# A Tutorial on ALGOL 68

by Andrew S. Tanenbaum

Vakgroep Informatica, Wiskundig Seminariunm, Vrije Universiteit, de Boelelaan 1081, Amsterdam, The Netherlands

This paper is an introduction to the main features of ALGOL 68, emphasizing the novel features not found in many other programming languages. The topics, data types (modes), type conversion (coercion), generalized expressions (units), procedures, operators, the standard prelude, and input/output, form the basis of the paper. The approach is informal, relying heavily on many short examples. The paper applies to the Revised Report, published in 1975, rather than to the original report, published in 1969.

Keywords and Phrases: ALGOL 68, ALGOrithmic Language, expression languages, programming languages, high-level languages, problem-oriented languages.

CR Categories: 4.20, 4.22.

This TEX edition was provided by W. B. Kloke, `mailto:klokew@acm.org`. It contains corrections published in *"Computing Surveys, Vol. , No. 3, September 1977, p.255f. .*

## INTRODUCTION

This paper is an introduction to ALGOL 68 – in plain English – for the the nonspecialist. In its short lifetime, ALGOL 68 has acquired something of an international reputation for being obscure. An early description of the language [8] was entitled "ALGOL 68 with Fewer Tears". The feeling has persisted. One recent author [11] has written, "The ALGOL 68 Report is one of the most unreadable documents which has ever been printed." It is our intention demonstrate that ALGOL 68 is neither inscrutable nor difficult, but rather is on extremely powerful programming language which is easily learned and which is applicable to a wide variety of problems.

One reason ALGOL 68 has been slow to be accepted is not hard to discover. The defining report used a completely new kind of grammar to define the language, instead of the now familiar and comfortable Backus-Naur grammar (BNF). This new grammar often called a vW-grammar (in honor of its inventor, A. van Wijngaarden), is context sensitive rather than context free. Like many new ideas, it takes some getting used to, just as BNF grammars did. The new grammar was introduced for some very good reasons. In particular, it allows not only the syntax, but also that part of the semantics having to do with declarations to be defined by the grammar. For example, the nonterminal <program> simply does not generate any program in which variables are undefined, multiply defined, or defined inconsistently with their usage. No English prose is needed to say that variables must not be defined twice, etc. Consequently, W-grammars provide a more complete and accurate definition than do BNF grammars.

During several years of experience with the language, several trouble spots came to light, particularly features of the language that were tricky to implement efficiently. A Revised Report [13] was published in 1975, describing a slightly modified language that does not have these problems. Furthermore, the original report itself was completely rewritten, in order to make it easier to understand. It is the revised language and the Revised Report that are described in this article. References to sections in the Revised Report are indicated by the letters RR preceding the section number.

Rather than attempting to explore every nook and cranny of ALGOL 68, we concentrate on the major features, illustrating them with many examples. Readers wishing a book length introduction to ALGOL 68 are referred to the books listed in Section 11, Where To From Here?

The ALGOL 68 Report introduced a veritable cornucopia of new terminology to the computing community, all of which are precisely defined in the Revised Report (RR 2.1). This was done to force the reader to rely on the Report's definitions, rather than to rely on his previous experience with similar concepts that nonetheless may differ from the Report's definitions in subtle, but crucial, ways.

Nevertheless, to avoid inundating the reader, we try to shun when possible the bus tokens (RR 1.3.3e), invisible production trees (RR 1.1.3.2h), primal environs (RR 2.2.2a), incestuous unions (RR 4.7), notions (RR 1.1.3.1c), protonotions (RR 1.1.3.1b), metanotions (RR 1.1.3.1d), hypernotions (RR 1.1.3.1e), paranotions (RR 1.1.4.2), and their ilk, for more familiar nomenclature. As a starter, we refrain from using "assignation" when "assignment" does just as nicely, and we use "integer" rather than "integral" as an adjective.

Before plunging into the description of the language itself, it is perhaps worthwhile to say something about the principles of its design. One of the key ideas is that of orthogonality. An orthogonal language has a small number of basic constructions, and rules for combining them in regular and systematic ways. A very deliberate attempt is made to eliminate arbitrary restrictions.

The concept of orthogonal design may be made clearer by an example of nonorthogonal design. Many programming languages (for example, FORTRAN, ALGOL 60, and PL/1) have a concept of data types that includes arrays. They also have a concept of functions as rules for mapping parameters onto results. Logically one might expect to be able to combine the "orthogonal" (that is, independent) concepts of data types and functions to construct functions that take an array as parameter and yield an array as the result. An arbitrary restriction that allows arrays to be used as parameters but prohibits them to be used as results is an example of nonorthogonal design. A fundamental principle of ALGOL 68 is that arbitrary rules like this restriction are only used to resolve situations which might otherwise be syntactically or semantically ambiguous.

Another principle, related to that of orthogonality, is the principle of extensibility. ALGOL 68 provides a small number of primitive data types, or modes, as well as mechanisms for the user to extend these in a systematic way. For example, the programmer may create his own data types and his own operators to manipulate them. This philosophy may be contrasted with, say APL, which provides a very large number of standard operators, rather than a very small number and the machinery for programmers to define new ones. Together, orthogonality and extensibility tend to produce a "compact" yet powerful language.

## 1. Bird's-Eye View of ALGOL 68

In the following subsections we briefly mention some basic features of ALGOL 68 that are common to many programming languages in order to be able to use them in subsequent examples.

### 1.1 Program Structure

An ALGOL 68 program consists of a sequence of symbols enclosed by **begin** and **end**, or by parentheses. Some symbols are written in boldface type to distinguish them as keywords. Other symbols are variable identifiers (written in the same general font that is used in programs) and special characters such as ≠, =, (), +, −, etc.

Spaces and carriage returns (change to a new card) may be used freely to improve readability. Spaces are explicitly allowed "inside" identifiers. Thus *the three little pigs* and *thethreelittlepigs* are the same identifier. Identifiers (for example, variable names) may be arbitrarily long.

Comments are enclosed between comment symbols, of which four representations are allowed: ¢, #, **co**, and **comment**. The comment must begin and end with the same comment symbol. A comment may be inserted between any two symbols.

ALGOL 68 is a block structured language, like PL/1 and ALGOL 60. Blocks and procedures may be nested arbitrarily deep. Declarations may appear in any block. Semicolons are used to separate statements, similar to ALGOL 60 (and in contrast to PL/1, which uses them to terminate statements).

## 1.2 Data and Declarations

One of the most basic features of a programming language is the kind of data that it can manipulate. ALGOL 68 provides a rich collection of data types (described in Section 2, Modes). The ALGOL 68 Report uses the term mode instead of type, and we do too. Four of the simplest modes are integer, real (floating point), Boolean, and character. As one might expect, there are integer, real, Boolean, and character variables. All variables must be declared. Any variable used but not declared will be flagged by the compiler as an error. The declaration of a variable consists of a mode, followed by one or more identifiers. The following program illustrates variable declarations. An ALGOL 68 program must contain at least one statement; **skip** is a dummy statement that can be used to turn a collection of declarations into a syntactically valid program.

```
begin
    real e, x, y, z; ¢ 4 real variables ¢
    bool maybe; ¢ 1 boolean variable ¢
    char first initial, middle initial, grade desired;
          ¢ 3 character variables ¢
    int i, j, girlfriends; ¢ 3 integer variables ¢
    skip ¢ dummy statement ¢
end
```

Note that the declarations are separated by semicolons. The symbols **int**, **real**, **bool**, and **char** are not abbreviations; **integer**, **boolean**, and **character** are not allowed (although one can explicitly define them as modes if so desired).

## 1.3 Statements

Algol 68 is an expression language. This means that every construction in the language yields a value and in principle can appear on the right-hand side of an assignment. Nevertheless, certain constructions can also be used as statements. Among these constructions are assignment statements, **if** statements, procedure calls, **for** statements, **while** statements, **case** statements, and **goto** statements. Since these are all quite familiar from other programming languages, a few examples, shown in the next column, should suffice.

A few explanatory notes may be in order. Observe that **if** statements are closed by **fi** (**if** backwards). This solves the dangling else problem. Suppose **fi** were not used. Then the statement **if** *i*<0 **then** **if** *j*<0

**then** *print* ("hello") **else** *print* ("goodbye")

would be ambiguous, possibly meaning

```
if i<0
then if j<0 then print ("hello") fi
else print ("goodbye")
fi
```

or perhaps meaning

```
if i<0
then if j<0 then print ("hello")
    else print ("goodbye")
    fi
fi
```

With the **fi** there is no ambiguity. Furthermore, since both **then** and **else** parts must be explicitly closed, either may contain an arbitrary number of statements without the need for **begin end** as delimiters.

To simplify nested **if** statements, **else if** may be contracted to **elif**,

```
begin ¢ mentally insert the above declarations here ¢
   ¢ assignment statements ¢
   girlfriends := girlfriends−1;
   middle initial := "x";
   e := 2.78;
   ¢ if statements ¢
   if maybe
   then grade desired := "d"
   else grade desired := "f"
   fi; ¢ fi delimits if -- see note below ¢
   if x<0 then x := -x fi;
   if i = j+2
   then x := pi;
      y := 2×e;
      z := 3×e
   fi;
   ¢ procedure calls ¢
   ¢ sum ∧ initialize must be defined elsewhere ¢
   initialize; ¢ no parameters ¢
   sum(x, y, z);
   print (grade desired);
   ¢ for -- while statements ¢
   ¢ the following 5 statements are all equivalent ¢
   for k from 1 by 1 to j+3 while true
   do print (new line) od;
   for k from 1 by 1 to j+3
   do print (new line) od;
   for k from 1 to j+3
   do print (new line) od;
   for k to j+3
   do print (new line) od;
   to j+3
   do print (new line) od;
   while i<j ∨ i<0
   do i := i+1;
      j := j+2;
      print ((i, j))
   od;
   ¢ case statements ¢
   case i+4 in
      j := 0, j := 3, i := i−5, print(i)
   esac;
   case i in
      j := j+3,
      if j = 0 then j := i fi,
      print (i)
   out j := 4
   esac;
   ¢ goto statement ¢
   bed: goto bed
   end
```

providing the **fi** matching the contracted **if** is deleted. For example,

```
if $word$ = "oui"
then $print$ ("french")
else if $word$ = "yes"
    then $print$ ("english")
    else if $word$ = "ja"
        then $print$ ("dutch")
        else $print$ ("minor␣language")
        fi
    fi
fi
```

can be written as

```
if $word$ = "oui"
then $print$ ("french")
elif $word$ = "yes"
then $print$ ("english")
elif $word$ = "ja"
then $print$ ("dutch")
else $print$ ("minor␣language")
fi
```

Even with **elif**, **begin** and **end** are never needed as delimiters.

The **for** and **while** statements shown on page 158 are all special cases of a general **for** statement including both counting parts (**from**. . . **by**. . . **to**) and **while** parts. The **from**, **by**, and **to** parts are each optional, with default values of 1,1, and infinity, respectively. Each part may occur only once. The "controled variable" following **for** is automatically an integer; it can neither be declared nor assigned. If the same identifier occurs outside the statement, it is a different variable. This makes the controled variable inaccessible outside the loop (to give the compiler writer more freedom, and to make correctness proofs easier). Furthermore, the **from**, **by**, and **to** parts are evaluated once and for all before the loop begins. Subsequent changes to any of their variables have no effect on the step size or loop termination condition.

The **case** statement has an integer expression which selects the first, second, third, etc., clause if the expression is 1,2,3, etc., respectively. A clause is a statement (or a group of statements separated by semicolons and enclosed by **begin end** or parentheses). The clauses are separated by commas.

If an **out** clause is present, it will be selected when the expression exceeds the number of clauses or is less than 1. If the **out** clause is omitted and the expression is out of range, the **case** statement is skipped. **esac** is **case** backwards.

Input/output in ALGOL 68 is performed by calling certain input/output procedures, rather than by executing special statements. Procedures are provided for unformatted, formatted, and binary input/output. File and input/output devices can be handled in a consistent and machine-independent way. We examine these input/output procedures in a later section; for now, *read*($x$) is used for input and *print*($x$) is used for output. Each of these procedures may be passed a parenthesized list of variables as parameter, for example, *read*(($x$,$y$,$z$)) and *print*(($i$,$j$,$x$+$z$)).

(The reason for the extra parentheses is explained later on.) The calls print(new line) and print(new page) cause subsequent output to begin at the beginning of the next line or next page, respectively.

A sample ALGOL 68 program is shown on page 160.

## 2. MODES

One of the most powerful features of ALGOL 68 is its rich collection of data types (modes), and the facilities it provides programmers to define their own modes. Programmer-defined modes are constructed from primitive modes, using a few simple rules for creating new modes from old modes. In the following subsections we examine primitive modes, methods for constructing new modes, and finally the mode definition facility in its full glory.

An object is an entity stored in memory during the execution of a program. Integers and reals are typical objects. Each object has a unique mode, for example, **int**, **real**, **bool**, or **char**. Each object also has a value. It is objects that are assigned to variables. For example, an integer with value 3 (some bit pattern in memory) can be assigned to an integer variable. A variable should be thought of as a container (memory location) into which a certain class of objects can be put. Be aware that the container and this container are distinct kinds of entities.

```
begin ¢ This program reads two numbers: the price of an item, and the
      amount the customer gave to the cashier. It then calculates how
      much change he should get, and prints out the correct number of
      quarters, dimes, nickels, and pennies, minimizing the number of
      coins returned. The program only handles change up to 99 cents.
      ¢
    int price, amount paid, change, quarters, dimes, nickels, pennies;
    read ((price, amount paid)); ¢ read input data ¢
    change := amount paid - price;
    if change > 99 ∨ change < 0
    then print ("input␣data␣incorrect")
    else
        if change = 0 ¢ was the payment exact? ¢
        then print ("no␣change")
        else ¢ compute how many of each coin ¢
            quarters := 0; dimes := 0; nickels := 0;
            while change ≥ 25
            do quarters := quarters + 1 ;
                change := change - 25
            od;
            while change ≥ 10
            do dimes := dimes + 1;
                change := change - 10
            od;
            while change ≥ 5
            do nickels := nickels + 1;
                change := change - 5
            od;
            pennies := change
                ¢ print results ¢
                print ((new page, "the␣change␣is",
                        new line, quarters, "quarters",
                        new line, dimes, "dimes",
                        new line, nickels, "nickels",
                        new line, pennies, "pennies",
                        new line))
        fi ¢ this matches if change = 0...¢
    fi ¢ this matches if change > 99...¢
end
```

## 2.1 Primitive Modes

We have already seen how to declare **int**, **real**, **bool**, and **char** variables. These are not the only possibilities, however. A list of the predefined modes with a brief description of each follows:

| | |
|---|---|
| **int** | integer; |
| **real** | real number; |
| **char** | character; |
| **bool** | boolean; |
| **string** | string of characters; |
| **compl** | complex number (2 reals); |
| **bits** | machine word full of bits; |
| **bytes** | machine word full of characters; |
| **sema** | Dijkstra semaphore [4]; |
| **format** | mode used with formatted I/O; |
| **file** | mode used for input/output. |

For some applications, the number of bits in an integer or **real** is insufficient. To accommodate these situations, ALGOL 68 allows primitive modes of **long int**, **long long int**, etc., and **long real**, **long long real**, **long long long real**, etc. Furthermore, to accommodate applications where very many integers or reals are needed, but where fewer than the standard number of bits will suffice, there are modes of **short int**, **short short int** and **short real**, **short short real**, etc.

