THE INTERCAL PROGRAMMING LANGUAGE
REVISED REFERENCE MANUAL

Donald R. Woods
and
James M. Lyon

C-INTERCAL revisions:
Louis Howell
and
Eric S. Raymond

1. INTRODUCTION

> "But I don't want to go among mad people," Alice remarked.
>
> "Oh, you ca'n't help that," said the Cat: "We're all mad here. I'm mad. You're mad."
>
> "How do you know I'm mad?" said Alice.
>
> "You must be," said the Cat, "or you wouldn't have come here."

The names you are about to ignore are true. However, the story has been changed significantly. Any resemblance of the programming language portrayed here to other programming languages, living or dead, is purely coincidental.

1.1 Origin and Purpose

The INTERCAL programming language was designed the morning of May 26, 1972 by Donald R. Woods and James M. Lyon, at Princeton University. Exactly when in the morning will become apparent in the course of this manual.

Eighteen years later (give or take a few months) Eric S. Raymond perpetrated a UNIX-hosted INTERCAL compiler as a weekend hack. The C-INTERCAL implementation has since been maintained and extended by an international community of technomasochists, including Louis Howell, Steve Swales, Michael Ernst, and Brian Raiter.

(There was evidently an Atari implementation sometime between these two; notes on it got appended to the INTERCAL-72 manual. The culprits have sensibly declined to identify themselves.)

INTERCAL was inspired by one ambition: to have a compiler language which has nothing at all in common with any other major language. By 'major' was meant anything with which the authors were at all familiar, e.g., FORTRAN, BASIC, COBOL, ALGOL, SNOBOL, SPITBOL, FOCAL, SOLVE, TEACH, APL, LISP, and PL/I. For the most part, INTERCAL has remained true to this goal, sharing only the basic elements such as variables, arrays, and the ability to do I/O, and eschewing all conventional operations other than the assignment statement (FORTRAN "=").

1.2 Acronym

The full name of the compiler is "Compiler Language With No Pronounceable Acronym", which is, for obvious reasons, abbreviated "INTERCAL".

1.3 Acknowledgments

The authors are deeply indebted to Eric M. Van and Daniel J. Warmenhoven, without whose unwitting assistance this manual would still have been possible.

2. *FUNDAMENTAL CONCEPTS*

> "There's nothing like eating hay when you're faint," the White King remarked to Alice, as he munched away.
>
> "I should think throwing cold water over you would be better," Alice suggested: "-- or some sal-volatile."
>
> "I didn't say there was nothing *better*," the King replied. "I said there was nothing *like* it." Which Alice did not venture to deny.

In this section an attempt is made to describe how and why INTERCAL may be used; i.e., what it is like and what it is good for.

2.1 Sample Program

Shown below is a relatively simple INTERCAL program which will read in 32-bit unsigned integers, treat them as signed, 2's-complement numbers, and print out their absolute values. The program exits if the absolute value is zero. Note in particular the inversion routine (statements 6 through 14), which could be greatly simplified if the subroutine library (see section) were used.

A more detailed analysis of a program is made in section
of this manual.

```
        DO (5) NEXT
    (5) DO FORGET #1
        PLEASE WRITE IN :1
        DO .1 <- '∀":1~'#32768¢#0'"¢#1'~#3
        DO (1) NEXT
        DO :1 <- "'∀":1~'#65535¢#0'"¢#65535'
                ~'#0¢#65535'"¢"'∀":1~'#0¢#65535'"
                ¢#65535'~'#0¢#65535'"
        DO :2 <- #1
        PLEASE DO (4) NEXT
    (4) DO FORGET #1
        DO .1 <- "∀':1~:2'¢#1"~#3
        DO :1 <- "'∀":1~'#65535¢#0'"¢":2~'#65535
                ¢#0'"'~'#0¢#65535'"¢"'∀":1~'#0
                ¢#65535'"¢":2~'#0¢#65535'"'~'#0¢#65535'"
        DO (1) NEXT
        DO :2 <- ":2~'#0¢#65535'"
                ¢"'":2~'#65535¢#0'"¢#0'~'#32767¢#1'"
        DO (4) NEXT
    (2) DO RESUME .1
    (1) PLEASE DO (2) NEXT
        PLEASE FORGET #1
        DO READ OUT :1
        PLEASE DO .1 <- '∀"':1~:1'~#1"¢#1'~#3
        DO (3) NEXT
        PLEASE DO (5) NEXT
    (3) DO (2) NEXT
        PLEASE GIVE UP
```

2.2 Uses for INTERCAL

INTERCAL's main advantage over other programming languages is its strict simplicity. It has few capabilities, and thus there are few restrictions to be kept in mind. Since it is an exceedingly easy language to learn, one might expect it would be a good language for initiating novice programmers. Perhaps surprising, then, is the fact that it would be more likely to initiate a novice into a search for another line of work. As it turns out, INTERCAL is more useful (which isn't saying much) as a challenge to professional programmers. Those who doubt this need only refer back to the sample program in section . This 23-statement program took somewhere from 15 to 30 minutes to write, whereas the same objectives can be achieved by single-statement programs in either SNOBOL;

```
PLEASE INPUT POS(0) ('-' ! '')
+ (SPAN('0123456789') $ OUTPUT)
+ *NE(OUTPUT) :S(PLEASE)F(END)
```

or APL;

```
[1] →0≠□←⌶□
```

Admittedly, neither of these is likely to appear more intelligible to anyone unfamiliar with the languages involved, but they took roughly 60 seconds and 15 seconds, respectively, to write. Such is the overwhelming power of INTERCAL!

The other major importance of INTERCAL lies in its seemingly inexhaustible capacity for amazing one's fellow programmers, confounding programming shop managers, winning friends, and influencing people. It is a well-known and oft-demonstrated fact that a person whose work is incomprehensible is held in high esteem. For example, if one were to state that the simplest way to store a value of 65536 in a 32-bit INTERCAL variable is:

```
DO :1 <- #0¢#256
```

any sensible programmer would say that that was absurd. Since this is indeed the simplest method, the programmer would be made to look foolish in front of his boss, who would of course happened to turn up, as bosses are wont to do. The effect would be no less devastating for the programmer having been correct.

3. DESCRIPTION

> "I quite agree with you," said the Duchess; "and the moral of that is -- 'Be what you would seem to be' -- or, if you'd like it put more simply -- 'Never imagine yourself not to be otherwise than what it might appear to others that what you were or might have been was not otherwise than what you had been would have appeared to them to be otherwise."
> "I think I should understand that better," Alice said very politely, "if I had it written down: but I ca'n't quite follow it as you say it."
>
> "That's nothing to what I could say if I chose," the Duchess replied, in a pleased tone.

The examples of INTERCAL programming which have appeared in the preceding sections of this manual have probably seemed highly esoteric to the reader unfamiliar with the language. With the aim of making them more so, we present here a description of INTERCAL.

3.1 Variables

INTERCAL allows only 2 different types of variables, the **16-bit integer** and the **32-bit integer.** These are represented by a spot (.) or two-spot (:), respectively, followed by any number between 1 and 65535, inclusive. These variables may contain only non-negative numbers; thus they have the respective ranges of values: 0 to 65535 and 0 to 4294967295. Note: .123 and :123 are two distinct variables. On the other hand, .1 and .0001 are identical. Furthermore, the latter may *not* be written as 1E-3.

3.2 Constants

Constants are 16-bit values only and may range from 0 to 65535. They are prefixed by a mesh (#). Caution! Under no circumstances confuse the mesh with the interleave operator, except under confusing circumstances!

3.3 Arrays

Arrays are represented by a tail (,) for 16-bit values, or a hybrid (;) for 32-bit values, followed by a number between 1 and 65535, inclusive. The number is suffixed by the word SUB, followed by the subscripts, separated optionally by spaces. Subscripts may be any expressions, including those involving subscripted variables. This occasionally leads to ambiguous constructions, which are resolved as discussed in section . Definition of array dimensions will be discussed later in greater detail, since discussing it in less detail would be difficult. As before, ,123 and ;123 are distinct. In summary, .123, :123, #123, ,123, and ;123 are all distinct.

3.4 Operators

INTERCAL recognizes 5 operators -- 2 binary and 3 unary [1]. Please be kind to our operators: they may not be very intelligent, but they're all we've got. In a sense, all 5 operators are binary, as they are all bit-oriented, but it is not our purpose here to quibble about bits of trivia.

3.4.1 Binary Operators

The binary operators are **interleave** (also called **mingle**) and **select**, which are represented by a change (¢) and a sqiggle [sic] (~), respectively. (In C-INTERCAL's ASCII environment, EBCDIC ¢ is replaced by a big money ($), but ¢ can be used if encoded in Latin-1 as \0xA2 or UTF-8 as 0xC2A2.)

The interleave operator takes two 16-bit values and produces a 32-bit result by alternating the bits of the operands. Thus, #65535¢#0 has the 32-bit binary form 101010....10 or 2863311530 decimal, while #0¢#65535 = 0101....01 binary = 1431655765 decimal, and #255¢#255 is equivalent to #65535.

The select operator takes from the first operand whichever bits correspond to 1's in the second operand, and packs these bits to the right in the result. Both operands are automatically padded on the left with zeros to 32 bits before the selection takes place, so the variable types are unrestricted. If more than 16 bits are

---

1. Other operators are also available with C-INTERCAL's non-standard dialects; see section .

selected, the result is a 32-bit value, otherwise it is a 16-bit value. For example, #179˜#201 (binary value 10110011˜11001001) selects from the first argument the 8th, 7th, 4th, and 1st from last bits, namely, 1001, which = 9. But #201˜#179 selects from binary 11001001 the 8th, 6th, 5th, 2nd, and 1st from last bits, giving 10001 = 17. #179˜#179 has the value 31, while #201˜#201 has the value 15.

Perhaps a simpler way of understanding the operation of the select operator would be to examine the logic diagram in Figure 1, which performs the select operation upon two 8-bit values, A and B. The gates used are Warmenhovian logic gates, which means the outputs have four possible values: low, high, undefined (value of an uninitialized flip-flop), and oscillating (output of a NOR gate with one input low and the other input connected to the output). These values are represented symbolically by '0', '1', '?', and '∅'. Note in particular that, while NOT-0 is 1 and NOT-1 is 0 as in two-valued logic, NOT-? is ? and NOT-∅ is ∅. The functions of the various gates are listed in Table 1.

