDSUBROUTINE, and BLOCKDATA...ENDBLOCKDATA) are inherited from FORTRAN.

- The ROUTINE...ENDROUTINE structure defines a parameterless routine that lies within a major segment and shares its data.

- A recursive version of ROUTINE...ENDROUTINE can have parameters and local variables. Like an ordinary routine, it belongs to a major segment.

- A program can have several COROUTINE...ENDCOROUTINE structures, which combine to form a PROCESS. Each coroutine can ACTIVATE other coroutines or SUSPEND the entire process.

- The use of IF...ELSEIF...ELSE...ENDIF is derived from LISP and ALGOL 68. The same structure has been incorporated into FORTRAN 77.

- DO...ENDDO is an important control structure when the zero iteration count is handled correctly, as it is in CRAY-1 FORTRAN (CFT).

- FOR...ENDFOR parallels the DO...ENDDO structure. While less efficient, it has somewhat greater power and flexibility.

- UNTIL...ENDUNTIL, called the situation case structure, is not a loop but a means for declaring the scope of one or more labels.

- REPEAT...ENDREPEAT is a simple looping structure that repeats either indefinitely or a given number of times.

- LOOP...WHILE...ENDLOOP is a general looping structure that derives from Pascal's "while-do" and "repeat-until" statements but also allows a third kind of loop that terminates in the middle.

- LINK...ENDLINK is a specialized looping structure primarily intended to traverse a linked list of records.

- WITH...ENDWITH declares a scope within which a specific pointer is to be used by default to access records.

- WHEN...ENDWHEN selects a block of statements to be executed according to the value of a variable that belongs to an enumerated type. This is similar to Pascal's case statement, but the SKOL version allows an ELSE clause.

DATA STRUCTURES

The following data structures and associated operations are included in SKOL:

- The WORD type defines an integer or an integer array in which each word contains one or more bit-addressable fields, with each field having its own name. Whether or not a variable is declared in a WORD statement, fields within it can be extracted or modified by the BYTE pseudo function.

- RECORD...ENDRECORD resembles the Pascal record but without variant records or nesting. SKOL does provide for arrays of records, arrays within records, partial-word fields within records, and arrays of subdivided words within records.

- TABLE...ENDTABLE structures are somewhat analogous to records, but they define space that is allocated dynamically in contrast to the static record structures. Tables are supported by a set of routines that are unique to the CRAY-1.

- The built-in POINTER type is a means of declaring an integer that points only to records of a given type or entries in a given table.

- TEMPLATE...ENDTEMPLATE is a data structure that allows the definition of named fields within itself, and can be associated with a BASE POINTER variable so that it overlays one or more selected areas of memory.

- The STRING type defines an integer array with an associated length variable that can be operated on by SIZE, LENGTH, DELETE, INSERT, MOVE, REPLACE, and CATENATE.

- The TYPE statement declares enumerated data types that can be referenced by the WHEN...ENDWHEN control structure and by the operators IN, FIRST, and LAST. TYPE statements can be nested.

- TYPE CHAR is a predefined enumerated type that, if declared, enhances the portability of a SKOL program while reducing its efficiency. The built-in operators ORD, CHR, VALUE, READSTRING, and WRITESTRING are associated with TYPE CHAR.

OTHER FEATURES

The following SKOL features aid the programming effort in other ways:

- Automatic indentation of the source text displays a program's structure and helps prevent errors.

- Conditional code generation allows the programmer to suppress specified sections of SKOL code.

- INPUT and OUTPUT statements allow the side-by-side specification of a variable and its edit descriptor, thus eliminating the need for FORMAT statements.

- Structured text is written using sentences with SKOL keywords that cause automatic indentation just as in executable SKOL code.

- A TRACE statement causes the printing of a variable's new value whenever an assignment is made to that variable.

- A VALIDATE statement enables or disables the printing of run-time debugging messages, such as reporting the incorrect use of a WHEN or UNTIL structure or tracing the program's flow of control.

EXTENSIBILITY

Extensibility may be SKOL's most important attribute. It permits the invention of new data structures or control structures to suit an immediate need.

The SKOL user will probably never need additional control structures. Data structures, however, are not understood as well as control structures and are still subject to experimentation. Macros defined by the user can render queues, stacks, linked lists, and trees more accessible. By easing the burden ordinarily associated with using such structures in languages such as FORTRAN, macros encourage the selection of the ideal data structure for a particular application.

DIAGNOSTIC MESSAGES

Diagnostic messages issued during compilation of a SKOL program have two sources: the SKOL macro translator and the FORTRAN compiler. Errors detected by SKOL must first be corrected before the translated program is compiled, unless a %E translator directive is present in the SKOL source code.

Messages issuing from the FORTRAN compiler may not always relate in an obvious way to the source input to SKOL, since they are in response to the FORTRAN code generated by the SKOL macro translator. As a result, a knowledge of FORTRAN and access to the CFT Reference Manual are helpful in debugging. Appendix D of the CFT manual contains a list of messages produced by the FORTRAN compiler.

To help the SKOL user avoid having to list the generated FORTRAN code to understand a compiler diagnostic message, SKOL appends the appropriate SKOL source line number to each line of FORTRAN it generates. If it discovers an error in pass 1, CFT lists the erroneous FORTRAN line even when the listing is turned off on the CFT control statement (see section 10). If all variables are declared, CFT can detect most errors in pass 1.

Errors made in the declarations section of a major segment are usually not discovered by CFT until the first executable statement is encountered. Using the optional BEGIN statement to separate the declarations from the executable code causes a CONTINUE statement to be inserted in the generated FORTRAN code as the first executable statement. CFT then associates any errors in the declaration section with the CONTINUE statement. Thus, any ambiguity concerning the source of the error is removed.

FORMAL SYNTAX REPRESENTATION

The following conventions are used in this publication to represent formal syntax.

- Optional items are enclosed in braces.
- Lowercase italic type, as in *block*, represents variable code.
- Uppercase standard type, as in MAIN, stands for a keyword.
- Items stacked vertically, preceded and/or followed by braces, represent a choice of one of the items. For instance, the following means one of the a_i may be chosen:

$$\left\{\begin{matrix} a_1 \\ a_2 \\ \cdot \\ \cdot \\ \cdot \\ a_n \end{matrix}\right\}$$

SYNTAX 2

Unless otherwise indicated, the syntax for SKOL is identical to that of FORTRAN. For example, the basic assignment statement is of the form:

variable = expression

However, a number of differences in syntax and notation do exist between the two languages.

CHARACTER SET

The FORTRAN character set (see the CFT Reference Manual) is contained within the SKOL character set. In addition, the following characters are used in a SKOL source program:

Symbol	Character
$	Dollar sign
_	Underscore
|	Vertical bar
[	Left bracket
]	Right bracket
;	Semicolon
&	Ampersand
^	Up arrow
<	Less than
>	Greater than

Characters needed in defining macros, but not in ordinary SKOL source statements, are:

Symbol	Character
%	Percent sign
@	At sign
#	Pound sign

Characters used only to display the macro expansion process (when the %T translator directive is in effect) are:

Symbol	Character
{	Left brace
}	Right brace

LITERAL STRINGS

Any character available to the programmer is permitted within a literal string. Such strings specify constants of various types as well as the patterns and replacement parts of macro definitions. SKOL, in contrast to FORTRAN, requires literal strings to begin and end with apostrophes; no other form is valid. An apostrophe within a literal string is indicated by two apostrophes.

A literal string, with certain exceptions, may have an H, L, or R suffix, as in FORTRAN. The meanings of these suffixes are as follows:

Suffix	Internal representation of string
H	Characters are left-justified with blank fill
L	Characters are left-justified with zero fill
R	Characters are right-justified with zero fill

No suffix is permitted if a string represents any of the following:

- The pattern or the replacement part of a macro definition
- An element in a type CHAR declaration
- Any single-character constant when the CHAR type is in effect
- Any multicharacter constant in a CATENATE or string assignment statement

If the CHAR type is not declared, SKOL provides an R suffix for all single-character literals without suffixes. All other literal strings without suffixes are given an implied H suffix by FORTRAN. If character strings with various lengths and no suffixes are mixed, remember that single-character literals are right-justified and longer strings are left-justified.

IDENTIFIERS USED IN SKOL

In this publication, a distinction is made between SKOL identifiers and FORTRAN identifiers. Identifiers are called symbolic names in FORTRAN.

FORTRAN identifiers are passed to the FORTRAN compiler unchanged. They must be no longer than eight alphanumeric characters on the CRAY-1. The first character must be a letter. Lowercase and uppercase letters can be used interchangeably.

SKOL identifiers are replaced by other forms during the preprocessing stage. They are limited in length only by the length of the input record (usually 72 characters) and can also contain dollar signs and underscore characters. The length limitation is necessary because the line break character is replaced by a blank, which is not a valid character in a SKOL identifier. While FORTRAN ignores embedded blanks, SKOL requires both SKOL and FORTRAN identifiers to be free of embedded blanks.

When creating a SKOL identifier, it is a good idea to choose an identifier that is not a valid FORTRAN identifier (for instance, by using an underscore character). Doing so ensures that an identifier that the programmer assumes will be changed into a different form, is in fact changed into a different form.

Although a SKOL identifier currently can begin with a digit, such identifiers cause errors in the SKOL cross-reference listing and should be avoided. Identifiers beginning with I00 are not valid in source programs, because SKOL uses this form to generate its own internal identifiers.

Wherever a FORTRAN identifier is required, a SKOL identifier can be substituted through the prior use of a DEFINE statement. Substitution by means of the DEFINE statement is described in section 6 of this publication.

STATEMENT LENGTH

SKOL source text is basically free form, with no significance assigned to the end of a line (except as noted above) or to specific columns. Statements are delimited by colons and semicolons.

Comments and translator directives are exceptions to the free form rule. Normally, program text must lie between columns 1 and 72, although the %C translator directive allows the rightmost column number to be changed from 72 to any value between 2 and 80. Translator directives are described in section 10 of this publication.

A literal string may be continued on successive lines, although it would normally be broken up for readability. Continuing a literal string without closing it and re-opening it results in the insertion of a blank character (representing a line break) between column 72 of one line and column 1 of the next. The maximum number of lines that can be occupied by a single literal string is 25.

STATEMENT TERMINATORS

Every SKOL statement must have its proper terminator (a colon for statements that introduce structures, a semicolon for all other SKOL statements, and a period for structured text statements, described in Appendix D.) The success of the pattern matching process, and SKOL's ability to provide good diagnostic messages, depends on the presence of these terminators.

COMMENTS

A comment can be introduced in three ways: with two hyphens, with a quotation mark, or with the three characters <<:. No comment form can be used inside a literal string (a string enclosed by apostrophes).

The first method is convenient for short comments of one or two lines and also can be used in constructing boxes. Blanks are not permitted between the two hyphens. All characters on the same line and to the right of the hyphens are ignored, as are the hyphens themselves.

Using the second method, a comment string must begin and end with quotation marks. This form is used to delimit comments that are in paragraph form and for the special case of commenting within a line of code.

The third method is used to include very high-level code (structured text) as comments in a SKOL program. Structured text comments are indented as in SKOL code, using the same structural keywords. Except for these keywords, structured text statements can be in any language (see Appendix D). Comments introduced by <<: must be terminated by the two characters >>.

VARIABLE DECLARATIONS

SKOL provides the data types INTEGER, SMALL INTEGER, LOGICAL, DOUBLE, REAL, COMPLEX, CHARACTER, WORD, STRING, POINTER, and BASE POINTER. The user can also define enumerated data types with the TYPE statement, described in section 4 of this publication. The first seven data types listed (INTEGER through CHARACTER) correspond exactly to FORTRAN data types of the same names except for SMALL INTEGER, which corresponds to INTEGER*2. BASE POINTER corresponds to CFT's POINTER type.

All variables must be declared at the top of the major program segment in which they are used. The form of such a declaration is:

$$\boxed{type\text{: } fid_1 \ \{,fid_2 \ \ldots \ ,fid_n\};}$$

type A data type

fid_i A FORTRAN identifier, optionally followed by a dimension specification

A variable declared at the beginning of a program segment is available to that segment and to any routines or coroutines defined in that segment. An example of a type declaration section follows:

Example:

```
INTEGER: I, J, B, RATING(-5..5), DV;
REAL: X, Y, Z, MAT(100, 0..9);
LOGICAL: FOUND;
CHARACTER*7: NAMES(50);
```

As the above example illustrates, both the lower and upper bounds of an array can be specified. This optional form for declaring array dimensions in SKOL is permissible only if the FORTRAN compiler allows the corresponding form:

```
RATING (-5:5)
```

EQUIVALENCE STATEMENT

SKOL provides its own form for the FORTRAN equivalence statement. The form of a SKOL equivalence statement is:

$$\text{EQUIVALENCE: } fid_1 = fid_2 \{, fid_3 = fid_4 \ \ldots \ , fid_{2n-1} = fid_{2n}\};$$

fid_i The FORTRAN identifier of a variable or array whose type (and dimension, if an array) is declared elsewhere

A FORTRAN equivalence statement is generated for each pair of *fid*s:

EQUIVALENCE (fid_1,fid_2)

BASED VARIABLES

A variable of the type BASE POINTER furnishes a base address for a corresponding variable or array. The form of the declaration is:

$$\text{BASE POINTER FOR } fid_1 : fid_2;$$

fid_1 The FORTRAN identifier of a variable or array whose type (and dimension, if an array) is declared elsewhere

fid_2 The FORTRAN identifier of the variable which is to contain the base address of fid_1 and thus determine its location

A base pointer variable has the implied type SMALL INTEGER. It must be a simple variable. It can appear in a common list, an EQUIVALENCE statement, or as a dummy argument in a subroutine.

A base pointer variable can be set with a LOC function reference (see the CFT Reference manual), or it can be set as an absolute address, as in the following example.

Example:

```
        INTEGER: JCB(128);              -- JCB is an array (the
        BASE POINTER FOR JCB: PJCB;     -- Job Control Block) occupying
        DATA: PJCB/0/;                  -- the first 128 words of memory.

        COMMON POOL(100000);
        REAL: A(1), B(1), C(1);
        BASE POINTER FOR A: IA;
        BASE POINTER FOR B: IB;
        BASE POINTER FOR C: IC;
        EXTERNAL INTEGER: LOC;
      BEGIN:
        IA = LOC(POOL);                 -- IA is at beginning of blank common.
        IB = IA + 1000;                 -- IB begins 1000 words later.
        IC = IB + N;                    -- IC begins another N words later.
```

In effect, the three real arrays are equivalenced to POOL and possibly to each other, depending on the subscript usage. However, CFT makes no checks for possible equivalence overlays that cause vector dependencies as it does when the EQUIVALENCE statement is used.

Any change in the value of a base pointer variable causes all subsequent references to the based variable to refer to the new location. Thus the based variable can be imagined as an empty structure that can be moved about and made to overlay any area in memory. See section 5 for a description of the TEMPLATE structure, which allows the programmer to overlay memory with partial-word definitions.

EXTERNAL DECLARATIONS

Any subroutine that is called, and any non-intrinsic function that is referenced, must be declared at the beginning of the major program segment that invokes it. For an external function, the declaration must include the function's type:

EXTERNAL *type*: fid_1 {,fid_2 ... ,fid_n};

fid_i A FORTRAN identifier naming an external function

type The type of the variable returned by the function or functions. It can be either a FORTRAN data type or a SKOL scalar type, but not a subtype or any other SKOL type such as POINTER or SMALL INTEGER.

For an external subroutine, the type is omitted:

```
EXTERNAL: fid1 {,fid2 ... ,fidn};
```

fid$_i$ A FORTRAN identifier naming an external subroutine

All functions that return Boolean values are intrinsic and need not be declared. Examples are GETPOS, SHIFT, AND, OR, MASK, and CSMG. (Intrinsic functions are documented in the CFT Reference Manual, Appendixes B and C.)

Declaring every variable and external name makes it less likely that the generated FORTRAN code will need examining in order to understand the error messages issued by CFT. An undeclared variable, used inconsistently (for instance, first as a logical variable and then as an integer), produces a CFT diagnostic message only at the end of the major segment. A backward reference to a line number within the generated FORTRAN is produced, but the customary line of source code with its appended SKOL line number is not forced out. When all variables are declared, such difficult error messages become very rare and a full listing of the generated FORTRAN is rarely needed to understand an error message.

IMPLICIT VARIABLE DECLARATION

By redefining the built-in constant $DEFTYPE, the user can override SKOL's requirement that all variables be declared. The standard (default) value for $DEFTYPE is NONE, which causes the statement IMPLICIT SKOL to be generated as the first statement of every major segment. IMPLICIT SKOL, which is unique to CFT, is exactly like CFT's IMPLICIT NONE except that variable names beginning with I00 are implicitly assigned the type INTEGER and so escape the requirement that they be declared.

The following redefinitions of $DEFTYPE are valid. In each case, the corresponding IMPLICIT statement is generated at the beginning of every major segment introduced after the redefinition.

SKOL Redefinition	Generated FORTRAN
CONSTANT: $DEFTYPE = NONE;	IMPLICIT SKOL
CONSTANT: $DEFTYPE = INTEGER;	IMPLICIT INTEGER(A-Z)
CONSTANT: $DEFTYPE = LOGICAL;	IMPLICIT LOGICAL(A-H,J-Z)
CONSTANT: $DEFTYPE = *ftype*;	IMPLICIT *ftype*(A-H,O-Z)

ftype A FORTRAN data type, such as REAL, DOUBLE, COMPLEX, or CHARACTER

DATA STATEMENT

The DATA statement is like that of FORTRAN, with a colon and semicolon added. Everything in the body of a DATA statement is passed on to FORTRAN without change, except phrases of the form (FOR ...). Such a phrase, which is called a FOR clause, provides the user with an iterative form for assigning data, as the example below shows. The FOR clause is described more completely in section 9.

The DATA statement takes the following form:

DATA: var_1/*value*/ {,var_2/*value*/ ... ,var_n/*value*/};

var_i A variable or an array name (or a series of names separated by commas)

value An initial value (or a series of values separated by commas) assigned to the variable(s)

Example:

```
COMMON/LAB002/BLANK, T(5), S, N, K;
.
.
.
BLOCKDATA:
  DATA: (FOR I=3 TO 5: T(I))/ 33, 96, 101/;
ENDBLOCKDATA;
```

Note that COMMON statements without FOR clauses are exactly as in FORTRAN, with the addition of the final semicolon.

RELATIONAL OPERATORS

Relational operators are used to compare two arithmetic expressions or two character expressions to each other. Relational operators are coded differently in SKOL than in FORTRAN. The FORTRAN operators and their SKOL equivalents are shown in table 2-1:

Table 2-1. FORTRAN and SKOL relational operators

FORTRAN	SKOL
.EQ.	=
.NE.	<>
.LT.	<
.GT.	>
.LE.	<=
.GE.	>=

LOGICAL OPERATORS

SKOL has five logical operators (NOT, AND, OR, EQV, and XOR) that are inherited from FORTRAN and one operator (IN) that is taken from the Pascal language. The delimiting periods required by FORTRAN are omitted in SKOL.

IN, AND, OR, EQV, and XOR cannot be used as functions or in any way other than as logical operators.

The IN operator is different from the other logical operators in that it requires its right-hand operand to be either an integer range or the name of a user-defined scalar type or subtype. Its left-hand operand must be of a matching type. See section 4 for a complete description.

The FORTRAN logical operators produce a result of type LOGICAL when used with operands and expressions of type LOGICAL. When used with operands and expressions of mixed or arbitrary type, they produce a Boolean result. The only restriction is that no operand can occupy more than one word in memory. A Boolean datum is considered to be a string of 64 bits.

The order of precedence among the logical operators is, from the highest to the lowest: NOT, IN, AND, OR, XOR, EQV. Except for the inclusion of IN, this is the same precedence order used in FORTRAN.

In an expression composed entirely of logical operators, parentheses are unnecessary, although their use aids readability. However, parentheses are often a necessity when arithmetic operators or commas are present alongside logical operators. In these cases, the parentheses automatically supplied by SKOL would be inserted in the wrong places.

ODD AND EVEN

The keywords ODD and EVEN, when used as function names, are translated by SKOL into logical expressions by the following macro definitions:

```
%' ODD(#)' = '(( #1 AND 1) <> 0)'
%' EVEN(#)' = '(( #1 AND 1) = 0)'
%'$NOT.(ODD(#))' = '( EVEN( #1 ))'
%'$NOT.(EVEN(#))' = '( ODD( #1 ))'
```

The SKOL programmer can treat ODD and EVEN as if they were intrinsic FORTRAN functions, returning logical values. Their names should not appear in a declaration such as EXTERNAL LOGICAL: ODD because the generated FORTRAN code contains no function call.

The argument (parameter 1 in the macro definitions) can be of any type. In CFT, logical operators such as AND treat their operands as Boolean expressions and produce a Boolean result. This result, if assigned to a variable of type LOGICAL or if used as a logical expression, is interpreted as TRUE if the sign bit is set and FALSE if it is not set. (See the sections on logical expressions and Boolean expressions in the CFT manual.)

Notice the presence of blanks around #1 in each of the replacement parts of the four macros. They are present in case #1 happens to contain a character string for which a macro pattern has been defined, such as the name of a constant or of another built-in SKOL function.

BOOLEAN

SKOL provides a built-in function named BOOLEAN to convert an expression of any arbitrary type to the type Boolean. Thus, any 64-bit data can be treated as a bit string and assigned to a variable or array element of any type without type conversion taking place. The BOOLEAN function is implemented by the following macro definition:

```
%' BOOLEAN(#)' = '" OR(" #1 ,0)'
```

The SKOL programmer can treat BOOLEAN exactly like a standard FORTRAN function, except that it should never be declared.

The quotation marks in the replacement part of the BOOLEAN macro prevent the error message, 'OR' USED AS FUNCTION. The spaces around #1 allow the actual parameter to be matched again if necessary. The space before the OR makes the generated FORTRAN more readable. For more about macro definitions, see Appendix A.

PROGRAM CONTROL STRUCTURES 3

SKOL offers a set of program control structures that enable the user to solve problems in a structured and logical manner.

Matching beginning and ending statements such as DO: ENDDO; and IF: ENDIF; reduce coding errors and, with the help of SKOL's automatic indentation, increase readability.

This section describes all of the nestable control structures provided by SKOL except for the WHEN (scalar case) structure, which is described in section 4, and the record-oriented WITH and LINK structures, which are described in section 5.

IF ... ELSEIF ... ELSE ... ENDIF

This basic control structure begins with an IF statement and ends with an ENDIF statement. The ELSE and ELSEIF statements are optional.

```
IF lexp1:
  block
ELSEIF lexp2:
  block
  :
ELSEIF lexpn:
  block
ELSE:
  block
ENDIF;
```

lexp$_i$	A logical expression
block	A sequence of statements

Each logical expression is tested in sequence until a true one is found; then the corresponding block is executed. If none of the logical expressions is true, the block following the ELSE is executed. When the ELSE statement is not present, the program acts as if an empty block (containing no statements) had been specified following an ELSE.

The ELSEIF statements and the corresponding blocks can be omitted, resulting in the simpler structure:

```
IF lexp:
  block
ELSE:
  block
ENDIF;
```

lexp A logical expression

block A sequence of statements

The ELSE statement and the corresponding block can also be omitted.

Example:

```
IF LINE(I) = CH:
  OUTPUT (CH::C, ' AT ', I:I3);
ELSEIF LINE(I) IN DIGIT:
  DV = VALUE(LINE(I));
  OUTPUT (LINE(I):C, :2X, DV::I2);
ELSE:
  OUTPUT ('UNEXPECTED SYMBOL');
ENDIF;
```

IF FIRST TIME

Two special forms of the IF statement cause execution of a block of code only on the first pass, or on the first few passes. With these special forms, the phrases FIRST TIME and FIRST *n* TIMES are treated as if they represented logical variables. They allow the execution of a subroutine's initialization code, for instance, just once or exactly *n* times. The structure has the general form:

```
IF FIRST {TIME:    }
         {n TIMES:}
  block
ENDIF;
```

n An integer expression

block A sequence of statements

ELSE and ELSEIF statements may be added to the structure, but the introductory IF statement cannot be extended by adding logical operators such as AND, OR, or NOT.

UNTIL ... THEN ... ENDUNTIL

The situation case structure was proposed by Charles Zahn[4] as a substitute for the unrestricted GO TO statement. It takes one of two forms. The following form is the more general one:

```
UNTIL sid1 OR sid2 {OR sid3 ... OR sidn}:
  block
THEN:
  [label1]: block
  [label2]: block
  .
  .
  .
  [labeln]: block
ENDUNTIL;
```

sid_i A SKOL identifier naming a situation

block A sequence of statements

$label_i$ A situation label matching one of the situation identifiers in the UNTIL statement or a series of such labels separated by commas. Each sid_i must occur exactly once as a label.

The first block is executed just once unless it contains a loop structure. Control is transferred out of the UNTIL block when one of the situations named in the UNTIL statement arises. A situation is said to arise when one of the following occurs:

- A statement that consists only of one of the sid_i is executed.
- A subroutine has been called that executes a RETURN *iexp* statement signalling a corresponding situation (named by an argument of the form *=sid_i) in the CALL statement that invoked it (see section 8).
- A READ statement returns with an end-of-file or an error signalling a situation named by END=sid_i or by ERR=sid_i (see section 9).

Control passes from the statement signalling the situation to the corresponding block following the THEN statement. If no situation arises before the end of the UNTIL block is reached, the program is in error. It will abort with the message, "NO SITUATION WAS SIGNALLED AT LINE *nn*" (indicating the line containing the THEN statement). If VALIDATE(-UNTIL) is in effect (see section 6), this check is suppressed and the program will not abort.

Any of the blocks in the THEN clause can be null, indicating that no action is required for a certain situation except the act of leaving the UNTIL block. If any block in a SKOL program is null, the terminating semicolon can be omitted.

Each situation label in the THEN clause must be enclosed in brackets and followed by a colon.

Example:

```
UNTIL MATCH OR NO_MATCH:
  DO I = 1 to N:
    IF X = TABLE(I):
        MATCH;
    ENDIF;
  ENDDO;
  NO_MATCH;
THEN:
  [MATCH]: COUNT(I) = COUNT(I)+1;
  [NO_MATCH]: N = N+1; TABLE(N) = X;
    COUNT(N) = 1;
ENDUNTIL;
```

UNTIL ... ENDUNTIL

If only one situation is named in the UNTIL statement, the THEN statement and the block following it are not needed and must be omitted from the situation case structure:

```
UNTIL sid:
  block
ENDUNTIL;
```

sid A SKOL identifier naming the situation

block A sequence of statements

The block is executed just once unless it contains a loop structure. When the named situation arises, the UNTIL block is terminated. As with the previously described situation case structure, if the end of the block is reached before the situation arises, the program is in error and will abort unless the check is suppressed by VALIDATE(-UNTIL).

Example:

```
UNTIL NON_BLANK:
  DO I = 1 TO 80:
    IF CH(I) <> ' 'R: NON_BLANK; ENDIF;
  ENDDO;
  NON_BLANK;
ENDUNTIL;
```

This program fragment delivers the index of the first nonblank character found in array CH if there is one. If not, the value of I at the end of the UNTIL structure is 81.

Note the use of apostrophes to surround the single-character literal in the example. In SKOL, literal strings must always be delimited by apostrophes. Also note the presence of an R suffix with the usual FORTRAN meaning to denote the format of the literal character. The R is optional; it is assumed by default for single-character literals unless the CHAR type is declared (see appendix B for a description of the CHAR type).

These two situation case structures are equivalent to a restricted form of GO TO statement. The restriction is that the GO TO, which in effect replaces each sid_i in the UNTIL block, must jump to the statement following the ENDUNTIL (either directly or indirectly through a THEN block). In practice, the UNTIL ... ENDUNTIL structure may encompass an entire MAIN segment, function, or subroutine. A particular sid_i, however, needn't necessarily be located within the structure. It may be encountered instead in a routine or coroutine invoked from within the UNTIL block.

The UNTIL statement does not imply looping. If looping is required, a DO, FOR, LOOP, REPEAT, or LINK structure must be part of the UNTIL block.

Do not confuse situation names with logical variables. Situation names exist only in the SKOL code; they are replaced by statement labels in the generated FORTRAN.

NOTE

Using the same SKOL identifiers as situations in two situation case structures in the same MAIN segment, function, or subroutine can produce unwanted results.

REPEAT ... ENDREPEAT

REPEAT is the most basic repetitive statement in SKOL. The REPEAT structure takes the form:

```
REPEAT {iexp TIMES}:
  block
ENDREPEAT;
```

iexp An integer expression with a non-negative value

block A sequence of statements

When the optional TIMES phrase is absent, the repetition is infinite. Take care to ensure that the infinite REPEAT block contains a situation statement that terminates an enclosing UNTIL structure. The finite version is shown in the following example:

Example:

```
REPEAT 5 TIMES:
  X = 2X + I;
ENDREPEAT;
```

LOOP ... WHILE ... ENDLOOP

A useful repetitive statement is a loop structure proposed by Ole-Johan Dahl[5] that replaces the "while-do" and "repeat-until" statements of Pascal. Its form is:

```
LOOP:
  block
WHILE lexp:
  block
ENDLOOP;
```

block A sequence of statements

lexp A logical expression

The first of the two blocks is executed, and the logical expression is tested. If the logical expression is false, the loop is exited; if it is true, the second block is executed. Control then returns to the first block, and the sequence is repeated.

Example:

```
SUM = 0.0; COUNT = 0;
LOOP:
  INPUT (I:I5, X:F10.5);
WHILE I>0:
  SUM=SUM+X; COUNT = COUNT+I;
ENDLOOP;
OUTPUT ($SKIP2, COUNT, SUM);
```

Either the first or the second block may be empty. To aid readability when the second block is empty, the programmer may insert a CONTINUE statement (which the FORTRAN compiler ignores). When the first block is empty, readability can be improved by placing the LOOP and WHILE statements on the same line.

