The Programming Language Slim

B H Venter

Department of Computer Science, University of Fort Hare
Private Bag X1314, Alice, Republic of Ciskei, Southern Africa
e-mail: bhv@cs.ufh.ac.za

Abstract

This is the *provisional* defining report of the programming language Slim.

Key words: programming language design, very high-level languages, sets, lists, maps, persistent data

Computing Reviews Category: D.3 PROGRAMMING LANGUAGES

1 Introduction

Slim is a general-purpose programming language based on **s**ets, **li**sts and functional **m**appings between sets. The language has great expressive power and can be used to advantage in all sorts of large programming projects. Furthermore, "toy" programs are very simple to write in Slim, which makes Slim suitable as a language for novices.

The aim of this article is to give a *short* (but complete) definition of Slim suitable for implementors and experienced Slim programmers.

2 Program Structure

A Slim module is one and the same as the file used to store the source text of the module. It is assumed that there is a Slim development environment that allows modules to be created and maintained, and that allows files (modules) to be named with Slim identifiers.

A Slim module is a sequence of statements, procedures and functions:

module = {statement | procedure | function}
statement =
(export | import | transaction | selection | iteration |
conditional-statement | variable-definition | constant-definition |
function-definition | constraint-definition) ';'

If the end of a statement coincides with the end of a source line, the semicolon is optional.

! A comment is introduced and terminated by a !
! or terminated by the end of a source line
-- comments starting with -- always go to the end of source line

A subset of the procedures, functions, constants and variables defined by a module can be made accessible to other modules by exporting their identifiers:

export = **export** identifiers
identifiers = identifier {[','] identifier}

A module makes *all* the identifiers exported by another module visible to itself, simply by naming the other *module* in an **import** statement:

import = **import** identifiers

An imported identifier can be distinguished from local identifiers or other imported identifiers, when necessary, by prepending the name of the exporting module, separating it with a period from the identifier.

Any module can be executed as a program, using an environmental command. Furthermore, when a program executes an **import** statement, all the modules listed in the statement are executed as "procedures". However, a given module is executed at most once. Executing a module upon first importation allows the module to initialize its private data structures before any attempt is made to use the module.

3 Database Modules

Modules with the keyword **database**[1] prepended to them are *not* executed when first imported by a program. These modules are executed only once in their lifetimes, by environmental command. Once set executing, a database module stays in execution until the module is deleted from the system.

Since database modules stay in execution, the variables defined by these modules are persistent (they retain their values between executions of programs that access them). Moreover, concurrent executions of programs that import the same database module, all access the same set of variables.

Database variables can only be accessed by statements nested inside a transaction statement:

transaction = **transaction** {statement} **end**

Transactions are atomic (all or nothing) and serializable (apparently mutually exclusive). A transaction commits (completes successfully) when it reaches the **end** of the transaction, or when a **commit** instruction is executed. A transaction fails (aborts) if a **fail** instruction is executed, or if it turns out to be impossible to commit the transaction.

After a **commit** or **fail** instruction is executed, the next instruction to be executed is the one following the **end** of the transaction concerned.

A failed transaction has no effect on any database variables — any changes that were made are undone. All other state changes (updates to local variables, input/output, etc.) are preserved. The built-in boolean function, Committed, may be used to determine if the previously executed transaction has committed or failed.

4 Types, Variables and Constants

A type is simply a collection (set or list) of values. The predefined set, Entity, represents the set of all possible values.[2] Boolean, Number and String are predefined non-overlapping subsets of Entity, and represent the set {False, True}, the set of all numbers, and the set of all character strings, respectively. Elements of Number and String can be denoted using the notation for numbers and strings defined further on. False and True are predefined constants.

The predefined set, Character, is the set of all strings of length one. Likewise, Integer is the predefined subset {min(Integer)..max(Integer)} of Number. Undefined is a predefined constant representing a value which is used when no other value is sensible. For example, $1/0 =$ Undefined.

