PROSE Programming Language

System Design Specification

Author: Mark R. Bannister

Version: 0.6

Issue Date: 3rd September 2004

Table of Contents

Please email cambridge@users.sourceforge.net if you have any suggestions for improving this specification.

Index of Tables

1 Introduction

This document is a design specification for the PROSE language system. The programming language is known as a *system* because its framework can be viewed as separate entities, or modules, interacting with each other. It is also more than just a language because all iteraction with the underlying computer system is managed through a platform independent framework.

PROSE is an adventure down paths both familiar and new, revisiting old programming haunts and exploring new regions barely visited by previous travellers. It carves a new road through the landscape, sometimes using paths already well trodden, and you will sometimes stumble upon surprising and rewarding new perspectives that you have not discovered before.

The PROSE language system is a hierarchically structured event-driven language. It is not an object-oriented language, even though it will sometimes be confused as such. We do however, like all good languages, borrow ideas from those who have gone before us, including OO concepts.

PROSE is born out of a need to create powerful and flexible UNIX and Windows networked applications, while making the development process as easy, productive and enjoyable as possible. It is designed to be an accessible language for developers as well as system and network administrators.

The PROSE programming language attempts to address the following needs:

- Highly flexible, yet remaining as simple as possible to use, and easy to read.
- Very powerful, implementing code that is faster than standard C libraries whenever possible.
- Limitless, not having any unnecessary bounds.
- Highly structured, encouraging programmers to develop readable and re-useable code.
- Especially easy to create interfaces between PROSE code and other languages, computers, networks, directories, databases and devices.
- A full range of security features that allow the language to be used in sensitive environments.
- Cross-platform, allowing developers to create code that will work without modification on Windows and UNIX platforms.
- Inner workings that are extensible and easy to manipulate, enabling fast and efficient debugging.

2 Overview of core features

2.1 Hierarchical framework

It has been demonstrated many times that the best way to organise data that both a computer and a human will have to interpret, is to represent it in a hierarchical format. Filesystems, process tables, DNS, NIS+ and LDAP are all examples of this.

The heart of PROSE is a large, flexible, fast-access hierarchy represented as a "hash tree", that is a tree structure where parent nodes each have an optimised hash table which is used for referencing any number of child nodes.

The tree (known hereafter as the "nexus") defines method names and code, variable names and data storage, provides native interfaces to other hierarchical systems such as DNS, LDAP and filesystems, and enables access to the tree structure behind other running PROSE processes on local or remote computer systems.

Each node (or "object") in the nexus can have any number of attributes assigned to it (each attribute having a name and a value), can have any number of child nodes attached to it, can be linked with other nodes in the tree so they can represent the same information or branch, and can be associated with program code so that changes to the node can produce automatic side-effects. For example, a filesystem can be traversed and a new file created by adding a node to the correct portion of the nexus.

2.2 Low-level and high-level languages

In order to maintain the structure of the nexus, a low-level language is provided for performing all basic functionality. This is called PROSE Assembly Language, or PAL, which provides the bridge between the programmer and the low-level PROSE engine.

The PROSE engine is responsible for managing the nexus, such as creating and deleting objects, referencing objects, assigning attributes to objects and executing program code. It is the sole responsibility of the PROSE engine to maintain data integrity in the nexus, and to perform memory tidying and clean-up operations as and when required.

The PROSE language looks similar to other UNIX scripting tools. This similarity is not an accident, new languages can be picked up more quickly by programmers who already have some familiarity with the syntax. PAL can be mixed into a normal PROSE script, alongside the higher-level commands. All program code is attached to objects in the nexus as modules which are executed whenever required by the lower-level PROSE engine.

PROSE scripts can either be pre-compiled into byte-code, or they can be compiled at run-time (behaving much like an interpreter). Either way, the nexus is organised at run-time with code modules, scalar variables and data structures.

2.3 Code modules

Compiled byte-code is attached to objects relating to method names, so that the code can be easily referenced by the PROSE engine as and when needed. The methods are grouped together in the nexus according to module, where a module may contain any number of source files and share variables and data structures.

2.4 Variables and data structures

Variables and data structures are also stored as branches, objects and attributes in the nexus. They may be referenced using standard C-type variable syntax, or also directly addressed using a path to identify the object in its hierarchical context. There is no language limitation to the size of the data values that may be used, only the bounds of available system memory.