Examples:

```
LOOP:
  B = FUNC(A,B);
WHILE B>EPSILON:
  CONTINUE;
ENDLOOP;

LOOP: WHILE X>2,0:
  X = SQRT(X);
ENDLOOP;
```

Dahl's loop structure covers most simple iterations without requiring the "rather big" situation case structure (to paraphrase Knuth[6]). When a programmer uses the simpler structure, he is making it clear that he has a simple iteration, with exactly one condition that is tested exactly once each time around the loop.

DO ... ENDDO

The DO structure in SKOL is essentially identical to the DO-loop in FORTRAN. The CRAY-1 FORTRAN compiler provides such a powerful DO-loop that the similar FOR structure provided by SKOL is usually unnecessary. The DO structure has two versions, one finite and one infinite. The finite version takes the form:

```
DO simple ivar = iexp TO iexp {BY iexp}:
  block
ENDDO;
```

simple ivar	A non-subscripted integer variable
iexp	Any integer expression accepted in a DO-loop by the destination compiler
block	A sequence of statements

When the BY phrase is omitted, the default increment is 1. The increment expression can be a negative value. The BY phrase optionally may appear before the TO phrase instead of after it.

No upper bound is present in the infinite version of the DO structure. To maintain readability, the increment phrase (BY) must be specified. The infinite DO takes the following form:

```
DO simple ivar = iexp BY iexp:
  block
ENDDO;
```

simple ivar	A non-subscripted integer variable
iexp	Any integer expression accepted in a DO-loop by the destination compiler
block	A sequence of statements

The infinite form of the DO structure requires the same warning concerning eventual termination as the infinite REPEAT.

The finite form may not execute the block at all if the iteration phrase specifies an empty progression, such as

```
DO I = 1 BY 1 TO 0:
```

I remains unchanged in that case. If the phrase reads instead,

```
DO I = 1 BY 1 to 10:
```

the value of I following normal termination is determined by the host FORTRAN compiler. With the CFT compiler, I retains its final value, which is one increment past the upper bound (11 in this case).

Examples of the finite and infinite forms respectively follow:

<u>Example 1:</u>

```
DO I = LENGTH(WORD) TO 1 BY -1:
  IF WORD(I) = EOL:
    WORD(I) = '?'R;
    J = 1;
  ENDIF;
ENDDO;
```

<u>Example 2:</u>

```
DO I = 0 BY -(INC+1):
  :
  :
ENDDO;
```

FOR ... ENDFOR

The FOR statement is similar to the DO statement, but the more efficient DO statement should be used whenever possible.

As with the DO statement, the FOR statement has finite and infinite versions. The forms of the two versions, with the finite form first, follow:

```
FOR ivar = iexp {BY iexp} TO iexp:
  block
 ENDFOR;
```

```
FOR ivar = iexp BY iexp:
  block
ENDFOR;
```

ivar	A simple or subscripted integer variable
iexp	An integer expression
block	A sequence of statements

NOTE

When using the finite version of the FOR structure, the integer variable must eventually match the integer expression in the TO phrase exactly or the structure continues to loop. If VALIDATE(FOR) is enabled (see section 6), SKOL checks at the entrance to the loop to make sure the increment is nonzero and the initial value will eventually reach the final value.

When the BY phrase is omitted, the default increment is 1.

When using the infinite version of the FOR structure, a situation statement must be inserted inside it to ensure proper termination.

The integer variable retains its final value when the FOR loop is exited normally.

Example:

```
FOR P(J) = 1 TO NDIM:
  :
ENDFOR;
```

INCR, DECR

SKOL provides increment and decrement statements for general use and for use with loops that have no built-in means of updating a control variable. These statements are of the form:

$$\left\{\begin{matrix}\text{INCR}\\ \text{DECR}\end{matrix}\right\} \textit{var} \ \{\text{BY } \textit{exp}\};$$

var Any variable

exp An expression

Omitting the BY phrase defaults the increment or decrement value to 1.

USER-DEFINED TYPES AND THE SCALAR CASE STRUCTURE

4

The WHEN ... ENDWHEN control structure (also called the scalar case structure) is treated apart from the other program control structures, because it is necessary to first introduce the concept of the user-defined scalar type. Only user-defined scalar types or subtypes can be specified following the selector expression of a WHEN statement; standard types, such as INTEGER, LOGICAL, and STRING, cannot.

In addition to their role in the scalar case structure, scalar types (also called enumerated types[7]) allow names to be used instead of numbers. They may be considered a kind of structured data. Each element of a scalar type may be a subtype with elements of its own, or it may be simply a constant.

The character data type, CHAR, is built into SKOL, but it must be declared explicitly by the programmer if it is to be used. Declaring CHAR permits the use of single-character constants in scalar case structures, but CHAR requires special handling and reduces program efficiency. For these reasons, CHAR is not considered an integral part of the language and is described in appendix B.

TYPE DECLARATION

The TYPE statement names a new scalar data type and enumerates its elements. It takes the following form:

```
TYPE sid = (member1, member2 {... ,membern});
```

sid A SKOL identifier

$member_i$ Either a SKOL identifier naming an element or a subtype defined with the following syntax:

sid = (*$member_1$*, *$member_2$* {... ,*$member_n$*})

This syntax definition is recursive; the number of subtype levels is unlimited.

The name of the new scalar data type may be used to refer to all of its members, and the name of each subtype refers to all of the members within it. An empty list should not appear on the right side of an equal sign except in the special case of the CHAR scalar type (see appendix B).

The scope of any TYPE declaration depends on whether the declaration lies within or outside of a major program segment. A "local" type, declared within a major segment, cannot be referred to outside that segment. A "global" type, declared before any segment or between two major segments, can be referred to by any statement that physically follows it in the program text.

An example of a simple TYPE declaration, without subtypes, is as follows:

Example:

```
TYPE DAYS_OF_WEEK = (SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
                     THURSDAY, FRIDAY, SATURDAY);
```

The constants belonging to the type DAYS_OF_WEEK are the seven SKOL identifiers listed. They are assigned internal values in the order listed; SUNDAY is 1, MONDAY is 2, and so on. Thus, the following logical expression is true:

```
FRIDAY>MONDAY
```

A hierarchy of subtypes can be created, as in the following example:

Example:

```
TYPE AUTO = (
     GEN_MOTORS = (CHEVY, PONTIAC, CADDIE),
     FORD = (MUSTANG, MERCURY = (MONARCH, COUGAR)),
     VW = (BEETLE, RABBIT, DASHER) );
```

The next example gives the same TYPE declaration in a more graphic representation. (This kind of indentation is not provided by SKOL.)

Example:

```
TYPE AUTO =
  (GEN_MOTORS =
    (CHEVY
    ,PONTIAC
    ,CADDIE
    )
  ,FORD =
    (MUSTANG
    ,MERCURY =
      (MONARCH
      ,COUGAR
      )
    )
  ,VW =
    (BEETLE
    ,RABBIT
    ,DASHER
    )
  );
```

In this case, the values of type AUTO are the values of the subtypes GEN_MOTORS, FORD, and VW: that is, all of the individual constants. The values of FORD are MUSTANG and MERCURY (which is to say the constants MUSTANG, MONARCH, and COUGAR). MERCURY, in turn, consists of the constants MONARCH and COUGAR.

Variables may be declared to belong to scalar types (but not to their subtypes) in a manner analogous to ordinary FORTRAN declarations.

scalar: var_1 {,var_2 ... ,var_n};

scalar A previously declared TYPE identifier

var_i A FORTRAN identifier naming a variable or an array of any dimensionality

Example:

```
AUTO: FAMILY(2), MINE, CUBIC(-1..3,10,0..9);
```

NUMERIC TYPES

A supplementary form of the TYPE statement is provided for use with the scalar case statement, which is described next. The NUMERIC type is convenient when suitable names for the type's components do not exist. The form is:

```
TYPE sid = NUMERIC (number1,{number2  ... ,numbern});
```

sid	A SKOL identifier
$number_1$ ... $number_n$	All the numbers, listed in their natural order and without missing values, required for the scalar case statement

The first number may be any positive or negative integer, including 0.

NOTE

Numbers are treated as character strings by SKOL; only FORTRAN sees them as integers. Therefore, they must be copied verbatim when used in a scalar case statement. Be careful that the numbers are in the form needed. In this context, for instance, 01 is not the same as 1.

Numeric types cannot have subtypes, and a subtype cannot be numeric. The scope rules are the same as for ordinary scalar types.

SCALAR CASE STRUCTURE (WHEN ... ENDWHEN)

The WHEN structure in SKOL corresponds to the case statement of Pascal.[8] It begins with a WHEN statement that specifies a selector expression and a scalar type. The selector expression need not be of the given type, but it must resolve to an integer expression that agrees in value with that type. The WHEN statement selects one of the labeled blocks for execution according to the current value of the selector expression. Upon completion of the selected block, control goes to the statement following the ENDWHEN. The form is:

```
WHEN selector-exp:scalar IS:
    [label1]: block
    [label2]: block
    :
    [labeln]: block
    [ELSE]: block
ENDWHEN;
```

selector-exp	An integer expression
scalar	A type identifier declared in a previous TYPE statement that determines the range of permissible values for the selector expression
$label_i$	A constant or subtype of the scalar type or a series of such labels separated by commas
block	A sequence of statements

Unless an ELSE label is present, each constant belonging to the indicated scalar type must occur as a label exactly once. ELSE, which must be the last label if present, takes care of all constants not explicitly used as labels. If the selector expression is not within the range of values that belong to the scalar type, an error is diagnosed at run time if VALIDATE(WHEN) is enabled. (The VALIDATE statement is described in section 6.) Except for ELSE, the order of the labels is arbitrary, but labels must be unique within a given WHEN structure. Labels are placed inside of brackets; two or more labels may be collected inside a single pair of brackets and separated by commas.

Scalar case structures may be nested; in other words, one or more of the WHEN ... ENDWHEN structures may be completely contained within another.

Example:

```
WHEN FAMILY(K):FORD IS:
  [COUGAR, MUSTANG]: J = J+1;
  [ELSE]: J = J-1;
ENDWHEN;
WHEN MINE:AUTO IS:
  [MERCURY, RABBIT]:
    M = M+1;
  [BEETLE]:
    N = N+1;
  [GEN_MOTORS]:
    WHEN MINE:GEN_MOTORS IS:
      [PONTIAC, CADDIE]:
        L = L+1;
      [ELSE]:                          -- CHEVY
        I = I+1;
    ENDWHEN;
  [ELSE]:                              -- MUSTANG and DASHER
    I = I+1;
ENDWHEN;
```

If the value of FAMILY(K) in the first WHEN statement does not match any of the constants in the subtype FORD, then control falls through to the statement after the first ENDWHEN. If, however, VALIDATE(WHEN) is enabled, the program aborts with an error message.

IN

A special kind of logical expression is available to determine whether a variable is contained in a scalar type or subtype or in an integer range. Its form is:

$$\boxed{exp \left\{\text{NOT}\right\} \text{ IN } \left\{\begin{matrix} scalar \\ [exp_1..exp_2] \end{matrix}\right\}}$$

scalar	A scalar type or subtype
exp, exp_1, exp_2	Integer expressions such that $exp_1 < exp_2$

If the integer expression preceding IN is a member of the type or subtype, or if it falls within the range bounded by the other two expressions, the logical expression is true (false for NOT IN). As with other logical expressions, the IN expression must be enclosed in parentheses when it is combined with other expressions in a statement.

Examples:

```
NUM = CH(K) IN ['0'R..'9'R];  -- NUM is a logical variable.
IF MINE IN FORD: ...
ELSEIF LTH NOT IN [1..MAX_LENGTH]: ...
```

NOTE

Never use IN as the name of a variable or function. It will be replaced by macros associated with the logical operator IN, and unwanted results will follow.

FIRST, LAST

Two other functions are available for use with scalars. FIRST and LAST return the first element and the last element, respectively, of a scalar type or subtype. Their forms are as follows:

$$\left\{\begin{matrix}\text{FIRST}\\ \text{LAST}\end{matrix}\right\}(sid)$$

sid A SKOL identifier naming any scalar type or subtype

Examples:

```
FIRST(DAYS_OF_WEEK) is SUNDAY
LAST(DAYS_OF_WEEK) is SATURDAY
```

NOTE

Never use FIRST or LAST as the name of an array or function. Unwanted results will occur.

COMPLEX DATA STRUCTURES 5

Partial-word variables, templates, tables, and records are complex data structures allowing related data of different types to be stored together. Partial-word variables allow the packing of more than one variable into a machine word. Templates may contain full-word and partial-word variables, but are otherwise similar to FORTRAN arrays. Records are static data structures used to store related data of different types. The record concept is derived in large part from Pascal.[9] Tables are dynamically allocated data structures supported by a set of table management subroutines.

PARTIAL-WORD VARIABLES

WORD declarations and the BYTE function provides two allied methods for packing more than one variable into a machine word. Both methods can be used in tables, templates, and records.

Using the WORD declaration, the programmer can declare an identifier for any bit string (called a byte) that lies within a single word or for any set of bit strings, each of which occupies a fixed position within each word of an array. WORD declarations can appear alone or within templates, tables, and records. The same syntax for WORD declarations is used in tables, templates, and records. The form for a partial-word declaration is as follows:

```
WORD: {word-id} [byte-id1( {start},width)
                ,byte-id2( {start},width)
                    :
                ,byte-idn( {start},width)] ;
```

word-id An optional FORTRAN identifier that can be used to refer to the complete 64-bit word, or a FORTRAN array specifier of one or more dimensions

byte-id$_i$ A SKOL identifier that can be used to refer to part of the word

start Starting bit number of the byte.

wiath Number of bits in the byte

The left-most bit in a word is numbered 0. If *start* is omitted, the first or next available bit in the word is assigned, and the following comma becomes optional. The *start* values in any particular *word-id* need not be in order, and fields may overlap. However, the default value when *start* is omitted is the bit position immediately following the preceding field (if any).

The WORD declaration causes *word-id* to be declared as an integer.

Example:

```
WORD: FPVAL [SIGN(1), EXP(15), SIGNED_EXP(0,16),
      MANTISSA(48)];
```

In the example, FPVAL is defined as an integer variable; that is, the SKOL translator passes the following line to FORTRAN:

```
INTEGER FPVAL
```

Four partial word variables, or bytes, are declared at the same time in this case. SIGN refers to bit 0 of FPVAL, and EXP refers to the next 15 bits. SIGNED_EXP acquires the value of the first 16 bits, the same ones used by SIGN and EXP. Bits 17 through 64 are assigned to MANTISSA.

The following example illustrates the use of an array specifier as the *word-id*.

Example:

```
WORD: MATRIX(0..9,0..9)[SIGN(1), EXP(15), SIGNED_EXP(0,16),
      MANTISSA(48)];
```

Each of the 100 words in MATRIX has the same structure as the single word FPVAL. MANTISSA(I,J) refers to the right-most 48 bits of MATRIX(I,J).

Byte identifiers are not passed on to FORTRAN; rather, each reference to a byte is expanded into an instance of the BYTE pseudo function, which SKOL defines as follows:

BYTE (*word-id*, *start*, *width*)

word-id Any expression representing a full 64-bit word

start The starting bit number of the byte

width The number of bits in the byte

The BYTE pseudo function may be coded explicitly rather than allowing it to be generated implicitly by a WORD declaration. It may appear on either side of the equal sign in an assignment statement and in general may be used as if it were an integer variable or a Boolean expression. For instance, to operate on a byte within a variable (X) that has been declared normally, the following could be coded:

Example:

```
J = BYTE (X, 48, 16);
BYTE (X, 48, 16) = 0;
INCR BYTE (X, 48, 16);
K = IABS (BYTE (X, 48, 16));
```

The *word-id* (X in the example) may be a variable or expression of any type, although it usually is an integer variable or array element. Referring to the first line, the value of the last 16 bits of X does not change if another value is subsequently assigned to J; nor does J change if a new value is subsequently assigned to X.

Most of the standard SKOL macros are transportable to other machines. To transport the partial word feature (both the WORD and BYTE keywords), simply redefine the two BYTE macro definitions that can be found near the end of the SKOLTXT file.

TEMPLATES

A template is a data structure that allows the declaration of full-word and partial-word fields within itself and is always declared inside a major segment. A template can occupy a fixed area in memory, or it can be used with a BASE POINTER to provide an alternative for the space it currently overlays.

In normal use, a template is associated with a pointer which must then be assigned an address value before the field names that belong to the template have meaning. By reassigning a new address value to the pointer, the template can be moved to another area of memory, associating new values with the template's field names.

Templates can be used to access previously unstructured data, such as the contents of a memory pool, a stack, or a heap. They can also be used to access structured data, such as operating system tables, in the same way that assembly language programs access structured data using GET and PUT macros. Any number of templates can overlay the same area in memory, creating alternative definitions for the overlaid variables.

A template declaration has the following form:

```
TEMPLATE fid {(0..ubound)} :
  WORD wn1: {word-id1} {[byte-list1]};
  WORD wn2: {word-id2} {[byte-list2]};
  .
  .
  .
  WORD wnn: {word-idn} {[byte-listn]};
ENDTEMPLATE;
```

fid — A FORTRAN identifier which becomes the name of an array

ubound — An optional integer constant that defines the upper bound of the array; if (0..*ubound*) is omitted, (0..0) is assumed. If the parentheses are present but the "0.." is omitted, a warning is given and (0..0) is assumed.

wn_i — Word number is an integer between 0 and *ubound* that specifies an array index for the corresponding *word-id*.

$word\text{-}id_i$ — A SKOL identifier that can be used to refer to the entire 64-bit word, or an array specifier of the form *word-id*(0..*n*).

$byte\text{-}list_i$ — An optional list of one or more byte identifiers specified using the form:

```
byte-id({start},width)
```

byte-id — A SKOL identifier used to refer to the byte

start — Starting bit number of the byte. If omitted, the first or next bit number in the word is assigned, and the following comma becomes optional. *start* values need not be given in ascending order in a word, and fields may overlap.

width — Number of bits in the byte

Fields defined by *byte-ids* can overlap within a particular word, and words or arrays defined by *word-ids* can coincide or overlap.

A template is normally associated with a pointer variable the same way a variable name or array name is. (See Based Variables in section 2.) The form for declaring a template's base pointer is:

BASE POINTER FOR fid_1: fid_2;

fid_1 The FORTRAN identifier used to name a template

fid_2 The FORTRAN identifier of the variable which will contain the base address of the template and determine its location

In the absence of a BASE POINTER declaration for a specific template, that template is allocated space like any ordinary array. The amount of space allocated depends only on the *ubound* value; if *ubound* is zero, one word is allocated.

In normal use (that is, with a BASE POINTER), a template occupies no space of its own but provides an alternative definition for the space it currently overlays. When a template is used this way, its *ubound* specification's purpose is to enable bounds checking to be turned on for references in the template. Bounds checking is turned on with the ON=O option on the CFT statement or with the CDIR(BOUNDS) SKOL statement (see section 10). Bounds checking can never be turned on for an array whose beginning and ending indices are identical, as when *ubound* is 0.

TABLES

A table is a data structure whose size can vary between 0 and all of the available memory in the CRAY-1. A program can declare up to 63 tables, each with a different structure. Each table consists of a variable number of entries, and each entry consists of a fixed number of words. Like a template, a table entry's components, or fields, are defined with WORD declarations. The major syntactical difference is that *word-ids* are required in a table declaration and are optional in a template declaration.

Table space is allocated dynamically just after blank common, while the tables' base addresses and lengths are maintained in a common block labeled TM. Dynamic allocation gives tables an advantage over records, which are fixed in size and number once they are declared.

A table declaration takes the following form:

```
TABLE tn {(length)}:
  WORD wn1: word-id1 {[byte-list1]};
  WORD wn2: word-id2 {[byte-list2]};
  .
  .
  .
  WORD wnn: word-idn {[byte-listn]};
ENDTABLE;
```

tn — An integer between 1 and 63 inclusive or a SKOL identifier defined as such an integer

length — An optional integer constant or variable that defines the number of words in every table entry; if it is omitted, 1 is assumed. Better code is generated if 1 is omitted.

wn_i — A word number; an integer between 0 and *length*-1 that specifies the offset (in words) from the beginning of a table entry for the corresponding *word-id*.

$word\text{-}id_i$ — A SKOL identifier that refers to the complete 64-bit word, or a FORTRAN array specifier of the form *word-id*(0..*n*)

$byte\text{-}list_i$ — An optional list of one or more byte identifiers specified using the form:

```
byte-id ({start} ,width)
```

byte-id — A SKOL identifier used to refer to the byte

start — Starting bit number of the byte. If omitted, the first or next bit number in the word is assigned, and the following comma becomes optional. *start* values need not be given in ascending order in a word, and fields may overlap.

width Number of bits in the byte

If *byte-list* is omitted, its enclosing brackets may also be omitted.

TABLE POINTERS

Table entries cannot be accessed directly. They are reached through pointer variables. A pointer variable is declared to belong to a specific table and cannot be used to access an entry in any other table. Like the TABLE declaration, a pointer declaration must appear within a major segment and cannot be declared before the table is declared. The form of a pointer declaration is:

POINTER TO TABLE *tn*: sid_1 {(*size*)} {,sid_2 {(*size*)} ... ,sid_n {(*size*)}} ;

tn An integer between 1 and 63 inclusive that identifies the table or a SKOL identifier defined as such an integer

sid_1 A SKOL identifier naming the pointer

size An integer constant specifying, if present, the number of pointers in an array of pointers; only one dimension is allowed.

Example:

```
POINTER TO TABLE 1:  P,Q,R,TP(20);
```

Constant values can be used as table pointers if they are declared as follows:

CONSTANT POINTER TO TABLE *tn*: sid_1 = $iexp_1$ {,sid_2 = $iexp_2$ ...

,sid_n = $iexp_n$};

tn An integer between 1 and 63 inclusive that identifies the table or a SKOL identifier defined as such an integer

sid_i A SKOL identifier

$iexp_i$ An integer expression

Each declared pointer points to a single entry in the specified table.

Example:

```
CONSTANT POINTER TO TABLE 2:  HEAD=0;
```

The pointer HEAD can be used to access any field in the first entry of table 2.

Before any reference is made to a table entry, it must be allocated by TMATS or TMADW. In addition, before table space can be allocated, the tables must be initialized by a call to TMINIT. TMINIT requires as its input an array of interspace values, one value for each table to be used. Interspace values specify the number of words that are to be left unused at the end of each table as a buffer between it and the next table. At least one word is required at the end of any table with which TMSRC (table search) is used. (Alternatively, SKOL programmers can write a search loop in SKOL which is more readable and usually more efficient.) To initialize the table space, statements such as the following are required:

```
  CONSTANT: NTAB = ntab;      -- Maximum number of tables used
  TABLE_COMMON;               -- Provides access to the tables
  DATA: BTAB/NTAB*1/;         -- Defines table interspace values

BEGIN:
  CALL TMINIT;
```

The TABLE_COMMON declaration must be present in any program segment that uses tables and should follow all table declarations. It generates the following FORTRAN code:

```
INTEGER BTAB(64), NTAB(64)
COMMON /TM/ BTAB, NTAB
```

TMINIT rearranges the data in the BTAB (base address) array, shifting the interspace values to the left in each word and replacing them with the initial base addresses calculated from those interspace values. TMINIT also presets all of the LTAB (length) values to 0, making each table initially empty.

After TMINIT is called once, it must never be called again, and BTAB must never be modified by the user. LTAB must not be modified by the user except to discard the contents of a table by setting its length to 0, or to discard the final part of a table by reducing its length. For that purpose, SKOL provides a special pseudofunction, LENGTH(TABLE *tn*), which can be used on the left side of an assignment statement.

To illustrate, suppose a symbol table is implemented using a SKOL table structure. If the maximum length of a symbol's name is 32 characters, declarations such as the following might suffice:

Example:

```
CONSTANT: NTAB = 1, SYMBOL_TAB = 1;

TABLE SYMBOL_TAB (5):
  WORD 0: ATTRIBUTES [RELO(1), ADDR(40,24)];
  WORD 1: SYMBOLNAME (0..3);
ENDTABLE;

POINTER TO TABLE SYMBOL_TAB: S,T;
TABLE_COMMON;
DATA: BTAB /NTAB*1/;
```

Table 1 is thus established as having five words in each entry. The first word in each entry may be referred to by the construction ΛS.ATTRIBUTES, and parts of that word may be addressed as ΛS.RELO and ΛS.ADDR. A single entry in the table can be allocated and initialized by the following code:

```
CALL TMATS (SYMBOL_TAB, 5);
DO I = 0 TO 3:
  ΛS.SYMBOLNAME(I) = '        ';
ENDDO;
ΛS.ADDR = LOCATION;
ΛS.RELO = RELOCATE;
INCR S;
```

An alternative method uses TMADW to simultaneously allocate and initialize one word at a time:

```
CALL TMADW (SYMBOL_TAB, SHIFT(RELOCATE, 63)+LOCATION);
CALL TMADW (SYMBOL_TAB, '        ');
CALL TMADW (SYMBOL_TAB, '        ');
CALL TMADW (SYMBOL_TAB, '        ');
CALL TMADW (SYMBOL_TAB, '        ');
INCR S;
```

Note that while table numbers range from 1 (not 0) to 63, entry numbers (pointer values) and word numbers begin with 0. In both of the above examples, the S points are assumed to be the next available entry. When the program starts, S must be 0.

NOTE

If table entries are linked to each other, ambiguity between a pointer to entry 0 and the NIL pointer is possible. To avoid this ambiguity, declare NIL=-1 in a CONSTANT statement or make entry 0 a header record or dummy entry that is not pointed to by any other entry.

References to word 0 are more efficient than references to other words. For efficiency, one can redefine the SYMBOLNAME array in each entry as follows:

```
WORD 0: SYMBOLNAME(0..4);
```

The iteration range in the above example is then 1 to 4 instead of 0 to 3, and the DO-loop runs slightly faster. Two or more definitions can exist for WORD 0 or for any other word. It does not matter if some words are skipped over and not given any names.

With SYMBOLNAME starting at word 0, the following code can be used to search the table:

```
UNTIL HIT OR MISS:
  DO T = 0 TO S:
    UNTIL UNEQUAL:
      DO I = 1 TO 4:
        IF ∧T.SYMBOLNAME(I)<>NEWTOKEN(I):
          UNEQUAL;
        ENDIF;
      ENDDO;
      HIT;
    ENDUNTIL;
  ENDDO;
  MISS;
THEN:
  [HIT]: ...
  [MISS]: ...
ENDUNTIL;
```

However, if a search is time-consuming, a pointer field added to each entry allows the entries to be formed into a linked list.

RECORD TYPES

SKOL records are static structures; the maximum number of records allocated within a given record type is fixed by the RECORD declaration. A record type corresponds to a table and an individual record to a table entry. Whereas table entries consist of words, arrays of words, and partial words, records can contain any FORTRAN data type except CHARACTER and POINTER. SKOL pointers are also allowed in records.

Each record belonging to a particular record type consists of a fixed number of components called fields. Each field contains any simple type of data or an array. A FORTRAN identifier is used to name a record type. A SKOL identifier is assigned to each field by the programmer.

The macro translator replaces these SKOL identifiers with FORTRAN identifiers of the form:

xxxxn

xxxx The record type name

n An ordinal number assigned internally to the field

The fields receive their ordinal numbers, which are appended to the record name to give the field name used by FORTRAN, from the order listed by the programmer. For example, the first field listed for a record type named PERSON is renamed PERSON1 by the macro translator, the second field is PERSON2, and so on.

For all the field names to be translated into valid FORTRAN identifiers containing no more than eight characters, records containing between one and nine fields must have a type name no longer than seven characters. Likewise, records containing 10 to 99 fields can have a type name of no more than six characters, and records with 100 fields or more can have a maximum of five characters in the type name.

A record type is declared in the following manner:

```
RECORD fid(extent):
  ftype1: sid1 {(size)} {,sid2 {(size)} ... ,sidn {(size)}};
  ftype2: sid1 {(size)} {,sid2 {(size)} ... ,sidn {(size)}};
  .
  .
  .
  ftypen: sid1 {(size)} {,sid2 {(size)} ... ,sidn {(size)}};
ENDRECORD;
```

fid A FORTRAN identifier, subject to the length restrictions described above, naming the record type

extent An integer constant defining the maximum number of records of that type

$ftype_i$ A data type including the types POINTER and WORD; strings and user-defined types are not allowed

sid_i A SKOL identifier naming each field

size An optional dimension clause; each sid_i may represent either a single variable or an array of one or more dimensions

Example:

```
RECORD PERSON (100):
  POINTER: NEXT, FATHER, NEIGHBOR(0..7);
  INTEGER: NAME(15), AGE, ID_NUMBER, LEAVE(12);
  REAL: PAY;
  LOGICAL: MARRIED;
ENDRECORD;
```

Each of the 100 records declared in this example contains six fields that hold data on an individual and three that link records together. The pointer fields NEXT and FATHER and the eight NEIGHBOR pointers, all of which take integer values, point to other records.

SKOL records can be accessed in one of two ways: through dynamic allocation or by treating each field as a separate array, which it is. If dynamic allocation is chosen, it is necessary to employ the

keywords MAKEAVAIL, NEW, and FREE (described later). It is also necessary to reserve the first field in the record for SKOL's internal use. SKOL inserts integers into that field (NEXT in the above example) that link the records of a type together. Accessing records in the desired order requires correct values in the first field; if the programmer reassigns the values of the first field, unwanted results are possible. Therefore, it is appropriate to make the first field a POINTER and call it NEXT. The programmer is free to use subsequent pointer fields in any way.

If dynamic allocation is not required, and MAKEAVAIL, NEW, and FREE are not used, the first field of a record type need not be a pointer variable such as NEXT, and the *extent* assigned to a record type may be replaced optionally by a subscript range.

Example:

```
RECORD AFRAMES(0..3):
  INTEGER: MEMORY(0..16383);
    .
    .
    .
ENDRECORD;
```

All records of a particular type may be put into common storage by naming the type in a COMMON statement.