The number of different lengths and the number of bits in each is up to each ALGOL 68 compiler writer. However, the number of available lengths and the size of each is available to programs at run time to facilitate transfer of programs from one machine to another. For a computer with an 8-bit byte and a 32-bit word, a typical implementation might have: **short short int** (8 bits), **short int** (16 bits), **int** (32 bits), **long int** (64 bits), **long long int** (96 bits), and **long long long int** (128 bits).

The mode **string** defines a string of zero or more characters. Strings may be arbitrarily long, and strings of any length may be assigned to any string variable. In PL/1 terms, all strings are of maximum length equal to infinity and `VARYING`. The following is a valid ALGOL 68 program:

```
begin
    string s;
    s := "␣"; ¢ an empty string ¢
    s := "little"; ¢ a 6 character string ¢
    s := "hello␣there,␣mommies␣and␣daddies"
end
```

The modes **bits** and **bytes** are intended to give the programmer the ability to pack information into machine words to save space. The number of bits in an object of mode **bits** is not determined by the programmer, but by the ALGOL 68 compiler writer. It is to be expected that in most implementations an object of mode **bits** will occupy a full machine word. Operations are provided, among others, to insert, extract, and test the individual bits. The mode **bytes** is similar, providing a way to pack characters into machine words to save storage. How many characters to pack into a machine word is a decision left to the implementer. Like **int** and **real**, **bits** and **bytes** have **long** and **short** versions.

The modes **sema**, **format**, and **file** have specialized uses and are covered later on.

ALGOL 68 allows more complex modes to be constructed from other modes in a variety of ways. Roughly speeking, these ways involve arrays, structures, procedures, sets, and pointers. We examine each of these in turn.

## 2.2 Array Modes

Many problems involve data which are organized into vector or matrices. A vector is a one-dimenisonal sequence of objects, all of which have the same mode. A matrix is a two-dimensional ordering of objects of the same mode. Likewise, three, four, and higher dimensional arrays also consist of collections of objects of the same mode. The elements of an array may be of a primitive mode, such as **int**, or they may be of a constructed mode.

The official ALGOL 68 term for array is multiple value (RR 2.1.3.4); however, we continue to use the more familiar word "array." An array is a run-time object and therefore has a value and a mode. Array variables exist and may be declared and assigned values, just as variables of any other mode are. A one-dimensional array of integers has mode [ ]**int**, pronounced "row of integer"; a two-dimensional array of reals has mode [,]**real**, pronounced "row row of real"; a three-dimensional array of characters has mode [,,]**char**, pronounced "row row row of character." In general, the mode of an n-dimensional array is an opening square bracket followed by n-1 commas, a closing square bracket, and then the mode of the elements. Objects of different dimensions have different modes.

When array variables are declared, the bounds must be specified in order to allow sufficient space to be reserved. To declare a one-dimensional integer array variable named "month" which is to contain an array whose elements are numbered 1 to 12, one writes

[ 1:12]**int** *month*

The lower and upper bounds are integer expressions; they are separated by a colon. Much as you would expect,

```
[ 0 :n−1, 0:n−1]real physicist, chemist
```

declares *physicist* and *chemist* to be $n \times n$ **real** matrices. Note that *physicist* is a [,]**real** variable; the bounds are not part of the mode (unlike PASCAL). Thus if

```
[ 1:100, 3:9]real geologist
```

declares a nonsquare matrix, *physicist* and *geologist* have the same mode, albeit different sizes. The following program declares several variables:

```
begin int n, m;
    read((m, n)); ¢ read 2 integers ¢
    ¢ unlabeled statements may be followed by more declarations, i.e., it is
        not necessary to put all declarations first ¢
    [ -n:n]int hamlet; ¢ size depends on n ¢
    [ 1:m, 1:n]real macbeth;
    [ 1:10, 1:10, 1:10]bool othello;
    [ -100:-80]char richard 3;
    [ 0:9×m + 6×m×n]string henry 8;
    ¢ array elements may themselves be arrays ¢
    [ 1 :10][1:5, 1:5]int king lear;
    skip ¢ dummy statement ¢
end
```

Elements of arrays may be extracted by subscripting and trimming (see Section 3.4, Slices).

In the preceding program, *king lear* is a 10-element vector, each of whose elements is a 5×5 square matrix. A vector whose elements are matrices might be a more natural representation for, say, the successive digitized frames of television broadcasting, than a three-dimensional array. Note that *king lear*[*n*] can be used anywhere an object of mode [,]**int** is needed, for example, as an actual parameter. It can also be subscripted, as in *king lear*[*n*] [2,3], but not as in *king lear*[*n*,2,3].

An array variable may be declared to be flexible, in which case arrays of different sizes may be successively assigned to it, provided they are of the proper mode. A string variable is actually a flexible one-dimensional character array variable.

## 2.3 Structured Modes

Arrays are used to group together objects of the same mode. Structures (an 2.1.3.3) are used to group together objects whose modes need not be identical. A structure is composed of one or more fields, each having a name, or more properly, a field selector (RR 4.8.1f). Structures themselves are objects and have modes. The mode of a structure depends upon the modes of its fields, their order, and the field selectors. Two structured modes are the same if and only if the corresponding fields have the same modes and field selectors. Structured variables exist, and may be assigned to one field at a time or "all at once." Structures are called "records" in some programming languages. An example of a structured variable declaration is:

```
struct (string species, int number of feel, bool makes good pet) beastie
```

This declares *beastie* to be a variable with three fields whose field selectors are: species, number of feet, and makes good pet. To use any of the fields of *beastie*, one writes the field selector, followed by the word **of**, followed by the name of the structured variable, for example:

```
species of beastie := "brontosaurus";
number of feel of beastie := 4;
makes good pet of beastie := false
```

The extraction of one field of a structure is called selecting. Alternatively, it is possible to assign all three fields at once by using a structure display (RR 3.3.1h) on the righthand side of the assignment statement, for example:

```
beastie := ("guinea␣pig", 4, true)
```

Some examples of structured variables follow:

```
begin int n; read(n);
   struct (real value, string color, bool leaks, hasfireplace) house;
   struct ([1:3]char aircraft type, int wheels, max speed) plane;
   struct ([1:3]char area code, [1:7]char phone number) telephone;
   ¢ farm has 3 fields: crop, farmer and dairy ¢
   struct ([1:n]struct (string variety, real acres) crop, string farmer,
       bool dairy) farm;
   skip
end
```

## 2.4 Procedure Modes

In contrast to most programming languages, ALGOL 68 considers procedures to be objects, complete. with values and modes. Furthermore, there are procedure variables, to which procedures can be assigned. The mode of a procedure is uniquely determined by the mode of its parameters (if there are any) and the mode of the value it returns. A procedure that takes an integer as a parameter and returns a real as a value has mode **proc**(**int**)**real**. A procedure that takes a character and a Boolean matrix as parameters and returns a real vector as a value has mode **proc**(**char**, [,]**bool**)[ ]**real**.

A procedure that is not used as a function, that is, does not return any explicit value, is said to return **void**. For example, a procedure that accepts an **int** as parameter and cancels the corresponding flight (in an airline reservation system) has mode **proc**(**int**)**void**. A procedure which has no parameters, but which returns a **real**, such as *random*, has mode **proc real**. A procedure which has no parameters and which delivers no explicit value has mode **proc void**.

Both parameters and results may have any mode. Unlike FORTRAN, ALGOL 60, and PL/1, in ALGOL 68, procedures may yield strings, arrays, structures, pointers, or any other mode. Furthermore, there is no reason procedure modes cannot be used as parameters or results. For example, a procedure used to perform a numerical integration of a real function (that is, a **proc**(**real**)**real**) between two real limits might have mode

**proc**(**proc**(**real**)**real**, **real**, **real**)**real**.

The order of the parameters is significant; **proc**(**real**,**int**)**void** and **proc**(**int**,**real**)**void** are different modes. Because there are an infinite number of combinations of parameters and results, there are an infinite number of procedure modes, just as there are an infinite number of procedure modes.

As mentioned earlier, procedure variables exist, and can be assigned values. The following program illustrates this feature:

```
begin real x;
    ¢ f is a proc(real)real variable ¢
    proc(real)real f;
    x := 3.14;
    ¢ sin, cos, and tan are standard ¢
    f := sin; ¢ assign sin to f ¢
    print (f(r)); ¢ print sin (3.14) ¢
    f := cos; ¢ now assign cos to f ¢
    print (f(x)); ¢ print cos(3.14) ¢
    f := tan; ¢ now assign tan to f ¢
    print f((x)) ¢ three guesses ¢
end
```

When an integer variable acquires a new value, as in *i* := 3, the bit pattern for the integer 3 is put into location *i*. Obviously, assigning *sin* to *f* is not going to cause a copy of the procedure's machine code to be stuffed into the variable *f*. The ALGOL 68 compiler writer must determine how to implement this, but presumably he will assign pointers to the procedure's code and environment (or the equivalent) to *f*. Some examples of procedure variable declarations follow:

```
begin
    proc(real)real cotangent;
    proc(int, int)int integer divide;
    proc(int, int)bool coprime;
    proc(char, char)bool char compare;
    proc([, ]real, [, ]real)[, ]real matrix add;
    proc(string, string)string concatenate;
    proc(int)void page eject;
    proc(int)struct(string name, int age) find;
    proc(int)proc(int)int pick function;
    proc(proc(real)real, real, real)real simpson;
    skip ¢ dummy statement ¢
end
```

As we discuss later on, actual parameters in procedure calls must be of the mode expected, for example, a **proc**(**int**)**real** requires an **int** as a parameter and will not accept a **real**. Sometimes it is convenient to have a procedure with a formal parameter that can be any one of several modes. For example, we might want to write a procedure that accepts a vector parameter of mode [ ]**int**, [ ]**real**, or [ ]**compl** and checks to see if any elements are zero.

To permit this sort of flexibility, ALGOL 68 permits programmers to create a special kind of mode called a united mode. A variable united from **int** and **real** can be assigned either an **int** value or a **real** value. Similarly, a variable united from [ ]**int**, [ ]**real**, and [ ]**compl** can accept a vector of integers, reals, or complex numbers as a value (but not a vector of Booleans). United mode variables are declared as indicated here:

```
begin
   union (int, real) ir;
   union [ ]int, [ ]real, [ ]compl) irc;
   union (proc(int)real, proc(real)real) u;
   skip ¢ dummy statement ¢
end
```

It is possible at run time to determine the mode of the value currently occupying a variable of united mode. This is done by using a variation on the **case** statement (RR 3.4.1h) with clauses for the various possible modes. Each clause is headed by a mode and (optionally) by an identifier, followed by a colon. The clause corresponding to the current mode of the united variable is executed. Unlike the normal **case** statement, the order of the clauses is irrelevant.

```
begin
    union (int, real, bool, char, bits, bytes, [ ]int, [ ]real) kitchen sink;
    ¢ here are 4 valid assignments ¢
    kitchen sink := 3;
    kitchen sink := 3.14;
    kitchen sink := true;
    kitchen sink := "a";
    ¢ random is a standard proc real ¢
    if random < .5
    then kitchen sink := 1
    else kitchen sink := 2.78
    fi;
    ¢ now figure out whether random was < .5 ¢
    case kitchen sink in
        ( int i ) : print (("integer", i)),
        ( real r): print (("real", r))
    esac
end
```

In this example we determined the mode of the union by using the case clause, and used the value in *kitchen sink* once its mode was known. Observe the (**int** *i*) and (**real** *r*) in the **case** clause. To compute with the value in *kitchen sink* in the **int** part (first clause) we can use the identifier *i*, now known to be an **int**. The value of *i* is the value of *kitchen sink*. Likewise the identifier *r* can be used in the second clause in any context where a **real** number is allowed. If *j* had been declared as an integer variable in the preceding program, *j* := *kitchen sink* would have been forbidden (by the grammar) and would have been flagged by the compiler. The reason is obvious: At the time of the assignment the compiler cannot guarantee that *kitchen sink* contains an integer, and we would be in trouble if it contained a [ ]**real**. However, inside the first **case** clause it is guaranteed that *kitchen sink* contains an integer. To make life easier for the compiler writer, the new name *i* is introduced; there is no doubt about the mode of *i*. Although *j*:= *kitchen sink* would be forbidden, even inside the first clause, the assignment *j*:=*i* would be allowed (only) in the first clause.

You may be wondering how unions are implemented. Presumably the compiler will have to reserve enough space in a united variable for the largest of the alternatives (or if that is too painful, perhaps only a pointer will be stored). Also, there must be some information stored that tells which mode is the "current" one. Note that there are no objects or values of united modes, just variables.

## 2.6 Reference-to Modes

Most programming languages are somewhat lax about making a distinction between the address of a variable and the contents of that variable. The nature of the difficulty can be most easily seen by means of an example from FORTRAN:

```
SUBROUTINE SUM(I,J,K)
INTEGER I, J, K
I = J+K
RETURN
END
```

Now consider the result of the following call:

```
CALL SUM(1,2,3)
```

Although the subroutine declaration is grammatically correct, and the call is also grammatically correct, something is obviously wrong.

