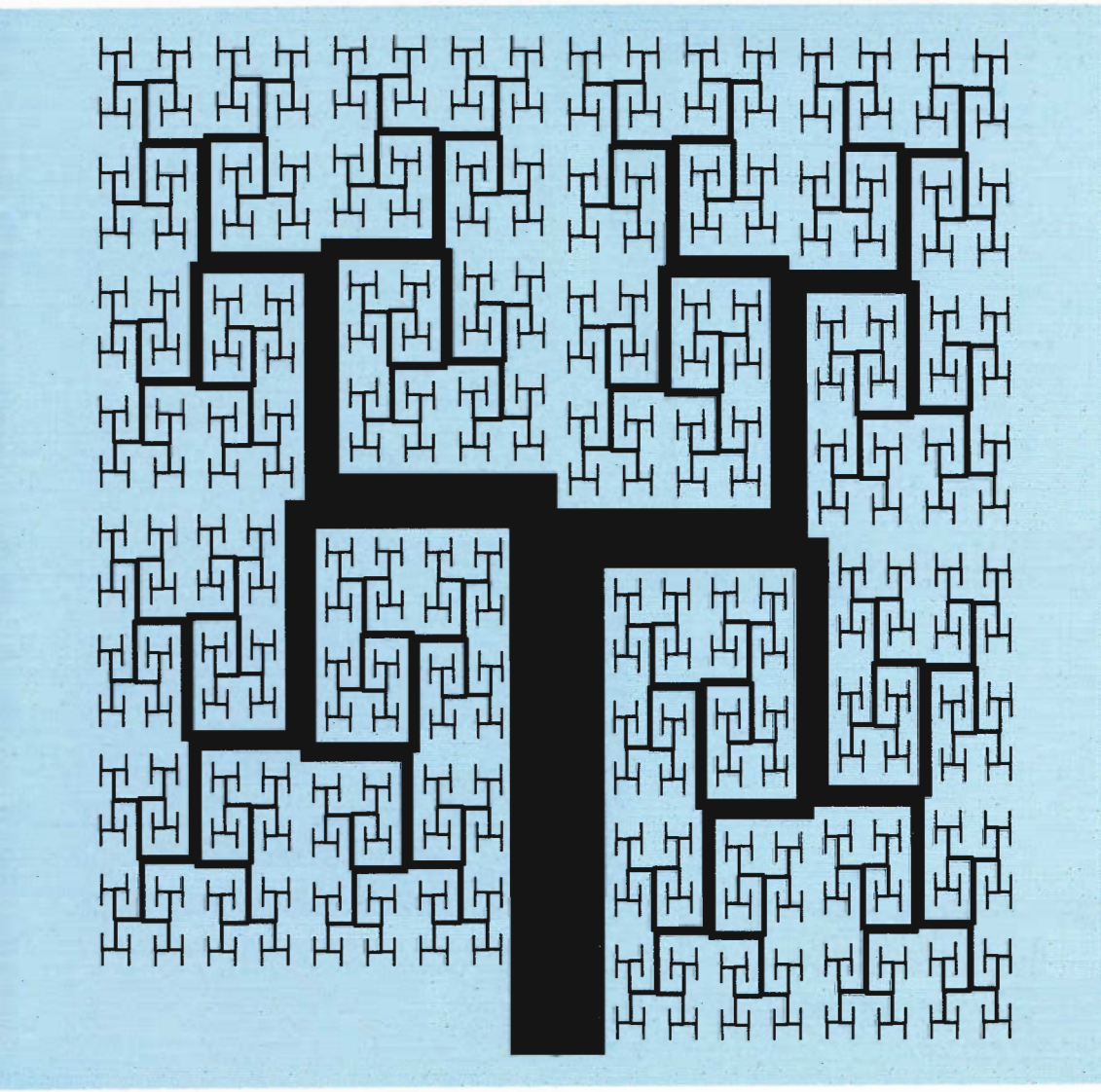

FLACC User Guide

FLACC User Guide

chion corporation

This manual corresponds to FLACC Version 1.6, November 1981.

This manual was set in 9-point Century Schoolbook and Geneva Regular type, on an Autologic APS5 phototypesetter, using the TEXTFORM® page formatting package developed at the University of Alberta, Edmonton, Alberta, Canada, T6G 2H1.

Table of Contents

1. Introduction

This manual describes the Full Language Algol 68 Checkout Compiler (FLACC). It includes information on how to prepare, compile, and execute programs written in Algol 68, using FLACC.

Algol 68 is a language designed by Working Group 2.1 of the International Federation for Information Processing (IFIP WG2.1), to be a successor to Algol 60. Its original definition was completed in 1968, and a revised definition appeared in 1974. The language is similar to, but not completely compatible with, Algol 60, Algol-W, and Burroughs Extended Algol. Algol 68 is a general purpose language, and has facilities comparable to those of PL/I.

Algol 68 is formally specified by the *Revised Report on the Algorithmic Language Algol 68* by A. van Wijngaarden et al. (Editors), published in Acta Informatica Volume 5, Numbers 1-3 (January 1975), and also in ACM SIGPLAN Notices Volume 12, Number 5 (May 1977). Any references to the above document are written as RR, possibly followed by a section number.

Algol 68 is more informally described in the *Informal Introduction to Algol 68* by Lindsey and van der Meulen, published by Elsevier North-Holland, and in *A Practical Guide to Algol 68* by Pagan, published by John Wiley & Sons.

FLACC runs on IBM 370 or equivalent machines.

FLACC was developed in consultation with Dr. Barry Mailloux of the Department of Computing Science at the University of Alberta in Edmonton. Chion received considerable support from the U of A Department of Computing Services.

1.1. How to Use this Guide

The most important section is Section 8: How to Run FLACC. Read this carefully before trying to use the system. Pay special attention to the parameters.

Section 11, Output from the System, describes the output you will get back from FLACC. It is best read with a listing at hand.

The remainder of the guide is a description of finer points of the system. You should read through it once, to make later consultation easier.

2. Features

The FLACC implementation offers a number of features:

- Full language
  The full language, as defined in RR, is implemented. This includes all of formatted, unformatted and binary transput, all of the standard prelude, parallel processing, united modes, long and short modes, heap allocation, and other features often left out of implementations.
- Extensive debugging aids
  These include a fully symbolic dump, a trace function, a trace of the last thirty branches prior to termination, profile gathering, a traceback of active locales at termination, and an assertion operator.
- Extensive error checking
  Checks include use of uninitialized or undeclared values, arithmetic overflows, subscripts out of bounds, scope checks, deadlock of parallel processes, and many others. Redundant checks are avoided by compile-time analysis.
- Fast compilation
  FLACC has been designed for student 'cafeteria' batch use. The compiler is completely memory-resident, and uses no utility files. Some execution speed was sacrificed to achieve faster compilation.
- Lucid diagnostics
  Messages are given in plain English. Diagnostics pinpoint exactly where the problem is, giving the coordinates in the source program.
- Low cost
  FLACC takes advantage of virtual memory, and avoids loading overlays or using external utility files. The entire system is reentrant, and can be loaded once for several runs (the object code produced is also reentrant). Limits can be imposed on the cpu time, and lines and pages printed, to further limit costs.
- Load-and-go operation
  FLACC runs either as a load-and-go system, or as a production compiler which produces object modules. When no object modules are produced, compilation is faster, and no loading or linkage editing is needed.

- Standard program format
  FLACC conforms exactly to the IFIP hardware representation standard for Algol 68 programs. This eases portability considerations.
- Operating system independence
  All system-related routines have been gathered into a single module with a rigidly-defined interface. The operating system functions required by FLACC have been deliberately kept simple and basic to achieve independence from any particular operating system. Version 1.6 runs under OS, SVS, MVS, VM/CMS, and MTS.

3. Structure of the System

The FLACC system is composed of three main parts: the compiler, the run-time system, and the operating system interface.

The operating system interface is a collection of subroutines to perform all transput, timing, loading, and other functions which require operating system services. It also performs control card analysis and job batching.

The run-time system is a large collection of routines which provide the services needed by user programs. These include memory management, garbage collection, process scheduling, transput functions, standard prelude operations, dumping and tracing, error checking, and sundry other services. The run-time system comes in two versions, one for load-and-go operation, and one for production use.

The compiler is composed of several phases. The first phase reads the source text. Next, the parenthesis structure of the program is determined, and, if necessary, corrected. Now a context-independent parse is done. After mode information has been consolidated, a context-dependent parse is performed. At this point, several important object code improvements are done. Finally, the data structures required at run time are produced, and object code is emitted.

4. The Language Implemented

FLACC implements the precise language defined by RR, and by the IFIP WG2.1 Commentaries published in the Algol Bulletin, issues 42 and 43.

4.1. Extensions

FLACC optionally supports the following extensions over standard Algol 68:

- FLACC programs can call external subroutines which use Fortran linkage conventions.
- There is a mechanism for trapping run-time errors. Exception handling allows you to write 'bullet proof' programs.
- Overprinting is allowed in transput. This extension includes the addition of a sameline routine, and of m alignments in formats.
- Variable-length lines are allowed in random-access files. This extension includes the addition of a clipline routine, and of a t alignment in formats.
- Files may have global scope. This allows more carefree use of event routines and formats, without the tedious copying of files in each new scope. The special rules handling the scope 'violations' associated with this feature are described in Section 6.
- Some extensions have been made to the standard prelude. These include a slightly more complete set of elementary trigonometric functions, some new transput facilities, and some operating system enquiries such as time of day. The standard prelude implemented is described in Appendix A.

4.2. Restrictions

Great effort was expended to avoid implementation restrictions in FLACC. There are, however, the following two restrictions:

- Program source lines must be not more than 65535 characters in length.
- A program may consist of at most 65535 lines.