Examples:

```
COMMON/BLOK/PERSON;      -- Puts PERSON into COMMON under
                         --  the label BLOK.
COMMON//PERSON;          -- Puts it into blank COMMON.
COMMON PERSON;           -- Invalid.
```

RECORD POINTERS

Records, like tables, cannot be accessed directly. They are reached through pointer variables, which are distinct from pointer fields within records. A pointer variable can point only to records of a particular type. Pointer variables must be declared as such in a statement of the following form:

POINTER TO *rtype*: sid_1 {(*size*)} {,sid_2 {(*size*)} ... ,sid_n (*size*)}} ;

rtype A record type identifier

sid_i A SKOL identifier naming the pointer

size An integer constant specifying, if present, the number of pointers in an array of pointers; only one dimension is allowed

A pointer variable identifies a record of a particular type. To continue with the example of the record PERSON:

Example:

```
POINTER TO PERSON: WORKER, FOREMAN(6), P, LAST;
```

A constant value can be used as a record pointer if it is declared as follows:

CONSTANT POINTER TO *rtype*: sid_1 = $iexp_1$ {,sid_2 = $iexp_2$ ... {,sid_n = $iexp_n$};

rtype A record type identifier

sid_i A SKOL identifier

$iexp_i$ An integer expression

The declared pointer points to a single record of the given type.

Example:

```
CONSTANT POINTER TO AFRAMES: AP0=0, AP1=1, AP2=2, AP3=3;
```

The pointer AP0 may be used to access any field in record zero of the type AFRAMES.

MAKEAVAIL statement

Once a record type has been declared, it must be made available for use with the MAKEAVAIL statement before the NEW and FREE statements can be executed on it. MAKEAVAIL returns all records of the designated type to the pool for re-use. The statement takes the form:

```
MAKEAVAIL rtype;
```

rtype A previously declared record type identifier

Example:

```
MAKEAVAIL PERSON;
```

NEW statement

Once the record type is made available and pointers are declared, a record may be assigned to a pointer with the NEW statement.

```
NEW pointer;
```

pointer A variable previously declared as a pointer to a record type. SKOL issues an error message if it has not been so declared.

Example:

```
NEW WORKER;
```

The statement in the above example allocates a new record from the PERSON type. If there are no records from that type available on the free list, the program aborts with an error message. To do one's own error checking, a form such as the following may be used:

Example:

```
IF PERSON_AVAILP = NIL:
   CALL ABORT ('No more PERSON records available');
ENDIF;
NEW WORKER (NOVALIDATE);
```

The SKOL identifier *rtype*_AVAILP refers to an internally maintained pointer to the next available record of that type. The NOVALIDATE clause disables SKOL's normal error checking for efficiency's sake.

FREE statement

When no more references are to be made to a record, the record may be released by the FREE statement. The form is the same as that of the NEW statement.

```
FREE pointer;
```

pointer A variable previously declared as a pointer to a record type. SKOL issues an error message if it has not been so declared.

POINTER USAGE

Accessing a specific field of a table entry or record requires an up arrow symbol, Λ, and a special form called the field designator.

```
Λ pointer.field
```

pointer A variable previously declared as a pointer to a record type or table

field A variable previously declared as a field of a record type or table

The pointer name must be subscripted if it is declared as an array; the same is true for the field name.

Example:

```
Λ WORKER.AGE = 25;
Λ WORKER.LEAVE(L) = 2;
Λ FOREMAN(N).NAME(1) = 'Z'R;
```

SKOL does not allow the use of two up arrows in a single statement to permit access of a field in a second table entry or record through a pointer field of the first. Two statements making use of two valid pointers are necessary, as the following examples illustrate:

Examples:

```
Λ(ΛP.NEXT).VAL = 1        -- Invalid in SKOL.

T = ΛP.NEXT;              -- Valid, assuming T and P have
ΛT.VAL = 1;               --  been declared as POINTERS to
                          --  the correct record type or table.
```

NOTE

SKOL checks to determine that the pointer and the field with which it is used both belong to the same record type or table. But no checking is done to ensure that a pointer value falls within the declared extent of the record type with which it is used, or that it points to a table entry for which space has been allocated.

WITH ... ENDWITH statement

A convenient short form for designating table and record fields is available in conjunction with the WITH statement. A pointer that is the object of a WITH statement may be omitted from the ensuing field designators.

```
WITH pointer:
  block
ENDWITH;
```

pointer A simple or subscripted variable previously declared as a pointer to a record type or a table

block A sequence of statements that may make use of the shortened form of the field designators

Within the block, any field referenced through the designated pointer is accessible by the following shorter form, which omits the pointer name:

```
Λ.field
```

field A SKOL identifier previously declared as a field of a record type or a table

Example:

```
WITH WORKER:
  Λ.AGE = 25;
  Λ.LEAVE(3) = 2;
  Λ.NAME(1) = 'Z'R;
ENDWITH;
```

NOTE

WITH structures cannot be nested; that is, no WITH block may contain another WITH block.

LINK ... ENDLINK statement

Records or table entries that are strung together in a linked list may be traversed from one particular point on the list to another by means of the LINK statement. The LINK command is a variation of the FOR statement that moves from record to record, or from entry to entry, via the specified pointer field. It takes the following form:

```
LINK FOR pointer = int-exp TO int-exp USING field:
  block
ENDLINK;
```

pointer A variable previously declared as a pointer to a particular record type or table

int-exp Any integer expression

field A variable previously declared as a field of the same record type or table and which contains appropriate pointer values

block A sequence of statements

Example:

```
SUM = 0;
LINK FOR P = WORKER TO NIL USING NEXT:
   SUM = SUM + ΛP.AGE;
ENDLINK;
```

The above example begins with a record indexed by the value of WORKER, adds the value of that record's AGE field to the variable SUM, and moves on to the record indexed by the value of the NEXT field. It moves from record to record until encountering NIL in a NEXT field. NIL is a standard identifier that indicates an undefined pointer value and customarily marks the end of a list. NIL is defined as zero by default, but it may be redefined if necessary. The final value of SUM is the total value of the AGE fields for all records accessed, including the one with NIL in its NEXT field.

Instead of using dynamic allocation (MAKEAVAIL, NEW, and FREE), a set of records may be accessed as an array of records, which it actually is. Pointer variables must still be used, but they may be treated as integers, which they are.

The following example assumes that the proper declarations have been made and that no records from PERSON are currently active.

Example:

```
LAST = NIL;
DO WORKER = 1 TO 50:
   ΛWORKER.NEXT = LAST;
ENDDO;
TOP = 50;
LINK FOR P = TOP TO NIL USING NEXT:
   WITH P:
      Λ.NAME(1) = '#';
      Λ.AGE = 20;
   ENDWITH;
ENDLINK;
```

Another example reveals how a number of statements described in this section may be used together.

Example:

```
      RECORD PERSON (100):
         POINTER: NEXT, PREVIOUS;
         INTEGER: AGE, INITIAL(3), ID_NUMBER;
         REAL: PAY_PER_WEEK;
         LOGICAL: MARRIED, ON_LEAVE;
      ENDRECORD;
      POINTER TO PERSON: WORKER, FIRST, LAST, P, Q;

   BEGIN:
      FORMAT PERSONNEL = (1X, 3R1, 1X, 2I5, F10.2);
      MAKEAVAIL PERSON;
      FIRST = NIL;
      LOOP:
         NEW P;
         READ ($IN, PERSONNEL) (FOR 3: ΛP.INITIAL), ΛP.ID_NUMBER,
            ΛP.AGE, SALARY;
         ΛP.PAY_PER_WEEK = SALARY/52.0;
         ΛP.MARRIED = ΛP.AGE >= 21 AND ΛP.ID_NUMBER >20000;
      WHILE ΛP.AGE >0;
         OUTPUT (P:I3,
            ΛP.ID_NUMBER:I7,
            ΛP.PAY_PER_WEEK:G12.5,
            ΛP.AGE:I3);
         ΛP.NEXT = FIRST;
         IF FIRST <> NIL:
            ΛFIRST.PREVIOUS = P;
         ELSE:
            LAST = P;
         ENDIF;
         FIRST = P;
      ENDLOOP;
      ΛFIRST.PREVIOUS = NIL;
      FREE P;
```

TEXT MANIPULATION 6

One reason for SKOL's usefulness is its extensibility. To supplement its built-in data structures and control statements, SKOL provides text manipulation features making it possible to add to or modify the language. This section describes constant declarations, macro definitions, text substitution, and debugging aids.

CONSTANT

A name may be assigned a constant value in SKOL with the CONSTANT declaration. After a constant is declared, the value replaces any occurrence of the constant identifier in the program text. The value assigned in a CONSTANT declaration is differentiated from the value assigned in a PARAMETER statement (described below). The rules for a constant value are as follows:

- The value may consist entirely, but not partly, of a literal string of any length, with or without an R, L, or H suffix.
- The value may consist of any arbitrary string of characters that does not contain a semicolon, unbalanced parentheses, or unbalanced brackets.

Such a declaration is of the form:

CONSTANT: sid_1 = *value*{,sid_2 = *value* ... ,sid_n = *value*};

sid_i	A SKOL identifier
value	A literal string of any length or any arbitrary string of characters that does not contain a semicolon, unbalanaced parentheses, or unbalanced brackets

The scope of any CONSTANT definition depends on whether the definition lies within or outside of a major program segment. A local constant, declared within a major segment, cannot be referred to outside that segment. A global constant, declared before any segment or between two major segments, can be referred to by any statement that physically follows it in the program text.

Example:

```
CONSTANT: PI = 3.14159, CM_PER_INCH = 2.54;
X = (2.0 * PI * RADIUS) * CM_PER_INCH;
.
.
.
CONSTANT: LIMIT = 50, HYPHEN = '-'R;
REAL: A(LIMIT, LIMIT), B(2, LIMIT);
DO I = 1 TO LIMIT:
  B(1,I) = 0;
ENDDO;
OUTPUT ((FOR 132: HYPHEN));
```

PARAMETER

SKOL's version of the PARAMETER statement operates in a manner similar to the CONSTANT statement except that FORTRAN performs the text substitution. However, a constant declared in a PARAMETER statement is always local to the major program segment (a MAIN segment, function, subroutine, or BLOCKDATA segment) in which it is declared, and the value must conform to FORTRAN rules. The PARAMETER statement has the same form as the CONSTANT statement:

PARAMETER: fid_1 = *value* {,fid_2 = *value* ... ,fid_n = *value*};

fid_i	A FORTRAN identifier
value	A constant or symbolic name of another constant

An identifier used in a PARAMETER statement must first appear in a type declaration, and the value assigned to it must be of that type.

Example:

```
REAL: PI;
LOGICAL: A;
PARAMETER: A = TRUE, PI = 3.1415926;
```

MACRO

Macro procedures with keyword parameters (but without positional parameters) may be defined anywhere in a SKOL program. Such macro definitions take the form:

```
MACRO sid (par1 = def1 {,par2 = def2  ...
      ,parn = defn}) = 'text'
```

sid	A SKOL identifier naming the macro
par_i	A SKOL identifier naming a formal parameter
def_i	An optional default value
text	A series of one or more complete SKOL statements

The scope of a macro definition is local to the major program segment in which it appears. If the definition appears outside of a major segment, it is global to all segments that follow it.

Example:

```
MACRO ORDER (REL =<, X,Y) =
  ';IF NOT (X REL Y): SWAP (X,Y); ENDIF;';
```

The keyword parameters may represent not only variables and expressions, but relations, operators, statements, and procedure names. In fact, any arbitrary string of characters that does not contain a literal string, a semicolon, unbalanced parentheses, or unbalanced brackets is a valid actual parameter or default value.

When a macro procedure is invoked, the actual parameters of the invocation are substituted for the formal parameters of the macro definition. The positions of the actual parameters in the invocation are not important. Parameters not specified in the invocation are given the assigned default values. If the definition does not specify a default value, an error message is generated. The form of a macro invocation is:

$$sid\ (par_1 = xexp_1\ \{,par_2 = xexp_2\ \ \ldots\ ,par_n = xexp_n\});$$

sid A SKOL identifier giving the name of a previously defined macro

par_i A parameter keyword corresponding to one in the previous macro definition

$xexp_i$ An extended expression that can be either a variable, an expression, a relation, an operator, a procedure name, or a statement. It cannot contain a semicolon, a literal string, unbalanced parentheses, or unbalanced brackets.

A macro invocation statement must begin with the name of the macro; that is, a macro cannot define a function. In the example below, SWAP is assumed to be a SKOL macro created by the user in another MACRO statement or in a DEFINE statement (see the example given under DEFINE).

Example:

```
ORDER (X = A(2), Y = B(K));
ORDER (REL = > = , Y = T, X = P(J));
```

These two lines are translated into the following program text:

```
IF NOT (A(2)< B(K)): SWAP (A(2), B(K)); ENDIF;
IF NOT (P(J)>= T): SWAP (P(J), T); ENDIF;
```

NOTE

Formal parameter names become keywords for all the source text within the scope of the MACRO declaration. Substitutions are made for them even if they occur outside of the context of a macro invocation.

DEFINE

The CONSTANT declaration is a special case of a more general pattern-matching statement called DEFINE, which takes the form:

```
DEFINE 'pattern' = 'replacement';
```

pattern and *replacement*
Sequences of characters and/or special operators

Subsequent occurrences of the pattern are replaced by a copy of the replacement text. Because of their special meanings, the single quote ('), the pound sign (#), and the at sign (@) must be doubled when used in their literal sense in a pattern or a replacement string.

The scope of a DEFINE statement is local to the major program segment in which it appears. If the DEFINE appears outside of a major segment, it is global to all segments that follow it.

Example:

```
DEFINE ';INITIALIZE;' = ';A = 0;B = 1;P(2) = 3;';
```

The initial semicolon in the pattern matches the semicolon or colon concluding the previous statement when

```
INITIALIZE;
```

is encountered. This happens because the standard SKOL macros replace all legitimate terminal colons with semicolons. Thus, a semicolon in a pattern matches with any colon in the text that is used correctly as a statement or label terminator. Statements that are terminated by colons are those that begin with the keywords BEGIN, COROUTINE, DO, ELSE, ELSEIF, ENTRY, FOR, FUNCTION, IF, LINK, LOOP, MAIN, RECORD, REPEAT, ROUTINE, SUBROUTINE, TABLE, TEMPLATE, THEN, UNTIL, WHEN, WHILE, and WITH. Labels within a WHEN block (in a scalar case structure) or a THEN block (at the end of a situation case structure) are also terminated by colons. Wherever else a colon is required, it is not a terminator and is eventually discarded. For instance, a colon is not replaced by a semicolon when it occurs in an INPUT or OUTPUT statement, a TRACE statement, a declaration statement, or before the IS in a WHEN statement.

If two or more identical patterns are defined within the same scope, the latest definition overrides any earlier definitions, and no warning is given. The order of definition is important even among patterns which start with the same series of characters.

POSITIONAL PARAMETERS

So far only simple macros with no parameters have been described. But by placing the special operator # at various places in the pattern, and by placing # followed by a digit in the replacement string, the user can define text substitution rules with parameters. In fact, SKOL is translated into FORTRAN by just such rules.

Example:

```
DEFINE ';SWAP(#,#);' = ';R99999 = #1;#1 = #2;#2 = R99999;';
SWAP(A(I,J),B);
```

The second line of the example is translated into:

```
R99999 = A ( I , J ) ; A ( I , J ) = B ; B = R99999 ;
```

Each # on the pattern side of a DEFINE statement matches any character sequence that is properly located with respect to the rest of the pattern. In the above example, the two parameters must be parenthesized and separated by a comma. The pattern must match a complete statement--everything between and including two semicolons--in the SKOL source. The #1 on the replacement side of the DEFINE statement refers to the matching text for the first # on the pattern side, #2 to the second, and so on for as many as nine parameters.

A # does not match a string with improperly matched parentheses or brackets. For this reason, the first comma between I and J in the example is taken as part of the first parameter rather than as part of the pattern. If that comma were matched with the comma in the pattern, parameter 1 would be

```
A(I
```

and parameter 2 would be

```
J),B
```

The SKOL programmer may inadvertently defeat the matching process by omitting a parenthesis.

SPECIAL HANDLING FOR BLANKS

SKOL's treatment of blanks tends to aid rather than impede the pattern matching performed by a DEFINE statement. SKOL has three rules with regard to blanks in the source text.

1. Blanks are inserted into the input stream between a pair of characters whenever one character is alphanumeric and the other is not. This rule isolates identifiers from the surrounding text for more precise matching. Literal strings are unaffected by this rule, except those in a DEFINE statement, where blanks are inserted so that the replacement text looks like text from the input stream. The special characters $ and _ as well as the operators # and @ (used in DEFINE statements) are considered to be alphanumeric characters for this purpose. Blanks are not inserted on the left side (the pattern) of a DEFINE statement.
2. An end-of-line character is treated as a blank.
3. Any number of consecutive blanks or end-of-line characters are treated as a single blank.

Returning to the code of the most recent example, the application of the first rule causes a string such as

```
SWAP(A(I,J),B)
```

to become

```
SWAP ( A ( I , J ) , B )
```

Any literal strings, such as the pattern and replacement strings in the DEFINE statement, are unaffected. Each line is scanned for a possible macro substitution immediately after it is read. At this point, the string

```
DEFINE
```

is replaced by

```
%
```

and blank insertion is temporarily enabled within the literal string that provides the replacement part of the definition. The % may be used in place of DEFINE and the blank that follows it. If the % is used, blank insertion remains disabled.

Whether or not blanks are present in a pattern, the presence of blanks in the source text does not prevent a match. In the first example under the DEFINE heading, for instance, a match is made even if blanks are in the source text between INITIALIZE and the surrounding semicolons.

Example:

```
BEGIN:
  INITIALIZE     ;
```

A match is made and substitution takes place in the following manner:

1. During the input phase, the end-of-line following BEGIN: is replaced by a blank, and the blanks following INITIALIZE are reduced to a single blank.
2. ;BEGIN: is matched by a standard SKOL macro and is replaced by the string ;CONTINUE; .
3. The pattern ;INITIALIZE; is found to match the source string ; INITIALIZE ; .
4. The replacement part of the macro definition, ; A = 0 ; B = 1 ; P (2) = 3 ; , is substituted for ; INITIALIZE ; .

While blanks are sometimes ignored in the source text, they are never ignored in the pattern part of a macro. In fact, blanks are often needed in the pattern to prevent unintentional matches. For instance:

Example:

```
DEFINE 'CON' = '6.023E25';
```

The above pattern matches any occurrence of CON, including those in the word CONSTANT and even in an identifier such as NEW_CONDITION. In order to match only the word CON, blanks should be inserted on either side of the pattern.

```
DEFINE ' CON ' = '6.023E23';
```

This macro matches only those occurrences of CON that stand alone.

ACCIDENTAL RECURSION

There is another potential error inherent in a statement such as the following:

Example:

```
DEFINE ' OR ' = '.OR.';
```

The difficulty arises because the replacement part, after blanks are inserted, contains an instance of the pattern within itself. The replacement part is scanned a second time and replaced again, leaving an incongruity such as the following in its place:

```
.. OR ..
```

This recursive replacement continues until the SKOL macro translator aborts with the message, "EXPANSION BUFFER OVERFLOW."

Surrounding the part of the replacement string that is in danger with quotation marks tells the translator not to rescan. The above example may be coded as follows to prevent rematching of the substituted text:

```
DEFINE ' OR ' = '".OR." ';
```

Note that the replacement part in this example must end in a blank to replace the final blank in the pattern, and this blank must not be inside the quotation marks. Including the trailing blank allows the subsequent text to match patterns beginning with blanks.

TRACE

TRACE is useful for adding temporary debugging output to a program. Changes to a variable may be traced through a program by using the following form:

TRACE *var* {:*fmt*};

And more than one variable may be traced through the more general form:

TRACE var_1 {:*fmt*} {,var_2 {:*fmt*} ... ,var_n {:*fmt*}} ;

var_i A variable name in one of the following forms:

v	A simple variable
a()	An array element (the empty parentheses can stand for any number of subscripts)
Λ.*v*	A simple variable that is a field within a table entry or record (matches either Λ.*v* or Λ *ptr*.*v*)
Λ.*a*()	An array element that is a field within a table entry or record (here too, the empty parentheses can stand for any number of subscripts)
a(:*sf*)	An element of a 1-dimensional array, for which the subscript value is to be printed
Λ[:*pf*].*v*	A simple variable within a table entry or record for which the pointer value is to be printed
Λ[:*pf*].*a*(:*sf*)	An element of a 1-dimensional array within a table entry or record for which the pointer value and/or the subscript value must be printed

fmt The output format for the traced variable's value

sf The output format for the subscript expression's value

pf The output format for the pointer's value

If :*fmt* is omitted, :O22 is assumed by default. If the colon prefix stands alone (no *fmt* is coded), :$GENFORM is used by default. Although $GENFORM is normally defined as O22, so that the two defaults act identically, the programmer can make them act differently by redefining $GENFORM at any point in the program. Only those variable assignments that follow the redefinition of $GENFORM are affected.

If :*sf* is omitted, the subscript value is not printed. If the colon prefix stands alone within the parentheses (no *sf* is coded), the subscript value is printed using the format supplied by $GENFORM.

If [:*pf*] is omitted, the pointer value is not printed. If the colon prefix stands alone within the square brackets (no *pf* is coded), the pointer value is printed using the format specified by $GENFORM.

Example:

```
TRACE Z, Λ.F;
```

The TRACE macro generates a specific macro for each variable in the list. For instance, TRACE Z causes all subsequent assignments to Z to be followed by the execution of

```
OUTPUT('*** TRACE (line @SQ):  Z = ', Z:O22,';');
```

and TRACE Λ.F causes all assignments to the field F to be followed by the execution of a statement such as

```
OUTPUT('*** TRACE (line @SQ):  Λ.F = ', Λ.F:O22,';');
```

Any assignment to the field F produces output. @SQ generates a character string representing the current line number in the SKOL source.

To illustrate the use of the *sf* and *pf* forms, suppose that a record contains a declaration such as:

```
REAL: A(20), B(20), C(20);
```

and that the following TRACE statement is in effect:

```
TRACE Λ.A(), Λ[:].B(:), Λ[:I3].C(:I5):F10.2;
```

Suppose further that the following code fragment appears after the TRACE statement:

```
WITH P:
  Λ.A(J) = X/2;
  Λ.B(J/2) = X;
  Λ.C(J*2) = SQRT(B(J));
ENDWITH;
```

The following OUTPUT statements would be generated, one after each of the three assignment statements:

```
OUTPUT ('*** Trace (line @SQ):', Λ.A(J));
OUTPUT ('*** Trace (line @SQ):', Λ.B(J/2)::F10.4, J/2);
OUTPUT ('*** Trace (line @SQ):', Λ.C(J*2), J*2::I5, ::I3);
```

Note that the last item in the final OUTPUT statement is ::I3, which generates the FORTRAN format item :I3. This item, being at the end of the format list, is ignored by CFT. However, the colon prefix may cause a problem in other versions of FORTRAN.

If the name of the pointer P appears explicitly in each of the assignment statements so that the WITH statement is superfluous, the last two OUTPUT statements will look like this:

```
OUTPUT ('*** Trace (line @SQ):', Λ P.B(J/2)::F10.4, J/2, P);
OUTPUT ('*** Trace (line @SQ):', Λ P.C(J*2), J*2::I5, P::I3);
```

The scope of a TRACE statement is local to the major program segment in which it appears. If the statement appears outside a major segment, it is global to all segments that follow it.

NOTRACE

Although any TRACE statement becomes inactive at the end of the major program segment in which it appears, the NOTRACE statement deactivates a TRACE statement. The NOTRACE statement cannot be used to deactivate all the TRACE statements at once. Traced variables can be deactivated only by specifying each one in a NOTRACE statement:

NOTRACE $\{var_1, var_2 \ldots, var_n\}$;

var_i A variable name in one of the following forms:

v	A simple variable
$a()$	An array element (the empty parentheses can stand for any number of subscripts)
$\Lambda.v$	A simple variable that is a field within a table entry or record (matches either $\Lambda.v$ or $\Lambda ptr.v$)
$\Lambda.a()$	An array element that is a field within a table entry or record (here too, the empty parentheses can stand for any number of subscripts)

No format specification is needed in a NOTRACE statement. Tracing is disabled for the specified entities, no matter what trace format was specified in a preceding TRACE statement. No checking is done to ensure that each variable in a NOTRACE statement also appears in an active TRACE statement. At any point in a major program segment, the most recent local TRACE or NOTRACE statement that contains a particular variable name is the one that is active for that variable. When a TRACE or NOTRACE statement appears outside any major segment, its scope is global. Global TRACE statements can be explicitly overridden inside a major segment, but resume their original effect when the segment ends.

VALIDATE

The SKOL translator can insert run-time consistency checks into the generated FORTRAN code for certain kinds of statements. The checks are defined by the user and communicated through the VALIDATE statement, which may appear anywhere in a program text, any number of times. The VALIDATE statement has the general form:

```
VALIDATE ({-} class1 {,{-} class2 ..., {-} classn});
```

class$_i$ One of the following:

WHEN	Check scalar case statements
UNTIL	Check situation case statements
FOR	Check the parameters of FOR statements
RECUR_UNDER	Check for recursion stack underflow
RECUR_OVER	Check for recursion stack overflow
LOOP	Trace DO, LOOP, REPEAT, FOR, and LINK statements
CALL	Trace CALL, EXECUTE, SUSPEND, and ACTIVATE statements
ALL	Do all of the above

The default is:

```
VALIDATE (ALL, -LOOP, -CALL);
```

Coding ALL as the only parameter enables all of the possible run-time checks. A minus sign preceding any parameter supresses checks for that class of statements. If ALL is the only parameter and is preceded by a minus sign, all validation is disabled. If ALL is preceded by a minus sign and followed by other parameters, validation is enabled only for the statement classes that are specifically listed.

The scope of a VALIDATE statement is local to the major program segment in which it appears. If the statement appears outside a major segment, it is global to all the segments that follow it.

Only LOOP and CALL are not enabled in the VALIDATE statement in SKOLTXT that establishes the default validation conditions. The defaults can be overridden at any point in the program with another VALIDATE statement.

Example:

```
VALIDATE (-ALL, WHEN, UNTIL);
```

In the example above, only the statements associated with WHEN and UNTIL are enabled. In the following example, SKOL traces all CALL, EXECUTE, SUSPEND, and ACTIVATE statements that appear between the two VALIDATE statements.

```
VALIDATE (CALL);
.
.
.
VALIDATE (-CALL);
```

WHEN

Enabling the WHEN parameter inserts an output statement at the end of the FORTRAN translation of the scalar case statement. If, during execution, the selector expression in a scalar case statement is not in the proper range, an error message is printed and the program aborts. However, if WHEN validation is disabled, an out-of-range selector expression causes the program to skip to the ENDWHEN statement.

UNTIL

VALIDATE(UNTIL) causes an error message to be printed and the program to abort if no situation statement permitting a jump out of the UNTIL block is ever executed. A loop structure is usually needed inside the UNTIL block. However, if UNTIL validation is disabled, the failure to signal a situation is ignored, and control falls through to the next statement (THEN or ENDUNTIL).

FOR

Enabling validation of FOR loops means that before each FOR statement is executed, checks are made to be sure that the increment value is nonzero and that the final value (if one is specified) can be reached exactly by applying the increment repeatedly to the initial value. If either check fails, an error message is printed and the program aborts. If FOR validation is disabled, either of these two errors can cause the program to loop indefinitely.

RECUR_UNDER, RECUR_OVER

RECUR_UNDER checks for stack underflow and RECUR_OVER for stack overflow during the execution of recursive routines.

LOOP, CALL

LOOP and CALL are the only parameters that cause output even if no error has been detected. When LOOP is enabled, each repetition of a LOOP, REPEAT, DO, FOR, or LINK statement produces the source line number, the statement's keyword, and the name and value of the control variable when appropriate.

When CALL is enabled, SKOL prints a line for each CALL, EXECUTE, SUSPEND, or ACTIVATE statement executed. The output is an image of the statement (or a partial image of the statement in the case of CALL), thus displaying the flow of control within the program.

CONDITIONAL COMPILATION

Conditional compilation allows the programmer to suppress specified sections of program code. SKOL offers two methods of conditional compilation: the first to specify a series of statements and the second to specify a single statement. Although it may serve many purposes, conditional compilation is intended primarily to assist in the testing of a program.

A SERIES OF CONDITIONAL STATEMENTS

Conditional code prefixes are SKOL identifiers chosen by the user. They determine whether or not a specified section of code is compiled. The code is specified in the following manner:

```
<<{NOT } prefix: code>>
```

prefix The conditional code prefix

code Any amount of SKOL source code

Each conditional code prefix must appear in a CONDITIONAL declaration. The declaration, which can appear anywhere in the program prior to its use, takes the following form:

CONDITIONAL: $prefix_1$ {,$prefix_2$ ... ,$prefix_n$};

$prefix_i$ A SKOL identifier

Prefixes can be enabled and disabled any number of times with a statement of the form:

{ENABLE / DISABLE} $prefix_1$ {,$prefix_2$ ... ,$prefix_n$};

$prefix_i$ A SKOL identifier previously declared as a conditional code prefix

The default is DISABLE. If an undeclared prefix is enabled or disabled, a warning message is generated. The scope of a CONDITIONAL, ENABLE, or DISABLE statement is local to the major segment in which it appears. If the statement appears outside a major segment, it is global to all the segments that follow it.

To illustrate, suppose E is a prefix currently enabled and D is a prefix currently disabled. The code in the following statements is then suppressed:

```
<<D: code>>
<<NOT E: code>>
```

The code in the following statements is compiled as if there were no enclosing brackets:

```
<<E: code>>
<<NOT D: code>>
```

Example:

```
SUBROUTINE SURVEY:
  CONDITIONAL: ABC;
  DISABLE ABC;
  <<ABC:
  RECORD SAMPLE(10):
    POINTER: NEXT;
    REAL: B, C(25, 30);
  ENDRECORD;
  >>
  POINTER TO SAMPLE: P, Q;

BEGIN:
  ENABLE ABC;
  <<ABC:
 ΛP.B = '''Herman''';
 ΛQ.C(I) = ΛP.B;
  >>
ENDSUBROUTINE;
```

In the example, the declaration for the record SAMPLE is not compiled because the prefix ABC is disabled immediately before the record declaration. The second section of conditional code is compiled but causes an error message, since it uses fields in the undeclared record. The pointer declaration lies outside of the conditional compilation sections and generates an error message because SAMPLE is not compiled.

