The Napier88 Reference Manual

Ron Morrison
Fred Brown
Richard Connor
Al Dearle

This document should be referenced as:
"The Napier88 Reference Manual
Universities of Glasgow and St Andrews, PPRR-77-89".

Contents

Chapter

Appendices

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 formulas 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

This version of the reference manual corresponds to release 1.0 of the Napier language. We refer to this language from now on as Napier88.

The Napier88 language was originally developed as part of the PISA project [AMP86]. However, as the name of this version suggests, there has been at least one year's work on the system since the end of PISA. The language itself is intended as a testbed for our experiments in type systems, programming environments, concurrency, bulk data objects and persistence. At the present time, we do not have implementations of all of our ideas in Napier88 but we have prototyped a number of them in PS-algol [PPR88] and intend to develop them in Napier88 and its derivatives. The form of the language was first conceived by Ron Morrison and Malcolm Atkinson but these early ideas have been greatly extended by Fred Brown, Richard Connor, Alan Dearle and Ron Morrison.

The prototype Napier88 system is almost entirely built by Fred Brown, Richard Connor and Alan Dearle [DEA88] with a little help from Ron Morrison. These four were the main architects of the evolved design of the language, the supporting abstract machine [BCC88, CBC89] and persistent storage architecture [BRO89]. Quintin Cutts is assisting with the Napier88 in Napier88 bootstrap. Both Quintin and Graham Kirby must be thanked for their extensive proof reading of this manual and their thorough testing of the Napier88 system. Our Visiting Fellows at St Andrews, Chris Marlin and John Rosenberg, also improved the quality of this document by their proof reading.

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 environment [DEA89], as well as variant selection, allows the dynamic binding required for orthogonal persistence [ABC83]. The type system is polymorphic, like ML [MIL79, MIL83], Russell [DD79] and Poly [MAT85] and uses the existentially quantified types of Mitchell & Plotkin [MP88] for abstract data types. A unique design aim of the implementation of the typed objects is that store may be non-uniform in object representation.

The language is persistent and is supported by a layered architecture designed by Fred Brown [BRO89]. This means that the user need never write code to move or convert data for long or short term storage. The form of persistence differs from that of PS-algol and is based on the idea of Krablin [KRA85] that commit algorithms can be built on top of a synchronisation primitive. We add to that the concept of a stable store, therefore separating the notions of stability and visibility in commitment [AMP86, MBB89].

The system is at present without bulk data objects, concurrency and exceptions. Work continues on these ideas.

The Napier88 system consists of the language and its persistent store. This persistent store is populated and, indeed, the system uses objects within the persistent store to support itself. A description of the structure and contents of the persistent store is given in Chapter 16.

Many people have contributed to the Napier88 design. Malcolm Atkinson played a major role [AM85, MBC87, MBB89], 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. Nick Capon managed the PISA project and Graham Pratten directed it with Ron and Malcolm.

1.1 Napier88 Overview

The Napier88 system consists of the language and its persistent store. The system is supported by a layered architecture which contains, among other things, the Persistent Abstract Machine Layer and the Stable Persistent Storage Layer. 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.

The Persistent Abstract Machine Layer provides the ability to execute Napier88 programs. 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 Persistent Abstract Machine (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 UNIX on which Napier88 resides. When an asynchronous interrupt occurs the PAM records it. Four of these events are made available to the user. When they occur, they act like an unexpected procedure call to a standard event procedure in the persistent store. Again, the user may redefine these procedures.

The persistent store is populated when the system is installed and, indeed, as we have seen the persistent store is used by the PAM itself for error and event procedures. The structure of the persistent store is described in this manual. The objects within the released persistent store are referred to as standard objects and it is hoped that users will add to the contents of the store. The persistent store may be stabilised, that is, it is transformed from one consistent state to the next. This must be performed by the user to preserve data in the persistent store. The store is always restarted from the last stabilised state. Programs which terminate normally generate an automatic *stabilise* operation.

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.

Before reading the rest of this manual, the user is referred to Appendix III on the layout of Napier88 programs.

2 Syntax Specification

It is important that a programming language can be formally defined since it gives implementors 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 or *productions* as they are normally called. Each production specifies the manner in which a particular syntactic category (e.g. a clause) can be formed. The syntactic category name is enclosed in the meta symbols < and >, thus distinguishing it from names or reserved words in the language. These 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.

Other meta symbols include | which allows a choice in a production. The square brackets [and] are also used in pairs to denote that an object 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 in Napier88 and care will be taken to keep the two concepts completely separate in this description.

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.

As may be expected with any reasonably powerful programming language, 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 sets of objects from the value space. These 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* [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 to their domain. This increases the power of the language.

3.1 Universe of Discourse

There are an infinite number of data types in Napier88 defined recursively by the following rules:

1. The scalar data types are integer, real, boolean, string, pixel, picture, file and null.

2. The type image is the type of an object consisting of a rectangular matrix of pixels.

3. For any data type t, *t is the type of a vector with elements of type t.

4. 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.

5. 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.

6. For any data 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 and result type t. The type of a resultless procedure is **proc**($t_1,...,t_n$).

7. For any procedure type, **proc**($t_1,...,t_n \rightarrow t$) and type identifiers $T_1,...,T_m$, **proc**[$T_1,...,T_m$] ($t_1,...,t_n \rightarrow t$) is the type **proc**($t_1,...,t_n \rightarrow t$) universally quantified by types $T_1,...,T_m$. These are polymorphic procedures.

8. **env** is the type of an environment.

9. For any type identifiers $W_1,...,W_m$, identifiers $I_1,...,I_n$ and types $t_1,...,t_n$, **abstype** [$W_1,...,W_m$] (I_1: t_1,...,I_n: t_n), is the type of an existentially quantified data type. These are abstract data types.

10. The type **any** is the infinite union of all types.

11. For any user-constructed data type t and type identifiers, $T_1,...,T_n$, t[$T_1,...,T_n$] is the type t parameterised by $T_1,...,T_n$.

In addition to the above data types, there are a number of other objects in Napier88 to which it is convenient to give a type in order that the compiler may check their use for consistency.

12. Clauses which yield no value are of type void, as are procedures with no result.

The world of data objects is defined by the closure of rules 1 and 2 under the recursive application of rules 3 to 12.

3.2 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

type nonvoid **is** literal | image | 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 font to distinguish them from the reserved words. 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** may 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.

3.3 Type Equivalence Rule

Two data objects have the same type if they are structurally equivalent, that is the types represent the same set of values. Thus, even if a type identifier is aliased, for example,

type ron **is** int

the fact that objects of type *ron* are integers cannot be hidden. Abstract data types can be used for this purpose. The meaning of structural equivalence for two types is that they represent the same set. For scalar types, the construction of these sets is obvious. For

constructed types, it is not so obvious and is defined as part of the semantics of the constructor.

For example, the types,

type man **is structure** (age : **int** ; size : **real**)

and

type house **is structure** (size : **real** ; age : **int**)

have the same type since they both represent the same set of objects.

3.4 First Class Citizenship

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

The additional civil rights that define first class citizenship are:

1 the right to be declared,
2 the right to be assigned to and to be assigned,
3 the right to have equality defined over them, and,
4 the right to persist.

4 Literals

Literals are one of the basic building blocks of a program and 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 only two literals of type boolean. They are the symbols **true** and **false**. They may be used with obvious meaning when a boolean literal is required.

<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> ::= <double_quote>[<char>]*<double_quote>
<char> ::= any ASCII character except " | <special_character>
<special_character> ::= <single_quote><special_follow>
<special_follow> ::= n | p | o | t | b | <single_quote> | <double_quote>

<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

A pixel literal is 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

Procedures are introduced into programs by their literal value. They are defined by:

<proc_literal> ::= **proc**[<type_parameter_list>]
([<named_param_list>][<arrow><type_id>]);<clause>
<type_parameter_list> ::= <lsb><identifier_list><rsb>
<named_param_list> ::= [**constant**] <identifier_list>:<type_id>
[;<named_param_list>]

t : type, **proc** [<type_parameter_list>]([<named_param_list>]

[<arrow><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 Primitive 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 Chapter 16.

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>...

<expression> : bool **or** <expression> : bool => bool
<expression> : bool **and** <expression> : bool => bool
[~]<expression> : 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 in nature. 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 member of a variant (see Chapter 10)
isnt	is not a member of a variant (see Chapter 10)

The syntactic rules for these are:

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

t : nonvoid , <expression> : t <eq_op> <expression> : t => bool
where <eq_op> ::= = | ~=

t : ordered,<expression> : t <co_op> <expression> : t => bool
where <co_op> ::= < | <= | > | >=

<expression> : variant <type_op><identifier> => bool
where <type_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 in Napier88 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> | <string_mult_op> | <pic_mult_op> | <pixel_mult_op>

t : arith , <expression> : t <add_op> <expression> : t => t
t : arith , <add_op> <expression> : t => t

<expression> : int <int_mult_op> <expression> : int => int
where <int_mult_op> ::= <star> | **div** | **rem**

<expression> : real <real_mult_op><expression> : real => real where <real_mult_op> ::= <star> | /

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 Chapter 16.

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>]*

<expression>: string <string_mult_op><expression>: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><bar><clause>)

<expression>: string (<clause>: int <bar> <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.

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

"a" < "z"

is true.

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 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. 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 Form a picture consisting of the text string. 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> ::= <lsb><clause>,<clause><rsb>

<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>

<lsb><clause> : real,<clause> : real <rsb> => 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>]*

<expression> : pixel ++<expression> : 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><bar><clause>)

<expression> : **pixel** (<clause> : **int** <bar><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.

5.9 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 value 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, 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

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>
<type_init> ::= <identifier>[<type_parameter_list>]**is**<type_id>

<type_parameter_list> ::= <lsb><identifier_list><rsb>
<type_id> ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **file** | **null** | **any** | **env** | **image** | <identifier>[<parameterisation>] | <type_constructor>

<parameterisation> ::= <lsb><type_identifier_list><rsb>
<type_identifier_list> ::= <type_id>[,<type_identifier_list>]
<type_constructor> ::= <vector_type> | <structure_type> | <variant_type> | <proc_type> | <abstype>

<vector_type> ::= <star><type_id>
<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>] [<arrow><type_id>])
<parameter_list> ::= <type_id>[,<parameter_list>]
<abstype> ::= **abstype**<type_parameter_list>(<named_param_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 or declaration in the sequence. 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>]

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

{ < sequence > }

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

The {} method is there to allow a clause to be written clearly on one line. For example,

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

However, if the clause is longer than one line, the first alternative should be used for greater clarity. Nonvoid blocks are sometimes called block expressions.

