NAS 12-563
CR 86300

N70-15480

BASEL: THE BASE LANGUAGE

FOR AN EXTENSIBLE LANGUAGE FACILITY

CASE FILE
COPY

Massachusetts
COMPUTER ASSOCIATES, Inc.

Massachusetts
COMPUTER ASSOCIATES, Inc. / Lakeside Office Park • Wakefield, Massachusetts 01880 • Area Code 617 245-9540

BASEL: THE BASE LANGUAGE

FOR AN EXTENSIBLE LANGUAGE FACILITY

by

Alice E. Fischer
Philippe Jorrand

June 28, 1968
CA-6806-2811

Table of Contents

page	
1	INTRODUCTION and OVERVIEW
3	OBJECTS
5	EXPRESSIONS
7	Data operators
7	Mode-descriptor operators
11	Constructors
16	DECLARATIONS
19	Mode identity and mode declarations
21	Procedures
26	Declaration of variables
28	PROGRAM STRUCTURE
29	Compound Expressions
30	Tuples
30	The Scope of a Name
31	CONTROL STATEMENTS

INTRODUCTION and OVERVIEW

BASEL is a primitive Algol-like programming language which is both self-extensible and is adaptable through lexical, syntactic and other higher-level extensions.

A full discussion of the implications of this design can be found in the companion paper, "On the Basis for ELF - an Extensible Language Facility". Briefly, though, this means that BASEL should satisfy the following design goals:

As a language, it should be an adequate means of communication not only between a human and a machine, but also between humans. That is, the language should be reasonably easy to write, compile, and read. It should also be easy to learn.

As a primitive language, it should be simple and should contain a minimum of constructs.

As an extensible language, it should be free from asymmetries and special cases. Because it is self-extensible, meanings for constructs need not be pre-defined if they can be defined conveniently in terms of other constructs, and in fact, meanings should not be pre-defined if there is any dispute about what a construct should mean.

As an ALGOL-like language, BASEL should not arbitrarily introduce new key words and constructs for old meanings, but abide, where possible, by tradition.

As a programming language, it should mirror the standard hardware operations of current computing equipment.

A BASEL program is a compound expression that begins with 'begin' and ends with 'end'. (Parentheses are synonyms for 'begin ... end'.) This compound expression may contain any number of simple expressions (or statements), which in turn may have compound expressions nested within them.

A BASEL program is executed by sequentially evaluating the simple expressions which are part of the top-level compound expression. This, of course, causes any nested expressions to be evaluated, and the values of these nested expressions may be used in the containing expressions.

BASEL has four kinds of primitive program elements:

a. Names identify the objects that a program manipulates. There are two kinds of objects: mode-descriptors and data.

b. Operators manipulate the objects. These include the assignment operator, operators such as plus and less than, and procedure calls.

c. Control statements are used to specify the order in which the expressions are executed.

d. Declarations are used to define objects and operators, and give them names.

The following example should illustrate the general character of BASEL.

	Refer to note#
.| This compound expression computes the sum of the first ten elements of the row A. |.	1,2
begin	3,4
let temp be a loc int;	5
0 ⟶ temp;	6
for i from 1 by 1 to 10	7
do val temp + A.(val i) ⟶ temp;	8,9,10
val temp	11
end	12

Notes on this example:

1. We assume that this compound expression is nested within a block in which A is declared to be a row of integer values, whose length is at least 10. (A row is similar to a one-dimensional FORTRAN array.)

2. Comments are delimited by '.| ... |.'

3. BASEL uses reserved words; by design, all of these are strings of underlined letters.

4. This example consists of a compound expression, delimited by 'begin ... end'. The entire body of this compound expression is the block which begins with 'let temp' and ends with 'val temp'.

5. Declarations begin with the symbol 'let'. This declaration defines 'temp' to be the name of an integer variable, and causes space to be allocated for 'temp'.

6. ' $\longrightarrow$ ' is the assignment operator. It causes the value on the left to be stored in the variable on the right.

7. 'for i from 1 by 1 to 10 do scope ' is an iteration control statement. It is given the following interpretation:
   1. Set the variable i to 1.
   2. If the current contents of i is greater than 10 then go to the statement following the scope. Otherwise follow the instructions in the scope.
   3. When the scope is finished, increment the current contents of i by 1. Go back to step 2.

8. 'val' is an operator whose operand must be a variable. (A variable may be thought of as a box which may contain a value.) 'val' causes the contents of the variable to be fetched.

9. '.' is the subscript operator, for example 'A.1' denotes the first element of the row 'A'.

10. Since A is a row of values (not variables)then A.(val i) is one of thos e values, and can be added to the value which was stored in the variable 'temp'.

11. The value of a block is the value of the last computed simple expression in it, in this case the final value of the variable 'temp'.

12. If this block were executed when A had the value row 10 of int[1,2,3,4, 5,5,4,3,2,1,] the value of the block would be 30.

OBJECTS

An object is an entity that can be operated upon or manipulated. BASEL has two kinds of objects: data objects and mode-descriptor objects(modes).