2.5 Interfaces to other systems

PROSE is supplied with interfaces to other systems, for example to filesystems and to the network. This allows file and directory access through normal operations on the nexus, and allows for example, two running PROSE programs on two separate computer systems to link their nexi together. A nexus can also be represented by, or be used to represent, an LDAP directory tree. This makes PROSE an ideal language for LDAP development.

3 Data types, scalar variables and conversions

3.1 Introduction

PROSE is a *loosely typed language*, meaning that the type of expressions is not necessarily known at compile time. A data type is important to a computer programming language, as it dictates how to interpret information, and what tasks can be performed on it.

Scalar variables are basic storage locations, and are referenced in programming languages using names much like nouns are used in natural languages. Declaring a variable is a method of introducing a new storage location to the programming language. However, in PROSE, variables do not need to be declared before their first use, although you can declare them up front if you prefer.

3.2 Variable names and attributes

Variables are typically assigned values using the format `name = expr` where `name` is the variable name, and `expr` is an expression that is evaluated and stored in the variable location. This is known as a *data item*.

Data items are stored as objects in the nexus. Like any such object, they possess more than just a single value, in fact they contain many *attributes*. Attributes are made up of a name and a value, and are addressed using the syntax `<object>:<attribute>` where `<object>` is the name of the object (in this case the variable name) and `<attribute>` the name of the attribute. Object attributes are, however, the subject of a later chapter. For now, variables will be identified only by the variable name, not the attribute.

3.3 Data type APIs

Every data type in PROSE is supported at the back-end by program code that makes itself available via a number of methods, known as the APIs (or Application Programming Interfaces). While this concept can be ignored by the average programmer, it are these building blocks that can be extended on by advanced programmers who wish to change the behaviour of built-in types, or create new data types of their own. Describing how to set-up a new data type is beyond the scope of this system specification, but is discussed in the *PROSE API Specification*.

3.4 Scalar variable identifiers

Variable names may contain any number of characters in the ASCII ranges `A` to `Z`, `a` to `z`, `0` to `9` and the underscore `_` character, but the name may not begin with a digit. A variable name may also contain any other character as long as that character is prefixed with a `\` backslash.

Here are some examples of valid variable names:

```
my_variable_name
My_Longer_Variable_Name
AnEvenLongerVariableNameWith123digits
this\ variable\ name\ contains\ spaces
just\.to\.confuse\.this\.one\.contains\.dots\.ha\.ha\!
```

It will become clearer in later chapters why it is necessary to support variable names with any character. Under normal circumstances, it is better not to use special characters in a variable name, as it makes your program code less easy for others to read.

3.5 Primitive data types

A scalar variable will contain one of the primitive data types listed in Table 1 below.

`bool`	boolean
`int`	fixed precision integer
`intm`	multi-precision integer
`rational`	multi-precision rational number
`float`	fixed precision floating-point number
`floatm`	multi-precision floating-point number
`string`	Unicode character string
`data`	binary data

Table 1 - Primitive data types

These data types will be examined in the following sections.

3.5.1 Constants

Variables are generally assigned values using constants, other variables, or a combination of both. A *constant* is a static representation of a data value in the program source file (as provided by the software developer), for example the number 42 appears in the source as the digit '4' followed by the digit '2'. This is a numeric constant, and is converted to an internal representation of the number 42 at run-time. This is not to be confused with other languages, such as C, that perform this conversion at compile-time.

Constants in PROSE are pseudo-objects. They aren't actually attached to the nexus like variables and methods, nor can they be referenced as if they are. However, constants do have methods associated with them, referenced using the syntax `<constant>.<methodname>` where `<constant>` is the particular constant of interest, and `<methodname>` the method you wish to call against that constant. These methods are discussed in later chapters.

3.5.2 Boolean type

A boolean type has one of two states, true or false. This state is assigned to a boolean variable using the keywords `true` and `false`, which are also known as *boolean constants*. This type is generally used in conjunction with conditional statements and expressions (see: *6.7 Conditional code* on page 63).

3.5.3 Numeric types

Integers are whole numbers, they cannot have a fractional component. Floating-point numbers allow for the representation of fractional components in decimal dotted notation, and also permit scientific exponents to alter the magnitude of the value. Rational numbers may also have fractional components but are notated with numerators and denominators.

