The Napier88 Reference Manual

Release 2.0

Ron Morrison

Fred Brown

Richard Connor

Quintin Cutts

Al Dearle

Graham Kirby

Dave Munro

Contents

John Napier (1550-1617)

John Napier was born in Merchiston, Edinburgh in 1550. He matriculated at St Salvator's College, University of St Andrews in 1563. Very little is known about him during this period although he did study in Paris and travel in Italy and Germany before returning to Scotland to marry in 1571.

This was the period of the Scottish Reformation and Napier was very committed to the Protestant cause. In 1594, he wrote his *Plaine Discovery of the whole Revelation of Saint John* which he addressed to King James VI in a letter. This was the first Scottish book on the interpretation of scripture and has a significant place in the history of theology in Scotland.

John Napier is best known as the inventor of Logarithms. While important steps in the theory had been taken in the sixteenth century, notably by Burgi, it was Napier who first brought the subject, in any large way, to the attention of mathematicians. This was in his *Mirifici logarithmorum canonis descripto* (1614), the first important work on mathematics produced in Great Britain, and one which inspired Briggs, the professor of geometry at Gresham College, London, to develop the system of common logarithms with the decimal base. Napier also invented Napier rods or bones for use in multiplication, a development of a well-known Oriental method, and a number of formulae in trigonometry relating to circular parts. His other mathematical works include *De arte logistica* (1573 but not published until 1839), *Rabdoligæ seu numerationis per vigulas libri duo* (1617), in which the rods are described, and *Mirifici logarithmorum canonis constructio*, published two years after his death.

Napier was also a great advocate of the decimal fraction system invented by Stevinus in 1585. Indeed, it appears that Napier introduced the decimal point into common usage and eliminated the use of notation to indicate fractional position.

1 Introduction

The Napier88:persistent programming system provides the following facilities:

- Orthogonal persistence
  – models of data independent of longevity
- Type completeness
  – no restrictions on constructing types
- Higher-order procedures
  – procedures are data objects
- Parametric polymorphism
  – generic forms which may be specialised for use
- Abstract (existential) data types
  – for sophisticated protection and viewing
- Collections of bindings
  – for name space control, incremental system construction and system evolution
- A strongly typed stable store
  - a populated environment of typed data objects that may be updated atomically
- Graphical data types
  – for line drawings and raster images
- Concurrent execution and data access
  – using threads, semaphores and transactions
- Support for reflective programming
  – for system evolution

The Napier88 system consists of the language and its persistent environment. The persistent store is populated and, indeed, the system uses objects within the persistent store to support itself. The implication of orthogonal persistence is that the user need never write code to move or convert data for long or short term storage [ABC+83]. The model of persistence in Napier88 is that of reachability from a root object. The persistent store is also stable, that is, it is transformed from one consistent state to the next. Stabilisation must be invoked explicitly by the user to preserve data except that programs which terminate normally generate an automatic *stabilise* operation. Execution against the persistent store is always restarted from the last stabilised state.

Concurrency is provided by threads and semaphores [Mun93] for co-operative concurrency and by the CACS system [SM92] for competitive concurrency and designer transactions. Thus the notions of stability and visibility in commitment are orthogonal [Kra85, AMP86, MBB+89]. The entire computation including the state of the programs, threads and transactions is stable and recoverable after a system crash.

The Napier88 language is in the algol tradition as were its predecessors S-algol [Mor79] and PS-algol [PS88]. Following the work of Strachey [Str67] and Tennant [Ten77] the languages obey the principles of correspondence, abstraction and type completeness. This makes for languages with few defining rules allowing no

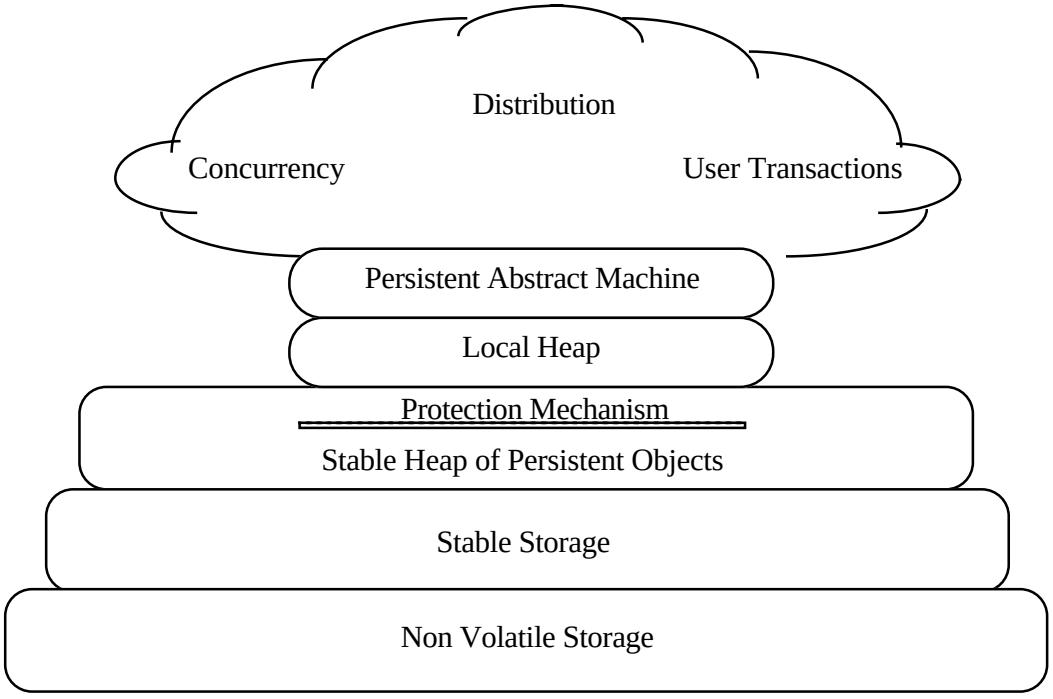
exceptions. It is the belief of the designers that such an approach to language design yields more powerful and less complex languages.

The Napier88 type system was evolving at the same time as Cardelli and Wegner [CW85] published their work. Many of the ideas are related to theirs and some have been borrowed from them. The philosophy is that types are sets of values from the value space. The type system is mostly statically checkable, a property we wish to retain wherever possible. However, some dynamic projection out of unions for types *any* and *env* [Dea89], as well as variant selection, allows the dynamic binding required for orthogonal persistence [ABC+83] and system evolution [MCC+93].

The type system is polymorphic, like ML [Mil78, MTH89], Russell [DD79] and Poly [Mat85] and uses the existentially quantified types of Mitchell & Plotkin [MP88, CMM91] for abstract data types. There is deliberately no type inference, to allow for explicit specialisation of polymorphic forms from the persistent store. A unique design feature of the implementation of the typed objects is that their storage format may be non-uniform [MDC+91]. The type system also includes graphical types for line drawing in an infinite two-dimensional real space and for manipulating raster images.

The type equivalence rule in Napier88 is by structure and both recursive and parameterised types are allowed in the type algebra, which in general leads to undecidable type checking. This is dealt with in Napier88 by a syntactic convention which allows the type checking to be sound, complete and co-complete [Con90].

The Napier88 system is designed as a layered architecture [Bro89] consisting of a compiler [Dea88, Con90, Cut92, Kir92], the Persistent Abstract Machine (PAM) [BCC+88, CBC+90] and persistent storage architecture [Bro89, BM92, Mun93]. All the Napier88 architectural layers are virtual in that, in any implementation, they may be implemented separately or together as efficiency dictates. Thus, they are definitional rather than concrete. In the current release the stable storage is provided by an after-look shadow paging mechanism [Bro89, BM92, Mun93]. The architecture is shown below:



Napier88 programs are executed in a strict left to right, top to bottom manner except where the flow of control is altered by one of the language clauses. On encountering an error state, the PAM generates a call to a standard error procedure held in the persistent store. These error procedures may be redefined by the user. The Persistent Abstract Machine also monitors interaction with the operating system in which Napier88 resides. When an asynchronous interrupt occurs the PAM records it and causes the appropriate procedure call to a standard event procedure in the persistent store. Again, the user may redefine the procedures used to intercept asynchronous interrupts.

There may be many incarnations of the stable persistent store and many activations of the PAM. However, only one PAM incarnation may work on one persistent store at any one time.

This version of the reference manual corresponds to release 2.0 of the Napier88 language. The language has only a few changes to that of release 1.0 [MBC+89a, MBC+89b] but the persistent environment has been significantly enriched and re-organised. The changes to the language are:

- a dynamic abstract witness model for abstract types, and
- type operators

A separate manual, the Napier88 Standard Library Reference Manual [KBC+94a] describes the persistent environment of the release. The main changes are the provision of a browser, a compiler for reflective programming, threads and semaphores, a new organisation of the object store to provide a navigation free store, distributed stores with remote scan and copy, and a hyper-programming system. The environment also provides a mechanism, through internet, for other sites to contribute programs and data which may then be accessed by remote scan and copy from other Napier88 stores. The mechanism for this is described in the Napier88 Release 2.0 Installation Guide [KBC+94b].

A third manual, the Napier88 to the Persistent Abstract Machine Compilation Rules Manual [BBC+94] describes the formal definition of Napier88 together with the rules to generate code for the Persistent Abstract Machine.

The Napier88 persistent programming system was originally planned as part of the PISA project [AMP86] and was intended as a testbed for our experiments in type systems, programming environments, concurrency, bulk data, object stores and persistence. The form of the Napier88 language was first conceived by Ron Morrison and Malcolm Atkinson but the main design and first implementation was done by Fred Brown, Richard Connor, Alan Dearle and Ron Morrison. Release 2.0 constitutes a major re-engineering, re-organisation and enhancement of the system by, in addition to the above, Quintin Cutts, Graham Kirby and Dave Munro.

Many people have contributed to the Napier88 design. Malcolm Atkinson played a major role [MBC+87, AM88, MBB+89], as did his research assistants Richard Cooper, Francis Wai & Paul Philbrow. At STC Technology Ltd., John Scott, John Robinson, Dave Sparks and Michael Guy aided, abetted and often criticised constructively the early designs.

Our Visiting Fellows at St Andrews, John Hurst, Chris Barter, Chris Marlin, John Rosenberg, Dave Stemple and Robin Stanton also contributed and influenced the design and the research undertaken in the context of Napier88.

Ron Morrison

2 Context Free Syntax Specification

The formal definition of a programming language gives programmers a precise description from which to work as well as providing implementors with a reference model. There are two levels of definition, syntactic and semantic. This section deals with the formal syntactic rules used to define the context free syntax of the language. Later, informal semantic descriptions of the syntactic categories will be given. The formal rules define the set of all syntactically legal Napier88 programs, remembering that the meaning of any one of these programs is defined by the semantics.

To define the syntax of a language another notation is required which is called a meta language and in this case a variation of Backus-Naur form is used.

The syntax of Napier88 is specified by a set of rules called *productions*. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. Syntactic categories have names which are used in productions and are distinguished from names and reserved words in the language. The syntactic categories can be mixed in productions with terminal symbols which are actual symbols of the language itself. Thus, by following the productions until terminal symbols are reached, the set of legal programs can be derived.

The meta symbols, that is those symbols in the meta language used to describe the grammar of the language, include | which allows a choice in a production. The square brackets [and] are used in pairs to denote that an term is optional. When used with a *, a zero or many times repetition is indicated. The reader should not confuse the meta symbols |, *, [and] with the actual symbols and reserved words in Napier88. To help with this reserved words will appear in **bold** and actual symbols will appear in outline bold. The names of the productions will appear in *italics*.

For example,

identifier ::= *letter* [*letter* | *digit* | _]*

indicates that an identifier can be formed as a letter, optionally followed by zero or many letters, digits or underbars.

The productions for Napier88 are recursive which means that there are an infinite number of legal Napier88 programs. However, the syntax of Napier88 can be described in about 80 productions.

The full context-free syntax of Napier88 is given in Appendix I.

3 Types and Type Rules

The Napier88 type system is based on the notion of types as a set structure imposed over the value space. Membership of the type sets is defined in terms of common attributes possessed by values, such as the operations defined over them. In the absence of polymorphism these sets or types partition the value space; polymorphic forms, which in Napier88 are polymorphic procedures and abstract data types, allow values to belong to more than a single type. The sets may be predefined, like *integer*, or they may be formed by using one of the predefined type constructors, like *structure*.