Some objects of each kind are pre-defined; new objects are defined by declarations. A mode-descriptor describes the size, internal structure and operational characteristics of any data object to which it is attributed. The pre-defined modes are:

Data Modes:	integer value	written int
	real value	real
	boolean value	bool
	character value	char
	unspecified	free
The mode of a mode-descriptor:		mode

Certain values having these modes have pre-defined names. These are:

the non-negative integers	written 0, 1, 2, ...
the non-negative reals	0., 3.96, .5, 3.7 e -2
the boolean values	true, false
the character values	'A', ..., 'Z', '1', '2', ..., ',', '+', ...

Because these names are conventional representations of their meanings, they are sometimes called 'literals'. Through declarations, the programmer can give other names to these values. In fact, it is possible to name any value that can be computed.

Occasionally procedures are written which can accept parameters of any mode, including modes which a future user might invent. A general output routine is an example of such a procedure. The formal parameter to such a routine would be specified with a 'free' or partially 'free' mode. (the 'free' mode may only be used to describe formal parameters.) Details of the way in which 'free' objects are handled are given in the section on procedures.

A programming language must provide for storing data values. We will use the term "variable" to mean a place in which a data-value may be stored. A variable can be viewed as a box of a particular size and shape determined by the mode of the value which it may contain. Each box has a unique address

through which it can be accessed. Declaring a variable causes a box of the appropriate shape to be created, and the box's address to be associated with the declared name.

We consider addresses themselves to be values, and therefore they are legitimate data objects capable of being the contents of other boxes. We use the term "pointer" for a variable which can store the address of a variable.

EXPRESSIONS

Just as BASEL has two kinds of objects, data objects and mode-descriptor objects, so BASEL has two kinds of expressions.

1. The data expression involves data operators and data procedure calls. It either has a data object as its value or has no value at all. Its mode is a mode expression.

2. The mode expression involves mode operators, mode procedure calls and data expressions. It has a mode-descriptor as its value. Its mode is mode.

An expression is one of the following:

1. The name of a data object is a data expression. (Recall that a literal is a name.) The name of a mode-descriptor is a mode expression.

2. An operator applied to its declared number of operands each of which is an expression of the appropriate mode for that operator is an expression.

3. A procedure call, which is a procedure-valued expression followed by a tuple-valued expression representing the actual parameter list, is an expression. (A tuple is a list of values). Each actual parameter must have a value whose mode agrees with the mode of the corresponding formal parameter in the procedure declaration.

4. A constructor, which is a mode-descriptor-valued expression followed by a tuple-valued expression, is a data expression. This causes the tuple to be turned into an object of the given mode. The mode of each value in the tuple must be the same as the mode of the corresponding component as described in the mode-descriptor.

5. An 'if' statement, a 'for' statement, or a 'when' statement is a data-expression.

The sections on control statements, operators, etc. define the ways in which each type of expression is evaluated. The result of evaluating an expression is called the value of the expression.

Data Operators

The pre-defined data-operators are listed in Appendix I. These are all generic operators, that is, the particular action performed by an operator depends on the modes of its operands. Operators are only pre-defined for certain combinations of operand-modes. The programmer can extend the definitions to cover other combinations.

A few of BASEL's pre-defined operators are both unusual and important. These have already been touched upon in examples, and a list of them is given below.

symbol	name of operation
$\longrightarrow$	assignment
val	fetch a value
.	select a component
is	identifty predicate (is the same object as)

Mode Operators

There are six mode-descriptor operators. Operands for these may be any pre-defined or programmer-defined data modes or mode expressions. The result of a mode-descriptor operator is a mode-descriptor.

1. row

A row is a homogeneous ordered set of values. The size of this set can be either fixed or variable. The members of the set are numbered, and can be accessed by number. If A is a row of values, then A.3 is the third value in that set.

The mode-descriptor operator 'row' builds the description of such a set out of the description of the mode of the values in the set and an integer-valued expression specifying the size of the set. The syntax for describing a fixed length row is:

row ⟨integer-valued expression⟩ of ⟨mode expression⟩

The syntax for describing a variable length row is:

row any of ⟨mode expression⟩

The selector 'length of' is used to access the current length of a row.

Examples:

row 2 of int

This describes a set of two integer values.

row any of char

This describes a variable length character string.

2. struct

A structure is a non-homogeneous set of values. The members of this set are named and accessible by name. If 'stack' is a structure which has a component named 'top' then 'stack.top' refers to that component.

The mode-descriptor operator 'struct' builds the description of such a set out of the descriptions of the modes of the components and the names to be given to these. The syntax for this operator is:

struct (⟨list of fields⟩)

where each field specifies the mode and name of a component, and has the syntax:

⟨mode of component⟩⟨name of component⟩

The fields are separated by commas.

Example:

struct (int numerator, int denominator)

This mode expression could be used to describe a rational number.

3. tuple

The concept 'tuple' underlies the concepts 'row', 'structure', and 'actual parameter list'. A tuple is a list of values out of which a row, a structure, or a parameter list can be built.