6.6 Scope Rules

The scope of an identifier is limited to 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 allow objects to be defined recursively. The recursive declarations are particularly useful and necessary in defining recursive procedures. For example, the following defines a recursive version of the factorial procedure:

rec let factorial = **proc** (n : **int** → **int**)

The effect of the recursive declaration is to allow the identifier to enter scope immediately. That is, after the <init_op>s 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>'s.

The initialising clauses for recursive declarations are restricted to literal values to avoid ambiguities in evaluation order.

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>]*

For example, the following

rec type int_list **is variant** (cons : **structure** (hd : **int** ; tl : int_list) ; empty : **null**)

defines a type for a list of integers.

Further examples of recursive type definitions are given in later chapters.

7 Clauses

The expression is a clause which allows the operators in the language to be used to produce data objects. There are other statements 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.

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 : <cl

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**
1,4 : "green"
3 : "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.

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. If the increment is 1 then the **by** clause may be omitted. 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 the **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>]
                                    [<arrow><type_id>]);<clause>
<type_parameter_list>           ::=     <lsb><identifier_list><rsb>
<named_param_list>  ::=         [constant] <identifier_list>:<type_id>
                                    [;<named_param_list>]

t : type, proc [<type_parameter_list>]([<named_param_list>]
                                [<arrow><type_identifier> : 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 type parameters 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 identical 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.

As 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 type quantifier acts as if the type is a new base type for type equivalence purposes. Thus objects of quantifier types are only equal if the type is derived from the same type identifier.

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 15) may also be considered methods of aggregation, but we have choosen 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<lsb><clause>[,<clause>]*<rsb>
<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<lsb><clause> : t
                              [,<clause> : t]* <rsb> => *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* will be 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* will have 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

Since vectors may be assigned, it is often necessary to interrogate the vector to find its bounds. The standard procedures *upb* and *lwb* are provided in Napier88 for this purpose. They 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.

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** ; sex : **bool** ; age, height : **int**)

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

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

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

creates the 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 structures must contain a field with that identifier. As with vectors, a constancy check is performed on assignment.

Field identifiers, when used as an index, 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.

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 *person* has type:

structure (name : **string** ; sex : **bool** ; 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 three 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 initially on. The origin of all images is 0, 0 and in this case the depth is 1.

Multi-plane images may be formed by the concatenation of the initialising 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.

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. The following gives the meanings of the operations, (D stands for destination and S for source):

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 intialising 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.

9.3.2 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 bit 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.

9.3.3 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 to one correspondence between source and destination. Automatic depth clipping at the destination is performed, and if the source is too small to fill all the destination's planes, then these planes will remain unaltered. The **limit** operation also works with the depth of the 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. *b* has depth origin 0 and dimensions 64 by 32.

The full syntax of the depth selection operation is

<exp6> ::= <expression>[(<clause><bar><clause>)]

<expression> : image (<clause> : int <bar><clause> : int) => image

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

9.3.4 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 variant *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>[<lsb><expression><rsb>](<identifier>:<expression>)

For example

let A = this_variant (b : 3.912)

declares a variant *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**

<exp3> : variant <**is** | **isnt**> <identifier> => bool

Thus,

A **is** b

is legal and will yield a boolean value. A syntax error will occur if the variant type does not contain the correct identifier tag.

10.4 Variant Usage

The value of variants 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.

It should be noted, however, that to assign to a variant, the full syntax must be used. For example, to assign a new variant value to *A*, the following may be written:

```
A := this_variant (b : 3.159)
```

The scope of the variant identifiers is such that they may only be used after the symbols **is**, **isnt** and '.

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 int_list is variant (cons : structure (hd : int ; tl : int_list) ; empty : null)
```

The first element of the list is formed by

```
let first = int_list (empty : nil)
let next := int_list (cons : struct (hd = 2 ; tl := first) )
```

and a procedure to reverse an integer list might be

```
let reverse_list = proc (list : int_list → int_list)
begin
        let temp := int_list (empty : nil)
        while list isnt empty do
        begin
                temp := int_list (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.

10.5 Projection out of Variants

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.

Thus, the example above may be expressed as:

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

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).

There an infinite number of such objects in the Napier88 type system.

11.2 Creation of Abstract Data Objects

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

<expression><lsb><specialisation><rsb>(<clause_list>)

For example,

let inc_int = **proc** (a : **int** → **int**) ; a + 1
let inc_real = **proc** (b : **real** → **real**) ; b + 1.0
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 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 our system is that the type checking on the use of these objects is required to be 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 brackets 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 [t] (x : t → t) ; 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.

The declaration of a witness type acts, which occurs in a type declaration or a **use** clause, as if the type is a new base type for type equivalence purposes. Thus objects of witness types are only equal if the type is derived from the same type identifier.

12 Parameterised Types

Type parameterisation is allowed within the type space. It is performed by qualifying the type identifier with another type in square brackets. An example of a parameterised type declaration is:

type pair [t] **is structure** (a : t ; b : t)

There are no objects of this type but it may be used as a shorthand for denoting other types. For example,

type int_pair **is** pair [**int**]

is equivalent to the type

structure (a : **int** ; b : **int**).

Parameterised types are most useful when used with recursive variants. For example, the generic type for lists could be:

rec type list [t] **is variant** (cons : **structure** (hd : t ; tl : list [t]) ; empty : **null**)

or,

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

if an identifier is required for the structure type.

13 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 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 Chapter 16.

13.1 File Literal

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

13.2 Persistence

When a stabilise operation is performed, the current state of each open file is recorded in the persistent store. When the Napier system is initialised it uses this state information to re-open those files that were open during the previous stabilise. For disk files the recorded state is used to set the position of the next read or write operation on the file, for terminal files the input mode of the terminal is reset and for raster files the image on the display is restored. If, for some reason, a file cannot be re-opened, it is marked as closed.

13.3 Equality and Equivalence

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

All values of type file have equivalent types.

14 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 [ABC83] that such a type check is required to support the binding of independently prepared programs and data in a type secure persistent object store.

14.1 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. This possibility becomes more interesting with the use of projection.

14.2 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> | <variant_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 areal"
                    default         :   "type is neither integer
nor real"
```

14.3 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.

15 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.

The type environment is written as **env** in Napier88.

15.1 Creating a New Environment

A new environment may be created by using the standard procedure *environment*. It has type:

proc (→ **env**)

Calling this procedure creates an environment with no bindings.

15.2 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>

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.

15.3 Using Bindings in Environments

The bindings in an environment may be brought into scope by the **use** clause. The syntax is:

```
<clause>      ::=   use <clause> with <signature> in clause
<signature>   ::=   [constant]<identifier_list>:<type_id>[;<signature>]

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 <clause>
```

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.

15.4 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 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 a binding in the given environment.

15.5 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
this **contains** a : int
this **contains constant** a
this **contains constant** a : int

all yield true; but,

this **contains** b

yields false.

15.6 Persistence

In accordance with the concept of orthogonal persistence, all data objects in Napier88 may persist. For each incarnation of the Napier88 object store there is a root of persistence which may be obtained by calling the predefined procedure *PS* which has type:

proc (→ **env**)

Thus, the distinguished root of the persistent store graph is of type environment. When a program is activated, the distinguished root will contain all the bindings for that universe. The standard bindings are defined in Chapter 16, one of these procedures, the procedure to create environments is bound to the identifier *environment*. The following program illustrates the use of this procedure:

```
let ps = PS ()
use ps with environment : proc ( → env) in
begin
        let new_env = environment ()
        ...
end
```

This program binds the root of persistence to the local identifier *ps*. The **use** clause binds the identifier *environment* to the environment creation procedure in the root environment. Inside the body of the **use** clause this procedure is called to create a new (empty) environment. The reader should note that the environment bound to *new_env* is not yet persistent. Objects that persist beyond the activation of the unit that created them are those which the user has arranged to be reachable from the root of persistence. To determine this the system computes the transitive closure of all objects starting with the root. Thus, in order to store objects in the persistent store, the user has to alter or add bindings that can be reached from the distinguished root. In order to make the environment *new_env* persist, the above example may be rewritten as:

```
let ps = PS ()
use ps with environment : proc ( → env) in
begin
        in ps let new_env = environment ()
end
```

The environment will only persist if the standard procedure *stabilise* is called before the program terminates or the program terminates normally, as this causes an automatic call of *stablise*.

15.7 Equality and Equivalence

Two values of type environment are equal if they refer to the same environment.

All objects of type environment have equivalent types.

16 The Standard User Environment

The root of persistence may be accessed by calling the predefined procedure:

PS : **proc** (→ **env**)

The environment obtained as a result of executing this procedure is called the *root environment*. The Napier88 system consists of the language and its persistent store. The structure and content of this persistent store is described in this Chapter. It is hoped that as the Napier88 user community grows this environment will grow with it. Users with code that they feel would contribute to this environment should contact one of the authors of this manual.

16.1 The Root Environment

The root environment supplied with the Napier88 release contains the following items, each of which is a constant:

environment : **proc** (→ **env**)
scan : **proc** (**env**, **proc** (**string**, typeRep, **bool**))
System : **env**
Time : **env**
Device : **env**
User : **env**
Vector : **env**
String : **env**
IO : **env**
Format : **env**
Arithmetical : **env**
Graphical : **env**
Font : **env**
Error : **env**
Event : **env**

Each of these is now described in turn.

environment : **proc** (→ **env**)

This procedure yields a new (empty) environment every time it is called.

scan : **proc** (e : **env** ; id : **proc** (**string**, typeRep, **bool**))

This procedure calls the procedure supplied as a parameter once for every binding in the environment *e*. Each invocation of the procedure is passed the name of the identifier in the binding, a representation of its type and a boolean to indicate whether or not the location is constant. The representation of the type contains a canonical string representation and a flag to indicate if the type contains any unbound quantifiers. *typeRep* is defined as follows:

type typeRep **is structure** (unboundQuantifiers : **bool** ; representation : **string**)

The environments listed above are described in the following sections. It should be noted that all the items contained in the following environments are constants unless otherwise stated.

16.2 System Environment

The system environment contains the following procedures:

stabilise : **proc** ()

This procedure records the entire state of the Napier88 system on non-volatile storage. The system may be restarted from the most recently stabilised point. *stabilise* is called automatically on normal program termination.

diskgc : **proc** ()

This procedure performs an explicit garbage collection of the entire persistent store.

abort : **proc** ()

This procedure shuts down the Napier88 system. No stabilisation is performed.

16.3 Time Environment

date : **proc** (→ **string**)

This procedure gives the date and time in the following format:

Wed Apr 26 17:32:54 BST 1989

time : **proc** (→ **int**)

This procedure returns the CPU time used by the Napier88 system since it was initialised. The time is measured in 60 Hz clock ticks.

16.4 Device Environment

getScreen : **proc** (fd : **file** → **image**)

If the file *fd* is a raster device, this procedure returns the image associated with that device. If the file is not a raster file, a call will be made to the error procedure named *getScreen* in the error environment with the parameter supplied to the original call.

locator : **proc** (fd : **file** ; data : ***int**)

If the file *fd* is a mouse or tablet device, this procedure copies the locator information into the vector supplied as a parameter.

If the file is not a mouse or tablet device, a call will be made to the error procedure named *locator* in the error environment with the parameters supplied to the original call.

The elements of the vector are filled in as follows:

1	if the file is a tablet, this element is its X dimension, otherwise 0,
2	if the file is a tablet, this element is its Y dimension, otherwise 0,
3	the locator X position,
4	the locator Y position,
5	the state of button 1,
6...n	the state of button n - 4.

If the vector has more elements than the information available, the extra elements are not filled in. If it has too few elements, only the ones supplied are filled in. If the device is a mouse, the X and Y positions are relative to the last locator instruction. If the device is a tablet, the X and Y positions are absolute.

colourMap : **proc** (fd : **file** ; p : **pixel** ; i : **int**)

For the device specified by *fd*, provided that it is a raster device, set the colour map entry for the pixel *p* to be the integer *i*. The interpretation of *i* is device dependent.

If the file is not a raster file, a call will be made to the error procedure named *colourMap* in the error environment with the parameters supplied to the original call.

colourOf : **proc** (fd : **file** ; p : **pixel** → **int**)

If the file *fd* is a raster device, this procedure returns the colour map entry associated with pixel *p* in the device specified by *fd*.

If the file is not a raster file, a call will be made to the error procedure named *colourOf* in the error environment with the parameters supplied to the original call to obtain the integer result.

getCursor : **proc** (fd : **file** → **image**)

If the file specified by *fd* is a raster device, the procedure returns the cursor associated with that device.

If the file is not a raster file, a call will be made to the error procedure named *getCursor* in the error environment with the parameters supplied to the original call to obtain the image.

setCursor : **proc** (fd : **file** ; curs : **image**)

If the file specified by *fd* is a raster device, the cursor is set to be a copy of the image *curs*.

If the file is not a raster file, a call will be made to the error procedure named *setCursor* in the error environment with the parameters supplied to the original call.

getCursorInfo : **proc** (fd : **file** ; data : ***int**)

If the file specified by *fd* is a raster device, this procedure provides the cursor information for the device *fd*.

If the file is not a raster file, a call will be made to the error procedure named *getCursorInfo* in the error environment with the parameters supplied to the original call.

The elements of the vector are filled in as follows:

element 1: the cursor's X position,
element 2: the cursor's Y position,
element 3: the rasterop rule used to display the cursor

The interpretation of the rasterop rule may be found in the description of *rasterOp* in this Chapter.

If the vector has more than three elements, the extra ones are ignored. If the vector has less than three only the ones supplied are filled in.

setCursorInfo : **proc** (fd : **file** ; data : ***int**)

If the file specified by *fd* is a raster device, this procedure alters the cursor information for the device fd according to the contents of the vector.

If the file is not a raster file, a call will be made to the error procedure named *setCursorInfo* in the error environment with the parameters supplied to the original call.

The elements of the vector are interpreted as follows:

element 1: specifies the cursor's X position,
element 2: specifies the cursor's Y position,
element 3: specifies the rasterop rule used to display the cursor.

The interpretation of the rasterop rule may be found in the description of *rasterOp* in this Chapter.

If the vector has more than three elements, the extra ones are ignored. If the vector has less than three only the ones supplied are filled in.

16.5 User Environment

This environment is empty, it is hoped that users will store their environments here.

16.6 Vector Environment

lwb : **proc** [t] (V : *t → **int**)

This procedure returns the lower bound of the vector *V*.

upb : **proc** [t] (V : *t → **int**)

This procedure returns the upper bound of the vector *V*.

16.7 String Environment

length : **proc** (s : **string** → **int**)

This procedure returns the number of characters in the string *s*.

asciiToString : **proc** (i : **int** → **string**)

This procedure returns the single character string corresponding to the ASCII code of *i* **rem** 128.

stringToAscii : **proc** (s : **string** → **int**)

This procedure returns the ASCII code for the single character string s(1|1).

letter : **proc** (s : **string** → **bool**)

This procedure returns true if the first character of the string *s* is a letter. The test performed is:

```
s >= "a" and s <= "z" or s >= "A" and s <= "Z"
```

```
digit          : proc (s : string → bool)
```

This procedure returns true if the first character of the string *s* is a decimal digit. The test performed is:

```
s >= "0" and s <= "9"
```

16.8 IO Environment

The IO environment contains the following values:

```
PrimitiveIO    : env
...
```

The remainder of this environment is described in 16.8.2.

16.8.1 PrimitiveIO Environment

This environment maps the I/O facilities of UNIX onto the Napier88 system.

```
create         : proc (f : string ; m : int → file)
```

This procedure creates a file with name *f* in mode *m*, *m* is the decimal value of the (Unix) file protection bitmap.

If the create fails, **nilfile** is returned.

```
open           : proc (f : string ; m : int → file)
```

This procedure opens the file with name *f* in mode *m*. Legal modes are:

0 - read only,
1 - write only, and,
2 - read and write.

If the open fails, nilfile is returned.

```
close          : proc (f : file → int)
```

This procedure closes the file associated with file descriptor *f*. The integer returned is 0 if the operation was successful and -1 otherwise.