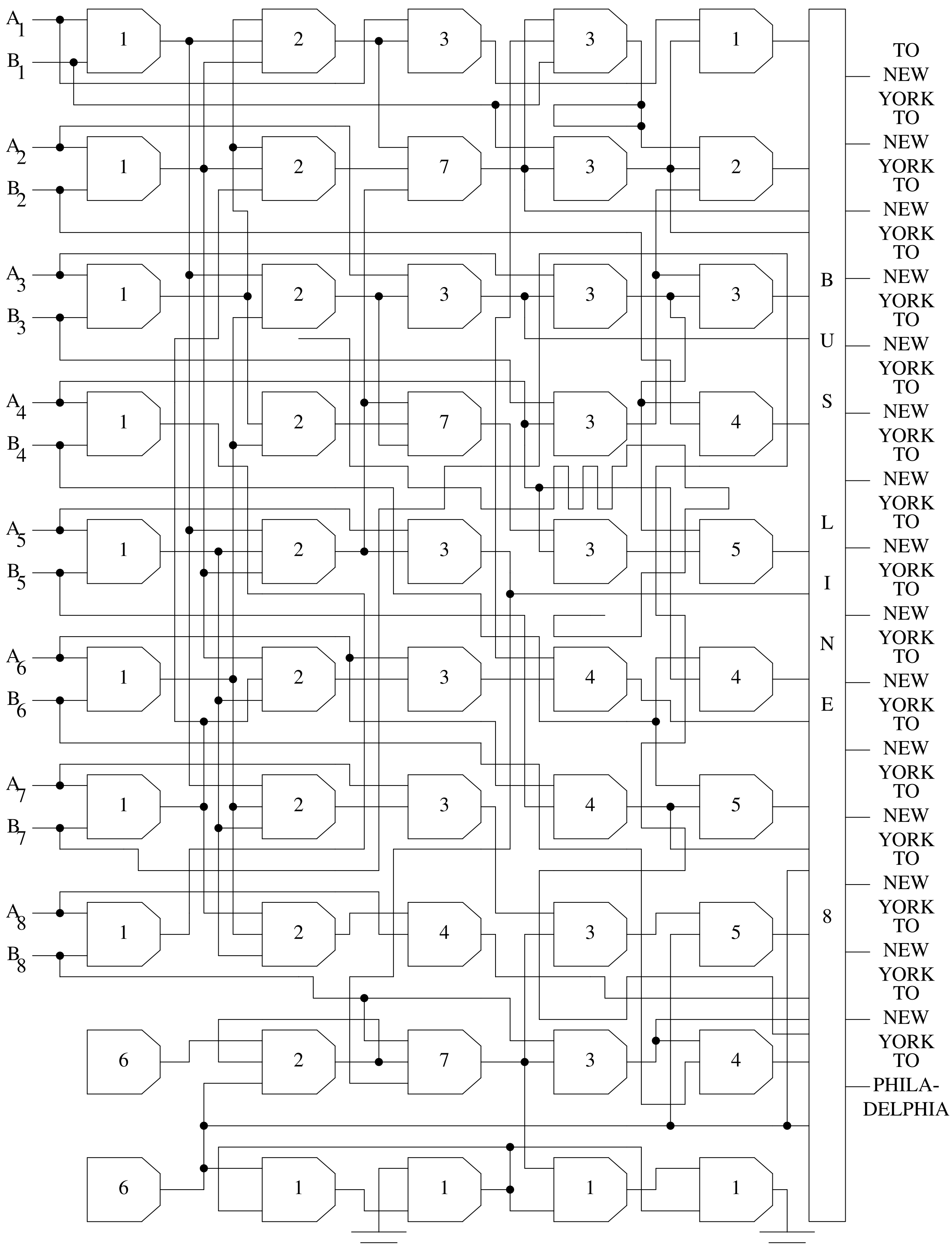


FIGURE 1. CIRCUITOUS DIAGRAM

1. Logic gate. Inputs A, B. Output O = AB.
2. Logic gate. Inputs A, B, C. Output O = A+BC.
3. Logic gate. Inputs A, B. Output O = A+AB.
4. Logic gate. Inputs A, B. Output O = AB ⊕ -(A+-B)
5. Logic gate. Inputs A, B. Output O = A⊕A + AA
6. Uninitialized flip-flop. Inputs none. Output O = ?
7. Flip-flop-flap. Inputs A, B, C. Output O = 1 if A=0 or B+C=0 and A=1. O = 0 if AC=1, B=0. O = ∅ if AB=1, C=0. O = ? if ABC=1. O as yet undetermined for other Warmenhovian inputs. See Figure 2.
8. Bus line.

Table 1. Logical (and other) functions.

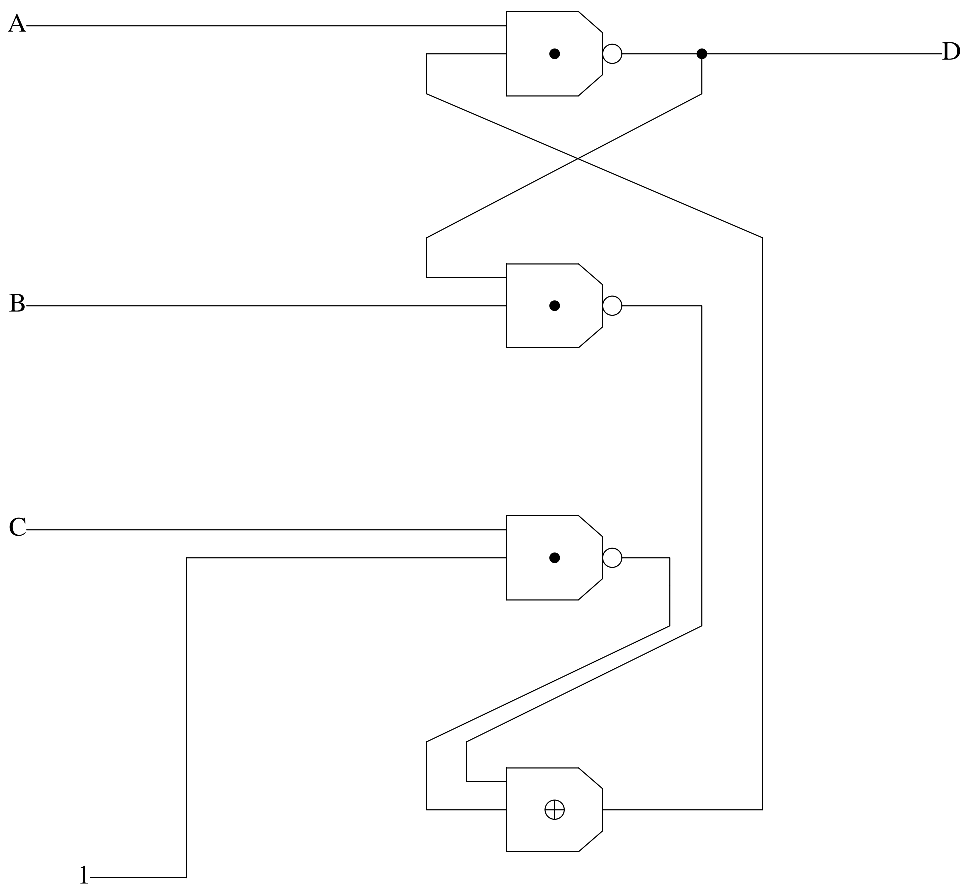
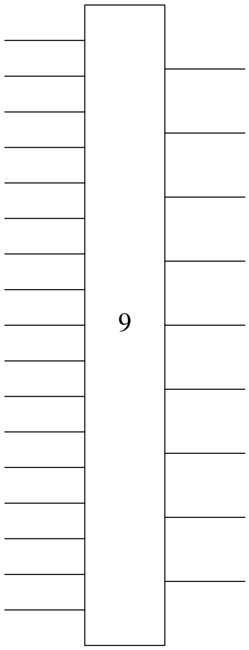


FIGURE 2. FLIP FLAP FLOP



GATE TYPE 9. BLACK BOX
INPUTS A1-8, B1-8.
OUTPUT O1-8 = A1-8˜B1-8

FIGURE 1 (CONTINUED). NEW YORK

3.4.1.1 Return type of SELECT

INTERCAL-72 defined the return type of a select operation to depend on the number of bits selected. The C-INTERCAL compiler takes the easier route of defining the return type to be that of the right operand, independent of its actual value. This form has the advantage that all types can be determined at compile time. Putting in run-time type-checking would add significant overhead and complication, to effect a very minor change in language semantics.

The only time this distinction makes any difference is when a unary operator is applied to the select result. This happens extremely rarely in practice, the only known instance being the 32-bit greater-than test in the standard library, where an XOR operator is applied to the result of selecting a number against itself. The INTERCAL-72 authors first select the result against #65535¢#65535 to insure that XOR sees a 32-bit value. With the current compiler this extra step is unnecessary, but harmless.

The cautious programmer should write code that does not depend on the compiler version being used. We therefore suggest the following guideline for determining the select return type:

A select operation with a 16-bit right operand returns a 16-bit value. The return type of a select operation with a 32-bit right operand is undefined, but is guaranteed to be an acceptable input to a mingle operation so long as 16 or fewer bits are actually selected. Correct code should not depend on whether the return type is 16 or 32 bits.

3.4.2 Unary Operators

The unary operators are & (logical **AND**), V (logical **OR**), and ∀ (logical **XOR**). This last character is obtained by overpunching a worm (–) on a V (V). (In C-INTERCAL, ∀ is replaced by what (?). The compiler recognizes V<backspace>- as a valid equivalent, in case you are concerned about compatibility with the Princeton compiler. In version 0.20 and later, the Unicode character U+2200 FOR ALL, which looks exactly like V<backspace>- or an upside-down 'A' is also accepted in UTF-8 form.) The operator is inserted between the spot, two-spot, mesh, or what-have-you, and the integer, thus: .&123, #∀123. Multiple unary operators may not be concatenated, thus the form #∀&123 is invalid. This will be covered later when precedence is discussed. These operators perform their respective logical operations on all pairs of adjacent bits, the result from the first and last bits going into the first bit of the result. The effect is that of rotating the operand one place to the right and ANDing, ORing, or XORing with its initial value. Thus, #&77 (binary = 1001101) is binary 0000000000000100 = 4, #V77 is binary 1000000001101111 = 32879, and #∀77 is binary 1000000001101011 = 32875.

3.4.3 Precedence

Precedence of operators is as follows:

(The remainder of this page intentionally left blank)[2]

2. Keep in mind that the aim in designing INTERCAL was to have no precedents.

This precedence (or lack thereof) may be overruled by grouping expressions between pairs of sparks (’) or rabbit-ears ("). Thus ’#165¢#203’~#358 (binary value ’10100101¢11001011’~101100110) has the value 15, but #165¢’#203~#358’ has the value 34915, and #165¢#203~#358 is invalid syntax and is completely valueless (except perhaps as an educational tool to the programmer). A unary operator is applied to a sparked or rabbit-eared expression by inserting the operator immediately following the opening spark or ears. Thus, the invalid expression #∀&123, which was described earlier, could be coded as ’∀#&123’ or ’∀"&#123"’. Note: In the interests of simplifying the sometimes overly-complex form of expressions, INTERCAL allows a spark-spot combination (’.) to be replaced with a wow (!). Thus ’.1~.2’ is equivalent to !1~.2’, and ’V.1¢.2’ is equivalent to "V!1¢.2’".

Combining a rabbit-ears with a spot to form a rabbit (⁚) is not permitted, although the programmer is free to use it should he find an EBCDIC reader which will properly translate a 12-3-7-8 punch.

Sparks and/or rabbit-ears must also be used to distinguish among such otherwise ambiguous subscripted and multiply-subscripted expressions as:

```
,1 SUB #1 ~ #2
,1 SUB ,2 SUB #1 #2 #3
,1 SUB " ,2 SUB " ,3 SUB #1 " #2 " " #3 "
```

The third case may be isolated into either of its possible interpretations by simply changing some pairs of rabbit-ears to sparks, instead of adding more ears (which would only confuse the issue further). Ambiguous cases are defined as those for which the compiler being used finds a legitimate interpretation which is different from that which the user had in mind. See also section .

4. STATEMENTS

> "There's glory for you!"
>
> "I don't know what you mean by 'glory,'" Alice said.
>
> Humpty Dumpty smiled contemptuously. "Of course you don't -- till I tell you. I meant 'there's a nice knock-down argument for you!'"
>
> "But 'glory' doesn't mean 'a nice knock-down argument.'" Alice objected.
>
> "When *I* use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean -- neither more nor less."

In this section is described the format of INTERCAL statements.