The constructors obey the *Principle of Data Type Completeness* [Str67, Mor79]. That is, where a type may be used in a constructor, any type is legal without exception. This has two benefits. Firstly, since all the rules are very general and without exceptions, a very rich type system may be described using a small number of defining rules. This reduces the complexity of the defining rules. The second benefit is that the type constructors are as powerful as is possible since there are no restrictions on their domain.

3.1 Universe of Discourse

The following base types are defined in Napier88:

1. The scalar data types are *int*, *real*, *bool*, *pixel*, *file* and *null*.

2. Type *string* is the type of a character string; this type embraces the empty string and single characters.

3. Type *pic* is the type of a conceptual line drawing, modelled in an infinite 2-D real space; this type embraces single points.

4. Type *image* is the type of a value consisting of a rectangular matrix of pixels.

5. Type *env* is the type of an environment; values of this type consist of a collection of bindings.

6. Type *any* is an infinite union type; values of this type consist of a value of any type together with a representation of that type.

The following type constructors are defined in Napier88:

7. For any type t, $*t$ is the type of a vector with elements of type t.

8. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, *structure* $(I_1: t_1,...,I_n: t_n)$ is the type of a structure with fields I_i and corresponding types t_i, for i = 1..n and n ≥ 0.

9. For identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, *variant* $(I_1: t_1,...,I_n: t_n)$ is the type of a variant with identifiers I_i and corresponding types t_i, for i = 1..n and n ≥ 0.

10. For any types $t_1,...,t_n$ and t, *proc* $(t_1,...,t_n \rightarrow t)$ is the type of a procedure with parameter types t_i, for i = 1..n, where n ≥ 0, and result type t. The type of a resultless procedure is *proc* $(t_1,...,t_n)$.

11. *proc* $[T_1,...,T_m]$ $(t_1,...,t_n \rightarrow t)$, where the definitions of types $t_1,...,t_n$ and t may include the use of the type variables $T_1,...,T_m$, is

the type of a procedure which is universally quantified over these type variables for m > 0 and n ≥ 0. These are polymorphic procedures.

12. *abstype* $[W_1,...,W_m]$ $(I_1: t_1,...,I_n: t_n)$, where the definitions of types $t_1,...,t_n$ may include the use of the type variables $W_1,...,W_m$, is the type of a structure which is existentially quantified over these type variables for m > 0 and n ≥ 0. These are abstract data types.

The world of data values is defined by the closure of rules 1 to 6 under the recursive application of rules 7 to 12.

In addition to the above, clauses which yield no value are of type *void*.

3.2 The Type Algebra

Napier88 provides a simple type algebra which allows the succinct definition of types within programs. As well as the base types and constructors already introduced, types may be defined with the use of

- aliasing
- recursive definitions
- type operators

3.2.1 Aliasing

Any legal type description may be aliased by an identifier to provide a shorthand or conceptually meaningful representation for that type. For example

```
type ron is int
type man is structure (age : int ; size : real)
type either is variant (first : ron ; second : man)
```

After its introduction an alias may be used in place of the full type description.

3.2.2 Recursive Definitions

Further expressibility may be achieved in the type algebra by the introduction of recursive types. Recursive types allow the definition of user-defined types for values with regular structures. The reserved word **rec** introduced before a type alias allows instances of that alias to appear in the type definition. Mutually recursive types may also be defined by the grouping of aliases with ampersands. In this case binding of identifiers within the mutual recursion group takes precedence over identifiers already in scope.

```
rec type intList is variant (cons : intNode ; tip : null)
     &   intNode is structure (head : int ; tail : intList)
```

3.2.3 Type Operators

Type operators allow families of types to be defined; operators may be specialised to provide particular types. These operators are simple functions over types; note however that they can always be statically resolved. Type operators are defined by an overloading of the syntax for type aliasing, with formal parameters being provided in square brackets after the alias. For example,

```
type heteroPair [a, b] is structure (first : a ; second : b)
type homoPair [t] is structure (first, second : t)
```

Operators are applied by the use of the identifier followed by specialising types in square brackets. For example,

```
type intRealPair is heteroPair [int, real]
type intPair is homoPair [int]
```

Notice that operator identifiers may not appear without being fully specialised.

Sometimes it is convenient to define higher-order operators:

```
type pairOperAppInt [oper [t]] is structure (first, second : oper [int])
```

Notice that in this case the *t* in the inner brackets may not be used as a formal parameter, and is simply an indication of the arity of the formal parameter *oper*. Identifiers used in such contexts have no extent.

3.2.4 Recursive Operators

Napier88 does not distinguish syntactically between recursive type operators and operators over recursive types. For example the following is a generic description of the family of list types:

```
rec type  list [s] is variant (cons : node [s] ; tip : null)
     &    node [t] is structure (head : t ; tail : list [t])
```

The uncontrolled introduction of recursive type operators leads to the ability to describe types over which no decidable structural equivalence algorithm is known. There is a restriction in Napier88 on the definition of recursive operators as follows:

The specialisation of a recursive operator on the right hand side of its own definition may not include any types which are constructed over its own formal parameters.

This rule extends through dependencies in sets of mutually recursive definitions; for example *list [*t]* would not be allowed on the right hand side in the above example because of the way the definition of *list* depends upon the definition of *node*. This rule

precludes the description of some useful type operators and types; for example the following may not be used to describe the type of an array of any dimension:

```
rec type array [t] is variant (simple : t ; higherOrder : array [*t])
```

The restriction has been introduced to allow fully decidable typechecking in Napier88 while less restrictive schemes are under investigation.

3.3 Type Equivalence

Type equivalence in Napier88 is based upon the meaning of types, and is independent of the way the type is expressed within the type algebra. Thus any aliases, recursion variables, and operator applications are fully factored out before equivalence is assessed. This style of type equivalence is normally referred to as structural equivalence.

The structural equivalence rules are as follows:

- Every base type is equivalent only to itself.
- For two constructed types to be equivalent, they must have the same constructor and be constructed over equivalent types.
- The bounds of a vector are not significant for type equivalence.
- For structure, variant and abstype constructors the labels are a significant part of the type, but their ordering is not.
- For procedure and polymorphic procedure types, the parameter ordering is a significant part of the type construction.

The definition of type equivalence for types which involve the type variables of polymorphic procedures and abstract data types is somewhat more subtle, and is defined in the appropriate sections of this manual.

Napier88 has no subtyping or implicit coercion rules. Values may be substituted by assignment or parameter passing only when their types are known statically to be equivalent.

The types of all expressions in Napier88 are inferred. There is no other type inference mechanism; in particular, the types of all procedure parameters and results must be explicitly stated by the programmer.

3.4 Type Rules

The type rules form a second set of rules to be used in conjunction with the context free syntax to define well-formed programs. The generic types that are required for the formal definition of Napier88 can be described by the following:

```
type arith     is    int | real
type ordered   is    arith | string
type literal   is    ordered | bool | pixel | pic | null | proc | file | image
```

type nonvoid **is** literal | structure |variant | env | any |
abstype | parameterised | poly | *nonvoid

type type **is** nonvoid | void

In the above, the generic type arith can be either an int or a real, representing the types integer and real in the language. In the type rules, the concrete types and generic types are written in shadow face to distinguish them from the reserved words, meta–symbols and actual symbols. Each of the type categories given above corresponds to one of the type construction rules and will be described later in this manual.

To check that a syntactic category is correctly typed, the context free syntax is used in conjunction with a type rule. For example, the type rule for the two-armed **if** clause is

t : type, **if** *clause* : bool **then** *clause* : t **else** *clause* : t => t

This rule may be interpreted as follows: t is given as a type from the table above. It can be any type including void. Following the comma, the type rule states that the reserved word **if** must be followed by a clause which must be of type boolean. This is indicated by : bool. The **then** and **else** alternatives must have clauses of the same type t for any t. The resultant type, indicated by =>, of this production is also t, the same as the alternatives.

The type rules will be used throughout this manual, in conjunction with the context-free syntax rules, to describe the language. A complete set of type rules for Napier88 is given in Appendix II.

3.5 First Class Citizenship

The application of the *Principle of Data Type Completeness* [Str67, Mor79] ensures that all data types may be used in any combination in the language. For example, a value of any data type may be a parameter to or returned from a procedure. In addition to this, there are a number of properties possessed by all values of all data types that constitute their civil rights in the language and define first class citizenship. All values of data types in Napier88 have first class citizenship.

The additional civil rights that define first class citizenship are:

- the right to be declared,
- the right to be assigned to and to be assigned,
- the right to have equality defined over them, and,
- the right to persist.

4 Literals

Literals are the basic building blocks of Napier88 programs that allow values to be introduced. A literal is defined by:

literal ::= *int_literal* | *real_literal* | *bool_literal* | *string_literal* | *pixel_literal* | *picture_literal* | *null_literal* | *proc_literal* | *image_literal* | *file_literal*

4.1 Integer Literals

These are of type integer and are defined by:

int_literal ::= [*add_op*] *digit* [*digit*]*
add_op ::= + | -

int_literal => int

An integer literal is one or more digits optionally preceded by a sign. For example,

1	0	1256	-8797

4.2 Real Literals

These are of type real and are defined by

real_literal ::= *int_literal*.[*digit*]*[e *int_literal*]

real_literal => real

Thus, there are a number of ways of writing a real literal. For example,

1.2	3.1e2	5.e5
1.	3.4e-2	3.4e+4

3.1e-2 means 3.1 times 10 to the power -2 (i.e. 0.031)

4.3 Boolean Literals

There are two literals of type boolean: **true** and **false**. They are defined by

bool_literal ::= **true** | **false**

bool_literal => bool

4.4 String Literals

A string literal is a sequence of characters in the character set (ASCII) enclosed by double quotes. The syntax is

string_literal ::= "[*char*]*"
char ::= any ASCII character except " | *special_character*
special_character ::= *special_follow* |

' if not followed by a *special_follow*

special_follow ::= m | p | o | t | b | ' | "

string_literal => string

The empty string is denoted by "". Examples of other string literals are:

"This is a string literal", and,

"I am a string"

The programmer may wish to have a double quote itself inside a string literal. This requires using a single quote as an escape character and so if a single or double quote is required inside a string literal it must be preceded by a single quote. For example,

"a'"" has the value a", and,

"a''" has the value a'.

There are a number of other special characters which may be used inside string literals. They are:

'b	backspace	ASCII code 8
't	horizontal tab	ASCII code 9
'n	newline	ASCII code 10
'p	newpage	ASCII code 12
'o	carriage return	ASCII code 13

4.5 Pixel Literals

There are two literals of type pixel: **on** and **off**. They are defined by

pixel_literal ::= **on** | **off**

pixel_literal => pixel

4.6 Picture Literal

There is only one picture literal. It is used to define a picture with no points.

picture_literal ::= **nilpic**

nilpic => pic

4.7 Null Literal

There is only one literal of the type **null**. It is used to ground recursion in variant types.

null_literal ::= **nil**

nil => null

4.8 Procedure Literals

A procedures is introduced into a program by its literal value. They are defined by:

proc_literal ::= **proc** [*type_parameter_list*] ([*named_param_list*] [-> *type_id*]); *clause*
type_parameter_list ::= [*identifier_list*]
named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

t : type, **proc** [*type_parameter_list*] ([*named_param_list*] [-> *type_identifier* : t]); *clause* : t

For example,

```
proc [t] (n : t → t ) ; n
```

is a procedure literal.

The meaning and use of procedures is described in Chapter 8.

4.9 Image Literal

There is only one image literal. It is used to define the image with no pixels. It has dimensions 0 by 0 and depth 0.

image_literal ::= **nilimage**

nilimage => image

4.10 File Literal

There is only one file literal. It is used to denote a file value that is not bound to a file in the file system.

file_literal ::= **nilfile**

nilfile => file

5 Expressions and Operators

5.1 Evaluation Order

The order of execution of a Napier88 program is strictly from left to right and top to bottom except where the flow of control is altered by one of the language clauses. This rule becomes important in understanding side-effects in the store. Parentheses in expressions can be used to override the precedence of operators.

When an error occurs in the system, a standard error procedure is called automatically. The standard error procedures are stored in the standard environment and may be altered by the user using the Napier88 facilities for updating environments.

An event may also occur during the execution of a Napier88 program. An event acts like an unexpected procedure call. Events are also defined in the standard environment and may be manipulated in the same manner as errors. Further details of events and errors may be found in the Napier88 Standard Library Reference Manual [KBC+94a].