The problem is not that the actual parameters are of the wrong type. 1 is declared an integer, and the number 1 is certainly an integer. The trouble occurs because the lefthand side of an integer assignment must evaluate to the address of a variable, not to an integer value. Few compilers will even give a precise error message at run time, let alone at compile time. Typically an address pointing into the run-time constant table is passed as a parameter, and the value of the constant 1 is changed to 5 so that subsequent `N = 1+1` statements set **n** equal to 10 (decimal, not binary). In ALGOL 68 integer variables and integer values have different modes; so the error we are considering will be detected at compile time as a parameter mismatch.

An integer variable in ALGOL 68 has mode **ref int**, (**ref** is a shortened form of reference to); a real variable has mode **ref real**, etc. Consider the ALGOL 68 program

**begin int** *i*; *i* := 3 **end**

In this program, *i* is an integer variable and has mode **ref int**. The constant 3, on the other hand, has mode **int**. A **ref int** corresponds to the address of a memory location into which an integer can be put, whereas an **int** is a value, not an address.

This distinction is very important and bears repeating. An integer constant and an integer variable are different kinds of objects and have different modes. The mode of the former is **int** and the mode of the latter is **ref int**. The value of an integer variable is its memory address. Of course, given an integer variable one can ask about both its value and the value of the integer it eentains, but those are clearly different objects.

The ALGOL 68 rule for an integer assignment is that the left-hand side must be, or be convertible to, an object of mode **ref int**, while the right-hand side must be, or be convertible to, an object of mode **int**. Precisely the same considerations hold for other modes, of course. Although procedure definitions are discussed later, the ALGOL 68 version of `SUM` is presented here for contrast with the FORTRAN version.

**proc** *sum* = (**ref int** *i*, **int** *j*, *k*) **void**: *i*:=*j*+*k*

Here *i* is clearly a different mode from *j* and *k*. Furthermore, the call *sum*(1,2,3) is invalid because the modes of the actual parameters (**int**, **int**, **int**) do not match the modes of the formal parameters (**ref int**, **int**, **int**). If *i* had been specified as mode **int** instead of **ref int**, then the assignment *i* := *j*+*k* would have been detected as an error because the left-hand side of an integer assignment must evaluate to something of mode **ref int**. Either way the error would have been detected by the compiler, which is obviously better than its subsequent appearance as an obscure program bug.

We have consistently said that the righthand side must be, or be convertible to, to an object of mode **int**, rather than having said that the right-hand side must be an object of mode **int**. This choice of words was deliberate.

Consider the following program:

**begin int** *i*, *j*; *i* := 3; *j* := *i* **end**

In this program, *i* and *j* are both of mode **ref int**. In the first assignment the right-hand side has mode **int** as it should, but in the second assignment the right-hand side has mode **ref int**. Thus it would appear that *j* := *i* is forbidden. Fortunately, there exists an automatic conversion between mode **ref int** and **int**, which is called dereferencing. Conversions between data types are familiar from other programming languages; for example, nearly all programming languages allow an integer to be written in a position where a **real** number is required, with automatic conversion implied.

In exactly the same way, ALGOL 68 often allows an object of mode **ref m** to be written when an object of mode **m** (some arbitrary mode) is expected, with automatic conversion implied. Such automatic mode conversions are called coercions. There are six kinds, of which dereferencing is one. Integer to real coercion, called widening, is another. Chapter 6 of the Revised Report gives the exact rules about which coercions are allowed in what situations.

It should be pointed out that although widening from **int** to **real** is typically an actual operation performed on integers at run time, dereferencing need not be performed at run time. If the computer has an instruction to move the contents of location *i* to location *j*, the compiler writer is obviously allowed to use it. No one is going to compel him to first put the address of *i* in a register and then explicitly dereference it using indirect addressing before storing the contents of *i* in *j*.

Dereferencing is more than simply a syntactic trick to allow variables on the righthand side of assignments. Since **ref int** is a valid mode, the curious reader may wonder if reference-to-integer variables exist. The answer is yes. Just as an integer variable is a location in memory intended to hold an integer, a reference-to-integer variable is a location in memory ntended to hold an object of mode **ref int**, that is, the address of an integer variable. In other words, a reference-to-integer variable can contain a pointer to an integer variable. It cannot contain a pointer to a real variable or to any other kind of variable, however. Likewise, a reference-to-complex variable may contain only a pointer to a complex variable.

Consider the following program:

```
begin
    ref int pt;
    int i, j;
    i := 0; j := 4;
    if random < 0.5
    then pt := i
    else pt := j
    fi
end
```

If *pt* is dereferenced once, it yields either the address of *i* or the address of *j*. If it is dereferenced twice, it yields either 0 or 4. Barring some unusual hardware, dereferencing a pointer twice is going to involve some run-time action. Note that *pt* itself has mode **ref ref int**.

Finally we get back to the subject of mode construction. The rule for creating pointer modes is simple. If **m** is some arbitrary mode, then **ref m** is also a mode of pointers to **m**. Applied repeatedly we discover that, **m**, **ref m**,

**ref ref m**, **ref ref ref m**, etc., are all distinct modes. The program at the top of page 167 shows how to declare some modes involving pointers:

```
begin
    [ ]ref int a; ¢ a row of pointers ¢
    ref [ ]real b; ¢ a pointer to a vector ¢
    ref [ ]ref char c; ¢ a pointer to a pointer vector ¢
    struct (ref int p, ref real q) d; ¢ 2 pointers ¢
    proc ref bool e; ¢ proc yielding a pointer ¢
    ref proc bool f; ¢ pointer to a proc bool ¢
    union (ref int, ref bool) g; ¢ either of 2 pointers ¢
    skip ¢ dummy statement ¢
end
```

Variables involving **ref** "something" are typically used in list processing applications. The distinction between the mode of a variable and the mode of the objects that can be assigned to it is crucial, but often initially confusing to people accustomed to other programming languages. Variables have mode **ref** "something," and can contain objects of mode "something." In addition, a variable is itself an object, with a mode and a value. The mode of an integer variable is **ref int**, and its value is the address where the integer is stored. Thus an integer variable can be regarded as an object of mode **ref int**, and it can be assigned to a pointer variable whose mode is **ref ref int**.

In some programming languages (for example, PL/1), a pointer can point to an object of any mode. This is a frequent source of errors. Often a pointer somehow ends up pointing to a variable of the wrong mode, or worse yet, points into the program itself or to unused memory. By strictly categorizing pointers according to what they may point to, ALGOL 68 greatly reduces the opportunities for making errors.

## 2.7 Mode Declarations

We have now seen how ALGOL 68 programmers can construct new modes from primitive modes through the use of arrys, structures, procedures, unions, and references. ALGOL 68 provides a mechanism for programmers to give names to newly created modes, so they can be used in the same way that built-in modes are used. New modes are declared by means of a mode declarator (RR 4.2) as illustrated in the next column.

Mode declarations are used to create new data types. It is possible for user-definer modes to be used to create still more complex modes, as in family, which uses **person**. ALGOL 68 provides the ability for the programmer to build up an entire library of mode definitions tailored to his particular application.

```
begin int a, size; read ((n, size));
    mode vector = [1:n]real;
    mode matrix [1:n, 1:m]real;
    mode rational = struct (int num, denom);
    mode functionset = [1:n]proc(real)real;
    mode book =struct (string title, author, publisher, int pages, year,
         bool paperback);
    mode magazine =struct (string title, publisher, int subscribers, publ
         frequency);
    mode library = [1:size]union(book, magazine);
    mode person =struct (string initials, ref person ma, pa, int age, bool
         too fat);
    mode family =struct (person mommy, daddy, [1:2]person child);
    mode bridgehand = [1:13]struct(char rank, suit);
    mode word = [0:15]bool;
    mode memory = [0:4095]word;
    mode instruction1 = struct (int opcode, address1, address2, address3);
    mode instruction2 = [1:4]int;
    mode flight = struct (string plane, pilot, movie, bool nonstop,
           [ 1:size]struct(string name, [1:10]char phone) passenger);
    mode multireal = union (real, long real, long long real);
    mode tree =struct (int value, ref tree right, left);
    mode integer =int;
    mode interger =int; ¢ for bad spellers ¢
    skip ¢ dummy statement ¢
end
```

A few comments about mode declarations may be helpful. The modes **instruction**1 and **instruction**2 each consist of four integers. If *add* is declared to be **instruction**1 and *sub* is declared to be **instruction**2, then the fields of *add* are accessed via the field selections:

*opcode* **of** *add*, *address*1 **of** *add*, *address*2 **of** *add*, *address*3 **of** *add* ,

whereas the components of *sub* are aceessed by subscripting:

*sub* [1], *sub* [2], *sub* [3], *sub*[4] .

Which choice is made depends upon the application.

The mode **tree** is interesting. It has three fields, an integer and two pointers. In terms of allocating space for tree variables, it hardly matters that the pointers point to objects of mode **tree**. Binary trees and graphs are widely used in computer science, so modes like this are valuable. A mode that is defined in terms of itself, like **tree**, is called a recursive mode. Note that although the nonrecursive mode declarations are used merely for convenience, the recursive modes really require the mode definition facility (try declaring a variable with the same mode as **tree** just using a **struct**(**ref**. . . ) ).

One must exercise some care when defining recursive modes. For example:

> **mode bush** = **struct** (**int** *v*, **bush** *h*, *t*)

is incorrect. Suppose that an **int** requires one word of memory, and a **bush** requires k words of memory. Then a declaration like

> **bush** *blueberry*

would require that the variable *blueberry* be allocated enough memory to store one object of mode **int** (one word) and two objects of mode **bush** (2k words) for a total of 2k+ 1 words. This contradicts our statement that a **bush** requires only k words. The mode declaration is impossible. A **bush** can hardly contain two **bush**es and then some. In contrast, the mode **tree** presents no such problem since it only claims space for an **int** and two addresses (pointers), not two objects of mode **tree**. As you might expect, ALGOL 68 allows all the modes that are intuitively reasonable and prohibits those that are not (RR 7.4).

Same modes can be "spelled" in more than one way. For example, in

> **mode m**1 = **union** (**int**, **real**);
> **mode m**2 = **union** (**real**, **int**)

**m**1 and **m**2 represent the same mode. On the other hand,

> **mode m**3 = **struct**(**int** *i*, **real** *r*);
> **mode m**4 = **struct**(**int** *j*, **real** *r*);
> **mode m**5 = **struct**(**real** *r*, **int** *i* );

are three different modes because the field selectors are part of the mode. Mode equivalence is dealt with in RR 7.3.

At least one aspect of the orthogonal design of ALGOL 68 may now be clearer. From the 11 primitive modes listed in Section 2.1 and the five simple mode construction rules listed in Sections 2.2 through 2.6, one has the ability to create a large and powerful collection of new data types.

In contrast, PL/1 is not orthogonally designed; there are no simple rules telling which combinations of attributes are allowed and which are not. A complete specification of the allowed "modes" in PL/1 can only be encoded by giving a large table of compatible and incompatible attributes. This difference is characteristic of other aspects of ALGOL 68 and PL/1 as well.

## 2.8 Using New Modes

Variables of user-created modes are defined in the same way that variables of the primitive modes are: first the mode, then a list of one or more identifiers. Declarations are commands to the compiler to reserve storage for variables. Keep in mind that the compiler needs to know how much storage to reserve. When declaring an array variable, one must specify the actual bounds (evaluated at run time) in order for the compiler to reserve enough space. On the other hand, when declaring a pointer to an array (for example, **ref** [ ]**int**), the bounds are not needed, since the only storage reserved is that required for the pointer, not the array; a pointer to a big array takes up the same space as a pointer to a small array. However, to be used, the pointer must appear on the left-hand side of an assignment, with some array (itself declared with bounds) on the right-hand side.

The rules for when bounds are and are not needed are given in RR 4.6.

When a mode declaration contains a variable or an expression in an array bound, for example, $n$ in mode **vector** above, the question arises whether the value of $n$ at mode declaration time or at variable declaration time is the one that is used. Consider this program:

```
begin
    int n;
    n := 3;
    mode vector = [1:n]int;
    n := 25;
    vector x;
    n := 75;
    vector y;
    skip ¢ dummy statement ¢
end
```

It is the value of $n$ at variable declaration time that matters; $x$ has 25 elements and $y$ has 75 elements. The value of $n$ at mode declaration time is irrelevant. In a certain sense, variable declarations are "carried out" at run time, providing more flexibility than most languages allow. (Of course, a clever compiler writer will try to do as much as possible at compile time.)

Variables may be declared with initial values by following the identifier with a "becomes" symbol (:=) and the initial value. Structures and arrays may also be initialized, with parenthesized lists of values. It is also possible to partially initialize structures or arrays by using **skip** for some of the fields or elements. The value of **skip** is undefined; these elements or fields must be initialized by explicit assignment before being used.

Some sample variable declarations are shown below.

## 3. UNITS

```
begin ¢ mentally insert the mode declarations of section 2.7 here ¢
    int a := 3, size := 2;
    char c := "q";
    real length := 2.503;
    vector v := (14.2, -9.1, 3.5678);
    matrix a := ((1.0, 2.0, 3.0), (0.6, -0.9, 100.0), (1.1, 2.1, 3.4));
    rational rat := (1, 2), tar := (3, 4);
    functionset f := (sin, cos, tan);
    book censored := (skip, skip, skip, skip, skip, false);
    magazine cs := ("computing␣surveys", "acm", 22000, 4);
    library mini := (censored, cs);
    person tom := ("trj", skip, skip, 40, true);
    person mary := ("mej", skip, skip, 41, false);
    family jones := (mary, tom, (skip, skip));
    bridgehand south := (
            ( "A", "S"), ("K", "S"), ("Q" "S"), ("J" "S"),
            ( "A", "H"), ("Q", "H" ), ("9", "H" ), ("7", "H" ),
            ( "K", "D"), ("Q", "D"), ("T", "D"),
            ( "Q", "C"), ("J", "C"));
    word w; memory mem;
    flight twa 156 := ("747", "bill", "frankenstein", true, skip);
    skip ¢ dummy statement ¢
end
```

Like other programming languages, ALGOL 68 requires that expressions be placed in certain contexts, for example, on the right-hand side of assignments, as actual parameters in procedure calls, and as subscripts. Expressions are called units in ALGOL 68 and are much more general than in many other programming languages. In the following subsections we discuss 15 kinds of units. The complete list is given in RR 5.1A.