4.1 General Format

Statements may be entered in 'free format'. That is, more than one statement may occur on a single card, and a statement may begin on one card and end on a later one. Note that if this is done, all intervening cards and portions thereof must be part of the same statement. That this restriction is necessary is immediately apparent from the following example of what might occur if statements could be interlaced.

```
DO .1 <- ".1¢'&:51~"#∀1c!12~;&75SUB"∀'V.1~
DO .2 <- '"!1¢"&';∀79SUB",&7SUB:173"'~!∀9¢
.2'¢,&1SUB:5~#33578"'"'"~'#65535¢"∀'V#&85'"'
#8196'"'~.1"¢.2'~'#&5¢"'#1279¢#4351'~#65535"'
```

The above statements are obviously meaningless. (For that matter, so are the statements

```
DO .1 <- ".1¢"&:51~"#∀1¢!12~;&75SUB"∀'V.1~
.2'¢,&1SUB:5~#333578"'"'"~#65535¢"∀'V#&85'"'
DO .2 <- '"!1¢"&';∀79SUB",&7SUB:173"'~!∀9¢
#8196'"'~.1"¢.2'~'#&5¢"'#1279¢#4351'~#65535"'
```

but this is not of interest here.)

Spaces may be used freely to enhance program legibility (or at least reduce program illegibility), with the restriction that no word of a statement identifier (see section) may contain any spaces.

4.2 Labels

A statement may begin with a **logical line label** enclosed in wax-wane pairs (()). A statement may not have more than one label, although it is possible to omit the label entirely. A line label is any integer from 1 to 65535, which must be unique within each program. The user is cautioned, however, that many line labels between 1000 and 1999 are used in the INTERCAL System Library functions.

4.3 Identifiers and Qualifiers

After the line label (if any), must follow one of the following statement identifiers: DO, PLEASE, or PLEASE DO. These may be used interchangeably to improve the aesthetics of the program. The identifier is then followed by either, neither, or both of the following optional parameters (qualifiers): (1) either of the character strings NOT or N'T, which causes the statement to be automatically abstained from (see section) when execution begins, and (2) a number between 0 and 100, preceded by a double-oh-seven (%), which causes the statement to have only the specified percent chance of being executed each time it is encountered in the course of execution.

4.4 Statements

Following the qualifiers (or, if none are used, the identifier) must occur one of the 14 valid operations. (Exception: see section .) These are described individually in sections
through .

4.4.1 Calculate

The INTERCAL equivalent of the half-mesh (=) in FORTRAN, BASIC, PL/I, and others, is represented by an angle (<) followed by a worm (-). This combination is read 'gets'. 32-bit variables may be assigned 16-bit values, which are padded on the left with 16 zero bits. 16-bit variables may be assigned 32-bit values only if the value is less than 65535. Thus, to invert the least significant bit of the first element of 16-bit 2-dimensional array number 1, one could write:

```
,1SUB#1#1 <- '∀",1SUB#1#1"¢#1'˜'#0¢#65535'
```

Similarly to SNOBOL and SPITBOL, INTERCAL uses the angle-worm to define the dimensions of arrays. An example will probably best describe the format. To define 32-bit array number 7 as 3-dimensional, the first dimension being seven, the second being the current value of 16-bit variable number seven, and the third being the current value of the seventh element of 16-bit array number seven (which is one-dimensional) mingled with the last three bits of 32-bit variable number seven, one would write (just before they came to take him away):

```
;7 <- #7 BY .7 BY ",7SUB#7"¢':7˜#7'
```

This is, of course, different from the statement:

```
;7 <- #7 BY .7 BY ,7SUB"#7¢':7˜#7'"
```

INTERCAL also permits the redefining of array dimensioning, which is done the same way as is the initial dimensioning. All values of items in an array are lost upon redimensioning, unless they have been STASHed (see section), in which case restoring them also restores the old dimensions.

4.4.2 NEXT

The NEXT statement is used both for subroutine calls and for unconditional transfers. This statement takes the form:

DO (label) NEXT

(or, of course,

PLEASE DO (label) NEXT

etc.), where (label) represents any logical line label which appears in the program. The effect of such a statement is to transfer control to the statement specified, and to store in a push down list (which is initially empty) the location from which the transfer takes place. Items may be removed from this list and may be discarded or used to return to the statement immediately following the NEXT statement. These operations are described in sections
and
respectively. The programmer is generally advised to discard any stack entries which he does not intend to utilize, since the stack has a maximum depth of 79 entries. A program's attempting to initiate an 80th level of NEXTing will result in the fatal error message, "PROGRAM HAS DISAPPEARED INTO THE BLACK LAGOON."

4.4.3 FORGET

The statement PLEASE FORGET exp, where exp represents any expression (except colloquial and facial expressions), causes the expression to be evaluated, and the specified number of entries to be removed from the NEXTing stack and discarded. An attempt to FORGET more levels of NEXTing than are currently stacked will cause the stack to be emptied, and no error condition is indicated. This is because the condition is not considered to be an error. As described in section , it is good programming practice to execute a DO FORGET #1 after using a NEXT statement as an unconditional transfer, so that the stack does not get cluttered up with unused entries:

```
        DO (123) NEXT
        .
        .
(123)   DO FORGET #1
```

4.4.4 RESUME

The statement PLEASE RESUME exp has the same effect as FORGET, except that program control is returned to the statement immediately following the NEXT statement which stored in the stack the last entry to be removed. Note that a rough equivalent of the FORTRAN computed GO TO and BASIC ON exp GO TO is performed by a sequence of the form:

```
        DO (1) NEXT
        .
        .
  (1)   DO (2) NEXT
        PLEASE FORGET #1
        .
        .
  (2)   DO RESUME .1
```

Unlike the FORGET statement, an attempt to RESUME more levels of NEXTing than has been stacked will cause program termination. See also section .

4.4.5 STASH

Since subroutines are not explicitly implemented in INTERCAL, the NEXT and RESUME statements must be used to execute common routines. However, as these routines might use the same variables as the main program, it is necessary for them to save the values of any variables whose values they alter, and later restore them. This process is simplified by the STASH statement, which has the form DO STASH list, where list represents a string of one or more variable or array names, separated by intersections (+). Thus

PLEASE STASH .123+:123+,123

stashes the values of two variables and one entire array. The values are left intact, and copies thereof are saved for later retrieval by (what else?) the RETRIEVE statement (see section). It is not possible to STASH single array items.

4.4.6 RETRIEVE

PLEASE RETRIEVE list restores the previously STASHed values of the variables and arrays named in the list. If a value has been stashed more than once, the most recently STASHed values are RETRIEVEd, and a second RETRIEVE will restore the second most recent values STASHed. Attempting to RETRIEVE a value which has not been STASHed will result in the error message, "THROW STICK BEFORE RETRIEVING."

4.4.7 IGNORE

The statement DO IGNORE list causes all subsequent statements to have no effect upon variables and/or arrays named in the list. Thus, for example, after the sequence

```
DO .1 <- #1
PLEASE IGNORE .1
DO .1 <- #0
```

16-bit variable number 1 would have the value 1, not 0. Inputting (see section) into an IGNOREd variable also has no effect. The condition is annulled via the REMEMBER statement (see section). Note that, when a variable is being IGNOREd, its value, though immutable, is still available for use in expressions and the like.

Though the INTERCAL-72 manual laid down that the value of an IGNOREd variable cannot change, it was unclear about whether or not a statement which appears to change an IGNOREd variable is executed or not. This may appear to be a "If a tree falls in the forest ..." type of question, but if the statement in question has other side effects it is not.

Since another mechanism already exists for ABSTAINing from a statement, C-INTERCAL's IGNORE only prevents the changing of the specific variable in question, not the execution of the entire statement. In the present version of the language this only makes a difference for the WRITE IN (see) and RETRIEVE (see) statements. Attempting to WRITE IN to an IGNOREd variable will cause a number to be read from the input, which will be discarded since it cannot be stored in the variable. Using RETRIEVE on an IGNOREd variable will pop the variable's stash stack without actually changing its value.

4.4.8 REMEMBER

PLEASE REMEMBER list terminates the effect of the IGNORE statement for all variables and/or arrays named in the list. It does not matter if a variable has been IGNOREd more than once, nor is it an error if the variable has not been IGNOREd at all.

4.4.9 ABSTAIN

INTERCAL contains no simple equivalent to an IF statement or computed GO TO, making it difficult to combine similar sections of code into a single routine which occasionally skips around certain statements. The IGNORE statement (see section) is helpful in some cases, but a more viable method is often required. In keeping with the goal of INTERCAL having nothing in common with any other language, this is made possible via the ABSTAIN statement.

This statement takes on one of two forms. It may not take on both at any one time. DO ABSTAIN FROM (label) causes the statement whose logical line label is (label) to be abstained from. PLEASE ABSTAIN FROM gerund list causes all statements of the specified type(s) to be abstained from, as in

```
      PLEASE ABSTAIN FROM STASHING
      PLEASE ABSTAIN FROM IGNORING + FORGETTING
      PLEASE ABSTAIN FROM NEXTING
or    PLEASE ABSTAIN FROM CALCULATING
```

Statements may also be automatically abstained from at the start of execution via the NOT or N'T parameter (see section).

If, in the course of execution, a statement is encountered which is being abstained from, it is ignored and control passes to the next statement in the program (unless it, too, is being abstained from).

The statement DO ABSTAIN FROM ABSTAINING is perfectly valid, as is DO ABSTAIN FROM REINSTATING (although this latter is not usually recommended). However, the statement DO ABSTAIN FROM GIVING UP is not accepted, even though DON'T GIVE UP is.

4.4.10 REINSTATE

The REINSTATE statement, like the ABSTAIN, takes as an argument either a line label or a gerund list. No other form of argument is permitted. For example, the following is an invalid argument:

Given: $x \neq 0$, $y \neq 0$, Prove: $x+y=0$
Since $x \neq 0$, then $x+1 \neq 1$, $x+a \neq a$, $x+y \neq y$.
But what is y? y is anything but 0.
Thus $x+y \neq$ anything but 0.
Since x+y cannot equal anything but 0, $x+y=0$.
Q.E.D.

REINSTATEment nullifies the effects of an abstention. Either form of REINSTATEment can be used to "free" a statement, regardless of whether the statement was abstained from by gerund list, line label, or NOT. Thus, PLEASE REINSTATE REINSTATING is not necessarily an irrelevant statement, since it might free a DON'T REINSTATE command or a REINSTATE the line label of which was abstained from. However, DO REINSTATE GIVING UP is invalid, and attempting to REINSTATE a GIVE UP statement by line label will have no effect. Note that this insures that DON'T GIVE UP will always be a "do-nothing" statement.

4.4.11 GIVE UP

PLEASE GIVE UP is used to exit from a program. It has the effect of a PLEASE RESUME #80. DON'T GIVE UP, as noted in section , is effectively a null statement.

4.4.12 Input

Input is accomplished with the statement DO WRITE IN list, where list represents a string of variables and/or elements of arrays, separated by intersections. Numbers are represented on cards, each number on a separate card, by spelling out each digit (in English) and separating the digits with one or more spaces. A zero (0) may be spelled as either ZERO or OH. Thus the range of (32-bit) input values permissible extends from ZERO (or OH) through FOUR TWO NINE FOUR NINE SIX SEVEN TWO NINE FIVE. (For the convenience of aviators, C-INTERCAL accepts the spelling NINER for NINE. In the service of internationalization, C-INTERCAL also accepts input digits in Sanskrit, Basque, Tagalog, Classical Nahuatl, Georgian, Kwakiutl, and Volapuk.)