The mode-descriptor operator 'tuple' builds the description of such a list out of the descriptions of the modes of the values in the list. The syntax for this operator is:

tuple (⟨list of the modes of the components⟩)

Examples

tuple(int, int)

This is the description of a list of two integer values. An object of this mode could be turned into a row, a structure, or an actual parameter list.

tuple (int, real)

This is the description of a list of two values whose modes are not homogeneous. An object of this mode could be turned into a structure or a parameter list, but could not be made into a row.

4. loc

A variable is a box (a location) in which a value may be stored.

The mode-descriptor operator 'loc' builds the description of a variable given the description of the value which it may store. Note that this implies that a variable may store values of only one mode; a real variable may store a real value but not an integer value.

The syntax for this operator is:

loc ⟨mode of the value which this variable may contain⟩

Example:

loc int

This describes a variable which may store an integer value.

5. proc

A procedure is a parameterized description of a value. (This value is usually specified by giving an algorithm by which to compute it.) Since procedure calls are used in expressions, it is reasonable that we should be interested in the domain and range of each procedure. This information is embodied in the procedure's mode.

The mode-descriptor operator 'proc' builds the description of the mode of a procedure out of the modes of its parameters and the mode of its result. The syntax for this operator is:

proc (⟨list of modes of the parameters⟩) ⟨mode of result⟩

If the procedure has no parameters, then its mode is written:

proc ( ) ⟨mode of result⟩

If the procedure returns no result, its mode is written:

proc (⟨modes of the parameters⟩) none

Note that 'none' is not a mode, but serves as a place holder, to make the syntax unambiguous.

Examples:

proc (real) real

This describes the mode of an ordinary trigonometric function.

proc (int) none

This could be used to describe a procedure to open the file designated by the integer code.

proc ( ) struct (int, int)

This could be used to describe a procedure which returns the current time of day expressed as two integers representing hours and hundredths of hours.

proc ( ) none

This could describe the mode of a DUMP procedure.

6. union

Occasionally it is useful to permit a variable to store values of more than one mode, or to permit an expression to produce a value whose mode depends on the data. The mode-descriptor operator 'union' is used to build the description of a mode which is not completely fixed, but can be one of a fixed finite set of modes. The mode of a variable declared to be a 'union' can vary dynamically among this fixed set of modes, depending on the mode of the value most recently stored in it. The mode of an expression can vary if that expression contains a conditional whose 'then' and 'else' clauses are expressions of different modes.

The syntax for this mode-descriptor operator is:

union (⟨set of mode expressions⟩)

Example:

union (real, int)

This expression describes a numeric value.

loc union (real, int)

This expression, then, describes a variable which can store any numeric value.

Constructors

BASEL is a language in which an unbounded number of modes can be defined, and therefore must provide a general way for writing a value of any mode. The simplest solution does not work; it is not generally possible to write the components of an object, correctly grouped, and deduce the mode of the object from the mode of its components. A date(year and day) and a rational number(numerator and denominator)may both have the structure of a pair of integers, but must be treated differently. One obvious solution is to use mode names(or mode expressions)as "constructors". That is, a value of mode X would

be written:

X [⟨ list of values of components, correctly grouped⟩]

This notation is simple, easy to remember, and has the added advantage that it reminds us that an X cannot be constructed out of inappropriate components any more than a procedure can be called with inappropriate parameters.

Example

If the mode 'complex' is defined by the mode expression

```
struct (real r, real i)
```

then the following denote complex values:

```
complex [ 1.0, 0.0]
complex [ pi + 2. , sin [ pi / 2.]]
```

Examples of Expressions:

Data Expressions

4	The value of this expression is 4.
M	Assume that M has been declared to be a data object. The value of this expression is the meaning that was given to M in its declaration. (If M is a variable its meaning is the address of the space allocated for it, not the value stored in that address.)
val J	If J has been declared to be a variable, then the value of this expression is the value that is stored in J.
2 + val J	The value of this expression is the result of adding 2 to the value which is the result of the of the expression 'val J'

2 + val J * |val K |	Conventional precedence relations are observed; val takes precedence over both + and *, and * takes precedence over +. So the value of this expression is 2 added to the result of multiplying the value stored in J by the absolute value of the number stored in K.
5 → J	This causes 5 to be stored in the variable J. The value of an assignment expression is the value of the expression on the left side of the '→'; the value of this expression is 5.
5 → J → K	' → ' is a left-associative operator, so this would be parsed as (5 → J) → K. That is, this causes 5 to be stored in the variable J, and the result of this expression, which is 5, is then stored in the variable K. The value of the entire expression is 5.
log[pi]	This is a call on the data-procedure 'log'. 'pi' is a real value, and the result of this expression a real value.
sin[pi * .5] + cos[pi]	The value of this expression is the sum of the values returned by the two procedures 'sin' and 'cos'.