5.2 Boolean Expressions

Objects of type boolean in Napier88 can have the value true or false. There are only two boolean literals, **true** and **false**, and three operators. There is one boolean unary operator, ~, and two boolean binary operators, **and** and **or**. They are defined by the truth table below:

a	b	~a	a **or** b	a **and** b
true	**false**	**false**	**true**	**false**
false	**true**	**true**	**true**	**false**
true	**true**	**false**	**true**	**true**
false	**false**	**true**	**false**	**false**

The precedence of the operators is important and is defined in descending order as:

~
and
or

Thus,

~a **or** b **and** c

is equivalent to

(~a) **or** (b **and** c)

This is reflected in the syntax rules which are:

expression ::= *exp1* [**or** *exp1*]*
exp1 ::= *exp2* [**and** *exp2*]*
exp2 ::= [≈] *exp3* ...

exp1 : bool **or** *exp1* : bool => bool
exp2 : bool **and** *exp2* : bool => bool
[~] *exp3* : bool => bool

The evaluation of a boolean expression in Napier88 is non-strict. That is, in the left to right evaluation of the expression, no more computation is performed on the expression than is necessary. For example,

true or *expression*

gives the value **true** without evaluating *expression* and

false and *expression*

gives the value **false** without evaluating *expression*.

5.3 Comparison Operators

Expressions of type boolean can also be formed by some other binary operators. For example, a = b is either **true** or **false** and is therefore boolean. These operators are called the comparison operators and are:

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
=	equal to
~=	not equal to
is	is a particular member of a variant
isnt	is not a particular member of a variant
contains	is present in an environment (see 14.5)

The syntactic rules for the comparison operators are:

exp2 ::= [~] *exp3* [*rel_op exp3*]
rel_op ::= *eq_op* | *co_op* | *variant_op*
eq_op ::= = | ~ =
co_op ::= < | <= | > | >=
variant_op ::= **is** | **isnt**

t : nonvoid, *exp3* : t *eq_op exp3* : t => bool
where *eq_op* ::= = | ~=

t : ordered, *exp3* : t *co_op exp3* : t => bool
where *co_op* ::= < | <= | > | >=

expression : variant *variant_op identifier* => bool
where *variant_op* ::= **is** | **isnt**

Note that the operators <, <=, > and >= are defined on integers, reals and strings whereas = and ~= are defined on all Napier88 data types. The interpretation of these operations is given with each data type as it is introduced. The operators **is** and **isnt** are for testing a variant identifier and are defined in Chapter 10.

Equality for types other than scalar types and strings is defined as identity.

5.4 Arithmetic Expressions

Arithmetic may be performed on data objects of type integer and real. The syntax of arithmetic expressions is:

exp3 ::= exp4 [*add_op exp4*]*
exp4 ::= exp5 [*mult_op exp5*]*
exp5 ::= [*add_op*] *exp6*
mult_op ::= *int_mult_op* | *real_mult_op* | ...
exp6 ::= ...

t : arith, *exp4* : t *add_op exp4* : t => t
t : arith, *add_op exp6* : t => t

exp5 : int *int_mult_op exp5* : int => int
where *int_mult_op* ::= * | **div** | **rem**

exp5 : real *real_mult_op exp5* : real => real
where *real_mult_op* ::= *| /

The operators mean:

+	addition
-	subtraction
*	multiplication
/	real division
div	integer division throwing away the remainder
rem	remainder after integer division

In both **div** and **rem** the result is negative only if exactly one of the operands is negative.

Some examples of arithmetic expressions are

a + b	3 + 2	1.2 + 0.5	-2.1 + a / 2.0

The language deliberately does not provide automatic coercion from integer to real, but the transfer may be explicitly invoked by the standard procedure *float* and the standard

procedure *truncate* is provided to transfer from real to integer. These are described in the Napier88 Standard Library Reference Manual [KBC+94a].

The evaluation of an arithmetic expression may cause the standard error procedures *unaryInt*, *Int*, *unaryReal* and *Real* to be called.

5.5 Arithmetic Precedence Rules

The order of evaluation of an expression in Napier88 is from left to right and based on the precedence table:

*	/	**div**	**rem**
+	-		

That is, the operations *, /, **div**, **rem** are always evaluated before + and -. However, if the operators are of the same precedence then the expression is evaluated left to right. For example,

6 **div** 4 **rem** 2 gives the value 1

Brackets may be used to override the precedence of the operator or to clarify an expression. For example,

3 * (2 - 1) yields 3 not 5

5.6 String Expressions

The string operator, ++, concatenates two operand strings to form a new string. For example,

"abc" ++ "def"

results in the string

"abcdef"

The syntax rule is:

exp4 ::= *exp5* [*string_mult_op exp5*]*

exp5 : string *string_mult_op exp5* : string => string
where *string_mult_op* ::= ++

A new string may be formed by selecting a substring of an existing string. For example, if s is the string "abcdef" then s (3 | 2) is the string "cd". That is, a new string is formed by selecting 2 elements from s starting at character 3. The syntax rule is:

exp6 ::= *expression* (*clause* | *clause*)

expression : string (*clause* : int | *clause* : int) => string

For the purposes of substring selection the first character in a string is numbered 1. The selection values are the start position and the length respectively.

To compare two strings, the characters are compared in pairs, one from each string, from left to right. Two strings are considered equal only if they have the same characters in the same order and are of the same length, otherwise they are not equal.

The characters in a string are ordered according to the ASCII character code. Thus,

```
"a" < "z"
```

is true.

The *null* string is less than any other string. Thus the less-than relation can be resolved by taking the characters pair by pair in the two strings until one is found to be less than the other. When the strings are not of equal length then they are compared as above and then the shorter one is considered to be less that the longer. Thus,

```
"abc" < "abcd"
```

The other relations can be defined by using = and <.

The evaluation of a string expression may cause the standard error procedures *concatenate* and *subString* to be called.

5.7 Picture Expressions

The picture drawing facilities of Napier88 allow the user to produce line drawings in two dimensions. The system provides an infinite two dimensional real space. Altering the relationship between different parts of the picture is performed by mathematical transformations, which means that pictures are usually composed of a number of sub-pictures.

In a line drawing system, the simplest picture is a point. For example, the expression,

```
[0.1, 2.0]
```

defines the point (0.1, 2.0).

Points in pictures are implicitly ordered. A binary operation on pictures operates between the last point of the first picture and the first point of the second. The resulting picture has as its first point, the first point of the first picture, and as its last, the last point of the second.

There are two infix picture operators. They are ^, which forms a new picture by joining the first picture to the second by a straight line from the last point of the first picture to the first point of the second. ++ also forms a new picture by including all the subpictures of both the operand pictures. The other transformations and operations on pictures are:

shift The new picture consists of the points obtained by adding the x and y shift values and the x and y co-ordinates of the points in the old picture. The ordering of the points is preserved.

scale The new picture consists of the points obtained by multiplying the x and y scale values with the x and y co-ordinates of the points in the old picture, respectively. The ordering of the points is preserved.

rotate The new picture consists of the points obtained by rotating the x and y co-ordinates of the points in the old picture clockwise about the origin by the angle indicated in degrees. The ordering of the points is preserved.

colour The new picture is the old one in a new colour.

text The new picture consists of the text string converted to a picture representation. The two points represent the base line of the string, which will be scaled to fit.

A text expression may cause the standard error procedure *Text* to be called while the picture is being drawn.

The full syntax of picture expressions is:

exp4 ::= *exp5* [*pic_mult_op exp5*]*
pic_mult_op ::= ^ | ++

expression : **pic** *pic_mult_op expression* : **pic** => **pic**

value_constructor ::= *picture_constr* | *picture_op* | ...
picture_constr ::= ⟦*clause*, *clause*⟧
picture_op ::= **shift** *clause* **by** *clause*, *clause* |
scale *clause* **by** *clause*, *clause* |
rotate *clause* **by** *clause* |
colour *clause* **in** *clause* |
text *clause* **from** *clause*, *clause* **to** *clause*, *clause*

⟦*clause* : **real** , *clause* : **real**⟧ => **pic**
shift *clause* : **pic** **by** *clause* : **real**, *clause* : **real** => **pic**
scale *clause* : **pic** **by** *clause* : **real**, *clause* : **real** => **pic**
rotate *clause* : **pic** **by** *clause* : **real** => **pic**
colour *clause* : **pic** **in** *clause* : **pixel** => **pic**
text *clause* : **string** **from** *clause* : **real** , *clause* : **real**
to *clause* : **real**, *clause* : **real** => **pic**

5.8 Pixel Expressions

Pixels may be concatenated to produce another pixel of a greater depth using the operator ++.

exp4 ::= *exp5* [++*exp5*]*

exp5 : **pixel** ++ *exp5* : **pixel** => **pixel**

For example,

```
let b = on ++ off ++ off ++ on
```

A pixel has depth representing the number of planes in the pixel. The planes are numbered from 0 and new pixels can be formed from subpixels of others. The syntax is

exp6 ::= *expression* (*clause* | *clause*)

expression : **pixel** (*clause* : **int** | *clause* : **int**) => **pixel**

For example, assuming the declaration of *b* above,

b (1 | 2) is the pixel **off** ++ **off**

This last expression is interpreted as the pixel formed by starting at plane 1 in *b* and selecting 2 planes.

The evaluation of a pixel expression may cause the standard error procedures *pixelOverflow* and *subPixel* to be called.

Two pixels are equal if they have the same depth and the corresponding planes have the same value.

5.9 The Persistent Store

There is one predefined procedure in Napier88 and it allows access to the persistent store. It is defined by

exp6 ::= PS ()

PS () => **any**

The structure of the persistent store is described in the Napier88 Standard Library Reference Manual [KBC+94a].

5.10 Precedence Table

The full precedence table for operators in Napier88 is:

/ * **div** **rem** ^

\+ - ++

~

= ~= < <= > >= **is** **isnt**

and

or

6 Declarations

6.1 Identifiers

In Napier88, an identifier may be bound to a data object, a procedure parameter, a structure field, a variant label, an abstract data type label or a type. An identifier may be formed according to the syntactic rule

identifier ::= *letter* [*id_follow*]
id_follow ::= *letter* [*id_follow*] | *digit* [*id_follow*] | _ [*id_follow*]

That is, an identifier consists of a letter followed by any number of underscores, letters or digits. The following are legal Napier88 identifiers:

x1 ronsObject look_for_Record1 Ron

Note that case is significant in identifiers.

6.2 Variables, Constants and Declaration of Data Objects

Before an identifier can be used in Napier88, it must be declared. The action of declaring a data object associates an identifier with a typed location which can hold values. In Napier88, the programmer may specify whether the location is constant or variable. A constant may be manipulated in exactly the same manner as a variable except that it may not be updated.

When introducing an identifier, the programmer must indicate the identifier, the type of the data object which is usually deduced, whether it is variable or constant, and its initial value. Identifiers are declared using the following syntax:

let *identifier init_op clause*
init_op ::= = | :=

let *identifier init_op clause* : nonvoid => void

A variable is declared by

let identifier := *clause*

For example,

```
let a := 1
```

introduces an integer variable with initial value 1. Notice that the compiler deduces the type.

A constant is declared by

let identifier = *clause*

For example,

```
let discrim = b * b - 4.0 * a * c
```

introduces a real constant with the calculated value. The language implementation will detect and flag as an error any attempt to assign to a constant.

6.3 Declaration of Types

Type names may be declared by the user in Napier88. The name is used to represent a set of objects drawn from the value space and may be used wherever a type identifier is legal. The syntax of type declarations is:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*
type_init ::= *identifier* [*type_operator_list*] **is** *type_id*
type_operator_list ::= ⟦*type_operator* [, *type_operator*]⟧
type_operator ::= *identifier* | *identifier* ⟦*type_operator*⟧

type_id ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **null** | **any** | **env** | **image** | **file** | *identifier* [*parameterisation*] | *type_constructor*

parameterisation ::= ⟦*type_list*⟧
type_list ::= *type_id* [, *type_list*]
type_constructor ::= ❋*type_id* | *structure_type* | *variant_type* | *proc_type* | *abstype*

structure_type ::= **structure** (([*named_param_list*]))
named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

variant_type ::= **variant** (([*variant_fields*]))
variant_fields ::= *identifier_list* : *type_id* [; *variant_fields*]

proc_type ::= **proc** [*type_parameter_list*] (([*parameter_list*] [-> *type_id*]))
parameter_list ::= *type_id* [, *parameter_list*]

abstype ::= **abstype** *type_parameter_list* ((*named_param_list*))
type_parameter_list ::= ⟦*identifier_list*⟧

Thus,

```
type al is bool
```

is a type declaration aliasing the identifier *al* with the boolean type. They are the same type and may be used interchangeably. Examples of type declarations will be given in later chapters.