Attempting to write in a value greater than or equal to SIX FIVE FIVE THREE SIX for a 16-bit variable will result in the error message, "DON'T BYTE OFF MORE THAN YOU CAN CHEW."

(See also section
for a description of C-INTERCAL's character I/O.)

4.4.13 Output

Values may be output to the printer, one value per line, via the statement DO READ OUT list, where the list contains variables, array elements, and/or constants. Output is in the form of "extended" Roman numerals (also called "butchered" Roman numerals), with an overline ($\overline{\ }$) indicating the value below is "times 1000", and lower-case letters indicating "times 1000000". Zero is indicated by an overline with no character underneath. Thus, the range of (32-bit) output values possible is from $\overline{\ }$ through $\overline{\mathrm{iv}}$ccxciv$\overline{\mathrm{CMLXVII}}$CCXCV. Note: For values whose residues modulo 1000000 are less than 4000, M is used to represent 1000; for values whose residues are 4000 or greater, $\overline{\mathrm{I}}$ is used. Thus #3999 would read out as MMMCMXCIX [3] while #4000 would readout as $\overline{\mathrm{IV}}$. Similar rules apply to the use of $\overline{\mathrm{M}}$ and i for 1000000, and to that of m and $\overline{\mathrm{i}}$ for 1000000000.

(See also section
for a description of C-INTERCAL's character I/O.)

4.4.14 COME FROM

In which we try to precisely define a statement that should never have been born, but is nevertheless one of the more useful statements in INTERCAL.

4.4.14.1 Background

The earliest known description of the COME FROM statement in the computing literature is in [R. L. Clark, "A linguistic contribution to GOTO-less programming," Commun. ACM 27 (1984), pp. 349-350], part of the famous April Fools issue of CACM. The subsequent rush by language designers to include the statement in their languages was underwhelming, one might even say nonexistent. It was therefore decided that COME FROM would be an appropriate addition to C-INTERCAL.

4.4.14.2 Description

There are two useful ways to visualize the action of the COME FROM statement. The simpler is to see that it acts like a GOTO when the program is traced backwards in time. More precisely, the statements

---

3. The original INTERCAL-72 manual claimed that #3999 should render as MMMIM, but the C-INTERCAL developers have been unable to find an algorithm that does this and is consistent with the rest of the rules.

```
(1)  DO <any statement>
      .
      .
      .
(2)  DO COME FROM (1)
```

should be thought of as being equivalent to

```
(1)  DO <any statement>
(2)  DO GOTO (3)
      .
      .
      .
(3)  DO NOTHING
```

if INTERCAL actually had a GOTO statement at all, which of course it doesn't.

What this boils down to is that the statement DO COME FROM (label), anywhere in the program, places a kind of invisible trap door immediately after statement (label). Execution or abstention of that statement is immediately followed by an unconditional jump to the COME FROM, unless the (label)ed statement is an executed NEXT, in which case the jump occurs if the program attempts to RESUME back to that NEXT statement. It is an error for more than one COME FROM to refer to the same (label).

Modification of the target statement by ABSTAIN or by the % qualifier affects only that statement, not the subsequent jump. Such modifications to the COME FROM itself, however, do affect the jump. Encountering the COME FROM statement itself, rather than its target, has no effect.

4.5 Comments

Unrecognizable statements, as noted in section , are flagged with a splat (*) during compilation, and are not considered fatal errors unless they are encountered during execution, at which time the statement (as input at compilation time) is printed and execution is terminated. This allows for an interesting (and, by necessity, unique) means of including comments in an INTERCAL listing. For example, the statement:

```
*    PLEASE NOTE THAT THIS LINE HAS NO EFFECT
```

will be ignored during execution due to the inclusion of the NOT qualifier. User-supplied error messages are also easy to implement:

```
*    DO SOMETHING ABOUT OVERFLOW IN ;3
```

as are certain simple conditional errors:

```
* (123) DON'T YOU REALIZE THIS STATEMENT SHOULD ONLY BE ENCOUNTERED
                ONCE?
        PLEASE REINSTATE (123)
```

This pair of statements will cause an error exit the second time they are encountered. Caution!! The appearance of a statement identifier in an intended comment will be taken as the beginning of a new statement. Thus, the first comment example above could not have been:

```
*    PLEASE NOTE THAT THIS LINE DOES NOTHING
```

The third example, however, is valid, despite the appearance of two cases of D-space-O, since INTERCAL does not ignore extraneous spaces in statement identifiers.

5. OUTSIDE COMMUNICATION

> She puzzled over this for some time, but at last a bright thought struck her. "Why, it's a Looking-glass book, of course! And, if I hold it up to a glass, the words will all go the right way again."

In which we try to remedy the fact that, due to I/O limitations, INTERCAL can not even in principle perform the same tasks as other languages. It is hoped that this addition will permit INTERCAL users to waste vast quantities of computer time well into the 21st century.

5.1 Motivation

One of the goals of INTERCAL was to provide a language which, though different from all other languages, is nevertheless theoretically capable of all the same tasks. INTERCAL-72 failed to accomplish this because its I/O functions could not handle arbitrary streams of bits, or even arbitrary sequences of characters. A language which can't even send its input directly to its output can hardly be considered as capable as other languages.

5.2 Turing Text Model

To remedy this problem, character I/O is now provided in a form based on the "Turing Text" model, originally proposed by Jon Blow. The C-INTERCAL programmer can access this capability by placing a one-dimensional array in the list of items given to a WRITE IN or READ OUT statement. On execution of the statement, the elements of the array will, from first to last, be either loaded from the input or sent to the output, as appropriate, in the manner described below. There is currently no support for I/O involving higher-dimensional arrays, but some form of graphics might be a possible 2-D interpretation.

The heart of the Turing Text model is the idea of a continuous loop of tape containing, in order, all the characters in the machine's character set. When a character is received by the input routine, the tape is advanced the appropriate number of spaces to bring that character under the tape head, and the number of spaces the tape was moved is the number that is actually seen by the INTERCAL program. Another way to say this is that the number placed in an INTERCAL array is the difference between the character just received and the previous character, modulo the number of characters in the machine character set.

Output works in just the opposite fashion, except that the characters being output come from the other side of the tape. From this position the characters on the tape appear to be in reverse order, and are individually backwards as well. (We would show you what it looks like, but we don't have a font with backwards letters available. Use your imagination.) The effect is that a number is taken out of an INTERCAL array, subtracted from the last character output -- i.e., the result of the last subtraction -- and then sent on down the output channel. The only catch is that the character as seen by the INTERCAL program is the mirror-image of the character as seen by the machine and the user. The bits of the character are therefore taken in reverse order as it is sent to the output. Note that this bit reversal affects only the character seen by the outside world; it does not affect the character stored internally by the program, from which the next output number will be subtracted. All subtractions are done modulo the number of characters in the character set.

Two different tapes are used for input and output to allow for future expansion of the language to include multiple input and output channels. Both tapes start at character 0 when a program begins execution. On input, when an end-of-file marker is reached the number placed in the array is one greater than the highest-numbered character on the tape.

5.3 Example Program

If all this seems terribly complicated, it should be made perfectly clear by the following example program, which simply maps its input to its output (like a simplified UNIX "cat"). It assumes that characters are 8 bits long, but that's fine since the current version of C-INTERCAL does too. It uses the standard library routines for addition and subtraction.

```
        DO ,1 <- #1
        DO .4 <- #0
        DO .5 <- #0
        DO COME FROM (30)
        DO WRITE IN ,1
        DO .1 <- ,1SUB#1
        DO (10) NEXT
        PLEASE GIVE UP
(20)    PLEASE RESUME '?.1$#256'~'#256$#256'
(10)    DO (20) NEXT
        DO FORGET #1
        DO .2 <- .4
        DO (1000) NEXT
        DO .4 <- .3~#255
        DO .3 <- !3~#15'$!3~#240'
        DO .3 <- !3~#15'$!3~#240'
        DO .2 <- !3~#15'$!3~#240'
        DO .1 <- .5
        DO (1010) NEXT
        DO .5 <- .2
        DO ,1SUB#1 <- .3
(30)    PLEASE READ OUT ,1
```

For each number received in the input array, the program first tests the #256 bit to see if the end of file has been reached. If not, the previous input character is subtracted off to obtain the current input character. Then the order of the bits is reversed to find out what character should be sent to the output, and the result is subtracted from the last character sent. Finally, the difference is placed in an array and given to a READ OUT statement. See? We told you it was simple!

6. TriINTERCAL

> "-- oh dear, how puzzling it all is! I'll try if I know all the things I used to know. Let me see: four times five is twelve, and four times six is thirteen, and four times seven is -- oh dear! I shall never get to twenty at that rate!"

In which it is revealed that bitwise operations are too ordinary for hard-core INTERCAL programmers, and extensions to other bases are discussed. These are not, strictly speaking, extensions to INTERCAL itself, but rather new dialects sharing most of the features of the parent language.

6.1 Motivation

INTERCAL is really a pretty sissy language. It tries hard to be different, but when you get right down to its roots, what do you find? You find bits, that's what. Plain old ones and zeroes, in groups of 16 and 32, just like every other language you've ever heard of. And what operations can you perform on these bits? The INTERCAL operators may arrange and permute them in weird and wonderful ways, but at the bit level the operators are the same AND, OR and XOR you've seen countless times before.

Once the prospective INTERCAL programmer masters the unusual syntax, she finds herself working with the familiar Boolean operators on perfectly ordinary unsigned integer words. Even the constants she uses are familiar. After all, who would not immediately recognize #65535 and #32768? It may take a just a moment more to figure out #65280, and #21845 and #43690 could be puzzles until she notices that they sum to #65535, but basically she's still on her home turf. The 16-bit limit on constants actually works in the programmer's favor by insuring that very long anonymous constants cannot appear in INTERCAL programs. And this is in a language that is supposed to be different from any other!

6.2 Abandon All Hope...

Standard INTERCAL is based on variables consisting of ordinary bits and familiar Boolean operations on those bits. In pursuit of uniqueness, it seems appropriate to provide a new dialect, otherwise identical to INTERCAL, which instead uses variables consisting of trits, i.e. ternary digits, and operators based on tritwise logical operations. This is intended to be a separate dialect, rather than an extension to INTERCAL itself, for a number of reasons. Doing it this way avoids word-length conflicts, does not spoil the elegance of the Spartan INTERCAL operator set, and dodges the objections of those who might feel it too great an alteration to the original language. Primarily, though, giving INTERCAL programmers the ability to switch numeric base at will amounts to excessive functionality. So much better that a programmer choose a base at the outset and then be forced to stick with it for the remainder of the program.

6.3 Compiler Operation

The same compiler, ick, supports both INTERCAL and TriINTERCAL. This has the advantage that future bug fixes and additions to the language not related to arithmetic immediately apply to both versions. The compiler recognizes INTERCAL source files by the extension '.i', and TriINTERCAL source files by the extension '.3i'. It's as simple as that. There is no way to mix INTERCAL and TriINTERCAL source in the same program, and it is not always possible to determine which dialect a program is written in just by looking at the source code.

6.4 Data Types

The two TriINTERCAL data types are 10-trit unsigned integers and 20-trit unsigned integers. All INTERCAL syntax for distinguishing data types is ported to these new types in the obvious way. Small words may contain numbers from #0 to #59048; large words may contain numbers from #0$#0 to #59048$#59048. Errors are signaled for constants greater than #59048 and for attempts to WRITE IN numbers too large for a given variable or array element to hold.

Note that though TriINTERCAL considers all numbers to be unsigned, nothing prevents the programmer from implementing arithmetic operations that treat their operands as signed. Three's complement is one obvious choice, but balanced ternary notation is also a possibility. This latter is a very pretty and symmetrical system in which all 2 trits are treated as if they had the value -1.