```
seek           : proc (f : file; offset, key : int → int)
```

This procedure sets the position of the next read or write from file *f* to be the position *offset*. The position to which the offset is relative is determined by *key*, which may have the following values:

0 - start of file,
1 - current position, or
2 - end of file.

The procedure returns the position in the file if the operation was successful and -1 otherwise.

ioctl : **proc** (fd : **file** ; data : ***int** ; command : **int** → **int**)

The ioctl commands for the first UNIX implementation correspond exactly to those supported by the UNIX ioctl system call. The ioctl instruction will not execute the specified command unless it is applicable to a 4.2BSD compatible terminal and the vector of integers contains sufficient integer elements to hold the parameters or results of the specified command. The supported commands (fully described in Section 4 of the 4.2BSD manual set) are:

TIOCSETP, TIOCSETN, TIOCSETC, TIOCSLTC, TIOCSETD, TIOCFLUSH, TIOCSTI, TIOCSPGRP, TIOCLBIS, TIOCLBIC, TIOCEXCL, TIOCNXCL, TIOCHPCL, TIOCSBRK, TIOCCBRK, TIOCSDTR, TIOCCDTR, TIOCSTOP, TIOCSTART, TIOCGETP, TIOCGETC, TIOCGLTC, TIOCGETD, TIOCGPRG, TIOCOUTQ, FIONREAD and FIONBIO.

In order to preserve the state of a terminal over a checkpoint, the terminal file descriptor records the 4 state structures used by 4.2BSD. These are the *sgttyb* structure, the *tchars* structure, the *ltchars* structure and a word of local flags. Whenever a specified command updates one of these state structures, the change is also recorded in the file descriptor.

readBytes : **proc** (fd : **file** ; buffer : ***int** ; start, length : **int** → **int**)

This procedure reads at most *length* bytes from the file *fd* into the vector of integers at position *start*. The position start specifies the byte offset from the start of the vector's elements. If the operation completes successfully, the procedure returns the number of bytes read and -1 otherwise.

writeBytes : **proc** (fd : **file** ; buffer : ***int** ; start, length : **int** → **int**)

This procedure writes at most *length* bytes to the file *fd* from the vector of integers at position *start*. The position *start* specifies the byte offset from the start of the vector's elements. If the operation completes successfully, the procedure returns the number of bytes written and -1 otherwise.

getByte : **proc** (word, index : **int** → **int**)

This procedure returns the byte at position *index* from the integer *word.* This procedure will call the *getByte* procedure in the error environment if an illegal index is used.

setByte : **proc** (word, index, value : **int** → **int**)

The result of this procedure is the integer formed by taking the integer *word* and replacing the byte at position *index* by *value.* This procedure will call the *setByte* procedure in the error environment if an illegal index is used.

errorNumber : **proc** (→ **int**)

This procedure returns the error number of the last primitive I/O operation. The error numbers are those returned by the last Unix I/0 operation and may be found in intro(2) in the SUNOS 4.0 Unix Manual.

16.8.2 The IO Environment

The rest of the I/O environment contains the following:

stdOut : **file**

This is a file variable that is initially connected to the Unix control terminal for the Napier88 system.

writeByte : **proc** (i : **int**)

This procedure writes bitwiseAnd (i,255) as a byte to the file *stdOut*. This procedure may call the *writeByte* error procedure in the error environment if an error occurs.

writeString : **proc** (s : **string**)

This procedure writes the string *s* to the file *stdOut*. This procedure may call the *writeString* error procedure in the error environment if an error occurs.

writeBool : **proc** (b : **bool**)

This procedure writes the boolean *b* to the file *stdOut*. This procedure may call the *writeBool* error procedure in the error environment if an error occurs.

writeInt : **proc** (i : **int**)

This procedure writes the integer *i* to the file *stdOut*. This procedure may call the *writeInt* error procedure in the error environment if an error occurs.

writeReal : **proc** (r : **real**)

This procedure writes the real *r* to the file *stdOut*. This procedure may call the *writeReal* error procedure in the error environment if an error occurs.

integerWidth : **int**

Integers written out using *writeInt* will be displayed, left justified, in this number of characters. If the number does not fit within this space, the exact number of characters is used. *integerWidth* is a variable with an initial value of 12.

realWidth : **int**

Reals written out using *writeReal* will be displayed, left justified, in this number of characters. If the number does not fit within this space, the exact number of characters is used. *realWidth* is a variable with an initial value of 14.

spaceWidth : **int**

spaceWidth spaces are written out after any integer or real number written using *writeInt* or *writeReal*. *spaceWidth* is a variable with an initial value of 2.

makeWriteEnv : **proc** (f : **file** → **env**)

This procedure creates an environment that contains the procedures *writeByte*, *writeString*, *writeBool*, *writeInt* and *writeReal*, each of which operates on the file *f* instead of the file *stdOut*. Each procedure may call the error procedures described above. The environment also contains the variables *integerWidth*, *realWidth* and *spaceWidth*, to control the operation of *writeInt* and *writeReal* on the file *f*. The initial values of the three variables are 12, 14 and 2 respectively.

stdIn : **file**

This is a file variable that is initially connected to the Unix control terminal for the Napier88 system.

endOfInput : **proc** (→ **bool**)

This procedure reads one 8 bit byte as an integer from the file *stdIn*. If the read is successful, false is returned. If an I/O error occurs this procedure will call the *endOfInputIOE* procedure in the IO Errors environment. If the end of input is encountered, this procedure will return true. The byte read is made available to the next *endOfInput, readByte*, *readChar*, *peekByte, peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the byte cannot be made available, a call is made to the *endOfInputUnread* procedure in the IO Errors environment. The result obtained from any of the error procedures is returned as the result of the *endOfInput* procedure.

readByte : **proc** (→ **int**)

This procedure reads one 8 bit byte as an integer from the file *stdIn*. If an I/O error occurs this procedure will call the *readByteIOE* procedure in the IO Errors environment. If the end of input is encountered, this procedure will call the *readByteEOI* procedure in the IO Errors environment. The result obtained from either of the error procedures is returned as the result of the *readByte* procedure.

readChar : **proc** (→ **string**)

This procedure reads one character from the file *stdIn.* If the end of input is encountered during this operation, this procedure will call the *readCharEOI* procedure in the IO Errors environment. If an I/O error occurs, this procedure will call the *readCharIOE* procedure in the IO Errors environment. The result obtained from either of the error procedures is returned as the result of the *readChar* procedure.

peekByte : **proc** (→ **int**)

This procedure reads one 8 bit byte as an integer from the file *stdIn*. If an I/O error occurs this procedure will call the *peekByteIOE* procedure in the IO Errors environment. If the end of input is encountered, this procedure will call the *peekByteEOI* procedure in the IO Errors environment. The byte read is made available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the byte cannot be made available, a call is made to the *peekByteUnread* procedure in the IO Errors environment. The result obtained from any of the error procedures is returned as the result of the *peekByte* procedure.

peekChar : **proc** (→ **string**)

This procedure reads one character from the file *stdIn.* If the end of input is encountered during this operation this procedure will call the *peekCharEOI* procedure in the IO Errors environment. If an I/O error occurs, this procedure will call the *peekCharIOE* procedure in the IO Errors environment. The character read is made available to the next *endOfInput*, *readByte*, *readChar*, *peekByte*, *peekChar*, *readString*, *readLine*, *readBool*, *readInt* or *readReal* operation. If the character cannot be made available, a call is made to the *peekByteUnread* procedure in the IO Errors environment. The result obtained from any of the error procedures is returned as the result of the *peekChar* procedure.

readString : **proc** (→ **string**)

This procedure ignores the layout characters " ","'t" and "'n" and reads a string literal (a string in quotes) from the file *stdIn.*

If the first character after any layout characters is not a double quote, this procedure will call the *readStringBadChar* procedure in the IO Errors environment. The erroneous character will have been read. If an I/O error occurs, this procedure will call the *readStringIOE* procedure in the IO Errors environment. If the end of input is encountered during this operation, this procedure will call the *readStringEOI* procedure in the IO Errors environment. The result obtained from any of the error procedures is returned as the result of the *readString* procedure.

readLine : **proc** (→ **string**)

This procedure reads from the file *stdIn* up to and including a newline character. It concatenates and returns the characters as a string without the newline character. If the end of input is encountered during this operation, this procedure will call the *readLineEOI* procedure in the IO Errors environment. If an I/O error occurs, this procedure will call the *readLineIOE* procedure in the IO Errors environment. The result obtained from either of the error procedures is returned as the result of the *readLine* procedure.

readBool : **proc** (→ **bool**)

This procedure ignores the layout characters " ","'t" and "'n" and reads one boolean from the file *stdIn*. If the characters after any layout characters do not form a boolean, this procedure will call the *readBoolBadChar* procedure in the IO Errors environment. The file *stdIn* will have been read up to and including the first erroneous character. If an I/O error occurs, the procedure *readBool* will call the *readBoolIOE* procedure in the IO Errors environment. If the end of input is encountered during this operation this procedure will call the *readBoolEOI* procedure in the IO Errors environment. The result obtained from any of the error procedures is returned as the result of the *readBool* procedure.

readInt : **proc** (→ **int**)

This procedure ignores the layout characters " ","'t" and "'n" and reads one integer from the file *stdIn.* If the first character after any layout characters is not a digit or a sign which is followed by a digit, this procedure will call the *readIntBadChar* procedure in the IO Errors environment. The erroneous character will have been read. If the end of input is encountered before the first digit, this procedure will call the *readIntEOI* procedure in the IO Errors environment. If an I/O error occurs, the procedure *readInt* will call the *readIntIOE* procedure in the IO Errors environment.

This procedure reads characters from the file *stdIn* until it has parsed an integer. The parsing may involve reading the first character following the integer. When this occurs the extra character is made available to the next *endOfInput, readByte, readChar, peekByte, peekChar, readString, readLine, readBool, readInt* or *readReal* operation. If the character cannot be made available, a call is made to the *readIntUnread* procedure in the IO Errors environment.

When an integer has been successfully parsed it is converted into an integer value. If an arithmetic error occurs during the conversion, a call is made to the *readIntOverflow* procedure in the IO Errors environment.

The result obtained from any of the error procedures *readIntBadChar, readIntEOI, readIntIOE, readIntUnread* or *readIntOverflow* is returned as the result of the *readInt* procedure.

readReal : **proc** (→ **real**)

This procedure ignores the layout characters " ","'t" and "'n" and reads one real from the file *stdIn.* If the first character after any layout characters is not a digit or a sign which is followed by a digit, this procedure will call the *readRealBadChar* procedure in the IO Errors environment. The erroneous character will have been read. If the end of input is encountered before the first digit, this procedure will call the *readRealEOI* procedure in the IO Errors environment. If an I/O error occurs, the procedure *readReal* will call the *readRealIOE* procedure in the IO Errors environment.

This procedure reads characters from the file *stdIn* until it has parsed a real. The parsing may involve reading the first character following the real. When this occurs the extra character is made available to the next *endOfInput, readByte, readChar, peekByte, peekChar, readString, readLine, readBool, readInt* or *readReal* operation. If the character cannot be made available, a call is made to the *readRealUnread* procedure in the IO Errors environment.

When a real has been successfully parsed it is converted into a real value. If an arithmetic error occurs during the conversion, a call is made to the *readRealOverflow* procedure in the IO Errors environment.

The result obtained from any of the error procedures *readRealBadChar, readRealEOI, readRealIOE, readRealUnread* or *readRealOverflow* is returned as the result of the *readReal* procedure.

makeReadEnv : **proc** (f : **file** → **env**)

This procedure creates an environment that contains the procedures *endOfInput, readByte, readChar, peekByte, peekChar, readString, readLine, readBool, readInt* and *readReal*, each of which operates on the file *f* instead of the file *stdIn.* Each of these procedures may call the error procedures described above.

16.9 Format Environment

iformat : **proc** (i : **int** → **string**)

This procedure returns the integer *i* represented as a string.

fformat : **proc** (r : **real** ; w, d : **int** → **string**)

This procedure returns the string representing *r* with *w* digits before the decimal point and *d* digits after it. This procedure calls the procedure called *fformat* in the error environment if there are more than *w* digits before the "." or if *d* is a negative number. In which case, the value returned by the error procedure is returned by this procedure.

eformat : **proc** (r : **real** ; w, d : **int** → **string**)

This procedure returns the string representing *r* with *w* digits before the decimal point and *d* digits after with an exponent. This procedure calls the procedure called *eformat* in the error environment if *w* or *d* is a negative number. In which case, the value returned by the error procedure is returned by this procedure.

gformat : **proc** (r : **real** → **string**)

This procedure returns the string representing *r* in *eformat* or *fformat* whichever is most suitable. It does this by calling one of the procedures *eformat* or *gformat* above.

16.10 Arithmetical Environment

maxint : **int**

The maximum integer possible in the implementation.

maxreal : **real**

The maximum real possible in the implementation.

pi : **real**

The value of π in the implementation.

epsilon : **real**

The largest value, ε, such that $1.0 + \varepsilon = 1.0$ in the implementation.

sqrt : **proc** (x : **real** → **real**)

This procedure returns the positive square root of *x* when $x \geq 0$. On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

exp : **proc** (x : **real** → **real**)

This procedure returns e to the power *x*. On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

ln : **proc** (x : **real** → **real**)

This procedure returns the logarithm of *x* to the base e where $x > 0$.

On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

sin : **proc** (x : **real** → **real**)

This procedure returns the sine of *x* (given in radians). On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

cos : **proc** (x : **real** → **real**)

This procedure returns the cosine of *x* (given in radians). On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

atan : **proc** (x : **real** → **real**)

This procedure returns the arctangent of *x* (given in radians) where - pi / 2 < atan (x) < pi / 2

On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

truncate : **proc** (x : **real** → **int**)

This procedure returns the integer *i* such that |i| ≤ |x| < |i| + 1 where i * x ≥ 0.

On an error, this procedure calls *unaryReal* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryReal* is used as the result of this procedure.

rabs : **proc** (x : **real** → **real**)

This procedure returns the absolute value of real number *x*.

abs : **proc** (x : **int** → **int**)

This procedure returns the absolute value of *x*. On an error, that is if x = -maxint -1, this procedure calls *unaryInt* in the error environment with *x* as a parameter. In this case, the result obtained from the call of *unaryInt* is used as the result of this procedure.

float : **proc** (x : **int** → **real**)

This procedure returns *x* expressed as a real number.

shiftRight : **proc** (value, count : **int** → **int**)

This procedure returns the first parameter shifted right *count* bit places bringing in zeros at the high order end.

shiftLeft : **proc** (value, count : **int** → **int**)

This procedure returns the first parameter shifted left *count* bit places bringing in zeros at the low order end.

bitwiseAnd : **proc** (value1, value2 : **int** → **int**)

This procedure returns the logical (bitwise) 'and' of *value1* and *value2.*

bitwiseOr : **proc** (value1, value2 : **int** → **int**)

This procedure returns the logical (bitwise) 'or' of *value1* and *value2.*

bitwiseNot : **proc** (value : **int** → **int**)

This procedure returns the bitwise not of *value.*

16.11 Graphical Environment

The graphics environment contains the following objects:

Outline : **env**
Raster : **env**

These environments are described in the following sections.

16.11.1 Outline Environment

This environment contains the implementation and user procedures for Outline graphics [MOR82, MBD86b]. The environment contains three entries: *Odraw*, *makeErase* and *makeDraw*.

```
Odraw : proc ( point : proc (int, int, int) ;
               join : proc (int, int, int, int, int) ;
               checkColour : proc (pixel → int) ;
               defaultColour : int ;
               Clipx1, Clipx2, Clipy1, Clipy2 : int ;
               t : pic ;
               xmin, xmax, ymin, ymax : real)
```

This procedure implements the Outline graphics. It is used by the Napier88 system.