Mode Expressions:

tuple (real, real, real)	This describes a triple of real values that may be used as the actual parameter list in a procedure call or constructor function call.
row 3 of real	This describes a numbered triple of real values that may be accessed individually by number (subscript). They may be manipulated by any operator or procedure that works on a row of real values.

struct(real r, real i, real c)	This describes a triple of real values that may be accessed by name. This pair may be manipulated by any operator or procedure that works on a structure of three real numbers named 'r', 'i' and 'c'.
row val I + 7 of loc real	This describes a set of real variables, the size of this set is the value of the expression 'val I + 7'. If this mode-expression is used in a declaration, the extent of the row will be evaluated at time of entry of the block in which the mode-expression is used as a declarator. If this expression is used as a constructor function (to construct a row of real variables), the extent will be evaluated before the list of expressions whose values are to be made members of the row.
row 5 of row 6 of int	This describes a 5 x 6 array of integer values.
row 3 of proc (real)real	This describes a set of 3 procedures, each of which takes a real parameter and returns a real result.
struct (real r)	A structure with one component may be used when the programmer wishes to modify the way in which basic operators behave. For instance, arithmetic on radians is sometimes done modulo 2.* pi. The mode expression here could be used to describe the size of an object expressed in radians. In defining addition on these objects the programmer would specify that the 'r' parts of two radians be added (using

real arithmetic), and the result be reduced modulo 2.*pi (using real division). The result of this would then be labelled as a radian quantity.

struct(loc int level, row 50 of loc bool elem)

This mode-expression might be used to describe a push-down-stack which can hold boolean values. The integer variable 'level' would store the index of the current top of the stack.

loc real — This describes a real variable.

loc proc (real) real — This describes a procedure variable, in which the 'sin' function might be stored.

loc loc real — This describes a variable which can hold the address of a real variable, that is, it describes a real pointer.

loc loc loc real — This describes a pointer to a real pointer.

union (atom, list) — This might be used to describe the mode of a list element.

union (real, int) — This might be used to describe a numeric value.

union (int, loc int, loc loc int)

This might be used to describe an object from which an integer value can be obtained.

row any of char — This might be used to describe a character string of any length.

loc row any of char	This would then describe a space in which any character string could be stored.
free	This can be used to describe a formal parameter to a data procedure. This procedure will then accept an actual parameter of any mode.
loc free	This can be used to describe a formal parameter to a data procedure. Any variable (that is, any object whose mode is loc something) will be accepted as an actual parameter.
row any of free	A formal parameter of this mode can be matched to any actual parameter which is a row. (A row of any length of elements of any mode whatsoever.)

DECLARATIONS

Declarations are used to define and name objects and operators.

Every declared object is given a name. This name can have one of the following forms:

a. A string of upper case letters, lower case letters, underscores and numerals, starting with a letter.

Examples

```
alpha
b2
PosT
number_one_son
```

b. An underscored string of characters.

Examples

sine
b o x
PL/I
"%&8

There are no restrictions on which type of name may be given to an object. All reserved words and pre-defined names are of form (b). In order to make it clear which objects are pre-defined, names of form (b) are not used for declared objects in this paper.

Each object or operator is completely characterized by its name, mode, and meaning.

Examples

The mode descriptor

real

is completely characterized by this name, its mode which is

mode

and its meaning, which is pre-defined and describes the amount of space occupied by a real and internal structure of that space.

The data object

2

is characterized by its name; by its mode, which is "integer value", symbolized by

int ;

and by its meaning, which is a pre-defined representation of the second positive integer.

Declarations are used to combine these primitive objects into new objects. The syntax for a value-declaration is:

let ⟨name⟩ be ⟨expression⟩;

Examples

let pi be 3.14159;

is a declaration which creates the data object whose name is

pi

whose mode is the mode of the expression '3.14159', which is

real

and whose meaning is the value of the expression '3.14159'.

let complex be struct (real r, real i)

is a declaration which creates a mode descriptor whose name is

complex

whose mode is the mode of the expression to the right of 'be', which is

mode

and whose meaning is the value of the mode expression

struct (real r, real i).

which describes the amount of space occupied by a complex object, and the internal structure of that space. Specifically, a complex object requires as much space as two reals, and those two real components are to be accessed by the names 'r' and 'i'.

Mode identity and mode declarations

As mentioned before, it is desirable to be able to define objects with the same structure but which represent different kinds of things, and must be treated differently. It is also desirable to be able to name a value, then use the name and an expression which computes that value interchangably. The question then arises, what should a mode name mean? Should it be simply a shorthand for a long description, and be interchangable with the mode expression used in declaring the name? Or should it define a class of objects, thus enabling the programmer to define different modes, even if those modes have the same structure. The following examples illustrate why both conventions are useful.

Example 1:

A programmer is dealing with vectors in 2-space. He is using both polar and cartesian coordinate representations of these, and wishes to define vector addition. Addition is, of course, done differently for these two representations.