Numbers appearing in program code used to initialise a numeric type are called *numeric constants*. Numeric constants are themselves a data type, and treated like objects in the nexus just like variables and program code. The benefit of this will be seen in later chapters.

Computer systems typically need to be told what precision to use when storing and manipulating numeric types. The precision dictates how many bits (binary notation) should be used when working with the number. Fixed precision numeric types are dependent upon the numeric representation used in the underlying computer system, and will be platform specific. However, the default fixed precision numeric types are guaranteed to use a minimum of 64 bits. It is also possible in PROSE to request fixed precision numbers of a different minimum precision to that of the default.

A multi-precision numeric type (also known as arbitrary precision arithmetic) uses algorithms that are slower than with fixed precision, but will extend the number of bits used to represent a number as required to complete a calculation without losing the information required by the user - the minimum precision. This allows for mathematical calculations to be performed on numbers of any magnitude (system memory permitting).

Integers and floating-point numbers in PROSE can be either standard C types (fixed precision), or GNU MP types (multi-precision). A rational number is a number that always has a numerator and denominator, denoted by the syntax `[n/d]`, and this is always the GNU MP type.

3.5.3.1 Integer constants

Integer constants contain digits in the ASCII range `0` to `9` only, and may be prefixed with an optional sign (`+` or `-`). The base, which defaults to decimal, may be specified as hexadecimal (base 16) by prefixing the characters with `0x`, octal (base 8) by prefixing with an extraneous `0`, or binary (base 2) by prefixing with the characters `0b`. In base 16, the additional ASCII characters `a` to `f` are permitted in integer constants. These constants are converted to the appropriate C integer type, by default with a minimum of 64-bits precision. If post-fixed with the letter `M`, the constant is converted to the GNU MP integer type.

Table 2 below gives some examples of integer constants.

`15`	decimal value positive 15
`-57`	decimal value negative 57
`0x15`	hexadecimal value 21
`015`	octal value 13
`0b1111`	binary value 15
`13827193M`	decimal value 13827193, converted to GNU MP integer
`0x3a899cM`	hexadecimal value 3836316, converted to GNU MP integer

Table 2 - Integer constants

These are the *default* integer constants. It is possible to change the type of these constants using `typeset -d constant`.

3.5.3.2 Rational number constants

Rational number constants are enclosed in `[` square brackets `]`, and may contain either a single integer, or two integers denoting a fraction - a numerator and denominator separated by a forward slash `/` character. Each integer used in the constant may be prefixed in the same way as ordinary integers, specifying negative or positive, and to change the base to binary, octal or hexadecimal. These constants are always converted to the GNU MP rational number type.

Table 3 below gives some examples of rational number constants.

`[1]`	value 1/1
`[1/3]`	one third
`[3/10]`	three tenths

Table 3 - Rational number constants

These are the *default* rational number constants. It is possible to change the type of these constants using `typeset -d constant`.

3.5.3.3 Floating-point constants

Floating-point constants contain digits and one dot. They may also contain an optional exponent as with C, and by default are converted to the appropriate C floating-point type, by default with a minimum of 64-bits precision. If post-fixed with the letter `M`, the constant is converted to the GNU MP floating-point type. An optional integer may follow the letter `M`, which specifies that the new GNU MP floating-point value should have at least this number of bits in precision.

Table 4 below gives some examples of floating-point constants.

`1.3701`	Standard C floating-point constants
`-.006`	
`2.22044604e-5`	
`1.3701M`	GNU MP floating-point constants
`-.006M64`	*(minimum 64-bit precision)*
`2.22044604e-5M128`	*(minimum 128-bit precision)*

Table 4 - Floating-point constants

These are the *default* floating point constants. It is possible to change the type of these constants using `typeset -d constant`.

3.5.3.4 GNU MP constants

It is possible with the GNU MP library to specify any base between 2 and 36 when initialising integer, floating-point and rational numbers. However, to avoid confusing the syntax for number constants any further it is not possible to select a base other than binary, octal, decimal or hexadecimal in a constant. A method attached to the constant object is used to access other number bases, using the format `<constant>.<methodname>` where `<constant>` is the numeric constant and `<methodname>` the name of the method you wish to invoke. This, however, is the subject of a later chapter.

3.5.4 Character strings

Character strings are used in a language to represent the written form of a natural language. In PROSE, they contain a list of Unicode characters, which allow for the representation of many symbols from many different written languages worldwide.