6.5 Operators

The TriINTERCAL operators are designed to inherit the relevant properties of the standard INTERCAL operators, so that both can be considered as merely different aspects of the same Platonic ideal. (Not that the word "ideal" is ever particularly relevant when used in connection with INTERCAL.)

6.5.1 Binary Operators I

The binary operators carry over from the original language with only minor changes. The mingle operator ($) creates a 20-trit word by alternating trits from its two 10-trit operands. The select operator (˜) is a little more complicated, since the ternary tritmask may contain 0, 1, and 2 trits. If we observe that the select operation on binary operands amounts to a bitwise AND and some rearrangement of bits, it seems appropriate to base the select for ternary operands on a tritwise AND in the analogous fashion. We therefore postpone the definition of select until we know what a tritwise AND looks like.

6.5.2 Unary Operators

The unary operators in INTERCAL are all derived from the familiar Boolean operations on single bits. To extend these operations to trits, we first ask ourselves what the important properties of these operations are that we wish to be preserved, then design the tritwise operators so that they behave in a similar fashion.

6.5.2.1 Unary Logical Operators

Let's start with AND and OR. To begin with, these can be considered "choice" or "preference" operators, as they always return one of their operands. AND can be described as wanting to return 0, but returning 1 if it is given no other choice, i.e., if both operands are 1. Similarly, OR wants to return 1 but returns 0 if that is its only choice. From this it is immediately apparent that each operator has an identity element that "always loses", and a dominator element that "always wins".

AND and OR are commutative and associative, and each distributes over the other. They are also symmetric with each other, in the sense that AND looks like OR and OR looks like AND when the roles of 0 and 1 are interchanged (De Morgan's Laws). This symmetry property seems to be a key element to the idea that these are logical, rather than arithmetic, operators. In a three-valued logic we would similarly expect a three-way symmetry among the three values 0, 1 and 2 and the three operators AND, OR and (of course) BUT.

The following tritwise operations have all the desired properties: OR returns the greater of its two operands. That is, it returns 2 if it can get it, else it tries to return 1, and it returns 0 only if both operands are 0. AND wants to return 0, will return 2 if it can't get 0, and returns 1 only if forced. BUT wants 1, will take 0, and tries to avoid 2. The equivalents to De Morgan's Laws apply to rotations of the three elements, e.g., 0 -> 1, 1 -> 2, 2 -> 0. Each operator distributes over exactly one other operator, so the property "X distributes over Y" is not transitive. The question of which way this distributivity ring goes around is left as an exercise for the student.

In TriINTERCAL programs the whirlpool (@) denotes the unary tritwise BUT operation. You can think of the whirlpool as drawing values preferentially towards the central value 1. Alternatively, you can think of it as drawing your soul and your sanity inexorably down ...

On the other hand, maybe it's best you *not* think of it that way.

A few comments about how these operators can be used. OR acts like a tritwise maximum operation. AND can be used with tritmasks. 0's in a mask wipe out the corresponding elements in the other operand, while 1's let the corresponding elements pass through unchanged. 2's in a mask consolidate the values of nonzero elements, as both 1's and 2's in the other operand yield 2's in the output. BUT can be used to create "partial tritmasks". 0's in a mask let BUT eliminate 2's from the other operand while leaving other values unchanged. Of course, the symmetry property guarantees that the operators don't really behave differently from each other in any fundamental way; the apparent differences come from the intuitive view that a 0 trit is "not set" while a 1 or 2 trit is "set".

6.5.2.2 Binary Operators II

At this point we can define select, since we now know what the tritwise AND looks like. Select takes the binary tritwise AND of its two operands. It shifts all the trits of the result corresponding to 2's in the right operand over to the right (low) end of the result, then follows them with all the output trits corresponding to 1's in the right operand. Trits corresponding to 0's in the right operand, which are all 0 anyway, occupy the remaining space at the left end of the output word. Both 10-trit and 20-trit operands are accepted, and are padded with zeroes on the left if necessary. The output type is determined the same way as in standard C-INTERCAL.

6.5.2.3 Unary Arithmetic Operators

Now that we've got all that settled, what about XOR? This is easily the most-useful of the three unary INTERCAL operators, because it combines in one package the operations ADD WITHOUT CARRY, SUBTRACT WITHOUT BORROW, BITWISE NOT-EQUAL, and BITWISE NOT. In TriINTERCAL we can't have all of these in the same operator, since addition and subtraction are no longer the same thing. The solution is to split the XOR concept into two operators. The ADD WITHOUT CARRY operation is represented by the new sharkfin (^), while the old what (?) represents SUBTRACT WITHOUT BORROW. The reason for this choice is so that what (?) will also represent the TRITWISE NOT-EQUAL operation.

Note that what (?), unlike the other four unary operators, is not symmetrical. It should be thought of as rotating its operand one trit to the right (with wraparound) and then subtracting off the trits of the original number. These subtractions are done without borrowing, i.e., trit-by-trit modulo 3.

6.5.3 Examples

The TriINTERCAL operators really aren't all that bad once you get used to them. Let's look at a few examples to show how they can be used in practice. In all of these examples the input value is contained in the 10-trit variable .3.

In INTERCAL, single-bit values often have to be converted from {0,1} to {1,2} for use in RESUME statements. Examples of how to do this appear in the original manual. In TriINTERCAL the expression "^.3$#1"~#1 sends 0 -> 1 and 1 -> 2. If the 1-trit input value can take on any of its three possible states, however, we will also have to deal with the 2 case. The expression "V.3$#1"~#1 sends {0,1} -> 1 and 2 -> 2. To test if a trit is set, we can use "V'"&.3$#2"~#1'$#1"~#1, sending 0 -> 1 and {1,2} -> 2. To reverse the test we use "?'"&.3$#2"~#1'$#1"~#1, sending 0 -> 2 and {1,2} -> 1. Note that we have not been taking full advantage of the new select operator. These last two expressions can be simplified into "V!3~#2'$#1"~#1 and "?!3~#2'$#1"~#1, which perform exactly the same mappings. Finally, if we need a 3-way test, we can use "@'"^.3$#7"~#4'$#2"~#10, which obviously sends 0 -> 1, 1 -> 2, and 2 -> 3.

For an unrelated example, the expression "^.3$.3"~"#0$#29524" converts all of the 1-trits of .3 into 2's and all of the 2-trits into 1's. In balanced ternary, where 2-trits represent -1 values, this is the negation operation.

6.6 Beyond Ternary...

While we're at it, we might as well extend this multiple bases business a little farther. The ick compiler actually recognizes filename suffixes of the form '.Ni', where N is any number from 2 to 7. 2 of course gives standard INTERCAL, while 3 gives TriINTERCAL. We cut off before 8 because octal notation is the smallest base used to facilitate human-to-machine communication, and this seems quite contrary to the basic principles behind INTERCAL. The small data types hold 16 bits, 10 trits, 8 quarts, 6 quints, 6 sexts, or 5 septs, and the large types are always twice this size.

As for operators, ? is always SUBTRACT WITHOUT BORROW, and ^ is always ADD WITHOUT CARRY. V is the OR operation and always returns the max of its inputs. & is the AND operation, which chooses 0 if possible but otherwise returns the max of the inputs. @ is BUT, which prefers 1, then 0, then the max of the remaining possibilities. Rather than add more special symbols forever, a numeric modifier may be placed directly before the @ to indicate the operation that prefers one of the digits not already represented. Thus in files ending in '.5i', the permitted unary operators are ?, ^, &, @, 2@, 3@, and V. Use of such barbarisms as 0@ to represent & are not permitted, nor is the use of @ or ^ in files with either of the extensions '.i' or '.2i'. Why not? You just can't, that's why. Don't ask so many questions.

As a closing example, we note that in balanced quinary notation, where 3 means -2 and 4 means -1, the negation operation can be written as either

```
DO .1 <- "^’"^.3$.3"~"#0$#3906"’$’"^.3$.3"~"#0$#3906"’"~"#0$#3906"
```

or as

```
DO .1 <- "^.3$.3"~"#0$#3906"
DO .1 <- "^.1$.1"~"#0$#3906"
```

These work because multiplication by -1 is the same as multiplication by 4, modulo 5.

Now go beat your head against the wall for a while.

7. SUBROUTINE LIBRARY

> "You don't know what you're talking about!" cried Humpty Dumpty. "How many days are there in a year?"
>
> "Three hundred and sixty-five," said Alice.
>
> "And how many birthdays have you?"
>
> "One."
>
> "And if you take one from three hundred and sixty-five, what remains?"
> "Three hundred and sixty-four, of course."
>
> Humpty Dumpty looked doubtful. "I'd rather see that done on paper," he said.

INTERCAL provides several built-in subroutines to which control can be transferred to perform various operations. These operations include many useful functions which are not easily representable in INTERCAL, such as addition, subtraction, etc.

7.1 Usage

In general, the operands are .1, .2, etc., or :1, :2, etc., and the result(s) are stored in what would have been the next operand(s). For instance, one routine adds .1 to .2 and stores the sum in .3, with .4 being used to indicate overflow. All variables not used for results are left unchanged.

7.2 Available Functions

At the time of this writing, only the most fundamental operations are offered in the library, as a more complete selection would require prohibitive time and core to implement. These functions, along with their corresponding entry points (entered via DO (entry) NEXT) are listed below.

```
(1000) .3 <- .1 plus .2, error exit on overflow
(1009) .3 <- .1 plus .2
       .4 <- #1 if no overflow, else .4 <- #2
(1010) .3 <- .1 minus .2, no action on overflow
(1020) .1 <- .1 plus #1, no action on overflow
(1030) .3 <- .1 times .2, error exit on overflow
(1039) .3 <- .1 times .2
       .4 <- #1 if no overflow, else .4 <- #2
(1040) .3 <- .1 divided by .2
       .3 <- #0 if .2 is #0
(1050) .2 <- :1 divided by .1, error exit on overflow
       .2 <- #0 if .1 is #0

(1500) :3 <- :1 plus :2, error exit on overflow
(1509) :3 <- :1 plus :2
       :4 <- #1 if no overflow, else :4 <- #2
(1510) :3 <- :1 minus :2, no action on overflow
(1520) :1 <- .1 concatenated with .2
(1525) This subroutine is intended solely for internal
       use within the subroutine library and is therefore
       not described here. Its effect is to shift .3
       logically 8 bits to the left.
(1530) :1 <- .1 times .2
(1540) :3 <- :1 times :2, error exit on overflow
(1549) :3 <- :1 times :2
       :4 <- #1 if no overflow, else :4 <- #2
(1550) :3 <- :1 divided by :2
       :3 <- #0 if :2 is #0

(1900) .1 <- uniform random no. from #0 to #65535
(1910) .2 <- normal random no. from #0 to .1, with
             standard deviation .1 divided by #12
```

7.3 Automagical Inclusion Of The Subroutine Library

The C-INTERCAL compiler will automatically include the system library if a DO (1xxx) NEXT statement is used, and if no (1xxx) labels are defined anywhere, where (1xxx) is a label in the 1000-1999 range, inclusive. This was not an INTERCAL-72 feature.

8. PROGRAMMING HINTS

> "If any one of them can explain it," said Alice, "I'll give him sixpence. *I* don't believe there's an atom of meaning in it." The jury all wrote down, on their slates, "*She* doesn't believe there's an atom of meaning in it," but none of them attempted to explain the paper.
>
> "If there's no meaning in it," said the King, "that saves a world of trouble, you know, as we needn't try to find any. And yet, I don't know," he went on, spreading out the verses on one knee, and looking at them with one eye: "I seem to see some meaning in them, after all...."