The programmer wishes to declare the two modes as follows:

name:	polar	cart
mode:	mode	mode
meaning:	row 2 of real	row 2 of real

Then the definitions of polar and cartesian addition, in terms of pre-defined real addition would be:

```
let + mean proc (polar A, polar B)
                    (polar[ sqrt [ (A.1 * cos A.2 +
                    B.1 * cos B.2)↑ 2 + (A.1 * sin A.2 +
                    B.1 * sin B.2) ↑ 2 ], arctan [ ...]]);

let + mean proc ( cart A, cart B )
                    (cart [A.1 + B.1, A.2 + B.2 ]);
```

In order to make these two definitions useful, the mode name of the operands must be checked.

Example 2:

The programmer wishes to define mnemonic names for his new modes, but does not wish to make objects declared with these names incompatible with objects and operators declared by writing out the full mode expression. In this case, the programmer wants to use a mode name as a shorthand. This is analogous to declaring a name for a constant like 'pi'.

Since both interpretations of a mode name are useful, BASEL has two mode declaration forms. These are:

1. To declare a mode such that the mode name is to identify a separate class of objects:

   let ⟨name⟩ name ⟨mode expression⟩;

2. To declare a mode whose name is to be a shorthand for its meaning:

   let ⟨name⟩ be ⟨mode expression⟩

Examples:

```
let cart name   row 2 of real;
let polar name  row 2 of real;

let intvar be loc int;
let intpoint be loc loc int;
```

Thus it is possible to define different addition operations over 'cart' objects and 'polar' objects, but anything defined for 'loc int' automatically applies to 'intvar', and vice versa.

Procedures

A procedure is a literal, parametrically dependent value, and consists of a formal parameter list followed by a procedure body.

A mode procedure (that is, a procedure whose body consists of a mode expression) may take both data objects and mode descriptors as parameters. The result of such a procedure is a fixed mode-descriptor. Mode procedures may be declared with either the 'name' or the 'be' form of the mode declaration. The result of executing a named mode procedure is a mode-descriptor which is named by an encoding of the procedure's name and the actual parameters of the call.

Examples

```
let vector be proc ( int N ) ( row N of real );
```

This declaration defines 'vector' to be a shorthand for the mode procedure proc(int N) (row N of real). Either the procedure could be applied to an actual procedure list consisting of one integer value, as follows:

```
vector [ 5]
proc (int N) (row N of real ) [ 5]
```

The result will be the fixed mode-descriptor

```
row 5 of real
```

```
let stack name proc (int L, mode M)
                    (struct (loc int level, row L of loc M elem));
```

This declaration defines 'stack' to be a named mode procedure. Declaring an object, say S, to be a

```
stack [100, char]
```

is not the same as declaring it to be a

```
proc (int L, mode M)
            (struct (loc int level, row L of loc M elem )) [100, char
```

but causes an additional attribute to be associated with S; that is S is tagged with the information that it is a 'stack' of 100 characters. The only operators that will be applicable to S are those that have been declared for 'stacks'.

A data procedure (that is, a procedure whose body consists of a data expression) may take only data objects as parameters. (Permitting mode objects as parameters would allow a greater extent of mode-variability than we are now prepared to implement efficiently. The companion paper contains a fuller explanation of this restriction.)

The mode of a formal parameter of a data procedure may be specified to be partially or completely 'free'. (A mode expression in which 'free' occurs is said to describe a partially-free mode.) Such a procedure may be executed with any parameter whose mode "matches" the partially free mode. By "match" we mean that the two modes must be identical except that where 'free' occurs in the formal parameter's mode, any well-formed mode expression may occur in the actual parameter's mode. Within such a procedure, one may access and test the mode of the actual parameter using the selector 'mode of', the predicate 'is' and the statement 'when'.

Example:

A formal parameter P which has mode

```
row 3 of loc free
```

can be bound to an actual parameter of any of an infinite number of modes, including

```
row 3 of loc real
row 3 of loc struct (int A, int B)
row 3 of loc proc (int) real
```

The body of this procedure might contain the test:

```
when mode of P is row 3 of loc real
   then printreal val (P.1)
   else ...
```

A procedure, like any other object, is completely characterized by its name, mode and meaning.

Examples

The declaration

```
let mean be proc ( real A, real B ) ( (A+B) /2.);
```

defines a data procedure named

mean

Its mode is

```
proc ( real, real ) real
```

and its meaning is a piece of code which computes the arithmetic mean function (average) when executed with two real values as parameters.

The declaration

```
let matrix be proc ( int M, int N ) (row M of row N of real ) ;
```

defines a mode-descriptor procedure named

matrix .

Its mode is

```
proc ( int, int ) mode
```

and its meaning is a representation of a parameterized mode-descriptor which produces the descriptor of an M x N matrix when applied to a pair of integer parameters.

A generic procedure is a family of procedures all having the same name. A particular member of this family is chosen as the meaning of the procedure name in a given context by matching the modes of the actual parameters, given in that context, to the modes of the formal parameters in one of the generic procedure's members.