Unicode character strings are enclosed in either `'` single `'`, `"` double `"` or ` ` angled ` quotes, and these are known as *string constants*. The decision of which quoting symbol to use is usually entirely arbitrary, as by default they all have equal meaning, the only exception being that whichever quote is used to open a string constant must also be used to close it.

These are the *default* string constants. It is possible to change the type of these constants, based on the type of quotation mark, using `typeset -d constant`.

Standard C escape codes may be used inside character strings, with a few additions, as illustrated in Table 5 below.

\a	alert (bell)
\b	backspace
\f	new page (form feed)
\n	new line (line feed)
\r	carriage return
\t	horizontal tab
\v	vertical tab
\<number>	insert octal code <number>, may not exceed \377
\x<number>	insert hexadecimal code <number>, may not exceed \xff
\u<number>	insert Unicode <number>, represented as a hexadecimal number in the range \u0000 to \uffff
\\	backslash
\′	protect single quote from closing string
\″	protect double quote from closing string
\`	protect angled quote from closing string

Table 5 - Escape codes

Strings may contain any character, and may fold onto any number of new lines. Unlike C, strings are not terminated by a nul (\0) character. There is no limit to the length of a character string, other than that imposed by the available system memory.

3.5.5 Binary data type

A binary data type is used to store information in a raw uninterpreted format. There are no binary data constants, but values assigned to binary data types are stored as is with no change to their present internal representation except that which is required to attain platform-independence. In such a form, it is possible to view and manipulate data items at the lowest possible level, using methods that do not necessarily know the meaning of the underlying data. Binary data types may also contain bytecode that can be attached to the nexus as new methods or functions, and then executed as such.

3.6 Variable declaration and initialisers

Variables do not need to be declared before they are first used. You will therefore never get a compile-time error relating to the use of variables.

If a variable is set without a previous declaration, the data type becomes that which is appropriate for the value being set. This is determined after type conversion is completed for the expression (see: *3.7.1 Implicit type conversion* on page 18).

If a variable is referenced before it has been assigned a value, then an appropriate *default value* is assumed in place of the variable. The default value's type will be a string unless used in a numeric expression, in which case it will be an integer. If the variable's type has been previously declared, the context of the expression is ignored and the default value determined based upon the declaration. The default value is *not* assigned to the variable. Storage space is not allocated for a variable until it is explicitly assigned a value. Table 6 below lists the default values assumed for each primitive data type.

`false`	boolean
`0`	integer
`[0/1]`	rational
`0.0`	floating-point
`""`	string
`0`	binary data

Table 6 - Default values

A variable's type may be explicitly declared before use, or changed after use, using the `typeset` keyword. The variable contents are not overwritten unless the declaration contains an initialiser. Initialisers are evaluated at run-time, which means you can include variables in the initialisers if required, as demonstrated in the following example:

```
typeset int myvar1                  # declare myvar1 as a C-type integer
typeset intm myvar2                 # declare myvar2 as a GNU MP integer
typeset string myvar3 = "Massenet"  # declare and initialise a string
typeset string myvar4 = myvar3      # and another string
typeset float myfloat1, myfloat2    # declare two C-type floats
typeset rational rat1 = [1/3], rat2 = [2/3]
                              # declare and initialise two rationals
```

Referencing an undeclared variable and thus setting its type implicitly is *not* equivalent to explicitly declaring the variable's type (see: *3.8 Polymorphism* on page 22).

3.6.1 Declarator options

It is possible to supply a list of comma-separated options in parentheses following the type name in a `typeset` statement if required. These options are intepreted on a type-specific basis. If no options are supplied, then the default options are assumed, as set by the `typeset -d options` keyword.

3.6.1.1 Integer precision and signing

When declaring a fixed precision integer number, the declarator option is a single positive or negative number specifying the minimum number of bits required in the integer. A positive number requests an unsigned integer, a negative number requests a signed integer. Signed and unsigned integers are treated exactly like their corresponding C-type (see: section 5.1 of [R1] *C: A Reference Manual, Fifth Edition*).