## 3.1 Denotations

The simplest form of a unit is a denotation (usually called a constant in other programming languages). Typical denotations of mode **int**, **real**, **bool**, **char**, and **string** are: 4, 3.6, **true**, "x", and "hi". "Constants" of array and structured modes are also allowed. They are called row displays and structure displays, respectively, and consist of parenthesized lists of values. For example,

[ 1:2, 1:3]**int** *r*2 := ((1, 2, 4), (8, 16, 32))

illustrates the use of a row display. Denotations are described in Chapter 8 of the Revised Report.

## 3.2 Variables

The next simplest unit is the variable. In this statement,

**begin** **int** *i*, *j*; *i* := 3; *j* := *i* **end**

*j*, a variable, is used as a unit in the second assignment. (Remember that it is dereferenced to an integer unit.)

## 3.3 Formulas

A formula (ita 5.4.2) is an operator and its operand or operands. A monadic formula has one operand, for example, **abs** *i*, -*x*, and **sign** *y*. Dyadic formulas have two operands, for example, *i*-*j*, *x* <*y*, and "abc" + "xyz".

ALGOL 68 has well over 100 "built-in" operators (listed in RR 10.2) and provides a mechanism that allows programmers to define new ones, just as it provides a mechanism to define new modes. Operators are akin to procedures. Each operator expects to have one or two operands of specific modes, and delivers a result of a specific mode. The same symbol may represent two different operators (cf. GENERIC in PL/1).

In the following program:

```
begin
    int i := 1, j := 2, k;
    real x:= 0.1, y := 0.3, z;
    k := i+j;
    z := x+y
end
```

the first + represents an operator with integer operands and an integer result, whereas the second + represents a different operator with real operands and a real result. Very likely they will require different hardware instructions.

A formula may be used as an operand. For example, the formula *i*+*j* could be used as an operand of <, as in *i*+*j* < *k*, which is a formula yielding a Boolean result (assuming + has higher precedence than <, which it has).

## 3.4 Slices

Arrays may be subscripted as in other languages. When an object of mode [ ]**m** is subscripted, an object of mode **m** is yielded. ALGOL 68 also permits a generalization of subscripting called trimming, yielding some cross section of the original array. If *z* has been declared by [1:10]**int** *z*, then *z*[1 :7], *z*[1 :10], and *z*[2:5] are examples of units (slices) and can be used in assignments, actual parameters, etc. For example:

```
begin
    [ 1 :10]int a, b;
    [ 1:20]real x; [1:20, 1:20]real xx;
    read((a, b, x, xx));
    b[1:4] := a[1:4]; ¢ assigns 4 elements ¢
    b[3:9] := a[1:7]; ¢ assigns 7 elements ¢
    b[1:10] := a[1:10]; ¢ assigns a to b ¢
    b := a; ¢ same as above ¢
    xx[4, 1:20] := x; ¢ assign to row 4 of xx ¢
    xx[8:9, 7] := x[1:2] ¢ xx[8,7] := x[1]; xx[9,7] := x[2] ¢
end
```

A trimmer, such as 1:4 in the first assignment does not affect the dimensionality of the array, whereas a subscript (just one bound, with no colon) reduces it by one, as in *xx*[4,1 :20].

All combinations of trimming and subscripting are valid. For example, if s is a three-dimensional array, *s*[*i*,*j*,*k*], *s*[*i*,*j*,*k*1:*k*2], *s*[*i*,*j*1:*j*2,*k*1:*k*2], and *s*[*i*1:*i*2,*j*1:*j*2,*k*1:*k*2] can be used as a variable, and one-, two-, and three-dimensional arrays, respectively. Furthermore, *s*[*i*,*j*1:*j*2,*k*], *s*[*i*1:*i*2,*j*,*k*1:*k*2] and other combinations are also allowed. In an assignment, the bounds must "match," as described in an 5.3.2. Subscripting and trimming are collectively called slicing.

## 3.5 Selections

A selection consists of a field selector, the symbol of, and a structure to be selected from. The field selector must be an identifier and cannot be computed (because it is not an object). The structure being selected from may, however, be the result of evaluating an expresston.

If a mode involves both structures and rows, a unit derived from an object of that mode may involve slicing (subscripting or trimming) and selecting. Slicing binds more tightly than selecting; so *tail* **of** *dog* [*k*] means *tail* **of** (*dog*[*k*]) and not (*tail* **of** *dog*)[*k*]. If *tail* **of** *dog* yields an array, then (*tail* **of** *dog*)[*k*] is the correct way to extract the kth element of that array. When combining selecting and slicing, keep in mind that any array can be sliced and that any structure can be selected from. Here are some examples:

```
begin int m := 25, n := 40, k := 2;
    mode person = struct (string initials, int age);
    mode course = struct (person prof, [1 :n]person student);
    mode dept = [1:m]course;
    person smith, jones, brown, davis;
    course painting, drawing, etching;
    dept art;
    ¢ begin assigning values ¢
    initials of smith := "rbs";
    age of smith :47;
    jones := ("tmj", 32);
    prof of painting := ("jed", 47);
    prof of drawing := smith;
    prof of etching := prof of painting;
    art[1:3] := (painting, drawing, etching);
    prof of art[2] := jones; ¢ smith quit ¢
    age of prof of art[2] := 39;
    ( student of art[2])[1] := davis;
    age of (student of art[2])[1] := 18;
    ( student of art[k+1])[k-1] := ("tns", 19)
end
```

This may look imposing at first, but it is really quite logical. The key is to keep track of the mode of the objects. When faced with an array, like (*student* **of** *art*[2]), one slices. When confronted with a structure, like *prof* **of** *art*[2], one selects a field from it. If you still think ALGOL 68 is unnecessarily complicated, try to rewrite the preceeding program in FORTRAN.

## 3.6 Procedure Calls

The mode of a procedure is uniquely determined by the modes of its parameters and its result. If a procedure returns mode **m**, then a call of that procedure is a unit of mode **m** and may be used anywhere a unit of mode **m** is needed. If a procedure *p*1 has mode **proc**(**real**,**bool**)**int**, then *a*[*p*1(3.14,**true**)] show a call of *p*1 used as a subscript. Similarly, if a procedure *p*2 has mode **proc**(**int**)**bool**, then **if** *p*2(6) **then** *print*(*k*) **fi** is legitimate.

A procedure call has two parts: the procedure to be called, and the parameter list. The first part may be the result of a computation, for example,

```
begin int i; real x, y;
    [ 1:3]proc(real)real f := (sin, cos, tan);
    to 100 ¢ repeat 100 times ¢
    do read ((i, x));
        ¢ i selects sin, cos, or tan to call ¢
        y := f[i](x);
        print (y)
    od
end
```

A function with no parameters (that is, of mode **proc int**) is not "called". Instead the procedure name is written with no parameter list. For technical reasons this is not regarded as a procedure call, but as a type conversion (coercion) from mode **proc m** to mode **m**. It is called deproceduring (RR 6.3) and is completely analogous to the widening coercion from **int** to **real** in **real** *x* : 3 or the dereferencing coercion in (**int** *i* := 1,*j*; *j* := *i*). If deproceduring did not exist, then **real** *x* := *random* would have to be prohibited, since *random* has mode **proc real** and on the right-hand side in the preceding example a **real** is needed.

## 3.7 Assignments

An assignment (called an assignation in the Revised Report) consists of a destination (the left-hand side), a "becomes" symbol (:=), and a source (the right-hand side). A **ref m** assignment has a **ref m** unit as the destination and an **m** unit, or something coerceable to an **m** unit, as the source (RR 5.2.1.1).

Having detected a **ref m** destination, the compiler will coerce the source by all possible means to **m**. We emphasize that after all coercion the destination is mode **ref m** and the source is mode **m**.

An assignment can stand by itself as a statement, or be used itself as a unit. It may, for example, be used as a source in another assignment (but not as a destination to avoid certain ambiguities). For example, $j := k$ is an assignment and as such may be used as the source in $i :=$ *source*, yielding $i := j := k$. Because ALGOL 68 allows assignments as sources, it also gets multiple assignments, as an extra added attraction, for free. Furthermore, $a[i := i+ 1]$ is a perfectly valid way of subscripting the array $a$: first the assignment is carried out, and then the newly assigned value of $i$ is used as a subscript.

## 3.8 Generators

ALGOL 68 provides two storage management strategies: local and heap. Local storage consists of a last-in, first-out stack. Whenever a procedure is called (and perhaps when a **begin end** block is entered, depending on the implementation) a new stack frame is created for all local variables needed in it. When it is exited, the storage is released by resetting the stack pointer to the value it had prior to entry. This leads to a simple and efficient method for allocating storage.

All variables declared in the usual way use the local storage discipline. In addition, the programmer may explicitly request more stack storage to be reserved by using a local generator, **loc**, followed by a specification of the mode desired (the mode is needed because **loc** $[1{:}n]$**compl** may take much more space than **loc bool**). The value of the generator **loc m** is the address of the the object, that is, a pointer to it, and as such has mode **ref m**.

An example may make the use of local generators clearer.

```
begin ¢ calculate something ¢
    begin ¢ demonstrate triangular arrays ¢
        int n; read(n);
        [ 1:n]ref[ ]real triangle;
        for k from 1 to a
        do triangle[k] := loc[1:k]real;
            ¢ fill in some values ¢
            for j from 1 to k
            do triangle[k][j] := k+j od
        od
    end
        ¢ storage used by triangle is now released ¢
end
```

Numerical analysts often deal with symmetric $n \times n$ matrices. Using a representation of $n$ columns of $n$ elements each is wasteful of storage. The preceding program declares triangle to be a row of pointers, each pointing to a different real vector. The vectors pointed to are created during execeution of the program, each newly created vector being one element larger than its predecessor. When the preceding block is exited, all the storage reserved can be released. That is why these generators are called local generators: the effect is local to the block they occur in.

The array *hamlet* in the example at the end of Section 2.2 is allocated by essentially the same mechanism as the array triangle just given. That is why unlabeled statements can be allowed before declarations.

Incidently, the declaration of triangle should be carefully noted. Actual bounds are needed in the first brackets, but not in the second because triangle is a vector of pointers. The compiler has to know how large the vector is in order to reserve space for it, but for the purposes of allocating space to triangle, it does not matter what is being pointed to. In fact, bounds are never needed in a mode following a **ref**.

The other storage management scheme is the heap. The heap is a single homogeneous section of memory from which storage can be acquired by heap generators, of the form **heap m**, where **m** is the specification of the mode of the object needed. Because heap objects are not dependent on the stack discipline, they do not vanish when the block in which they were created is exited. When the heap is exhausted, a run-time garbage collector has to come in and recycle the garbage. For example,

```
begin ref[ ]real ptr;
    to 1 000 000 ¢ repeat a million times ¢
    do ptr := heap[1:1000]real od
end
```

is a lovely little test to see whether your garbage collector is working properly. Passing through the loop the first time, a piece of the heap is allocated for a 1000-element real array and the address of the array is assigned to *ptr*. Passing through the loop the next time, the same thing happens, overwriting the address of the first array, which now becomes garbage because there is no way to access it. On some subsequent pass, all the free space on the heap will be gone, and garbage collection will be automatically invoked to recover unused storage.

Note that if a local instead of a heap generator had been used in the preceding example, the stack frame would have kept growing and growing until all of memory was full. Since stack storage is only released at procedure (or possibly block) exit, the program eventually would have been aborted with a "stack overflow" message.

## 3.9 Nil

In list processing applications, it is necessary to have some marker to indicate the end of a list. When programming in Assembly Language, zero is often used. In ALGOL 68 a special symbol, **nil** (RR 5.2.4), is provided to end lists.

## 3.10 Identity Relations

When performing list processing, it is sometimes necessary to compare two pointers to see if they point to the same object. This can be done using identity relations. Identity relations are also used to compare pointers to **nil**. In practice, it is usually necessary that the programmer specify the mode required using a cast (see next Section 3.11). For example, consider a variable, *ptr*, declared by: **ref person** *ptr*, that is, *ptr* can point to an object of mode **person**. To see if *ptr* points to **nil**, one uses the construction

```
ptr :=: ref person (nil).
```

The identity relators :=: and :≠: are not operators (because they act on an infinite number of modes), but they may be regarded roughly as operators of infinitely low precedence. Thus, for example,

```
if i <j ∧ ptr :=:nil then
```

means

```
    if(i <j ∧ ptr) :=: nil then
```

which is probably not what was intended. To illustrate heap generators, **nil**, and identity relations, we give a simple program that reads in people's bowling scores and stores the information as a singly linked list. In phase two, names are looked up and the scores are retrieved.

```
begin
    mode person = struct(string name, int score, ref person next);
    ref person first := nil, ptr;
    string bowler; int bowled;
    bool still looking;
    make term (stand in, "␣");
    while read(bowled, bowler)); bowled > 0
    do first := heap person := (bowler, bowled, first)
    od;
    ¢ phase 2. look ap the scores ¢
    while read ((newline, bowler)); bowler ≠ ""
    do ptr := first; still looking := true;
        while (ptr :≠: ref person (nil)) ∧ still looking
        do if name of ptr = bowler
            then print((bowler, score of ptr, newline));
                still looking := false
            else ptr := next of ptr
            fi
        od;
        if still looking
        then print((bowler, "not␣in␣our␣league", new line))
        fi
    od
end
```

Some comments may be helpful. The condition in a **while** statement consists of zero or more statements followed by a unit. In the first **while** statement in the preceding program, two variables are first read, and then the condition (*bowled* $> 0$) is evaluated. The data are arranged in such a way that there is one person per card, first a score, and then a name. It is necessary to specify the string delimiter for the name, and this is done by the call to the procedure *make term*, defining space as the string delimiter for the standard input file, *stand in* (see Section 9, Input/Output).

Let us imagine that the first two people are named Adam and Eve, with bowling scores of 105 and 107, respectively. For the execution of the initial **while** loop, first points to **nil**. After the loop has been executed once, an object of mode **person** has been created, with its three fields initialized to (`"Adam"`, 105, **nil**). The variable first then points to this object. Passing through the loop the next time, a second object of mode **person** is created, with its fields initialized to (`"Eve"`, 107, *pointer to first object*). Now first points to Eve, which points to Adam, which points to **nil**.

## 3.11 Casts

Most of the time it is not necessary to specify the mode of a source, destination, operand, etc., explicitly. It is usually obvious from context. However, to handle those situations that are inherently ambiguous, the required mode may be specified explicitly using a construction called a cast (RR 5.5.1), one form of which consists of a mode followed by a parenthesized unit, as in **ref int**(*i*).