```
type drawFunction is variant (imageDraw : proc (image, pic, real, real, real, real) ;
                              fileDraw : proc (file, pic, real, real, real, real) ;
                              fail : null)

makeDrawFunction : proc (devType : string → drawFunction)
```

This procedure returns an Outline draw function for the device type specified by *devType*. If the device specified is unknown to the procedure, the fail branch of the variant is returned. The devices supported by the first Napier88 release are:

"image"	Napier88 raster image
"t4010"	Tektronix 4010
"t4006"	Tektronix 4006
"t4107"	Tektronix 4107
"cx4800"	cx4800 printer
"L800"	QMS laserprinter
"g6320"	colour plotter

The outline draw function returned by *makeDraw* takes a picture and a bounding rectangle in the infinite two dimensional real space over which all pictures are defined. The picture is clipped to the area of the bounding box. The box is then scaled and shifted to fit the area of the device on which it is drawn.

The four real parameters to the draw function describe the bounding box and correspond to the xmin, xmax, ymin, ymax parameters to *Odraw*, respectively. If the xmin and xmax or ymin and ymax parameters are equal then the bounding box has a zero dimension, and a call is made to the *Draw* procedure in the graphical errors environment.

If the picture being drawn contains a text statement whose end points are coincident, a call is made to the *Text* procedure in the graphical errors environment. The *Text* procedure may be used to construct a picture to replace the erroneous text statement.

The mapping of a picture onto a device is performed using real arithmetic which, in certain circumstances, may result in arithmetic errors. If any arithmetic errors do

occur, the appropriate procedure in the arithmetical errors environment will be called.

```
type eraseFunction is variant (     imageErase : proc (image, pixel) ;
                                    fileErase : proc (file, pixel) ;
                                    fail : null)

makeEraseFunction   : proc (devType : string → eraseFunction)
```

This procedure returns an Outline erase procedure for the device type specified by *devType*. If the device specified is unknown to the procedure the fail branch of the variant is returned, otherwise a procedure is returned that may be used to clear the display of the device.

16.11.2 Raster Environment

This environment contains the implementation and the user procedures for the Napier88 raster graphics [MBD86a].

```
xDim        : proc (i : image → int)
```

This procedure returns the X dimension of the image *i*.

```
yDim        : proc (i : image → int)
```

This procedure returns the Y dimension of the image *i*.

```
zDim        : proc (i : image → int)
```

This procedure returns the number of planes in the image *i*.

```
pixelDepth  : proc (p : pixel → int)
```

This procedure returns the number of planes in the pixel *p*.

```
rasterOp    : proc (S, D : image ; r: int)
```

Performs a raster operation from *S* (source) onto *D* (destination) using the combination rule *r* **rem** 16 which is interpreted as follows:

0	S **and** ~S	8	S **and** D
1	~ (S **or** D)	9	~S **xor** D
2	~S **and** D	10	D
3	~S	11	~S **or** D
4	S **and** ~D	12	S
5	~D	13	S **or** ~D
6	S **xor** D	14	S **or** D
7	~ (S **and** D)	15	S **or** ~S

where **on** maps to **true** and **off** maps to **false**.

```
line        : proc (i : image ; x1, y1, x2, y2 : int ; p : pixel ; r : int)
```

This procedure draws a line from the point (x1, y1) to the point (x2, y2) on the image *i*. The line is drawn with the pixel *p* which is combined with the image pixels using the raster rule specified by *r*. The interpretation of the raster rule may be found in *rasterOp* above.

getPixel : **proc** (i : **image** ; x, y : **int** → **pixel**)

This procedure returns the pixel found at position (x, y) in image *i*. This procedure calls the procedure of the same name in the error environment if an illegal index is specified.

setPixel : **proc** (i : **image** ; x, y : **int** ; p : **pixel**)

This procedure sets the pixel at position (x, y) in image *i* to be *p*.This procedure calls the procedure of the same name in the error environment if an illegal index is specified.

16.12 Font Environment

The font environment contains the following items:

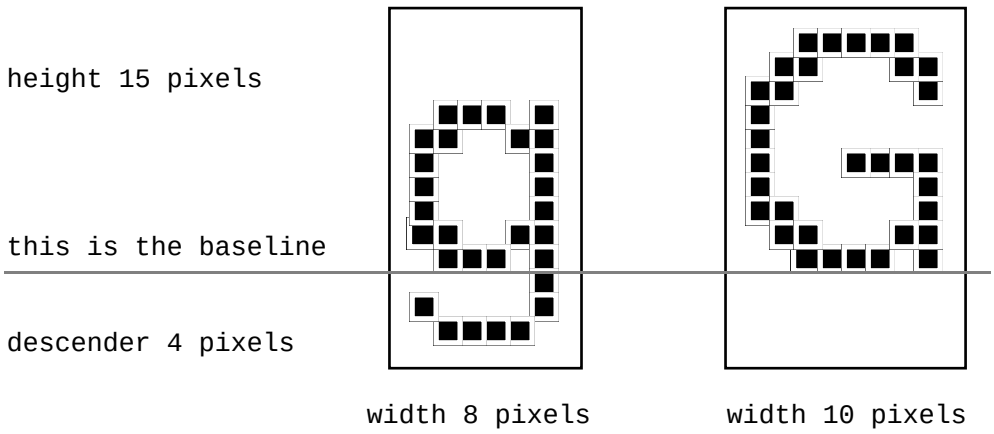
cmrB14, courR10, screenB14, screenR7,
cmrR14, courR12, screenR11, serifR10,
courB10, courR14, screenR12, serifR11,
courB12, gallantR19, screenR13, serifR12,
courB14, screenB12, screenR14, serifR14,
serifR16 : FontPack

Each of these is of type *FontPack* which is defined below.

```
type Font is structure (constant characters : *image;
                        constant fontHeight : int;
                        constant descender : int;
                        constant info : string)

type FontPack is structure (font : Font;
                            stringToTile, charToTile : proc (string → image))
```

The *fontHeight* is the height of the characters in the font. The *descender* is the distance from the bottom of an image holding a character to the base line. The widths of characters in a font may vary, but the programmer may examine these by taking the x dimension of the appropriate image. For example,



The vector called *characters* contains font images, so they may be indexed by the ASCII character code. Information about the font may be found in the string *info*.

The procedures *charToTile* and *stringToTile* found in each *FontPack* take as a parameter a string and return a representation of the same string contained in an image. The procedure *charToTile* works for single character strings and gives back

the actual image kept in the font, rather than a copy of it. This is provided as an optimisation.

16.13 Error Environment

The Error environment contains the following items:

```
Arithmetical    : env
Graphical       : env
String          : env
Structure       : env
Vector                  : env
Variant                 : env
Environment     : env
IO              : env
Format                  : env
```

The procedures that are called in the event of an error are stored in these environments. Each procedure is a variable and the user may change them by assignment. By default, all the error procedures write out an appropriate error message and halt the execution of the Napier88 system.

16.13.1 Arithmetical Errors Environment

```
unaryInt        : proc (rator : string ; rand : int → int)
```

This procedure may be called during unary minus and abs operations with the *rator* parameter being "-" and "abs", respectively. The *rand* parameter is the integer on which the original operation was attempted.

```
Int             : proc (rator : string ; rand1, rand2 : int → int)
```

This procedure may be called during the operations *plus*, *times*, *minus*, *div* and *rem* with the rator parameter being "+", "*", "-", "div" and "rem", respectively. In each case, the parameters *rand1* and *rand2* are the integers on which the original operation was attempted.

```
unaryReal       : proc (rator : string ; rand : real → real)
```

This procedure may be called during the operations unary minus, *sin*, *cos*, *exp*, *ln*, *sqrt*, *atan* and *truncate* with the *rator* parameter being "-", "sin", "cos", "exp", "ln", "sqrt", "atan" and "truncate", respectively. In each case, the parameter *rand* is the real number on which the original operation was attempted.

```
Real            : proc (rator : string ; rand1, rand2 : real → real)
```

This procedure may be called during the operations *plus*, *times*, *minus* and *divide* with the *rator* parameter being "+", "*", "-" and "/", respectively. In each case, the parameters *rand1* and *rand2* are the real numbers on which the original operation was attempted.