For the user looking to become more familiar with the INTERCAL language, we present in this section an analysis of a complex program, as well as some suggested projects for the ambitious programmer.

Considering the effort involved in writing an INTERCAL program, it was decided in putting together this manual to use an already existing program for instructive analysis. Since there was only one such program available, we have proceeded to use it. It is known as the "INTERCAL System Library."

8.1 Description

The program listing is in section . It is in the same format as would be produced by the Princeton INTERCAL compiler in FORMAT mode with WIDTH=62 (see section). For a description of the functions performed by the Library, see section .

8.2 Analysis

We shall not attempt to discuss here the algorithms used, but rather we shall point out some of the general techniques applicable to a wide range of problems.

Statements 10, 14, 15, and 26 make up a virtual "computed GO TO". When statement 10 is executed, control passes eventually to statement 16 or 11, depending on whether .5 contains #1 or #2, respectively. The value of .5 is determined in statement 9, which demonstrates another handy technique. To turn an expression, exp, with value #0 or #1, into #1 or #2 (for use in a "GO TO"), use "∀'exp'¢#1"˜#3. To reverse the condition (i.e., convert #0 to #2 and leave #1 alone) use "∀'exp'¢#2"˜#3.

Certain conditions are easily checked. For example, to test for zero, select the value from itself and select the bottom bit (see statement 54). To test for all bits being 1's, select the value from itself and select the top bit. The test for greater than, performed in statements 219 and 220 on 32-bit values, employs binary logical operations, which are performed as follows:

```
'∀.1¢.2'˜'#0¢#65535'
```

for 16-bit values or, for 32-bit values:

```
"'∀":1˜'#65535¢30'"¢":2˜'#65535¢#0'"'˜'#0
¢#65535'"¢"'∀":1˜'#0¢#65535'"¢":2˜'#0
¢#65535'"'˜'#0¢#65535'"
```

(The proofs are left as an exercise to the reader.)

Testing for greater-than with 16-bit values is somewhat simpler and is done with the pair of statements:

```
DO .C <- '∀.A¢.B'˜'#0¢#65535'
DO .C <- '&"'.A˜.C'˜'"∀'∀.C˜.C'¢#32768"
       ˜"#0¢#65535"'"¢".C˜.C"'˜#1
```

This sets .C (a dummy variable) to #1 if .A > .B, and #0 otherwise. The expression may be expanded as described above to instead set .C to #1 or #2.

Note also in statement 220 the occurrence of ˜"#65535¢#65535". Although these operations select the entire value, they are not extraneous, as they ensure that the forthcoming ∀s will be operating on 32-bit values.

In several virtual computed GO TOs the DO FORGET #1 (statement 15 in the earlier example) has been omitted, since the next transfer of control would be a DO RESUME #1. By making this a DO RESUME #2 instead, the FORGET may be forgotten.

In statement 64, note that .2 is STASHed twice by a single statement. This is perfectly legal.

Lastly, note in statements 243 and 214 respectively, expressions for shifting 16- and 32-bit variables logically one place to the left. Statement 231 demonstrates right-shifting for 32-bit variables.

8.3 Program Listing

```
        (1000)  PLEASE IGNORE .4
                PLEASE ABSTAIN FROM (1005)
        (1009)  DO STASH .1 + .2 + .5 + .6
                DO .4 <- #1
                DO (1004) NEXT
        (1004)  PLEASE FORGET #1
                DO .3 <- '∀.1¢.2'~'#0¢#65535'
                DO .6 <- '&.1¢.2'~'#0¢#65535'
                PLEASE DO .5 <- "∀!6~#32768'¢#1"~#3
                DO (1002) NEXT
                DO .4 <- #2
        (1005)  DO (1006) NEXT
*       (1999)  DOUBLE OR SINGLE PRECISION OVERFLOW
        (1002)  DO (1001) NEXT
        (1006)  PLEASE FORGET #1
                DO .5 <- '∀"!6~.6'~#1"¢#1'~#3
                DO (1003) NEXT
                DO .1 <- .3
                DO .2 <- !6¢#0'~'#32767¢#1'
                DO (1004) NEXT
        (1003)  DO (1001) NEXT
                DO REINSTATE (1005)
        (1007)  PLEASE RETRIEVE .1 + .2 + .5 + .6
                DO REMEMBER .4
                PLEASE RESUME #2
        (1001)  DO RESUME .5
        (1010)  DO STASH .1 + .2 + .4
                DO .4 <- .1
                DO .1 <- '∀.2¢#65535'~'#0¢#65535'
                DO (1020) NEXT
                PLEASE DO .2 <- .4
                PLEASE DO (1009) NEXT
                DO RETRIEVE .1 + .2 + .4
                PLEASE RESUME #1
        (1020)  DO STASH .2 + .3
                DO .2 <- #1
                PLEASE DO (1021) NEXT
        (1021)  DO FORGET #1
                DO .3 <- "∀!1~.2'¢#1"~#3
                PLEASE DO .1 <- '∀.1¢.2'~'#0¢#65535'
                DO (1022) NEXT
                DO .2 <- !2¢#0'~'#32767¢#1'
                DO (1021) NEXT
        (1023)  PLEASE RESUME .3
        (1022)  DO (1023) NEXT
                PLEASE RETRIEVE .2 + .3
                PLEASE RESUME #2
        (1030)  DO ABSTAIN FROM (1033)
                PLEASE ABSTAIN FROM (1032)
        (1039)  DO STASH :1 + .5
                DO (1530) NEXT
                DO .3 <- :1~#65535
                PLEASE DO .5 <- :1~'#65280¢#65280'
                DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
```

```
               DO (1031) NEXT
        (1032) DO (1033) NEXT
               DO (1999) NEXT
        (1031) DO (1001) NEXT
        (1033) DO .4 <- .5
               DO REINSTATE (1032)
               PLEASE REINSTATE (1033)
               DO RETRIEVE :1 + .5
               PLEASE RESUME #2
        (1040) PLEASE STASH .1 + .2 + .2 + :1 + :2 + :3
               DO .2 <- #0
               DO (1520) NEXT
               DO STASH :1
               PLEASE RETRIEVE .2
               DO .1 <- .2
               DO .2 <- #0
               PLEASE DO (1520) NEXT
               DO :2 <- :1
               DO RETRIEVE .1 + .2 + :1
               DO (1550) NEXT
               PLEASE DO .3 <- :3
               DO RETRIEVE :1 + :2 + :3
               DO RESUME #1
        (1050) PLEASE STASH :2 + :3 + .5
               DO :2 <- .1
               PLEASE DO (1550) NEXT
               DO .5 <- :3~'#65280¢#65280'
               DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
               DO (1051) NEXT
               DO (1999) NEXT
        (1051) DO (1001) NEXT
               DO .2 <- :3
               PLEASE RETRIEVE :2 + :3 + .5
               DO RESUME #2
        (1500) PLEASE ABSTAIN FROM (1502)
               PLEASE ABSTAIN FROM (1506)
        (1509) PLEASE STASH :1 + .1 + .2 + .3 + .4 + .5 + .6
               DO .1 <- :1~#65535
               PLEASE DO .2 <- :2~#65535
               DO (1009) NEXT
               DO .5 <- .3
               PLEASE DO .6 <- .4
               DO .1 <- :1~'#65280¢#65280'
               DO .2 <- :2~'#65280¢#65280'
               DO (1009) NEXT
               DO .1 <- .3
               PLEASE DO (1503) NEXT
               DO .6 <- .4
               DO .2 <- #1
               DO (1009) NEXT
               DO .1 <- .3
               DO (1501) NEXT
        (1504) PLEASE RESUME .6
        (1503) DO (1504) NEXT
```

```
        (1501)  DO .2 <- .5
                DO .5 <- '∀"'&.6¢.4'~#1"¢#2'~#3
                DO (1505) NEXT
        (1506)  DO (1502) NEXT
                PLEASE DO (1999) NEXT
        (1505)  DO (1001) NEXT
        (1502)  DO :4 <- .5
                DO (1520) NEXT
                DO :3 <- :1
                PLEASE RETRIEVE :1 + .1 + .2 + .3 + .4 + .5 + .6
                DO REINSTATE (1502)
                DO REINSTATE (1506)
                PLEASE RESUME #3
        (1510)  DO STASH :1 + :2 + :4
                DO :1 <- "'∀":2~'#65535¢#0'"¢#65535'
                     ~'#0¢#65535'"¢"'∀":2~'#0¢#65535'
                     "¢#65535'~'#0¢#65535'"
                DO :2 <- #1
                DO (1509) NEXT
                PLEASE RETRIEVE :1
                DO :2 <- :3
                PLEASE DO (1509) NEXT
                DO RETRIEVE :2 + :4
                PLEASE RESUME #1
        (1520)  PLEASE STASH .3 + .4
                DO .3 <- .1~#43690
                DO (1525) NEXT
                PLEASE DO .4 <- 'V.3¢".2~#43690"'~'#0¢#65535'
                DO .3 <- .1~#21845
                PLEASE DO (1525) NEXT
                DO :1 <- .4¢"'V.3¢".2~#21845"'~'#0¢#65535'"
                PLEASE RETRIEVE .3 + .4
                DO RESUME #1
        (1525)  DO .3 <- '"'"'"!3¢#0'~'#32767¢#1'"¢#0'
                     ~'#32767¢#1'"¢#0'~'#16383¢#3'"¢#0'
                     ~'#4095¢#15'
                PLEASE RESUME #1
        (1530)  DO STASH :2 + :3 + .3 + .5
                DO :1 <- #0
                DO :2 <- .2
                DO .3 <- #1
                DO (1535) NEXT
        (1535)  PLEASE FORGET #1
                DO .5 <- "∀!1~.3'¢#1"~#3
                DO (1531) NEXT
                DO (1500) NEXT
                DO :1 <- :3
                PLEASE DO (1533) NEXT
        (1531)  PLEASE DO (1001) NEXT
        (1533)  DO FORGET #1
                DO .3 <- !3¢#0'~'#32767¢#1'
                DO :2 <- ":2~'#0¢#65535'"¢"'":2~'#32767
                     ¢#0'"¢#0'~'#32767¢#1'"
                PLEASE DO .5 <- "∀!3~.3'¢#1"~#3
```

```
        DO (1532) NEXT
        DO (1535) NEXT
(1532)  DO (1001) NEXT
        PLEASE RETRIEVE :2 + :3 + .3 + .5
        DO RESUME #2
(1540)  PLEASE ABSTAIN FROM (1541)
        DO ABSTAIN FROM (1542)
(1549)  PLEASE STASH :1 + :2 + :4 + :5 + .1 + .2 + .5
        DO .1 <- :1~#65535
        PLEASE DO .2 <- :2~'#65280¢#65280'
        DO .5 <- :1~'#65280¢#65280'
        DO (1530) NEXT
        DO :3 <- :1
        DO .2 <- :2~#65535
        PLEASE DO (1530) NEXT
        DO :5 <- :1
        DO .1 <- .5
        DO (1530) NEXT
        DO :4 <- :1
        PLEASE DO :1 <- ":3~'#65280¢#65280'"
             ¢":4~'#65280¢#65280'"
        DO .5 <- ':1~:1'~#1
        DO .2 <- :2~'#65280¢#65280'
        DO (1530) NEXT
        PLEASE DO .5 <- '"':1~:1'~#1"¢.5'~#3
        DO .1 <- :3~#65535
        DO .2 <- #0
        DO (1520) NEXT
        PLEASE DO :2 <- :1
        PLEASE DO .1 <- :4~#65535
        DO (1520) NEXT
        DO (1509) NEXT
        DO .5 <- !5¢":4~#3"'~#15
        DO :1 <- :3
        DO :2 <- :5
        DO (1509) NEXT
        PLEASE DO .5 <- !5¢":4~#3"'~#172
        DO .5 <- '∀"!5~.5'~#1"¢#1'~#3
        PLEASE RETRIEVE :4
(1541)  DO :4 <- .5
        DO (1543) NEXT
(1542)  DO (1544) NEXT
        PLEASE DO (1999) NEXT
(1543)  DO (1001) NEXT
(1544)  DO REINSTATE (1541)
        PLEASE REINSTATE (1542)
        PLEASE RETRIEVE :1 + :2 + :5 + .1 + .2 + .5
        DO RESUME #2
(1550)  DO STASH :1 + :4 + :5 + .5
        DO :3 <- #0
        DO .5 <- '∀"':2~:2'~#1"¢#1'~#3
        PLEASE DO (1551) NEXT
        DO :4 <- #1
        PLEASE DO (1553) NEXT
```

```
        (1553)  DO FORGET #1
                DO .5 <- '∀":2~'#32768¢#0'"¢#2'~#3
                DO (1552) NEXT
                DO :2 <- ":2~'#0¢#65535'"¢"'":2~'#32767
                     ¢#0'"¢#0'~'#32767¢#1'"
                PLEASE DO :4 <- ":4~'#0¢#65535'"¢"'":4
                     ~'#32767¢#0'"¢#0'~'#32767¢#1'"
                DO (1553) NEXT
        (1552)  DO (1001) NEXT
        (1556)  PLEASE FORGET #1
                DO :5 <- "'∀":1~'#65535¢#0'"¢":2
                     ~'#65535¢#0'"'~'#0¢#65535'"¢"'
                     ∀":1~'#0¢#65535'"¢":2~'#0¢
                     #65535'"'~'#0¢#65535'"
                DO .5 <- '∀"'&"':2~:5'~'"'∀"'∀":5~:5
                     "~"#65535¢#65535"'~'#65535¢#0'"¢
                     #32768'~'#0¢#65535'"¢"'∀":5~:5"
                     ~"#65535¢#65535"'~'#0¢#65535'"'"¢"'
                     :5~:5'~#1"'~#1"¢#2'~#3
                DO (1554) NEXT
                DO :5 <- :3
                DO (1510) NEXT
                PLEASE DO :1 <- :3
                DO :3 <- "'V":4~'#65535¢#0'"¢":5~'#65535
                     ¢#0'"'~'#0¢#65535'"¢"'V":4~'#0
                     ¢#65535'"¢":5~'#0¢#65535'"'~'#0
                     ¢#65535'"
                DO (1555) NEXT
        (1554)  PLEASE DO (1001) NEXT
        (1555)  DO FORGET #1
                DO .5 <- "∀':4~#1'¢#2"~#3
                DO (1551) NEXT
                DO :2 <- ":2~'#0¢#65534'"¢":2~'#65535¢#0'"
                DO :4 <- ":4~'#0¢#65534'"¢":4~'#65535¢#0'"
                PLEASE DO (1556) NEXT
        (1551)  DO (1001) NEXT
                PLEASE RETRIEVE :1 + :4 + :5 + .5
                PLEASE RESUME #2
        (1900)  DO STASH .2 + .3 + .5
                DO .1 <- #0
                DO .2 <- #1
                PLEASE DO (1901) NEXT
        (1901)  DO FORGET #1
                DO %50 .1 <- 'V.1¢.2'~'#0¢#65535'
                DO .2 <- !2¢#0'~'#32767¢#1'
                PLEASE DO .5 <- "∀!2~.2'¢#1"~#3
                DO (1902) NEXT
                DO (1901) NEXT
        (1902)  DO (1001) NEXT
                DO RETRIEVE .2 + .3 + .5
                PLEASE RESUME #2
        (1910)  PLEASE STASH .1 + .3 + .5 + :1 + :2 + :3
                DO .3 <- #65524
                DO :1 <- #6
```

```
              DO (1911) NEXT
*             PLEASE NOTE THAT YOU CAN'T GET THERE FROM HERE
       (1912) DO (1001) NEXT
       (1911) DO FORGET #1
              PLEASE DO (1900) NEXT
              DO :2 <- .1
              DO (1500) NEXT
              PLEASE DO :1 <- :3
              DO .1 <- .3
              DO (1020) NEXT
              PLEASE DO .3 <- .1
              DO .5 <- '∀"!3~.3'~#1"¢#2'~#3
              DO (1912) NEXT
              DO .1 <- #12
              PLEASE DO (1050) NEXT
              DO RETRIEVE .1
              DO (1530) NEXT
              DO :2 <- #32768
              DO (1500) NEXT
              PLEASE DO .2 <- :3~'#65280¢#65280'
              PLEASE RETRIEVE .3 + .5 + :1 + :2 + :3
              DO RESUME #1
```