To understand why casts are needed, examine this program:

```
begin int i := 0, k := 1;
   ref int ptr := i;
   ptr := k;
   print (i)
end
```

Consider what *ptr* := *k* does. On one hand, it looks like an innocuous assignment of the address of *k* to a pointer variable (*ptr* has mode **ref ref int**, and *k* has mode **ref int**).

But on the other hand, suppose both *ptr* and *k* were dereferenced, yielding a **ref int** object as destination and an **int** object as source. If that happened, *i* would be assigned the value 1, quite different from assigning *k* to *ptr*. To avoid the occurrence of this ambiguity, the ALGOL 68 grammar was constructed in such a way as to prevent dereferencing destinations. This means that the preceding program prints 0, not 1.

Now comes the 64 dollar question: suppose you actually intended the second interpretation; how can that be achieved? Answer: use a cast; that is, replace the assignment by **ref int** (*ptr*) := *k*. The cast explicitly forces *ptr* to be converted to mode **ref int**. Since *k* cannot be assigned to a **ref int**, *k* is dereferenced. (Dereferencing is allowed for sources.)

When a cast is used, two modes are involved: the starting mode (the mode of the object inside the parentheses), and the goal mode (the mode listed before the open parenthesis). If there is no coercion path between the starting and the goal modes, the cast is invalid. For example, if *ptr* has mode **ref proc ref int**, then **real**(*ptr*) is a valid cast because *ptr* can be dereferenced, deprocedured, dereferenced, again, and widened. However, **bits** (3.14) is invalid because there is no coercion path from **real** to **bits**. Coercion is discussed in more detail in Section 4, Coercions.

## 3.12 Choice Clauses

ALGOL 68 allows **if** "statements" and **case** "statements" to be used as units if they produce the proper mode. This is illustrated by the following program:

```
begin ¢ examples of choice clauses ¢
    int i, j, k;
    real x := 0.1, y := 0.2, z := 3.1;
    read ((i, j, k ));
    [ 1:10]bool a, b;
    x := if i<0 then .3/x else z+4.0 fi;
    for n from i to if j = 2 then 1 else k fi
    do b[n] := true od;
    b[case i in 6, 3 out 4 esac] := false;
    if i = 0 then j else k fi :=
        if j>0 then j+1 else k-1 fi;
    [ if i>0 then 1 else k-1 fi :10] int c;
    z := if i>3 then sin else cos fi (3.14);
end
```

Three kinds of choice clauses exist: Boolean, integer, and united.

• The boolean choice clause is the familiar **if** . . . **then** . . . **else** . . . **fi** construction. If the **else** part is absent and the condition is false, the result is undefined.

• The second choice clause has the form **case** . . . **in** *clause*1, *clause*2, *clause*3, . . . , *clause n* **out** . . . **esac**. The integer expression between **case** and **in** selects one of the clauses by indexing into the clause list. Thus the order of the clauses is critical. If there is no **out** part, and the expression is out of range, the result is undefined.

• The third choice clause takes a united variable (such as *kitchen sink* used in Section 2.5) and selects one of the clauses based upon its current mode. The order of the clauses for this type of choice clause is irrelevant.

Variables may be declared in the condition, or integer parts, initialized, and then used in the succeeding parts; that is, their scope encompasses the entire choice clause. For example:

```
begin ¢ print smaller of 2 numbers ¢
    if int i, j; read ((i, j)); i<j
    then print (i)
    else print (j)
    fi
end
```

ALGOL 68 (and common sense) requires that all the possible choices in a unit be of the same mode, or be coerceable to the same mode. The unit **if** $i$<0 **then** 4 **else** $j$ **fi** can be used anywhere an integer unit is expected, because the **then** part is already an integer and the **else** part can be converted to one by dereferencing it. The unit cannot be used as a destination, however, because the **then** part cannot be converted to a **ref int**; there is no "referencing" coercion.

Now consider the assignment in:

```
begin int i, k; read (k);
    i:= if k<0 then 6 else true fi
end
```

This assignment is incorrect because an integer unit is needed as the source, and it is not possible to coerce all choices to mode **int**; namely, **true** cannot be turned into an integer. The problem of making sure all choices can be converted to the proper mode is called balancing (RR 3.2.1e). Since an assignment may have long **case** units, both as source and destination, just determining the proper mode of the assignment may itself be a substantial task for the compiler. However, the grammar was constructed in such a way as to insure that there is only one possibility.

ALGOL 68 allows **if**, **then**, **else**, and **fi** to be written as (, |, |, and ), respectively. Thus **if** $k$<0 **then** $i$ **else** $j$ **fi** can be written ($k$<0 | $i$ | $j$). This is often convenient in constructions like:

```
x := (i<0 |y |z) + (i<0 | 4.0 | z+2)
```

## 3.13 Closed Clauses

ALGOL 68 is an expression language. This means that every executable statement or group of statements can (at least potentially) deliver a value. A serial clause (RR 3.2) is a series of zero or more declarations and/or "statements" followed by a unit. The mode and value of the serial clause consist of the mode and value of the final unit. A closed clause is a serial clause enclosed by **begin end** or by parentheses. A closed clause has the mode and value of the serial clause, that is, of the last unit in the clause.

Some examples of closed clauses follow:

```
begin ¢ closed clauses ¢
    begin int i; read(i); i end;
    begin real x; read(x); sin(x) end;
    begin int i, j; read((i, j)); i+j end;
    begin [1:10]int a;
        for i from 1 to 10 do a[i] := i×i od; a
    end;
    ( int i; i := 20);
    ( "horse");
    ( (10, 20, 30, 40));
    ( (((((0))))));
    ( int i; (i := 3))
end
```

Since closed clauses are units, they may be used in the same way any other units are used, even if this seems peculiar at first.

Closed clauses may be used as sources; subscripts; **from**, **by**, or **to** parts in **for** statements; etc. For example, the following statement is perfectly valid:

```
k := (int i; read(i); i+1)
```

## 3.14 Skip

There is a special unit, **skip**, which is explicitly undefined. It takes on whatever mode is needed. As we have seen earlier, it can be used to omit the initialization of an element or field of a row or structure display, or to serve as a dummy statement. Do not confuse **skip** with **nil**; **nil** is a specific value that can be tested for; **skip** is just "filler" to make a construction syntactically correct, vaguely analogous to `CONTINUE` statements in FORTRAN.

## 3.15 Routine Texts

Since ALGOL 68 allows procedure variables, it is only natural that it also allow procedure units, so that there is something to assign to these procedure variables. A routine text is a procedure body, headed by the formal parameter list, if there is one. We discuss routine texts in the context of procedures and operators later. As a preview, we give a few examples of routine texts as sources:

```
begin proc(real)real f;
    proc int p; proc void q;
    f := (real r) real: 3.14/r;
    f := (real s) real: s+4.0;
    f := (real t) real: sin(cos(t));
    p := int: 3;
    p := int: (int k; read(k); k);
    q := void: print ("hello")
            ¢ note: no procedures have been called ¢
end
```

Note that the formal parameters can be used in the body of the routine text, following the colon.

## 3.16 Other Units

For the sake of completeness, we note that loop clauses (for loops), jumps (**goto** statements), formats, parallel clauses, and collateral clauses (for example, row displays) are also units in the technical sense (RR 5.1A).

## 4. COERCIONS

Coercion is the Algol 68 term for automatic mode conversion. Unlike some languages (notably PL/1) that allow practically anything to be converted into practically anything else, Algol 68 has very few automatic conversions. Automatic conversions often lead to unexpected and unwanted results, so Algol 68 was specifically designed to keep them well in hand.

There are exactly six kinds of coercions, each converting some class of modes into another. The six kinds of coercions are:

| coercion | input mode | output mode |
|---|---|---|
| dereferencing | **ref m** | **m** |
| deproceduring | **proc m** | **m** |
| widening | **int** | **real** |
| widening | **real** | **compl** |
| widening | **bits** | [ ]**bool** |
| widening | **bytes** | [ ]**char** |
| rowing | **m** | [ ]**m**, [,]**m** *etc.* |
| rowing | **char** | **string** |
| uniting | **m** | **union** (**m**,**m**1, . . . ) |
| voiding | **m** | (no mode at all) |

We have already discussed dereferencing (Section 2.6, Reference-to Modes), widening (Section 2.6), and deproceduring (Section 3.6, Procedure Calls): Widening also applies to the **long** and **short** forms of **int**, **real**, **bits**, and **bytes**.

Rowing can convert a unit into a one element array where required by the context, such as in [1:1]**int** *a* := 3. Uniting turns a unit into a **union** where required by the syntax, as in **union**(**int**, **real**) *u* := 4. Rowing and uniting happen when needed and are of little interest to the average garden variety programmer.

As is probably apparent by now, constructions called statements (for example, assignments, procedure calls) in most languages are called units in Algol 68 and can be used as sources, parameters, etc. Sometimes, however, the value of a unit is not needed. Consider what happens to the value of *i*:=*j* in the closed clause:

( **int** *i*, *j* := 3; *i* := *j*; *i*+*j* )

it is discarded after the assignment is performed. Technically this is called voiding (RR 6.7). Algol 68 "statements" are properly called **void** units. Chapter 6 of the Revised Report describes the coercions in full.

Note that there is no coercion from **real** to **int**. However, the monadic operators **round** and **entier** operate on reals and deliver integers, rounding and truncating, respectively.

## 5. CONTEXTS

Not every coercion is allowed in every context (for example, in source, destination, subscript, actual parameter). In Section 3.11 Casts, we saw that ambiguities could result if dereferencing of destinations was allowed. Each context has an intrinsic strength. The strength specifies which coercions are allowed. There are five strengths: strong, firm, meek, weak, and soft. In some contexts no coercions at all are allowed.

Strong contexts are those in which the mode of the unit is uniquely determined by the context. For example, in

```
( real x; x := big hairy mess)
```

the destination is known to be of mode **ref real**. Any and all coercions may be used (repeatedly) to turn the source into an object of mode **real**. If the source is an **int**, it can be widened; if it is a **ref real**, it can be dereferenced; if it is a **proc real**, it can be deprocedured; if it is a **ref proc ref int**, it can be dereferenced to **proc ref int**, deprocedured to **ref int**, dereferenced again to **int**, and finally widened to **real**. Some examples of strong contexts are: sources in assignments, initial values in declarations, actual parameters, and procedure bodies.

In other contexts some coercions must be prohibited to avoid ambiguities. For example, consider:

```
begin int i := 2, j;
   real x := 3.0, y;
   j := i+i;
   y := x+x
end
```

The + in *i*+*i* is an operator that operates on integers and yields an integer. The + in *x*+*x* is a different operator acting on reals. The two operators correspond to different hardware instructions. The compiler tells which operator is to be used by looking at the modes of the operands. If operands could be widened, the operands of the first + could be dereferenced and then widened, yielding reals. Then the compiler could not tell which operator was meant. Operands are always in firm context.

If every kind of unit were allowed in every context, certain ambiguities would arise, as can be easily seen by means of an example. Consider what would happen to the integer assignment *i* := *j*+2 if *j*, which is an operand of +, were replaced by the assignment *k* := 3. We would have *i* := *k* := 3+2, which is perfectly legal, but not what was intended. It adds 3+2 and then assigns the result, 5, to *k* and *i*. If we wrote the operand *k* := 3 as a closed clause (*k* := 3), we would get *i*:= (*k* := 3)+2, which first assignsi 3 to *k*, and then 5 to *i*. This is quite a different result than in the first case! To avoid ambiguities, ALGOL 68 only allows constructions in positions where no confusion can arise.

## 6. PROCEDURES

In ALGOL 68 procedures are objects and have values, just as any other objects. Procedure variables exist and may be declared, just as other variables. They may also be initialized to some value of the appropriate mode; for example, by using a routine text:

```
proc real p1 := real: 1.0/(1.0+random);
proc int p2 := int: (int k; read(k); k);
proc(int)int p3 := (int k)int:(k+1) over 2
```

Procedures (and all other modes) may also be declared in a slightly different way, by use of what is technically called an identity declaration (RR 4.4.1 a). This form consists of **proc**, the identifier, an equals sign, and a routine text. In this form the identifier is no longer a variable and cannot be assigned a new procedure. Some examples of this form are:

```
begin ¢ proc declarations ¢
    proc next = (int k) int: k+1;
    proc bump = (ref int k) void: k := k+1;
    proc less = (int j, k) bool: j<k;
    proc readin = int: (int k; read(k); k);
    proc eject = void: print(new page);
    proc dot product = (int n, [ ]real a, b) real:
        begin real sum := 0;
            for k from 1 to n do sum := sum+a[k]×b[k]
            od;
            sum
        end;
    skip ¢ dummy statement ¢
end
```

Note that the procedure body (the part following the colon) is a unit. In the dot product example, the unit is a closed clause.

## 6.1 Parameter Mechanism

A procedure may be declared with an arbitrary number of formal parameters, but calls to the procedure must supply precisely the proper number of actual parameters, no more and no less. The *n*th actual parameter is accessed by using the identifier of the *n*th formal parameter, just as in FORTRAN, PL/1, ALGOL 60, etc. Thus the order of the formal parameters is very important.

Unlike these other languages, however, the modes of the parameters are specified directly in the formal parameter list. The mode specified before the identifier of each formal parameter is the mode of that parameter. In the declaration:

```
proc recip = (real x) real: 1.0/x
```

the mode of $x$ is **real**, It is not **ref real**. By way of contrast, in the variable declaration **real** $x$, $x$ is of mode **ref real**. This difference is crucial to the understanding of the ALGOL 68 parameter mechanism.

The parameter passing works as follows. At compile time, the compiler determines which coercion path is needed. The actual parameters are first evaluated, and then coerced if run time coercion is required.† (An actual parameter is a unit, and might be a closed clause 10 pages long.) Copies of the values yielded are then passed to the procedure. This may be regarded as a generalization of the call by value used in ALGOL 60, except that in ALGOL 68 a parameter may be of any mode, including **ref** "something," in which case an address is passed.

To shed more light on the parameter mechanism, let us begin with a syntactically incorrect program:

```
begin int n := 4; ¢ incorrect program ¢
    proc wrong = (int k ) void: k := k+1;
    wrong(n);
    print(n)
end
```

† Uncorrected text of the previous 2 sentences: The actual parameters are first coerced to the modes specified by the formal parameters (if necessary). Then each actual parameter is evaluated.