```
getByte         : proc (index, word : int → int)
```

This procedure may be called during the execution of the *getByte* procedure which may be found in the PrimitiveIO environment. It is called when the byte index supplied to the *getByte* procedure is not between 0 and 3. The parameters to this error procedure are those supplied to the original *getByte* procedure call.

setByte : **proc** (index, word, byte : **int** → **int**)

This procedure may be called during the execution of the *setByte* procedure which may be found in the PrimitiveIO environment. It is called when the byte index supplied to the *setByte* procedure is not between 0 and 3. The parameters to this error procedure are those supplied to the original *setByte* procedure call.

16.13.2 Graphical Errors Environment

Draw : **proc** (p : **pic** ; xmin, xmax, ymin, ymax : **real**)

This procedure is called if either the *xmin* or *xmax* parameters to *Odraw* or the *ymin* and *ymax* parameters to *Odraw* are equal.

Text : **proc** (s : **string** ; x1, y1, x2, y2 : **real** → **pic**)

This procedure is called when a picture is created for a text statement whose end points [x1, y1] and [x2, y2] are coincident.

getPixel : **proc** (i : **image** ; x, y : **int** → pixel)

This procedure may be called during the *getPixel* procedure which may be found in the raster graphics environment. It is called whenever the *x* or *y* coordinates supplied to that procedure lie outwith the bounds of the image supplied as a parameter. The parameters to this error procedure are those supplied to the original *getPixel* procedure call.

setPixel : **proc** (i : **image** ; x, y : **int** ; p : **pixel**)

This procedure may be called during the *setPixel* procedure which may be found in the raster graphics environment. It is called whenever the *x* or *y* coordinates supplied to that procedure lie outwith the bounds of the image supplied as a parameter. The parameters to this error procedure are those supplied to the original *setPixel* procedure call.

pixelOverflow : **proc** (p : **pixel** → **pixel**)

This procedure may be called during the pixel concatenation operation ++. It is called if the pixel depth overflows the implementation size. The parameter is the first 24 planes.

subPixel : **proc** (p : **pixel** ; start, noOfPlanes : **int** → **pixel**)

This procedure may be called during the pixel indexing operation. It is called if the *start* plane is less than zero, if the *start* plane is greater than or equal to the pixel depth, if the number of planes requested are less than 1 or if the planes selected are not a subset of the original pixel. The original pixel, the start plane and the number of planes are supplied to the error procedure as parameters.

makeImage : **proc** (x, y : **int** ; p : **pixel** → **pixel**)

This procedure may be called during the image creation operation. It is called if either the *x* or the *y* dimension is less than 1. The parameters to the error procedure are the *x* and *y* dimension and the initialising pixel.

subImage : **proc** (i : **image** ; start, noOfPlanes : **int** → **image**)

This procedure may be called during the image indexing operation. It is called if the *start* plane is less than zero, if the *start* plane is greater than or equal to the image depth, if the number of planes requested are less than 1 or if the planes selected are not a subset of the original image. The original image, the start plane and the number of planes are supplied to the error procedure as parameters.

limitAt : **proc** (i : **image** ; x, y : **int** → **image**)

This procedure may be called during the '**limit** i **at** x,y' operation. It is called if $x < 0$ or $x \geq xDim$ (i) or $y < 0$ or $y \geq yDim$ (i). The image on which the operation was attempted and the x and y indexing coordinates are supplied as parameters to the error procedure.

limitAtBy : **proc** (i : **image** ; x1, x2, y1, y2 : **int** → **image**)

This procedure may be called during the '**limit** i **to** x1 **by** y1 **at** x2,y2' operation. It is called if $x2 < 0$ or $x2 \geq xDim$ (i) or $y2 < 0$ or $y2 \geq yDim$ (i) or if the sub image requested is not totally enclosed within the original image. The image on which the operation was attempted and the indexing coordinates are supplied as parameters to the error procedure.

imagePixelConstant: **proc** (i : **image**)

This procedure is called if an update operation (via a raster operation) is attempted on an image of constant pixels. The image on which the operation was attempted is supplied as a parameter to the error procedure.

getScreen : **proc** (f : **file** → **image**)

This procedure is called by the *getScreen* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file on which the original *getScreen* operation was attempted.

locator : **proc** (f : **file** ; vec : ***int**)

This procedure is called by the *locator* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file and the vector with which the original *locator* operation was attempted.

colourMap : **proc** (f : **file** ; p : **pixel**, ; i : **int**)

This procedure is called by the *colourMap* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file, the pixel and the integer passed as parameters to the original *colourMap* operation.

colourOf : **proc** (f : **file** ; p : **pixel** → **int**)

This procedure is called by the *colourOf* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file and the pixel passed as parameters to the original *colourOf* operation.

getCursor : **proc** (f : **file** → **image**)

This procedure is called by the *getCursor* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file passed as a parameter to the original *getCursor* operation.

setCursor : **proc** (f : **file** ; i : **image**)

This procedure is called by the *setCursor* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file and the image passed as a parameter to the original *setCursor* operation.

getCursorInfo : **proc** (f : **file** ; vec : ***int**)

This procedure is called by the *getCursorInfo* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file and the vector passed as a parameter to the original *getCursorInfo* operation.

setCursorInfo : **proc** (f : **file** ; vec : ***int**)

This procedure is called by the *setCursorInfo* procedure found in the Device environment if the file supplied to it is not a raster device. The error procedure is passed the file and the vector passed as a parameter to the original setCursorInfo operation.

16.13.3 String Errors Environment

concatenate : **proc** (s1, s2 : **string** → **string**)

This procedure is called by the ++ operator whenever the new string would exceed the maximum string length (the maximum string length is maxint characters in this implementation of Napier88). The error procedure is passed as parameters the two strings which were to be concatenated.

subString : **proc** (s : **string** ; start, length : **int** → **string**)

This procedure is called by the substring operator |. It is called if the string to be dereferenced is a nilstring or if the *start* position is less than 1 or if the *length* is less than 0 or if the finish position is after the end of the string. The error procedure is passed as parameters the original string, the start position and the substring length.

16.13.4 Structure Errors Environment

structureFieldConstant : **proc** ()

This procedure is called when an assignment is attempted to a constant field within a structure.

16.13.5 Vector Errors Environment

vectorElementConstant: **proc**[t] (vec : *t ; index : **int** ; value : t)

This procedure is called when an assignment is attempted to a location within a vector of constant elements. The error procedure is passed as parameters the vector, the index and the value which was to be assigned to the location.

vectorIndexAssign : **proc**[t] (vec : *t ; index : **int** ; value : t)

This procedure is called whenever an assignment is attempted to an illegal address within a vector. An illegal address could be an address less than the lower bound of the vector or an address greater than the upper bound of the vector. The error

procedure is passed the original vector, the index and the value to be assigned to the location.

vectorIndexSubs : **proc**[t] (vec : *t ; index : **int** → t)

This procedure is called whenever an illegal index is made into a vector. An illegal index could be an address less than the lower bound of the vector or an address greater than the upper bound of the vector. The error procedure is passed the original vector and the index.

makeVector : **proc**[t] (lwb, upb : int ; initial : t → *t)

This procedure is called whenever a vector creation is attempted with the lower bound greater than the upper bound. The error procedure is passed the lower bound, the upper bound and the initialising value.

16.13.6 Variant Errors Environment

varProject : **proc** (t : typeRep ; expected, was : **string**)

This procedure is called when a variant is illegally dereferenced. The type *typeRep* is defined as follows:

type typeRep **is structure** (unboundQuantifiers : **bool** ; representation : **string**)

For example, with the variant,

rec type list **is variant** (cell : **structure** (hd : **int** ; tl : list) ; tip : **null**)

and the value,

let a = list (tip : **nil**)

then the illegal projection,

a'cell

would result in the following call of this procedure:

```
varProject ( typeRep( false,
        "rec type t0 is variant (cell : t1 ; tip : null)
        &       t1 is structure (hd : int ; tl : t0)"), "cell", "tip")
```

It should be noted that after this error the Napier88 system cannot continue and will halt, even if the error handling procedure should return normally.

16.13.7 Environment Errors Environment

envRedeclaration : **proc** (**env**, **string**, typeRep, **bool**)

This procedure is called if a binding is added to an environment that already contains the identifier. The procedure is passed the environment being updated, the name of the identifier being declared, a representation of the new type and a flag to indicate whether or not the identifier was to be constant.

envProject : **proc** (**env**, **string**, typeRep, **bool**)

This procedure is called if a signature in an environment projection cannot be matched by the environment. The procedure is passed the environment being searched, the identifier being searched for, a representation of the expected type and a flag to indicate whether or not the identifier was expected to be constant.

envDrop : **proc** (**env**, **string**)

This procedure is called if an attempt is made to drop an unbound identifier from an environment. The procedure is passed the environment being updated and the identifier that was to be dropped.

16.13.8 IO Errors Environment

writeByte : **proc** (f : **file** ; theByte, errorNo : **int**)

This procedure is called when an error occurs in the *writeByte* procedure in the IO environment. The procedure is passed the byte being written, the file being written to, and the I/O error number indicating the error.

writeString : **proc** (f : **file** ; s : **string** ; next, errorNo : **int**)

This procedure is called when an error occurs in the *writeString* procedure in the IO environment. The procedure is passed the string being written, the file being written to, the number of characters successfully written and the I/O error number indicating the error.

writeBool : **proc** (f : **file** ; s : **string** ; next, errorNo : **int**)

This procedure is called when an error occurs in the *writeBool* procedure in the IO environment. The procedure is passed the string representation of the boolean being written, the file being written to, the number of characters successfully written and the I/O error number indicating the error.

writeInt : **proc** (f : **file** ; s : **string** ; next, errorNo : **int**)

This procedure is called when an error occurs in the *writeInt* procedure in the IO environment. The procedure is passed the string representation of the integer being written, the file being written to, the number of characters successfully written and the I/O error number indicating the error.

writeReal : **proc** (f : **file** ; s : **string** ; next, errorNo : **int**)

This procedure is called when an error occurs in the *writeReal* procedure in the IO environment. The procedure is passed the string representation of the real being written, the file being written to, the number of characters successfully written and the I/O error number indicating the error.

endOfInputIOE : **proc** (f : **file** ; errorNo : **int** → **bool**)

This procedure is called when an error occurs in the *endOfInput* procedure in the IO environment. The procedure is passed the file being read and the I/O error number.

readByteIOE : **proc** (f : **file** ; errorNo : **int** → **int**)

This procedure is called when an error occurs in the *readByte* procedure in the IO environment. The procedure is passed the file being read and the I/O error number.

readByteEOI : **proc** (f : **file** → **int**)

This procedure is called when end of input is encountered in the *readByte* procedure in the IO environment. The procedure is passed the file being read.

readCharIOE : **proc** (f : **file** ; errorNo : **int** → **string**)

This procedure is called when an error occurs in the *readChar* procedure in the IO environment. The procedure is passed the file being read and the I/O error number.

readCharEOI : **proc** (f : **file** → **string**)

This procedure is called when end of input is encountered in the *readChar* procedure in the IO environment. The procedure is passed the file being read.

peekByteIOE : **proc** (f : **file** ; errorNo : **int** → **int**)

This procedure is called when an error occurs in the *peekByte* procedure in the IO environment. The procedure is passed the file being read and the I/O error number.

peekByteEOI : **proc** (f : **file** → **int**)

This procedure is called when end of input is encountered in the *peekByte* procedure in the IO environment. The procedure is passed the file being read.

endOfInputUnread : **proc** (f : **file** ; theByte, errorNo : **int** → **int**)

This procedure is called when the byte read by the *endOfInput* procedure in the IO environment cannot be made available to the next read operation on the same file. The procedure is passed the file being read, the byte read and the I/O error number.

peekByteUnread : **proc** (f : **file** ; theByte, errorNo : **int** → **int**)

This procedure is called when the byte read by the *peekByte* procedure in the IO environment cannot be made available to the next read operation on the same file. The procedure is passed the file being read, the byte read and the I/O error number.

peekCharIOE : **proc** (f : **file** ; errorNo : **int** → **string**)

This procedure is called when an error occurs in the *peekChar* procedure in the IO environment. The procedure is passed the file being read and the I/O error number.

peekCharEOI : **proc** (f : **file** → **string**)

This procedure is called when end of input is encountered in the *peekChar* procedure in the IO environment. The procedure is passed the file being read.

peekCharUnread : **proc** (f : **file** ; theByte,errorNo : **int** → **string**)

This procedure is called when the character read by the *peekChar* procedure in the IO environment cannot be made available to the next read operation on the same file. The procedure is passed the file being read, the byte corresponding to the character read and the I/O error number.

readBoolIOE : **proc** (f : **file** ; readSoFar : **string** ; errorNo : **int** → **bool**)

This procedure is called when an error occurs in the *readBool* procedure in the IO environment. The procedure is passed the file being read, the characters that had been read when the error occurred excluding any layout characters and the I/O error number.

readBoolEOI : **proc** (f : **file** ; readSoFar : **string** → **bool**)

This procedure is called when end of input is encountered in the *readBool* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read when the end of input was detected excluding any layout characters.

readBoolBadChar : **proc** (f : **file** ; readSoFar : **string** → **bool**)

This procedure is called when an erroneous character is read by the *readBool* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read up to and including the erroneous character excluding any layout characters.

readStringIOE : **proc** (f : **file** ; readSoFar : **string** ; errorNo : **int** → **string**)

This procedure is called when an error occurs in the *readString* procedure in the IO environment. The procedure is passed the file being read, the characters that had been read, excluding the leading double quote, when the error occurred and the I/O error number.

readStringEOI : **proc** (f : **file** ; readSoFar : **string** → **string**)

This procedure is called when end of input is encountered in the *readString* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read, excluding the leading double quote, when the end of input was detected.

readStringBadChar : **proc** (f : **file** ; readSoFar : **string** → **string**)

This procedure is called when a double quote character is not the first non layout character found by the *readString* procedure in the IO environment. The procedure is passed the file being read and the character that was read instead of the expected double quote.

readLineIOE : **proc** (f : **file** ; readSoFar : **string** ; errorNo : **int** → **string**)

This procedure is called when an error occurs in the *readLine* procedure in the IO environment. The procedure is passed the file being read, the characters that had been read when the error occurred and the I/O error number.

readLineEOI : **proc** (f : **file** ; readSoFar : **string** → **string**)

This procedure is called when end of input is encountered in the *readLine* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read when the end of input was detected.

readIntIOE : **proc** (f : **file** ; readSoFar : **string** ; errorNo : **int** → **int**)

This procedure is called when an error occurs in the *readInt* procedure in the IO environment. The procedure is passed the file being read, the characters that had been read when the error occurred excluding any layout characters and the I/O error number.

readIntEOI : **proc** (f : **file** ; readSoFar : **string** → **int**)

This procedure is called when end of input is encountered in the *readInt* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read when the end of input was detected excluding any layout characters.

readIntBadChar : **proc** (f : **file** ; readSoFar : **string** → **int**)

This procedure is called when a digit or a sign which is followed by a digit, is not the first non layout character found by the *readInt* procedure in the IO environment. The procedure is passed the file being read and the character that was read instead of the expected digit or sign which is followed by a digit.

readIntUnread : **proc** (f : **file** ; theInteger : **string** ; theByte, errorNo : **int** → **int**)

This procedure is called when an extra character read while parsing an integer in the *readInt* procedure in the IO environment cannot be made available to the next read operation. The procedure is passed the file being read, the characters that form the integer, the byte corresponding to the extra character that was read and the I/O error number.

readIntOverflow : **proc** (f : **file** ; theInteger : **string** → **int**)

This procedure is called when an arithmetic error occurs converting the integer parsed in the *readInt* procedure in the IO environment into an integer value. The procedure is passed the file being read and the characters that form the integer.

readRealIOE : **proc** (f : **file** ; readSoFar : **string** ; errorNo : **int** → **real**)

This procedure is called when an error occurs in the *readReal* procedure in the IO environment. The procedure is passed the file being read, the characters that had been read when the error occurred excluding any layout characters and the I/O error number.

readRealEOI : **proc** (f : **file** ; readSoFar : **string** → **real**)

This procedure is called when end of input is encountered in the *readReal* procedure in the IO environment. The procedure is passed the file being read and the characters that had been read when the end of input was detected excluding any layout characters.

readRealBadChar : **proc** (f : **file** ; readSoFar : **string** → **real**)

This procedure is called when a digit or a sign which is followed by a digit, is not the first non layout character found by the *readReal* procedure in the IO environment. The procedure is passed the file being read and the character that was read instead of the expected digit or sign which is followed by a digit.

readRealUnread : **proc** (f : **file** ; theReal : **string** ; theByte, errorNo : **int** → **real**)

This procedure is called when an extra character read while parsing a real in the *readReal* procedure in the IO environment cannot be made available to the next read operation. The procedure is passed the file being read, the characters that form the real, the byte corresponding to the extra character that was read and the I/O error number.

readRealOverflow : **proc** (f : **file** ; theReal : **string** → **real**)

This procedure is called when an arithmetic error occurs converting the integer parsed in the *readReal* procedure in the IO environment into a real value. The procedure is passed the file being read and the characters that form the real.

16.13.9 Format Errors Environment

fformat : **proc** (r : **real** ; ipart, fpart : **int** → **string**)

This procedure is called by the *fformat* procedure in the Format environment. It is called if there are more than *ipart* places before the "." or if *fpart* is a negative number.

eformat : **proc** (r : **real** ; ipart, fpart : **int** → **string**)

This procedure is called by the *eformat* procedure in the Format environment. It is called if either *ipart* or *fpart* is a negative number.

16.14 Event Environment

The Napier88 system recognises a small range of asynchronous events. These are a hangup signal, an interrupt signal, a quit signal and a timer interrupt. On completion of a particular event procedure, the procedure will return to the running program.

The Event environment contains the procedures that are called when one of the above events is detected by the system. These procedures are variables and the user may change them by assignment. The default procedures are described below.

The Unix signals referred to may be found in §3 of the Unix Manual under Signal.

hangup : **proc** ()
This procedure is called if the Napier88 system receives a Unix SIGHUP signal. By default, this procedure stops the Napier88 system.

interrupt : **proc** ()
This procedure is called if the Napier88 system receives a Unix SIGINT signal. By default, this procedure does nothing.

quit : **proc** ()
This procedure is called if the Napier88 system receives a Unix SIGQUIT signal. By default, this procedure stops the Napier88 system.

timer : **proc** ()
This procedure gets called 30 times per second. By default, the procedure does nothing.

17 References

[ABC83] Atkinson,M.P., Bailey,P.J., Chisholm,K.J., Cockshott,W.P. & Morrison,R.
"An approach to persistent programming". Computer Journal 26,4 (November 1983), 360-365.

[AM85] Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment". Proc of the Appin Workshop on Data Types and Persistence, Universities of Glasgow and St Andrews, PPRR-16, (August 1985),1-25. In **Data Types and Persistence**. Springer-Verlag (1988), 1-20. (Eds Atkinson, M.P., Buneman, O.P. & Morrison, R.)

[AMP86] Atkinson, M.P., Morrison, R & Pratten, G..
"Designing a persistent information space architecture". 10th IFIP World Congress, Dublin (September 1986),115-120. North-Holland, Amsterdam.

[BCC88] Brown, A.L., Carrick, R., Connor, R.C.H., Dearle, A. & Morrison, R.
"The Persistent Abstract Machine". PPRR-58-88, Universities of St Andrews and Glasgow (1988).

[BRO89] Brown, A.L.
"Persistent Object Stores". PhD. Thesis, University of St Andrews (1988).

[CBC89] Connor, R.C., Brown, A.L., Carrick, R., Dearle, A. & Morrison, R.
"The Persistent Abstract Machine". 3rd International Workshop on Persistent Object Systems, Newcastle, N.S.W., (January 1989), 80-95.

[CW85] Cardelli, L. & Wegner, P.
"On understanding types, data abstraction and polymorphism". ACM.Computing Surveys 17, 4 (December 1985), 471-523.

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

[DEA88] Dearle, A.
"On the Construction of Persistent Programming Environments". PhD. Thesis, University of St Andrews (1988).

[DEA89] Dearle, A.
"Environments : A flexible binding mechanism to support system evolution". 22nd Hawaii International Conference on System Sciences, Hawaii, U.S.A. (Jan 1989), 46-55.

[KRA85] Krablin, G.L. "Building flexible multilevel transactions in a distributed persistent environment". proceedings of Data Types and Persistence Workshop, Appin, (August 1985), 86-117. In **Data Types and Persistence**. Springer-Verlag (1988), 213-234. (Eds Atkinson, M.P., Buneman, O.P. & Morrison, R.)

[MAT85] Matthews, D.C.J.
"Poly manual". Technical Report 65 (1985), University of Cambridge, U.K.

[MBB89] Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A., Hurst, A.J. & Livesey, M.J. "Language Design Issues in

Supporting Process-Oriented Computation in a Persistent Environment". 22nd Hawaii International Conference on System Sciences, Hawaii, U.S.A. (Jan 1989), 736-745.

[MBC87] Morrison, R., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A. & Atkinson, M.P.
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object-Oriented Environment". Journal on Software Engineering (Dec 1987), 199-204.

[MBD86a] Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"An integrated graphics programming environment". 4th UK Eurographics Conference, Glasgow (March 1986). In Computer Graphics Forum 5, 2 (June 1986), 147-157.

[MBD86b] Morrison, R., Bailey, P.J., Davie, A.J.T., Brown, A.L. & Dearle, A.
"A persistent graphics facility for the ICL PERQ computer". Software, Practice & Experience 16, 4 (April 1986), 351-367.

[MIL79] Milner, R.
"A theory of type polymorphism in programming". JACM 26(4), 792-818. (1979).

[MIL83] Milner, R.
"A proposal for standard ML". Technical Report CSR-157-83. University of Edinburgh. (1983).

[MOR79] Morrison, R.
"On the Development of Algol". PhD Thesis, University of St Andrews, (1979).

[MOR82] Morrison, R.
"Low cost computer graphics for micro computers". Software, Practice & Experience 12, 8 (August 1982), 767-776.

[MP88] Mitchell, J.C. & Plotkin, G.D
"Abstract types have existential type". ACM.TOPLAS 10, 3 (July 1988), 470-502.

[PPR88] PS-algol Reference Manual - Fourth Edition. Universities of Glasgow and St Andrews. PPRR-12-88 (1988).

[STR67] Strachey, C.
"Fundamental concepts in programming languages". Oxford University Press, Oxford (1967).

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_parameter_list>]**is**<type_id>

<type_parameter_list> ::= <lsb><identifier_list><rsb>

Type descriptors:

<type_id> ::= **int** | **real** | **bool** | **string** | **pixel** | **pic** | **null** | **any** | **env** | **image** | **file** |

<identifier>[<parameterisation>] | <type_constructor>

<parameterisation> ::= <lsb><type_identifier_list><rsb>

<type_identifier_list> ::= <type_id>[,<type_identifier_list>]

<type_constructor> ::= <star><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>]

[<arrow><type_id>])

<parameter_list> ::= <type_id>[,<parameter_list>]

<abstype> ::= **abstype**<type_parameter_list>(<named_param_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>

<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>]

<env_decl> ::= **in**<clause>**let**<object_init> |

in<clause>**rec let**<rec_object_init>[&<rec_object_init>]*

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><bar><clause>) |

<expression>(<dereference>) |

<expression>'<identifier> |

<expression><lsb><specialisation><rsb> |

<expression>([<application>]) |

<structure_creation> | <variant_creation> |

<clause> **contains** [**constant**] <identifier>[:<type_id>] |

any (<clause>) |

<name>

<dereference> ::= <clause>[,<dereference>]

<specialisation> ::= <type_identifier_list>

<application> ::= <clause_list>

<structure_creation> ::=
<identifier>[<lsb><specialisation><rsb>]([<clause_list>])

<variant_creation> ::= <identifier>[<lsb><specialisation><rsb>]
(<identifier>:<expression>)

<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>

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

<vector_element_init> ::= <range>**of**<clause> | <range>**using**<clause> |
@<clause>**of**<lsb><clause>[,<clause>]*<rsb>

<range> ::= <clause>**to**<clause>

<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>]

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

<struct_init_list> ::= <identifier><init_op><clause>[;<struct_init_list>]

<picture_constr> ::= <lsb><clause>,<clause><rsb>

<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>

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> ::= <double_quote>[<char>]*<double_quote>

<char> ::= any ASCII character except " | <special_character>

<special_character> ::= <single_quote><special_follow>

<special_follow> ::= n | p | o | t | b | <single_quote> | <double_quote>

<pixel_literal> ::= **on** | **off**

<null_literal> ::= **nil**

<proc_literal> ::= **proc**[<type_parameter_list>]([<named_param_list>]

[<arrow><type_id>]);<clause>

<picture_literal> ::= **nilpic**

<image_literal> ::= **nilimage**

<file_literal> ::= **nilfile**

Miscellaneous and microsyntax:

<lsb> ::= [

<rsb> ::=]

<star> ::= *

<bar> ::= |

<add_op> ::= + | -

<mult_op> ::= <int_mult_op> | <real_mult_op> | <string_mult_op> |

<pic_mult_op> | <pixel_mult_op>

<int_mult_op> ::= <star> | **div** | **rem**

<real_mult_op> ::= <star> | /

<string_mult_op> ::= ++

<pic_mult_op> ::= ^ | ++

<pixel_mult_op> ::= ++

<rel_op> ::= <eq_op> | <co_op> | <type_op>

<eq_op> ::= = | ~=

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

<type_op> ::= **is** | **isnt**

<arrow> ::= →

<single_quote> ::= '

<double_quote> ::= "

<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 |

n | 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

type nonvoid **is** literal | image | 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> ::= [**in**<clause> : env] **let**<object_init> |
[**in**<clause> : env] **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 :

<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 :

<expression> : **bool** **or** <expression> : **bool** => **bool**

<expression> : **bool** **and** <expression> : **bool** => **bool**

[~]<expression> : **bool** => **bool**

t : **nonvoid**, <expression> : t <eq.op> <expression> : t => **bool**
where <eq_op> ::= = | ≠

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

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

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

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

t : **arith** , <expression> : t <add_op><expression> : t => t

t : **arith** , <add_op> <expression> : t => t

t : **int**, <expression> : t <int_mult_op> <expression> : t => t
where <int_mult_op> ::= <star> | **div** | **rem**

t : **real** , <expression> : t <real_mult_op> <expression> : t => t
where <real_mult_op> ::= <star> | /

t : **string**, <expression> : t <string_mult_op> <expression> : t => t
where <string_mult_op> ::= ++

t : **pic**, <expression> : t <pic_mult_op> <expression> : t => t
where <pic_mult_op> ::= ^ | ++

t : **pixel** , <expression> : t <pixel_mult_op> <expression> : t => t
where <pixel_mult_op> ::= ++

t : **nonvoid** , <standard_expression> : t => t

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 <bar> <clause> : int) => string

<expression> : image (<clause> : int <bar> <clause> : int) => image

<expression> : pixel (<clause> : int <bar> <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**<lsb><clause> : t
[,<clause> : t]* <rsb> => *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>]

<lsb><clause> : real ,<clause> : real <rsb> => 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

<double_quote>[<char>]*<double_quote> => string

on | **off** => pixel

nil => null

t : type, **proc** [<type_parameter_list>]([<named_param_list>]
[<arrow><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 a procedure and its parameters. For example,

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

is valid but,

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

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**	**null**
of	**off**	**on**	**onto**	**or**	**output**		
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**						

Appendix V

Running the Napier88 System

There are four commands that control the execution of the Napier88 system. For convenience, when a program is compiled it may be compiled against a set of pre-compiled type declarations. For the present release only, these type declarations are stored in a PS-algol database which must be specified when placing the declarations in and using them from the database. The four Napier88 commands are: nps, npc, npr and npd.

1 nps

This command is used to save a set of type declarations for later use by programs wishing to compile against them. The command is parameterised by the name of a file containing the type declarations and the database name, password and key under which the type declarations should be saved. The database must already exist. It may also be parameterised by other type environments held in PS-algol databases. When this occurs the type environments provide the outer scoping levels within which the type declarations are compiled. The first named environment is the outermost scope and the source file provides the innermost scope. The general form of the command is:

nps sourcefile dbName dbPassword dbKey [-l] [-t dbN dbP dbK]*

The -l flag produces a source listing.

For example, to save a set of type declarations, given in the file "types.N", in the database "fred" with password "bloggs" under the key "sometypes", the following could be used:

nps prog.N fred bloggs sometypes

The Napier88 source file must consist purely of type declarations or the command will fail.

If a source file is not specified as a parameter, the command enters interactive mode. First the command will prompt for a list of type environments against which a source file may be compiled. Each type environment is at an inner scope level with respect to any previously specified environments. To finish specifying type environments, enter <return> in response to the command's prompt: *database:* .

Once any type environments have been specified, the command will prompt for a source file to be compiled and the database, password and key under which its type declarations should be saved. The source file is compiled against the complete set of type environments that have been specified. When the type declarations have been saved the command will prompt for another source file to compile. To finish saving new type environments, enter <return> in response to the command's prompt: *Filename:* .

To interactively save a set of type declarations, given in the file "types.N", in the database "fred" with password "bloggs" under the key "sometypes", the following could be used:

```
nps                          ! the command
Source type environment      ! request for an environment to compile against
Database: <return>           ! none to specify
Filename: types.N            ! the source file to compile
```

```
Destination type environment      ! request for where to save the new environment
Database: fred                    ! the database name
Password: bloggs                  ! its password
Name: sometypes                   ! the  new environment's key
Filename: <return>                ! no more source files to be compiled
```

2 npc

This command is used to compile Napier88 programs and is parameterised by the name of the source file. It may also be parameterised by other type environments held in PS-algol databases. When this occurs the type environments provide the outer scoping levels within which these declarations are compiled. The first named environment is the outermost scope and the source file provides the innermost scope. The result of the compilation is placed in a file whose name is constructed from that of the source file. The general form of the command is

npc sourcefile [-l] [-t dbName dbPassword dbKey]*

The -l flag produces a source listing.

For example, to compile a program in a file *prog.N*, the following could be used:

npc prog.N

The result of this compilation would be placed in a file *prog.out*. If the source filename does not end in *.N* then a filename is constructed by appending *.out* to the source filename.

If a source file is not specified as a parameter, the command enters interactive mode. First the command will prompt for a list of type environments against which a source file will be compiled. Each type environment is at an inner scope level with respect to any previously specified environments. To finish specifying type environments enter, <return> in response to the command's prompt: *database:* .

Once any type environments have been specified, the command will prompt for a source file to be compiled. The source file is compiled against the complete set of type environments that have been specified. When the compilation is complete the command will prompt for another source file to compile. To finish compiling, enter <return> in response to the command's prompt: *Filename:* .

To interactively compile the program in the file "prog.N", against the type environment held in the database "fred" with password "bloggs" under the key "sometypes", the following could be used:

```
npc                               ! the command
Source type environment           ! request for an environment to compile against
Database: fred                    ! the database name
Password: bloggs                  ! its password
Name: sometypes                   ! the environment's key
Source type environment           ! request for another environment
Database: <return>                ! no more to specify
Filename: prog.N                  ! the source file to be compiled
Filename: <return>                ! no more source files to be compiled
```

3 npr

This command is used to run the Napier88 system. If the command is parameterised, it must be given the name of a compiled Napier88 program that will

be executed. Otherwise, if no parameters are specified, the Napier88 system will be restarted from the state preserved by the most recent stabilise operation.

4 npd

This command is used to disassemble a compiled Napier88 program into PAM code. It is parameterised by the name of a file containing a compiled program and will display a listing of the program's PAM code one procedure at a time, starting with the main program. For example, to disassemble a program in a file *prog.out*, the following could be used:

```
npd prog.out
```

An optional parameter may be supplied to yield a count of how many times each PAM instruction occurs in the file. For example,

```
npd prog.out count
```

Appendix VI

Installing the Napier88 System

1 Installing the System

Before installing the Napier88 system read all of sections 1 to 4.

1. If not present, install a PS-algol system.
2. Copy the release tape into file store.
3. Create a shell variable named *NPRDIR* initialised with the pathname of the directory containing the release.
4. Add the directory $NPRDIR/bin to your search path. The commands now available
5. Load the Napier88 compiler into its PS-algol database using the command *nprloadc*
6. Create a new stable store using the command *nprformat*.
7. Initialise the stable store with the Napier command *nprloadstdenv*.

2 Directory Information

The release tape contains the following directories:

bin this contains the commands required to use the Napier88 system,
comp this contains the compiled version of the Napier88 compiler written in PS-algol,
exec this contains the executable programs used by the commands in *bin*,
stdenv this contains the compiled Napier88 programs to load the standard environment.

3 Shell Variables

There are several shell variables that allow the Napier system to be dynamically configured. They are:

NPRDIR - this variable defines the pathname for the release directory. All the commands held in the *bin* directory use this variable to construct the pathnames of the executable programs to be run. If *NPRDIR* is not defined the pathname used is /usr/lib/napier88.

NPRSTORE - this variable defines the pathname for the UNIX file containing the stable store. If *NPRSTORE* is not defined then the pathname used for the stable store is the name of the processor prefixed by $NPRDIR/sstore. e.g. on a SUN named *regal* the pathname would be $NPRDIR/sstore.regal. If the desired size of stable store is too large for the disk partition containing the release directory, a symbolic link can be used to map the store's pathname onto a larger disk partition.

NPRHEAP - this variable defines the size of local heap (in bytes) used by the Napier88 interpreter, by default this is 1 megabyte.

NPRKEYS - this variable defines the pathname for a UNIX file containing encrypted machine Identifiers. If not defined the pathname used is $NPRDIR/exec/PSKEYS.

4 The Napier Commands

The *bin* directory contains the following commands used to install and run the Napier88 system:

nprloadcompiler

This command runs the compiled PS-algol programs in the *comp* directory that load the napier compiler into a PS-algol database. The command does not require any parameters.

nprformat

This command creates a UNIX file containing an empty stable store. It takes two parameters, the size of the stable store in 8 kilobyte blocks and the number of blocks to be used as shadow store. e.g. to create an 8 megabyte store with 1 megabyte of shadow store the following command would be used:

```
nprformat   1024    128
```

The pathname of the UNIX file is as defined by the shell variable *NPRSTORE*. Since the stable store may be a very large file, it may defeat an incremental dumping strategy. This problem may be overcome by using a symbolic link to relocate the UNIX file on a different disk partition and employing an alternative dumping strategy.

nprloadstdenv

This command runs the compiled Napier88 programs in the *stdenv* directory that construct the standard user environment for the Napier88 system. The command does not require any parameters.