A variable is associated with a type by a variable-definition statement:

variable-definition = identifiers ':' expression ['←' expression]

[1] Not yet implemented.

[2] By definition, Entity $\subset$ Entity and Entity $\in$ Entity

The expression following the colon must result in a collection of values. The collection represents the type of the variables being defined. For example, given that i : {1}, it follows that *i* can only have the value 1 or that its value is Undefined. Type expressions may not include function calls.

Constant definitions are used to provide names for values. The following syntax is used:

constant-definition = identifiers '=' expression

The expressions providing the values of constants, or the initial values of variables, are executed as if the definitions are assignment statements. Multiple identifier constant definitions and multiple identifier variable definitions with explicit initialization are treated in the same way as the assignment of an expression to multiple variables. The latter is explained later on.

"Forward" references to yet to be executed definitions[3], will yield Undefined. Variables defined inside loops will be repeatedly initialized, including those that are not explicitly initialized. Variables that are not explicitly initialized are implicitly initialized to Undefined. Since constants may not be redefined, it follows that their definitions may not appear inside loops.

5 Sets

Slim provides various notations for sets. First of all, the expression:

set of X

is a way of writing "the set of all subsets of the set X". The expression can also be used as a type constructor. For example, the variable Num-set defined as follows:

Num-set : **set of** Number

is restricted to holding a *set* of numbers as its current value.

Particular values of type **set** can be denoted in the following styles:

{1, 2, 3, 7, 6, 5, 4}
{1..3, 7..4 **by** −1}
{**all** x ∈ Integer | 1 ≤ x ≤ 3 **or** 7 ≥ x ≥ 4}[4]
{**all** x+1 **for** x ∈ Integer | 0 ≤ x ≤ 6}

using the following syntax rule:

set =
  '{' {expression [('..'|**to**) expression [**by** expression]] [',']} '}' |
  '{' **all** [expression **for**] iterator '}'
iterator =
  identifiers '∈' generator {',' identifiers '∈' generator} [('|'|**where**) expression]
generator = expression | identifier ∈ generator

The notation using '..' is valid only if both the expressions result in integers, or if both result in characters. The expression following **by** must result in an integer. If omitted, the increment is 1. Note that {7..4} = {} ≠ {7..4 **by** −1 }.

[3] Procedures and procedural functions may be called before execution reaches their definitions.

[4] The current implementation does not support iterators over sets with more than max(Integer) elements. Such expressions, of which this example is one, always yield the empty set.

6 Maps and Lists

The expression:

maps from $\mathrm{S_1, S_2, \ldots, S_{n-1}}$ **to** $\mathrm{S_n}$

denotes the set of all the subsets of the Cartesian product $S_1 \times S_2 \times \ldots \times S_n$ that represent *functional* mappings from $S_1 \times S_2 \times \ldots \times S_{n-1}$ to S_n.

The expression:

lists of X

is a special short-hand form for the following:

maps from {1..max(Integer)} **to** X

Values of type **list** may be denoted using a syntax similar to values of type **set**:

list =
  '[' {expression [('..'|**to**) expression [**by** expression]] [',']} ']' |
  '[' **all** [expression **for**] iterator ']'

The list

L = ['a', 'a' .. 'c']

represents the mapping:

L(1) = 'a', L(2) = 'a', L(3) = 'b', L(4) = 'c',
L(5) = Undefined, L(6) = Undefined, ..., L(∞) = Undefined

The predefined type "String" can be regarded as shorthand for "**lists of** Character". Elements of type String can be denoted using the following syntax:

string = '‘' {character} '’'

Problem characters can be included into strings by preceding them with back slashes. Control characters are problem characters which can be denoted either by the numbers representing their ASCII encodings, or by using the standard ASCII mnemonics. Thus:

'\'\'\\\13\lf'

is a string containing the five characters:

’ ‘ \ cr lf

7 Expressions

expression =
  ['−'] expr {operator expr} |
  expression **forsome** iterator |
  expression **forall** iterator
expr =
  name | number | string | set | list | '('expression ')' |
  if expression **then** expression **else** expression |
  (**set**|**sets**) **of** expression | (**list**|**lists**) **of** expression |
  (**maps** | **map**) [**from**] expression {expression} **to** expression

operator =
  exponentiation ? $\times$ / + $-$ = $\neq$ > $\geq$ < $\leq$ $\subseteq$ $\not\subseteq$ $\in$ $\notin$ $\cap$ **not** **and** **or**
name = ([identifier '.'] identifier | **result**){'('{expression}')'}
number = digits['.'digits]['$\times$10' $^{[\text{'}-\text{'}]\text{digits}}$]
digits = '0' |...| '9' {'0' |...| '9'}
identifier = ('A' |...| 'Z' | 'a' |...| 'z') {['-'] letter}
letter = 'A' |...| 'Z' | 'a' |...| 'z' | '0' |...| '9'

An expression of the form:

expr operator expr operator ... operator expr

is evaluated in order of operator priority, scanning from left to right. The operator priorities are as follows:

highest	?
	exponentiation
	$\times$ / $\cap$
	+$-$
	= $\neq$ > $\geq$ < $\leq$ $\subseteq$ $\not\subseteq$ $\in$ $\notin$
	not
	and
lowest	**or**

Before evaluation, an expression is first scanned from left to right "rewriting" sequences of the form:

expr1 comparison-op expr2 comparison-op expr3

with the form:

(expr1 comparison-op expr2 **and** expr2 comparison-op expr3)

exponentiation $\times$ /
: are restricted to numeric operands. Division results in Undefined if the divisor is zero. The exponentiation operator is represented by superscription. For example, x^5 is an expression of the form expr operator expr, where the operator is "exponentiation".

Expr1 + Expr2
: adds up two numbers, or concatenates two strings or two lists such that:

'abc' + 'def' = 'abcdef'
[1, 2, 3] + [4, 5, 6] = [1, 2, 3, 4, 5, 6]

When applied to a pair of sets, + results in the set union.

Expr1 $-$ Expr2
: subtracts one number from another. When applied to a pair of sets, it results in the set difference.

= $\neq$ > $\geq$ < $\leq$
: compare either two numbers or two character strings with each other. In the case of strings, the shorter string of the two is padded with trailing null characters until it is as long as the other string. The strings are then compared, character by character, until two characters are not equal, or all characters have been compared. The result of the last character comparison is the result of the string comparison. Characters are compared according to their ASCII encodings. The operators = and $\neq$ can also be used to compare two arbitrary values with each other.

Expr1 $\subseteq$ Expr2
: tests whether Expr1 is a subset of Expr2. Both expressions must result in set values. $\not\subseteq$ does the opposite.

Expr1 $\in$ Expr2
: tests if Expr1 is an element of collection Expr2. $\notin$ does the opposite.

Expr1 $\cap$ Expr2
: results in the intersection of *sets* Expr1 and Expr2.

not and or
: operate on boolean values. The latter two evaluate their second operands only if needed.

Expr1 **forsome** Id1 $\in$ Expr2
: tests whether Expr1 can be true when evaluated with Id1 bound to *some* element in the list or set of values resulting from evaluating Expr2. In other words, the expression is true unless Expr1 is false for *all* the values that may be bound to Id1.

Expr1 **forall** Id1 $\in$ Expr2 | Expr3
: tests whether Expr1 will be true when evaluated with Id1 bound to *any* element in the list or set of values resulting from evaluating Expr2, for which it is also true that Expr3 evaluates to true. In other words, Expr1 must be true for *all* the values, satisfying Expr3, that may be bound to Id1.

if Expr1 **then** Expr2 **else** Expr3
: results in Expr2 if Expr1 is true, and in Expr3 otherwise. Expr2 and Expr3 are evaluated only if selected.

Expr1 ? Expr2
: evaluates Expr1 and returns the result if it is not Undefined. Otherwise evaluates and returns the value of Expr2.