The problem here is that *k* has mode **int** and not **ref int**. The destination of an assignment must be of mode **ref** "something;" the assignment *k* := *k*+1 will be flagged by the compiler as incorrect. Now let as try again.

```
begin int n := 4; ¢ correct program ¢
    proc right = (ref int k )void: k := k+1;
    right( n);
    print (n)
end
```

This program will print 5. When a formal parameter is declared **int** rather than **ref int**, the corresponding actual parameter is protected from being changed. This often helps catch bugs.

To illuminate the more subtle aspects of the parameter mechanism, consider these two programs:

```
begin int i := 0;
    proc jekyll = (int a) void:
        ( i := i+1; print(a));
    jekyll (i )
end

begin int i := 0;
    proc hyde = (ref int a) void:
        ( i := i+1; print (a));
    hyde (i )
end
```

The call *jekyll*(*i* ) is executed in the following steps. Since the formal parameter is of mode **int**, the actual parameter, *i*, is dereferenced to yield an integer. A copy of this integer value is then passed to *jekyll* (on the stack, in a register, or some other way). Then *jekyll* increments *i*. Finally, *jekyll* accesses the actual parameter passed to it and prints it. The number 0 is printed.

The call *hyde*(*i* ) is executed differently. The formal parameter in program 2 is of mode **ref int**; so *i* is not dereferenced, because it is already in the proper mode. A copy of the address of *i* is made and put on the stack, in a register, or elsewhere. After incrementing *i*, *hyde* picks up the actual parameter, the address of *i*, dereferences it, getting 1, and then prints the number 1.

An object of mode **int** is passed to *jekyll*, but an object of mode **ref int** is passed to *hyde*. In a sense, *jekyll* uses the ALGOL 60 call-by-value parameter mechanism, whereas *hyde* uses something similar to a call-by-reference mechanism. ALGOL 68 effectively gives the programmer some control over how parameters are passed via the modes of the formal parameters.

In summary, the parameter mechanism has three key features:

1) A formal parameter is written as a mode followed by an identifier. A formal parameter written as **int** *k* really has mode **int**, not mode **ref int**.
2) An actual parameter may be any unit, and any coercion may be used on it, but the result after coercion must match the mode of the formal parameter. If a formal parameter has mode **ref real**, the actual parameter must yield a real variable; the value 3.14 will not suffice. The calling, and not the called, procedure performs the coercions.
3) A copy is made of the actual parameter (after coercion). This copy is what is passed (conceptually). All references to the formal parameter use this copy. Thus the parameter is only evaluated once (as opposed to the call-by-name mechanism used in ALGOL 60, where the parameter is reevaluated on every access).

## 6.2 More About Procedures

Unlike PL/1, parameters and results in ALGOL 68 may have any mode; pointers, arrays, structures, unions, and even procedures are all allowed. As an example of using a procedure as a parameter, consider the following procedure for computing the sum: $f(1) + f(2) + f(3) + \ldots + f(n)$.

```
proc sum = (int n, proc(real)real f)real:
    begin real sum := 0;
        for i to n do sum := sum+f(i) od;
        sum
    end
```

In this example, *i* is allowed as a parameter to a **proc**(**real**)**real** because parameters are strong units and therefore can be widened. A typical call to *sum* might be *sum*(100,*cos*), which would yield *cos*(1) + *cos*(2) + ...+ *cos*(100).

In ALGOL 68 a routine text is a unit and as such can be used as an actual parameter. In the previous examples, routine texts were used to the right of the equals sign. Some examples of routine texts as actual parameters of *sum* are:

```
sum(100, (real x) real: 1/x)
    sum(50, (real x) real: sin(x))
    sum(k+1, (real x) real: random)
```

ALGOL 60 fanciers will notice that using a routine text as an actual parameter is essentially equivalent to Jensen's device [5], but a lot less sneaky.

It is sometimes useful to be able to write procedures that accept a variable number of parameters. Although strictly speaking this is not possible in ALGOL 68, something very close is possible. A procedure with one formal parameter, an array, must have one actual parameter, also an array. However, this array may have an arbitrary number of elements, provided it is of the proper mode. Remember that a one-dimensional integer array has mode [ ]**int** no matter how large it is; the bounds are not part of the mode.

When it is expected that a procedure be called with different sized arrays as parameters, there must be some way of determining the bounds of its actual parameters. Two operators are provided for this purpose: **lwb** and **upb**. If *vec* is a one-dimensional array of any mode, **lwb** *vec* and **upb** *vec* have the values of the lower and upper bounds, respectively. For a higher dimensional array, *n* **lwb** *q* and *n* **upb** *q* return the lower and upper bounds of the *n*th subscript, respectively. The lower bound of a row display is 1.

The following program declares a procedure, *outp*, that accepts an integer array as parameter and prints each of its elements on a new line. Note that calls to *outp* (and also to *read*) have two sets of parentheses. One set is needed to enclose the parameter list and one set is needed to construct the row display;

```
begin int i, j, k;
    proc outp = ([ ]int a) void:
        begin for i to upb a
            do print ((new line, a[i])) od
        end;
    read ((i, j, k));
    outp ((1, 2, 3, 4));
    outp ((i, j, 7, k, j+1, k-4));
    outp ((if i<j then 1 else k fi,
                if j>0 then 4 else 2 fi,
                10, k+6, j, -k, -6, +7, 0))
end
```

that is, (1,2,3,4) is a unit, but 1,2,3,4 is nothing. This is why *print*((*x*,*y*)) and not *print*(*x*,*y*) has been used to print two variables.

It is possible to write procedures that accept any one of a prespecified list of modes as a parameter by making the formal parameter a union. The following example is a program that accepts parameters of mode **int**, **real**, or **bool** and returns the mode as a string:

```
begin int k := 0; , union(real, bool) u := 4.0;
    proc mohd = (union(int, real, bool)a) string:
        case a in
            ( int): "int",
            ( real): "real",
            ( bool): "bool"'
        esac;
    print((mohd(k ), "␣", mohd(u)))
end
```

The output of this program consists of `int real`.

## 7. OPERATORS

The following program defines a new mode, **vector**, and a procedure, *vecadd*, to add two vectors:

```
begin int n; read(n);
    mode vector = [1:n]real;
    vector v1 , v2, v3, v4, v5;
    proc vecadd = (vector x, y) vector:
        begin vector sum;
            for j to upb x
            do sum[i] := x[i] + y[i] od;
            sum
        end;
    ¢ read in 4 vectors ¢
    read ((v1, v2, v3, v4));
    v5 := vecadd(vecadd(vecadd(v1, v2), v3), v4);
    print (v5)
end
```

The statement *v*5 := *vecadd*(*vecadd*(*vecadd* (*v*1,*v*2),*v*3),*v*4), although ghastly to look at, is quite correct. Because *vecadd*(*v*1,*v*2) is a call, and hence a unit, it may be used as an actual parameter to another call of *vecadd.*

The difficulty with the preceding expression is that although it is perfectly acceptable to the ALGOL 68 compiler, for many applications, infix operators (that is, operators placed between the operands) are much more natural than nested procedure calls. Algol 68 solves this problem by allowing programmers to define new infix operators, just as they can define new modes.

Operators are defined very much as procedures are. First comes **op**, followed by the operator symbol (which may also be a boldface word), then an equals sign, and a routine text. An operator must have either one or two parameters, no more and no fewer. Like those of a procedure, the parameters and the result of an operator may be of any mode. Let us try the vector addition program again, using an operator this time.

```
begin int n; read(n);
    mode vector = [1:n]real;
    vector v1, v2, v3, v4, v5;
    op + = (vector x, y) vector:
        begin vector sum;
            for i to upb x
            do sum[i] := x[i] + y[i] od;
            sum
        end;
    ¢ read in 4 vectors ¢
    read ((v1 , v2, v3, v4));
    v5 := v1 + v2 + v3 + v4;
    print (v5)
end
```

Just to prove that any mode can be used as an operand or as a result of an operator, we present an operator that takes an **int** and a **proc void** as operands, does something useful, and delivers nothing.

```
begin
    op × = (int n, proc void p) void:
        to a do p od; ¢ deprocedure p n times ¢
    proc eject = void: print (new page);
    proc skip = void: print (new line);
    2 × eject; ¢ skip 2 pages ¢
    3 × skip ¢ skip 3 lines ¢
end
```

## 7.1 Operator Identification

Operators have one complication which procedures lack: the same symbol can be used to represent different routine texts. This property is called `GENERIC` in PL/1. The + in 1 + 2 is a completely different + than the + in 8.711+8.72. When the ALGOL 68 compiler sees an operator symbol, it determines which operator definition to use by looking at the modes of the operands. If they have modes **m**1 and **m**2, it looks to see if an operator with that symbol and those modes has been defined. If so, it uses it. If not, it begins coercing the operands to see if they can be converted into some other modes for which an operator exists.

The process of determining which operator a symbol corresponds to is called operator identification (RR 7.2). It is one of the great achievements of the ALGOL 68 Revised Report that this entire process has been described completely in the grammar; that is, the nonterminal <program> simply does not generate any ambiguous programs. No English text is needed to describe what is and what is not permitted.

After an operator has been identified, the evaluation of its formula is the same as that for procedure calls, including the parameter mechanism. Even Jensen's device will work if you provide a routine text as an operand.

To illustrate how operator identification works, consider the following program:

```
begin
    prio ? = 1;
    op ? = (int i, j) real: i+j;
    op ? = (int i, real x) real: i−x;
    op ? = (real x, int i) real: i+x+19;
    op ? = (real x, y) real: (x<y|x|y);
    print ((2?9, 6?2.0, 3.14?8, 9.2?9.9))
end
```

This program yields: 11.0, 4.0, 30.14, 9.2. Each of the four occurrences of ? in the print procedure invokes a different routine text.

Not only can one define new operators on existing modes (for example, "?" on **int**s) and existing operators on new modes (for example, + on **vector**s) and new operators on new modes (for example, **invert** on **matrix**), but one can even redefine the existing operators on the existing modes. If you really want to redefine + on integers to mean subtract, that is your business; the compiler will not complain.

More realistically, someone writing a simulator for a two's complement computer on a one's complement computer (for example, a PDP-11 simulator running on a CDC Cyber) might be very concerned about the specific bit patterns used to represent integers, rather than just their numerical values. In particular, he might want to redefine integer arithmetic to prevent -0 from ever occurring.

Or a numerical analyst might want to redefine real arithmetic to handle rounding differently, or to print a warning message when too much significance has been lost.

## 7.2 Operator Priorities

When someone writes *print*(6+3×5) he expects to get 21, because multiplication has higher precedence (priority) than addition. In ALGOL 68 the priority of an operator symbol can be set by the programmer. For example,

```
begin
    prio + = 3, × = 2;
    print (6+3×5)
end
```

will print 45, that is, (6+3) × 5. Monadic operators all have priority 10 and cannot be changed. Dyadic operators may have priorities 1 to 9. This means that -1 ↑ 2 is +1, not -1 because it is equivalent to $(-1)^2$.

## 8. STANDARD PRELUDE

Section 10.2 of the Revised Report consists of several hundred definitions of modes, operators, procedures, and values. Collectively they are called the standard prelude. Every ALGOL 68 program is presumed to be declared within the scope of these declarations. The modes, operators, etc., declared in the standard prelude may be used in any ALGOL 68 program. In fact, that is precisely why they are there. The standard prelude is written (almost entirely) in ALGOL 68.

It can now be pointed out that the basic nucleus of ALGOL 68 (the part defined by the grammar) is really much smaller than one might expect. For example, some of the "primitive" modes are not really primitive at all, but are defined in the standard prelude; for example,

```
mode compl = struct(real re, im)
```

appears in RR 10.2.2f. Furthermore, none of the operators, trigonometric functions, or input/output procedures are part of the language proper. An implementor who was not concerned at all about compilation or execution efficiency, either in time or in space, could have nearly the whole standard prelude textually substituted in front of every ALGOL 68 program, saving himself a great deal of work.

## 8.1 Environment Enquiries

The standard prelude begins with the environment enquiries (RR 10.2.1). These enquiries allow a program to learn properties of the implementation it is running under without having to deduce them by experiment. The largest integer is called *max int*, the largest real is called *max real*, the smallest positive real is called *small real*, the number of bits in an object of mode **bits** is called *bits width*, etc. For example, here is a program to determine the largest integer in an implementation: (*print*(*max int*)).

Since each implementor has the freedom to decide how many long and short integers he wants to provide, environment enquiries are provided to allow the program to find out how many there are. These include *int lengths*, *real lengths*, *bits lengths*, and *bytes lengths* among others. The purpose of these and the other environment enquiries is to ease the task of exchanging programs between computers. For example, a program needing integers of at least 47 bits could first check the value of *max int*; finding it less than $2^{47} - 1$, it could use **long int**s instead of **int**s.

## 8.2 Standard Prelude Operators

10.2.2 of the Revised Report lists the standard modes. Section 10.2.3.0a of the Revised Report lists the priorities of all the standard operators, followed by the definitions of the standard operators. For example, the operators on Boolean operands are as follows:

**op** ∨ = (**bool** *a*, *b*) **bool**: (*a*|**true**|*b*);
**op** ∧ = (**bool** *a*, *b*) **bool**: (*a*|*b*|**false**);
**op** ¬ = (**bool** *a*) **bool**: (*a*|**false**|**true**);
**op** = = (**bool** *a*, *b*) **bool**: (*c*∧*b*)∨(¬*a*∧¬*b*);
**op** ≠ = (**bool** *a*, *b*) **bool**: ¬ (*a*=*b*);
**op abs** = (**bool** *a* ) **int**: (*a*|1|0)

From the standard prelude one can see precisely which operators are defined on which operands, and what they do. For example, to determine if **abs true** has a value of 0 or 1, a glance at the standard prelude will show that it has a value of 1. Few languages offer such precise definitions of their operators as ALGOL 68.

Subsequent sections of the standard prelude define the operators for comparison, arithmetic, string handling, etc. If one wants to see exactly what + on strings (concatenation) means, one can consult RR 10.2.3.10i. A very small number of operators are defined in English, such as - on reals. To provide a full definition one would have in fact had to define how floating point arithmetic works. This would have wreaked havoc with implementations on computers whose floating point hardware worked differently. The implementor would either have had to ignore the standard prelude, or simulate floating point operations in software.

ALGOL 68 allows mixed mode arithmetic. The formula 3.14+6 yields the real 9.14. The mechanism by which this happens can now be safely revealed: The operator + is defined for parameters of modes (**int**, **int**), (**int**, **real**), (**real**, **int**), and (**real**, **real**). Four definitions are necessary because operands are firm and because widening is forbidden in firm positions (to avoid ambiguities in operator identification).