The exact number of bits selected is implementation defined, and will usually be the fastest possible integer representation, with a guaranteed minimum number of bits. Here are some examples:

```
typeset int(32) myvar1        # request min. 32 bits unsigned
typeset int(-32) myvar2       # request min. 32 bits signed
```

It is possible to change the default from signed 64 bit integers as follows:

```
typeset -d options int(-32)  # default is now min. 32 bits signed
typeset int myvar1           # this is now a 32 bit signed integer
```

If more bits are requested in a fixed precision integer than can be supported on the platform, by default this is converted to a request for an `intm` type. This behaviour may not be desirable, and can be amended using `typeset -d nosupport` as demonstrated below:

```
typeset -d nosupport int()=ERROR  # error if options not supported
typeset -d nosupport int()=OTHER  # use less precision
```

3.6.1.2 Floating-point precision

When declaring a fixed precision floating-point number, the declarator option is a single positive number specifying the minimum number of bits required. The exact number of bits selected is implementation defined, and will usually be the fastest possible floating-point representation, with a guaranteed minimum number of bits. Here is an example:

```
typeset float(32) myvar1      # request min. 32 bits precision
```

It is possible to change the default from 64 bits as follows:

```
typeset -d options float(32) # change default to min. 32 bits
typeset float myvar1         # this is now a 32 bit float
```

If more bits are requested in a fixed precision floating-point number than can be supported on the platform, by default this is converted to a request for a `floatm` type. This behaviour may be amended using `typeset -d nosupport` as demonstrated below:

```
typeset -d nosupport float()=ERROR    # error if not supported
typeset -d nosupport float()=OTHER    # use less precision
```

3.6.1.3 Multi-precision floating-point options

When declaring a multi-precision floating-point number, the declarator option is a single positive number specifying the minimum number of bits required, exactly as described above for fixed precision floating-point numbers. The only difference is that `typeset -d nosupport` has no effect, as GNU MP types support any minimum precision requested.

3.7 Type conversion

Any type can be converted to any other type, there are no exceptions to this rule. You will therefore never get a run-time error relating to the use of variables. When a conversion takes place from one data type to another, it is called a *cast*.

Casts can occur automatically when expressions use mixed types, known as *implicit type conversion*, or may be an operation that is specifically requested on a variable, which is called *explicit type conversion*. These are discussed in the following sections.

3.7.1 Implicit type conversion

PROSE casts between data types automatically. As an expression is parsed, it is broken down into its constituent parts, called *atoms*. An attempt is made to retain the value of the expression in the simplest possible data type, as the value of each atom in the expression is obtained, but without losing precision.

When a more complex type is encountered, the expression is converted to this type. This is a one-way conversion, a more complex type will not be automatically cast to a simpler type (as this will lose precision), unless the issue is forced by a previous type declaration or an explicit cast.

The data types in Table 7 below indicate the order of complexity, the simplest data types first.

1	boolean
2	integer
3*	multi-precision integer
4*	rational number
5	floating-point number
6*	multi-precision floating-point number
7	Unicode character string
8	binary data

Table 7 - Data type complexity levels

Type conversion can only be determined at run-time. Here are some examples:

```
myvar1 = 5                      # set myvar1 to 5
myvar2 = 3 + myvar1 + [1/3]     # myvar2 will contain [25/3]
myvar3 = myvar2 + [2/3] + 0.1   # myvar3 will contain 9.1
myvar4 = myvar3 + 1 + "%"       # myvar4 will contain "10.1%"
myvar5 = myvar4 + 10            # myvar5 will contain "10.1%10"
```

If a number has been cast to a GNU MP numeric type (indicated by an asterisk * in Table 7), then it will not at any point be reverted to a C numeric type by means of an implicit type conversion. Therefore in the previous example, although the floating-point constant `0.1` is a C-type constant, it is cast to a GNU MP type before the add operation occurs.

Expressions are always subject to these automatic casting rules regardless of the type of the target variable. If the target variable has a strict type as declared by an earlier `typeset`, and final type after parsing the entire expression does not match the type of the target variable, then it is subjected to one final cast. This is important, the following code segment will display `1020`, not `30`:

```
typeset int myvar1

myvar2 = "10"
myvar1 = myvar2 + 10
print myvar1 + 10
```

In the previous example, `myvar2 + 10` evaluates to `"1010"`, because according to the rules defined in Table 7 above, the number `10` has to be cast to a string type before it can be added to `myvar2` which is already a string. This is then assigned to `myvar1` that has been explicitly declared as an integer, so the value is cast to the final target type. The expression `myvar1 + 10` is then an integer add operation. If the `typeset` command had been omitted, then the expression `myvar2 + 10` would have remained as a string and `myvar1 + 10` would become a string append operation, resulting in `"101010"` not `1020`.