This form of conditional compilation may also be useful in specifying code within a statement, but it is not recommended to suppress parts of statements. Some statements, such as TRACE and OUTPUT, have their argument lists processed piece by piece so that any brackets may be passed on to the generated FORTRAN code without being noticed.

Example:

```
If X<<ABC: - Y>>= 0:
```

STRUCTURED TEXT

A special use of the angle brackets to suppress code generation is described in Appendix D, Structured Text. Briefly, structured text is enclosed in angle brackets without a prefix, as in <<:...>>. Within the brackets, periods rather than semicolons are the statement terminators, and most error messages are suppressed.

A SINGLE CONDITIONAL STATEMENT

The DEBUG prefix allows conditional compilation of a single statement. When applied to an OUTPUT or TRACE statement, for instance, it permits the programmer to enable debugging output only when it is needed.

If DEBUG appears at the beginning of a statement, that statement is compiled only if DEBUG is currently enabled. It is ignored if DEBUG is currently disabled. Enabling or disabling DEBUG is accomplished by coding:

```
{ENABLE }
{DISABLE}  DEBUG;
```

DISABLE is the default. Enabling DEBUG generates a macro that discards any DEBUG prefixes and processes the prefixed statements normally. Disabling DEBUG, on the other hand, generates a macro that discards all statements prefixed with DEBUG. In this context, "statement" is defined as any string of source text that ends with a semicolon rather than a colon. A statement that ends with a colon can be followed by a semicolon when it is prefixed by DEBUG.

However, the <<...>> form is preferable for all single statements and groups of statements. This method retains the automatic indentation feature of SKOL, allows syntax errors to be reported, and allows distinctions to be made between different classes of debugging code.

Example:

```
DEBUG OUTPUT(A);
ENABLE DEBUG;
DEBUG OUTPUT(B);
DISABLE DEBUG;
DEBUG OUTPUT(C);
```

In the above example, the value of B is printed, and the values of A and C are not.

ASSERT

The ASSERT statement is a tool for debugging and program verification. Like DEBUG statements, ASSERT statements are conditionally compiled; they generate no code unless preceded by an ENABLE ASSERT statement. The format of the ASSERT statement is:

```
ASSERT lexp;
```

lexp A logical expression that is expected to be true whenever the ASSERT statement is executed

If an ASSERT statement is preceded by an ENABLE ASSERT statement, and no intervening DISABLE ASSERT is present, then the ASSERT statement is replaced by the following code:

```
IF NOT (lexp):
  IF FIRST $ASSERT_LIMIT TIMES:
    $ASSERT_ACTION('False assertion: lexp at line @SQ');
  ENDIF;
ENDIF;
```

DISABLE ASSERT is the default. The scope of a DISABLE or ENABLE statement is local to the major segment in which it appears. If the statement appears outside a major segment, it is global to all the segments that follow it.

The parentheses in

```
IF NOT (lexp):
```

allow SKOL to optimize the logical expression, using the DeMorgan laws of negation propagation. Each pair of apostrophes within the replacement string designates a single literal apostrophe. If the parameter (*lexp*) itself contains additional apostrophes, an error results because the first use of *lexp* in the replacement string requires single apostrophes, while the second use requires double apostrophes.

$ASSERT_LIMIT and $ASSERT_ACTION are defined in SKOLTXT with the following statements:

```
CONSTANT: $ASSERT_LIMIT = 3;
DEFINE ';$ASSERT_ACTION(' = ';OUTPUT(';
```

If these default definitions are still active when the ASSERT statement is expanded, the code becomes:

```
IF NOT (lexp):
  IF FIRST 3 TIMES:
    OUTPUT ('False assertion: lexp at line @SQ');
  ENDIF;
ENDIF;
```

Upon execution of this code, output occurs only when the *lexp* is false. No more than three lines of output can be produced during the program's entire run.

The presence of the $ASSERT_LIMIT test prevents runaway output when a false assertion is contained in a loop. The use of the intermediate $ASSERT_ACTION statement allows the programmer to direct the output to another file or even to call an error-handling subroutine, as the following definitions demonstrate:

```
DEFINE ';$ASSERT_ACTION(' = ';OUTPUT TO ERRFILE(';
DEFINE ';$ASSERT_ACTION(' = ';REMARK(';
DEFINE ';$ASSERT_ACTION(' = ';CALL REMARK2(';
DEFINE ';$ASSERT_ACTION(' = ';INCR ERRCOUNT;REMARK(';
DEFINE ';$ASSERT_ACTION(' = ';CALL ERRCHECK;OUTPUT(';
```

In the above examples, the left parentheses could be omitted from each pattern and replacement, but the opening semicolons could not. Were the semicolons omitted, the default definition would take precedence because its semicolon would allow an earlier match.

By including the text string as a parameter in the redefinition of $ASSERT_LIMIT, the message might be sent to two or more files at once:

```
DEFINE ';$ASSERT_ACTION(#);'=';OUTPUT(#1);REMARK(#1);';
```

REMARK is used here for brevity; CALL REMARK2 produces the same output to the $LOG file when the only argument is a literal string.

STRINGS

7

Variables containing character strings of varying length, but limited to a predetermined maximum length, can be created by means of the STRING declaration. The declaration takes the following form:

STRING: fid_1(*iconst*) {,fid_2(*iconst*) ... ,fid_n(*iconst*)} ;

fid_i A FORTRAN identifier

iconst An integer constant that defines the maximum length of the string

Example:

```
STRING: NAME (30), PHRASE (10);
```

The built-in function SIZE returns a string's maximum length, and the function LENGTH returns a string's current length. Together they can give the number of positions that remain vacant. Their forms are:

{SIZE / LENGTH}(*string*)

string An identifier used in a STRING declaration

LENGTH, which is represented by an internal SKOL variable that is unique for each string, can be used as a pseudo function on the left side of an assignment statement. It is implicitly reset by assignment statements that update the string. It needs to be set explicitly by the programmer only after the string has been modified by reading data into it.

An optional conversion between external codes and internal ordinal numbers is defined by the programmer's TYPE CHAR declaration (see appendix B). If the CHAR type is declared, special statements must be used for the input and output of character string data. Input is accomplished through the READSTRING command and output through WRITESTRING. The READSTRING and WRITESTRING commands are also described in appendix B.

A string and its length variable can be put into common storage by naming the string in a COMMON statement.

Examples:

```
COMMON PHRASE;
COMMON/TOKENS/NAME,PHRASE;
```

DELETE

Before using a string it should be initialized to the empty string by either the DELETE command or the assignment statement, which is described later in this section. The DELETE command takes the form:

```
DELETE {string;    }
       {substring; }
```

string — An identifier used in a declaration

substring — One of three special forms denoting a part of a string:

1. *string* (*index*$_1$..*index*$_2$)
2. *string* (*index*..|*length*)
3. *string* (*length*|..*index*)

string — An identifier used in a string declaration

index — An integer within the bounds of the maximum length of the string

index$_1$,*index*$_2$ — Integers within the bounds of the maximum length of the string such that *index*$_2$ is greater than *index*$_1$

length — A positive integer less than or equal to the maximum length of the string

Form 1 denotes the substring consisting of all characters between and including the two limits. If only one index is present, the limits are equal, and the substring consists of a single character.

Form 2 indicates a substring that begins with an element identified by the index and contains the number of characters indicated by the length.

In form 3, the index identifies the last character in the substring, and the length gives the number of characters in the substring.

Examples:

```
NAME(2..K+2)                  -- Refers to the second character
                              --   through character K+2.
PHRASE(4|..LENGTH(PHRASE))    -- LENGTH(PHRASE)-3 through
                              --   LENGTH(PHRASE).
NAME(1..|5)                   -- Characters 1 through 5.
PHRASE(5)                     -- Character 5.
```

REPLACE

The REPLACE statement substitutes the second string operand for the first. It is of the form:

```
REPLACE {substring} BY {NULL      };
        {string    }   {char-exp  }
                       {substring}
```

string	An identifier used in a string declaration
substring	A designated part of a string
char-exp	A single literal character or a variable containing a single character

To REPLACE a string by NULL is the same as to DELETE the string.

Examples:

```
REPLACE PHRASE(2..4) BY 'Z';
REPLACE NAME BY PHRASE(2);
```

INSERT

INSERT uses a single position within a string as a reference and adds another string or substring before or after it. The form is as follows:

```
INSERT {char-exp  } {BEFORE} string (index);
       {substring} {AFTER  }
```

string	An identifier used in a string declaration
index	An integer within the bounds of the maximum length of the associated string
substring	A designated part of a string
char-exp	A single literal character or a variable containing a single character

Examples:

```
INSERT NAME(1..3) BEFORE PHRASE(2);
INSERT CH AFTER PHRASE(LENGTH(PHRASE)-3);
```

SPECIAL NOTATION

The general substring notation described under DELETE allows the special case of a zero length, making it possible for the REPLACE statement to perform insertions. In most cases, this method is no better (and is less readable) than using the INSERT statement. However, the REPLACE statement can be used to define a single, all-powerful replacement subroutine that performs replacements, deletions, and insertions. (SKOL transforms nontrivial DELETE statements and all INSERT statements into REPLACE statements. Each REPLACE operation is actually a call to a library routine, which aborts the calling program if asked to expand a string beyond its maximum length.)

A length of zero indicates that an operation is to take place outside of the accompanying index. For instance,

```
PHRASE(2..|0)
```

denotes the position before the second character of PHRASE. Similarly,

```
PHRASE(1..|0)
```

indicates the position before the first character of PHRASE and

```
PHRASE(0|..LENGTH(PHRASE))
```

the position after the last. The last two are both outside the bounds of the string called PHRASE.

Example:

```
REPLACE PHRASE(1..|0) BY NAME (3|..K);
```

Effectively this statement inserts three characters, NAME(K-2), NAME(K-1), and NAME(K), before the first character of PHRASE.

CATENATE

For the sake of efficiency, a string concatenation statement is implemented separately from other string manipulation statements. The form of the statement is:

```
CATENATE string-exp1 {& string-exp2 ... & string-expn}
         ONTO string;
```

string	An identifier used in a string declaration
string-exp$_i$	A string expression that can be either a literal string, a named string, a substring, or an expression that resolves into a single character

When the catenation operation is complete, the contents of the string named after the keyword ONTO remain unchanged but are followed by the values of the string expressions, which are appended from left to right in the order listed. The length of the string is updated.

Examples:

```
CATENATE NAME(2..4) ONTO PHRASE;
CATENATE CH & NAME(1..|2) ONTO PHRASE;
```

In the second example, the first element is a single-character expression, and the second is a substring.

STRING ASSIGNMENT STATEMENT

Values may be assigned to a string with an assignment statement of the form:

string = *string-exp*$_1$ {& *string-exp*$_2$ ... & *string-exp*$_n$};

string	An identifier used in a string declaration
string-exp$_i$	A string expression that can be either a literal string, a named string, a substring, or an expression that resolves into a single character

Example:

```
NAME = 'J' & BLANK & FAMILY(1..LENGTH(FAMILY));
NOTHING = '';        -- The same as DELETE NOTHING.
```

MOVE

MOVE performs both a REPLACE and a DELETE. It replaces the second string with the first string and deletes the first string. It takes the form:

```
MOVE {string    } TO {string    } ;
     {substring}     {substring}
```

string An identifier used in a string declaration

substring A designated part of a string

Example:

```
MOVE OUTLINE(2..10) TO S(0|..1);
```

The statement in the example moves the second through tenth characters of OUTLINE to the nine positions following S(1) and deletes OUTLINE(2..10).

NOTE

Literal character strings containing more than one character cannot be used in MOVE, REPLACE, DELETE, or INSERT statements.

STRING COMPARISON EXTENSION

SKOL can be extended to perform other useful operations on strings: for example, a test of two strings of variable length to determine if they contain the same characters. Although such an operation is not built into the language, it can be added by means of a macro definition (DEFINE statement) and a supporting logical function subprogram. The desired end result is a notation that may be embedded in a logical expression, such as the following:

```
EQUAL (STR1, STR2)
```

A substitution for EQUAL must be previously defined.

```
DEFINE ' EQUAL(#,#)' = 'EQU999(#1, SIZE(#1), LENGTH(#1), #2,
                        SIZE(#2), LENGTH(#2))';
```

Any occurrence of EQUAL(STR1, STR2) will then invoke the following function:

```
FUNCTION EQU999: LOGICAL (STR1, S1, L1, STR2, S2, L2):
   -- Test equality of two strings.
   INTEGER: S1, S2, L1, L2, I;
   CHAR: STR1(S1), STR2(S2);
BEGIN:
   UNTIL ALL_SAME OR MISMATCH:
      IF L1<>L2: MISMATCH; ENDIF;
      FOR I = 1 to L1:
         IF STR1(I) <> STR2(I): MISMATCH; ENDIF;
      ENDFOR;
      ALL_SAME;
   THEN:
      [ALL_SAME]: EQU999 = TRUE;
      [MISMATCH]: EQU999 = FALSE;
   ENDUNTIL;
ENDFUNCTION;
```

Each program module in which EQUAL is used must include

```
EXTERNAL LOGICAL: EQU999;
```

among its declarations.

PROGRAM MODULES 8

A SKOL program is made up of one or more modules, including one main program segment and any number of FORTRAN-style subprograms (all of which are called major segments in this publication).

The major segments called subroutines, functions, and block data modules are carried over from FORTRAN virtually intact. Each assumes the same form as the main segment, with a section of local declarations, a main body of executable statements, and optional local routines of its own.

In FORTRAN, all statements must lie within a main program or subprogram. That rule applies to most statements in SKOL, the exceptions being those macro statements that generate other macros rather than FORTRAN code. The keywords of the exceptional statements are:

CONDITIONAL
CONSTANT
DEBUG
DEFINE
DISABLE
ENABLE
MACRO
NOTRACE
TRACE
TYPE
VALIDATE

Such statements can be made global to a set of major segments by placing them outside of and before those segments. For instance, if a scalar type must be available to all parts of a program, the TYPE statement should appear at the top of the program text.

On the other hand, such statements become local to a given major segment when they appear inside it. For example, if a SKOL identifier is used where a FORTRAN identifier is called for, but only within a particular subroutine, then a CONSTANT statement specifying the desired replacement should be placed soon after the SUBROUTINE statement. If, however, the SKOL identifier appears in the subroutine's list of dummy arguments, the CONSTANT statement must precede the SUBROUTINE statement so that the replacement macro becomes global and affects all subsequent text.

Within each major segment, variable declarations and external subprogram declarations must precede all executable statements. The keyword BEGIN, followed by a colon, separates the declarations from the code that follows.

The optional local routines must appear at the end of a major segment. They can be simple routines without parameters and local variables, recursive routines with parameters and local variables, or coroutines belonging to a process.

Routines must be arranged in top-down order, because SKOL requires the definition of a routine to follow all references to it. To be consistent with routines, the major segments should also be arranged in top-down order, with the MAIN segment first.

THE MAIN SEGMENT

The MAIN segment takes the following form:

```
MAIN {(fid)}:
  dec
BEGIN:
  block
ENDMAIN;
```

fid An optional FORTRAN identifier naming the main segment

dec A series of declarations

block A sequence of executable statements

The MAIN segment is translated into a subroutine by the SKOL macro translator. If the optional FORTRAN identifier is missing, the default name is MAINSUB. The macro translation for a MAIN statement such as

```
MAIN (X):
```

is as follows:

```
CALL X
END
SUBROUTINE X
```

The dummy subroutine is created so that the RETURN statements generated by ROUTINE and COROUTINE headers (described below) are legal in FORTRAN.

SUBROUTINES

Subroutines can be defined anywhere outside the MAIN segment of a SKOL program. A subroutine segment has the following general form:

```
SUBROUTINE fid{(darg1 {,darg2 ... ,dargn})} :
  dec
BEGIN:
  block
ENDSUBROUTINE;
```

fid A FORTRAN identifier giving the name of the subroutine

$darg_i$ A dummy argument representing a variable name, an array name, an external segment name, or an asterisk associated with an alternate return specifier (see the RETURN statement, described later in this section)

dec A series of declarations

block A sequence of executable statements

As in FORTRAN, a subroutine is invoked by the CALL statement, which takes the same form in SKOL as in FORTRAN.

```
CALL fid{(arg1 {,arg2 ... ,argn})};
```

fid A FORTRAN identifier giving the name of the subroutine

arg_i An actual argument corresponding to a dummy argument in the SUBROUTINE statement and taking the form of an expression, an array name, an external segment name, or an alternate return specifier of the form * = *situation*

FUNCTIONS

As with subroutines, functions can be defined anywhere outside of the MAIN segment of a SKOL program. A function segment has the following general form:

```
FUNCTION fid: type (darg1 {,darg2 ... ,dargn} ):
  dec
BEGIN:
  block
ENDFUNCTION;
```

fid	A FORTRAN identifier giving the name of the function
type	The data type to be associated with the function name
$darg_i$	A dummy argument representing a variable name, an array name, or an external procedure name
dec	A series of declarations
block	A sequence of executable statements

SKOL functions are invoked in the same manner as FORTRAN functions. Statement functions can be defined in the same way they are defined in FORTRAN. When the function name appears in an executable statement, the function body is executed, causing a value to be assigned to the function name. The type of the result of a statement function or external function reference is the same as the type of the function name, which must appear in a declaration statement at the beginning of the major segment. External functions must also be declared EXTERNAL. Intrinsic functions need not be declared. The type of a function can be any standard FORTRAN type or a SKOL scalar type but not a subtype or any other SKOL type such as a STRING or POINTER.

ENTRY STATEMENT

An ENTRY statement may be used in a subroutine or a function. Just as in FORTRAN, the statement creates an additional entry point (with its own dummy arguments) to a subroutine or function. In a subroutine, an ENTRY statement's dummy arguments can be alternate return specifiers (* = *situation*) just as in the CALL statement. The ENTRY statement establishes an external name. When that name is called from another program segment, execution begins at the first statement following the ENTRY statement.

Two restrictions distinguish the SKOL ENTRY statement from its FORTRAN counterpart:

1. Execution of the subroutine or function as invoked by its primary name ends just before the ENTRY statement. A RETURN statement is inserted by the SKOL translator before the ENTRY statement if one is not already present. Thus, there is no overlap of instructions between the subroutine or function executed through the primary entrance and the subroutine or function executed through the ENTRY statement entrance. The two might be thought of as separate segments except that they share the declarations under the primary entrance and can share routines and processes, as the following example demonstrates.

   Example:

```
SUBROUTINE MAINENT:
  :
  EXECUTE R;
ENTRY SECENT:
  :
  EXECUTE R;
  ROUTINE R:
    :
  ENDROUTINE;
ENDSUBROUTINE;
```

2. The ENTRY statement must be at the same indentation level as the SUBROUTINE or FUNCTION statement that begins the segment. Indentation is described in section 10 of this publication.

The ENTRY statement takes the following form:

ENTRY *fid* {(*darg*$_1$ {,*darg*$_2$ ... ,*darg*$_n$})} :

fid A FORTRAN identifier

darg$_i$ A dummy argument corresponding to an actual argument in the CALL statement

RETURN STATEMENT

As in FORTRAN, a RETURN statement causes return of control to the calling segment. It may appear in any major segment. When encountered in a MAIN segment, it causes the normal termination of the job step. The form of a RETURN statement is:

```
RETURN {iexp};
```

iexp In a subroutine, an integer expression selecting an alternate return

Only the simple form RETURN, followed by a semicolon, is allowed in a MAIN or FUNCTION segment. In a SUBROUTINE segment with one or more asterisks (alternate return specifiers) as dummy arguments, the RETURN statement can alter the normal direction of the program flow. The value of *iexp*, when it is present, references the *iexp*th asterisk in the dummy argument list of a SUBROUTINE or ENTRY statement. The *iexp*th asterisk, in turn, references the corresponding actual argument (of the form * = *situation*) in the CALL statement. Thus the UNTIL block enclosing the CALL is terminated and control transfers to its end, where there may be a THEN block containing a situation label.

Example:

```
UNTIL ZERO OR ERROR:
  :
  CALL ANALYZE (A, B, * = ZERO, * = ERROR);
  :
THEN
  [ZERO]: ...
  [ERROR]: ...
ENDUNTIL;

SUBROUTINE ANALYZE (X1, X2, *, *):
  REAL: X1(1), X2(1);
  :
  RETURN 1;        -- Signal ZERO
  :
  RETURN 2;        -- Signal ERROR
```

BLOCKDATA SEGMENTS

The SKOL BLOCKDATA segment is like the BLOCKDATA subprogram of FORTRAN with the addition of the standard SKOL syntax rules. It takes the form:

```
BLOCKDATA {fid}:
  dec
ENDBLOCKDATA;
```

fid The optional FORTRAN identifier giving the name of the BLOCKDATA segment

dec A sequence of declarations and DATA statements

ROUTINES

Simple routines without parameters can be defined and executed within a major program segment. All variables declared in the major segment are global to such routines. Routines are defined at the end of a major segment.

Programmers may often prefer routines to subroutines, because routines allow the sharing of all variables and encourage top-down, modular programming. SKOL requires the definition of a routine to follow all references to it, thus preventing accidental recursion and enforcing a top-down arrangement of routines. The form of a routine is:

```
ROUTINE r-id:
  block
ENDROUTINE;
```

r-id A SKOL identifier naming the routine

block A sequence of executable statements

A routine must be invoked within the major segment that contains it; it cannot remain idle. If a routine is defined but not invoked, or if it is invoked from a point after its definition, SKOL issues an error message. The form of an invocation is:

```
EXECUTE r-id;
```

r-id A SKOL identifier naming the routine

Example:

```
EXECUTE IN_CARD;
:
ROUTINE IN_CARD:
  :
ENDROUTINE;
```

A routine can also call, by means of an EXECUTE statement, another routine that is defined after the calling routine in the same major segment.

SKOL inserts a RETURN statement before the first definition of a routine or coroutine if the user has not done so already. RETURN prevents execution of the major segment from falling through into the routine or coroutine. The RETURN statement, if found in a subroutine or function, returns control to the segment from which that subroutine or function was called. When encountered in a MAIN segment, RETURN terminates the program. RETURN is illegal in FORTRAN main programs. It is legal in SKOL only because SKOL generates a dummy main program and puts all the main segment's code into that subroutine.

PROCESSES AND COROUTINES

A process consists of a set of coroutines, each of which can activate another coroutine or suspend the process. When a process is suspended, control goes back to the statement following the one from which it was initially activated. A process must be declared and its coroutines defined and activated within a single major segment. The process declaration takes the form:

PROCESS *p-id* = (*co-id*$_1$ {,*co-id*$_2$ ... ,*co-id*$_n$});

p-id A SKOL identifier naming the process

co-id$_i$ A SKOL identifier naming a coroutine that is part of the process and is defined at the end of the major segment

Before a process can be activated, it must be initialized with a START statement. The execution of the START statement identifies the first coroutine to be activated and ensures that the remainder of the coroutines are inactive at the beginning. The START declaration takes the form:

```
START p-id AT co-id;
```

p-id A SKOL identifier naming the process

co-id A SKOL identifier of the first coroutine to be activated

The process is activated by the following statement in the major program segment:

```
ACTIVATE p-id;
```

p-id A SKOL identifier naming the process

When a process is activated, control transfers to the process and remains there until the process is suspended by a statement of the following form in one of its coroutines:

```
SUSPEND p-id;
```

p-id A SKOL identifier naming the process

A coroutine can also execute an ACTIVATE statement to suspend itself in favor of another coroutine belonging to the same process. This form of the ACTIVATE statement has as its argument the name of a coroutine rather than a process:

```
ACTIVATE co-id;
```

co-id A SKOL identifier naming the coroutine to be activated

When a coroutine is activated after having suspended itself, control passes to the statement following the last ACTIVATE statement that the coroutine executed. The first activation of a coroutine transfers control to its first statement.

Coroutines, like routines, are defined at the end of a major program segment. The coroutine definition is of the form:

```
COROUTINE co-id:
  block
ENDCOROUTINE;
```

co-id A SKOL identifier naming the coroutine

block A sequence of executable statements

<u>Example:</u>

```
  PROCESS LIFE = (PRODUCE, CONSUME);
BEGIN:
  START LIFE AT PRODUCE;
  ACTIVATE LIFE;
  :
  COROUTINE PRODUCE:
    :
    ACTIVATE CONSUME;
    :
  ENDCOROUTINE;

  COROUTINE CONSUME:
    :
    ACTIVATE PRODUCE;
    :
    SUSPEND LIFE;
    :
  ENDCOROUTINE;
```

When the ACTIVATE LIFE statement is encountered for the first time, control is transferred to the beginning of the coroutine named in the START statement. In the above example, PRODUCE is activated first. If the same or another ACTIVATE LIFE is subsequently executed, control is passed back to the point at which the last SUSPEND LIFE was executed, unless a START statement has been executed more recently.

Some widely used FORTRAN compilers, for instance WATFIV, do not support implementation of the extended DO and thus do not permit an ACTIVATE or a SUSPEND command within a DO-loop. Changing the DO statement to the less efficient FOR statement solves the problem. The CFT compiler does support the extended DO.

If execution of a coroutine reaches the ENDCOROUTINE statement, the program aborts with an error message.

SKOL inserts a RETURN statement before the first definition of a coroutine or routine if the user has not done so already. RETURN prevents execution of the major segment from falling through into the coroutine or routine, instead returning control to the calling segment. When encountered in the MAIN segment, RETURN terminates the program. RETURN is illegal in FORTRAN main programs. It is legal in SKOL's MAIN segment only because SKOL generates a dummy main program and puts all of the MAIN segment's code into a subroutine.

Coroutines have no parameters. All variables are declared in the enclosing major segment and are global to the process.

Thorough commenting makes the flow of control within a process readable. The statement

```
BEGIN:
```

aids readability in routines and coroutines when inserted between comments about entry/exit conditions and executable statements. The SKOL macro translator replaces BEGIN: with CONTINUE.

RECURSIVE ROUTINES

Routines that invoke themselves recursively can be defined in SKOL. They can have integer call-by-value parameters (whose values are passed into the routine at the beginning of recursive execution but are not passed back out at the end) and local integer variables. Recursive routines are predeclared with a statement of the form:

```
RECURSIVE (iconst): r-id1(*{,* ... ,*}) {,r-id2
     (*{,* ... ,*}) ... ,r-idn(*{,* ... ,*})} ;
```

iconst An integer constant giving the maximum size of the recursion stack

r-id$_i$ A SKOL identifier naming the routine

The size of the stack, specified by the integer constant, limits the number of recursive calls. Each call requires one word on the stack for each parameter and one word for each local variable that must be saved. If the limit is exceeded and the RECUR_OVER option has been enabled either explicitly in a VALIDATE statement or through default, an error message is generated.

Each * in the RECURSIVE declaration saves a place for one parameter in the corresponding routine definition. As with simple routines and coroutines, recursive routines are defined at the end of a major program segment. The ROUTINE definition statement includes the parameters and any local variables following the LOCAL keyword. The recursive routine takes the following form:

```
ROUTINE r-id (par-id1 {,par-id2 ... ,par-idn})
  {LOCAL (loc-id1 {,loc-id2 ... ,loc-idn})}:
       block
ENDROUTINE;
```

r-id A SKOL identifier naming the routine

par-id$_i$ A SKOL identifier naming a parameter

loc-id$_i$ A SKOL identifier naming a local variable

block A sequence of executable statements

A recursive routine is invoked by an EXECUTE statement of the form:

```
EXECUTE r-id (exp1 {,exp2 ... ,expn});
```

r-id A SKOL identifier naming the routine

exp_i An integer expression specifying a parameter

<u>Example:</u>

```
RECURSIVE (100): TREE (*), WHAT (*,*), P;
:
EXECUTE WHAT (2-J, CH);
:
ROUTINE WHAT (BRANCH_MOVE, PIECE_TYPE) LOCAL (LSON, RSON):
  :
ENDROUTINE;
```

Three potentially recursive routines are introduced in the RECURSIVE statement shown above. TREE has one parameter, WHAT has two, and P has none. A stack of 100 words is used to implement each of the routines. When the EXECUTE statement transfers control to the WHAT routine, the two parameter expressions, 2-J and CH, are evaluated, and their values are assigned to BRANCH_MOVE and PIECE_TYPE respectively. (This mode of parameter communication is known as call by value.) A change in a formal parameter, for instance PIECE_TYPE, during the execution of the recursive routine does not change the value of CH after execution is complete. CH keeps the same value coming out of WHAT that it had going in. The two variables declared as locals, LSON and RSON, have unique values for each recursive call; they receive values inside the routine.

Recursive routines in SKOL are subject to a significant restriction. Unpredictable behavior results when a recursive call is made from within a FOR or DO loop that has different increment values or different final values on different levels of recursion. If a parameter value, for instance, is used to mark a loop's increment or final value, unexpected behavior results, as in both of the following examples.

Examples:

```
ROUTINE TREE (J) LOCAL (A, B, SEARCH):
   DO SEARCH = 1 TO J-1:
      :
      EXECUTE TREE(NEW_FAMILY);
      :
   ENDDO;
ENDROUTINE:
```

```
ROUTINE TREE (J) LOCAL (A, B, SEARCH):
   FOR SEARCH = 1 TO TOP BY J:
      :
      EXECUTE TREE(NEW_FAMILY);
      :
   ENDFOR;
ENDROUTINE;
```

A safe way to perform the same routine is to use the LOOP ... WHILE structure, making the control, increment, and limit variables local to the routine, as shown in the following example.

Example:

```
ROUTINE TREE (J) LOCAL (A, B, C):
   LOOP:
      :
   WHILE B<>A:
      :
      A = A + C;
   ENDLOOP;
ENDROUTINE;
```

To make recursive calls safely from within the situation case structure, a special form of the UNTIL statement is necessary. A single-situation UNTIL statement of the following form must be used.

```
UNTIL GLOBAL sid:

  block

ENDUNTIL;
```

sid A SKOL identifier corresponding to a situation name somewhere in the *block*

block A sequence of statements containing a call to a recursive routine outside of the situation case structure

The presence of the word GLOBAL causes all situation terminations with the indicated name to reset the recursion stack to its status at the most recent entry to the UNTIL block.