All other restrictions encountered in FLACC are due to either memory size or processing time limits being exceeded.

4.3. Differences

There are a few extremely subtle differences between FLACC transput and standard transput. Transput is described in considerable detail in Section 6.

4.4. Pragmats

The only pragmats accepted are those required by the IFIP hardware representation standard. These are RES, UPPER, POINT and PAGE. These pragmats are described in Section 5.

4.5. Character Set and Collating Sequence

The character set used internally is EBCDIC as defined for the IBM TN and T11 print trains (see IBM Form GX20-1850, the 'yellow card'). The TN EBCDIC collating sequence is used for character and string comparisons. There are 256 distinct character values.

Local installations may change the definition of square brackets and vertical bar ([,], |). The distributed version places these characters at hexadecimal AD, BD, and 4F, respectively.

The character set allowed for program text is exactly that required by the IFIP hardware representation standard. This character set is described in Section 5.

4.6. Treatment of Undefined and Skip

RR specifies many situations which lead to 'undefined' results. In FLACC, nearly all of these situations cause termination with an error message. The two most important exceptions to this rule are the treatment of SKIP, and collaterality.

SKIP yields some value which can be manipulated without error. Therefore, if SKIP is assigned to a tag, that tag is considered to be initialized. The exact value of SKIP is not defined. In FLACC it has the property that it is 'almost always wrong'. Thus use of SKIP often leads to 'obviously wrong' results.

Collaterality poses a much more important problem. The rule of collaterality leaves undefined the order in which some operations are done. The most widespread example is the order of evaluation in procedure argument lists, and in formulas. Choosing different orders can result in different results if there are side-effects. FLACC always picks a consistent ordering, except in parallel clauses, where random scheduling between tasks can be requested.

FLACC does not allow the use of uninitialized tags or

variables. FLACC does *not* initialize all values to SKIP, as can be inferred from RR.

4.7. Standard Environment Enquiries

Algol 68 provides for several predefined values which describe the implementation. The values FLACC assigns to these are given below.

FLACC implements two lengths of INT, REAL, BITS, and BYTES. Extra LONGs and SHORTs may be given, but have no run-time effect.

```
INT int lengths = 1,
INT int shorths = 2,
INT int width = 10,
INT short int width = 5,
INT max int = 2**31-1,
SHORT INT short max int = SHORTEN (2**15-1),
INT real lengths = 1,
INT real shorths = 2,
INT real width = 16,
INT short real width = 6,
REAL max real = about 7.237*10**75,
SHORT REAL short max real = about SHORTEN
    (7.237*10**75),
INT exp width = 2,
INT short exp width = 2,
REAL small real = about 2.220*10**-16,
SHORT REAL short small real = about SHORTEN
    (9.537*10**-7),
INT bits lengths = 1,
INT bits shorths = 2,
INT bits width = 32,
INT short bits width = 16,
INT bytes lengths = 1,
INT bytes shorths = 2,
INT bytes width = 4,
INT short bytes width = 2,
INT max abs char = 255,
CHAR null character = REPR 0,
CHAR flip = "T",
CHAR flop = "F",
CHAR errorchar = "*",
CHAR blank = " ";
```

5. Source Program Format

FLACC conforms exactly to the *Report on the Standard Hardware Representation* by Hansen and Boom, published by IFIP in the ACM SIGPLAN Notices, Volume 12, Number 5 (May 1977). That document defines a standard means of representing Algol 68 programs. References to the above document are written as SHR, possibly followed by a section number.

5.1. Character Set

The following sixty characters are acceptable for program text:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
" # $ % ' ( ) * + , - . / : ; < = > @ [ ] _ |
space
```

Additionally, the lower case letters are acceptable. Depending upon the stropping regime (next section), they may be folded into the upper case letters.

There are no restrictions on the character set used in comments. String and character denotations may contain any character except that quotes (") and apostrophes (') must occur in pairs.

It is poor practice to have a string denotation which contains a line break (newline). To overcome this, SHR introduces the string break. This consists of closing the denotation on one line (with a quote), then opening it again on the next line (with a quote). Depending on the setting of the BOUNDARY parameter (Section 8), FLACC may insist on the use of string breaks.

Note that, as required by RR, FLACC allows parentheses to be used in place of square brackets.

5.2. Stropping Regimes

Algol 68 has two classes of symbols or words: bold and plain. Language keywords, such as IF, are bold. Operators and modes are also bold. Tags (variable names) are plain.

Some means is needed to distinguish bold and plain words. This is accomplished by a 'stropping' regime. (Strop comes from apostrophe, which was the character used in many Algol 60 implementations to designate keywords.) SHR defines three such regimes: POINT, RES, and UPPER.

In POINT mode, a bold word is immediately preceded by a point (.), and a plain word is not. Case is irrelevant.

In RES mode, the point may be omitted from any of the sixty-one reserved words (RR9.4.1):

AT, BEGIN, BITS, BOOL, BY, BYTES, CASE, CHANNEL, CHAR, CO, COMMENT, COMPL, DO, ELIF, ELSE, EMPTY, END, ESAC, EXIT, FALSE, FI, FILE, FLEX, FOR, FORMAT, FROM, GO, GOTO, HEAP, IF, IN, INT, IS, ISNT, LOC, LONG, MODE, NIL, OD, OF, OP, OUSE, OUT, PAR, PR, PRAGMAT, PRIO, PROC, REAL, REF, SEMA, SHORT, SKIP, STRING, STRUCT, THEN, TO, TRUE, UNION, VOID, WHILE.

Depending on the setting of the EXTERNAL parameter (Section 8), the word EXTERNAL may also be reserved.

Case is irrelevant. In order to obtain a plain word which is in the above list, the underscore (_) is provided as a suffix character. Thus end of file, which would normally be three bold words, can be written end_of_file, one plain word. The underscore is not considered to be part of the tag. Thus foozle and foozle_ are the same tag.

In UPPER mode, bold words are given in upper case, and plain words in lower case.

The stropping regime to be used is selected with pragmats (next section), or parameters (Section 8).

5.3. Pragmats

FLACC supports four pragmats: POINT, UPPER, RES, and PAGE.

POINT, UPPER, and RES select the corresponding stropping regime, as described in the previous section.

The PAGE pragmat causes the line following it to be printed on the top of a new page in the listing.

5.4. Carriage Control

FLACC allows you to put carriage control characters in the first column of each line of the source program. This is under the control of the INFORM parameter (see Section 8). The allowed carriage control characters are:

1	skip to new page
–	triple space
0	double space
space	single space
+	overprint

Note that lines having + carriage control characters are treated as comment lines, and are ignored by the compiler.

6. The Transput System

This section describes the transput facilities provided by FLACC. These differ very slightly from those in RR.

6.1. Overview

Transput in Algol 68 can be as simple or as complicated as you want to make it. If you don't much care about the format of your data, then simple get and put calls will normally do what you want. If you want complete control over the format of your data, you can have that, too.

Algol 68 does transput (I/O) to books (datasets or external files) on a line-by-line basis, although many facilities ignore line boundaries. Each line is a row of characters. Lines are sometimes grouped into pages. Thus, a position in a book is determined by three integers: its page, line, and character numbers.

There are three important positions in each book: the current position, the logical end of file, and the physical end of file. The logical end of file determines how much data you can read; the physical end of file determines how much data you can write. Both of these positions are normally determined by the operating system.

The current position can be moved in two ways. It can be moved by reading or writing some characters from or into a line, or by calling one of several layout routines. These are sameline, newline, newpage, set, reset, space, backspace, and set char number.

In order to process the contents of a book, you must open it. This involves declaring a file (which is an internal structure used to keep track of the book), and calling either open or establish. open is intended for books which already exist, while establish is intended to create new books. establish does different things under different operating systems. In OS, VS and CMS, establish is treated exactly like open, except that the last parameter (line length) is used to set the logical record length before opening the dataset. In OS and VS, established files must already exist, or be described as 'new' in the JCL, while in CMS, they may be created implicitly by the operating system. In MTS, established files are created if they do not already exist (which requires an actual idf, see Section 6.3), and are emptied otherwise. (RR defines an additional function, create, intended to create a book with default characteristics. However, since one of the defaults

is the name of the book (dataset name), FLACC does not support it.)

When you have finished processing a book, you must close it. To do this, you call either close or scratch. (RR defines an additional routine, lock, which prevents you from reopening the book. However, since none of the operating systems FLACC runs on supports this feature, neither does FLACC.)

Another feature of Algol 68 transput is the associate function, which allows you to associate an array of characters with a file, instead of a book. This allows 'core-to-core I/O'.

There are three kinds of transput: unformatted, formatted, and binary. The first two produce human-readable strings which represent values, while the last transfers internal (unintelligible) representations of values. In unformatted and binary transput, FLACC determines the layout of the data. In formatted transput, you determine the layout, using formats.

A feature of Algol 68 transput is straightening. This allows you to input or output arrays and structures as a whole, with the transput system breaking them open to get at the required parts.

You can define some procedures (called event routines) which will gain control when certain events occur (such as end of line or end of file). RR also allows you to define character translations that occur during transput, although FLACC does not.

In order to model the different kinds of access and datasets your operating system provides, Algol 68 has channels. You give a channel when you open a book, and this selects what operations you can perform on the contents of the book. As an example, some channels allow only writing operations, and some allow only sequential access.

A handy feature of Algol 68 is that when your program begins execution, three files are already open for you: standin, standout, and standback. In FLACC, standin reads from the same place as the compiler so normally you put your data after your program; standout sends its output to the same place as the compiler so your output normally follows the source listing; and standback is associated with a 256-character array so you can use it for conversions.

6.2. Channels Provided

FLACC provides five channels: standin channel, standout channel, printer channel, sequential channel, and standback channel.

standin channel is a read-only, sequential-access channel. standout channel is a write-only, sequential-access channel. printer channel is also a write-only, sequential-access channel, but it supports carriage control characters. sequential channel is a read-write, sequential-access channel. standback channel is a read-write, random-access channel.

You can call reset on all channels, although you cannot call it on the files standin or standout. You can do all three forms of transput (unformatted, formatted and binary) on all channels. You can call sameline on only printer channel. You can call set and clipline on only standback channel. All channels can support variable-length lines (except standback channel under OS, VS, or CMS). There is only one page on standback channel.

6.3. Idfs

When opening or establishing a book, an idf (identification) string must be given. In FLACC, there are two kinds of idfs: logical and actual. Logical idfs are used to access books indirectly, via names assigned by operating system commands. Actual idfs are used to access books directly by their file or dataset name. All actual idfs must begin with an asterisk (*), which is stripped off by FLACC.

Null idfs are allowed on only standin channel and printer channel. These idfs select the input and output datasets of the compiler, respectively.

In OS and VS, logical idfs are DDnames, such as "SYSOUT". Actual idfs are not supported in the OS and VS versions of FLACC.

In CMS, logical idfs are also DDnames, while actual idfs (with the asterisk removed) are passed to FILEDEF to define a (constructed) internal DDname, so the file can be opened. For example, the terminal may be accessed using "*TERMINAL", and the disk file TEMP DATA might be accessed using "*DISK TEMP DATA A (RECFM F LRECL 80".

In MTS, logical idfs are logical unit names, such as "SCARDS", while actual idfs are FDnames, such as "*-P+-Q(5)", or "**PRINT*".

In all systems, idfs are translated to upper case before being passed to the operating system.

6.4. Carriage Control

The support of carriage control characters on printer channel is automatic and transparent to the user. When one of sameline, newline, newpage, or reset is done, the previous line is written out, and the carriage control character is set for the next line (to +, space, 1, or 1, respectively). The same actions are performed by m, l, and p format alignments, respectively. When a file is opened, its carriage control character is set to 1.

6.5. Page Handling

On channels other than printer channel, newpage has no apparent meaning. On these channels, newpage performs the same external actions as newline. That is, the current line is written, or a new line is read.

An exception to this rule is that newpage will not perform any external action (will not read or write a buffer), if the last operation done on the file was a newline. This prevents empty lines at the bottoms of pages when newline and newpage are called in succession.

Files which are opened using establish and associate have the number of pages given in the call. Files which are opened using open have maxint pages, unless they are on stand back channel, in which case they have one page.

6.6. Conversion Routines

RR provides for three conversion routines: whole, fixed, and float. These are intended to fall between unformatted and formatted transput in terms of amount of control over layout versus complexity.

FLACC provides two additional routines: scientific and convert. Use scientific when you want numbers in scientific notation with a specific precision (other than the maximum, which unformatted output gives you). (float does not always give scientific notation.) Use convert when you want 'nice, readable' output with a minimum of fuss; it normally gives the shortest string which represents its argument.

Two other useful routines are lpad and rpad. These routines pad strings to specified lengths using specified pad characters.

6.7. Event Routines

FLACC differs from RR primarily in the way it handles event routines. Under some very obscure circumstances, they are called in a different order, and a different number of times.

Input builds strings one character at a time, which are then converted to the required values. Similarly, output converts values to strings, then outputs them one character at a time. Immediately before any character is transput, an internal routine get good posn is called. It ensures a suitable current position for the next character. This may involve calling event routines for the line ended, page ended and file ended conditions.

Here is a skeleton of get good posn:

```
BOOL done := FALSE;
WHILE NOT done DO
IF      buffer is empty AND (reading OR on standback
            channel)
THEN    read in a line
ELIF    (current page > max page) OR (reading AND
            at end of file)
THEN    call on logical file end or on physical file end
ELIF    current line > max line
THEN    call on page end
ELIF    (NOT reading AND current char > max char)
            OR (reading AND current char > size of
            buffer)
THEN    call on line end
ELSE    done := TRUE
FI
OD;
```

Each time an event routine is called, a check is made to ensure that the file is still open, and has the appropriate read/write setting. If an event routine returns TRUE, a check is made to ensure that either the file or current position has changed, or that the condition for which the routine was called has been resolved.

If the file end event routine returns FALSE, the program is terminated with an error message. If the page end or line end routine returns FALSE, newpage or newline is called, respectively.

The char error, value error and format end event routines are handled as required by RR.

6.8. Binary Transput

There are no restrictions on when binary transput can be done in FLACC; it may be freely interspersed with formatted or unformatted transput, and done on any file.

FLACC binary transput behaves like unformatted transput of rows of characters. The internal representations of values are transput as byte streams.

The number of bytes transput is as follows:

SHORT INT	2
INT	4
SHORT REAL	4
REAL	8
SHORT BITS	2
BITS	4
SHORT BYTES	2
BYTES	4
SHORT COMPL	8
COMPL	16
CHAR	1
BOOL	1 (zero for FALSE, nonzero for TRUE)
STRING	1 per character

6.9. Random-Access Files

Algol 68 has facilities for handling random-access transput. These are set, reset, clipline, newline, and newpage.

set moves the current position to any desired point in the file. reset moves the current position to the beginning of the file. clipline truncates the current line to the current character position, and resets the maximum allowed length to its open-time value, in preparation for writing it out with a changed length. newline and newpage move to the next line or page. Since FLACC random-access files have only one page, newpage is of limited usefulness.

To call clipline from a format, use the t alignment.

Normally, only existing text in a random-access file may be replaced, and lines cannot have their lengths changed. However, through the use of clipline, the length of lines may be changed, provided that the operating system supports variable-length random-access files. For example,

```
open(f,"some idf",standbackchannel);
set(f,1,5,3);
put(f,"c");
close(f);
```

changes the third character in the fifth line to c, without changing any other part of the line. However,

```
open(f,"some idf",standbackchannel);
set(f,1,5,1);
clipline(f);
put(f,"abc");
close(f);
```

replaces the fifth line with abc.

Note that when the position in the file changes (with a set, reset, newline, or newpage, the new line is implicitly read. If any changes have been made to the old line, it is implicitly written.

In OS, VS, and CMS, FLACC random-access files are the same as Fortran direct files. That is, they are BDAM fixed-length datasets, which must be created and filled as sequential datasets before being used for random access. FLACC does not use keys; records are addressed by relative record number. The number is adjusted so that the first record is number one. If a short line is written (after using clipline), the operating system interface will pad it with blanks.

In MTS, line files are used. The position corresponds to the internal line number (1500 is line 1.5). Lines may be of variable length. The file may be pre-created sequentially (using another channel), or may be created randomly. If it is created randomly, then clipline must be used for each line, since non-existent lines have a length of zero when read.

6.10. Three Pitfalls

Algol 68 transput has the following three upsetting quirks:

Unformatted input of strings normally leaves the current position at the end of a line, so that subsequent calls to get result in empty strings being read. This can be avoided by calling newline between input calls.

Binary input of strings does not work as might be expected. The string read into is not flexed: its length remains unchanged. This means that calling getbin with an empty string (such as one you have just declared) results in no characters being read.

Scope restrictions on files make it inconvenient to associate event routines or formats with them. Inevitably, a new file must be declared in a local environ, and the file copied. To get around this, FLACC allows (under the control of the FILECHECK parameter; Section 8) files to be of global scope, and ignores certain scope violations in their use. This is detailed in the following section.

6.11. Scope of Files

When the FILECHECK parameter is in effect (Section 8), the actions described in this section do not occur. Instead, files are handled as required by RR.

FLACC allows files to have global scope. This gives rise to scope violations when event routines, formats and texts are associated with them. These violations do not result in termination. Rather, they are noted for later action.

When a local object (an event routine, a format, or a text) is associated with a file, the locale having the same scope as the object is flagged. Also, the file is made to refer to that locale.

When a flagged locale is left (for example, during a block exit or a jump), all files are checked to see if they refer to it. If so, then the event routine(s), format(s), or text(s) associated with the locale are 'reset'.

Event routines are reset by associating the default event routine with the file. The default event routine always returns FALSE. When a format is reset, the file is set to the 'no format' state. When a text is reset, the file is closed.

FLACC always closes files, and ensures that the last line is written out, even if you do not close the file, or if the program terminates abnormally.

6.12. Operating System Facilities

FLACC uses a standard, operating system independent interface to obtain access to operating system facilities.

Version 1.6 of FLACC has three interfaces: MTSINT, OSINT, and CMSINT. All support memory allocation, job initialization and batching, loading, timing and I/O.

MTSINT supports line and sequential files. Either indexed or sequential I/O can be performed on line files, but only sequential I/O can be done on sequential files. Both positive and negative line numbers may be used with line files. Files can be created, destroyed and renamed, using establish, scratch, and reidf. Files can be accessed using either logical or actual idfs (see Section 6.3).

OSINT supports QSAM and RECFM=F BDAM datasets. Partitioned datasets are supported only when the member names are given in the JCL. Random access files must be pre-formatted using sequential I/O before being opened for random I/O, and have only positive line numbers. establish cannot create a dataset, and scratch and reidf are

not supported at all. Only logical idfs are supported (see Section 6.3).

CMSINT uses the OS simulator, and provides the same services as OSINT, except that actual idfs are supported (see Section 6.3).

7. Debugging Aids

In addition to its extensive error checking, FLACC provides several debugging aids. These are described in this section.

7.1. Tracing

The trace function is designed to let you optionally print the values of your variables during execution. It is called in exactly the same manner as print, but produces output only if the value of trace flag is TRUE. A tracef routine is also provided, which is analogous to printf.

7.2. Assertions

The ASSERT operator allows you to place assertions in your programs. If its operand evaluates to TRUE, then it does nothing. If, however, its operand is FALSE, it causes an error termination.

7.3. Profiling

You can gather an execution profile of your program. After termination, a bar graph is printed which shows how often each decision point in the program was passed. Profiling is optional, and is under the control of the PROFILE parameter (Section 8).

7.4. Traceback

At termination, FLACC optionally prints a traceback giving the program source location in each active locale. If there are multiple tasks at termination, a traceback is given of each one. The traceback indicates clearly the nesting of calls and tasks at termination. Traceback is under the control of the DUMP parameter (Section 8).

7.5. Branch Trace

A table of the last thirty branches taken by your program is optionally printed when it terminates. These branches include task switches in parallel programs. This trace often gives needed information about 'where the program was'. The branch trace is under the control of the DUMP parameter (Section 8).

7.6. Symbolic Dump

Probably the most useful debugging feature of FLACC is its symbolic dump. Under the control of the DUMP parameter (Section 8), when your program terminates, the current values of some or all objects in memory are dumped. The format of this dump depends on the level selected by the DUMP parameter.

The level 1 dump consists only of the traceback and branch trace.

The level 2 dump is an abbreviated dump intended for beginning students and 'working' programs. It dumps the values of all simple tags; that is, those which are not arrays or structures. Strings and semaphores are also dumped. Tags are grouped according to the locale in which they are declared. Locales are labelled consistently with the traceback, so nesting information can be easily determined.

The level 3 dump is complete. It reflects the state of the Algol 68 virtual machine exactly. In fact, instructors may find it useful in teaching how Algol 68 handles values.

In the level 3 dump, all bindings and subnames are explicitly shown, and all mappings implied by structures and rows are detailed. Because this would normally lead to a lot of waste paper, all structured and rowed values are compressed onto as few lines as possible. This reduces the quantity of output considerably, while still leaving a fairly readable dump. As in the level 2 dump, locales are labelled consistently with the traceback.

8. How to Run FLACC

This section describes how to use FLACC.

8.1. OS, SVS, and MVS Job Setup

Most installations will have a catalogued JCL procedure for running FLACC. In this case, to run the compiler, you give the following:

```
// EXEC FLACC,PARM='parameters'
//SYSPUNCH DD object module dataset
input to FLACC
/*
```

The SYSPUNCH DD line is needed only if the DECK parameter is specified.

If your installation does not have a catalogued procedure for FLACC, or if you are creating your own procedure, the complete JCL to run the compiler follows:

```
//stepname EXEC PGM=FLACC,PARM='parameters'
//STEPLIB DD DSN=dataset containing
    compiler,DISP=SHR
//SYSPRINT DD listing dataset, usually SYSOUT=A
//SYSTERM DD terminal dataset, usually TERMINAL
//SYSPUNCH DD object module dataset
//SYSIN DD input dataset, usually *
```

The SYSTERM and SYSPUNCH DD lines are needed only if the TERM and DECK parameters are specified, respectively.

To run an object module, use the following JCL:

```
// EXEC LKEDG,PARM.GO='parameters'
//LKED.SYSLIB DD DSN=dataset containing run-time
    library,DISP=SHR
//SYSIN DD dataset containing object module
//GO.SYSPRINT DD output dataset, usually
    SYSOUT=A
input to program
/*
```

Note that GO.SYSPRINT must appear, even if the program produces no output on standout. GO.SYSIN must be explicitly supplied if there is no inline input to the program. Use:

```
//SYSIN DD DUMMY,DCB=BLKSIZE=80
```

8.2. CMS Job Setup Most installations will have a FLACC EXEC file. To run the compiler in this case, use:

```
FLACC program ( parameters
```

The program is taken from the file program ALGOL68, the listing is sent to program LISTING, and the object module (if any) is sent to program TEXT. If no parameters are specified, the parenthesis is not required.

If your installation does not have a FLACC EXEC, or if you are writing your own, then use the following:

```
FILEDEF SYSIN input description
FILEDEF SYSPRINT output description
FILEDEF SYSTERM TERMINAL
FILEDEF SYSPUNCH object module description
FLACC parameters
```

The SYSTERM and SYSPUNCH definitions are needed only if the TERM and DECK parameters are specified, respectively.

Here is how object modules are run in CMS:

```
GLOBAL TXTLIB dataset containing run-time library
FILEDEF SYSIN input description
FILEDEF SYSPRINT output description
LOAD file containing object module
START * parameters
```

Note that SYSIN and SYSPRINT must both be defined, even if apparently unused. If the program has no input from standin, then use:

```
FILEDEF SYSIN DUMMY
```

You may wish to put the GLOBAL command in your PROFILE EXEC. It needs to be issued only once after IPL'ing CMS.

8.3. MTS Job Setup Here is how FLACC is run in MTS:

```
$RUN *FLACC SCARDS=input SPRINT=output
    SPUNCH=object SERCOM=errors
    PAR=parameters
```

SPUNCH and SERCOM are required only if the DECK and TERM parameters are specified, respectively.

Here is how object modules are run in MTS:

```
$RUN object+*FLACCLIB SCARDS=input
    SPRINT=output PAR=parameters
```

In both cases, if SCARDS, SPRINT or SERCOM is omitted,

the terminal is the default. At most installations, the +*FLACCLIB is unnecessary, because the interface adds a $CONTINUE WITH *FLACCLIB RETURN line at the end of each object module.

8.4. Input to the Compiler

In all systems, the input to FLACC is the same, and depends on the setting of the BATCH parameter. Here is a two-run job in BATCH mode:

```
/COMPILE parameters
    first Algol 68 source program
/EXECUTE
    data for the first program
/COMPILE parameters
    second Algol 68 source program
/EXECUTE
    data for the second program
/EXECUTE
    another set of data for the second program
/END
```

All control lines start in column one, regardless of the setting of the INFORM parameter. Case is irrelevant. In BATCH mode, if no /EXECUTE line is given, execution is suppressed. A program may be executed several times with different data by giving more than one /EXECUTE line. Program data may be terminated by a /END line, which is completely optional.

In NOBATCH mode, there is only one run per job. The /COMPILE line is optional, and /END lines are ignored. A /EXECUTE line is needed only if data is present.

8.5. Other Control Lines

In addition to /COMPILE, /EXECUTE, and /END lines, FLACC accepts /OPTIONS, /TITLE, /STITLE, /EJECT, /SPACE, and /INCLUDE lines. These lines have an effect only at compile time, except /INCLUDE lines, which are also interpreted at run time, but only in BATCH mode.

/OPTIONS lines are used to change the values of parameters. The only parameters allowed on /OPTIONS lines are INFORM, POINT, RES, UPPER, LIBLIST, NOLIBLIST, LIST, NOLIST, BOUNDARY, NOBOUNDARY, and OUTFORM.

/TITLE and /STITLE allow you to put titles and subtitles on your source listing. The portion of the line following the first character after the /TITLE or /STITLE is used. Both types of line cause a page skip in the source listing. /TITLE lines reset subtitles.

/EJECT lines cause page skips in the source listing.

/SPACE lines leave blank lines in the source listing. Follow /SPACE with a number. If you omit this number, one is assumed. If too few lines remain on the page, a page is skipped.

/INCLUDE lines are used to include other files into the input stream. They can be used both at compile and execution time, and can be nested. Follow /INCLUDE with an idf (Section 6) in quotes (').

8.6. Parameters

FLACC accepts many parameters. These can be given in the parameter to FLACC, or on /COMPILE and /OPTIONS lines. Parameters may be in either case, and separated by blanks, commas or semicolons.

All parameters, except for BATCH, NOBATCH, TERSE, NOTERSE, TERM, and NOTERM may appear on /COMPILE lines.

The only parameters allowed on /OPTIONS lines are INFORM, POINT, RES, UPPER, LIBLIST, NOLIBLIST, LIST, NOLIST, BOUNDARY, NOBOUNDARY, and OUTFORM.

The TERM parameter, in conjunction with the LIST parameter, determines where compiler and interface error messages go. When TERM is specified, SYSTERM (SERCOM in MTS) is assumed to be connected to a terminal, and error messages are sent there. If LIST is specified, then error messages are also sent to SYSPRINT (SPRINT in MTS). NOTERM,NOLIST is a special case; here messages are sent to SYSPRINT in listing format, but listing titles are suppressed. Error messages sent to the terminal include a copy of the source line in error.

Parameters accepted are as follows:

TERM, NOTERM
: Determines whether compiler error messages are sent to the terminal (see above).

TIME=xxx.xxx
: Sets the amount of execution CPU time allowed in seconds. Can be abbreviated to T=.

CTIME=xxx.xxx
: Sets the amount of compilation CPU time allowed in seconds. Can be abbreviated to CT=.

LINES=xxx
Sets the number of lines output allowed on standout during execution. Can be abbreviated to L=.

DUMPLINES=xxx
Sets the number of lines output allowed during the post-termination dump. Can be abbreviated to DL=.

PAGES=xxx
Sets the number of pages output allowed on standout during execution. Can be abbreviated to P=.

LINECT=xxx
Sets the number of lines per page in the source listing, and also in standout output. Can be abbreviated to LCT=.

SIZE=(min,max,res) or SIZE=amount
Sets the amount of work space obtained during run. If the first form is used, res bytes are reserved for system use (buffers and such), and somewhere between min and max bytes are allocated as work space. If the second form is used, amount bytes are allocated as work space. You can suffix numbers in this parameter with K to indicate multiples of 1024, or KK to indicate multiples of 1048576. Can be abbreviated to S=, or to three separate parameters, MN=, MX=, and MR=.

INFORM=(bgn,end,asa)
Sets the input line format. bgn sets the first column used by the compiler, and end sets the last one used. asa must be either A or N, and indicates whether the first column of each line contains a carriage control character. If asa is A, then bgn must be at least 2. In OS, VS, and CMS, if end is 72, then compiler input lines which are 80 characters long and whose last four columns are numeric will have their last eight columns (73-80) used as sequence numbers, which will appear in the listing. Using NUM datasets in TSO, or setting SEQUENCE ON in CMS will fulfill these requirements. In MTS, line numbers are used instead of sequence numbers. INFORM can be separated into three abbreviated parameters: IB=, IE=, and IA=.

UPPER, RES, POINT
Sets the stropping regime.

LIST, NOLIST
Determines whether a source listing is produced.

LIBLIST, NOLIBLIST
Determines whether /INCLUDEd lines are printed in the source listing. LIBLIST is overridden by NOLIST. Can be abbreviated to LIBL, NOLIBL.

BOUNDARY, NOBOUNDARY
Determines whether tags, comments and string denotations are allowed to cross line boundaries. NOBOUNDARY allows this, and conforms to RR specifications. Can be abbreviated to BOUND, NOBOUND.

PROFILE, NOPROFILE
Determines whether a run-time profile is gathered. Can be abbreviated to PROF, NOPROF.

DUMP=x
Sets the level of the post-execution information provided, following a normal termination. 0 requests no output at all, 1 requests a traceback and branch trace only, 2 requests a partial dump, and 3 requests a full dump.

EDUMP=x
Sets the level of the post-execution information provided, following an abnormal or error termination. Has the same meaning as DUMP. If the value of DUMP is greater than the value of EDUMP, the value of DUMP is used.

OUTFORM=x
Determines the format of the source listing. 1 leaves the source unchanged; 2 produces output acceptable as UPPER input; 3 produces only upper case, and underlines bold words; 4 produces only lower case, and underlines bold words; 5 produces only upper case, and prints bold words twice; 6 produces only lower case, and prints bold words twice; 7 produces only upper case, and prints bold words thrice; 8 produces only lower case, and prints bold words thrice. Can be abbreviated to OUTF=.

FILECHECK, NOFILECHECK
Determines whether files are checked for scope violations. FILECHECK conforms to RR specifications. Can be abbreviated to FCHK, NOFCHK.

BATCH, NOBATCH
Determines whether one or several programs are to be compiled.

SCHEDULE=x
Sets the level of parallel scheduling. 0 requests minimal scheduling; 1 requests round-robin simulated timeslicing; 2 requests random repeatable simulated timeslicing; 3 requests random non-repeatable simulated timeslicing. If object modules are to use this parameter, it must be specified at both compile and execution time. Can be abbreviated to SCHED=.

XREF, NOXREF
Determines whether a tag cross-reference is printed.

MODETABLE, NOMODETABLE
Determines whether a table of all modes used in the program is printed. Can be abbreviated to MODET, NOMODET.

RUN, NORUN
Determines whether execution is suppressed. RUN has no effect when DECK is specified.

WARN, NOWARN
Determines whether compiler warning messages are suppressed.

DECK, NODECK
Determines whether an object module is produced. If DECK is specified, execution will be suppressed.

EXTERNAL, NOEXTERNAL
Determines whether the external linkage specifications are allowed. Also determines whether the word EXTERNAL is reserved. NOEXTERNAL conforms to RR.

MALIGN, NOMALIGN
Determines whether m alignments are allowed in formats. NOMALIGN conforms to RR.

TALIGN, NOTALIGN
Determines whether t alignments are allowed in formats. NOTALIGN conforms to RR.

EXTEND, NOEXTEND
Determines whether FLACC extensions to the standard prelude are recognized. NOEXTEND conforms to RR.

TERSE, NOTERSE
: Determines whether the interface prints headings, the version number, /COMPILE lines, and parameter values. If TERSE is chosen, then the interface will not produce any output.

STANDARD
: Selects the 'standard' settings for various parameters. When STANDARD is selected, FLACC conforms exactly to RR. The parameter settings selected are: NOBOUNDARY, FILECHECK, NOEXTERNAL, NOMALIGN, NOTALIGN, NOEXTEND, INFORM=(1,32767,N).

The default settings for TERM, LIST, SIZE, and INFORM in the distributed version depend on the operating system interface. In OS and VS, they are NOTERM, LIST, SIZE=(64K,16KK,64K), INFORM=(1,72,N). In CMS, they are TERM, LIST, SIZE=(64K,16KK,64K), INFORM=(1,72,N). In MTS, they are SIZE=128K, INFORM=(1,32767,N), and NOTERM, LIST in batch, or, at a terminal, either TERM, NOLIST if SPRINT points at the terminal, or TERM, LIST if it does not.

The other default parameter settings of the distributed version are as follows (defaults may vary at different installations):

TIME=25000, CTIME=25000, LINES=100000,
PAGES=100000, DUMPLINES=500, LINECT=60,
OUTFORM=2, DUMP=0, EDUMP=2, SCHEDULE=0,
UPPER, LIBLIST, BOUNDARY, NOPROFILE,
NOFILECHECK, NOXREF, NODECK, NOMODETABLE,
WARN, RUN, EXTERNAL, MALIGN, TALIGN, EXTEND,
TERSE, NOBATCH

9. External Linkage

A large body of software exists which, for one reason or another, was developed in languages other than Algol 68. Much of this other-language software is in the form of Fortran-callable subroutine libraries.

FLACC has an external linkage facility which allows access to Fortran or Assembler language subprograms. This section describes the types of linkage which are possible, and the linkage conventions which must be used.

9.1. Formal Syntax

The following syntax is appended to the grammar given by RR. If you are uncomfortable with this definitional approach, simply skip this part.

A) **EXTERNAL :: procedure FARAMETY yielding FRET.**
B) **FARAMETY :: FARAMETER FARAMETY; EMPTY.**
C) **FARAMETER :: FODE parameter; reference to flexible row of character parameter.**
D) **FODE :: FOAD; reference to FOAD; ROWS of FOAD; reference to ROWS of FOAD.**
E) **FRET :: FOAD; row of character; void.**
F) **FOAD :: SIZETY integral; SIZETY real; boolean; character; structured with SIZETY real field letter r letter e SIZETY real field letter i letter m.**

a) **strong EXTERNAL NEST unit : external symbol, external module indication.**
b) **external module indication : character denotation; row of character denotation.**
c) *** external unit : strong EXTERNAL NEST unit.**

9.2. Informal Syntax and Semantics

FLACC provides for a new language construct called an external unit. The external unit may be used to call subroutines which have been compiled by the IBM Fortran IV (G and H) compilers. The external unit may also be used to call Assembler language subroutines which follow standard Fortran linkage conventions.

Consider the following:

```
PROC (REAL) REAL sine = EXTERNAL "DSIN";
```

In this example, the external unit yields a procedure which is presumably the Fortran DSIN function. (Note that the FLACC standard precision corresponds to Fortran double precision.) The string denotation is an external module indication, and, in this case, is simply the external name of the routine.

The mode of the external function is specified by the (strong) context; thus the following example is invalid, since the compiler cannot determine the mode of the procedure:

```
PROC cosine = EXTERNAL "DCOS";
```

9.3. Data Types

Fortran lacks a rich variety of data types: a relatively small subset of the possible Algol 68 parameter modes can be passed in a sensible manner. The types which may be passed to Fortran subroutines are:

```
INT     REF INT     [,...] INT     REF [,...] INT
REAL    REF REAL    [,...] REAL    REF [,...] REAL
COMPL   REF COMPL   [,...] COMPL   REF [,...] COMPL
BOOL    REF BOOL    [,...] BOOL    REF [,...] BOOL
CHAR    REF CHAR    [,...] CHAR    REF [,...] CHAR
```

In addition, all other lengths of INT, REAL, and COMPL are allowed. Note that FLACC does not allow procedure values to be passed to external subroutines, and that no means is provided to access Fortran common blocks.

The correspondence between Fortran types and Algol 68 modes is as follows:

Fortran	*Algol 68*
INTEGER*2	SHORT INT
INTEGER	INT
REAL	SHORT REAL
REAL*8	REAL
COMPLEX	SHORT COMPL
COMPLEX*16	COMPL
LOGICAL	BOOL
LOGICAL*1	CHAR

9.4. An Example

In the following example, four entry points to a common plotting package are declared for use by a FLACC program.

```
PROC VOID plots = EXTERNAL "PLOTS";
PROC (SHORT REAL, SHORT REAL, INT) VOID plot =
    EXTERNAL "PLOT";
PROC (SHORT REAL, SHORT REAL, SHORT REAL, []
    CHAR, SHORT REAL, INT) VOID symbol =
    EXTERNAL "SYMBOL";
PROC (INT) VOID new pen = EXTERNAL "NEWPEN";
```

This example will not work properly under MTS in load-and-go mode, because the common blocks needed by the subroutines will not be merged by the system loader. Use the DECK option to bypass this problem.

9.5. Multi-Dimensional Arrays

Multi-dimensional arrays are allowed as parameters, although there is a surprising amount of linkage overhead involved. The following illustrates why array copying may be needed:

```
[5,5,5] INT array3;
REF [,] INT array2 = array3 [,3,];
PROC (REF [,] INT) VOID zonk = EXTERNAL "ZONK";
zonk(array2);
```

The two-dimensional array variable array2 is not contiguous in memory. There is no clean, transparent way to deal with non-contiguous arrays in Fortran.

The FLACC linkage interface alleviates the problem by constructing a compacted copy of the array. The copy is then passed to the Fortran subroutine. When the subroutine returns to the Algol 68 caller, the (possibly modified) compacted array elements are copied back to their original locations. This value-result calling convention is used by Fortran only for scalars. However, the FLACC linkage interface uses value-result conventions for all variable parameters.

The FLACC linkage interface also ensures that multi-dimensional array subscripts are given in the same order in the Fortran subprogram as in the calling Algol 68 program.

9.6. I/O Environment

Normally when Fortran subroutines do their own I/O, the Fortran run-time library (IBCOM) must be initialized. If you are comtemplating calling such subroutines from Algol 68 you should refer to the Fortran Programmer's Guide at your installation.

9.7. Assembler Subroutines

FLACC uses the same linkage conventions in calling Assembler language subprograms as in calling Fortran subroutines.

In addition to the Fortran parameter types, Assembler language subroutines may be passed string variables. An Assembler language function may also yield a string as its value:

```
PROC (REF STRING) STRING reverse = EXTERNAL
    'REVERSE';
```

When a string variable is passed, FLACC uses a two-word descriptor consisting of a pointer in the first word and the string length in the second. The subroutine is allowed to change both the pointer and the length. When returning a string value, place the address of the descriptor in general register 0.

Note that REF STRING parameters are passed very differently from both [] CHAR and REF [] CHAR parameters. The descriptor is passed only in the REF STRING case; otherwise the normal Fortran convention is used.

BOOL values are passed as fullword 0 or 1, for FALSE and TRUE, respectively. FLACC accepts zero or non-zero results returned in general register 0, and interprets them as FALSE and TRUE, respectively.

CHAR values are passed as single bytes. When returning a CHAR value, put it in the high-order byte of general register 0.

To return a REAL value, place it in floating-point register 0.

To return a COMPL value, load the real part into floating-point register 0, and the imaginary part into floating-point register 2.

9.8. Return Codes

Both Fortran and Assembler language subroutines are capable of setting the return code in general register 15. The FLACC procedure return code can be used to obtain the return code from the last external call. Exercise caution when using return code within parallel environments.

9.9. Re-entrancy

For each external unit, FLACC allocates a work word in the primal environ. The work word is initialized to 0 in a prelude. On entry to an external subroutine, general register 2 points to the work word for that subroutine. The work word is intended for use by re-entrant subroutines, to store the address of their work areas.

9.10. Scope

The scope of the procedure value yielded by an external unit is the scope of the primal environ.

9.11. Dynamic Loading

When NODECK and RUN are specified (Section 8), FLACC loads external units using the dynamic loading facilities (if any) of the host operating system. The external module indication may consist of the external name, optionally followed by a comma and an idf (Section 6) for the appropriate library. For example:

```
PROC (INT) VOID control = EXTERNAL "CNTL,EXTLIB";
```

If the external module indication does not include a library idf, the operating system default library is used. The loader is called separately for each external unit. If you load a collection of subroutines which share Fortran-style common sections, you should check with a systems programmer to ensure that the commons are being merged properly.

If the external module indication includes a library idf, and the DECK parameter has been specified, the idf is ignored.

9.12. Parallelism

FLACC performs automatic serialization of all external calls. Thus, in a parallel environment, when one task calls an external subroutine, all other tasks under FLACC jurisdiction are suspended until control is returned to the FLACC environment.

10. Exception Handling

This section describes the FLACC exception handling extension.

10.1. Introduction

FLACC provides a basic facility for recovering from run-time errors.

Here is a simple example:

```
BEGIN
    on error(overflow exception, ovf);
    i := j * k
    EXIT
    ovf: i := maxint
END
```

A handler is set up by calling on error, typically with a standard prelude tag and a label. When an error associated with the tag occurs, it is trapped, and the program jumps to the label. Note that using EXIT allows the handler to substitute a value for the interrupted clause or expression. (Note there is no semicolon either preceeding or following the EXIT.) It is not possible to resume interrupted code.

10.2. New Indicants

The FLACC exception handling extension defines three new procedures, a mode, and several tags:

- MODE EXCEPTION
- PROC EXCEPTION new exception
- PROC (EXCEPTION, PROC VOID) VOID on error
- PROC (EXCEPTION) VOID raise
- EXCEPTION any exception
- EXCEPTION bounds exception
- EXCEPTION deadlock exception
- EXCEPTION divide by zero exception
- EXCEPTION memory limit exception
- EXCEPTION overflow exception
- EXCEPTION range exception
- EXCEPTION transput exception
- EXCEPTION underflow exception

new exception is used to create new (user-declared) exception values. Normally these are bound in an identity declaration to a tag.

on error is used to set up a trap for an exception. Its second parameter is a PROC VOID; labels are procedured as needed.

raise is used to raise an exception. Most exceptions are raised implicitly by the run-time system, but user-declared ones must be raised explicitly by calling raise. Standard exceptions (such as overflow exception) may also be raised by calling raise.

any exception, as its name implies, can be used to trap any exception. A handler for any exception will be used if there is no more specific handler available. In this sense it is like an OUT in a CASE clause.

transput exception can be used to trap any error arising during transput. This includes opens which fail, conversion errors, end of file, set errors, etc.

overflow exception, underflow exception, and divide by zero exception are most useful in machine-portable numeric software.

bounds exception includes all errors arising from slicing rows, such as bounds mismatches during assignation, and subscripts out of range.

range exception includes all out-of-range arguments to prelude procedures and operators, such as REPR, lpad, sin, etc.

deadlock exception and memory limit exception do the obvious things.

10.3. Non-Interceptible Errors

There are five 'mortal errors': line limit exceeded, page limit exceeded, time limit exceeded, an exception raised which has no handler, and an exception handler returning rather than jumping.

10.4. Range of Handlers

A handler is in effect for the range corresponding to its necessary environ. When labels are used, this environ is the environ in which the label is defined. It may be different if on error is called with a procedure rather than a label, depending on what non-local tags the procedure references.

Exception handlers may be nested, but note that the nesting corresponds to the necessary environs of the handlers, and not the environs of the calls to on error. For example, consider:

```
BEGIN # environ 1 #
    ...
    BEGIN # environ 2 #
        ...
        on error(any exception, exit2);
        BEGIN # environ 3 #
            on error(any exception, exit1);
            ...
        END; # environ 3 #
        exit2: ...
    END; # environ 2 #
    ...
    exit1: ...
END # environ 1 #
```

The first handler (exit2) is in effect from the first call of on error to the end of environ 2, whereas the second handler (exit1) is in effect from the end of environ 2 to the end of environ 1. This is because the more local handler masks the more global one. Had the handlers been for different exceptions (and the first one not been for any exception), then the second handler (exit1) would have been in effect from the second call of on error to the end of environ 1.

If two handlers are set up for the same exception and the same environ, the later one replaces the earlier one. Once a handler is set up, it can be replaced, but it is not possible to revert to the 'no handler' state without leaving the handler's necessary environ.

10.5. Invocation of Handlers

Before a handler is invoked, all environs more local (newer) than the necessary environ are released. This may involve returning from calls, and terminating parallel clauses, that are more local than the handler's necessary environ. Note that, as expected, handlers are inherited by calls.

When a handler for an environ is invoked, all handlers for that environ are reset. Thus, to trap (locally) a second exception inside a handler, on error must be called after the handler for the first exception is entered.

10.6. User-Defined Exceptions

User-defined exceptions can be defined like this:

```
BEGIN
    EXCEPTION singular exception = new exception;
    PROC invert = ...
        ... raise(singular exception); ...;
    BEGIN
        on error(singular exception, singular);
        invert(...)
        EXIT
        singular: ...
    END
END
```

10.7. A More Structured Approach

For those who find labels and jumps distasteful, it is possible to define a more structured way of handling exceptions. This involves defining an operator which takes a 'suspect' procedure together with a handler specification. The handler specification might be a two field structure display, consisting of the exception expected, and a procedure to handle it. For example:

```
MODE ERROR = STRUCT(EXCEPTION e, PROC VOID
    h);
MODE TRY = VOID, USE = VOID;
PRIO ON = 1;
OP ON = (PROC VOID suspect, ERROR handler) VOID:
BEGIN
    on error(e OF handler, error);
    suspect
    EXIT
    error: h OF handler
END
```

There are a few clever tricks used here. Because a structure display cannot be an operand, it must appear as a cast. Thus a call to ON is written as ON ERROR. Also, PROC VOID values are not called when used as parameters, but are called when free-standing as clauses. TRY and USE are used to improve readability.

This facility might be used in the following way:

```
(TRY: i := j * k)
ON ERROR(overflow exception, USE: i := maxint)
```

The parentheses around the suspect procedure are required by the language syntax (routine texts are not operands).

Of course, a more elaborate scheme could be invented using a list of exceptions and handlers instead of just one.

11. Output from the System

This section describes the output that FLACC produces.

11.1. Parameter Values

When the NOTERSE parameter is specified (Section 8), the operating system interface starts each run with a banner identifying the version of FLACC and the installation copy number and name.

The /COMPILE line, if any, is shown next. If there are errors in this line, they follow, provided that the NOTERM parameter (Section 8) is in effect. If TERM is in effect, errors are sent to the terminal, rather than the listing.

The parameter values for the run are given next. Following this is a line saying how much memory has been allocated for the run.

None of this output (except for error messages about bad parameters) appears if TERSE is in effect.

11.2. The Source Listing

FLACC source listings are quite elaborate.

At the top of each page, two (double spaced) lines are reserved for the current title and subtitle. If no /TITLE or /STITLE lines are used, then these two lines are left blank, except for the page number, and the time and date of the compilation, which appear at the right-hand end.

Within the body of a page in the listing, there are five fields. These are the error flag, the source line number, the coordinate line number, the nesting depth, and the source text.

The error flag (if any) appears as three asterisks (***) at the left-hand side of the page. Only lines which evoke error or warning messages have error flags. For each error or warning, there is an underscore (_) in the source text field to point at the text in error, unless an OUTFORM value of 3 or 4 is used (requesting underlining of bold words), in which case a point (.) is used instead.

Source line numbers appear in columns 4 through 13. In MTS, these are the line numbers in the source file. In OS, VS and CMS, these may be blank, or may be the sequence fields from the source dataset, provided that the INFORM parameter has been set appropriately, and the

editor has been instructed to number the dataset. See the description of INFORM in Section 8.

Column 14 will normally be blank, but will contain a minus sign (−) if the line was included as a result of a /INCLUDE line. If the NOLIBLIST parameter is in effect, such lines are not listed.

Columns 15 through 19 contain the source line coordinate. This value (which can be easily recognized because of its leading zeros) is used by both the compiler and the run-time system to identify the line. It is used in error messages, the cross-reference, traceback, profile, and dump.

Columns 21 through 31 contain the nesting depth indication. For each level of nesting encountered on the line, there is a character in the nesting depth field, provided the maximum depth does not exceed 10. The outermost layer of the program has a depth of 0. For each BEGIN, IF, CASE, FOR (or other bold word starting a loop), (, or [, the nesting depth is increased by one. For each corresponding END, FI, ESAC, OD,), or], it is decreased by one. When one or two nesting levels appear on a line, the appropriate digit(s) are placed in the field. When more than two appear, only the first and last digits appear, and the middle ones are replaced with minus signs (−). When the nesting depth exceeds 10, a plus sign (+) is put at the right-hand end of the field.

The source text appears in columns 33 through 132. If the source line is longer than 100 characters, it is split across as many lines as necessary.

The appearance of the source text depends on the setting of the OUTFORM parameter. For example, the input line