6.4 Sequences

A sequence is composed of any combination, in any order, of declarations and clauses. The type of the sequence is the type of the last clause in the sequence. Where the sequence ends with a declaration, which by definition is of type *void*, the sequence is

of type *void*. If there is more than one clause in a sequence then all but the last must be of type *void*.

sequence ::= *declaration* [; *sequence*] | *clause* [; *sequence*]

sequence : **void** ? => **void**
t : **type**, *declaration* : **void** ; *sequence* : t => t
t : **type**, *clause* : **void** ; *sequence* : t => t
t : **type**, *clause* : t => t

6.5 Brackets

Brackets are used to make a sequence of clauses and declarations into a single clause. There are two forms, which are:

begin
    sequence
end

and

{*sequence*}

t : **type**, **begin** *sequence* : t **end** => t
t : **type**, {*sequence* : t} => t

{ and } allow a sequence to be written clearly on one line as a clause. For example,

```
let i := 2
for j = 1 to 5 do {i := i * i ; writeInt (i)}
```

However, if the sequence is longer than one line, the first alternative gives greater clarity. Nonvoid sequences are sometimes called block expressions.

6.6 Scope Rules

The scope of an identifier is limited to the rest of the sequence following the declaration. This means that the scope of an identifier starts immediately after the declaration and continues up to the next unmatched } or **end**. If the same identifier is declared in an inner sequence, then while the inner name is in scope the outer one is not.

6.7 Recursive Object Declarations

It is sometimes necessary to define values recursively. For example, the following defines a recursive version of the factorial procedure:

```
rec let factorial =    proc (n : int → int)
                       if n = 0 then 1 else n * factorial (n - 1)
```

The effect of the recursive declaration is to allow the identifier to enter scope immediately. That is, after the *init_op* and not after the whole declaration clause, as is

the case with non-recursive declarations. Thus, the identifier *factorial* used in the literal is the same as, and refers to the same location as, the one being defined. Chapter 8 gives an example of mutually recursive procedures.

Where there is more than one identifier being declared, all the identifiers come into scope at the same time. That is, all the names are declared first and then are available for the clauses after the *init_op*.

The initialising clauses for recursive declarations are restricted to literal values.

The full syntax of object declarations is:

object_decl ::= **let** *object_init* |
rec let *rec_object_init* [& *rec_object_init*]*
object_init ::= *identifier init_op clause*
rec_object_init ::= *identifier init_op literal*
init_op ::= = | :=

declaration => void
where *object_decl* ::= **let** *object_init* | **rec let** *rec_object_init*
[& *rec_object_init*]*
where *object_init* ::= *identifier init_op clause* : nonvoid
where *rec_object_init* ::= *identifier init_op literal* : nonvoid
where *init_op* ::= = | :=

6.8 Recursive Type Declarations

The full syntax of type declarations is:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*
type_init ::= *identifier* [*type_operator_list*] **is** *type_id*
type_operator_list ::= ⟦*type_operator* [, *type_operator*]⟧
type_operator ::= *identifier* | *identifier* ⟦*type_operator_list*⟧

For example, the following

```
rec type intList is variant (cons : intNode ; tip : null)
        & intNode is structure (head : int ; tail : intList)
```

defines a type for a list of integers.

7 Clauses

The expressions described in Chapter 5 are clauses which allow the operators in the language to be used to produce data objects. There are other kinds of clauses in Napier88 which allow the data objects to be manipulated and which provide control over the flow of the program.

7.1 Assignment Clause

The assignment clause has the following syntax:

clause ::= *name* := *clause*

t : nonvoid, *name* : t := *clause* : t => void

For example,

```
discriminant := b * b - 4.0 * a * c
```

gives *discriminant* the value of the expression on the right. Of course, the identifier must have been declared as a variable and not a constant. The clause alters the value denoted by the identifier. Assignments may also be made to vector elements and fields of structures and abstract data types.

The semantics of assignment is defined in terms of equality. The clause,

a := b

where *a* and *b* are both identifiers, implies that after execution *a* = *b* will be true. Thus, as will be seen later, assignment for scalar types means value assignment and for constructed types it means pointer assignment.

7.2 if Clause

There are two forms of the **if** clause defined by:

if *clause* **do** *clause* |
if *clause* **then** *clause* **else** *clause*

if *clause* : bool **do** *clause* : void => void
t : type, **if** *clause* : bool **then** *clause* : t **else** *clause* : t => t

In the single armed version, if the condition after the **if** is true, then the clause after the **do** is executed. For example, in the clause

```
if a < b do a := 3
```

the value 3 will be assigned to *a*, if *a* is smaller than *b* before the **if** clause is executed.

The second version allows a choice between two actions to be made. If the first clause is **true**, then the second clause is executed, otherwise the third clause is executed.

Notice that the second and third clauses are of the same type and the result is of that type. The following contains two examples of **if** clauses:

```
if x = 0 then y := 1 else x := y - 1
let temp = if a < b then 1 else 5
```

7.3 case Clause

The **case** clause is a generalisation of the **if** clause which allows the selection of one item from a number of possible ones. The syntax is:

case *clause* **of** *case_list* **default** : clause
case_list ::= *clause_list* : *clause* ; [*case_list*]

t : type ; t1 : nonvoid, **case** *clause* : t1 **of** *case_list*
default : *clause* : t => t
where *case_list* ::= *clause_list* : *clause* : t ; [*case_list*]
where *clause_list* ::= *clause* : t1 [, *clause_list*]

An example of the use of the **case** clause is

```
case next_car_colour of
        6,4   :     "green"
        3 - 2 :     "red"
default       :     "any"
```

During the execution of this clause, the value *next_car_colour* is compared in strict order, i.e left to right, top to bottom, with the expressions on the left hand side of the colon. When a match is found the clause on the right hand side is executed. Control is then transferred to the next clause after the **case** clause. If no match is found then the default clause is executed. The above **case** clause has result type **string**.

7.4 repeat ... while ... do Clause

There are three forms of this clause which allow loops to be constructed with the test at the start, the end or the middle of the loop. The three forms are encapsulated in the two production alternatives:

repeat *clause* **while** *clause* [**do** *clause*] | **while** *clause* **do** *clause*

repeat *clause* : void **while** *clause* : bool [**do** *clause* : void] => void
while *clause* : bool **do** *clause* : void => void

In each of the three forms the loop is executed until the boolean clause is **false**. The **while do** version is used to perform a loop zero or many times, whereas the **repeat while** is used for one or many times.

An example of the **repeat** ... **while** ... **do** clause is

```
let factorial := 1 ; let i := 0
repeat
begin
        writeString ("Factorial ") ; writeInt (i)
        writeString (" is ") ; writeInt (factorial)
        writeString ("'n")
end
while i < 8 do {i := i + 1 ; factorial := factorial * i}
```

7.5 for Clause

The **for** clause is included in the language as syntactic sugar where there is a fixed number of iterations defined at the initialisation of the loop. It is defined by:

for *identifier* = *clause* **to** *clause* [**by** *clause*] **do** *clause*

for *identifier* = *clause* : int **to** *clause* : int
[**by** *clause* : int] **do** *clause* : void => void

in which the clauses are: the initial value, the limit, the increment and the clause to be repeated, respectively. The first three are of type int and are calculated only once at the start. The **by** clause may be omitted where the increment is 1. The identifier, known as the control constant, is in scope within the void clause, taking on the range of values successively defined by initial value, increment and limit. That is, the control constant is considered to be declared at the start of the repetition clause. The repetition clause is executed as many times as necessary to complete the loop and each time it is, the control constant is initialised to a new value, starting with the initial loop value, changing by the increment until the limit is reached. An example of a **for** clause is:

```
let factorial := 1 ; let n = 8
for i = 1 to n do factorial := factorial * i
```

With a positive increment, the **for** loop terminates when the control constant is initialised to a value greater than the limit. With a negative increment, the **for** loop terminates when the control constant is initialised to a value less than the limit.

8 Procedures

8.1 Declarations and Calls

Procedures in Napier88 constitute abstractions over expressions, if they return a value, and clauses of type void if they do not. In accordance with the *Principle of Correspondence* [Str67], any method of introducing a name in a declaration has an equivalent form as a parameter.

Thus, in declarations of data objects, giving a name an initial value is equivalent to assigning the actual parameter value to the formal parameter. Since this is the only type of declaration for data objects in the language, it is also the only parameter passing mode and is commonly known as *call by value*.

Like declarations, the formal parameters representing data objects must have a name, a type and an indication of whether they are variable or constant. A procedure which returns a value must also specify its return type. The scope of the formal parameters is from their declaration to the end of the procedure clause. Procedures are defined as literals with the following syntax:

proc_literal ::= **proc** [*type_parameter_list*] ([*named_param_list*] [->*type_id*]); *clause*
type_parameter_list ::= [*identifier_list*]
named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

t : type, **proc** [*type_parameter_list*] ([*named_param_list*] [-> *type_id* : t]); *clause* : t

Thus, the integer identity procedure, called *int_id*, may be declared by:

```
let int_id = proc (n : int → int) ; n
```

The syntax of a procedure call is:

expression ([*application*])
application ::= *clause_list*

t : type, *expression* : proc ([*clause_list*]) => t
where *clause_list* ::= *clause* : nonvoid [,*clause_list*]

There must be a one-to-one correspondence between the actual and formal parameters and their types. Thus, to call the integer identity procedure given above, the following could be used,

```
int_id (42)
```

which will evaluate to the integer 42.

The type of *int_id* is written **proc** (**int** → **int**).

To complete the *Principle of Correspondence* for procedures, the parameters may be made constant. Variable parameters may be assigned to, but since they are local variables this only has local effect. Constant parameters may not be assigned to. For example, the parameter *n* in *int_id* is not assigned to and is more appropriately a constant. Therefore, the declaration should be:

```
let int_id = proc (constant n : int → int) ; n
```

Note that the constancy of the parameter is not part of the type, a notion that is important when deciding type equivalence.

8.2 Recursive Declarations

Recursive and mutually recursive declarations of procedures are allowed in Napier88. For example,

```
rec let tak = proc (x, y, z : int → int)
                  if x <= y then z else tak (  tak (x - 1, y, z),
                                               tak (y - 1, z, x),
                                               tak (z - 1, x, y) )
```

declares the recursive Takeuchi procedure.

Mutually recursive procedures may also be defined. For example,

```
rec let expression = proc () ; repeat exp1 () while have ("or")
& exp1 = proc () ; repeat exp2 () while have ("and")
& exp2 = proc ()
             case symb of
                  "identifier" : next_symbol ()
             default : {mustbe ("(") ; expression () ; mustbe (")")}
```

declares three mutually recursive procedures.

8.3 Polymorphism

Polymorphism permits abstraction over type. For example,

```
let id = proc [t] (constant x : t → t ) ; x
```

declares a procedure that is the identity procedure for all types. The square brackets signify that the procedure type is universally quantified by a type, *t*, and that once given that type, the procedure is from type *t* to *t*. To call this procedure the programmer may write,

```
id [int] (3)        which yields 3, or,
```

id [**real**] (4.2) which yields 4.2

or the type parameter may be used by itself. For example,

id [**int**] which yields a procedure equivalent to *int_id* above.

Thus, one procedure, *id*, is in fact, an infinite number of identity procedures, one for each type as it is specialised. The square brackets for quantifier type variables are used to signify that types are not part of the value space of the language, but are based on the philosophy that types are sets of values.

The type of *id* is written as

proc [t] (t → t)

in Napier88. Procedures of these polymorphic types are first class and may be stored, passed as parameters and returned as results, etc.

The advantage of the polymorphic abstraction should be obvious in the context of software reuse. For example, a procedure to sort a vector of integers may be written and another procedure to sort a vector of reals. By using the polymorphism in Napier88, one procedure for all types, instead of a different one for each type, may be written. This greatly reduces the amount of code that has to be written in a large system.

8.4 Equality and Equivalence

Two procedures are equal in Napier88 if and only if their values are derived from the same evaluation of the same procedure expression. For the cognoscenti, this means that they have the same closure.