An interesting new idea in operators is that of combining arithmetic and assignment. For example, RR 10.2.3.11d states:

**op** +:= = (**ref int** *a*, **int** *b*) **ref int**: *a* := *a*+*b*

This enables one to write: *n* + := 1 rather than *n* := *n*+1. The "plus and becomes" operator, +:=, may also ease the task of optimizing the object code, especially on computers which can add directly to memory. Similar operators exist for real numbers and the other operations; for example, −:= means "subtract and becomes."

A number of standard mathematical functions are provided in RR 10.2.3.12 including *sqrt*, *exp*, *ln*, *cos*, *arccos*, *sin*, *arcsin*, *tan*, and *arctan*. Anyone who prefers sines as operators rather than as procedure calls need only write:

**op sin** = (**real** *x*) **real**: *sin*(*x*).

If you don't care what values your functions return, you may enjoy *random* (RR 10.5.1b). And finally, *pi* is defined as a real value close to you-know-what (RR 10.2.3.12a). Standard prelude declarations may be overridden simply by supplying other declarations.

## 9. INPUT/OUTPUT

ALGOL 60 was widely criticized for not discussing such mundane matters as input/output. That is one problem from which ALGOL 68 will not suffer. An extremely powerful and flexible set of input/output procedures is defined in the standard prelude. A variety of input/output styles is provided, ranging from the lowly *print* procedure to formatted input/output on files with user control over conversion codes, error handling, and the like. The ALGOL 68 term for input/output is transput.

## 9.1 Books, Channels and Files

A book is a collection of information in the form of a three-dimensional character array (RR 10.3.1.1a).† Books are comparable to what some other languages call data sets. A book consists of a certain number of pages, each page consisting of a certain number of lines, each line consisting of a certain number of characters.

For example, line printer output may consist of many pages of 60 lines, each line having 132 characters. Each position in the output can be described by a triple (*page*, *line*, *char*). Likewise, a multifile magnetic tape can be modeled with page = file number, line = record number, and character position within a record. A book on a card reader might have only 1 page with many 80-character lines. Books may not be read or written by being subscripted; instead special procedures are provided for reading and writing. We have already seen two of these: *read* and *print*.

A channel (RR 10.3.1.2) corresponds to an input/output device type, for example, a disk, card reader, plotter, holographic store, or on-line experimental rat. A file (RR 10.3.1.3) provides the machinery to use a particular channel.

An object of mode **file** is actually a structure specifying a book, a channel, the current position on the file (page, line, char), the conversion code to use, and a number of procedures of mode **proc**(**ref file**) **bool**, as well as a few other details. A typical procedure is: *page mended*. When the program has filled up a page, *page mended* is automatically called. Programmers may supply their own versions of *page mended*: for example, eject to a new page, print a heading, and return **true**, indicating that the difficulty has been corrected. A new version of *page mended*, *p*, can only be associated with a file *f* by the call

*on page end*(*f*, *p*)

and not by directly referencing the field selector *page mended*. The other procedures handle end of file, end of line, end of format, and invalid data detected.

To access an existing book via a particular channel, declare a **file** and call the procedure *open* to associate the book and channel with it. Open has three parameters: the file, an identification string, and the channel. The identification string and channel are installation dependent. To close

† Addendum: The mode of a book is **ref flex**[ ]**flex**[ ]**flex**[ ]**char**, not [,,]**char**, as could have been inferred.

a file, call *close* with the file as parameter. To create a new book, call *create*, specifying the file and channel. Three files are declared and opened in the standard prelude (RR 10.5.1c): *stand in*, *stand out*, and *stand back*. These files correspond to the normal input and output files, and the binary scratch files. At some installations the files may be card reader, printer, and magnetic tape; at others they may be an on-line terminal, an on-line terminal and a disk. These files need (must) not be declared by the programmer.

Here is a simple program to copy 1000 lines from file1 to file2:

```
begin file in, out;
   string s;
   open(in, "file1", stand in channel);
   open(out, "file2", stand out channel);
   for i from 1 to 1000
   do get (in, (s, new line));
      put (out, (s, new line))
   od;
   close (in);
   close (out)
end
```

The procedure *open* defines a correspondence between an Algol 68 file name and a preexisting operating system file name. The procedures *get* and *put* are the analogs of *read* and *print*. In fact, *read*($x$) is declared (RR 10.5.1e) as *get* (*stand in*,$x$) and *print* is declared (RR 10.5.1d) as *put*(*stand out*,$x$). The inclusion of *new line* in the calls to *get* and *put* is needed to advance the current position to the start of the next line.

Algol 68 supports random access books as well as sequential books. Each installation must decide which channels are random access, and which are not. Typically, disks and drums will be random access, whereas card readers and paper tape punches will not be. If a book is randomly accessible via a file $f$, the procedure call *set possible*($f$) will yield **true**, if not, it will yield **false**. To set the current position of file $f$ to ($p$,$l$,$c$) call *set*($f$,$p$,$l$,$c$).

A list of some (but not all) of the file handling procedures declared in the standard prelude follows; $f$ represents a file; $p$ represents a **proc**(**ref file**)**bool**; $c$ represents a character, and $x$ represents a variable, a constant, or a row display.

| | |
|---|---|
| *get possible* (*f*) | **true** if *f* is readable |
| *put possible* (*f*) | **true** if *f* is writeable |
| *bin possible* (*f*) | **true** if binary transput ok |
| *set possible* (*f*) | **true** if *f* is random access |
| *reset possible* (*f*) | **true** if *f* is rewindable |
| *chein* (*f*) | yields *f*'s channel |
| *page number* (*f*) | yields the current page |
| *line number* (*f*) | yields the current line |
| *char number* (*f*) | yields the current char |
| *lock* (*f*) | protects *f* from further access |
| *scratch* (*f*) | detach and burn the book |
| *get* (*f*,*x*) | read *x* from file *f* |
| *put* (*f*,*x*) | write *x* to file *f* |
| *new page* (*f*) | advance to a new page |
| *new line* (*f*) | advance to a new line |
| *space* (*f*) | advance one character |
| *backspace* (*f*) | go back one character |
| *set* (*f*,*pg*,*l*,*c*) | *current pos* := (*pg*,*l*,*c*) |
| *reset* (*f*) | rewind to (1,1,1) |
| *on logical file end*(*f*,*p*) | make *p* the procedure |
| *on page end* (*f*,*p*) | to be called when the |
| *on line end* (*f*,*p*) | corresponding event |
| *on format end* (*f*,*p*) | occurs on file *f* |
| *make term* (*f*,"c") | make *c* string terminator on *f* |

It is also possible to perform transput directly to a three-dimensional character array in memory rather than to an external book (cf. `ENCODE/DECODE` in CDC 6000 FORTRAN). To make an array buffer the pseudobook of file *f*, call *associate*(*f*,*buffer*).

## 9.2 Formatless Transput

The simplest form of transput is formatless transput, of which *read*, *print*, *get*, and *put* are the most important examples. Since *get* works precisely like *read*, except on arbitrary files instead of on *stand in*, and *put* is analogous to *print*, we concentrate on *read* and *print*.

*Read* and *print* have modes that ordinary programmers cannot construct. Roughly speaking, the mode of *print* is **proc**([ ]**union**(**int**, **real**, **bool**, **char**, [ ]**int**, [,,]**int**, and everything else that can be printed, and **proc**(**file**)**void**))**void**. *Read* has a similar mode.

The procedure *get* is given in its entirety in RR 10.3.3.2a. For the beginner, the following rules will be enough to get started. The input book is regarded as a continuous stream of values separated by delimiters.

Integers, reals, and complex numbers may be signed. Reals and complex numbers may contain a decimal point and an exponent part, indicated by the letter "e". When a character variable is to be read, the next character is taken (even blank), except at the end of a line or page, when the line or page will be advanced first. Strings are delimited by end of line or by any character from a special termination string associated with the file.

When reading vectors and matrices, the order in which the elements are read is important. The question of how an array (or structure) is turned into a linear sequence of elements is called straightening (RR 10.3.2.3). In short, vectors are read from lowest element to highest element. Matrices are read in row order, beginning with the first row, then the second row, etc.

The procedures *new line*, *new page*, *space*, and *backspace* may be passed as parameters to *read*. The first three advance the current position before reading. The last one moves it backwards before reading, but not beyond the beginning of the current line. Using *backspace*, input data can be reread.

*Print* works as follows. For each mode of data there is a standard format that is used. The widths of the fields are implementation dependent, depending on *max int* and *max real*. *Print* refrains from splitting numbers across lines or pages; if the number will not fit, the line or page is advanced before printing. The procedures *new line*, *new page*, *space*, and *backspace* may be included as parameters to *print*, and both *read* and *print* expect a single parameter. If this is a row display, an extra set of parentheses is required, for example,

*print*((*new page*, "`title`", *new line*, *x*, *y*, *z*))

For people who are slightly discriminating about what their output looks like, but who are nevertheless too lazy to use formatted output, the procedures *whole*, *fixed*, and *float* may be helpful (RR 10.3.2.1). The calls

*print*(*whole*(*i*, *size*)); ¢ *e.g.* +3 ¢
*print*(*fixed*(*x*, *size*, *d*)); ¢ *e.g.* 6.02 ¢
*print*(*float*(*x*, *size*, *d*, *e*); ¢ *e.g.* 1.214*e*-07 ¢

output the integer *i* or real *x* in a field of width **abs** *size*. If *size* is positive, an explicit sign is printed; if *size* is negative, plus signs are suppressed. The integer *d* specifies the number of places to the right of the decimal point. The integer *e* specifies the number of digits in the exponent field.

## 9.3 Formatted Transput

The standard prelude declares four procedures for formatted transput: *readf*, *printf*, *getf*, and *putf*. Inasmuch as *readf* and *printf* are merely calls to *getf* and *putf* with *stand in* and *stand out*, respectively, used as files, it is not necessary to examine all four of them. For simplicity we discuss only *readf* and *printf*. Note that *readf*, *printf*, *getf*, and *putf* are the formatted analogs of *read*, *print*, *get*, and *put*.

There is a mode **format** whose values describe how values are to be layed out on the output or are expected to appear on the input. A simple format text (that is, "denotation") and its meaning on output is

```
$ p"m␣=␣"5d, "n␣=␣"5d $
```

This first advances to a new page, then prints the string m =, then the value of a variable as five digits, then the string n =, and finally another value as five more digits. We discuss the construction of format texts in a subsequent paragraph. For now, it is sufficient to say that **format** is a mode (declared in the standard prelude in RR 10.3.4.1.1a) and may be manipulated like any other mode; that is, [ ]**format**, **proc**(**int**)**format**, and **ref format** are all perfectly valid modes. Variables of mode **format** exist and may be assigned values, namely, format texts.

Associated with each file is a format that applies to that file. The format may be changed whenever a new one is needed, but a format remains in effect until explicitly changed. The four formatted input/output procedures each process their parameters sequentially. If a parameter is a unit, it is transmitted according to the format currently associated with the file. However, if the parameter is a format, it supersedes the current format and is used for transmitting units until it itself is explicitly superseded. Note that a format can remain associated with a file over a time spanning many input/ output calls. (Contrast this with FORTRAN, PL/1, and other languages which require exactly one explicit format on each input/ output call.)

The procedure *printf* expects a single parameter, roughly [ ]**union**(all transputtable modes, **format**). Here are some examples of calls to the formatted transput procedures:

```
begin real x, y, z;
    file f; open (f, "a", disk 1);
    readf($l 5d, 7d $); ¢ new format for stand in ¢
    printf($l 10x 6d $); ¢ new format for stand out ¢
    putf(f, $ p "heading", 9d $); ¢ new format for f ¢
    printf((x, y, z)); ¢ use existing format ¢
    printf(($l 9d $, x, y, z)); ¢ use this format ¢
    close (f)
end
```

A format text can be used directly in a call to one of the formatted transput procedures, as a source in an assignment to a format variable, as the result of a procedure yielding **format**, etc. Format texts are delimited by $ as we have seen. Between the dollar signs are a series of pictures, separated by commas. Each value input or output is controled by some picture, although a picture need not input or output a value; for example, it may merely eject to a new page. Pictures may be replicated as in 2(5*d* 4*x*, 7*d* 2*x*), which means 5*d* 4*x*, 7*d* 2*x*, 5*d* 4*x*, 7*d* 2*x*. Replicators need not be constants; the letter "*n*" followed by a closed clause is also acceptable (among other possibilities).

Pictures can be subdivided into literal strings, alignments, and patterns. Literal strings, such as `"x␣="` or `"page␣heading"` are output as is, or are expected to be exactly so on input. Literal strings may be repeated by putting an integer in front; for example, 7"x" is the same as "xxxxxxx".

Alignments describe changes in the current position of the book, such as "go to the next line before reading or printing." There are six alignments (RR 10.3.4.1.1f):

| code | meaning |
|---|---|
| `p` | advance to new page |
| `l` | advance to new line |
| `x` | advance one character |
| `y` | backspace one character |
| `q` | output/expect one blank |
| `k` | move to specific character position |

The alignments may also be replicated; for example, *p* 2*l* 5*q* on output means go to the next page, skip 2 lines and 5 spaces. The difference between x and q is this: on input x just skips, whereas q expects blanks; on output after backspace, x skips and q overwrites with blanks. The alignments *p*,*l*,*x*,*y*, and *k* cause the procedures *new page*, *new line*, *space*, *backspace*, and *set char number* to be called, respectively.