8 Assignment statements

assignment = name {name} ('$\leftarrow$'|'$\overset{+}{\leftarrow}$'|'$\overset{-}{\leftarrow}$'|'$\overset{\times}{\leftarrow}$'|'$\overset{/}{\leftarrow}$') expression

An assignment statement changes the value to which the name on the left-hand side of the $\leftarrow$ refers, from whatever it is currently, to the value of the expression — provided that the new value is a member of the set of values with which the variable has been typed (if not, the target variable is made to refer to the value Undefined).

When more than one name appears on the left-hand side of an assignment statement, the expression on the right-hand side must result in a list of values. The first of these values are then assigned to the left-most target variable, and so on. If there are fewer values than variables, the remaining variables are set to Undefined. If there are more values than variables, the remaining values are ignored.

The forms:

$$\text{A} \overset{+}{\leftarrow} \text{E},\ \ \text{A} \overset{-}{\leftarrow} \text{E},\ \ \text{A} \overset{\times}{\leftarrow} \text{E},\ \ \text{and A} \overset{/}{\leftarrow} \text{E}$$

are shorthand for

$$\text{A} \leftarrow \text{A} + \text{E},\ \ \text{A} \leftarrow \text{A} - \text{E},\ \ \text{A} \leftarrow \text{A} \times \text{E},\ \ \text{and A} \leftarrow \text{A} \ / \ \text{E}$$

When the target variable is of type **map**, the value of the variable can be changed incrementally. For example:

```
       x : list of Number
!a!    x ← [1, 2, 3..300]        !non-incremental change
!b!    x(2) ← 5                  !incremental change
       !now x = [1, 5, 3..300]
```

In other words, after statement !a! the expression x(2) will result in 2, and after the definition of x is modified in statement !b!, the expression x(2) will result in 5.

Assignments to undefined variables implicitly associate them with the type Entity.

9 Input/Output

For the convenience of writers of simple programs, a simple I/O mechanism for text is built into Slim. More sophisticated I/O must be done with the aid of library modules. Every Slim program using the built-in I/O mechanism is assumed to have access to a standard input source and a standard output sink — usually the keyboard and a screen window, respectively.

Textual input items are read in by means of the **get** statement:

get = **get** name [':' expression] {name [':' expression]}

An input item is formed by skipping characters until a non-separator character (not a blank, tab, comma, semi-colon or "end of line") is found. If this character is a left curly bracket, left square bracket, or open-quote, the subsequent characters are parsed into a set, list or string value, according to Slim syntax rules. If not, all characters up to the next separator character are read in as an input item. If this item can be parsed as a number and the number is a member of the type of the corresponding variable, this is done. Otherwise the item is treated as a string. The resulting set, list, string or number is then assigned to the corresponding variable, subject to normal type checking. Variables can be defined as part of the **get** statement, by using the optional ':' syntax.

Note that sets and lists can only be read in if they are specified by enumeration and do not contain expressions or variables.

Textual output is performed by means of the **put** statement:

put = **put** {[','][' ']expression} ['&']

Output expressions must yield strings, or values which have denotations acceptable to **get**. Where there is a choice of denotations (such as 5, 5.0000, 0.5×10^1, and so on) the *shortest* denotation is used. The maximum length of a numeric output item depends on the implementation, but should be no fewer than 8 digits.

If an expression appearing in a **put** statement is preceded by separators (' ' or ',') in the program source, the corresponding output item is similarly separated from the item preceding it on the output line (if any).

Normally, a **put** appends a newline character sequence to its last item, so that the next **put** will start on a new line. However, if the last item is followed by &, the newline sequence is omitted and the next **put** statement will carry on with the same output line.

Output can be generated in columns by using a variant of **put**, namely **putcol**. This variant does not use the minimum width needed for an output item, but uses the width needed for the corresponding output item of the preceding **put**. Hence:

put 'Column one '' Column two'
putcol 'five' 6

produces

```
Column one  Column two
five                 6
```

Note that string items have blanks appended to them, while numeric items have blanks prepended to them. Note also that any separating blanks prepended by the **put** are also prepended by the dependent **putcol** statements.