In common with all aggregate objects in Napier88, equality means identity.

Two procedure types are structurally equivalent if they have the same parameter types in one-one correspondence and the same result type. For polymorphic procedures, there is the additional constraint that they have the same number of quantifiers used in a consistently substitutable manner.

In terms of types as sets, the polymorphic procedures are infinite intersections of types [CW85].

The declaration of a quantifier type variable acts as if the type is a new base type for type equivalence purposes. Thus quantifier type variables are only equivalent if they are derived from the same instantiation of the same type variable (identifier). As a consequence, a value of a quantifier type variable that has been injected into an infinite union may only be projected onto the same quantifier type variable.

9 Aggregates

Napier88 allows the programmer to group together data objects into larger aggregate objects which may then be treated as single objects. There are three such object types in Napier88: vectors, structures and images. If the constituent objects are of the same type, a vector may be used and a structure otherwise. Images are collections of pixels. Vectors, structures and images have the same civil rights as any other data object in Napier88. Both abstract data types (Chapter 11) and environments (Chapter 14) may also be considered methods of aggregation, but we have chosen to treat them separately.

All aggregate data objects in Napier88 have pointer semantics. That is, when an aggregate data object is created, a pointer to the locations that make up the object is also created. The object is always referred to by the pointer which may be passed around by assignment and tested for equality. The location containing the pointer and the constituent parts of the aggregate data object may be independently constant or variable.

9.1 Vectors

9.1.1 Creation of Vectors

A vector provides a method of grouping together objects of the same type. Since Napier88 does not allow uninitialised locations, all the initial values of the elements must be specified. The syntax is:

vector_constr	::=	[**constant**] **vector** *vector_element_init*
vector_element_init	::=	*range* **of** *clause* | *range* **using** *clause* |
		@clause **of** ⟦*clause* [, *clause*]*⟧
range	::=	*clause* **to** *clause*

```
t : nonvoid, vector range of clause : t => *t
t : nonvoid, vector range using clause : proc (int -> t ) => *t
t : nonvoid, vector @ clause : int of ⟦clause : t [, clause : t]*⟧ => *t
        where range ::= clause : int to clause : int
```

For example,

```
vector @1 of [ 1, 2 ,3, 4 ]
```

is a vector of integers, whose type is written as *int, with lower bound 1 and variable locations initialised to 1, 2, 3 and 4. Similarly,

```
let abc := vector @1 of [ 1, 2, 3, 4 ]
```

introduces a variable *abc* of type *int and the initial value expressed above.

Multi-dimensional vectors, which are not necessarily rectangular, can also be created. For example,

```
let Pascal = constant vector @1 of [
                        constant vector @1 of [ 1 ],
                        constant vector @1 of [ 1, 1 ],
                        constant vector @1 of [ 1, 2, 1 ],
                        constant vector @1 of [ 1, 3, 3, 1 ],
                        constant vector @1 of [ 1, 4, 6, 4, 1 ],
                        constant vector @1 of [ 1, 5, 10, 10, 5, 1 ] ]
```

Pascal is of type **int. It is constant, as are all its elements. This is a fixed table.

The use of the word **constant** before **vector** indicates that the elements are to be constant. The checking for constancy will be performed when an assignment is made to the element. The pointer constancy is determined by the *init_op*, which is = in this case and so indicates that the pointer is also constant.

The above form of vector expression is sometimes very tedious to write for large rectangular vectors with a common initial value. Therefore another form of vector expression is available. For example

```
vector -1 to 3 of -2
```

produces a five element integer vector with all the elements variable and initialised to -2. The lower bound of this vector is -1 and the upper bound is 3. The element initialising expression is evaluated only once and the result assigned to each of the elements.

A third form of vector initialisation is provided to allow the elements of a vector to be initialised by a function over the index. For example,

```
let squares = proc (n : int → int) ; n * n
let squares_vector = constant vector 1 to 10 using squares
```

In the initialisation, the procedure *squares* is called for every index of the vector in order from the lower to upper bound. The corresponding element is initialised to the result of its own index being passed to the procedure. In the above case, the vector *squares_vector* has elements initialised to 1, 4, 9, 16, 25, 36, 49, 64, 81, and 100.

The initialising procedure must be of type

proc (**int** → t)

and the resulting vector is of type **t*. This style of initialisation is particularly useful for vectors with constant elements.

The creation of a vector may call the standard error procedure *makeVector*.

9.1.2 upb and lwb

It is often necessary to interrogate a vector to find its bounds. The standard procedures *upb* and *lwb* are provided in Napier88 for this purpose. They are defined in the Napier88 Standard Library Reference Manual [KBC+94a] and are of type

proc [t] (*t → **int**).

9.1.3 Indexing

To obtain the elements of a vector, indexing is used. For vectors, the index is always an integer value. The syntax is:

exp6 ::= *expression* ⟨*dereference*⟩
dereference ::= *clause* [, *dereference*]

t : **nonvoid**, *expression* : *t (*clause* : **int**) => t

For example,

a (3 + 4)

selects the element of the vector *a* which is associated with the index value 7. Multi-dimension vectors may be indexed by using commas to separate the indices.

Indexing expressions may call the standard error procedure *vectorIndexSubs* and assignment to a vector element may call *vectorIndexAssign* and *vectorElementConstant*.

9.1.4 Equality and Equivalence

Two vectors are equal if they have the same identity, that is, the same pointer. Two vectors are type equivalent if they have equivalent element types. Notice that the bounds are not part of the type.

9.2 Structures

9.2.1 Creation of Structures

Objects of different types can be grouped together into a structure. The fields of a structure have identifiers that are unique within that structure. The structures are sets of labelled cross products from the value space. A structure may be created in two ways, the first of which has the following syntax:

structure_constr ::= **struct** ⟨[*struct_init_list*]⟩
struct_init_list ::= *identifier init_op clause* [; *struct_init_list*]

struct ⟨*struct_init_list*⟩ => structure
where *struct_init_list* ::= *identifier init_op clause* : **nonvoid** [; *struct_init_list*]

For example,

```
struct (a = 1 ; b := true)
```

creates a structure whose first field is a constant integer with the identifier *a* and whose second field is a variable boolean with the identifier *b*.

Structures may also be created using a type identifier. The syntax of structure types is:

structure_type ::= **structure** ⟨[*named_param_list*]⟩
named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

For example, a structure type may be declared as follows:

```
type person is structure (constant name : string ; age, height : int)
```

This declares a structure type, *person*, with three fields of type string, int and int, respectively. The *name* field is constant. It also declares the field identifiers, *name*, *age* and *height*.

To create a structure from a type declaration, the type identifier followed by the initialising values for the fields is used.

structure_creation ::= *identifier* [⟦*specialisation*⟧] (⟪[*clause_list*]⟫)

For example,

```
let ron = person( "Ronald Morrison", 42, 175 )
```

creates a structure of type *person* defined above. The initialising values must be in one-one correspondence with the structure type declaration.

9.2.2 Indexing

To obtain a field of a structure, the field identifier is used as an index. For example, if *ron* is declared as above, then,

ron (age)

yields 42. For the indexing operation to be legal, the structure must contain a field with that identifier. As with vectors, a constancy check is performed on assignment.

Field identifiers, when used as indices, are only in scope within the brackets following a structure expression. Thus these identifiers need only be unique within each structure type.

A comma notation may be used for vectors or structures when the elements or fields are themselves structures or vectors. The indexing of vectors and structures may therefore be freely mixed. For example, if *v* is a vector of vectors of persons then *v(i)(j)(name)* and *v(i,j,name)* and *v(i,j)(name)* are equivalent expressions.

Attempted assignment to a constant field of a structure will cause the standard error procedure *structureFieldConstant* to be called.

9.2.3 Equality and Equivalence

Two structures are equal if they have the same identity (pointer).

The type of a structure is the set of the field identifier-type pairs. Thus the structure *ron* has type:

```
structure (name : string ; age : int ; height : int)
```

Two structures have equivalent types when the types have the same set of identifier-type pairs for the fields. Note that the order of the fields is unimportant.

9.3 Images

9.3.1 Creation of Images

An image is a rectangular grid of pixels. Images may be created and manipulated using the raster operations provided in the language. The creation of images is defined by

image_constr ::= [**constant**] **image** *clause* **by** *clause image_init*
image_init ::= **of** *clause* | **using** *clause*
subimage_constr ::= **limit** *clause* [**to** *clause* **by** *clause*]
[**at** *clause*, *clause*]

image *clause* : **int** **by** *clause* : **int** **of** *clause* : **pixel** => **image**
image *clause* : **int** **by** *clause* : **int** **using** *clause* : **image** => **image**
limit *clause* : **image** [**to** clause : **int** **by** *clause* : **int**]
[**at** *clause* : **int** , *clause* : **int**] => **image**

The integer values following **at** above must be ≥ 0 and are subjected to an upper bound check. All other integer values must be > 0. If these conditions are violated, the standard error procedure *makeImage* is called.

An image is a two dimensional object made up of a rectangular grid of pixels. An image may be created as follows:

```
let c = image 5 by 10 of on
```

which creates *c* with 5 pixels in the X direction and 10 in the Y direction, all of them initiallised to on. The origin of all images is 0, 0 and in this case the depth is 1.

Multi-plane images may be formed by using multi-plane pixels, such as in,

```
let a = image 64 by 32 of on ++ off ++ on ++ on
```

Images are first class data objects and may be assigned, passed as parameters or returned as results. For example,

```
let b := a
```

will assign the existing image *a* to the new one *b*. In order to map the operations usual on bitmapped screens, the assignment does not make a new copy of *a* but merely copies the pointer to it. Thus the image acts like a vector or structure on assignment.

9.3.2 Raster Operations

There are eight raster operations which may be used as described in the following syntax.

raster ::= *raster_op clause* **onto** *clause*
raster_op ::= **ror** | **rand** | **xor** | **copy** | **nand** | **nor** | **not** | **xnor**

raster_op clause : **image onto** *clause* : **image => void**

thus, the clause

```
xor b onto a
```

performs a raster operation of *b* onto *a* using **xor**. Notice that *a* is altered *in situ* and *b* is unchanged. Both images have origin 0, 0 and automatic clipping at the extremities of the destination image is performed.

The raster operations are performed by considering the images as bitmaps and altering each bit in the destination image according to the source bit and the operation. Multiple plane raster operations are discussed in 9.3.4. The following gives the meanings of the operations (D stands for destination and S for source):

Operation	Interpretation	Result
ror	inclusive or	S or D
rand	and	S and D
xor	exclusive or	S xor D
copy	overwrite	S
nand	not and	~(S and D)
nor	not inclusive or	~(S or D)
not	not the source	~S
xnor	not exclusive or	~S xor D

Images may also be created by using an initialising image as a background pattern. For example,

```
let d = constant image 64 by 512 using abc
```

will create the image *d* of size 64 x 512 and then copy the image *abc* onto it as many times as is necessary to fill it in both directions, starting at 0, 0. This style of initialisation is particularly useful for setting up images with constant pixels and images of regular patterns.

Rastering onto an image of constant pixels causes the standard error procedure *imagePixelConstant* to be called.

9.3.3 Indexing

The **limit** operation allows the user to set up aliases to parts of images. For example,

```
let c = limit a to 1 by 5 at 3, 2
```

sets *c* to be that part of *a* which starts at 3, 2 and has size 1 by 5. *c* has an origin of 0,0 in itself and is therefore a window on *a*.

Rastering sections of images on to sections of other images may be performed by, for example,

```
xor   limit a to 1 by 4 at 6, 5 onto
      limit b to 3 by 4 at 9, 10
```

Automatic clipping on the edges of the limited region is performed. If the starting point of the limited region is omitted, then 0,0 is used and if the size of the region is omitted then it is taken as the maximum possible. That is, it is taken from the starting point to the edges of the host image. Limited regions of limited regions may also be defined.

If the source and destination images overlap, then the raster operation is performed in such a manner that each pixel is used as a source before it is used as a destination.

The evaluation of the **limit** operation may cause the standard error procedures *limitAt* and *limitAtBy* to be called.

9.3.4 Depth Selection

All the operations that have already been seen on images (raster, limit and assignment) work more generally with depth. Thus the raster operations perform the raster function plane by plane in one - one correspondence between source and destination. Automatic depth clipping at the destination is performed, and if the source has fewer planes than the destination, then the extra planes will remain unaltered. The **limit** operation works over all the planes of an image.