Example

A programmer might define the generic procedure

log

to have two members, one taking a real parameter, which might be written as:

proc (real x) (⟨program to compute the log of a real value⟩)

the other taking a complex parameter, which might be written as:

proc (complex x) (⟨program to compute the log of a complex value⟩)

Then in the context

log [3.7]

the procedure

proc (real x) (...)

is taken as the meaning of "log", while in the context

log [complex [1.1, 3.7]]

the procedure

proc (complex x) (...)

is taken as the meaning of "log".

The ability to define new mode-descriptors makes it possible for the programmer to combine the pre-defined mode-descriptors into patterns that better reflect his concept of his data. Having done this he would then declare operators to work over these new forms, such as '+' over a pair of complex numbers or 'union' over two sets, etc. An operator is a generic procedure for which a special syntactic form has been declared.

Example

'+' is an operator whose meaning consists of the two procedures defining addition over a pair of real values, and over a pair of integer values. These two procedures can be represented by:

proc (real A, real B) (⟨basic machine-dependent definition of floating point addition⟩)

proc (int A, int B) (⟨basic machine-dependent definition of integer addition⟩)

The following declaration of the syntax for '+' is built into the processor:

let + be infixL prec < *;

This indicates that a call on one of the members of '+' is not to be written in the form of a procedure call, but rather, the operator's name is to be written between its two operands.
The 'L' of 'infixL' indicates that a series of additions are to be done from left to right, and the expression "prec < *" gives the operator '+' a lower precedence than the previously declared operator '*'.

New operators may be declared.

Example

let sin be prefix prec = val;

declares sin to be a new operator which will be called by writing its name in front of its operand. Its precedence is the same as the precedence of 'val'. Procedures are attributed to "sin" by later statements.

New procedure bodies may be declared to belong to an already existing operator.

Example:

let + mean proc (real A, complex B) (complex [A+B.r, B.i]);

This declaration defines '+' between a real and a complex. The result is a complex value.

Declaration of Variables

A variable, also, is completely characterized by its name, its mode and its meaning. The mode of a variable is the loc of the mode of the value which may be stored in it. The meaning of a variable is the address of the box allocated for it.

Examples

If X is a real variable, then the mode of X is

loc real

and the meaning of X is the address of the particular box allocated for X when X was declared. 'X' used in an expression stands for the address of this box.

If TRIGFUN is a proc (real)real variable, then the mode of TRIGFUN is

loc proc (real) real

and the meaning of TRIGFUN is the address of the box allocated for it. TRIGFUN may be used as follows:

sin → TRIGFUN	This assigns a value to the variable.
(val TRIGFUN) [pi] + .5	'val' applied to this variable produces a function which takes one real parameter. This function may then be applied to the actual parameter list '[pi]', and the result will be a real value, which may be added to the real value '.5'.

When a programmer declares a variable he does not usually care what address is made the meaning of the variable. Rather, he cares that this is the address of a space of the proper size and shape, and that this space has

not previously been allocated for another variable. The syntax for a variable declaration reflects this:

let ⟨name⟩ be a ⟨mode-expression⟩;

The word 'an' is an acceptable variant of 'a'.

Such a declaration causes a space of the right size to be allocated, and makes the address of that space the meaning of the declared name.

Examples

let intvar be loc int;	'intvar' is a new mode-descriptor which describes an integer variable.
let INDEX be a loc int;	'INDEX' is an integer variable.
let J be an intvar;	'J' is also an integer variable
let intpoint be loc loc int;	'intpoint' is a new mode, which describes a pointer to an integer variable.
let P1 be a loc loc int;	'P1' is a pointer to an integer variable.
let P2 be an intpoint;	'P2' is also an integer pointer.
let x be A (val J);	'x' is another name (a synonym) for the j'th element of the row A, which we assume has been previously declared.

These variables may be used as follows:

6 → J	The value of the expression '6' is stored in J.
J → P2	The value of the expression 'J' is stored in P2, that is, P2 is set to point at J.
val INDEX → J	J is set to the value that INDEX holds.
val P2 → P1	P1 is set to point at what P2 points at.
6 + val J → J	The value of J is incremented by 6.

PROGRAM STRUCTURE

The block is the basic unit of program structure. Blocks are made out of declarations and expressions, and in turn are used to make compound expressions and tuples.

Formally, a block is a series of zero or more declarations, each terminated by a semicolon, followed by a series of zero or more expressions, terminated by semicolons or 'exit's. A block (and the last expression in it) is terminated by a comma, a compound expression delimiter or a tuple delimiter. The scope of definition of a label or identifier is the innermost block that contains the declaration of that label or identifier.

During execution, a block is always entered at its beginning. The expressions within a block are executed sequentially, except of course, go to commands are obeyed. Control leaves a block after the physically last expression in it has been evaluated, or when an 'exit' is encountered. An 'exit' is like an instruction to go to an imaginary label at the end of the block.

The value, if any, of the last-evaluated expression in a block is taken as the value of the block. If control was terminated by an 'exit', the last-evaluated expression is the one immediately preceeding the 'exit', otherwise it is the physically last expression in the block.

Several 'exit's may be written in the same block, with the restriction that if any of the expressions preceeding an 'exit' has a value, then all of them must.