Example:

```
ROUTINE TEST (I):
   :
   UNTIL GLOBAL CHECK:
      IF I = J: CHECK;
      ELSE: EXECUTE TEST (I+1);
      ENDIF;
   ENDUNTIL;
   :
ENDROUTINE;
```

LOCAL POINTER DECLARATIONS

Parameters and local variables of recursive routines must be integers, as noted previously. However, since the SKOL pointer type becomes an integer in the generated FORTRAN code, pointers to tables or records can be used as routine parameters and local variables. Any pointer whose name appears in a ROUTINE statement must be declared as a LOCAL POINTER rather than an ordinary pointer. The forms of a local pointer declaration are as follows:

```
LOCAL POINTER TO rtype: sid_i {(size)} {,sid_2
                        {(size)} ... ,sid_n {(size)}} ;

LOCAL POINTER TO TABLE tn: sid_i {(size)} {,sid_2
                           {(size)} ... ,sid_n {(size)}} ;
```

rtype A record type identifier

tn An integer between 1 and 63 inclusive that identifies the table or a SKOL identifier defined as such an integer

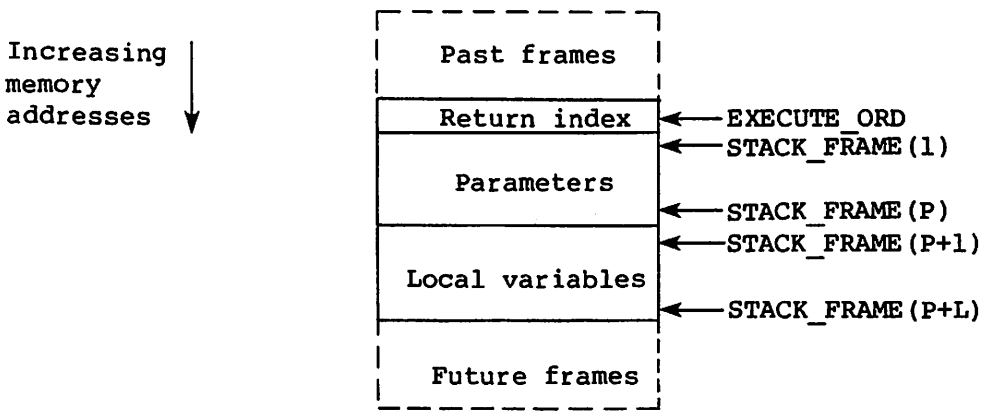
sid_i A SKOL identifier naming the pointer

size An integer constant specifying, if present, the number of pointers in an array; only one dimension is allowed.

For efficiency, avoid the use of local pointers. Instead, move parameters and local variables from the recursion stack into ordinary pointer variables before using them.

ACCESSING THE RECURSION STACK

The SKOL identifiers EXECUTE_ORD and STACK_FRAME are reserved for use in referring to the recursion stack. Only the current frame of the recursion stack, containing the parameters and local variables that belong to the current recursion depth, is accessible. The stack frame looks like this:



EXECUTE_ORD must be treated as a simple variable, while STACK_FRAME must be treated as an array. EXECUTE_ORD is equivalent to STACK_FRAME(0). The first cell in the stack frame, STACK_FRAME(0), holds an index value that allows the recursive routine to return to the proper caller. The index value corresponds to the physical order of the EXECUTE statements in the program.

INPUT/OUTPUT 9

SKOL reads and writes data in a manner similar to FORTRAN. The READ, WRITE, DECODE, ENCODE, and FORMAT statements in SKOL are nearly identical to the corresponding FORTRAN statements. However, the more convenient INPUT and OUTPUT statements allow the programmer to specify both the variable names and the format items together in their natural order in a single statement. The INPUT and OUTPUT statements permit the use of all the FORTRAN editing symbols, but keywords are used for carriage control instead of 1-character literals.

Two additional statements, LOG and REMARK, provide for output to the job's log file. The log file reports which of the job's control statements were executed and contains messages issued by the operating system as well as by user programs. In form, the LOG statement is analogous to WRITE, and REMARK is analogous to OUTPUT.

OUTPUT

SKOL permits one or more items in an OUTPUT statement. When more than one is present, the items are separated by commas. The general form of the OUTPUT statement is as follows:

```
OUTPUT {TO file} (item1 {,item2 ... ,itemn});
```

file A local dataset name, a FORTRAN unit number, or an expression that resolves into one or the other. If the phrase "TO *file*" is omitted, $OUT is assumed.

$item_i$ An output item

Each output item can have any of the following forms:

```
'string'
var:fmt
var::fmt
var
:fmt
$cc
```

string Any literal string contained within apostrophes

var Any variable or array element that has been assigned a value; it can also be an array name or a FOR clause (described below in this section)

fmt An optional FORTRAN editing format

$cc One of the carriage control keywords listed in table 9-1.

Table 9-1. Carriage control specifiers

KEYWORD	SUBSTITUTED TEXT At start of list	Within list	MEANING
$PAGE	'1'	/'1'	Page eject
$SKIP	'0'	/'0'	Skip one line, leaving a blank line
$SKIP2	/'0'	//'0'	Skip two lines
$SKIP3	//'0'	///'0'	Skip three lines
$OVER	'+'	/'+'	Overwrite the previous line; the availability of this option to SKOL users depends on their computer's printer software.
$LINE	' '	/' '	Advance to next line

If no carriage control keyword is present at the beginning of an OUTPUT list, and the first item does not begin with a colon that introduces X, T, or / editing, $LINE is assumed by default.

Apostrophes ('), as opposed to asterisks, quotation marks ("), or FORTRAN's H editing symbol, must be used to specify all literal strings. SKOL requires literals to be enclosed in apostrophes because of the special treatment it gives such strings (for instance, all blanks are retained and no pattern matching occurs between pairs of apostrophes). Expressions to be printed cannot contain apostrophes; for example,

```
OUTPUT (ALPHA('B'));
```

is illegal.

A colon precedes all format specifications. All FORTRAN editing descriptors, as described in the CRAY-1 FORTRAN (CFT) Reference Manual, are valid SKOL format items. The FORTRAN colon edit descriptor must be represented by a pair of periods. A comma terminates the format item unless the comma is enclosed in parentheses, as in

```
IARRAY:10(I5,2X)
```

If necessary, a format item can be broken up into two items, as in

```
IARRAY:9(I5,2X), :I5
```

<u>Example:</u>

```
OUTPUT (P:I3,
    ∧P.ID_NUMBER:I7,
    ∧P.PAY_PER_WEEK:G12.5,
    ∧P.AGE:I3);
```

A few lines of output using the format of the example appear as follows:

```
100  12349  115.38     36
 99  11002  38.477    102
 98    457  1.9231      9
```

As in FORTRAN, the *n*X descriptor creates *n* blank spaces, and the T*n* descriptor puts the next character in the *n*th position of the output record. The slash (/) moves the print carriage to the first position of the next line. A colon must precede each of these edit descriptors when they stand alone. The limit of a formatted line is 133 characters, including the carriage control character.

A special edit descriptor, SF*n*, is provided by SKOL. Coding *var*:SF*n* is equivalent to coding RNB(*var*:A*n*). The external function RNB replaces trailing nulls with blanks, producing a more readable output. The letters SF should be read as space fill.

If a colon but no format follows a variable, the default format (O22) is used. The internal representation of the value is converted to a 22-digit octal number.

Example:

```
OUTPUT(:2X, 'INDEX ', J:, :/2X, 'VALUE ', A(J):);
```

The presence in the list of an initial colon (as in :2X) suppresses the carriage control character (' ') that is normally inserted by default. An initial :T1 specification would also suppress the carriage control character without indenting the output. This example prints the following lines, assuming J is 5 and A(J) is 150_8.

```
INDEX 0000000000000000000005
VALUE 0000000000000000000150
```

In SKOL, it is easy to print a variable's name or an expression's representation along with its value. Such output is specified by replacing the single colon between the variable name or expression and the format with a double colon. The following will be printed: two blanks, the variable name or expression, one blank, an equal sign, another blank, the value of the variable or expression, and a semicolon. The value is printed in whatever format follows the double colon. If no format is present, $GENFORM (O22) is used by default. If a list item consists only of a variable name or expression, it is treated as if it were suffixed by ::$GENFORM. If the variable name or expression is of type real (as in the third example below), the O22 format gives the internal floating-point representation of the word in octal.

Examples:

```
OUTPUT(LENGTH(OUTLIN)::I3);
OUTPUT(LENGTH(OUTLIN));
OUTPUT(3.14159);
OUTPUT(22./7.0, 22./7.0::F7.5);
```

These statements cause the following output, assuming the current length of OUTLIN is 121:

```
LENGTH(OUTLIN) = 121;
LENGTH(OUTLIN) = 0000000000000000000171;
3.14159 = 0400026220771740156063;
22./7.0 = 0400026222222222222222;  22./7.0 = 3.14286;
```

Special treatment is given to ASCII debugging output. If ::An, ::Rn, or ::SFn is specified, not only is the variable name printed along with its ASCII value, but the value is enclosed in apostrophes for readability.

Mixing several of the output forms in a single OUTPUT statement permits the programmer to create combinations such as the following:

Example:

```
OUTPUT ($PAGE, :3X, J:, A(J)::G12.5, :5X, CH::R1, '$');
```

This statement produces the following output on the first line of a new page, assuming J is 5, A(J) is 142.36, and CH is P.

```
00000000000000000000005  A(J) =  142.36    ;       CH = 'P';$
```

INPUT

Although any of the list-item forms available with the OUTPUT statement can also be used in an INPUT statement, some of them are not necessarily helpful. There is no reason, for example, to use the double colon with input. Carriage control is likewise impractical. The input list should never begin with a *$cc* item; $LINE is not inserted by default.

The general form of the INPUT statement is as follows:

INPUT {FROM *file*} (*item*$_1$ {,*item*$_2$ ... ,*item*$_n$});

file A local dataset name, a FORTRAN unit number, or an expression that resolves into one or the other. If the phrase "FROM *file*" is omitted, $IN is assumed.

item$_i$ An input item

The form of each input item is:

item$_i$ {*var*:*fmt* / :*fmt*}

var The variable, array, or array element to be read; a FOR clause is also valid.

fmt An optional FORTRAN editing format

Example:

```
INPUT (:5X, J:I3, A(J):F10.5, :X, WHAT:L1);
```

INPUT statements read by default from the local dataset equated to the constant $IN, and OUTPUT statements write by default to the local dataset equated to the constant $OUT. The standard SKOL macros define $IN as unit 5 and $OUT as unit 6. Then at run time the CRAY-1 Operating System associates unit 5 with the standard input file, '$IN', and unit 6 with the standard output file, '$OUT'. In addition, the standard SKOL macros define $PUNCH as unit 7 and $TEMP as unit 8. Programmers may redefine any of these in the program text by means of text substitution, such as through the use of a CONSTANT declaration.

The current definition of $GENFORM governs the default format of the OUTPUT statement. FORTRAN format O22 (which prints in octal) is the current value of $GENFORM, but it may be changed by the programmer's own CONSTANT declaration.

Example:

```
CONSTANT: $IN = PARAM1, $OUT = 'SYSOUT', $GENFORM = G12.3;
```

REMARK

The REMARK statement sends a single output record to the job's log file. The form of the REMARK statement is like that of the OUTPUT statement, except that no carriage control items or :/ specifications are permitted, and an initial $LINE item is not assumed by default. The form is as follows:

REMARK ($item_1$ {,$item_2$ ... ,$item_n$});

$item_i$ An output item containing any of the items legal in an OUTPUT statement except carriage control items

The first character specified in the list of items goes into column one of the output record. If the line begins with the standard COS message prefix, which is:

xxxxxb-b

x Any character

b A blank

then that prefix begins the line. If the above is not specified as the first item, then

FT060*b*-*b*

is supplied by default. The resulting line, including the prefix, cannot contain more that 80 characters.

Example:

```
REMARK('MS001 - ', :10X, ERRCNT: I4, ' ERRORS');
```

One REMARK statement writes one output line to the log file. Two REMARK statements are necessary to write two lines.

FORMAT

When a specific form of an INPUT, OUTPUT, or REMARK statement is used several times in a program, it may prove expedient to establish that form in a FORMAT statement and then refer to it in READ, WRITE, or LOG statements. Any subsequent reference to the identifier assigned to the FORMAT statement supplies the desired format and removes the necessity of coding it explicitly each time. The FORMAT statement is of the form:

FORMAT *sid* = (*list*);

sid A SKOL identifier

list A series of format items, separated by commas, using forms specified by the host FORTRAN compiler

The FORTRAN colon edit descriptor must be represented by a pair of periods.

READ, WRITE

After the FORMAT has been defined, it can be used in READ and WRITE statements, which take the following form:

$$\left\{\begin{matrix}\text{READ}\\ \text{WRITE}\end{matrix}\right\} \quad (file\ \{,format\})\ var_1\ \{,var_2\ \ldots\ ,var_n\};$$

file A local dataset name such as '$IN' or '$OUT', a FORTRAN unit number, or an expression that resolves into either a FORTRAN unit number or a local dataset name

format The optional SKOL identifier for a previously defined FORMAT statement

var_i A variable name, an array name, an array element, a FOR clause, or (in the case of WRITE) an expression

If the optional *format* identifier is not included, unformatted I/O is performed.

Example:

```
FORMAT PERSON = (1X, 2I5, F10.2);
READ ($MYFILE, PERSON) Λ WORKER.AGE, J, Λ WORKER.PAY;
FORMAT PRETTY = ('1', 1X, I3, ')', F10.5, /, ' ');
WRITE ($OUT, PRETTY) K, B(K);
```

DECODE, ENCODE

As in FORTRAN, DECODE is like a formatted READ and ENCODE like a formatted WRITE, with an array in memory taking the place of an input/output device. The general syntax is:

$$\left\{\begin{matrix}\text{DECODE}\\ \text{ENCODE}\end{matrix}\right\} \quad (iexp,\ format,\ fid)\ var_1\ \{,var_2\ \ldots\ ,var_n\};$$

iexp An integer expression specifying the number of characters (not exceeding 152) to be decoded from, or encoded to, the packed array

format The SKOL identifier for a previously defined FORMAT statement

fid A FORTRAN identifier naming the first word in the memory array that contains (or is to receive) data in printable form, packed in A8 format

var_i A variable name, an array name, an array element, a FOR clause, or (in the case of ENCODE) an expression

LOG

The LOG statement, like WRITE, requires a previously defined format but directs its output to the job's log file. As with REMARK, only one 80-character line can be written. If the formatted line doesn't begin with the standard message prefix,

xxxxxb-b

(with *x* representing any character and *b* representing a blank), then the prefix FT060 is automatically inserted. The statement's form is:

```
LOG (format) var1 {,var2 ... ,varn};
```

format The SKOL identifier for a previously defined FORMAT statement

var_i A variable

DATA ITERATION (THE FOR CLAUSE)

All of the input/output statements discussed so far can make use of a FOR clause to specify an iterated data list in a manner similar to the implied DO list of FORTRAN. An item containing a FOR clause takes the following form in an INPUT, OUTPUT, or REMARK statement:

```
(FOR ivar = iexp TO iexp  BY iexp : var): fmt
```

ivar A simple integer variable

iexp A simple integer expression

var An array to which the input value is assigned

fmt A format specification sufficient to handle all of the iterations

If the BY phrase is omitted, the default increment is 1.

In a READ, WRITE, DECODE, ENCODE, or LOG statement, the format specification is omitted.

The format specification must contain no commas unless they are within parentheses that are part of the format. In theory the format could be omitted or preceded by two colons instead of one. However, the default format of O22 is inappropriate for a group of variables, and the 2-colon form produces obscure output (the FOR clause is output literally but with all spaces removed).

Example:

```
INPUT ((FOR OUTLIN = 1 TO 10: LENGTH(OUTLIN)): (5X, I3));
```

FOR clauses can be nested three deep. An inner FOR clause takes the place of the variable in the next outermost FOR clause, permitting I/O statements such as the following:

Example:

```
INPUT ((FOR I = 1 TO 3: (FOR J = 1 TO 3: I,J,A(I,J))):
  3(3(2I2, F10.5, 5X), //));
```

A FOR clause can sometimes be shortened to one of the following:

1.	(FOR *iexp*: *var*)
2.	(FOR *iexp*, *jexp*: *var*)
3.	(FOR *iexp*, *jexp*, *kexp*: *var*)

iexp, *jexp*, *kexp*
: Simple integers or integer variables

var
: An array to which the value is assigned

The SKOL macro translator expands these three forms into the following but uses internal names instead of i, j, and k:

1. (FOR i=1 TO *iexp*: *var*(i))

2. (FOR i=1 TO *iexp*:
   (FOR j=1 TO *jexp*: *var*(i,j))

3. (FOR i=1 TO *iexp*:
   (FOR j=1 TO *jexp*:
   (FOR k=1 TO *kexp*: *var*(i,j,k)))

Example:

```
FORMAT PERSON = (1X, 3A1, 1X, 2I5, F10.2);
READ ($IN, PERSON) (FOR 3: ΛP.INITIAL), ΛP.ID_NUMBER, ΛP.AGE,
     SALARY;
WRITE ($TEMP, PERSON) (FOR 3: ΛP.INITIAL), ΛP.ID_NUMBER,
      ΛP.AGE, SALARY;
```

Any of these forms of the FOR clause can be used in DATA statements, again in a manner similar to FORTRAN's implied DO list. (For an example, see section 2 of this publication under the DATA STATEMENT heading).

END = ... , ERR = ...

Some versions of FORTRAN, including CRAY-1 FORTRAN, permit the programmer to decide the direction of the program flow when a READ statement encounters an end-of-file or an error. The SKOL version of such a READ statement is used within an UNTIL block or in a routine called from within an UNTIL block. It takes the form:

READ (*file*{,*format*}{,END = *situation*}{,ERR = *situation*})
    var_1 {,var_2 ... ,var_n};

file	The name of an input file
format	The optional SKOL identifier for a previously defined FORMAT statement
situation	An identifier in an UNTIL statement
var_i	A variable name, an array name, an array element, or a FOR clause

Example:

```
UNTIL END_OF_DATA:
   READ ($IN, PERSON, END = END_OF_DATA)(FOR 3: ΛP.INITIAL),
       ΛP.ID_NUMBER, ΛP.AGE, SALARY;
ENDUNTIL;
```

Where the use of the END = form is impossible or impractical, one can achieve the same result by testing the logical function END_OF_FILE:

```
END_OF_FILE (file)
```

file A local dataset name such as '$IN' or '$OUT', a FORTRAN unit number, or an expression that resolves into either a FORTRAN unit number or a local dataset name

Examples:

```
FLAG = END_OF_FILE(TEMP);
IF END_OF_FILE($IN):
```

In order to avoid a diagnostic message when END_OF_FILE is used, the corresponding FORTRAN integer function must be declared:

```
EXTERNAL INTEGER: IEOF;
```

PAGE, SKIP

The PAGE and SKIP statements perform carriage control functions. They take the form:

```
{PAGE} file;
{SKIP}
```

file A local dataset name such as '$OUT', a FORTRAN unit number, or an expression that resolves into either a FORTRAN unit number or a local dataset name

PAGE skips to the first line of the next page; SKIP prints a blank line.

BUFFER IN, BUFFER OUT, UNIT, BUFFER_IN_LENGTH

BUFFER IN, BUFFER OUT, and UNIT are processed by being passed on to the FORTRAN compiler without change (like any statement that does not match a SKOL macro). However, because LENGTH is both a SKOL keyword (for string length) and a FORTRAN intrinsic function, the SKOL user must code BUFFER_IN_LENGTH when the FORTRAN function, LENGTH, is meant.

DIRECT-ACCESS I/O

Direct-access datasets (not to be confused with older random datasets) are implemented in CRAY-1 FORTRAN exactly as defined in the American National Standards Institute (ANSI) 77 programming language, FORTRAN, ANSI X3.9-1978 standard. The implementation includes new statements (OPEN, CLOSE, and INQUIRE) and a new parameter (REC=*n*) in READ and WRITE statements. SKOL differs from CFT in two ways.

In the OPEN, CLOSE, and INQUIRE statements, the CFT form ERR=*s* (specifying a statement label for error processing) is replaced in SKOL by ERR=*sit* (specifying a situation label).

In the READ and WRITE statements, the CFT keyword forms UNIT=*u* and FMT=*f* are not allowed in SKOL. Only the positional form is allowed for these two parameters, although the keyword form is accepted for ERR=*sit*, END=*sit*, and REC=*n*.

The RECL=*rl* parameter in OPEN and INQUIRE requires a record length *rl* in characters (not words).

USING SKOL ON THE CRAY-1 10

The recommended control statements for using SKOL on the CRAY-1 (assuming the SKOL source is in $IN) are:

```
SKOL.
CFT,L=0,E=3,I=$SKF,ON=A,OFF=Q.
LDR.
```

SKOL AND CFT CONTROL STATEMENTS

The SKOL macro translator and the CFT compiler are loaded and executed when their respective control statements are encountered in the job deck.

The format of the SKOL control statement is:

```
SKOL,I=idn,L=ldn,X=xdn,E=edn,O=odn,M=mdn,VIEW,LISTOFF,''''=x,@=y,#=z.
```

Parameters:

I=*idn* Specifies the dataset containing the SKOL source input. The default is $IN. Except for $IN and datasets with names of the form FT*nn*, all input datasets are accessed and rewound before they are read.

L=*ldn* Specifies the dataset to receive the indented SKOL listing. The default is $OUT. If L=0, no listing dataset is written.

X=*xdn* Specifies the dataset to receive the optional cross-reference listing for the SKOL source. The default, chosen if X appears alone, is *ldn*. If the X parameter is omitted, no cross-reference listing is written.

A separate program, SKOLREF, generates the cross-reference listing. The X parameter causes SKOL to associate the dataset name *xdn* with the alias $SKX and copy parts of the source input to the intermediate datasets $SKM and $SKI in preparation for SKOLREF. Therefore, whenever the X parameter appears on a SKOL control statement, a SKOLREF control statement should follow.

E=*edn* Specifies an optional dataset to receive a listing of all SKOL error messages, erroneous SKOL source lines, and macro expansion trace output. If the L dataset is $OUT, the default is no separate listing of errors and trace information; otherwise, the default for E is $OUT. If E appears alone, $OUT contains the listing of errors and trace information, no matter what the value of L.

In general, E overrides L. If the E dataset and the L dataset are the same, only the error and trace information is written to that dataset.

If the E and L datasets are different, the error lines (SKOL diagnostic messages and erroneous SKOL source lines) are written to both the E and L datasets, while the macro trace lines (generated by %H, %O, %T, %X, and %Y translator directives) are written only to the E dataset.

O=*odn* Specifies the dataset to receive the FORTRAN code for the CFT compiler. The default is $SKF.

M=*mdn* Specifies one to eight SKOL source input files, separated by colons if more than one is specified, containing supplementary SKOL definitions. If the SKOL source is in $IN, do not specify $IN here.

Except for $IN and datasets with names of the form FT*nn*, all input datasets are accessed if necessary and rewound before they are read. The file SKOLTXT specifies macro definitions for the basic SKOL language and is automatically read; it must not be named here.

Prior to any accesses, the system directory is searched for SKOLTXT and any M dataset whose name is not $IN or FT*nn*.

VIEW Enables the listing of translator directives that are ordinarily invisible. VIEW has the same effect as the translator directive %V, which is described below in this section.

LISTOFF Disables the listing of the SKOL source program until a %L is encountered

''''=*x* Specifies a single character to replace the apostrophe in SKOL's output. Since this option renders output unusable by CFT, it should only be used for debugging.

@=*y* Specifies a single character to replace the @ operator in SKOL's trace output. The use of this option does not interfere with the translation process.

#=*z* Specifies a single character to replace the # operator in SKOL's trace output. The use of this option does not interfere with the translation process.

The CFT control statement has the following form:

```
CFT,I=idn,L=ldn,B=bdn,C=cdn,E=n,ON=string,OFF=string,
                                                      TRUNC=nn,AIDS=aids,OPTION=option.
```

Parameters:

I=*idn* Specifies a source input dataset that must match the *odn* dataset in the SKOL control statement.

L=*ldn* Specifies a listable output dataset for the FORTRAN program. L=0 suppresses all list output except for warnings and error messages, which are written on $OUT.

B=*bdn* Specifies a binary output dataset. The default dataset is $BLD.

C=*cdn* Specifies a CAL dataset. The default is no assembly language output.

E=*n* Specifies the highest CFT message level to be suppressed. The default value is 3, but a value of 4 is recommended.

ON,OFF=*string*

Specifies a string of up to 15 letters, each of which enables or disables a particular compiler option (available compiler options are described in part 3 of the CFT manual, publication SR-0009). The following options are of special interest to SKOL users:

E,L Both are ON by default. If either is OFF, the list-control CDIRs (compiler directives) are not recognized, and the commands LIST CFT, NOLIST CFT, LIST CAL, NOLIST CAL, and EJECT have no effect. Turning E OFF disables all CDIRs.

G OFF by default. Turning it ON has no effect, because SKOL inserts CDIR(NOCODE) at the start of each subprogram.

C,T ON by default. They generate tables as part of the CFT listing. The SKOL user may want to turn them OFF.

O OFF by default. Since it inserts checks for array references that are out of bounds, it should be turned ON when debugging a program.

Q ON by default so that CFT aborts after 100 errors are detected. Turning it OFF produces a complete error listing.

A OFF by default so that fatal CFT errors do not prevent the continuation of the job. The SKOL user may want to turn it ON.

TRUNC=*nn* Specifies the number of bits, within the range of 0 to 47, to be truncated for all floating-point results. The default value is 0.

AIDS=*aids* Number of vectorization inhibition messages.

aids	Number of messages
LOOPNONE	None
LOOPPART	3 per compiler block; 100 messages per compilation (default)
LOOPALL	All

OPTION=*option*

Optimization level

option	Assumption
NOZEROINC	Constant increment integers cannot be incremented by zero variables (default)
ZEROINC	Constant increment integers can be incremented by zero variables

SKOL, FORTRAN, AND CAL LISTINGS

The user can specify listings for all or part of a SKOL program in the original SKOL, the generated FORTRAN, and the CRAY-1 Assembly Language (CAL) generated by CFT.

The LISTOFF parameter on the SKOL control statement specifies the initial listing status of the SKOL source code. The listing can be disabled or enabled at any point in the program with the %L- or %L translator directives, which are described below.

The L parameter and the E and L compiler options, which can be specified with the ON and OFF parameters on the CFT control statement, determine the initial status of the FORTRAN listing. The FORTRAN listing can be enabled or disabled by inserting one of the following statements in the SKOL source:

```
{LIST   }
{NOLIST} CFT;
```

The following steps describe how to list only part of a program.

1. Direct the CFT listing to a file by replacing L=0 on the CFT statement with L=*file*. The listing goes to $OUT by default if the L parameter is omitted.

2. Turn the listing off at the beginning of the program with the statement,

   NOLIST CFT;

3. Turn the listing on before the code to be listed with the statement,

   LIST CFT;

4. Turn the listing off again at the end of the section of desired code.

CAL listing can be enabled and disabled anywhere in the program. Each block of generated CAL code within which the CAL listing is momentarily enabled will be printed. The form for enabling or disabling the CAL listing is as follows:

```
{LIST  }
{NOLIST} CAL;
```

CAL listing is also disabled by the NOLIST CFT statement.

CFT COMPILER DIRECTIVES

The LIST and NOLIST statements generate compiler directives (CDIR$ cards) for CFT to act on. SKOL provides a general CDIR statement for generating CDIR$ cards. The form of the CDIR statement is:

```
CDIR (dir);
```

dir Any of the compiler directives described in part 3 of the CFT Reference Manual, publication SR-0009.

The following SKOL statements are translated into CDIR macros, which generate compiler directives.

```
LIST CFT;          CDIR(LIST);
NOLIST CFT;        CDIR(NOLIST); CDIR(EJECT);
LIST CAL;          CDIR(CODE);
NOLIST CAL;        CDIR(NOCODE);
LIST BLOCK;        CDIR(CODE, NOCODE);
EJECT;             CDIR(EJECT);
FLOWTRACE ON;      CDIR(FLOW);
FLOWTRACE OFF;     CDIR(NOFLOW);
```

Compiler directives not shown above must be generated from a CDIR statement coded by the SKOL programmer. The characters within the parentheses of a CDIR statement are not submitted to the normal pattern matching process, so they must always be literal copies of CFT's compiler directives.

ACCESSING SKOLTXT

Normally, no special control statements other than the SKOL, SKOLREF, and CFT statements are required, since SKOL performs its own accesses.

To redefine the language, the programmer can access a different version of the SKOL macros (DN=SKOLTXT). The ACCESS statement must appear before the SKOL statement in the job deck.

TRANSLATOR DIRECTIVES

The action of the SKOL translator can be modified for a given run, or a portion of a run, through the use of translator directives placed within the SKOL source. Any line containing a percent sign in column 1 is a directive to SKOL unless column 2 contains an apostrophe. This is an exception to SKOL's free format rule. No other commands can appear on the same line as the directive; whatever follows on the same line is treated as a comment. None of these directives are needed for a normal SKOL run. All of them, with the exception of %R, are portable. Some systems, lacking a means to test for end-of-file, may require the %% directive. Except for the single blank required after %R, no embedded blanks are allowed. The translator directives and their meanings are given in table 10-1.

Table 10-1. Translator directives

Command	Action
%A	Insert subsequent SKOL source lines as comments in the output to the FORTRAN compiler.
%A-	Suppress the output of SKOL source lines as FORTRAN comments.
%B	Insert blanks where needed to separate identifiers from the surrounding text. This directive has no effect on literal strings (text enclosed in apostrophes). A %B directive is near the end of the SKOLTXT file; it should never be necessary for the user to disable blank insertion with a %B- directive.