10 Selection

Executing a selection statement causes one of several nested groups of statements to be executed, depending on the value of one or more expressions. The general form of a selection statement is as follows:

selection =
  if expression
    {statement}
  {**elsif** expression
    {statement} }
  [**else**
    {statement}]
  end

Executing a selection statement of the form

if expression statement1 **else** statement2 **end**

will cause the expression to be evaluated and, if the resulting value is True, it will cause statement1 to be executed. Otherwise statement2 will be executed.

An **elsif** substatement is equivalent to an **else** followed by a nested **if** statement made up from what follows the **elsif** up to the **end**. In other words, **elsif** is a syntactic shorthand for **else if** that allows one to dispense with multiple **end**s at the end of the outer **if** statement.

11 Conditional statements

conditional-statement =
  (**commit** | **fail** | **return** [expression] | **exit** [number] |
  put | get | procedure-call | assignment) [**when** expression]

A statement of the form S **when** E is equivalent to a statement of the form **if** E, S **end**.

12 Iteration

Executing an iteration statement causes a nested group of statements to be executed zero or more times.

iteration =
  iteration-scheme {statement} (**loop** | **until** expression)
iteration-scheme =
  repeat | **while** expression | **for** iterator

The basic form of iteration:

repeat {statement} **loop**

will execute the enclosed statements, repeatedly, until an **exit** statement takes effect. For example:

> **repeat** statement1; **exit**; statement2; **loop**

will cause statement1 to be executed once, and statement2 not to be executed at all. The optional number that may follow an **exit** specifies how many nested loops must be terminated at once by the single exit statement.

Note that:

> **repeat** statement1 **until** expression

is equivalent to

```
repeat
  statement1
  exit when expression
loop
```

and that

> **while** expression, statement1 **loop**

is equivalent to

```
repeat
  exit when not expression
  statement1
loop
```

and that

```
for id1, id2 ∈ expr1, id3 ∈ id4 ∈ expr2 | expr3
  statement
loop
```

has the same effect as

```
Lst1 ← mklist(expr1); M ← 1
while M ≤ length(Lst1)
  Id1, Id2 ← Lst1(M); Lst2 ← mklist(expr2); N ← 1
  while N ≤ length(Lst2)
    Id4 ← Lst2(N); Lst3 ←mklist(Id4); K ←1
    while K ≤ length(Lst3)
      Id3 ←Lst3(K); if expr3, statement, end
      K +← 1
    loop
    N +← 1
  loop
  M +← 1
loop
```

13 Predefined functions

Predefined functions can be used without importing them from some module. In other words, they are "built-in". The predefined functions are not defined in detail.

Instead, their definitions must be inferred from their names, common sense, and the fact that the following Slim conditions should always be true:

max(x, m) $\in$ {**all** y $\in$ x | m(y) $\geq$ m(z) **forall** z $\in$ x}
max(x) = max(x, m) **and** m(y) = y **forall** y $\in$ x
min(x, m) $\in$ {**all** y $\in$ x | m(y) $\leq$ m(z) **forall** z $\in$ x}
min(x) = min(x, m) **and** m(y) = y **forall** y $\in$ x
floor(x) = max({**all** y $\in$ Integer | y $\leq$ x})
ceil(x) = min({**all** y $\in$ Integer | y $\geq$ x})
round(x) = floor(x + 0.5)
int(x) = **if** x $\geq$ 0 **then** floor(x) **else** ceil(x)
abs(x) = **if** x $\geq$ 0 **then** x **else** $-$x
sign($-$2) = $-$1 **and** sign(0) = 0 **and** sign(2) = 1
frac(x) = abs(x) - int(abs(x))
div(x, y) = floor(x/y)
mod(x, y) = x $-$ y $\times$ div(x, y)
rem(x, y) = sign(x) $\times$ mod(abs(x), abs(y))
odd(x) = (mod(x, 2) = 1)
even(x) = (mod(x, 2) = 0)
x = mknum(mkstr(x)) **and** mkstr(x) $\in$ String **forall** x $\in$ Number
mkstr({1}) = '{1}' **and** mkstr([1, 2]) = '[1 2]'
mkstr(Undefined) = 'Undefined' **and** mkstr(Entity) = 'Unprintable'
['A'..'Z'] = fmap(upcase, ['a'..'z'])
x = upcase(x) **forall** x $\in$ Character $-$ {'a'..'z'}
['a'..'z'] = fmap(locase, ['A'..'Z'])
x = locase(x) **forall** x $\in$ Character $-$ {'A'..'Z'}
[0..127] = fmap(code, ['\0'..'\127'])
{'\0'..'\127'} = fmap(char, {0..127})
slice('abcd' [2..3]) = 'bc' **and** slice('abcd' [4..5]) = 'd' **and** slice('abcd' [6..7]) = ''
trim(x,n) = slice(x, [n+1..length(x)$-$n]) **and** trim(x,1) = trim(x)
trim_left(x,n) = slice(x, [n+1..length(x)]) **and** trim_left(x,1) = trim_left(x)
trim_right(x,n) = slice(x, [1..length(x)$-$n]) **and** trim_right(x,1) = trim_right(x)
split(x, y) = [y([1..index(x, y)$-$1]), y([index(x, y)+length(x)..length(y)])
rep(x, y)(i) = x **forall** i $\in$ {1..y}
reverse(x) = **if** length(x) $\leq$ 1 **then** x **else** reverse(rest(x)) + first(x)
indexed(x) = [**all** [x(i), i] for i $\in$ domain(x)]
index(x, y) = first[**all** i $\in$ domain(y) | x = slice(y, [i..i+length(x)$-$1])])
length(x) = max({**all** i $\in$ Integer | x(i) $\neq$ Undefined})
size(x) = length(mklist(x))
mkmap(L)(P(1)) = P(2) **forall** P $\in$ L
mkset(x) = {**all** y $\in$ x} **and** x $\in$ **list of** Entity
mkset(f) = {**all** [x, f(x)] **for** x $\in$ domain(f)}
  and f in **map from** Entity **to** Entity
domain(f) = {**all** x $\in$ Entity | f(x) $\neq$ Undefined}
delete-some(x) = [e, x$-${e}] **forsome** e $\in$ x
some({}) = some([]) = Undefined
some({x}) = some([x]) = x
length([**all** x $\in$ [**all** some(y) **for** i $\in$ [1..1000]] | x = z])
  $\in$ [1000/(size(y)+1) .. 1000/size(y) $-$ 1] **forall** z $\in$ y]
mklist(x) = [**all** y $\in$ x] **and** x $\in$ **set of** Entity
insert(x, i, y) = slice(x, [1..i]) + [y] + slice(x, [i+1..length(x)])
delete(x, i) = slice(x, [1..i$-$1]) + slice(x, [i+1..length(x)])
delete-first(x) = [first(x), rest(x)]
delete-last(x) = [last(x), all-but-last(x)]