8.4 Programming Suggestions

For the novice INTERCAL programmer, we provide here a list of suggested INTERCAL programming projects:

Write an integer exponentiation subroutine. :1 <- .1 raised to the .2 power.

Write a double-precision sorting subroutine. Given 32-bit array ;1 of size :1, sort the contents into numerically increasing order, leaving the results in ;1.

Generate a table of prime numbers.

Put together a floating-point library, using 32-bit variables to represent floating-point numbers (let the upper half be the mantissa and the lower half be the characteristic). The library should be capable of performing floating-point addition, subtraction, multiplication, and division, as well as the natural logarithm function.

Program a Fast Fourier Transform (FFT). This project would probably entail the writing of the floating-point library as well as sine and cosine functions.

Calculate, to :1 places, the value of pi.

(Working programs for each of these projects are included in the C-INTERCAL distribution's pit directory of sample code.)

9. ERROR MESSAGES

> Alice felt dreadfully puzzled. The Hatter's remark seemed to her to have no sort of meaning in it, and yet it was certainly English. "I don't quite understand you," she said, as politely as she could.

Due to INTERCAL's implementation of comment lines (see section 4.5), most error messages are produced during execution instead of during compilation. All errors except those not causing immediate termination of program execution are treated as fatal.

9.1 Format

All error messages appear in the following form:

```
ICLnnnI (error message)
    ON THE WAY TO STATEMENT nnnn
    CORRECT SOURCE AND RESUBMIT
```

The message varies depending upon the error involved. For undecodable statements the message is the statement itself. The second line tells which statement would have been executed next had the error not occurred. Note that if the error is due to 80 attempted levels of NEXTing, the statement which would have been executed next need not be anywhere near the statement causing the error.

9.2 Messages

Brief descriptions of the different error types are listed below according to message number.

000 An undecodable statement has been encountered in the course of execution. Note that keypunching errors can be slightly disastrous, since if 'FORGET' were misspelled F-O-R-G-E-R, the results would probably not be those desired. Extreme misspellings may have even more surprising consequences. For example, misspelling 'FORGET' R-E-S-U-M-E could have drastic results.

017 An expression contains a syntax error.

079 Improper use has been made of statement identifiers.

099 Improper use has been made of statement identifiers.

123 Program has attempted 80 levels of NEXTing.

129 Program has attempted to transfer to a non-existent line label.

139 An ABSTAIN or REINSTATE statement references a non-existent line label.

182 A line label has been multiply defined.

197 An invalid line label has been encountered.

200 An expression involves an unidentified variable.

240 An attempt has been made to give an array a dimension of zero.

241 Invalid dimensioning information was supplied in defining or using an array.

275 A 32-bit value has been assigned to a 16-bit variable.

436 A retrieval has been attempted for an unSTASHed value.

533 A WRITE IN statement or interleave (¢) operation has produced a value requiring over 32 bits to represent.

562 Insufficient data.

579 Input data is invalid.

621 The expression of a RESUME statement evaluated to #0.

632 Program execution was terminated via a RESUME statement instead of GIVE UP.

633 Execution has passed beyond the last statement of the program.

774 A compiler error has occurred (see section).

778 An unexplainable compiler error has occurred.

The following error codes are new in C-INTERCAL:

111 You tried to use a C-INTERCAL extension with the ‘traditional’ flag on.

127 Can’t find syslib.i file when it’s needed for magical inclusion.

222 Out of stash space, or attempt to stash an undeclared array.

333 Too many variables.

444 A COME FROM statement references a non-existent line label.

555 More than one COME FROM references the same label.

666 Too many source lines.

777 No such source file.

888 Can’t open C output file.

999 Can’t open C skeleton file.

998 Source file name with invalid extension (use .i or .[2-7]i).

997 Illegal possession of a controlled unary operator.

10. The C-INTERCAL Compiler

> "And yet it was a very clever pudding to invent."
>
> "What did you mean it to be made of?" Alice asked, hoping to cheer him up, for the poor Knight seemed quite low-spirited about it.
>
> "It began with blotting-paper," the Knight answered with a groan.
>
> "That wouldn't be very nice, I'm afraid --"
>
> "Not very nice *alone*," he interrupted, quite eagerly, "but you've no idea what a difference it makes, mixing it with other things -- such as gunpowder and sealing-wax."

This section describes the use of ick, the C-INTERCAL compiler. Information specific to the other known INTERCAL compilers appears in the following sections. These sections are less useful, since the other compilers are no longer available, but they are retained to provide a sense of history, and because any information regarding INTERCAL is really too scarce to discard.

10.1 Character Set

The C-INTERCAL compiler uses ASCII rather than EBCDIC. We follow the Atari implementation (see section) by (a) replacing the change sign (¢) with big money ($) as the mingle operator, and (b) replacing the bookworm (∀) symbol with what (?) as the XOR operator.

Version 0.18 and later versions accept the Latin-1 (ISO 8859-1) character set, which is a superset of ASCII. As a result, 0xA2 (¢), 0xA3 (quid), and 0xA4 (zlotnik) are accepted as mingle operators.

Version 0.20 and later versions also accept UTF-8 as an alternative to Latin-1. The above three currency operators will be accepted in either Latin-1 or UTF-8 format. UTF-8 format is required if you wish to use any of EURO-CURRENCY SIGN, COLON SIGN (that's Costa Rican dinero, not the Intercal two-spot), CRUZEIRO SIGN, FRENCH FRANC SIGN, LIRA SIGN, MILL SIGN, NAIRA SIGN, PESETA SIGN, RUPEE SIGN, WON SIGN, NEW SHEQEL SIGN, DONG SIGN, EURO SIGN, BENGALI RUPEE MARK, BENGALI RUPEE SIGN, or THAI CURRENCY SYMBOL BAHT. (Isn't internationalization wonderful?) Furthermore, you can use the Intercal character FOR ALL to represent the what operator.

10.2 Usage and Compilation Options

To compile an INTERCAL program 'foo.i' to executable code, just do

```
ick foo.i
```

There's a -c option that leaves the generated C code in place for inspection (suppressing compilation to machine code), a -d option that enables verbose parse reporting from the yacc/bison parser, a -t option that requires strict INTERCAL-72 compliance (rejecting COME FROM and the extensions for bases other than two), a -b option that disables the INTERCAL-72 random-bug feature (error 774), and an -O option that enables the (hah!) optimizer. Invoking ick -? prints a usage message.

Another compilation switch affects C-INTERCAL's runtime behavior. The '-C' option forces output in "clockface" mode, for superstitious users who believe writing "IV" upside-down offends IVPITER and would rather see IIII.