Table 10-1. Translator directives (continued)

Command	Action
%C*ij*	Scan subsequent input records from column 1 through the column designated by the two decimal digits, *ij*. If the CRAY-1 user specifies a line width greater than 72 and wants to use UPDATE to generate a compile file for input to SKOL, the UPDATE control statement must specify DW=*mm* (with *mm* greater than or equal to *ij*).
%D	Dump all subsequent macro definitions into the source listing as they are defined.
%D-	Disable the dumping of macro definitions.
%E*i* or %E*ij*	Allow the translated FORTRAN to be compiled (in the next job step) even if there are *i* or *ij* errors.
%F	Assume subsequent input text is FORTRAN; do not scan it for macro substitution.
%H	Print one line describing each addition or deletion to the label stack or the macro symbol stack and identifying the level of the stack. %H is automatically enabled by a %X or %X+ directive. Output is to $OUT or the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.
%H-	Disable the stack trace.
%I*i*	Indent subsequent SKOL statements, using *i* spaces for each level of nesting. Although the default is %I0, SKOLTXT contains a %I2 directive that takes effect as the source input begins.
%L	Begin listing lines of SKOL text. The SKOLTXT dataset contains a %L directive which takes effect as the source input begins. The LISTOFF keyword on the SKOL control statement can negate the effect of this %L directive.

Table 10-1. Translator directives (continued)

Command	Action
%L-	Stop listing SKOL text.
%O	Write the translated FORTRAN code to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT. %O is automatically enabled by a %X or %X+ directive.
%O-	Disable the listing of the generated FORTRAN code.
%P	Start a new page in the SKOL listing.
%Q	SKOL comments introduced by " are terminated by an end-of-line character unless closed by " on the same line. The use of this directive is dangerous.
%Q-	SKOL comments introduced by " must also be terminated by ". This is the default; the use of the %Q directive is dangerous.
%R *dataset*	Start reading input from the named dataset. An ACCESS is performed by SKOL if necessary, and the dataset is always rewound before reading (except for $IN). The dataset name must be no longer than seven characters, and there must be a single blank between the R and the dataset name.
%S	Resume processing input text as SKOL statements, assuming a %F directive has been used earlier.
%T	Trace macro expansion by printing the pattern matched and the replacement part (one line for each) for every match made in the SKOL source by the macro translator. No second line is printed if the replacement part is null. %T is automatically enabled by a %X directive. Output is to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.

Table 10-1. Translator directives (continued)

Command	Action
%T+	Perform the same function as %T, and also print one line to identify each parameter string. %T+ is automatically enabled by a %X+ directive. Output is to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.
%T-	Turn off the macro expansion trace. This is the default.
%U*i* or %U*ij*	Start reading input from the dataset associated with the FORTRAN unit number specified by decimal digit *i* or digits *ij*. There is a %U5 directive near the end of SKOLTXT that causes the translator to switch to FT05, or $IN, for source input; but special handling allows the source input dataset to be changed by the I parameter on the SKOL control statement. No datasets are rewound.
%V	View (enable listing of) directives that are usually not listed (%L, %P, %R, %U, and %%). The VIEW keyword, when placed on the SKOL control statement, has the same effect as placing a %V directive at the beginning of the source input.
%V-	Disable viewing of directives %L, %P, %R, %U, and %%. This is the default.
%X	Enable tracing of macro expansions except for parameter values. %X is equivalent to %T, %H, %O, and %Y. Output is to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.
%X+	Enable tracing of macro expansions including parameter values. %X+ is equivalent to %T+, %H, %O, and %Y. Output is to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.

Table 10-1. Translator directives (continued)

Command	Action
%X-	Disable tracing of the macro expansion process. This is the default.
%Y	Print any macro that requires a new source line to be read to continue a match, unless the character just before the new line is a semicolon. %Y is automatically enabled by a %X or %X+ directive. Output is to $OUT or to the error listing file (*edn*) if neither *edn* nor *ldn* (indented SKOL listing file) is $OUT.
%Y-	Disable the printing of macros that force a new source line to be read. This is the default.
%%	Terminate input from the current dataset and resume reading from the previous input dataset at a point following the %R or %U directive that caused the switch to the current dataset. If the current dataset is the primary source input dataset (corresponding to the I keyword on the SKOL control statement), then resume reading the SKOLTXT dataset immediately after the %U5 directive near its end. The SKOL macro translator creates a %% card when it senses a logical end-of-file, so no %% directives need ever be used.

TRANSLATING FORTRAN TO SKOL

The transition from CFT to SKOL can be accomplished gradually. Starting with a FORTRAN source file, simply insert a %F translator directive before the first statement and a %S directive at the end. %F means switch to FORTRAN, and %S means switch back to SKOL. The CFT control statement must be replaced by a SKOL statement and a new CFT statement resembling the one shown at the beginning of this section. For example:

```
CFT,I=infile, ... .
```

becomes

```
SKOL,I=infile.
CFT,I=$SKF, ... .
```

Within the FORTRAN source, the user can insert isolated fragments of SKOL code (such as OUTPUT statements) each preceded by a %S directive and followed by a %F directive.

In order to TRACE assignments to a variable, the user must insert a TRACE statement, surrounding it with %S and %F directives, and must convert the pertinent assignment statements to SKOL format by terminating them with semicolons and bracketing them with %S and %F directives. Continuation cards require special handling.

INDENTATION

The %I*i* control command sets the indentation scheme for a program listing. If %I0 is specified, the program is listed as it was keyed in, with whatever indentation was chosen by the programmer.

Following a %I*i* directive, SKOL indents *i* spaces for each nesting level. %I2 is the default setting. Nesting levels are associated with keywords that begin and end blocks. For example, if the keyword DO is detected by the translator, the next line's nesting level is incremented by one (two spaces of additional indentation if the default setting is used). The nesting level is decremented by one for the line containing the ENDDO. Beginnings and ends of blocks are thus at the same indentation level.

Internal SKOL rules control indentation for statements that exceed a single line. For all lines except those that have a % in column 1, the nesting level is shown at the left of the program text. The lowest level is level 0.

The maximum nesting level is currently 46. The maximum depends on the size of the macro symbol stack (see Appendix A). Lines are not indented past a certain nesting level. The value of the nesting level where indentation ceases is a function of the current values set by the %C and the %I translator directives, as table 10-2 shows for typical %I and %C settings.

Table 10-2. Nesting level where indentation ceases

	%I1	%I2	%I3
%C72	28	14	10
%C80	20	10	7

At nesting levels deeper than these values, all lines that are left-adjusted in the source text are indented the maximum amount. If the source text is already partially indented by the programmer, much deeper levels of indentation can be attained.

Besides making a SKOL program more readable, automatic indentation provides another advantage: all 72 (or 80) columns of each input line can be used no matter how deeply the line is nested.

If a line exceeds 72 columns (or the length established by a %C directive), the excess characters are shifted right on output. If UPDATE is used, the UPDATE sequence numbers are included in the text that is shifted, and they are vertically aligned near the right edge of the printed output.

MACRO TRANSLATION ERROR MESSAGES

When errors occur in the macro translation process, SKOL macros usually generate specific, helpful diagnostics. However, some error messages, such as those listed below, are obscure. Suggestions on what action to take follow each message. (A larger number of SKOL error messages is described in Appendix E and the complete set appears in the CRAY-OS Message Manual, publication SR-0039.)

UNCLOSED STRING (perhaps preceded by many other error messages)

The program ended or the input buffer overflowed before the closing apostrophe for a literal string was encountered. See if you have a run-on literal string. Check the column between the sequence number and the nesting level for an indication that a string is being continued.

UNCLOSED COMMENT

The program ended without the closing quotation mark for a comment. Check the column between the sequence number and the nesting level for an indication that a comment is being continued.

MACRO BUFFER OVERFLOW

Too many user-defined macros for the available space. Either make some of the macros local by placing their definitions inside a major segment or recompile the SKOL macro translator with a larger dimension for the array MM.

INPUT BUFFER OVERFLOW

A statement or a literal string extends over more than 25 source cards. Check the column between the sequence number and the nesting level for an indication that a string is being continued.

EXPANSION BUFFER OVERFLOW

Either a macro explosion has occurred (a user-defined macro or constant is causing a recursive match) or a user-defined macro expansion is too large for the buffer.

OUTPUT BUFFER OVERFLOW

A user-defined macro expansion is just a little too large for the buffer.

MACRO STACK OVERFLOW or LABEL STACK OVERFLOW

An error is present in a user-defined structural macro, or the nesting level exceeds 46.

MACRO STACK UNDERFLOW or LABEL STACK UNDERFLOW

An error is present in a user-defined structural macro, or the SKOL macros that protect against stack underflow have been redefined.

PARAMETER BUFFER OVERFLOW

The sum of the lengths of the actual parameters exceeds the space available for them.

BAD PARAMETER NUMBER

The character following a # operator in the replacement part of a macro definition is not a digit in the range from 1 to *n*, where *n* is the number of formal parameters in the pattern part of the macro.

INVALID PARAMETER ON % DIRECTIVE; *n* ASSUMED

A translator directive that required a number in columns 2 and 3, or a plus or minus sign in column 2, contained invalid digits or a number that was outside the range allowed for the directive. In all directives, a plus sign is treated as a 2, a blank as a 1, and a minus sign as a 0.

COULDN'T ACCESS *dataset*

The named dataset, either the source input (I) dataset or a supplementary dataset (M) or SKOLTXT itself, cannot be accessed. See if an ID (or some other ACCESS statement parameter) needs to be specified.

COULDN'T ASSIGN 'O' DATASET

The dataset that is to receive the generated FORTRAN code is already in use for some other purpose.

The SKOL macro translator generates the above messages, while the following messages are generated by the macros themselves.

UNEXPECTED COLON DISCARDED

See if there is a missing semicolon or colon on the previous line. See if a SKOL keyword is misspelled or misused.

BAD SYNTAX AT ' ... *text*'

The '*text*' is the remainder of a source statement. See if parentheses or brackets are unbalanced. See if a SKOL keyword is followed by a badly formed statement. A missing colon or semicolon can trigger this error.

UNEXPECTED '_'

See if a SKOL identifier is misspelled or undeclared.

UNEXPECTED '$'

This usually indicates an error in macro expansion; turn on the macro trace.

UNEXPECTED ' '

This message usually indicates a bad pointer usage.

The macro trace feature is useful in solving many macro translation problems, and it is also helpful in learning how the macro translation process works. To turn on the macro trace, insert a %X+ translator directive before the SKOL source lines to be traced. Insert a semicolon and a line containing a %X- translator directive after the lines. The semicolon forces out all of the trace information before the %X- turns off tracing.

Example:

```
%X+
OUTPUT((FOR 10: BUFF): 10A8);
;
%X-
```

APPENDIX SECTION

THE SKOL MACRO TRANSLATOR A

The SKOL macro translator is a FORTRAN program that reads a stream of free form source text and reformats it to conform to the textual rules for FORTRAN statements. As macro definitions are encountered in the input stream, they are stored away to become part of a body of rules for translating subsequent input.

This appendix describes how to construct a set of macro translation rules for a new language: that is, for extending a language such as SKOL in significant ways.

The translator consists of an initializer, an input processor, a macro translator, and an output processor. The flexibility and power of the translator is in large part due to the fact that almost all translation occurs in the macro translator portion. The translator is essentially table driven, with the table consisting of the macro definitions.

INITIALIZER

The first thing done in initialization is to read three lines from SKOLTXT, the standard input file. The first line contains text describing what version of the SKOL language is embodied in the macros that follow. The second and the third establish the format of the macro that generates all other macros.

The first line is printed as part of a header in the source program listing (the L dataset) as well as a comment in the generated FORTRAN (the O dataset).

The second line is used to establish a correspondence between symbols used in the translator documentation and actual characters to be used at each installation. For example, if no # is available, an available character (such as ?) can be substituted for it everywhere, including the second line of the SKOLTXT file. Replacing # with ? in this line is all that is needed to cause all subsequent occurrences of ? to be treated properly. The column correspondences are listed in table A-1.

Table A-1. Character correspondences

Column position	Symbol
1-10	0-9 (digits)
11-36	A-Z (upper case letters)
37	_ (underscore)
38	(blank)
39	.
40	(
41	)
42	,
43	=
44	+
45	-
46	*
47	/
48	[
49	]
50	<
51	>
52	;
53	:
54	'
55	"
56	{
57	}
59	$
60	%
61	#
62	@
63	^
64	&
65	?
66	!
67	|
68	a
69	z

If the lowercase letters a and z are replaced by the capital letters A and Z, the conversion of lowercase to uppercase (described below) is disabled.

The third line, described in table A-2, defines the seminal macro. This line defines the form that must be used to generate all other macros and should not be altered by anyone not familiar with the internal workings of the translator.

Table A-2. Input line establishing the macro definition format

Column	1 2 3 123456789012345678901234567890123456
Character	'.@.#. % '.#.'.= '.#.'.@.M G

The entire line is read using A2 format. Columns 1-6 define the internal representations of the three special operators. Each internal form is distinguished from the external form by the presence of the period. Columns 15-30 define the pattern part of the seminal macro (externally %'#'='#'). Columns 31-36 define the replacement part of the same macro (@MG). When SKOL recognizes the macro-generation pattern in the input stream, it substitutes @MG for it, acts on this special operator (storing a macro definition), and finally discards it from the input stream.

INPUT PROCESSOR

The input processor is called upon to read input lines of text and deliver slightly modified strings of characters in a single virtual stream to the macro translator portion. The input processor optionally prints the input lines in the source listing (the L dataset) and/or inserts them as comments in the generated FORTRAN (the O dataset).

Input lines with % in column 1 are interpreted as translator directives as described in section 10. Such lines generate no characters for the stream going to the macro translator.

All other lines are condensed by deleting comment strings beginning with "--" or enclosed in quotation marks and by replacing all sequences of two or more blanks by a single blank. If necessary, a blank is inserted between the end of one line and the beginning of the next. Naturally, a comment cannot begin within a literal (a text string delimited by apostrophes), nor are multiple blanks condensed within literals.

All lowercase alphabetic characters are translated to uppercase unless they occur in literals, comments, or translator directives.

During the translation of source text, an internal nesting level counter is incremented and decremented by occurrences of the operators @N+ and @N- respectively. @N+ and @N- are used in all the SKOL macros that delimit blocks of statements. The value of the nesting level counter controls optional automatic indentation of the source listing. See the description of automatic indentation in section 10.

MACRO TRANSLATOR

The stream of characters generated by the input processor is modified by text subsitution rules called macros, each consisting of a pattern part and a replacement part. The source input may be thought of as a single long string. The first substring found in the current input stream that matches the pattern specification of a macro is replaced by substitution text that is constructed using the replacement part of the same macro as a template. After a successful pattern match and replacement, the search for subsequent matches resumes at the beginning of the substituted text, which has just become the beginning of the current input stream. When no macros match any substring that begins with the first character of the current input stream, that first character is transmitted to the output processor and is never again eligible for macro translation. Pattern and replacement parts are scanned from left to right in determining pattern matches and constructing substitution text.

OUTPUT PROCESSOR

The stream of characters output from the macro translator is converted to lines of text acceptable as input to standard FORTRAN compilers by interpreting semicolons as end-of-statement indicators and employing the usual continuation line convention for statements too long for one line. Whenever an @ is encountered, the next word is interpreted as a binary integer and converted to 5-digit format, and the @ is discarded. When a digit or an @ occurs immediately after a semicolon, the decimal number is placed in columns 1-5, with a blank in column 6. Otherwise, the text following a semicolon is begun in column 7.

MACRO DEFINITION FORMAT

Text of the form

```
%'string' = 'string'
```

causes a new macro to be created whose pattern part is given by the first string and whose replacement part is given by the second string. The two strings are syntactically like ordinary literals in that they are delimited by apostrophes, and an embedded literal apostrophe is expressed by '' in accordance with the usual convention. In addition, any literal # or @ must be doubled to prevent it from being acted on immediately as a special operator.

For example, the macro definition

```
%'A''##B@@C' = 'X##1Y@@'''''
```

creates a new macro whose pattern part is

```
A'#B@C
```

and whose replacement part is

```
X#1Y@''
```

This macro causes A'#B@C to become X#1Y@''. In contrast, the macro definition

```
%'A''#B' = 'X#1@@'
```

creates a new macro that translates

```
A'123B    to   X123@
A'B       to   X@
A'CB      to   XC@
```

because of the special meaning associated with the single #. The # is used on the macro translator level much as it is used on the SKOL language level (see the description of the DEFINE statement in section 6). Its meaning in the context of macro translation is described below.

Text of the form

```
%'string' = *'string'
```

causes the second string to be concatenated onto the current replacement part of the most recently created macro whose pattern part is the first string. If none such exists, this form operates as if the * were absent. Actually, this concatenation format must be explicitly created by the following macro, which should be placed somewhere early in the input text (line 4 of SKOLTXT is safest):

```
%'%''#''=*''#'''='@MC'
```

SIMPLE TEXT SUBSTITUTION

The simplest form of macro specifies the substitution of one literal string for another, as in these two erroneous macros:

```
%'PI' = '3.14159'
%'N' = '7'
```

The error lies in the fact that these macros cause each occurrence of any of the following:

```
PIN
PI N
P IN
```

to be replaced by

```
3.141597
```

because blanks in the input text are considered null strings during pattern matching. (The · character will be used where necessary in this appendix to indicate the position of a blank.) The text

```
·UP·INTO·
```

would become

```
·U3.14159TO·
```

which is probably not what was intended. To guard against this difficulty, the macro must require blanks before and after the string to be translated.

```
%'·PI·' = '·3.14159·'
```

This macro matches only if the P is preceded by a blank and the I is followed by a blank. Also note that the blanks surrounding PI will disappear from the text unless explicitly replaced. Replacing the blank after PI is critically important, since that blank may delimit the start of another identifier or keyword.

The macros

```
%'·BIG·' = '·1.0E50·'
%'·N·' = '·3·'
%'·MAX·' = '·BIG·/·N·'
```

will translate

```
·MAX·
```

into

```
·BIG·/·N·
```

and then into

```
·1.0E50·/·3·
```

Without the careful blank replacement this translation would not be successful. Note that using DEFINE instead of % to preface macro definitions relieves the user of the need to provide blanks in the right-hand (replacement) part of the definition, but they must still be specified explicitly in the left-hand (pattern) part.

Because the macro translation process resumes the matching process at the start of the substitution text that has just been generated, infinite recursion is a potential danger. The innocent looking abbreviation macro

```
%'ABBR' = 'ABBREVIATION'
```

causes

```
ABBR
```

to become

```
ABBREVIATION
```

and then

```
ABBREVIATIONEVIATION
```

and the substitution continues endlessly. To avoid this, use a no-rescan delimiter (") to enclose a portion of substitution text that should not be reconsidered for further matching. The following macros

```
%'ABBR' = ' "ABBREVIATION" '
%'ABBR' = ' "ABBR"EVIATION '
%'ABBR' = ' "A"BBREVIATION '
```

are all safe against infinite recursion.

The simple substitution macros are useful in localizing the definition of certain constants like array sizes and tolerances, which can then be modified from compilation to compilation by simple textual changes in a macro such as

```
%' MAXLIM ' = ' 100 '
```

which applies eventually to such text as

```
INTEGER: A(MAXLIM), B(MAXLIM,2);
DO I = 1 TO MAXLIM:
IF J = MAXLIM:
```

PARAMETERS IN MACROS

The pattern part of a macro definition can contain up to nine formal (or "dummy") parameters, each of which represents a variable length character string. In a pattern, each parameter is denoted by the symbol #. For example,

```
'EXAMPLE#PATTERN#DEFINITION'
```

contains two formal parameters. The formal parameters are positional; that is, the first formal parameter is the first # encountered (reading left to right), the second formal parameter is the second # encountered, and so on. The corresponding actual parameters are detected and saved during the matching process. For example, in the string

```
EXAMPLE OF A PATTERN IN A MACRO DEFINITION
```

(assuming the previous example is the pattern to be matched), the first actual parameter is the string, "OF A", and the second actual parameter is the string, "IN A MACRO". The surrounding blanks are discarded and not included in the actual parameters.

After a macro is matched, it is expanded. The expansion process consists of deleting the program text that matched the pattern part of the macro and substituting the replacement part of the macro.

The replacement part can contain an arbitrary number of occurrences of formal parameters of the form #*i*, where *i* is a digit between 1 and 9. During expansion, each formal parameter #*i* of the replacement part is replaced by the *i*th actual parameter. A given formal parameter can appear zero or more times in the replacement part.

For example, the pattern part of the macro definition

```
%';SWAP(#,#);' = ';R=#1;#1=#2;#2=R;'
```

would match the program text

```
;SWAP(A(3,K),Y);
```

Any substring of the program text can match a formal (dummy) parameter and become an actual parameter except for the following restrictions:

1. The substring cannot contain an end-of-statement semicolon. This restriction prevents "run-away" macros from consuming large parts of the program.

2. Parentheses and square brackets must be correctly matched (balanced). This facilitates the construction of macros that treat expressions as indivisible units.

During the matching process the actual parameters are saved in a holding buffer. Upon completion of the matching process (that is, when the final character in the program text matches the final character in the pattern part), the expansion of the macro takes place. The actual parameters replace all occurrences of the corresponding formal parameters, producing

```
;R=A(3,K);A(3,K)=Y;Y=R;
```

The comma after the 3 does not match the comma in the pattern part, since A(3 is not correctly parenthesized. This is an example of the utility of the parenthesis rule. Each formal parameter occurs twice in the replacement part, and therefore each actual parameter occurs twice in the resulting string.

Blanks are trimmed off both sides of an actual parameter before the string goes into the replacement text. As a result, the above SWAP macro will translate

```
;·SWAP·(·X·,·Y·);
```

to

```
;R=X;X=Y;Y=R;
```

and if we had previously created an abbreviation macro like

```
%'·X·' = 'VOLUME'
```

it would not take effect because of the blank stripping. The easiest way to handle all such situations is to use DEFINE in place of %. However, it is also possible to cope with the problem by carefully adding blanks around each #*i*. The SWAP macro could be rewritten as

```
%';SWAP(#,#);' = ';R=·#1·;·#1·=·#2·;·#2·=R;'
```

and then

```
;SWAP(X,Y);
```

would be translated into

```
;R=·X·;·X·=·Y·;·Y·=R;
```

which then becomes

```
;R=VOLUME;VOLUME=·Y·;·Y·=R;
```

A 3-character string, #@*d*, when encountered in the pattern part of a macro definition, matches an arbitrary string of *n* characters where *n* is the integer value of character *d* in base 36 notation (1,2,3,4,5,6,7,8,9,A,B, ... Z). For example,

```
%'(#@2#@3)' = '#2,#1'
```

translates (25987) to 987,25. And

```
%'?!$#@Z' = ''
```

discards the sequence ?!$ and the next 35 characters. The only formal parameter of this type used in SKOLTXT is #@1.

NUMBER GENERATORS

Whenever @LG occurs in the replacement part of a macro, it specifies that when that replacement part is used to create substitution text for a matched pattern, the @LG is to be replaced by @*n*, where *n* is the value of an integer counter (greater than 10000) that is incremented by 10 each time an @LG is employed to create a number. Each number *n* that is created by @LG differs from all others by at least 10 so that *n*,*n*+1, ... *n*+9 are distinct from *m*,*m*+1, ... *m*+9 if both *m* and *n* are created by @LG. The @ preceding the number *n* is actually an internal non-standard character distinct from literal @. It is called "internal @."

These numbers preceded by an internal @ are converted to 5-digit decimal numbers in the generated FORTRAN and so can be used to create statement numbers or, when prefaced by I00, unique identifiers that are integers by default even when all other variable names must be declared.

Another internal integer counter can be initialized to zero (@I0), incremented by one (@I+) or decremented by one (@I-). Its current value, in the form @*n*, replaces @IC whenever @IC is encountered in a replacement part while creating substitution text. As with @LG, the @IC is replaced by @*n*, where the @ is an internal @.

The macro pattern string '@#@1' matches an internal @ followed by any single character and can be used to recognize text of the form @*n* produced by the special number operators, because the value of *n* occupies one full computer word, just as a character does. SKOL macros make use of @#@1 in this way.

STACKS FOR NUMBERS AND SYMBOLS

A number stack and a symbol stack are maintained to allow the correct macro translation of nested "parenthesized" language constructions. The number stack generally holds numbers created by @LG for use as FORTRAN statement numbers or to create auxiliary variables. The symbol stack holds single character symbols that are generally used to remember context as the translation proceeds.

When @*n*@LS*d* occurs in a replacement part, where *n* is a generated number and *d* a decimal digit, then nothing is actually placed into the substitution text, but *n* is pushed into the number stack at position *d*. The position corresponding to *d* = 0 is the actual stack top followed by positions 1,2, ... 9. Pushing or popping at position *d* operates as if position *d* were at the top of the stack, and numbers at positions 0,1, ... *d*-1 are not disturbed. The stack may contain as many as 100 items but can only be pushed or popped at the topmost 10 positions.

For example, an occurrence of @LG@LS*d* in a replacement part is processed from left to right, and the @LG becomes @*n*. @*n*@LS*d* becomes the null string while *n* is pushed onto the number stack at position *d*.

The symbols @LU*d* cause no substitution text to be generated, but the number stack is popped at position *d*, and the number previously at that position is discarded from the stack.

An occurrence of @LC*ij*, where *i* and *j* are decimal digits, causes @*n* to go into the substitution text where *n* = *m*+*j* and *m* is the current number at position *i* of the number stack.

Replacement text of the form @MS*x* causes the 1-character symbol *x* to be pushed onto the symbol stack while no substitution text is generated. @MU causes @*x* to be placed into the substitution text where *x* is the top symbol on the symbol stack, which is popped simultaneously. The text @MR*x* causes the top symbol to be replaced by *x* and generates no substitution text. In @*x*, the @ is literal.

As a simple but typical example of the use of these two stacks, we show macros for the implementation of an infinite repetition with the syntax:

;REPEAT: *block* ENDREPEAT;

In the case of the example, the *block* is merely a CONTINUE statement.

The macros are as follows:

```
%'$%X#@1= '$%X#1%@MU%'
%';REPEAT:'= '@N+@LG@LS0";@LC01CONTINUE";'
   $BUG.LOOP%OUTPUT(''*** Line @SQ:  REPEAT'');@MSV'
%';ENDREPEAT;'=  '@N-$%XV'
%'$%XV%@@V%'=  '%@@V%'
%'$%XV%@@1%'=  '@MS1?%XV'
%'?%XV'=  '$E%(MISSING ''REPEAT'');'
%'%@@V%'=  ';GOTO @LC01@LU0;'
```

The following example shows the macro expansion for the simple REPEAT structure:

```
1.      0
2.      0   main (REPTEST):
3.      1
4.      0   begin:
5.          %X+
        Pattern matched is ;REPEAT:
                           {10020} added to top of label stack (level  2)
                           'V' pushed onto macro stack (level  4)
              Expansion is ";@{10021}CONTINUE"; $BUG.LOOP%OUTPUT('*** Line
                              @{6}:  REPEAT');
               Followed by
                FORTRAN --        CONTINUE
                FORTRAN -- 10021 CONTINUE
        Pattern matched is $BUG.LOOP%#;
             Argument 1 is OUTPUT('*** Line @{6}:  REPEAT')
              Expansion is ;
               Followed by
6.      1     REPEAT:
7.      2       CONTINUE;
        Pattern matched is ;ENDREPEAT;
              Expansion is $%XV
8.      1     ENDREPEAT;
        Pattern matched is $%X#@1
             Argument 1 is V
                           'V' popped off of macro stack (level  4)
              Expansion is $%XV%@V%
               Followed by  ;
        Pattern matched is $%XV%@V%
              Expansion is %@V%
               Followed by  ;
        Pattern matched is %@V%
                           {10020} removed from top of label stack (level  2)
              Expansion is ;GOTO @{10021};
               Followed by  ;
                FORTRAN --        CONTINUE
                FORTRAN --        GOTO 10021
9.      1     ;
10.         %X-
11.     0   endmain;
12.     0
```

If every nested control structure is translated in such a way that both the macro symbol stack and the label stack are left in the state in which they are encountered, then a correctly structured program is translated in a consistent way.

The macro symbol stack can be used to recall what type of construction is ended by a particular END if several types of control use the same keywords, such as BEGIN and END, to bracket blocks of statements. But if the language being implemented has block terminator keywords that are specific to each construction, as is the case with SKOL, then the contextual recall provided by the macro symbol stack is not strictly necessary and only serves to provide a check on the structure of the program.

MACRO DEFINITIONS IN REPLACEMENT TEXT

A powerful feature of a macro-based language such as SKOL is the ability to embed macro definitions within the replacement part of other macros. This allows the creation of new macros whose form is determined by the text being processed. The following single macro could be used to provide a simple TRACE capability, causing all assignments to a given variable to be followed by a printout of the variable name and the newly assigned value.

```
%';TRACE #;' =
'%'';#1=##;''='"";" #1 = ##1 ; PRINT(''''#1='''',#1);'';'
```

If this macro is used to process the statement

```
;TRACE X;
```

then the initial substitution would yield

```
%';X=#;'='";" X = #1 ; PRINT(''X='',X);';
```

in which all the double apostrophes and double pound signs (#) in the macro definitions have been reduced to single characters, while the single pound signs have been processed as operators. The semicolon at the very end of the substituted text replaces the one terminating the original statement, TRACE X. It is needed to introduce whatever statement follows TRACE X. The rest of the substituted text defines a single new macro that will change any succeeding statement of the form:

```
;X = exp;
```

to these two statements:

```
";" X = exp ; PRINT('X=',X);
```

The spaces around X and *exp* allow further substitution, while the quotation marks around the first semicolon prevent the recursive matching that would otherwise occur.

IMPORTANCE OF ORDER IN MACRO CREATION

All attempts to match text with pattern parts of macros begin with the most recently created macro. Hence, if the macro definitions

```
%'ABC' = '2'
%'ABC' = '1'
```

occur in that order, ABC will be translated to 1, and the first macro is essentially useless. If the macro %'AB' = '3' is then created, all subsequent instances of ABC become 3C, and the first two macros are both useless.