first(x) = hd(x) = x(1)
rest(x) = tl(x) = slice(x, [2..length(x)]) **and** length(x) > 1
rest(x) = tl(x) = [] **and** length(x) = 1
last(x) = x(length(x))
all-but-last(x) = slice(x, [1..length(x)−1]) **and** length(x) > 1
all-but-last(x) = [] **and** length(x) = 1
reduce(f, x, y) = y **and** length(x) = 0
reduce(f, x, y) = f(reduce(f, all-but-last(x)), last(x)) **and** length(x) ≥ 1
reduce(f, x, Undefined) = reduce(f, x)
reduce(f, x) = Undefined **and** length(x) = 0
reduce(f, x) = x(1) **and** length(x) = 1
reduce(f, x) = f(reduce(f, all-but-last(x)), last(x)) **and** length(x) ≥ 2
union(x) = reduce('+', x,)
concat(x) = reduce('+', x, '')
sum(x) = reduce('+', x, 0)
sum(x, y) = [**all** x(i) + y(i) **for** i ∈ domain(x)]
prod(x) = reduce('×', x)
prod(x, y) = [**all** x(i) × y(i) **for** i ∈ domain(x)]
ascending(x) = (x(i) ≤ x(j) **forall** i ∈ domain(x), j ∈ domain(x) | i ≤ j)
descending(x) = (x(i) ≥ x(j) **forall** i ∈ domain(x), j ∈ domain(x) | i ≤ j)
ascending(sort(x, '≤')) **and** descending(sort(x, '≥'))
sort(x) = sort(x, '<')
eval(f, [x, y, z]) = f(x, y, z) = eval('f', [x, y, z])

14 Procedures

procedure = **proc** identifier ['(' parameter-identifiers')'] {statement} [**endp**]
procedure-call = identifier ['('{expression}')']
parameter-identifiers =
  identifiers [':' expression] {';' identifiers [':' expression]}

A procedure may be parameterized by using parameter identifiers in its definition, and binding these identifiers to parameter values when the procedure is called. The parameter identifiers used in a procedure definition may be associated with types by means of the ':' notation. Note the significance of the semicolon. When a parameter variable is not explicitly typed, its type is assumed to be Entity. Parameter variables are local variables, and "binding a value" to a parameter variable is simply a way of assigning a value to the corresponding local variable of the procedure being called.

Procedures are values of type Entity and have the same rights as other values. When a procedure is called with an incorrect number of parameter values, the list of values is truncated or expanded (with Undefined) as necessary. If a parameter value does not match the type of the corresponding parameter variable, the variable is bound to Undefined.

Executing a **return** statement while executing a procedure, causes the statement following the procedure-call to be executed next.

15 Functions

function-definition = identifier'('parameter-identifiers')' '=' expression
function =
  func identifier['('parameter-identifiers')'] → expression
    {statement}
  [**endf**]

A value of type **maps** $S_1, S_2, \ldots, S_{n-1}$ **to** S_n may be denoted as an expression that, when evaluated with actual values substituted for its parameter identifiers, results in the value $s_n \in S_n$ to which the function maps the supplied actual values, $s_1 \in S_1, s_2 \in S_2, \ldots, s_{n-1} \in S_{n-1}$. A functional map may also be defined as a number of statements that, when executed, sets **result** to the appropriate value $s_n \in S_n$.

A function defined as a group of statements is evaluated by executing the statements as if the function application were a procedure call. If a **return** statement is executed while evaluating a function, the value of the expression following the **return** is assigned to **result** and the evaluation terminated.

The operators = and ≠ can be used to compare function values, but a function value denoted as a computational method is treated as different from all other values. For example, the following prints False:

f(x) = x + 1; g(x) = x + 1; put f = g

16 Scope and visibility

An identifier is visible throughout the scope in which its definition is contained. Local identifiers take precedence over imported identifiers. When an identifier is imported from more than one module, the identifiers from the module last imported take precedence. Objects whose identifiers are hidden, or not exported, can be accessed by prepending the name of the defining module to their identifiers.

An iterator introduces a nested scope. Hence in

{**all** i ∈ Integer | predicate(i)}

i is local to the set constructor. Thus, i need not be defined elsewhere and, if it is defined, the set constructor will not change it.

Static binding is used for reference to non-local objects inside procedure and function bodies. In the case of a function defined as an expression, however, the free variables appearing in the expression are statically bound to the *values* of these variables at the time the function definition is executed.

When the first value in an argument list is an abstract value (see section 17) the scope used to look up the value of the identifier being applied to the argument list, is *not* the current scope, but the global scope of the module in which the corresponding abstract type was defined. If the identifier is not found in that scope, the scopes associated with the supertypes are tried in arbitrary order. If not found there, the scopes of super-supertypes are tried, and so on. Only if this fails, will normal (static) scoping be applied. Name clashes among supertypes must be resolved by introducing distinct names into the scopes of subtypes. Separate abstract types are normally defined in separate modules.