The mode of a block is the union of the modes of all the expressions preceeding 'exit's, and the last expression in the block. If the modes of all these expressions are the same, then that is the mode of the block.

Blocks may be used in two contexts,

1. As elements of a compound expression.
2. As elements of a tuple.

Compound expressions

A compound expression is a list of one or more blocks, separated by ','s. This list is bracketed by either 'begin . . . end' or by '(...)'. At most one of the blocks in a compound expression may have a value, and this value is taken as the value of the compound expression. The mode of this value is taken as the mode of the compound expression.

A compound expression may be used in place of any simple expression, and can serve one of two functions:

1. To group a series of actions together.
2. To indicate that a value is to be computed, then "held in hand" while some actions are performed, then control is to return to a higher level where that value may be used.

Example

The following piece of code is written three times; first with a box around each of the blocks, then with a box around each compound expression, third with a box around the tuple. The code is a definition of 'popping' a push-down stack named S.

```
begin
    if S.level = 0
    then error [ S ]
    else (val S.elem. (val level),
          val S.level - 1 ⟶ S.level;)
end
```

```
begin
    if S.level = 0
    then error [S]
    else ( val S.elem. (val level) ,
           val S.level - 1 ⟶ S.level; )
end
```

```
begin

   if S.level = 0
   then error[S]
   else ( val S.elem. ( val level ) ,
              val S.level - 1 --> S.level; )

end
```

A tuple is a series of blocks bracketed by "[...]". The value of a tuple is the series of values of its valued-blocks. The mode of a tuple is:

tuple (⟨list of modes of the valued-blocks in the tuple⟩). Tuples are used as actual parameter lists in procedure calls and constructor function calls. Building upon this basic use, the programmer may define operations over tuples.

The Scope of a Name

The scope of any declared name or label is the block in which it was declared. Declared names are also "known" in blocks which are properly contained within the block in which they were declared.

CONTROL STATEMENTS

1. if ⟨boolean-valued expression⟩ then ⟨expression⟩ else ⟨expression⟩

   a. The expression after 'if' must have boolean value. This expression is evaluated and the result is tested. The expression following either the 'then' or the 'else' is then evaluated, according to whether the 'if' clause resulted in 'true' or 'false'. The value (if any) of this expression is the value of the 'if' statement.

   b. If the expressions in the two clauses have the same mode, then this is the mode of the entire 'if' statement. If they have different modes, the mode of the 'if' statement is union (⟨mode of the 'then' clause⟩, ⟨mode of the 'else' clause⟩).

   c. If either clause has a value, then both must. This restriction allows the compiler to insure that the expression which contains the conditional is well formed.

2. when ⟨mode-test⟩ then ⟨expression⟩ else ⟨expression⟩

   a. This statement is a combination of an 'if' statement and a declaration. It is used as a conditional to test the current status of the mode of an object whose mode can vary dynamically. The expression in the 'when' clause is a conjunction each of whose terms has one of two forms:

      1. The form

         mode of ⟨data object name⟩ is ⟨mode expression⟩

         is used to test whether the mode of a data object is the same as a known mode.

         Example:

         mode of pi is real

         This mode-test yields the value true.

2. The form

mode of ⟨data object name⟩⟨structure-test operator⟩ is used to dissect a mode expression by testing the top-level mode operator in that expression. The structure-test operators are:

isloc
isrow
isstruct
istuple
isunion
isproc

Example:

mode of X isloc

The 'then' clause is executed if this condition is 'true', and within the scope of the 'then' clause, the data object is treated as if its mode actually were the given mode-expression, rather than a union or a free.

b. This statement allows the programmer to tell the compiler that within the scope of the 'then' clause it may compile acceptable code to handle an object with a 'union' mode.

c. Parts (b) and (c) of the explanation of the 'if' statement apply also to the 'when' statement.

3. go to ⟨label⟩

a. Control will pass to the statement that bears the given label.

4. The iteration statement

This statement has several optional parts, listed below in the order in which they may appear.

intentional_blank_page.tif

a. The 'for' section is optional. It consists of four clauses:

1. for ⟨name of local variable (this functions as a local declaration)⟩

   or

   for nonlocal ⟨expression whose value is the address of a variable which was declared previously⟩

2. from ⟨expression⟩

3. by ⟨expression⟩

4. to ⟨expression⟩

When the iteration statement is entered, the value of the 'from' expression is stored in the 'for' variable. Before each execution of controlled clause, the current value of the 'for' variable is tested against the current value of the 'to' expression. If the 'for' value is greater, iteration is terminated. After each execution of the controlled clause, the 'for' variable is incremented by the current value of the 'by' expression.

b. The 'while' section of the iteration statement is optional. It has the syntax:

   while ⟨boolean-valued expression⟩

This expression is evaluated before each execution of the controlled clause. Iteration is terminated the first time the resulting boolean value is 'false'.

c. The 'such that' part of the iteration statement is optional. It consists of just one clause which has the syntax:

   such that ⟨boolean-valued expression⟩

This expression is evaluated after evaluation of the increment and 'while' parts of the iteration statement, and before each evaluation of the controlled expression. If the value of the 'such that' expression is 'false', the controlled expression is not evaluated, and control passes directly back to the increment-test part of the iteration statement.

d. The controlled clause is designated by 'do'. The syntax is:

   do ⟨expression⟩

This expression is evaluated only when the 'to', 'while', and 'such that' tests all succeed. The controlled expression may or may not have a value. The value of the iteration statement is the series of values computed by the iterated execution of the controlled expression. Note that this series is not a tuple, but may be made into a tuple by enclosing the iteration statement in tuple brackets.