The depth of the image may be restricted by the depth selection operation. For example, assuming the earlier definition of *a*

```
let b = a( 1|2 )
```

yields *b* which is an alias for that part of *a* which has the two depth planes 1 and 2. 1 is the start plane and 2 is the number of planes. *b* has depth origin 0 and dimensions 64 by 32.

The full syntax of the depth selection operation is

exp6 ::= *expression* (*clause* | *clause*)

expression : image (*clause* : int | *clause* : int) => image

This indexing expression may call the standard error procedure *subImage*.

9.3.5 Equality and Equivalence

Two images are equal if they have the same pointer.

All images have equivalent types.

10 Variants

10.1 Variant Types

Variants are sets of labelled disjoint sums from the value space. A variant value has one of these identifier-value pairs. A variant type may be defined by

variant_type ::= **variant** (([*variant_fields*]))
variant_fields ::= *identifier_list* : *type_id* [; *variant_fields*]

For example,

```
type this_variant is variant (a : int ; b : real)
```

declares a type *this_variant* which may be an *a : int* or a *b : real.*

10.2 Variant Values

A variant value may be formed by naming the variant type and injecting the identifier-value pair into it. The syntax is:

variant_creation ::= *identifier* [[*specialisation*]] ((*identifier* : *clause*))

For example

```
let A := this_variant (b : 3.912)
```

declares a value *A* of type:

```
variant (a : int ; b : real)
```

with the value of value 3.912 injected with the identifier *b*. The variant type must contain the identifier-type pair that is used in the initialisation.

10.3 is and isnt

A variant object can be tested for having a particular identifier. The syntax is:

exp2 ::= *exp3* [*type_op identifier*]
type_op ::= **is** | **isnt**

expression : variant *type_op identifier* => bool
where *type_op* ::= **is** | **isnt**

Thus,

```
A is b
```

is legal and will yield the boolean value **true**. A compilation error will occur if the variant type does not contain the identifier tag.

10.4 Projection out of Variants

Variants are particularly useful when used in conjunction with recursive types. For example, the type definition for a list of integers might be:

```
rec type intList is variant (cons : intNode ; tip : null)
        & intNode is structure (head : int ; tail : intList)
```

The first element of the list is formed by

```
let first = intList (tip : nil)
let next := intList (cons : struct (hd = 2 ; tl := first))
```

In order to facilitate static type checking, a value injected into a variant is rebound to a constant location by the **project** clause. The syntax is:

project *clause* **as** *identifier* **onto** *project_list* **default** : *clause*
project_list ::= ... | *variant_project_list*
variant_project_list ::= *identifier* : *clause* ; [*variant_project_list*]

t : type, **project** *clause* : variant **as** *identifier* **onto** *variant_project_list*
default : *clause* : t => t
where *variant_project_list* ::= *identifier* : *clause* : t ; [*variant_project_list*]

The projected value is given a constant binding to the identifier following the **as**. The scope of the identifier is the clauses on the right hand side of the colons. This mechanism prevents side effects on the projected value inside the evaluation of the right hand side clauses and allows for static type checking therein. For projection, the variant is compared to each of the labels on the left hand side of the colons. The first match causes the corresponding clause on the right hand side to be executed. Within the clause, the identifier has the type of the projected value. Control passes to the clause following the **project** clause. Within the **default** clause, the constant identifier is bound to the original variant value.

For example, a procedure to reverse a list might be:

```
rec type intList is variant (cons : intNode ; tip : null)
        & intNode is structure (hd : int ; tl : intList)

let reverseList = proc (list : intList → intList )
begin
        let temp := intList (tip : nil) ; let done := false
        while ~done do
                project list as X onto
                cons :      begin
                                temp := intList (cons : struct (hd = X (hd); tl := temp) )
                                list := X (tl)
                            end
                default     : done := true
        temp
end
```

10.5 Variant Usage

The value of a variant may be projected by using the single quote (') notation. The syntax is

expression'identifier

For example, assuming the definition given for *A* above, A'b yields the value 3.912 of type real. The scope of the variant identifiers is such that they may only be used in variant injections and after the symbols **is**, **isnt** and '.

The above procedure to reverse an integer list might be written as

```
let reverseList = proc (list : intList → intList )
begin
        let temp := intList (tip : nil)
        while list isnt tip do
        begin
                temp := intList (cons : struct (hd = list'cons (hd); tl := temp) )
                list := list'cons (tl)
        end
        temp
end
```

The evaluation of the ' operation may cause the standard error procedure *varProject* to be called.

10.6 Equality and Equivalence

Two variant types are equivalent if they have the same set of identifier-type pairs.

Two variants are equal if they have equivalent types, the same identifier tags and equal values.

11 Abstract Data Types

Abstract data types may be used where the data object displays some abstract behaviour independent of representation type. Thus it is a second mechanism for abstracting over type.

11.1 Abstract Data Type Definition

Abstract data types may be introduced by the following syntax:

abstype ::= **abstype** *type_parameter_list* ⦇[*named_param_list*]⦈

Thus,

```
type TEST is abstype [i] (a : i ; constant b : proc (i → i) )
```

declares the type *TEST* as abstract. The type identifiers that are enclosed in the square brackets are called the witness type identifiers and are the types that are abstracted over.

A comparison can be made with polymorphic procedures which have universally quantified types. These abstract types are existentially quantified and constitute infinite unions over types [MP88].

The abstract data type interface is declared between the round brackets. In the above case, the type has two elements, a field *a* with type *i* and a constant procedure *b* with type

proc (i → i).

11.2 Creation of Abstract Data Objects

To create an abstract data object, the following syntax is used:

abstype_creation ::= *expression* ⟦*specialisation* ⟧ ⦇[*clause_list*]⦈

For example,

```
let inc_int = proc (a : int → int) ; a + 1
let this = TEST [int] (3, inc_int)
```

declares the abstract data object *this* from the type definition *TEST*, the concrete (as opposed to abstract) witness type int, the integer 3 and procedure *inc_int*. In the creation, the values must be in one-one type correspondence with the type definition.

Once the abstract data object is created, the user can never again tell how it was constructed. Thus *this* has type:

```
abstype [i] (a : i ; b : proc (i → i) )
```

and the user can never discover that the witness type is integer.

```
let that = TEST [int] (-42, inc_int)
```

creates another abstract data object. Although it is constructed using the same concrete witness type, this information is abstracted over, therefore *this* and *that* have the same type, namely,

```
abstype [i] (a : i ; b : proc (i → i) )
```

as does *also* below:

```
let inc_real = proc (b : real → real) ; b + 1.0
let also = TEST [real] (-41.99999, inc_real)
```

Thus a vector of the objects can be formed by:

```
let abs_TEST_vec = constant vector @1 of [this, that, also]
```

since they have the same type.

11.3 Use of Abstract Data Objects

Since the internal representation of an abstract data object is hidden, it is inappropriate to mix operations from one with another. That is, the abstract data object is totally enclosed and may only be used with its own operations.

A second requirement in the system is that the type checking on the use of these objects is static.

To achieve the above aims, the **use** clause is introduced to define a constant binding for the abstract data object. This constant binding can then be indexed to refer to the values in a manner that is statically checkable. The syntax of the **use** clause is

use *clause* **as** *identifier* [*witness_decls*] **in** *clause*

use *clause* : **abstype** **as** *identifier* [*witness_decls*] **in** *clause* : **void => void**

For example,

```
use abs_TEST_vec (1) as X in
begin
        X (a) := X (b) (X (a))
end
```

which will apply the procedure *b* to the value *a*, storing the result in *a*, for the abstract data object referred to by abs_TEST_vec (1). *X* is declared as a constant initialised to abs_TEST_vec (1).

This could be generalised to a procedure to act on any of the elements of the vector. For example,

```
let increment =  proc (this_one : TEST)
                 use this_one as X in
                 begin
                     X (a) := X (b) (X (a))
                 end

let lower = lwb [TEST] (abs_TEST_vec)
let upper = upb [TEST] (abs_TEST_vec)

for i = lower to upper do increment (abs_TEST_vec (i))
```

The scope of the identifiers in the interface is restricted to within the clause following the constant binding identifier.

In the **use** clause, the witness types may be named for use. For example,

```
use this as X [B] in
begin
        let id = proc (x : B → B) ; x
        let one := X (a)
        one := id [B] (one)
end
```

which renames the witness type as *B* and allows it to be used as a type identifier within the **use** clause.

11.4 Equality and Equivalence

An abstract data object is only equal to itself, that is equality means identity.

Two abstract data types are equivalent if they have the same identifiers with equivalent types in the interface and the same number of witness types used in a substitutable manner.

Two witness types are only equivalent if they derive from the same instance of the abstract data type. Thus a value of a witness type that has been injected into an infinite union may only be projected onto the corresponding witness of the same abstype instance.

12 Files

The file data type is used to access the I/O devices that are available to the host environment in which the Napier88 system is implemented. A file may refer to either a disk file, a terminal, a mouse, a tablet, an X-window, a socket, a shell or a raster graphics display. There are certain operations that are specific to each kind of file and a range of operations applicable to all files. A value of type file is implemented as a pointer to an object that describes the I/O device and its associated state. A set of standard procedures is provided to create and manipulate both file descriptors and the I/O devices they refer to. The operation of each of the standard procedures is fully described in the Napier88 Standard Library Reference Manual [KBC+94a].

12.1 File Literal

There is only one literal of type file, **nilfile**. See Section 4.10.

12.2 Equality and Equivalence

Two values of type file are equal if they are the same file.

All values of type file have equivalent types.

13 Type any

Type any is the type of the union of all values in Napier88. Values must be explicitly injected into and projected from type any. Both of these operations are performed dynamically and, in particular, the projection from any to another type involves a dynamic type check. We have argued elsewhere [ABC+83] that such a type check is required to support the binding of independently prepared programs and data in a type secure persistent object store.

13.1 Injection into Type any

Values may be injected into type any by the following syntax:

any (*clause*)

t : **nonvoid**, **any** (*clause* : t) => **any**

For example,

```
let int_any = any (-42)
```

which declares *int_any* to be the integer value -42 injected into type any.

Values of type any may be passed as parameters. For example, the following is an identity procedure for type any.

```
let id_any = proc (x : any → any) ; x
```

Thus polymorphic procedures may be written by using type any and injecting the parameters into any before the call and projecting the results after the call.

13.2 Projection from Type any

Values may be projected from type any by use of the **project** clause.

project *clause* **as** *identifier* **onto** *project_list* **default** : *clause*
project_list ::= *any_project_list* | ...
any_project_list ::= *type_id* : *clause* ; [*any_project_list*]

t : **type**, **project** *clause* : **any** **as** *identifier* **onto** *any_project_list*
default : *clause* : t => t
where *any_project_list* ::= *type_id* : *clause* : t ; [*any_project_list*]

The projected value is given a constant binding to the identifier following the **as**. The scope of the identifier is the clauses on the right hand side of the colons. This mechanism prevents side effects on the projected value inside the evaluation of the right hand side clauses and allows for static type checking therein. For projection, the type is compared to each of the types on the left hand side of the colons. The first match causes the corresponding clause on the right hand side to be executed. Within the clause, the identifier has the type of the projected value. After execution of the **project** clause, control passes to the clause following the **project** clause.

An example of projection is:

```
let write_type = proc (x : any → string)
                 project x as X onto
                     int     :    "type is integer"
                     real    :    "type is a real"
                 default     :    "type is neither integer nor real"
```

13.3 Equality and Equivalence

Two values of type any are equal if and only if they can be projected onto equivalent types and the projected values are equal.

All values of type any are type equivalent.

14 Environments

Environments [Dea89] are the infinite union of all labelled cross products. Environments differ from structures in that bindings may be added to or removed from environments dynamically. This mechanism is used in Napier88 to provide a method for dynamically composing block structure and thus controlling the name space. Environments also provide a method of storing and composing independently prepared programs and data, and thus control of the persistent object store in which the language resides.

A binding in Napier88 has four components: an identifier, a type, a value and a variable/constant location indicator [AM88]. The type environment is written as **env** in Napier88.

14.1 Creating a New Environment

A new environment is created by using the standard procedure *environment* of type:

proc (→ **env**)

Calling this procedure creates an environment with no bindings. The procedure is fully described in the Napier88 Standard Library Reference Manual [KBC+94a] .