It is important to keep the search order in mind for safe macro implementations.

IMPLEMENTATION OF PASCAL SCALAR TYPES AND CASE STATEMENT

The syntax for a scalar type declaration is:

```
TYPE·type-name=(constant-list);
```

An example is as follows:

```
TYPE·AGE_GROUP=(YOUTH,ADULT);
```

The macro

```
;TYPE # =
```

matches everything through the equal sign in the declaration and replaces it with

```
!typ%AGE_GROUP=
```

The string originally to the right of the equal sign is as yet unmatched, so the declaration line looks like the following at this point.

```
!typ%AGE_GROUP=(YOUTH, ADULT);
```

This line is matched by the macro,

```
!typ%#=(#);
```

and is replaced by the following six statements:

```
%' FIRST(AGE_GROUP'='1'
%'!sl!oAGE_GROUP'='!sl'
%'; AGE_GROUP : '='";INTEGER*2" '
!mem%AGE_GROUP$YOUTH , ADULT,;
!lastAGE_GROUP;
!i0 ;
```

The first of the six statements defines the FIRST value of TYPE AGE_GROUP as 1 (the FIRST and LAST functions are described in section 4 of this publication).

The second statement initializes a new macro with '!sl' as its replacement part. A little later, the !tally macro (invoked by !mem) concatenates the subtype names onto the replacement part of this new macro. Whenever it is not eventually deleted by the macros, it is an indication of an error in the source program.

The third statement shows that the type AGE_GROUP appears as type integer to FORTRAN. Any variables subsequently assigned to AGE_GROUP will be stored as 24-bit integers.

The fourth statement is the progenitor of a series of matches and replacements that assign ordinal numbers by picking off each of the type's members one at a time. The fourth statement itself is matched up to the first value and the comma that follows it. The remainder (ADULT,;) is tacked on at the end of the replacement part to produce:

```
!tally%AGE_GROUP$YOUTH; ADULT,;
```

This in turn is replaced by four statements.

```
%'!s1!oAGE_GROUP,'=*'YOUTH,'
%'!s3AGE_GROUP,'=*' YOUTH,'
%' YOUTH'='@ 1 '
!mem%AGE_GROUP$ ADULT,;
```

The third statement in this group assigns the ordinal number 1 to the value YOUTH. The fourth statement is a duplication of the fourth statement in the original set of six, only this time the value ADULT is present in the second parameter position instead of YOUTH. The ordinal count is incremented, and the above four statements are generated again with ADULT in the place of YOUTH. Since there are no further values, the fourth statement terminates with AGE_GROUP$; and won't match the macro pattern

```
!mem%#$#,
```

a third time.

Now that the ordinal count has reached it final value, the fifth statement of the original six is replaced by a macro that assigns the final value of the ordinal count (2 in this case) to LAST(AGE_GROUP).

The sixth statement is matched and replaced by the internal operator @I0, which resets the ordinal count to zero and is replaced in turn by an empty string.

Pattern matches and replacements can be traced through SKOL's macro expansion facility. See the %X, %T, %H, %O, and %Y translator directives in section 10 of this publication for information on macro expansion options. The following sample program will be traced:

```
MAIN:
  TYPE AGE_GROUP = (YOUTH, ADULT);
  AGE_GROUP: SUBJECT;
  LOGICAL: VOTER;
BEGIN:
  WHEN SUBJECT: AGE_GROUP IS:
    [YOUTH]:
      VOTER = FALSE;
    [ADULT]:
      VOTER = TRUE;
  ENDWHEN;
ENDMAIN;
```

In the following listing, the expansion (which is enabled here only for the relevant lines) precedes the SKOL source line to which it applies, because automatic indentation requires that the source listing output be delayed by one line.

```
At 14:46:09 on 06/09/80:  Cray-1 SKOL macro translation                    Translator revision level 1.08

 1.     0   MAIN:
 2.         %X+
        Pattern matched is ;TYPE # =
             Argument 1 is AGE_GROUP
                        Internal counter =  0
              Expansion is !typ%AGE_GROUP=
               Followed by  ( YOUTH , ADULT );
        Pattern matched is !typ%#=(#);
             Argument 1 is AGE_GROUP
             Argument 2 is YOUTH , ADULT
              Expansion is %' FIRST(AGE_GROUP)'='1' %'!s1!oAGE_GROUP,'='!s1' %'; AGE_GROUP : '='";INTEGER*2
                           " '!mem%AGE_GROUP$YOUTH , ADULT,;!lastAGE_GROUP;!i0 ;
        Pattern matched is %'#'='#'
             Argument 1 is  FIRST(AGE_GROUP)
             Argument 2 is 1
        Pattern matched is %'#'='#'
             Argument 1 is !s1!oAGE_GROUP,
             Argument 2 is !s1
        Pattern matched is %'#'='#'
             Argument 1 is ; AGE_GROUP :
             Argument 2 is ";INTEGER*2"
        Pattern matched is !mem%#$#,
             Argument 1 is AGE_GROUP
             Argument 2 is YOUTH
              Expansion is !tally%AGE_GROUP$YOUTH;
               Followed by  ADULT,;!lastAGE_GRO...
        Pattern matched is !tally%#$#;
             Argument 1 is AGE_GROUP
             Argument 2 is YOUTH
                        Internal counter =  1
              Expansion is %'!s1!oAGE_GROUP,'=*'YOUTH,' %'!s3AGE_GROUP,'=*' YOUTH ,' %' YOUTH '='@(1)'!mem%
                           AGE_GROUP$
               Followed by  ADULT,;!lastAGE_GRO...
        Pattern matched is %'#'=*'#'
             Argument 1 is !s1!oAGE_GROUP,
             Argument 2 is YOUTH,
        Pattern matched is %'#'=*'#'
             Argument 1 is !s3AGE_GROUP,
             Argument 2 is  YOUTH ,
```

```
Pattern matched is %'#'='#'
     Argument 1 is  YOUTH
     Argument 2 is @{1}
Pattern matched is !mem%##$#,
     Argument 1 is AGE_GROUP
     Argument 2 is ADULT
      Expansion is !tally%AGE_GROUP$ADULT;
       Followed by ;!lastAGE_GROUP;!i0 ...
Pattern matched is !tally%#$#;
     Argument 1 is AGE_GROUP
     Argument 2 is ADULT
                Internal counter =  2
      Expansion is %'!s1!oAGE_GROUP,'=*'ADULT,' %'!s3AGE_GROUP,'=*' ADULT ,' %' ADULT '='@{2}'!mem%
                   AGE_GROUP$
       Followed by ;!lastAGE_GROUP;!i0 ...
Pattern matched is %'#'=*'#'
     Argument 1 is !s1!oAGE_GROUP,
     Argument 2 is ADULT,
Pattern matched is %'#'=*'#'
     Argument 1 is !s3AGE_GROUP,
     Argument 2 is  ADULT ,
Pattern matched is %'#'='#'
     Argument 1 is  ADULT
     Argument 2 is @{2}
Pattern matched is !mem#$;
     Argument 1 is %AGE_GROUP
Pattern matched is !last#;
     Argument 1 is AGE_GROUP
      Expansion is %' LAST(AGE_GROUP)'='@{2}';
       Followed by !i0 ;
Pattern matched is %'#'='#'
     Argument 1 is  LAST(AGE_GROUP)
     Argument 2 is @{2}
        FORTRAN ---        IMPLICIT INTEGER (A-H,O-Z)
Pattern matched is !i0
                Internal counter =  0
      Expansion is
       Followed by ;
3.   1     TYPE AGE_GROUP = (YOUTH, ADULT);
Pattern matched is ; AGE_GROUP :
      Expansion is ";INTEGER*2"
       Followed by SUBJECT ;
```

```
4.      1     AGE_GROUP: SUBJECT;
5           %X-
6.      1     LOGICAL: VOTER;
7.      0   BEGIN:
8.          %X+
        Pattern matched is ;WHEN #:# IS:
             Argument 1 is SUBJECT
             Argument 2 is AGE_GROUP
                        {10020} added to top of label stack (level  2)
                        'C' pushed onto macro stack (level  4)
              Expansion is ";I00@{10022}=(" SUBJECT )-( FIRST(AGE_GROUP)-1); $BUG.WHEN%"IF(I00@{10022}.LT.1
                            .OR. I00@{10022}.GT.(" LAST(AGE_GROUP)-( FIRST(AGE_GROUP)-1))) CALL SKERR( $WHE
                           N ",@{9})";GOTO @{10023}; %'!wh@{10020}'='; @{10023}GOTO(!s1!oAGE_GROUP,),"I00@{1
                           0022}";@{10021}CONTINUE;' !s5
               Followed by
                FORTRAN --        LOGICAL VOTER
        Pattern matched is  FIRST(AGE_GROUP)
              Expansion is 1
               Followed by -1); $BUG.WHEN%"IF(I...
                FORTRAN ---        I0010022=( SUBJECT )-(1-1)
        Pattern matched is $BUG.#%
             Argument 1 is WHEN
              Expansion is ;
               Followed by "IF(I00@{10022}.LT.1...
        Pattern matched is  LAST(AGE_GROUP)
              Expansion is @{2}
               Followed by -( FIRST(AGE_GROUP)-...
        Pattern matched is  FIRST(AGE_GROUP)
              Expansion is 1
               Followed by -1))) CALL SKERR( $W...
        Pattern matched is  $WHEN
              Expansion is @{4}
               Followed by ",@{9})";GOTO @{1002...
                FORTRAN --        IF(I0010022.LT.1 .OR. I0010022.GT.(2-(1-1))) CALL SKERR(4,9)
                FORTRAN --        GOTO 10023
        Pattern matched is %'#'='#'
             Argument 1 is !wh@{10020}
             Argument 2 is ;@{10023}GOTO(!s1!oAGE_GROUP,),"I00@{10022}";@{10021}CONTINUE;
9.      1     WHEN SUBJECT: AGE_GROUP IS:
        Pattern matched is !s5E
              Expansion is !s2
```

```
          Followed by  YOUTH ]:
     Pattern matched is !s2#]:
          Argument 1 is YOUTH
                     (10030) added to top of label stack (level  3)
                     'B' pushed onto macro stack (level  5)
           Expansion is ";"!s3YOUTH,;@(10030)CONTINUE;
           Followed by
     Pattern matched is !s3#,
          Argument 1 is YOUTH
           Expansion is %'?@(10020)YOUTH,'='@(10030),'!s3
           Followed by ;@(10030)CONTINUE;
     Pattern matched is %'#'='#'
          Argument 1 is ?@(10020)YOUTH,
          Argument 2 is @(10030),
     Pattern matched is !s3;
                     (10030) removed from top of label stack (level  3)
           Expansion is ;
           Followed by @(10030)CONTINUE;
10.  2       [YOUTH]:
             FORTRAN -- 10030 CONTINUE
     Pattern matched is =#;
          Argument 1 is FALSE
           Expansion is "="( FALSE );
     Pattern matched is  FALSE
           Expansion is ".FALSE."
           Followed by );
11.  3         VOTER = FALSE;
     Pattern matched is ;[
           Expansion is ";"$%XB
           Followed by  ADULT ]:
             FORTRAN --       VOTER =(.FALSE. )
     Pattern matched is $%X#@1
          Argument 1 is B
                     'B' popped off of macro stack (level  5)
           Expansion is $%XB%@B%
           Followed by  ADULT ]:
     Pattern matched is $%XB%@B%
           Expansion is %@B%
           Followed by  ADULT ]:
     Pattern matched is %@B%
           Expansion is ";"GOTO @(10021);!s2
           Followed by  ADULT ]:
             FORTRAN --       GOTO 10021
```

```
          Pattern matched is !s2#]:
               Argument 1 is ADULT
                          {10040} added to top of label stack (level  3)
                          'B' pushed onto macro stack (level  5)
                Expansion is ";"!s3ADULT,;@{10040}CONTINUE;
                 Followed by
          Pattern matched is !s3#,
               Argument 1 is ADULT
                Expansion is %'?@{10020}ADULT,'='@{10040},'!s3
                 Followed by ;@{10040}CONTINUE;
          Pattern matched is %'#'='#'
               Argument 1 is ?@{10020}ADULT,
               Argument 2 is @{10040},
          Pattern matched is !s3;
                          {10040} removed from top of label stack (level  3)
                Expansion is ;
                 Followed by @{10040}CONTINUE;
 12.      2        [ADULT]:
                   FORTRAN --- 10040 CONTINUE
          Pattern matched is =#;
               Argument 1 is TRUE
                Expansion is "="( TRUE );
          Pattern matched is  TRUE
                Expansion is ".TRUE."
                 Followed by );
 13.      3          VOTER=TRUE;
          Pattern matched is ;ENDWHEN;
                Expansion is $%XD
 14.      1     ENDWHEN;
 15.          %X-
 16.      0   ENDMAIN;


269 = LONGEST MACRO TRACE STRING
32366 WORDS (34%) USED FOR MACRO STORAGE

%- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -%
```

A single macro handles the variable declaration by simply substituting

```
";INTEGER*2 "
```

for

```
; AGE_GROUP:
```

Note the double quotation marks around the replacement part to prevent a rescan that would cause an unwanted match.

The SKOL source line

```
WHEN SUBJECT: AGE_GROUP IS:
```

is matched by the WHEN macro and translated into roughly four lines of text. Other macros manipulate the error-checking apparatus and insert ordinal numbers for instances of FIRST and LAST. The process constructs two statement labels for a computed GOTO and creates a FORTRAN statement that checks for out-of-range variables. The variable I0010022 contains the value of the variable SUBJECT.

Three characters, !s5, remain from the long WHEN macro after the WHEN statement is processed. These characters are matched by the macro

```
!s5[
```

when the bracket beginning the statement

```
[YOUTH]:
```

is encountered in the SKOL source. The replacement part is

```
!s2
```

giving us

```
!s2YOUTH]:
```

which is matched and replaced by

```
%'?@ 10020 YOUTH,'='@ 10030 ,'!s3
```

The label 10030 gives the location for the computed GOTO to jump to when I0010022 takes on a value equal to the ordinal number of YOUTH. The value ADULT is matched with the label 10020 following another set of the same macros.

TRUE and FALSE are translated into .TRUE. and .FALSE., and parentheses are added, by general macros.

The resulting FORTRAN program turns out as follows:

```
      SUBROUTINE MAINSUB
      INTEGER*2 SUBJECT
      LOGICAL VOTER
      I0010022=( SUBJECT )-(1-1)
      IF(I0010022.LT.1 .OR. I0010022.GT.(2-(1-1))) CALL SKERR(4,9)
      GOTO 10023
10030 CONTINUE
      VOTER =(.FALSE. )
      GOTO 10021
10040 CONTINUE
      VOTER =(.TRUE. )
      GOTO 10021
10023 GOTO(10030,10040),I0010022
10021 CONTINUE
      END
```

Table A-3 describes the effect of macros found in the replacement parts of macros.

Table A-3. Operators in replacement parts of macros

Operator	Substitute text	Side effect
@CD	-	Generate a CFT compiler directive whose variable text is taken from the first actual parameter. Continuation cards are generated if necessary.
@CE	-	End a conditional generation block introduced by either @CN or @CG.
@CG	-	Begin a block within which FORTRAN code generation is enabled.
@CN	-	Begin a block within which no FORTRAN code is to be generated.
@FE	-	Force logical end-of-statement (for automatic indentation only). Treat the current source line as if its last character is a semicolon.

Table A-3. Operators in replacement parts of macros (continued)

Operator	Substitute text	Side effect
@FF	-	Force statement folding (for automatic indentation only). Treat the current line as an incomplete statement even if the last character is a colon.
@I+	-	The integer counter is increased by 1.
@I0	-	The internal integer counter is initialized to 0.
@IB	-	Insert blanks into the replacement part of the next macro created by an @MG operator.
@IC	@*n*	The binary value *n* is the current value of the integer counter.
@LC*ij*	@*n*	The value of *n* is the sum of the decimal digit *j* and the integer in the label stack at the position indicated by the decimal digit *i*.
@LG	@*n*	The label generator is increased by 10, so that the result (*n*) is a unique integer greater than 10000 and divisible by 10.
@LU*d*	-	The integer in the label stack at position *d* is popped off and discarded.
@MC	-	Find a macro whose pattern part matches the first parameter. If there is a matching macro, catenate the second parameter onto its replacement part; if not, create a new macro, as in @MG.
@MD	-	Delete all macro definitions and macro catenations that were stored in the macro buffer since the last @ME operator was encountered. If no @ME operator has been seen yet, @MD does nothing.

Table A-3. Operators in replacement parts of macros (continued)

Operator	Substitute text	Side effect
@ME	-	Marks the current end of all macro definitions and macro catenations. The effect is to define all existing macros as global, so that any that follow are local.
@MG	-	Create a new macro whose pattern is the string matching the first parameter and whose replacement part is the string matching the second parameter.
@MM	-	A copy of the text string matching the first parameter in the macro pattern is printed as an error message in the source listing, and if L is not $OUT, the current source line is printed.
@MR*x*	-	The character *x* replaces the current symbol at the top of the symbol stack.
@MS*x*	-	The character *x* is pushed onto the symbol stack.
@MU	@*x*	The character *x* is the symbol at the top of the symbol stack and is popped off.
@MW	-	Same as @MM except that instead of an error message, a warning is printed.
@N+	-	The current nesting level is increased by 1.
@N-	-	The current nesting level is decreased by 1.
@N*x*	-	If the character *x* matches the character at the top of the symbol stack, the current nesting level is decreased by 1 (as in @N-); if not, no action is taken.
@SQ	@*n*	The binary value *n* represents the current line number in the SKOL source.
@*n*@I=	-	The integer counter is set to *n*.
@*n*@LS*d*	-	The binary value *n* is pushed onto the label stack at the position indicated by character decimal digit *d* (0=top).

THE CHAR DATA TYPE

B

SKOL does not supply the character data type as a standard type. But it does allow the user to define type CHAR in the manner of other user-defined types (see section 4), treating it as a special case. If CHAR is declared as a type, a single-character constant can be coded simply by enclosing it in apostrophes, as in 'X'. If CHAR is not declared, 'X' is assumed to mean 'X'R, and the forms 'X'R, 'X'L, and 'X'H have the normal FORTRAN meanings.

The CHAR type need not be declared at all. Declaring it increases portability and permits the use of scalar case statements whose labels are single-character constants. But type CHAR introduces inefficiency during the reading and writing of character data.

This appendix describes type CHAR and special statements and phrases required for its implementation.

If the type CHAR is declared as in the example below, built-in character subtypes are available. The subtype ALPHABET contains the capital letters A through Z, and the subtype DIGIT represents the numbers 0 through 9. Subtypes ARITHMETIC, RELATIONAL, LOGICAL, PUNCTUATION, BRACKET, QUOTE, and SPECIAL are also available. To obtain all of the default character constants, simply declare

```
TYPE CHAR=;
```

Once the CHAR type is declared, all of its subtypes are available. By coding

```
TYPE CHAR = (ALPHABET=, DIGIT=,);
```

the default contents of just the ALPHABET and DIGIT subtypes can be declared. By listing the characters explicitly, the user can assign the internal ordinal numbers for each member of the CHAR type in whatever order desired.

Example:

```
TYPE CHAR =
     (NAME_SYMBOL = (DIGIT=, ALPHABET=, '_', '$'),
     DELIMITER=
          (ARITHMETIC = ('+', '-', '*', '/'),
          RELATIONAL = ('<', '=', '>'),
          LOGICAL = ('∧', '&', '|'),
          PUNCTUATION = (',', ';', ':', ' ', '.', '!', '?'),
          BRACKET = ('(', ')', '[', ']'),
          QUOTE = ('"', '''') ),
     SPECIAL = ('@', '#', '%'),
     EOL);      -- As EOL demonstrates, identifiers can
                -- be used for character constants.
```

The first character listed in the LOGICAL subtype is a "not" sign, used in SKOL as an up arrow, and the third is an "or."

This example would match exactly the default declaration (obtained by TYPE CHAR=;) if the EOL were omitted, and if DIGIT and ALPHABET were fully specified.

EOL has an internal value that depends on its position in the declaration of CHAR, but it has no external value. If an attempt is made to write it with WRITESTRING or the C format (described below) in an OUTPUT statement, the program aborts with the error message,

```
HOLE IN OUTPUT CHARACTER TABLE AT index;
     DISCOVERED AT LINE n.
```

The TYPE declaration should be made before CHAR is used in the program, preferably at the top of the program before the MAIN statement. When type CHAR is used, the first statement executed in the MAIN segment should be

```
CHAR_SETUP;
```

which uses a library routine to initialize a character translation table. Any MAIN segment, function, or subroutine that refers to the CHAR type must declare

```
CHAR_COMMON;
```

All of the subroutines required to implement the CHAR type are in $FTLIB.

In order to modify a subtype of CHAR, all character constants of that subtype must be listed explicitly. For instance, to move the question mark from PUNCTUATION to SPECIAL, all the characters in each of the two subtypes must be explicitly listed. Subtypes not affected by the change (NAME_SYMBOL, ARITHMETIC, RELATIONAL, LOGICAL, BRACKET, and QUOTE) may be declared using the default mode. Coding

```
NAME_SYMBOL=,
```

for instance, declares all 38 characters that are specified in the built-in subtype.

Characters must be read with the READSTRING statement or translated with ORD after being read normally if the CHAR type is declared. They must be written with either the WRITESTRING statement or the OUTPUT statement using the C format when CHAR is declared.

ORD, CHR

If the CHAR type is declared, the special functions ORD and CHR can be used to associate external characters and their internal integer representations exactly as in Pascal.[10] ORD takes a character and returns its ordinal value. CHR takes an integer and returns a character. For instance, assuming the default values for CHAR have been declared, ORD('B') yields the integer 12. CHR(44) yields the ASCII code for an equal sign, while CHR(12) yields the ASCII code for the letter B.

VALUE

If the CHAR type is declared and includes the DIGIT subtype, VALUE returns the integer value of a character expression. The character expression must resolve to a character belonging to the DIGIT subtype. The form is:

VALUE (*digit-exp*)

digit-exp An expression that resolves to the value of a character between 0 and 9

For example, if

```
VALUE(SOC_SEC_NO(I))
```

is coded, the Ith character of the string or array SOC_SEC_NO must contain a decimal digit for the expression to be valid.

NOTE

Never use VALUE as an array or function name if the DIGIT subtype of type CHAR is declared. The macro translator may confuse it with the built-in function just described, and unwanted results will follow.

READSTRING, WRITESTRING

String I/O requires special provisions when type CHAR is declared, since the conversion between external character format and internal ASCII representation is defined by the programmer's declaration, not by FORTRAN. READSTRING and WRITESTRING accomplish that conversion when used in the following form:

```
{READSTRING} {(file)} string {(index..index)} ;
{WRITESTRING}
```

file	A local dataset name, a FORTRAN unit number, or an expression that resolves into one or the other
string	An identifier declared as a string
index	An integer within the bounds of the maximum length of the string

READSTRING does not set the length of the string upon reading it. LENGTH(*string*) must be set by the programmer through assignment. The default index range for WRITESTRING is 1..LENGTH(*string*); for READSTRING it is 1..SIZE(*string*). If (*file*) is omitted, the standard input or output dataset name ($IN or $OUT) is used by default.

EXAMPLE

```
STRING: CARD(81);
READSTRING (MYFILE) CARD(2..81);
LENGTH(CARD) = 81;
CARD(1) = '1';          -- For page eject.
WRITESTRING CARD;       -- Default range is 1..LENGTH(CARD).
```

As the above example illustrates, the first character is used for carriage control on output. A 1 in the first output column advances the carriage to the first line of the next page, a 0 advances two lines, a plus sign causes no advance, and any other character (including a blank) advances one line.

In the above example, the first 80 characters in a record of MYFILE are converted to internal format and stored in positions 2 through 81 of a string named CARD, and '1' is stored in position 1. After a page eject, the characters in positions 2 through 81 are listed on $OUT.

THE C FORMAT

The :C format descriptor, which can be specified in an OUTPUT or a FORMAT statement, converts an internal representation for a character to external form if type CHAR is declared. The ::C specification performs the conversion, prints the variable name along with its value, and places an apostrophe on either side of the value.

EXAMPLE

```
OUTPUT (CH::C);
```

The output, assuming the CHAR type has been declared and the variable CH has been assigned a value of G, is as follows:

```
CH = 'G';
```

SKOL KEYWORDS

C

ACTIVATE, AFTER, AND, AT

BASE, BEFORE, BEGIN, BLOCKDATA, BOOLEAN, BTAB, BUFFER_LENGTH, BY, BYTE

CALL, CATENATE, CDIR, CHAR, CHARACTER, CHR, COMMON, COMPLEX, CONDITIONAL, CONSTANT, COROUTINE

DATA, DEBUG, DECODE, DECR, DEFINE, DELETE, DISABLE, DO, DOUBLE

ELSE, ELSEIF, ENABLE, END, ENDBLOCKDATA, ENDCOROUTINE, ENDDO, ENDFOR, ENDFUNCTION, ENDIF, ENDLINK, ENDLOOP, ENDMAIN, ENDRECORD, ENDREPEAT, ENDROUTINE, ENDSUBROUTINE, ENDTABLE, ENDTEMPLATE, ENDUNTIL, ENDWHEN, ENDWITH, END_OF_FILE, ENTRY, EQUIVALENCE, EQV, ERR, EVEN, EXECUTE, EXECUTE_ORD, EXIT, EXTERNAL

FALSE, FIRST, FOR, FORMAT, FREE, FROM, FUNCTION

GLOBAL

IF, IN, INCR, INPUT, INSERT, INTEGER, IS

LAST, LENGTH, LINK, LIST, LOCAL, LOG, LOGIC, LOGICAL, LOOP, LTAB

MACRO, MAIN, MAKEAVAIL, MOVE

NEW, NIL, NOLIST, NOT, NOTRACE, NOVALIDATE, NUMERIC

ODD, ONTO, OR, ORD, OUTPUT

PAGE, PARAMETER, POINTER, PROCESS

READ, READSTRING, REAL, RECORD, RECURSIVE, REMARK, REPEAT, REPLACE, RETURN, ROUTINE

SIZE, SKIP, SMALL, STACK_FRAME, START, STRING, SUBROUTINE, SUSPEND

TABLE, TEMPLATE, THEN, TIME, TIMES, TO, TRACE, TRUE, TYPE

UNTIL, USING

VALIDATE, VALUE

WHEN, WHILE, WITH, WORD, WRITE, WRITESTRING

XOR

STRUCTURED TEXT

D

As a special case of conditional code generation, SKOL provides an alternative way to specify comments. Any source text that occurs within the special bracketing character strings <<: and >> is treated as *structured text*. Structured text has special syntax rules that are a mixture of the rules for SKOL and English and can be used to express long comment strings or to include design specifications and very high-level code in a program.

Structured text, unlike conditional code, cannot be enabled to generate FORTRAN code because the conditional-code prefix (usually found between << and :) is missing. SKOLREF, the cross-reference listing generator, requires that the three characters <<: and the final two characters >> not be separated by spaces.

SKOL automatically indents structured text according to the rules for standard SKOL control structures, if the appropriate keywords (IF, LOOP, REPEAT, UNTIL, etc.) are used to begin structured text statements that end in colons, like actual SKOL code. Statements that do not introduce a SKOL structure must end in periods. Any punctuation is allowed in a statement, including extra colons and periods, as long as a period is not followed immediately by one of SKOL's structure-defining keywords.

Within structured text, generation of both FORTRAN code and new macros is suppressed. Most of the error messages that occur as a result of matching an English word in the structured text are also suppressed. However, messages about structural errors are not suppressed.

Structured text can be read like ordinary English if the following requirements are met:

- Periods must be used instead of semicolons to terminate statements. Colons have their standard SKOL meaning.

- A period immediately preceding a right parenthesis is exchanged with the parenthesis so the two characters .) can terminate the statement.

- If two or more periods occur in a row, each period is treated as an ordinary character. The statement is not terminated.

- Apostrophes are ignored so that contractions and the possessive case can be used. (Literal strings are stripped of enclosing apostrophes, so the first word in the string should not be a keyword such as IF or UNTIL if the string begins a sentence.)

The rules for writing structured text are:

- Bracket any structured text with the characters <<: and >>.

- Use ordinary comment syntax within structured text, if desired.

- Indentation for structured text is not necessary; as with SKOL code, the preprocessor ignores the indentation and supplies its own.

- Do not put any structured text on the same line with the introductory <<:. If a sentence is needed on the <<: line, use an ordinary comment, as in:

  ```
  <<:  --  Heading for structured text below.
  ```

- Use sentences beginning with SKOL keywords such as IF, LOOP, REPEAT, WHEN, and UNTIL, and ending with a colon, to define the structure of the structured text. Structural keywords are used as in SKOL code, except almost any text can appear between the keyword and the colon. Exceptions occur in the UNTIL and WHEN statements:

  WHEN statements, introducing a scalar case structure, must contain an inner colon and end with the word is:

  ```
  When xxxxxxx:xxx is:
  When xxxxxxx:[m..n] is:
  ```

  UNTIL statements must not contain an OR unless a THEN statement is at the end of the UNTIL structure. An OR inside parentheses, however, is not detected by the macro-translator...

Until x or y:	Requires a THEN statement
Until x:	Does not require a THEN statement
Until x (or y):	Does not require a THEN statement

Note in the examples that keywords are lowercase, except for the initial letter of a sentence, to resemble English.

- Terminate all ordinary statements with a period or a period followed by a right parenthesis. A colon terminator is reserved for use with statements that introduce structures.

- Statements such as ENDIF and ENDLOOP must be terminated with a period. Suggestion: Use two words, as in End if. and End loop.

- Avoid beginning narrative sentences with If, Else if, Until, or While. (When, Do, For, Loop, Repeat, and Then can be used to begin sentences.)

- Use punctuation such as ... or -- or ? to end a sentence when the next sentence is a continuation of the statement.

- If desired, reduce the number of lines needed to express an IF...ENDIF structure by using forms such as:

```
Does aaa exist?
  Then do bbb.
Is xxx true?  Then do yyy.
```

  Because is and does are not keywords, indentation results from the then sentence following a ? instead of a period.