Examples of iteration statements:

This statement computes the sum of the even numbered elements of the row A. This sum is stored in temp.

```
for non local J from 2 by 2 to 20
  do val temp + A.(val J) → temp,
```

This statement selects out the non-zero elements of the row 'array'.

```
for index from 1 to val indexmax
  such that array. (val index) ≠ 0
  keep array. (val index),
```

This statement tests whether X = one of the elements of the row Y.

```
(for index from 1 by 1 to rowlength
  do if X = Y. (val index)
     then go to L
     else;
  false exit
L: true)
```

APPENDIX I : Data Operators

This table lists the pre-defined data operators and the combinations of operand modes for which each operator is defined.

(Let $\mathcal{M}$ and $\mathcal{M}'$ stand for any mode).

		mode or description of operands		
symbol	operation represented	left	right	mode of result
→	Store the value on the left into the variable on the right. The result is the left hand operand.	$\mathcal{M}$	loc $\mathcal{M}$	$\mathcal{M}$
val	Fetch the value of the operand, which is a variable.		loc $\mathcal{M}$	$\mathcal{M}$
.	Select a component of a structure.	struct (...)	The name of one of that structure's components.	The mode of that component.
	Select one of the elements of a row.	row N of $\mathcal{M}$	An int >0 and ≤ N.	$\mathcal{M}$
	Select one of the bodies of a generic procedure.	proc ...	A list in parentheses of the modes of the formal parameters of one of that procedure's bodies.	The mode of that body.
aloc or ref	Causes a variable to be allocated to hold the value of the operand. Space is allocated in permanent storage, and will be deallocated during garbage collection or when the program is terminated. The address of this space is the result of the operator.		$\mathcal{M}$	loc $\mathcal{M}$

mode of	Returns the current mode of its operand.		m	mode
length of	Returns the current length of its operand.		row ...	int
is	Identity predicate.	m	m'	bool
¬	Logical negation.		bool	bool
&	Logical conjunction	bool	bool	bool
v	Logical disjunction	bool	bool	bool
=	Equality predicate	int	int	bool
≠	Is not equal to.	int	int	bool
>	Is greater than.	int	int	bool
≥	Is greater than or equal to.	int	int	bool
<	Is less than.	int	int	bool
≤	Is less than or equal to.	int	int	bool
ispos	True if operand is numerically greater than zero. False otherwise.	real		bool
isneg	True if operand is numerically less than zero. False otherwise.	real		bool
iszero	True if operand is numerically equal to zero. False otherwise.	real		bool
+	Addition.	real	real	real
-	Subtraction.			
*	Multiplication.	int	int	int
quorem	The result of A quorem B is a 2-tuple, [Q, R] such that Q * B + R = A. The sign of R will be the same as the sign of A.	int	int	tuple(int, int)

/	Division, the result of A/B being the same as the first member of the tuple which is the result of A quorem B.	int	int	int
	Floating point division	real	real	real
mod	The result of A mod B is the smallest non-negative integer congruent to A modulo B. This is the same as the remainder of A/B, the second member of the tuple resulting from A quorem B.	int	int	int
↑	Exponentiation, by means of repeated multiplication.	int	int	int
		real	int	real
abs |...|	Absolute value.		int	int
			real	real

Mode-conversion Operators

trunc — Converts a real to an int, truncating the fractional part.

Examples:

```
trunc -2.7        results in -2
trunc 3.68        results in  3
```

floor — Converts a real to an int, rounding to the next smaller integer.

Examples:

```
trunc -2.7        results in -3
trunc  3.68       results in  3
```

ceiling — Converts a real to an int, rounding to the next larger integer.

Examples:

```
ceiling -2.7      results in -2
ceiling 3.68      results in  4
```

abs — Converts a char to an int, which is the index of that character in the standard alphabet.

float — Converts an int to a real.

Converts a tuple (int, int) to a real. The tuple $[\alpha,\beta]$ is converted to the real number α e β .

Example:

float [356,-3] results in 356 e-3

Conclusion:

We believe that we have arrived at a useable and practical language in spite of our insistence that everything in the language be unambiguous. The examples in Appendix III of this report are written in basic BASEL, that is, they do not use any extensions except those which are defined within each example.

We hope that you find the examples readable, and agree with us that certain things are represented here in a transparent way, that would be far more obscure in a language like FORTRAN.