14.2 Adding Bindings to an Environment

Bindings are added to environments by means of declarations. The syntax is:

env_decl	::=	**in** *clause* **let** *object_init* |
		in *clause* **rec let** *rec_object_init* [& *rec_object_init*]*
object_init	::=	*identifier init_op clause*
rec_object_init	::=	*identifier init_op literal*
init_op	::=	= | ::

Thus the program segment,

```
let this = environment ()
in this let a = 3
```

creates an environment *this*. In the environment, it creates the binding with identifier *a*, value 3, type integer and constant, i.e. {a, 3, int, constant}. The binding is added to the environment *this*, but not to the local scope. The standard error procedure *envRedeclaration* is called if the binding to be added does not have a unique identifier within the environment.

Another binding may be added by writing:

```
in this rec let fac := proc (n : int → int)
                          if n = 0 then 1 else n * fac (n-1)
```

after which *this* now has the form {a, 3, int, constant} {fac, **proc**..., **proc** (**int** → **int**), variable}

Non-recursive declarations of bindings are added to environments one at a time. Recursive declarations are added simultaneously, although in the above case there is only one. This corresponds to the scoping rules for non-recursive and recursive declarations in blocks.

An example of mutually recursive procedures in an environment is given by the following:

```
rec type list [t] is variant (cons : node [t] ; tip : null)
        & node [s] is structure (hd : s ; tl : list [s])

rec type object is variant (ron : bool ; fred : list [object])

in this rec let show = proc (this : object → string)
                        project this as X onto
                              ron : if X then "true" else "false"
                              fred : "[" ++ showlist (X) ++ "]"
                        default : ""

        & showlist = proc (this : list [object] → string)
                    if this is tip then ""
                    else show (this'cons (hd)) ++ "," ++ showlist (this'cons (tl))
```

Notice that although both *show* and *showlist* refer to each other, neither appears in the local scope. It would seem that none of the calls on these procedures are bound at all. To achieve the desired bindings for mutually recursive procedures in environments, the rule is that the identifiers bind to the environment's objects being declared.

14.3 Using Bindings in Environments

The bindings in an environment are brought into scope by a **use** clause. The syntax is:

clause ::= **use** *clause* **with** *signature* **in** *clause*
signature ::= *named_param_list*
named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

t : type, **use** *clause* : env **with** *signature* **in** *clause* : t => t

For example, to use *fac* declared earlier, the programmer may write:

```
use this with fac : proc (int → int) in ... fac ...
```

The effect of the **use** clause is to bring the name *fac* into scope at the head of the clause after **in**. *fac* binds to the location in the environment. Therefore, local assignment to *fac* will alter the value in the environment.

Notice that only a partial match on the signature of the environment is necessary. For every binding, the identifiers in the **use** must be the same as in the environment binding and the types equivalent. The constancy is determined by the original binding although it may be separately specified as constant in the **use** clause. No update to a constant value is allowed at run time and the compiler will flag as a syntax error any assignment

to a binding specified as constant. Bindings in the environment that are not specified in the signature of the **use** clause are not in scope in the clause following **in** and may not be used.

The standard error procedure *envProject* is called if the signature in the **use** clause cannot be matched by the environment.

14.4 Removing Bindings from Environments

Bindings may be removed from environments by the **drop** clause. The syntax is:

clause ::= **drop** *identifier* **from** *clause*

drop *identifier* **from** *clause* : `env => void`

For example,

```
drop fac from this
```

The effect of the above is that the binding is no longer reachable from the environment. It does not imply the destruction of any object or any dangling reference, since other bindings to the value in the dropped binding will still be valid. The standard error procedure *envDrop* is called if the dropped identifier does not exist in the environment.

14.5 The contains Clause

An environment may be tested by the infix operator **contains** to determine if it contains a binding with certain characteristics. The syntax is

exp6 ::= *clause* **contains** [**constant**] *identifier* [: *type_id*]|

clause : `env` **contains** [**constant**] *identifier* [: *type_id*] `=> bool`

There are several forms of this which allow testing of an identifier in an environment binding, an identifier-type pair, an identifier constancy binding and an identifier constancy type binding. Thus, using the environment *this* given earlier:

this **contains** a	true
this **contains** a : **int**	true
this **contains constant** a	true
this **contains constant** a : **int**	true
this **contains** a : **string**	false
this **contains** b	false

14.6 Equality and Equivalence

Two values of type environment are equal if they refer to the same environment. All environments have equivalent types.

15 References

[ABC+83]* Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R. “An Approach to Persistent Programming”. Computer Journal 26, 4 (1983) pp 360-365.

[AM88] Atkinson, M.P. & Morrison, R. “Types, Bindings and Parameters in a Persistent Environment”. In **Data Types and Persistence**, Atkinson, M.P., Buneman, O.P. & Morrison, R. (ed), Springer-Verlag (1988) pp 3-20.

[AMP86] Atkinson, M.P., Morrison, R. & Pratten, G.D. “Designing a Persistent Information Space Architecture”. In Proc. 10th IFIP World Congress, Dublin (1986) pp 115-120.

[BBC+94]* Balasubramaniam, D., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Kirby, G.N.C., Morrison, R., Munro, D.S. & Scheuerl, S. “The Napier88 To the Persistent Abstract Machine Compilation Rules”. University of St Andrews Technical Report CS/94/8 (1994).

[BCC+88] Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. Universities of Glasgow and St Andrews Technical Report PPRR-59-88 (1988).

[BM92]* Brown, A.L. & Morrison, R. “A Generic Persistent Object Store”. Software Engineering Journal 7, 2 (1992) pp 161-168.

[Bro89]* Brown, A.L. “Persistent Object Stores”. Ph.D. Thesis, University of St Andrews (1989).

[CBC+90]* Connor, R.C.H., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R. “The Persistent Abstract Machine”. In **Persistent Object Systems**, Rosenberg, J. & Koch, D.M. (ed), Springer-Verlag, Proc. 3rd International Workshop on Persistent Object Systems, Newcastle, Australia (1990) pp 353-366.

[CMM91]* Connor, R.C.H., McNally, D.J. & Morrison, R. “Subtyping and Assignment in Database Programming Languages”. In Proc. 3rd International Workshop on Database Programming Languages, Nafplion, Greece (1991).

[Con90]* Connor, R.C.H. “Types and Polymorphism in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1990).

[Cut92]* Cutts, Q.I. “Delivering the Benefits of Persistence to System Construction and Execution”. Ph.D. Thesis, University of St Andrews (1992).

[CW85] Cardelli, L. & Wegner, P. “On Understanding Types, Data Abstraction and Polymorphism”. ACM Computing Surveys 17, 4 (1985) pp 471-523.

[DD79] Demers, A. & Donahue, J. “Revised Report on Russell”. Cornell University Technical Report TR79-389 (1979).

[Dea88]* Dearle, A. “On the Construction of Persistent Programming Environments”. Ph.D. Thesis, University of St Andrews (1988).

[Dea89]* Dearle, A. “Environments: A flexible binding mechanism to support system evolution”. In Proc. 22nd International Conference on Systems Sciences, Hawaii (1989) pp 46-55.

[KBC+94a]* Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Moore, V.S., Morrison, R. & Munro, D.S. “The Napier88 Standard Library Reference Manual Version 2.2”. University of St Andrews Technical Report CS/94/7 (1994).

[KBC+94b]* Kirby, G.N.C., Brown, A.L., Connor, R.C.H., Cutts, Q.I., Dearle, A., Morrison, R. & Munro, D.S. “The Napier88 Release 2.0 Installation Guide”. University of St Andrews (1994).

[Kir92]* Kirby, G.N.C. “Reflection and Hyper-Programming in Persistent Programming Systems”. Ph.D. Thesis, University of St Andrews (1992).

[Kra85] Krablin, G.L. “Building Flexible Multilevel Transactions in a Distributed Persistent Environment”. In Proc. 2nd International Workshop on Persistent Object Systems, Appin, Scotland (1985) pp 86-117.

[Mat85] Matthews, D.C.J. “Poly Manual”. University of Cambridge (1985).

[MBB+89]* Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A., Hurst, A.J. & Livesey, M.J. “Language Design Issues in Supporting Process-Oriented Computation in Persistent Environments”. In Proc. 22nd International Conference on System Sciences, Hawaii (1989) pp 736-744.

[MBC+87]* Morrison, R., Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Atkinson, M.P. “Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment”. Software Engineering Journal, December (1987) pp 199-204.

[MBC+89a]* Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “The Napier88 Reference Manual”. Universities of Glasgow and St Andrews Technical Report PPRR-77-89 (1989).

[MBC+89b] Morrison, R., Brown, A.L., Connor, R.C.H. & Dearle, A. “Napier88 Release 1.0”. University of St Andrews (1989).

[MCC+93]* Morrison, R., Connor, R.C.H., Cutts, Q.I., Kirby, G.N.C. & Stemple, D. “Mechanisms for Controlling Evolution in Persistent Object Systems”. Journal of Microprocessors and Microprogramming 17, 3 (1993) pp 173-181.

[MDC+91]* Morrison, R., Dearle, A., Connor, R.C.H. & Brown, A.L. “An Ad-Hoc Approach to the Implementation of Polymorphism”. ACM Transactions on Programming Languages and Systems 13, 3 (1991) pp 342-371.

[Mil78] Milner, R. “A Theory of Type Polymorphism in Programming”. Journal of Computer and System Sciences 17, 3 (1978) pp 348-375.

[Mor79] Morrison, R. “On the Development of Algol”. Ph.D. Thesis, University of St Andrews (1979).

[MP88] Mitchell, J.C. & Plotkin, G.D. “Abstract Types have Existential Type”. ACM Transactions on Programming Languages and Systems 10, 3 (1988) pp 470-502.

[MTH89] Milner, R., Tofte, M. & Harper, R. **The Definition of Standard ML**. MIT Press, Cambridge, Massachusetts (1989).

[Mun93]* Munro, D.S. “On the Integration of Concurrency, Distribution and Persistence”. Ph.D. Thesis, University of St Andrews (1993).

[PS88] “PS-algol Reference Manual, 4th edition”. Universities of Glasgow and St Andrews Technical Report PPRR-12-88 (1988).

[SM92]* Stemple, D. & Morrison, R. “Specifying Flexible Concurrency Control Schemes: An Abstract Operational Approach”. In Proc. 15th Australian Computer Science Conference, Hobart, Tasmania (1992) pp 873-891.

[Str67] Strachey, C. **Fundamental Concepts in Programming Languages**. Oxford University Press, Oxford (1967).

[Ten77] Tennant, R.D. “Language Design Methods Based on Semantic Principles”. Acta Informatica 8 (1977) pp 97-112.

*Available via *ftp* from: `ftp-fide.dcs.st-andrews.ac.uk/pub/persistence.papers`

or via *WWW* from: `http://www-fide.dcs.st-andrews.ac.uk:8080/Publications.html`

Appendix I

Context Free Syntax

Session:

session ::= *sequence*?

sequence ::= *declaration* [; *sequence*] | *clause* [; *sequence*]

declaration ::= *type_decl* | *object_decl*

Type declarations:

type_decl ::= **type** *type_init* | **rec type** *type_init* [& *type_init*]*

type_init ::= *identifier* [*type_operator_list*] **is** *type_id*

type_operator_list ::= ⟦*type_operator* [, *type_operator*]⟧

type_operator ::= *identifier* | *identifier* ⟦*type_operator_list*⟧

Type descriptors:

type_id ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **null** | **any** | **env** | **image** | **file** | *identifier* [*parameterisation*] | *type_constructor*

parameterisation ::= ⟦*type_list*⟧

type_list ::= *type_id* [, *type_list*]

type_constructor ::= ❋*type_id* | *structure_type* | *variant_type* | *proc_type* | *abstype*

structure_type ::= **structure** ((*named_param_list*))

named_param_list ::= [**constant**] *identifier_list* : *type_id* [; *named_param_list*]

variant_type ::= **variant** ((*variant_fields*))

variant_fields ::= *identifier_list* : *type_id* [; *variant_fields*]

proc_type ::= **proc** [*type_parameter_list*] ((*parameter_list*) [-> *type_id*])

parameter_list ::= *type_id* [, *parameter_list*]

abstype ::= **abstype** *type_parameter_list* ((*named_param_list*))

type_parameter_list ::= ⟦*identifier_list*⟧

Object declarations:

object_decl ::= **let** *object_init* |
rec let *rec_object_init* [& *rec_object_init*]*

object_init ::= *identifier init_op clause*

rec_object_init ::= *identifier init_op literal*

init_op ::= = | :=

Clauses:

clause ::= *env_decl* |
if *clause* **do** *clause* |
if *clause* **then** *clause* **else** *clause* |
repeat *clause* **while** *clause* [**do** *clause*] |
while *clause* **do** *clause* |
for *identifier* = *clause* **to** *clause* [**by** *clause*] **do** *clause*|
use *clause* **with** *signature* **in** *clause* |
use *clause* **as** *identifier* [*witness_decls*] **in** *clause* |
case *clause* **of** *case_list* **default** : clause |
raster |
drop *identifier* **from** *clause* |
project *clause* **as** *identifier*
onto *project_list* **default** : *clause* |
name := *clause* |
expression

env_decl ::= **in** *clause* **let** *object_init* |
in *clause* **rec let** *rec_object_init* [& *rec_object_init*]*

signature ::= *named_param_list*

witness_decls ::= *type_parameter_list*

case_list ::= *clause_list* : *clause* ; [*case_list*]

raster ::= *raster_op clause* **onto** *clause*

raster_op ::= **ror** | **rand** | **xor** | **copy** | **nand** | **nor** | **not** | **xnor**

project_list ::= *any_project_list* | *variant_project_list*

any_project_list ::= *type_id* : *clause* ; [*any_project_list*]

variant_project_list ::= *identifier* : *clause* ; [*variant_project_list*]