Patterns are used for converting values, for example, integers, reals, Booleans, or strings. They are described in detail in RR 10.3.4. The following is a rough summary of some patterns. Each pattern consists of one or more frames. A frame allows a certain class of character, for example, sign, digit, exponent symbol, or any character. A list of frames and the allowed characters in each follows:

| code | meaning |
|---|---|
| `-` | blank or minus sign |
| `+` | plus or minus sign |
| `z` | blank or digit |
| `d` | digit |
| `e` | letter e (exponent) |
| `.` | decimal point |
| `b` | Boolean (namely, 0 or 1) |
| `i` | letter i (for complex numbers) |
| `a` | any character |

Rather than attempting to give the precise rules for combining frames into patterns, we give some examples of how the integer 12345 would appear with various patterns. (The letter B is used to indicate a blank space in the output.) Note that z suppresses leading zeros.

| pattern | result of printing 12345 |
|---|---|
| `8d` | 00012345 |
| `6d` | 012345 |
| `7zd` | BBB12345 |
| `+7zd` | +BBB12345 |
| `-7zd` | BBBB12345 |
| `7z+d` | BBB+12345 |
| `7z-d` | BBBB12345 |

The following examples show how the real number 123.45 would appear with various patterns:

| pattern | result of printing 123.45 |
|---|---|
| `5d.2d` | 00123.45 |
| `4d.3d` | 0123.450 |
| `3zd.2d` | 11123.45 |
| `3z+d.2d` | 11+123.45 |
| `+d.4dezd` | +1.2345eB2 |
| `-d.4dezd` | B1.2345eB2 |

To understand how patterns work, first remove the replicators by writing out the pattern in full. For example, *+4zd.2d* means *+zzzzd.dd*. This pattern contains nine frames: one plus, four z's, one d, one point, and two more d's. A number output using this pattern will therefore occupy nine positions, The leftmost position will be a + or - sign. The next four positions will be digits, except that leading zeros will be converted to blanks. The four positions following this will be: one digit (even zero), one point, and two more digits (even 00). For example, 123.45 will be output as +BB123.45.

To allow leading signs to "float rightword" to the immediate left of the first nonzero digit, two special combinations are provided: Nz+d and Nz-d (N is some integer). This does not cause ambiguities, because putting the sign in the middie of the digits is clearly something special. The field width for Nz+d or Nz-d is N+2; for example, 6z+d means zzzzzz+d and gives an eight-position field.

A picture may consist solely of a literal string, an alignment, or a pattern, or a sequence of these. Thus $*p l 8d*$ is a format text with one picture, and $*p,l,8d*$ is an equivalent format text with three pictures.

As a final example of outputting numbers, consider this program:

```
begin int i := 2;
   print (($ p "hi" 2l, 3d, 2z+d, 3q5zd.d,
            z-d.2d $, j, -i, i+999, pi))
end
```

The output begins with "hi" on top of a new page, then a blank line, then

```
002BB-2BBBBB1001.0BB3.14
```

Characters and strings are read and written using "a" frames. Booleans are transput using "b" frames, with implementation defined characters flip and flop (as in, T and F), corresponding to **true** and **false**, respectively. Values of modes **bits** can be handled in binary, quaternary, octal, or hexadecimal, using 2r, 4r, 8r, or 16r, respectively, as illustrated in the following program:

```
begin bits n;
    ¢ print the first 100 integers in decimal, binary, octal; and hexadeci-
        mal, each in an 8 position field with 2 spaces between fields. ¢
    for j to 100
    do ¢ assign bit pattern of j to n (because only objects of mode bits can
            be output in nondecimal radices) ¢
        n := bin i;
        printf(( $l 7zd, 2q2r7zd, 2q8r7zd, 2q16r7zd $, i, n, n, n))
    od
end
```

Pictures may be replicated (as in 7*l* or 3*d*), and replicators may be **proc** **int**s. Furthermore, there is a facility that changes dynamically among several formats during transput, and a number of other sophisticated techniques.

The procedures *readbin*, *writebin*, *getbin*, and *putbin* are the analogs of *readf*, *writef*, *getf*, and *putf* for binary transput (cf. PL/1 record input/output).

## 10. SERIAL COLLATERAL, AND PARALLEL PROCESSING

In general, ALGOL 68 statements are executed one after another, in the order written. The semicolon can be regarded as a go-on operator that causes execution to continue.

The **void** units in a serial clause are executed sequentially, for example. In some situations, however, there is no inherent sequencing. For example, there is no reason for the first unit of a row display to be evaluated before the last one. Nor is there any reason for the left operand of a dyadic operator to be evaluated before the right operand. In some other programming languages operands are evaluated strictly from left to right, but nothing in classical mathematics suggests any precedent for this. Actions that have no specific ordering in time are said to be carried out collaterally.

In formulating ALGOL 68, the designers intentionally specified that the order of evaluating certain things, such as the left and right operands of a dyadic operator, be undefined. This was done to help compilers produce an optimized object code and to take advantage of multiprocessor systems.

By not fixing the order of evaluation of operands and certain other constructions, ALGOL 68 provides the compiler writer with the freedom to do the evaluations in the most efficient order. In some situations, evaluating the right operand before the left operand may be more efficient. For example, consider:

```
begin real x, y, z;
    proc f = (real x)real: (random < .5 | x | -x);
    read(x);
    y := (pi+x)/8;
    z :=f(x)+f(y);
    print (z)
end
```

On a computer with a single accumulator used for arithmetic, after evaluating $(pi{+}x)/8$, $y$ is very likely to be in the accumulator. It is also likely that better object code can be generated if $f(y)$ is evaluated before $f(x)$, because $y$ is already in the accumulator, and $x$ is not. If the ALGOL 68 specifications had required that left operands be evaluated before right operands, the compiler would have no choice but to do the call of $f(x)$ first, even though it is less efficient in that order.

The second reason for having the order of evaluation of certain constructions be explicitly undefined is that some computers have more than one processor and are capable of performing several computations in parallel. As the price of CPU's continues to fall, both in absolute terms and relative to total system cost, multi-CPU systems will become more and more common. Consider the following block, where $f$ is assumed to be a horrendously complicated function declared in an outer block:

```
begin [1:4]real x;
    int a, b, c, d;
    read((a, b, c, d));
    x := (f(a), f(b), f(c), f(d))
end
```

On a computer with four (or more) CPUs, the ALGOL 68 compiler might decide to have $f(a)$, $f(b)$, $f(c)$, and $f(d)$ all evaluated simultaneously, each on its own CPU. If the language required $f(a)$ to be evaluated before $f(b)$, this would be impossible.

Some of the constructions that are evaluated collaterally are: Source and destination in assignments; operands of a dyadic operator; elements of a row display; fields of a structure display; actual parameters in a call; **from**, **to**, and **by** parts of a **for** statement; subscripts and bounds in a slice; upper and lower bounds in an array declarations; array to be sliced and its subscripts; procedure to be called and its parameters; declarations separated by commas; and units of a collateral clause.

## 10.1 Collateral Clauses

A collateral clause is a list of units separated by commas and enclosed by **begin** and **end**, or by parentheses. The order in which the units of a collateral clause are evaluated is expressly undefined. An important kind of collateral clause is one composed of statements (technically **void** units). Whereas the statements of a serial clause are executed sequentially, the execution order of the statements in a collateral clause is explicitly undefined. An example of a **void** collateral clause is:

```
begin k := 3, x := 3.14, s := "a" end
```

Consider the following two programs; the first contains a closed clause, and the second contains a collateral clause:

```
begin ¢ program 1 ¢
    int k := 0;
    ( k:=k+1; k := k+1);
    print (k)
end
begin ¢ program 2 ¢
    int k := 0;
    ( k := k+1, k := k+1);
    print (k)
end
```

The only difference between the two programs is the use of a semicolon versus the use of a comma between the assignments. There is only one tiny spot of ink in typography, but a world of difference in meaning, as we shall see.

The first program prints 2, as you would expect; the second program requires closer scrutiny. Since the order of evaluation of the units in a collateral clause is undefined, the first one might be completed before the second one was started, giving 2 as an answer. However, on a computer with two CPUs the compiler might arrange to give each CPU one unit to process with the following sequence of actions occurring.

1) CPU 1 fetches $k$ into its accumulator;
2) CPU 2 fetches $k$ into its accumulator;
3) CPU 1 adds 1 to its accumulator;
4) CPU 2 adds 1 to its accumulator;
5) CPU 1 stores 1 into $k$;
6) CPU 2 stores 1 into $k$.

The result is that $k$ becomes 1 instead of 2. Depending upon the order of evaluation, the second program may print 1 or 2. Random numbers are very useful in computer science, but this is not a recommended technique for producing them. On the other hand,

```
begin int m := 0, n := 0;
    ( m := m+1, n := n+1);
    print((m, n))
end
```

operates correctly no matter what the order of evaluation is. The moral of this story is: Collateral clauses are an important programming technique for exploiting parallel processing, but some care is required in their use.

It should be noted, however, that race conditions of this kind are not unique to Algol 68; any language or system permitting parallel processes unrestricted access to a common data base can preduce the same peculiar effects. To be safe, one should avoid using collateral clauses which have an execution order that matters, or which modify each other's variables.

Collateral clauses may be nested, of course, allowing more complicated mixtures of collateral-serial execution to be described. For example,

```
( a; (b, (c; d), ((e, f); g)); h)
```

describes the fellowing situation (the letters are assumed to be **void** units, for example, procedure calls or closed clauses). First $a$ is executed. When $a$ is finished, three actions proceed collaterally:

1) $b$
2) $(c;d)$
3) $((e,f);\ g)$

If enough CPUs are available, $b$, $c$, $e$, and $f$ may all begin at once. When $c$ finishes, $d$ may start. When $e$ and $f$ are both finished, $g$ may start. If $e$ finishes before $f$, then $g$ must be held up until $f$ is also done. When $b$, $c$, ,$d$, $e$, $f$, and $g$ are all completed, $h$ begins.

Collateral clauses are primarily useful for allowing independent, non-communicating processes to run in parallel. For some applications however, the processes must communicate with each other. Typical examples are producer-consumer problems, where one process fills a shared buffer and the other one empties it. The two processes need to be synchronized to ensure that the producer stops when the buffer is full and that the consumer restarts the producer when it has (partially) emptied it again.

Dijkstra [4] has described a general synchronization method for parallel processing based on semaphores, and operators that increment and decrement them. An attempt to decrement a semaphore which has value 0 causes the decrementing process to be stopped. ALGOL 68 provides a mode **sema** (for semaphore) and two operators, **up** and **down**, to increment and decrement variables of mode **sema**. These are given in RR 10.2.4.

When semaphores are used in a collateral clause, the symbol **par** must appear directly before the opening **begin** or parenthesis. This is to warn the compiler. Such clauses are then called parallel clauses (RB 3.3.1c).

As a simple example of parallel processing using semaphores, consider the problem of two processes running in parallel, each of which needs exclusive access to a certain data base during part of its computation cycle. (Readers unfamiliar with this type of synchronization problem should see Brinch Hansen [2]. A semaphore, *mutex*, initialized to 1 (using the **level** operator) is used here to achieve mutual exclusion.

```
begin sema mutex := level 1;
    bool not finished := true;
    ¢ declare the data base here ¢
    proc producer = void:
        while not finished
        do down mutex;
            ¢ insert item into data base here ¢
            up mutex
        od;
    proc consumer = void:
        while not finished
        do down mutex;
            ¢ remove item from data base here ¢
            up mutex
        od;
    ¢ here is the parallel processing ¢
    par(producer, consumer)
end
```

## 11. Where To From Here?

Readers who want to continue their study of ALGOL 68 may wish to read Lindsey [8], Woodward and Bond [15], Woodward [14], Valentine [11], Branquart *et al.* [1], Cleaveland and Uzgalis [3], Peck [10], and the Revised Report, in roughly that order. For those readers who want a book length exposition, Learner and Powell [6], Peck [9], and Lindsey and van der Meulen [7] are recommended. For those who read German, van der Meulen and Kühling [12] is a good introductory text. In these references, beware of minor differences between the Revised Report, which is described in this article, and the original report, which is described in most of the references.

An even better way to learn ALGOL 68 is to write programs in this language. Compilers for various computers exist, including B6700, and ICL 1900. A fairly large subset of the language is even being implemented on a minicomputer (PDP-11).

ACKNOWLEDGMENTS

I wish to express my appreciation to the numerous people who have read and criticized this article, especially Jack Alanen, Willem Paul de Roever, Dick Grune, Ad König, Kees Koster, Efrem Mallach, John Peck, Mitchell Tanenbaum, Robert Uzgalis, Reind van de Riet, A. van Wijngaarden and P. M. Woodward.

References

[1] Branquart, P.; Lewi, J.; Sintzoff, M.; and Wodon, P. L. ”The composition of semantics in ALGOL 68,” Comm. ACM 14, 11 (Nov. 1971), 697-707.

[2] Brinch Hansen, Per. ”Concurrent programming concepts,” Computing Surveys 5, 4 (Dec. 1973), 223-245.

[3] Cleaveland, J. C.; and Uzgalis, R. C. Grammars for programming languages: What every programmer should know about grammar, American Elsevier Publ. Co., New York, 1976.

[4] Dijkstra, E. W. ”Cooperating sequential processes,” In Programming language, F. Genuys (Ed.), Academic Press, New York, 1968.

[5] Jensen, J.; and Naur, P. ”Call by name: An implementation of ALGOL 60 procedures,” BIT 1, (1961), 38.

[6] Learner, A.; and POWELL, A. J. An introduction to ALOOL 68 through problems, MacMillan, New York, 1974.

[7] Lindsey, C. H.; and van der Meulen, S. G. An informal introduction to ALGOL 68. North Holland Publ. Co., Amsterdam, The Netherlands, 1971.

[8] Lindsey, C. H. ”ALGOL 68 with fewer tears,” Computer J. 15, (1972), 176-188.

[9] Peck, J. E. L. An ALGOL 68 companion, Univ. of British Columbia, 1972.

[10] Peck, J. E. L. ”Two-level grammars in action,” in Proc. IFIP Congress 74, NorthHolland Publ. Co., Amsterdam, The Netherlands, 1974, 317-321.

[11] Valentine, S. H. ”Comparative notes on ALGOL 68 and PL/I,” Computer J. 17, (1974), 325331.

[12] van der Meulen, S. G.; and Kühling, P. Programmieren in ALGOL 68. Walter de Gruyter & Co., New York, 1974 (in German).

[13] van Wijngaarden, A.; Mailloux, B. J.; Peck, J. E. L.; Koster, C.H.A.; Sintzoff, M.; Lindsey, C. H.; Meertens, L. G. L. T.; and Fisker, R. G. ”Revised report on the Algorithmic Language Algol 68,” Acta Informatica 5, (1975), 1-236.

[14] Woodward, P. M. ”Practical experience with ALGOL 68” Software-Practice and Experience, 2, (1972), pp. 79.

[15] Woodward, P. M.; and Bond, S. G. ALGOL 68-R Users Guide, 2nd Ed. Her Majesty’s Stationery Office, London, England, 1974.

[16] Pagan, F. G., A practical Guide to Algol 68, John Wiley Inc., New York ,1976.