```
.BEGIN STOP .END
```

(in RES mode) would appear in the listing as:

Outform setting	*Appearance*
1 (as is)	.BEGIN STOP .END
2 (mixed case)	BEGIN stop END
3 (upper underlined)	BEGIN STOP END
4 (lower underlined)	begin stop end
5 (upper overprint 2)	**BEGIN** STOP **END**
6 (lower overprint 2)	**begin** stop **end**
7 (upper overprint 3)	**BEGIN** STOP **END**
8 (lower overprint 3)	**begin** stop **end**

OUTFORMs of 5 through 8 are recommended only on well-adjusted line printers or hard-copy terminals.

11.3. Mode Table

Under the control of the MODETABLE parameter, FLACC will produce a list of all modes used the program. It will also produce one if the program contains an error which requires mode information to understand.

Each mode is assigned a number, which is prefixed with a number sign (#). The table consists of entries, one to a line, each describing one 'level' of a mode. For example, the mode [][]INT would take three lines, as in:

```
MODE#1 INT
MODE#2 ()#1
MODE#3 ()#2
```

Structs, procedures, and unions are handled similarly. For example, if a program contains the following modes,

```
INT
STRUCT(INT a, b)
PROC(INT a, b)INT
UNION(INT, STRUCT(INT a, b))
```

then the mode table might be:

```
MODE#1 INT
MODE#2 STRUCT(#1 A,#1 B)
MODE#3 PROC(#1,#1)#1
MODE#4 UNION(#1,#2)
```

Refs, transient refs, and flexes are also broken out as separate lines in the mode table.

The mode numbers in the mode table are referred to by the cross-reference, by the post-execution dump, and by both compiler and run-time error messages.

11.4. Cross-Reference

Under control on the XREF parameter, FLACC will produce a tag cross-reference.

For each tag in the program, the name, mode, defining line, and referenced line(s) appear. Each separately declared version of each tag is individually cross-referenced.

11.5. Compiler Error Messages

All compiler error and warning messages appear together at the end of the listing. These are sorted by line number.

Each message consists of five fields: the level of severity, the source coordinates, the error message number, the mode number, and the text of the message.

The severity level is either W or E to indicate a warning or an error.

The source coordinates are two numbers: a line number and a character number, enclosed in parentheses. For example, (00010,00034). Some errors are not associated with any particular source location. These appear last, and have blank coordinate fields.

The mode number field appears only if the error has something to do with a mode, such as a mismatch between a parameter and an argument in a call. If it does appear, the text of the message refers to it if there is any confusion about which mode is involved in the error.

11.6. Compilation Summary

At the end of the listing, FLACC gives a summary line of the compilation. This contains the version number of FLACC (in case TERSE is specified), the number of errors detected during the compilation, the work space the compiler used, and how much object code was produced.

11.7. Terminal Messages

When the TERM parameter is in effect, then some messages are sent to the terminal in addition to those sent to the listing.

Any errors in parameter lists (or on /COMPILE lines) will evoke messages at the terminal.

Compiler error and warning messages are also sent to the terminal. These have a very different format from those in the listing. Preceeding each message, the affected source line is sent to the terminal, in 65-character pieces. Dollar signs ($) are used to mark error locations. The source line number is given, or, if this has not been supplied by the operating system interface, the source coordinate is given instead. On a separate line, the severity level is given (either E or W), together with the mode number (if any), and the error message.

Compiler error messages not associated with any particular line are given last, separated from any previous messages by a line of dashes.

At the end of the compilation, the compiler summary line is sent to the terminal.

11.8. Run-time Error Messages

When the program terminates abnormally, the source coordinates of the error are given together with an error message detailing the fault.

In load-and-go mode (that is, when NODECK and RUN have been specified), the source line is also given, unless the error occurs in a transput routine, such as read. A marker indicates the exact point of the error.

Following the error message is a line summarizing the amount of cpu time used in the run, and the number of storage regenerations (garbage collects) performed.

11.9. Traceback

Provided that DUMP is at least 1 (EDUMP in the event of abnormal termination), a traceback through all active environs is given.

The traceback begins with the newest environ (the one most recently entered), and proceeds outward to the primal environ.

There is one line for each active environ. It consists of a coordinate, a locale item number, and a locale name. The initial coordinate is the point of termination. This remains unchanged through nested environs until the outermost environ of a procedure is reached. The coordinate used on the next line is the point of the call to the procedure. This continues back through calls until the primal environ is reached.

Each locale is given a unique item number which is used in the dump (see below). Each locale also has a name, which always includes the coordinates of its beginning, and, in the case of procedures, the name of the procedure.

There is a separate traceback for each task that is active at termination. These are simply numbered, starting at 1. Each traceback starts at the coordinates where termination occurred, and goes back towards the primal environ. When the traceback of some task merges with a previous task (at the PAR BEGIN clause), a message appears stating which task it merges with, and the remainder of the merged task's traceback is suppressed. The last line of the merged task's traceback is a duplicate of some line of the earlier task's traceback.

11.10. Level 2 Symbolic Dump

When the DUMP value is 2 (EDUMP in the case of abnormal termination), an abbreviated dump of active storage at termination is given.

Each locale is given a unique item number. Item numbers are always prefixed with an at sign (@). The primal environ is always @1. Item numbers can be thought of as addresses. The item numbers in the dump correspond to those in the traceback.

The dump is ordered by increasing item number. These need not correspond to the age or nesting of the locales, because storage regenerations (garbage collections) can cause new locales to be placed among old ones.

The level 2 dump consists of a block of information for each locale. This starts with the locale item number, and its name. Then, provided it is not the primal environ, the item number of the next older locale is given. This chaining of locales corresponds to the traceback. Following this, the values of tags declared in the locale are given.

Depending on the mode, an actual value may be given, or just the mode number (see Mode Table description above). If the mode is INT, REAL, COMPL, short or long versions of these, BOOL, CHAR, SEMA, []CHAR, STRING, or REF or UNION of any of these, then the value is printed. Note that complete dereferencing is done.

If a value is uninitialized, or a declaration has not been elaborated, the tag is called UNDEFINED. In the case of strings, any undefined characters are printed as question marks (?), and a note is appended saying some characters are undefined.

Note in particular that rows of other than characters, and all structs, appear only as mode numbers.

11.11. Level 3 Symbolic Dump

When the DUMP value is 3 (EDUMP in the case of abnormal termination), a detailed dump of active storage at termination is given.

Each value is given a unique item number, which is prefixed with an at sign (@). The primal environ is always @1. Item numbers can be thought of as addresses. The item numbers of locales in the dump correspond to those in the traceback.

The dump is ordered by increasing item number. These

need not correspond to the age or nesting of the values, because storage regenerations (garbage collections) can cause new values to be placed among old ones.

Locales are dumped as a collection of lines, the first giving the item number and name of the locale, as they appear in the traceback. The next line (unless it is the primal environ) gives the item number of the next older locale. This chain corresponds to the traceback. Next, any tags declared in the locale are given, one per line. These lines consist of a tag and an item number. After the tags, values contained within the locale are dumped as separate items, one per line.

An item is always dumped as an item number, the first word of the mode spelling together with a mode number in parentheses, and a value. Item numbers are always prefixed by at signs (@), and mode numbers are always prefixed by number signs (#). Thus a typical item looks like:

```
35 (INT#1) 23
```

A value is dumped variously depending on its mode.

Uninitialized or as yet undeclared values (ones whose locale has been allocated, but whose declarations have not been elaborated) are given as UNDEFINED.

Since united values do not exist *per se*, the value of a UNION is treated as though it is of its current value's mode.

INT, REAL, BITS, BYTES, long and short versions of these, BOOL, and STRING values are given as denotations. Note that strings are distinguished from rows of characters.

Only the mode is given for FORMAT, FILE, CHANNEL and EMPTY values.

EXCEPTION values are given as integers.

PROC values consist of the tag the procedure denotation was ascribed to, if any, together with the coordinates of the beginning of the procedure denotation. If the denotation was not ascribed (it may be assigned instead), or was not ascribed to a tag (it may be ascribed to a field of a struct, or passed in a call, for instance), then no tag is given.

A REF value consists of a points-at symbol (−−>) followed by the item number of the value it refers to. For example, a REF INT tag i could appear as follows in the

dump of the locale:

```
@5 LOCALE OF (00001,00001)
    PREVIOUS LOCALE @1
    I@6
@6 (REF#2) ---> @7
@7 (INT#1) 2
```

If the ref is null, then NIL appears instead.

SEMA values appear as refs that point at integers.

STRUCT values are broken open much like locales, except that each field is not put on a new line. The field names are given in order as a tag followed by an item number. The field values follow the field names as separate items. For example, a STRUCT(INT a,b) might appear as:

```
@42 (STRUCT#5) A@43 B@44   @43 (INT#1) 1   @44
    (INT#1) 2
```

Row values consist of two parts. The first is the row value, which is a descriptor. The second is the bunch, which is the collection of values the row accesses. The row value is given in two parts: the mode, which includes the bounds of the row; and the mapping, which consists of sets of subscript values and the item number that they access. For example, a [2] INT might appear as:

```
@52 (ROW#4 1:2) (1)@96 (2)@97
```

The bunch, which usually appears in a separate place in the dump, might look like:

```
@96 (INT) 5   @97 (INT) -3
```

A flat row (one with a lower bound that is greater than the corresponding upper bound) is treated specially. RR requires that a 'ghost' element be allocated so that bounds checking can 'look past' the flat row to the element in the row of row case. In the case of a flat row of integers, the following might be given:

```
@12 (ROW#5 1:0) FLAT @66
...
@66 (GHOST) (INT#1) 0
```

The reason for this elaborate and very detailed format of dump is to explicitly show all of the mappings of names (refs), structs and rows onto values. In particular, where subnames exist, as in slices of rows, row selections from rows of structs, field selections from structs, or simple ascriptions such as ref parameters, the dump shows them.

11.12. Branch Trace

FLACC keeps a small (30 entry) table of the most recent branches taken during program execution. These branches occur in IF and CASE clauses, in loops, during calls, and, of course, during gotos.

This table allows you to see where the program 'came from' immediately prior to termination.

It appears if the value of DUMP (or EDUMP in the case of abnormal termination) is at least 1.

Each entry in the table consists of a source coordinate. Thus a (very short) table might look like:

(00005,00008)->(00010,00017)->(00000,00000)

The last value in the table is always zero.

11.13. Execution Profile

When the PROFILE parameter is in effect, FLACC counts how many times each 'branch point' in the program is passed (see Branch Trace, above).

After termination, a bar graph is given showing each branch point encountered, its count, the ratio of this count to the maximum one found (given as a percentage), and a histogram of asterisks (*). Each branch point is given on a separate line.

Appendix A: Standard Prelude

All operators, tags and routines defined in RR are supported by FLACC, although some representations of some operators are not. This appendix lists (in EBCDIC order) all standard prelude tags, routines and operators defined in FLACC. Any that are marked with * are not defined in RR. Each indicant not defined in RR is explained following the list.

Occurrences of L represent sets of LONG and SHORT, and occurrences of l are sets of long and short. That is, L INT means INT, LONG INT, SHORT INT, etc.

```
<                    * deadlock exception   l pi                  whole
<=                   * divide by zero       print                 write
+                      exception            * printer channel     write bin
+*                   * l e                    printf              writef
+:=                    errorchar              put                   ABS
+=:                    estab possible         put bin               AND
*                      establish              put possible          ARG
**                     l exp                  putf                * ASSERT
*:=                    l exp width          * raise                 BIN
-                      fixed                  l random              CONJ
-:=                    flip                 * range exception       DIVAB
/                      float                  read                  DOWN
/:=                    flop                   read bin              ELEM
/=                     get                    readf                 ENTIER
%                      get bin                real lengths          EQ
%*                     get possible           real shorths        * EXCEPTION
%*:=                   getf                   l real width          GE
%:=                    int lengths            reidf                 GT
>                      int shorths            reidf possible        I
>=                     l int width            reset                 IM
=                      l last random          reset possible        LE
* any exception       line number          * return code           LENG
  l arccos             l ln                 * rpad                  LEVEL
  l arcsin           * l ln 10              * sameline              LT
  l arctan             lock                 * scientific            LWB
  associate          * lpad                   scratch               MINUSAB
  backspace            make conv            * sequential channel    MOD
  bin possible         make term              set                   MODAB
  bits lengths         max abs char           set char number       NE
  l bits pack          l max int              set possible          NOT
  bits shorths         l max real           * set return code       ODD
  l bits width       * memory limit           l sin                 OR
  blank                exception              l small real          OVER
* bounds exception    * new exception          space                 OVERAB
  bytes lengths        newline                l sqrt                PLUSAB
  l bytes pack         newpage                stand back            PLUSTO
  bytes shorths        l next random          stand back channel    RE
  l bytes width        null character         stand in              REPR
  chan                 on char error          stand in channel      ROUND
  char in string     * on error               stand out             SHL
  char number        * on file end            stand out channel     SHORTEN
* clipline             on format end          standconv             SHR
  close                on line end            stop                  SIGN
  compressible         on logical file end    l tan '               TIMESAB
* convert              on page end          * time of day           UP
  l cos                on physical file end * trace                 UPB
* l cot                on value error       * tracef
* cpu time             open                 * trace flag
  create             * overflow exception   * transput exception
* date                 page number          * underflow exception
```

any exception (EXCEPTION) allows the program to recover from (almost) any error. See Section 10.

bounds exception (EXCEPTION) allows the program to recover from a bounds error. See Section 10.

clipline (PROC (REF FILE) VOID) is used on random access files. It truncates the current line to the current position, but performs no other action. Thus lines can be shortened by calling clipline then newline (or some other layout routine), or lengthened by calling clipline, then put, then newline (or some other layout routine).

convert (PROC (?NUMBER) STRING) converts its argument to its 'most readable' string representation. This is a very useful intermediate level of output formatting, especially when used in free-format messages, or in conjunction with lpad.

cot (PROC (L REAL) L REAL) is the multiplicative inverse of tan.

cpu time (PROC INT) returns the number of milliseconds of CPU time that the program has used since it started execution.

date (PROC STRING) returns a string of the form DD/MMM/YY, where DD is the day, MMM is the month, and YY is the year (for example, 03/JUL/68).

deadlock exception (EXCEPTION) allows the program to recover from deadlocks in parallel clauses. See Section 10.

divide by zero exception (EXCEPTION) allows the program to recover from division errors. See Section 10.

e (L REAL) is 2.71828....

ln 10 (L REAL) is ln(10).

lpad (PROC (STRING, INT, CHAR) STRING) pads its first argument to the length given by its second argument, using the character given by its third argument. Padding occurs on the left. If the string is already long enough, no action is taken.

memory limit exception (EXCEPTION) allows the program to recover from exhausting its available work space. See Section 10. Unless memory runs out because a row or struct is too large, at least one environ must be abandoned so that some space is available for the program to continue.

new exception (PROC EXCEPTION) creates a new exception value. See Section 10. new exception is normally used in a declaration.

on error (PROC (EXCEPTION, PROC VOID) VOID) is used to set up exception handlers. See Section 10.

on file end (PROC (REF FILE, PROC (REF FILE) BOOL) VOID) combines the functions of on logical file end and on physical file end.

overflow exception (EXCEPTION) allows the program to recover from arithmetic overflows. See Section 10.

printer channel (CHANNEL) is an additional channel (Section 6) which supports only sequential output, as well as overprinting and page skipping.

raise (PROC (EXCEPTION) VOID) is used to raise an exception. See Section 10.

range exception (EXCEPTION) allows the program to recover from out-of-range arguments to standard prelude routines. See Section 10.

return code (PROC INT) yields the return code of the last external routine called. See Section 9.

rpad (PROC (STRING, INT, CHAR) STRING) pads its first argument to the length given by its second argument, using the character given by its third argument. Padding occurs on the right. If the string is already long enough, no action is taken.

sameline (PROC (REF FILE) VOID) is used for overprinting. To overprint two lines, write the first line, then sameline, then the second line, and finally newline.

scientific (PROC (?NUMBER, INT) STRING) converts its first argument to a string in scientific notation, with the number of digits specified by its second argument. This is not the same as float, which may return more than one digit before the decimal point.

sequential channel (CHANNEL) is an additional channel (see Section 6) which supports sequential input and output.

set return code (PROC (INT) VOID) sets the return code that FLACC will give on completion of the run.

time of day (PROC STRING) returns a string of the form HH:MM:SS, where HH is the hour (24 hour clock), MM is the minute, and SS is the second (for example, 14:22:55).

trace (same mode as print) is used for tracing. It produces output only when trace flag is TRUE. Its output is sent to standout.

tracef (same mode as printf) is used for tracing. It produces output only when trace flag is TRUE. Its output is sent to standout.

trace flag (REF BOOL) controls whether trace and tracef produce output.

transput exception (EXCEPTION) allows the program to recover from errors during transput. See Section 10.

underflow exception (EXCEPTION) allows the program to recover from arithmetic underflows. See Section 10.

ASSERT (PROC (BOOL) VOID) is a unary operator. It terminates execution if its argument is FALSE, and does nothing if it is TRUE.

EXCEPTION (mode indicant) is a mode used for exception handling. See Section 10.

In Loving Memory of Barry Mailloux

Barry J. Mailloux was a driving force in the development of the Algol 68 language.

Aad van Wijngaarden charged Barry with the heavy responsibility of ensuring that Algol 68 was a practical design, one that could be implemented efficiently on the computers of the day.

Barry's 1968 doctoral dissertation "On the Implementation of Algol 68" addressed all of the key issues:

- multi-pass compilation strategy,
- symbol table organization,
- object code structure,
- run-time memory management, and
- run-time representation of modes.

When Barry returned to the University of Alberta as an Assistant Professor, he made it his mission to see that plan realized.

Barry was a fabulous mentor and a source of constant encouragement during the development of FLACC. He caused it all to happen in so many ways.

We hope that FLACC will be remembered as "Mailloux's Algol 68" – Chris & Colin