17 Abstract types

Abstract types are sets of abstract values. Abstract values have no predefined operators and cannot be denoted directly. Abstract values are used to provide unique object identifiers. Object attributes are modelled by means of maps from abstract values to attribute values.

The pseudo-function New returns an abstract value that has not yet been included in any set but Entity. New is a pseudo function because it updates the variables appearing in its "parameter list" in such a way that a complete new object can be created. For example, the constant definition:

$p = \text{New(Person, Name 'Ralph', Age 25)}$

amounts to

$p = \text{hd}([\textbf{all}\ e \in \text{Entity} \mid e \notin S\ \textbf{forall}\ S \in \text{Entity}\ \textbf{where}\ S \neq \text{Entity}])$
$\text{Person} \overset{+}{\leftarrow} \{p\}$
$\text{Name}(p) \leftarrow \text{'Ralph'}$
$\text{Age}(p) \leftarrow 25$

If a variable is typed to contain a set of abstract values and it is itself used as a base set for a powerset expression used to type another variable, it will be implicitly updated when the other variable is updated. For example, if

Person : set of Entity $\leftarrow\{\}$
Student : set of Person $\leftarrow\{\}$
s : Student $\leftarrow$New(Student)

then s $\in$ Student and s $\in$ Person. In other words, Person becomes a supertype of Student, and the claim that the current value of Student is always a subset of Person, is maintained by the language. Deleting an abstract value from a supertype automatically deletes it from all subtypes as well[5].

Note that New has no effect and returns Undefined if its first parameter is not a variable typed to contain a set of abstract values. Note also that the predefined set Entity plays the role of the ultimate supertype. The proper way to completely destroy an abstract value is thus to delete it from Entity.

Multiple supertypes[6] can be specified by using the intersect operator in a powerset expression. For example, if:

Student : set of Person $\cap$ Customer $\leftarrow\{\}$

then New(Student) will include the new abstract value in Student, Person and Customer.

Additional supertypes also arise when abstract values are explicitly included into an abstract type. For example:

Sports-club-member $\overset{+}{\leftarrow}\{s\}$

makes Sports-club-member an additional supertype of the abstract value s only (other values in Student are unaffected).

18 Constraint definitions

constraint-definition = (**assert** | **pre** | **post** | **inv**) expression

When a constraint expression does not evaluate to true, the program is terminated or otherwise the debugger is activated.[7]

assert conditions are checked when control reaches the statements in which they appear; **pre** conditions are checked just before the first statement of the procedure or function body in which they appear is executed; **post** conditions are checked just before control returns to the caller of the procedure or function in which they appear.

inv conditions must be specified at the module level and are implicitly added to the **pre** and **post** conditions of all functions and procedures appearing in that module.

[5]Not yet implemented.
[6]Not yet implemented
[7]The compiler used for programs which should never terminate must either prove the assertions correct at compile time, or generate warnings and no code to check them at run time.

19 Lexical and parsing conventions

The non-ASCII symbols used in this document may be specified in ASCII form as follows:

-	$\rightarrow$	$\leftarrow$	$\stackrel{op}{\leftarrow}$	$\times$	$\neq$	$\geq$	$\leq$	x^y	$\times 10^5$	$\subseteq$	$\in$	$\not\subseteq$ $\notin$	$\cap$
_	−>	:=	op:=	*	#	>=	<=	x ** y	e5	**subset**	**in**	**not** op	**intersect**

The first letter of any identifier is "case insensitive", the others not. **import, export, get, put** and **putcol** statements may not span source lines. Function applications must be parameterized, and successive parameter lists must be juxtaposed. Semicolons may not be used to separate a list of names, and must be used to separate a list of procedure calls appearing on the same source line. The expression following a **return** must start on the same source line and may only be separated from the **return** by blanks.