Expressions:

expression ::= *exp1* [**or** *exp1*]*

exp1 ::= *exp2* [**and** *exp2*]*

exp2 ::= [~] *exp3* [*rel_op exp3*]

exp3 ::= *exp4* [*add_op exp4*]*

exp4 ::= *exp5* [*mult_op exp5*]*

exp5 ::= [*add_op*] *exp6*

exp6 ::= *literal* | *value_constructor* | (*clause*) |
begin *sequence* **end** | {*sequence*} |
expression (*clause* | *clause*) |
expression (*dereference*) |
expression'*identifier* |
expression [*specialisation*] |
expression ([*application*]) |
clause **contains** [**constant**] *identifier* [: *type_id*]|
any (*clause*) |
PS () |
name

dereference ::= *clause* [, *dereference*]

specialisation ::= *type_parameter_list*

application ::= *clause_list*

name ::= *identifier* | *expression* (*clause_list*) [(*clause_list*)]*

clause_list ::= *clause* [, *clause_list*]

Value constructors:

value_constructor ::= *vector_constr* | *structure_constr* | *image_constr* |
subimage_constr | *picture_constr* | *picture_op* |
structure_creation | *variant_creation* |
abstype_creation |

vector_constr ::= [**constant**] **vector** *vector_element_init*

vector_element_init ::= *range* **of** *clause* | *range* **using** *clause* |
@*clause* **of** [*clause* [, *clause*]*]

range ::= *clause* **to** *clause*

structure_constr ::= **struct** ([*struct_init_list*])

struct_init_list ::= *identifier init_op clause* [; *struct_init_list*]

image_constr ::= [**constant**] **image** *clause* **by** *clause image_init*

image_init ::= **of** *clause* | **using** *clause*

subimage_constr ::= **limit** *clause* [**to** *clause* **by** *clause*]
[**at** *clause*,*clause*]

picture_constr ::= [*clause*, *clause*]

picture_op ::= **shift** *clause* **by** *clause*, *clause* |
scale *clause* **by** *clause*, *clause* |
rotate *clause* **by** *clause* |

colour *clause* **in** *clause* |
text *clause* **from** *clause*, *clause* **to** *clause*, *clause*

structure_creation ::= *identifier* [[*specialisation*]] ([*clause_list*])

variant_creation ::= *identifier* [[*specialisation*]] (*identifier* : *clause*)

abstype_creation ::= *expression* [*specialisation*] ([*clause_list*])

Literals:

literal ::= *int_literal* | *real_literal* | *bool_literal* | *string_literal* | *pixel_literal* | *picture_literal* | *null_literal* | *proc_literal* | *image_literal* | *file_literal*

int_literal ::= [*add_op*] *digit* [*digit*]*

real_literal ::= *int_literal*.[*digit*]*[e *int_literal*]

bool_literal ::= **true** | **false**

string_literal ::= "[*char*]*"

char ::= any ASCII character except " | *special_character*

special_character ::= '*special_follow* |
' if not followed by a *special_follow*

special_follow ::= n | p | o | t | b | ' | "

pixel_literal ::= **on** | **off**

null_literal ::= **nil**

proc_literal ::= **proc** [*type_parameter_list*] ([*named_param_list*] [->*type_id*]); *clause*

picture_literal ::= **nilpic**

image_literal ::= **nilimage**

file_literal ::= **nilfile**

Miscellaneous and microsyntax:

add_op ::= + | -

mult_op ::= *int_mult_op* | *real_mult_op* | *string_mult_op* | *pic_mult_op* | *pixel_mult_op*

int_mult_op ::= * | **div** | **rem**

real_mult_op ::= * | /

string_mult_op ::= + +

pic_mult_op ::= ^ | + +

pixel_mult_op ::= + +

rel_op ::= *eq_op* | *co_op* | *variant_op*

eq_op ::= = | ~ =

co_op ::= < | <= | > | >=

variant_op ::= **is** | **isnt**

identifier_list ::= *identifier* [, *identifier_list*]

identifier ::= *letter* [*id_follow*]

id_follow ::= *letter* [*id_follow*] | *digit* [*id_follow*] | _[*id_follow*]

letter ::= a | b | c | d | e | f | g | h | i | j | k | l | m |
m | o | p | q | r | s | t | u | v | w | x | y | z |
A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Appendix II

Type Rules

type arith **is** int | real

type ordered **is** arith | string

type literal **is** ordered | bool | pixel | pic | null | proc | file | image

type nonvoid **is** literal | structure |variant | env | any |
abstype | parameterised | poly | *nonvoid

type type **is** nonvoid | void

Session :

sequence : void ? => void

t : type, *declaration* : void ; *sequence* : t => t

t : type, *clause* : void ; *sequence* : t => t

t : type, *clause* : t => t

Object Declarations :

declaration => void
where *object_decl* ::= **let** *object_init* | **rec let** *rec_object_init*
[& *rec_object_init*]*
where *object_init* ::= *identifier init_op clause* : nonvoid
where *rec_object_init* ::= *identifier init_op literal* : nonvoid
where *init_op* ::= = | :=

Clauses :

in *clause* : env **let***object_init* | => void
in *clause* : env**rec let** *rec_object_init* => void

clause : env **contains** [**constant**] *identifier* [: *type_id*] => bool

if *clause* : bool **do** *clause* : void => void

t : type, **if** *clause* : bool **then** *clause* : t **else** *clause* : t => t

repeat *clause* : void **while** *clause* : bool [**do** *clause* : void] => void

while *clause* : bool **do** *clause* : void => void

for *identifier* = *clause* : int **to** *clause* : int
[**by** *clause* : int] **do** *clause* : void => void

t : type, **use** *clause* : env **with** *signature* **in** *clause* : t => t

use *clause* : abstype **as** *identifier* [*witness_decls*] **in** *clause* : void => void

t : type ; t1 : nonvoid, **case** *clause* : t1 **of** *case_list*
default : *clause* : t => t
where *case_list* ::= *clause_list* : *clause* : t ; [*case_list*]
where *clause_list* ::= *clause* : t1 [, *clause_list*]

raster_op clause : image **onto** *clause* : image => void

drop *identifier* **from** *clause* : env => void

t : type, **project** *clause* : any **as** *identifier* **onto** *any_project_list*
default : *clause* : t => t
where *any_project_list* ::= *type_id* : *clause* : t ; [*any_project_list*]

t : type, **project** *clause* : variant **as** *identifier* **onto** *variant_project_list*
default : *clause* : t => t
where *variant_project_list* ::= *identifier* : *clause* : t ; [*variant_project_list*]

t : nonvoid, *name* : t := *clause* : t => void

Expressions :

exp1 : bool **or** *exp1* : bool => bool

exp2 : bool **and** *exp2* : bool => bool

[~] *exp3* : bool => bool

t : nonvoid, *exp3* : t *eq_op exp3* : t => bool
where *eq_op* ::= = | ~=

t : ordered, *exp3* : t *co_op exp3* : t => bool
where *co_op* ::= < | <= | > | >=

expression : variant *variant_op identifier* => bool
where *variant_op* ::= **is** | **isnt**

t : nonvoid, **any** (*clause*) : t => any

expression : env **contains** [**constant**] *identifier* [: *type_id*] => bool

t : arith, *exp4* : t *add_op exp4* : t => t

t : arith, *add_op exp6* : t => t

exp5 : int *int_mult_op exp5* : int => int
where *int_mult_op* ::= * | **div** | **rem**

exp5 : real *real_mult_op exp5* : real => real
where *real_mult_op* ::= *| /

exp5 : string *string_mult_op exp5* : string => string
where *string_mult_op* ::= + +

exp5 : pic *pic_mult_op exp5* : pic => pic
where *pic_mult_op* ::= ^ | + +

exp5 : pixel *pixel_mult_op exp5* : pixel => pixel
where *pixel_mult_op* ::= + +

PS () => any

t : literal, *literal* : t => t

t : nonvoid, *value_constructor* : t => t

t : type, (*clause* : t) => t

t : type, **begin** *sequence* : t **end** => t

t : type, {*sequence* : t} => t

expression : string (*clause* : int | *clause* : int) => string

expression : image (*clause* : int | *clause* : int) => image

expression : pixel (*clause* : int | *clause* : int) => pixel

t : nonvoid, *expression* : *t (*clause* : int) => t

Value constructors:

t : nonvoid, **vector** *range* **of** *clause* : t => *t

t : nonvoid, **vector** *range* **using** *clause* : proc (int -> t) => *t

t : nonvoid, **vector** @ *clause* : int **of** [*clause* : t [, *clause* : t]*] => *t
where range ::= *clause* : int **to** *clause* : int

image *clause* : int **by** *clause* : int **of** *clause* : pixel => image

image *clause* : int **by** *clause* : int **using** *clause* : image => image

limit *clause* : image [**to** clause : int **by** *clause* : int]
[**at** *clause* : int , *clause* : int] => image

struct (*struct_init_list*) => structure
where *struct_init_list* ::= *identifier init_op clause* : nonvoid [; *struct_init_list*]

[*clause* : real , *clause* : real] => pic

shift *clause* : pic **by** *clause* : real , *clause* : real => pic

scale *clause* : pic **by** *clause* : real , *clause* : real => pic

rotate *clause* : pic **by** *clause* : real => pic

colour *clause* : pic **in** *clause* : pixel => pic

text *clause* : string **from** *clause* : real , *clause* : real
to *clause* : real , *clause* : real => pic

literals :

[*add_op*] *digit* [*digit*]* => int

int_literal.[*digit*]*[e *int_literal*] => real

true | **false** => bool

" [*char*]* " => string

on | **off** => pixel

nil => null

t : type, **proc** [*type_parameter_list*] ([*named_param_list*]
[-> *type_identifier* : t]); *clause* : t

nilpic => pic

nilimage => image

nilfile => file

Appendix III

Program Layout

Semi-Colons

As a lexical rule in Napier88, a semi-colon may be omitted whenever it is used as a separator and it coincides with a newline. This allows many of the semi-colons in a program to be left out. However, to help the compiler deduce where the semi-colons should be, it is a rule that a line may not begin with a binary operator. For example,

```
a *
b
```

is valid but,

```
a
* b
```

is not.

This rule also applies to the invisible operator between a vector, structure or image and its index list and between a procedure and its parameters. For example,

```
let b = a (1,2)
```

is valid but,

```
let b = a
        (1)
```

will be misinterpreted since vectors can be assigned.

Comments

Comments may be placed in a program by using the symbol !. Anything between the ! and the end of the line is regarded by the compiler as a comment. For example,

```
a + b          ! add a and b
```

Appendix IV

Reserved Words

abstype	and	any	as	at			
begin	bool	by					
case	colour	constant	contains	copy			
default	div	do	drop				
else	end	env					
false	file	for	from				
if	in	int	image	is	isnt		
let	limit						
nand	nil	nilfile	nilimage	nor	not	nilpic	null
of	off	on	onto	or			
pic	pixel	proc	project				
real	rec	rem	repeat	ror	rand	rotate	
scale	shift	string	struct	structure			
text	then	to	true	type			
use	using						
variant	vector						
while	with						
xnor	xor						