- To introduce tabular information without using a WHEN statement, end the introductory paragraph with ..., --, or a colon:

```
Sentence one.  Sentence two.  Sentence three...
  Table entry 1
  Table entry 2
  Last table entry.
```

- Any punctuation is allowed within a statement, including extra colons and periods, as long as a colon or period is not followed by one of SKOL's structure-defining keywords. (Violating this rule causes structure-related error messages.)

- Use the special structured text statements

```
Entry:
Exit:
Logic:
```

  as headers for introductory comments in routines, subroutines, and functions. (Do not use them at the top of the program where the nesting level is zero.) Entry and exit conditions should be given in some form for every module. Once these conditions are verified for a specific module, they remove the need to re-examine the code of the module whenever an outer module fails to perform as expected.

Following are three typical applications of structured text.

```
At 17:46:13 on 11/04/80:  Cray-1 SKOL macro translation                    Translator revision level 1.10

  1.     0
  2.     0   <<:  --  Preliminary design for pass 1 of an UPDATE-mod resequencer.
  3. X   0
  4. X   0   Loop:
  5. X   1     Read a record from PREDISTRIBUTION_SOURCE.
  6. X   0   While not end_of_file (PREDISTRIBUTION_SOURCE):
  7. X   1     If the input record begins with "*NEWDECK " (a phony DECK directive):
  8. X   2       Set DECKNAME = the 1..7 character token that follows "*NEWDECK ".
  9. X   2       Write a true DECK directive to RESEQUENCING_INPUT that consists of
 10. X   2         (1) the characters "*DK " in columns 1..4
 11. X   2         (2) the DECKNAME in columns 5..11.
 12. X   2       Set CARD_NUMBER = 1.  (The *DK card itself will receive the number 1.)
 13. X   1     Else:
 14. X   2       Increment CARD_NUMBER.
 15. X   2       Set NEW_SEQUENCE_NUMBER = DECKNAME & "." & CARD_NUMBER's external form.
 16. X   2       Extract OLD_SEQUENCE_NUMBER from columns 73..88 of the input record.
 17. X   2       Write a record to TRANSLATION_LIST consisting of
 18. X   2         (1) the OLD_SEQUENCE_NUMBER left-adjusted in columns 1..16
 19. X   2         (2) the NEW_SEQUENCE_NUMBER left-adjusted in columns 17..32.
 20. X   2       Copy the input record to RESEQUENCING_INPUT.  (The old sequence number
 21. X   2         can be omitted; UPDATE will ignore it anyway.)
 22. X   1     End if.
 23. X   0   End loop.
 24. X   0
 25. X   0   >>

 35933 words (38.6%) used for SKOLTXT macros

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - %
```

```
931.      0
932.      0   SUBROUTINE WCSTRING: WRK(100);
935.      1     INTEGER: F, CSTRING(0..0), CWORD, CHARS, INDEX, LASTCHAR, TRIMMODE;
936.      1     LOGICAL: ADJLEFT, ADJRIGHT;
937.      1     EXTERNAL: WRITEC, WRITECP;
938.      1
939.      1     <<:  --  Write the characters in CSTRING to file F (unless F = '0'L).
940. X    1
941. X    0   Entry:
942. X    1
943. X    1     If there is only one argument (F):
944. X    2       F specifies a new mode.  Values for F are in the range [0..3]:
945. X    2         0 (Default)     -  All characters up to the first null are to be output
946. X    2         1 (TRIM_RIGHT)  -  Trailing blanks are not to be output
947. X    2         2 (TRIM_LEFT)   -  Leading blanks are not to be output
948. X    2         3 (TRIM_ENDS)   -  Neither leading nor trailing blanks are to be output.
949. X    1     Else: (There are two arguments.)
950. X    2       F contains the local dataset name or FORTRAN unit number of the file to
951. X    2         be written.  When F is '0'L, the call to WCSTREAM is a no-op.
952. X    2       CSTRING is a single word or array containing a packed character string
953. X    2         which is left-adjusted and terminated by a null character.  When a
954. X    2         string contains exactly 8*n characters, n+1 words are required.  The
955. X    2         additional word provides the null terminator.  CFT literal strings
956. X    2         with an L suffix are guaranteed to be so terminated.
957. X    1     End if.
958. X    1
959. X    0   Exit:
960. X    1
961. X    1     The contents of CSTRING are written to file F.  When CSTRING begins with
962. X    1       a null character, an EOR record control word is written to file F.
963. X    1
964. X    1     >>
965.      1
```

```
602.     1
603.     1     ROUTINE PRINT_SYMBOLS:
604.     2
605.     2       <<:  --  Output the contents of T_SYM to the L dataset.
606. X   2
607. X   1     Entry:
608. X   2
609. X   2       HEAD_PTR points to the first record in the sorted list.
610. X   2       ACTUAL_COUNT indicates the number of records in the list.
611. X   2
612. X   1     Exit:
613. X   2
614. X   2       All the symbol entries have been printed.
615. X   2
616. X   1     Logic:
617. X   2
618. X   2       Link for HEAD_PTR = HEAD_PTR to NIL using SYM_:
619. X   3         Fix the location of the SYMARRAY template at HEAD_PTR.
620. X   3         If the current symbol is the same as the one that was output
621. X   4             the last time:
622. X   4           Erase the current symbol name.
623. X   3         Else:
624. X   4           Save the new name as the last symbol output.
625. X   4           Tally the symbol name as a unique new name.
626. X   3         End if.
627. X   3         Print the entry and all the line numbers that reference it.
628. X   3         Increment the tally of symbol entries output.
629. X   2       End link.
630. X   2
631. X   2       If the output tally doesn't agree with the sort tally:
632. X   3         Write an error message to the log file.  (But don't abort, since we're
633. X   3           done anyhow.)
634. X   2       End if.
635. X   2
636. X   2       >>
637.     2
```

ERROR MESSAGES

E

The following listing was produced by the SKOL macro translator and illustrates most of the diagnostic messages that are built into the standard macros. The notes at the end of the listing explain some of the less obvious messages. In the source text, SKOL and CFT keywords are in lowercase, while the names invented by the programmer are in uppercase.

```
At 17:57:27 on 08/07/80:  Cray-1 SKOL macro translation

  1.          %E99
  2.          %I-
  3.      0
  4.      0   --- TEST OF SKOL DIAGNOSTICS
  5.      0
  6.      0   type char=(alphabet=('A','B','C'),EOL,TAB,' ');
  7.      0
  8.      0   type COLOR=

*------------- ERROR   1:  BAD SUBTYPE SYNTAX 'NEG=NUMERIC(-1,0)'
  9.      0      (NEG=numeric(-1,0),REDS=(RED,PINK),GREEN,BLUE);
 10.      0

*------------- ERROR   2:  BAD CONSTANT DEFINITION 'BAD'
 11.      0   constant: PI=3.14,BAD,NDIM=2;
 12.      0
 13.      0   main (MSGTEST):
 14.      1
 15.      1      char: CH;

*------------- ERROR   3:  UNDIMENSIONED STRING 'STR3'
 16.      1      string: STR1(20),STR3,STR4(10);
 17.      1      COLOR: HUE;

*------------- ERROR   4:  UNEXPECTED COLON DISCARDED
 18.      1      SCAL1: SC1,SC2;
 19.      1      COLOR C1,C2;
 20.      1      record WHAT (N):

*------------- ERROR   5:  BAD TYPE 'COLOR' IN FIELD LIST
 21.      2          COLOR: C1;
 22.      2          char: CH1;
 23.      2          pointer: LL,RL;
 24.      1      endrecord;
 25.      1      pointer to WHAT: P, Q(4);

*------------- ERROR   6:  RECORD 'NONO' UNDECLARED
 26.      1      pointer to NONO: P2;
 27.      1      process XYZ = (AA,BB);
 28.      1      recursive (100): X1, X2(*);
 29.      1      integer: X(5);
```

```
*------------- ERROR  7:  MISSING PARENTHESES AROUND 'FOR' CLAUSE
 30.     1      data: for I=3 to 5: X(I) /3,4,5/;
 31.     1
 32.     0   begin:

*------------- ERROR  8:  RECORD 'NONO' UNDECLARED
 33.     1      makeavail NONO;

*------------- ERROR  9:  UNDECLARED COROUTINE OR PROCESS 'X1'
 34.     1      start XYZ at X1;
 35.     1      char_setup;
 36.     1      delete STR1;
 37.     1      replace STR1(1..10) by 'A';
 38.     1      macro MAC1 (KEY1=0, KEY2) = 'ZZ = KEY1; Z2(KEY2) = KEY1+1';
 39.     1
 40.     1      until S1 or S2 or S3:
 41.     2         if P: S4; endif;

*------------- ERROR 10:  UNDECLARED STRING 'SC1'
 42.     2         delete SC1;

*------------- ERROR 11:  UNDECLARED STRING 'STR2(1..3)'
*------------- ERROR 12:  UNDECLARED STRING 'STR2(1..3)'
*------------- ERROR 13:  UNDECLARED STRING 'STR2(1..3)'
 43.     2         catenate 'B' onto STR2(1..3);
 44.     2         MAC1 (KEY2 = 3);

*------------- ERROR 14:  BAD KEYWORD PARAMETER '3.2'
*------------- ERROR 15:  BAD KEYWORD PARAMETER '4'
*------------- ERROR 16:  OBLIGATORY PARAMETER 'KEY2' MISSING
 45.     2         MAC1 (3.2, 4);
 46.     2         loop:

*------------- ERROR 17:  UNDEFINED FORMAT 'PERSON'
 47.     3            I = I+1;  read ($in, PERSON) LIST;
 48.     3            if A = 0: repeat 7 times:

*------------- ERROR 18:  UNCLOSED REPEAT...TIMES FOUND AT ENDLOOP
*------------- ERROR 19:  UNCLOSED IF FOUND AT ENDLOOP
*------------- ERROR 20:  MISSING 'WHILE'
 49.     2         endloop;

*------------- ERROR 21:  BAD SYNTAX AT '...I = 10 BY - 1 DO BEGIN END'
 50.     2         for I = 10 by -1 do begin end;  S2;
 51.     1      then:

*------------- ERROR 22:  SITUATION LABEL 'S5' NOT DECLARED
*------------- ERROR 23:  UNEXPECTED '$'
*------------- ERROR 24:  BAD SYNTAX
*------------- ERROR 25:  UNDECLARED STRING ''Not a string''
*------------- ERROR 26:  UNDECLARED STRING ''Not a string''
 52.     2         [S5]:  writestring ($MYFILE) 'Not a string';
 53.     2         [S1,S3]:

*------------- ERROR 27:  UNDECLARED STRING 'STR2'
 54.     3            J = length(STR2);
```

```
*------------- ERROR  28:  BAD SYNTAX AT '...I = 1 TO 10 A = 0'
*------------- ERROR  29:  UNCLOSED SITUATION CASE GROUP FOUND AT ENDDO
*------------- ERROR  30:  UNCLOSED THEN FOUND AT ENDDO
*------------- ERROR  31:  MISSING 'DO'
 55.     0            do I = 1 to 10 A = 0; enddo;

*------------- ERROR  32:  UNDECLARED STRING 'STR2'
 56.     0            if size(STR2) = 0:

*------------- ERROR  33:  UNDECLARED CHARACTER '?'
 57.     1               STR1(2) = '?';
 58.     0            else:

*------------- ERROR  34:  UNDECLARED STRING 'STR2'
 59.     1               delete STR2;

*------------- ERROR  35:  UNDECLARED CHARACTER '+'
 60.     1               insert '+' after STR1(size(STR1));

*------------- ERROR  36:  UNDECLARED STRING 'STR4'
*------------- ERROR  37:  UNDECLARED STRING 'STR4'
*------------- ERROR  38:  UNDECLARED STRING 'STR4'
*------------- ERROR  39:  UNDECLARED CHARACTER 'D'
*------------- ERROR  40:  CHAR 'EOL' CANNOT BE OUTPUT
 61.     1               catenate 'D' onto STR4; output (EOL:C);

*------------- ERROR  41:  MISSING PARENTHESES AROUND 'FOR' CLAUSE
 62.     1               output (for I=1 to 10: chr(STR4(I)): 10R1);

*------------- ERROR  42:  UNDECLARED SCALAR TYPE OR SUBTYPE 'SCAL2'
*------------- ERROR  43:  UNDECLARED SCALAR TYPE OR SUBTYPE 'SCAL2'
*------------- ERROR  44:  UNDECLARED SCALAR TYPE OR SUBTYPE 'SCAL2'
 63.     1               when I:SCAL2 is:
 64.     2                  [2]:

*------------- ERROR  45:  UNDECLARED SCALAR TYPE OR SUBTYPE 'REDSPECTRUM'
*------------- ERROR  46:  UNDECLARED SCALAR TYPE OR SUBTYPE 'REDSPECTRUM'
*------------- ERROR  47:  UNEXPECTED COLON DISCARDED
 65.     3                     if C1 in REDSPECTRUM: A := 0; endif;
 66.     2                  [else]:
 67.     3                     when CH:alphabet is:
 68.     4                        ['A','D']:

*------------- ERROR  48:  UNCLOSED SCALAR CASE GROUP FOUND AT WHILE
*------------- ERROR  49:  UNCLOSED WHEN...IS FOUND AT WHILE
*------------- ERROR  50:  UNCLOSED SCALAR CASE GROUP FOUND AT WHILE
*------------- ERROR  51:  UNCLOSED WHEN...IS FOUND AT WHILE
*------------- ERROR  52:  UNCLOSED ELSE FOUND AT WHILE
*------------- ERROR  53:  MISSING 'LOOP'
 69.     0                     while B <= N: A = 1;

*------------- ERROR  54:  BAD FIELD 'NEXT'
 70.     0                     link for P = Q(2) to NIL using NEXT:

*------------- ERROR  55:  UNCLOSED LINK FOUND AT ENDWHEN
*------------- ERROR  56:  MISSING 'WHEN'
 71.     0               endwhen;
```

```
*------------ ERROR  57:  MISSING 'IF'
 72.     0            endif;

*------------ ERROR  58:  MISSING 'UNTIL'
 73.     0      enduntil;
 74.     0      until A or B:

*------------ ERROR  59:  WRONG NUMBER OF PARAMETERS FOR 'X2'
 75.     1         execute X2 (1, 0.4);

*------------ ERROR  60:  BAD 'FOR' PHRASE
 76.     1         for A(I) to N: K = 2;

*------------ ERROR  61:  BAD 'DO' PHRASE
 77.     2         do 7 times:
 78.     3            for I = 10 by -1:

*------------ ERROR  62:  UNCLOSED FOR...BY FOUND AT ENDDO
*------------ ERROR  63:  UNCLOSED UNTIL...THEN FOUND AT ENDDO
*------------ ERROR  64:  MISSING 'DO'
 79.     1         enddo;

*------------ ERROR  65:  MISSING 'UNTIL'
 80.     0      enduntil;
 81.     0

*------------ ERROR  66:  ROUTINE 'D' DECLARED BEFORE AN EXECUTE
 82.     0      routine D:

*------------ ERROR  67:  BAD SYNTAX AT '...J = 0'
*------------ ERROR  68:  BAD SYNTAX AT '...P < 1'
 83.     1         repeat J = 0;  B = 3;  until P < 1;

*------------ ERROR  69:  BAD SYNTAX AT '...TRUE THEN X = Y ELSE X = Z'
 84.     1         if true then X = Y else X = Z;

*------------ ERROR  70:  BAD SYNTAX AT '...X , XX'
 85.     1         output X,XX;

*------------ ERROR  71:  ILLEGAL BACKWARD ROUTINE REFERENCE 'D'
 86.     1         execute D;

*------------ ERROR  72:  BAD FIELD 'ZZ'
 87.     1         ^P.ZZ = ^Q(2).CH1;
 88.     1         with Q(K):

*------------ ERROR  73:  BAD FIELD 'XX'
 89.     2            output (^.CH1:C, ^.XX);
 90.     1         endwith;

*------------ ERROR  74:  BAD SYNTAX
*------------ ERROR  75:  UNEXPECTED COLON DISCARDED
*------------ ERROR  76:  UNEXPECTED COLON DISCARDED
 91.     1         output (A:, Z:;
 92.     1         activate XYZ;
 93.     1
```

```
*------------- ERROR  77:  UNCLOSED ROUTINE FOUND AT COROUTINE
  94.     1       coroutine AA:

*------------- ERROR  78:  BAD SYNTAX
*------------- ERROR  79:  POINTER 'P3' UNDECLARED
  95.     1          new P3; free Q(1);
  96.     1          N = 0; activate BB;

*------------- ERROR  80:  UNTREATED SCALAR VALUE 'GREEN'
*------------- ERROR  81:  UNTREATED SCALAR VALUE 'BLUE'
  97.     1          when HUE:COLOR is:  [REDS]:  endwhen;
  98.     0       endcoroutine;
  99.     0

*------------- ERROR  82:  ROUTINE 'EFG' DECLARED BEFORE AN EXECUTE
 100.     0       routine EFG:
 101.     1          incr K;

*------------- ERROR  83:  POINTER 'SS3' UNDECLARED
 102.     1          ^SS3.LL = NIL;
 103.     1          suspend XYZ;
 104.     0       endroutine;
 105.     0

*------------- ERROR  84:  WRONG NUMBER OF PARAMETERS FOR 'X1'
 106.     0       routine X1(P2) local(Z):

*----------- WARNING   1:  BAD VALIDATE PARAMETER '+ WHEN' IGNORED
 107.     1          validate (-loop, +when);

*------------- ERROR  85:  UNDECLARED STRING 'CARD'
*------------- ERROR  86:  UNDECLARED STRING 'CARD'
 108.     1          readstring CARD;

*------------- ERROR  87:  UNDECLARED SCALAR TYPE OR SUBTYPE 'DIGIT'
*------------- ERROR  88:  UNDECLARED SCALAR TYPE OR SUBTYPE 'DIGIT'
 109.     1          if (CH in DIGIT):
 110.     2             C = value(CH);
 111.     1          endif;

*------------- ERROR  89:  BAD SYNTAX IN SCALAR RANGE 'EOL,TAB'
*------------- ERROR  90:  BAD SYNTAX IN SCALAR RANGE 'EOL,TAB'
 112.     1          if CH in [EOL, TAB]:
 113.     2             C = ord(CH);
 114.     1          endif;
 115.     1          execute X1;
 116.     1          execute A;

*------------- ERROR  91:  UNDECLARED COROUTINE OR PROCESS 'EFG'
 117.     1          activate EFG;
 118.     0       endroutine;
 119.     0

*------------- ERROR  92:  ROUTINE 'D' DECLARED BEFORE AN EXECUTE
 120.     0       routine D:
```

```
*------------- ERROR  93:  PARAMETER OR LOCAL OUTSIDE SCOPE
*------------- ERROR  94:  PARAMETER OR LOCAL OUTSIDE SCOPE
 121.     1          Z = P2;

*------------- ERROR  95:  ARCHAIC FORM '.LT.'
 122.     1          if (A .LT. B):  A = B;  endif;

*------------- ERROR  96:  UNDECLARED STRING 'STR2'
*------------- ERROR  97:  UNDECLARED STRING 'STR2'
*------------- ERROR  98:  UNDECLARED STRING 'STR2'
 123.     1          move STR1 to STR2(1..3);

*------------- ERROR  99:  INCORRECT PLACE FOR STRING LENGTH
*------------- ERROR 100:  INCORRECT PLACE FOR STRING LENGTH
 124.     1          replace STR1(3|..|0) by CH;

*------------- ERROR 101:  UNDECLARED STRING 'STR3'
*------------- ERROR 102:  UNDECLARED STRING 'STR3'
*------------- ERROR 103:  UNDECLARED STRING 'STR3'
 125.     1          delete STR3(4..);

*------------- ERROR 104:  UNDECLARED STRING 'OUTLIN'
*------------- ERROR 105:  UNDECLARED STRING 'OUTLIN'
 126.     1          writestring OUTLIN;
 127.     0        endroutine;
 128.     0

*------------- ERROR 106:  ROUTINE OR COROUTINE 'BB' NOT DEFINED
*------------- ERROR 107:  ROUTINE OR COROUTINE 'X2' NOT DEFINED
*------------- ERROR 108:  ROUTINE OR COROUTINE 'A' NOT DEFINED
*------------- ERROR 109:  LABEL LEFT ON STACK
*------------- ERROR 110:  LABEL LEFT ON STACK
*------------- ERROR 111:  LABEL LEFT ON STACK
*------------- ERROR 112:  LABEL LEFT ON STACK
*------------- ERROR 113:  LABEL LEFT ON STACK
*------------- ERROR 114:  LABEL LEFT ON STACK
*------------- ERROR 115:  LABEL LEFT ON STACK
 129.     0   endmain;


 35698 words (38.4%) used for SKOLTXT macros
  1547 words ( 1.7%) used for global macros
  2678 words ( 2.9%) used for local macros

   1 macro translation warning

 115 macro translation errors

% - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - %
```

ERROR 1: A subtype cannot be NUMERIC, so NEG is disregarded. However, the rest of the type statement is processed without error.

ERROR 2: No value is assigned to BAD.

ERROR 3: No length is given for STR3.

ERROR 4: SCAL1 is unknown as a type and so is unmatched. The next line is in error too, since COLOR is not followed by a colon, but the error is not detected by SKOL and the statement is passed on to FORTRAN.

ERROR 5: User-defined scalar types are not allowed in a RECORD.

ERROR 7: The correct form is (FOR I = 3 TO 5: X(I)).

ERRORS 11-13: The CATENATE statement generates three references to LENGTH(STR2(1..3)), and each one produces a message.

ERRORS 14,15: A macro defined by a MACRO statement does not allow positional parameters.

ERROR 16: The parameter KEY2 has no default value (see line 38). The caller must assign a value to it.

ERRORS 18,19: The IF structure and the REPEAT structure begun on line 48 are not properly terminated.

ERROR 20: The LOOP structure on lines 46 through 49 contains no WHILE statement.

ERROR 21: The FOR statement requires a colon.

ERROR 22: S5 must be named in the UNTIL statement on line 40.

ERRORS 23,24: $MYFILE is not defined as a constant.

ERRORS 25,26: WRITESTRING does not accept literal strings.

ERROR 28: The DO statement requires a colon.

ERRORS 29,30: Both errors are caused by a missing ENDUNTIL.

ERROR 31: The ENDDO on line 55 caused this error because the corresponding DO statement was ignored.

ERROR 33: '?' is not among the characters declared on line 6.

ERRORS 42-44: The WHEN statement generates three references to the scalar type, which doesn't exist in this case.

ERRORS 48-53: Four missing statements (ENDWHEN, ENDWHEN, ENDIF, LOOP) produce these six messages. Each missing ENDWHEN causes two messages.

ERROR 54: P has been declared as a pointer to record type WHAT on line 25, but NEXT is not among the fields declared for WHAT on lines 20-24.

ERRORS 56-58: These are spurious messages caused by popping successive characters off the macro symbol stack in an attempt to find a LOOP corresponding to the WHILE on line 69.

ERROR 59: X2 is declared on line 28 as a recursive routine with only one parameter.

ERROR 62: An ENDFOR is needed to match the FOR on line 78.

ERROR 63: THEN, [A], [B], and ENDUNTIL are needed to match the UNTIL on line 74.

ERROR 64: The DO statement on line 77 is incorrectly formed and is not recognized as matching the ENDDO on line 79.

ERROR 65: A spurious message caused by the error on lines 77-79.

ERROR 69: The THEN in line 84 should be a colon. The rest of the line is badly formed as well.

ERROR 70: OUTPUT's I/O list must be enclosed in parentheses.

ERROR 72: ZZ is not a field of the record type WHAT to which the pointer P refers.

ERROR 73: Λ.XX is an abbreviation for Λ Q(K).XX. Q is an array of pointers to WHAT, but XX is not a field in WHAT.

ERROR 77: An ENDROUTINE is needed to match the ROUTINE statement on line 82.

ERRORS 80,81: An [ELSE] or [GREEN,BLUE] case label is needed on line 97. [REDS] is a synonym for [RED,PINK].

ERROR 84: The recursive routine is declared without parameters on line 28, but defined with one parameter on line 106.

WARNING 1: Only -, not +, is allowed as a prefix to a VALIDATE parameter. This warning does not prevent execution of any following job steps.

ERRORS 87,88: DIGIT could have been defined by specifying DIGIT=, on line 6, or by replacing line 6 with TYPE CHAR=;. Because DIGIT is undeclared, VALUE (on line 110) is passed on to FORTRAN without change.

ERRORS 89,90: The IN expression on line 112 generates two references to the scalar range. The comma must be replaced by two periods.

ERROR 92: All references to routines must be forward references.

ERRORS 93,94: Z and P2 are respectively a local variable and a parameter of recursive routine X1, defined on lines 106-118. The occurrences on line 121 are outside the valid scope.

ERROR 95: The SKOL relational operators <, >, =, <>, <=, and >= must be used. The parentheses are optional.

ERRORS 99,100:
The presence of two vertical bars in the REPLACE statement on line 124 prevents the string expression from being matched, so that both of them are treated merely as isolated characters.

ERRORS 106-108:
No definitions were encountered for coroutine BB, declared on line 27, for recursive routine X2, declared on line 27, or for routine A, referenced on line 116.

ERRORS 109-115:
The label stack still contains FORTRAN labels that would have been used in generating code for various missing END... statements.

The following SKOL program finds prime numbers. SKOL keywords are represented in lowercase, and names chosen by the programmer are in uppercase.

```
 1.   0   -- SKOL PROGRAM TO GENERATE PRIMES
 2.   0
 3.   0
 4.        %C80          -- ALLOW ALL 80 COLUMNS OF THE INPUT TO BE SCANNED.
 5.   0
 6.   0   main (SPRTEST):
 7.   1
 8.   1     constant: NPRIME = 100;
 9.   1     integer: PRIME (NPRIME), TRIAL, ITEST, NXTPRM, IPR;
10.   1
11.   0   begin:
12.   1     PRIME (1) = 2;
13.   1     PRIME (2) = 3;
14.   1     do IPR = 3 to NPRIME:
15.   2       until FOUND_NEXT_PRIME:
16.   3         do TRIAL = PRIME(IPR-1)+2 by 2:
17.   4           until IS_PRIME or IS_COMPOSITE:
18.   5             do ITEST = 2 to IPR-1:
19.   6               if mod (TRIAL,PRIME (ITEST)) = 0:
20.   7                 IS_COMPOSITE;
21.   6               elseif PRIME (ITEST)**2 > TRIAL:
22.   7                 IS_PRIME;
23.   6               endif;
24.   5             enddo;
25.   5             IS_PRIME;
26.   4           then:
27.   5             [IS_PRIME]:   NXTPRM = TRIAL;  FOUND_NEXT_PRIME;
28.   5             [IS_COMPOSITE]:
29.   4           enduntil;
30.   3         enddo;
31.   2       enduntil;
32.   2       PRIME (IPR) = NXTPRM;
33.   1     enddo;
34.   1
35.   1     output ($skip, :15X, 'THE ', NPRIME:I4, 'TH PRIME IS:', PRIME(NPRIME):I8);
36.   0   endmain;

         THE  100TH PRIME IS:     541
```

BIBLIOGRAPHY

G

1 C. T. Zahn, "The SKOL Programming Language Reference Manual," (Technical Memo No. 176, Computation Research Group, Stanford Linear Accelerator Center, Stanford, Cal., 1976).

2 A. J. Cook and L. J. Shustek, "MORTRAN2, a Macro-based Structured FORTRAN Extension," Conference Digest, (IEEE COMPCON, 1975).

3 Cook and Shustek, "A User's Guide to MORTRAN2," (Technical Memo No. 165, Computation Research Group, Stanford Linear Accelerator Center, Stanford, Cal., 1975).

4 Zahn, "A User Manual for the MORTRAN2 Macro-translator," (Technical Memo No. 167, Computation Research Group, Stanford Linear Accelerator Center, Stanford, Cal., 1975).

5 Cook, "Experience with Extensible, Portable FORTRAN Extensions," SIGPLAN Notices (1976), pp. 10-17.

6 Zahn, "A Control Statement for Natural Top-down Structured Programming," Programming Symposium: Proceedings, Colloque sur la Programmation, ed. B. Robinet (Berlin: Springer-Verlag, 1974), pp. 170-80.

7 N. Wirth, "The Programming Language Pascal," Acta Informatica, 1, (1971), pp. 31-63.

8 K. Jensen and Wirth, "Pascal: User Manual and Report," Lecture Notes in Computer Science, vol. 18, ed. G. Goos and J. Hartmains, (Berlin: Springer-Verlag, 1974).

9 D. E. Knuth, "Structured Programming with GOTO Statements," Computing Surveys, vol. 6, no. 4 (1974), pp. 275-78.

10 O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming (New York: Academic Press, 1972).

11 Knuth and Zahn, "Ill-chosen use of Event," CACM, vol. 18, no. 6 (1975).

12 Wirth, Algorithms + Data Structures = Programs, (Englewood Cliffs, N.J.: Prentice-Hall, 1976).

13 P. Henderson and R. Snowdon, "An Experiment in Structured Programming," vol. 12, BIT (1972), pp. 38-53.

READERS COMMENT FORM

CRAY-1 SKOL Reference Manual

SR-0033

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

FOLD

STAPLE

READERS COMMENT FORM

CRAY-1 SKOL Reference Manual

SR-0033

Your comments help us to improve the quality and usefulness of our publications. Please use the space provided below to share with us your comments. When possible, please give specific page and paragraph references.

NAME _____

JOB TITLE _____

FIRM _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

CRAY RESEARCH, INC.

CUT ALONG THIS LINE

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY CARD

FIRST CLASS PERMIT NO 6184 ST PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

CRAY
RESEARCH, INC.

Attention:
PUBLICATIONS

1440 Northland Drive
Mendota Heights, MN 55120
U.S.A.

FOLD

STAPLE

Cray Research, Inc.
Publications Department
1440 Northland Drive
Mendota Heights, MN 55120
612-452-6650
TLX 298444