10.3 Runtime Options

Every C-INTERCAL runtime also accepts certain options at runtime. These include [+/-]help, [+/-]traditional, and [+/-]wimpmode. The help option (with either + or -) triggers a 'usage' message. The +traditional option is presently a no-op.

Steve explains: "The wimpmode option is the most interesting. I found myself always running my test programs with filters on both ends to work around the 'nifty' INTERCAL number representations. This was so painful that I decided it would be *less* painful (and a lot less code) if I added a 'wimp' option. With the

+wimpmode option, the user is subjected to a humiliating message about what a wimp he or she is to use this mode, but after that is allowed to use conventional numerical notation. While such a mode doubtless violates to some extent the INTERCAL philosophy, the fact that an 'unbutcher' command has been posted clearly indicates the need for it. Anyway ... if you don't like it, don't use it ... the default is -wimpmode (i.e. *not* wimp mode)."

10.4 PLEASE Politesse Checking

A feature of INTERCAL-72 not documented in the original manual was that it required a certain level of politesse from the programmer. If fewer than 1/5th of the program statements included the PLEASE qualifier, the program would be rejected as insufficiently polite. If more than 1/3rd of them included PLEASE, the program would be rejected as excessively polite.

This check has been implemented in C-INTERCAL. To assist programmers in coping with it, the intercal.el mode included with the distribution randomly expands "do " in entered source to "PLEASE DO" 1/4th of the time.

10.5 Localization: Ancient Roman

Version 0.18 and later versions of C-Intercal have been localized for Ancient Roman as well as English-speaking locales. This means that Latin-language keywords are accepted as equivalents for the English-based ones traditionally used. See src/lexer.l for details, as we do not propose to break Priscian's head (publicly expose our ignorance of Latin) here.

11. The Atari Implementation

> "Perhaps it doesn't understand English," thought Alice. "I daresay it's a French mouse, come over with William the Conqueror." (For, with all her knowledge of history, Alice had no very clear notion how long ago anything had happened.)

The Atari implementation of INTERCAL differs from the original Princeton version primarily in the use of ASCII rather than EBCDIC. Since there is no "change" sign (¢) in ASCII, we have substituted the "big money" ($) as the mingle operator. We feel that this correctly represents the increasing cost of software in relation to hardware. (Consider that in 1970 one could get RUNOFF for free, to run on a $20K machine, whereas today a not quite as powerful formatter costs $99 and runs on a $75 machine.) We also feel that there should be no defensible contention that INTERCAL has any sense. Also, since overpunches are difficult to read on the average VDT, the exclusive-or operator may be written ?. This correctly expresses the average person's reaction on first encountering exclusive-or, especially on a PDP-11. Note that in both of these cases, the over-punched symbol may also be used if one is masochistic, or concerned with portability to the Princeton compiler. The correct over-punch for "change" is "c<backspace>/" and the correct over-punch for ∀ is "V<backspace>-". These codes will be properly printed if you have a proper printer, and the corresponding EBCDIC code will be produced by the /IBM option on the LIST command.

12. The Princeton Compiler

> "My name is Alice, so please your Majesty," said Alice very politely; but she added, to herself, "Why, they're only a pack of cards, after all. I needn't be afraid of them!"

The Princeton compiler, written in SPITBOL (a variant of SNOBOL), performs the compilation in two stages. First the INTERCAL source is converted into SPITBOL source, then the latter is compiled and executed.

It should be noted that the Princeton compiler fails to properly interpret certain multiply-subscripted expressions, such as:

```
",1SUB",2SUB#1"#2"
```

This is not a "bug". Being documented, it is merely a "restriction". Such cases may be resolved by alternating sparks and ears in various levels of expression nesting:

```
",1SUB',2SUB#1'#2"
```

which is advisable in any case, since INTERCAL expressions are unreadable enough as is.

Since there is currently no catalogued procedure for invoking the compiler, the user must include the in-line procedure shown in Figure 3 in his job before the compilation step. Copies of this in-line procedure may be obtained at any keypunch if the proper keys are struck.

The compiler is then executed in the usual manner:

```
// EXEC INTERCAL[,PARM='parameters']
//COMPILE.SYSIN DD *
{INTERCAL source deck}
/*
//EXECUTE.SYSWRITE DD *
{input data}
/*
```

The various parameters are described following the in-line procedure. At most one parameter from each set may apply to a given compilation; if more than one are specified, the results are undefined, and may vary depending upon the particular set of options. The default parameters are italicized.

```
//INTERCAL PROC
//COMPILE EXEC PGM=INTERCAL
//STEPLIB DD DSN=U.INTERCAL.LIBRARY,DISP=SHR
//         DD DSN=SYS1.FORTLIB,DISP=SHR
//SYSPRINT DD SYSOUT=A,DCB=(BLKSIZE=992,LRECL=137,RECFM=VBA)
//SYSPUNCH DD DUMMY
//SCRATCH DD DSN=&COMPSET,UNIT=SYSDA,SPACE=(CYL,(3,1)),DISP=(,PASS)
//EXECUTE EXEC PGM=EXECUTE,COND=(4,LT)
//SOURCES DD DSN=U.INTERCAL.SOURCES,DISP=SHR  4
//STEPLIB DD DSN=U.INTERCAL.LIBRARY,DISP=SHR
//         DD DSN=SYS5.SPITLIB,DISP=SHR
//         DD DSN=SYS1.FORTLIB,DISP=SHR
//SYSIN DD DSN=&COMPSET,DISP=(OLD,DELETE)
//SYSOBJ DD SYSOUT=B,DCB=(BLKSIZE=80,LRECL=80,RECFM=F)
//SYSPRINT DD SYSOUT=A,DCB=(BLKSIZE=992,LRECL=137,RECFM=VBA)
//SYSPUNCH DD DUMMY
// PEND
```

Figure 3. Inline procedure for using INTERCAL.

OPT
NOOPT
In the default mode, the compiler will print a list of all options in effect, including the defaults for unspecified parameter groups and the effective option for those sets where one was specified. If NOOPT is requested, it causes the default mode to be assumed.

OPTSUB
NOOPTSUB
NOSUB
Unless 'NOOPTSUB' is requested, the System Library is optimized, resulting in much more rapid processing of function calls. Specifying NOOPTSUB causes the non-optimized INTERCAL code shown in section
to be used, whereas NOSUB requests that the System Library be omitted altogether.

IAMBIC
PROSE
The IAMBIC parameter permits the programmer to use poetic license and thus write in verse. If the reader does not believe it possible to write verse in INTERCAL, he should send the authors a stamped, self-addressed envelope, along with any INTERCAL program, and they will provide one which is verse.

FORMAT
NOFORMAT
In FORMAT mode, each statement printed is put on a separate line (or lines). In NOFORMAT mode, the free-format source is printed exactly as input. In this latter case, statement numbers are provided only for the first statement on a card, and they may be only approximate. Also, unrecognizable statements are not flagged.

SEQ
NOSEQ
If the source deck has sequence numbers in columns 73 through 80, specifying 'SEQ' will cause them to be ignored.

SOURCE
NOSOURCE
If NOSOURCE is selected, all source listing is suppressed.

LIST
NOLIST
If LIST is specified, the compiler will provide a list of statement numbers catalogued according to type of statement. The compiler uses this table to perform abstentions by gerund.

WIDTH=nn
This sets the width (in number of characters) of the output line for FORMAT mode output. The default is **132.**

---

4. Pending acquisition of SPITBOL release 3.0, the SOURCES DD card must be replaced by the five cards:

```
//NOOPTPFX DD DSN=U.INTERCAL.SOURCES(NOOPTPFX),DISP=SHR
//NOOPTSUB DD DSN=U.INTERCAL.SOURCES(NOOPTSUB),DISP=SHR
//OPTPFX DD DSN=U.INTERCAL.SOURCES(OPTPFX),DISP=SHR
//OPTSUB DD DSN=U.INTERCAL.SOURCES(OPTSUB),DISP=SHR
//PRELIM DD DSN=U.INTERCAL.SOURCES(PRELIM),DISP=SHR
```

CODE
NOCODE

Include 'CODE' in the parameter list to obtain a listing of the SPITBOL code produced for each INTERCAL statement.

LINES=nn

This determines the number of lines per page, during both compilation and execution. The default is **60.**

DECK
NODECK

Selecting 'DECK' will cause the compiler to punch out a SPITBOL object deck which may then be run without reinvoking the INTERCAL (or SPITBOL) compiler.

KIDDING
NOKIDDING

Select NOKIDDING to eliminate the snide remarks which ordinarily accompany INTERCAL error messages.

GO
NOGO

Specifying 'NOGO' will cause the program to be compiled but not executed. EXECUTE/NOEXECUTE may be substituted for GO/NOGO, but this will result in an error, and GO will be assumed.

BUG
NOBUG

Under the default, there is a fixed probability of a fatal compiler bug being worked at random into the program being compiled. Encountering this bug during execution results in error message 774 (see section). This probability is reduced to zero under 'NOBUG'. This does not affect the probability (presumably negligible) of error message 778.

12.1 Other INTERCAL Compilers

There are no other INTERCAL compilers. [5]

---

5. This assertion in the INTERCAL-72 manual was blatantly contradicted by some notes on an Atari implementation included at the end of the manual. So, you expect compiler manuals to be consistent?

TONSIL A[6]

"Of course they answer to their names?" the Gnat remarked carelessly.

"I never knew them to do it."

"What's the use of their having names," the Gnat said, "if they wo'n't answer to them?"

"No use to *them*," said Alice; "but it's useful to the people that name them, I suppose. If not, why do things have names at all?"

"I ca'n't say," the Gnat replied. "Further on, in the wood down there, they've got no names -- however, go on with your list...."

The Official INTERCAL Character Set

Tabulated next are all the characters used in INTERCAL, excepting letters and digits, along with their names and interpretations. Also included are several characters not used in INTERCAL, which are presented for completeness and to allow for future expansion.

---

6. Since all other reference manuals have Appendices, it was decided that the INTERCAL manual should contain some other type of removable organ.

7. This footnote intentionally unreferenced.

[illegible]		Use (if any)
.	spot	identify 16-bit variable
:	two-spot	identify 32-bit variable
,	tail	identify 16-bit array
;	hybrid	identify 32-bit array
#	mesh	identify constant
=	half-mesh	
´	spark	grouper
‘	backspark	
!	wow	equivalent to spark-spot
?	what	unary logical XOR
"	rabbit-ears	grouper
".	rabbit	equivalent to ears-spot
|	spike	
%	double-oh-seven	percentage qualifier
−	worm	used with angles
<	angle	used with worms
>	right angle	
(	wax	precedes line label
)	wane	follows line label
[	U turn	
]	U turn back	
{	embrace	
}	bracelet	
*	splat	flags invalid statements
&	ampersand [10]	unary logical AND
V	V (or book)	unary logical OR
∀	bookworm (universal qualifier)	unary logical XOR (INTERCAL-72)
$	big money	binary mingle
¢	change	binary mingle (INTERCAL-72)
~	sqiggle	binary select
=	flat worm	
	overline	indicates "times 1000"
+	intersection	separates list items
/	slat	
\	backslat	
@	whirlpool	unary logical BUT (TriINTERCAL)
¬	hookworm	
^	shark (or simply sharkfin)	unary logical XOR (TriINTERCAL)
[illegible]	blotch	
°	bunghole	
®	name grabber	
×	andrew cross	
÷	norwegian minus	
©	text grabber	
§	swash	
μ	microscope	
±	interworm	
¬	half turn	
£	quid	binary mingle
¤	zlotnik	binary mingle

Table 2 (top view). INTERCAL character set.