Here is another example:

```
myvar1 = "1"
myvar2 = 2
myvar3 = 3
print myvar1 + myvar2 + myvar3
print myvar1 + (myvar2 + myvar3)
```

The first `print` command in the previous example will display `"123"`, whereas the second `print` command displays `"15"`. The parentheses force `myvar2` and `myvar3` to be evaluated first.

3.7.2 Explicit type conversion

The implicit type conversion rules above may be modified by using cast expressions. These are data type names enclosed in `(` parentheses `)` before an expression, to explicitly request the result of the expression is cast into a particular type - discarding the automatic casting rules in that case. So, for example:

```
myvar1 = "1"
myvar2 = 2
myvar3 = 3
print myvar1 + myvar2 + myvar3
print (int)myvar1 + myvar2 + myvar3
```

The first `print` command in the previous example will display `"123"`, whereas the second `print` command displays `6`. The explicit cast forces the result of the `myvar1` expression (the value of the `myvar1` variable) to be converted into a C-type integer before evaluation continues.

It is not considered an error to explicitly cast from one type to another that would result in the loss of precision, nor is an error reported. If you cast from a string to a number type and the string does not contain a valid number, the result is `0`.

Declarator options may also be specified in the cast expression, e.g. `(int(-16))V` would cast variable `V` into a signed integer of minimum 16-bits precision (see: *3.6.1 Declarator options* on page 16).

As previously discussed, a variable's data type may also be explicitly changed using the `typeset` keyword (see: *3.6 Variable declaration and initialisers* on page 15).

3.7.3 Boolean type conversions

Conversions to the boolean type must result in either a state of `true` or `false`. Table 8 below illustrates how this is determined on conversions from other data types.

unassigned variable	`false`
`int` or `intm`	value `0` is converted to `false`, any other value is `true`
`rational`	value `[0/1]` is converted to `false`, any other value is `true`
`float` or `floatm`	value `0.0` is converted to `false`, any other value is `true`
`string`	blank string `""` is converted to false, any other value is `true`
`data`	first byte `0` becomes `false`, otherwise `true`

Table 8 - Conversions to boolean types

3.7.4 Numeric type conversions

Conversions to numeric types must result in a number that is valid for the type in question. If this requires a loss of precision, then this loss will occur silently. That is, in all cases, if there is more data than can be represented by the type, the data is truncated to fit the precision.

3.7.4.1 Integer type conversions

Table 9 below illustrates how other types are converted to integers.

unassigned variable	`0`
`bool`	`false` is converted to 0, `true` becomes `1`
`rational`	rounded to nearest whole number
`float` or `floatm`	
`string`	if the string begins with an integer constant, this is converted to an integer, otherwise the conversion results to `0`
`data`	raw data is assumed to be in big endian format, and is set to integer type with no change in internal representation except to endianness

Table 9 - Conversions to integer types

3.7.4.2 Rational type conversions

Table 10 below illustrates how other types are converted to rationals.

unassigned variable	`[0/1]`
`bool`	`false` is converted to `[0/1]`, `true` becomes `[1/1]`
`int or intm`	integer becomes numerator, denominator set to `1`
`float` or `floatm`	rounded to nearest whole number and converted as with an integer
`string`	if the string begins with a rational constant, this is converted to a rational, otherwise the conversion results to `[0/1]`
`data`	raw data set to rational type with no change in internal representation, except that required to maintain platform-independence

Table 10 - Conversions to rational types

3.7.4.3 Floating-point type conversions

Table 11 below illustrates how other types are converted to floating-point numbers.

unassigned variable	`0.0`
`bool`	`false` is converted to `0.0`, `true` becomes `1.0`
`int or intm`	straight mathematical conversion
`rational`	straight mathematical conversion
`string`	if the string begins with a floating-point constant, this is converted to a floating-point, otherwise the conversion results to `0.0`
`data`	raw data set to floating-point type with no change in internal representation, except that required to maintain platform-independence

Table 11 - Conversions to floating-point types

3.7.5 String type conversions

Other data types converted to strings will always result in their respective constant, for example a boolean type with a `false` state will be converted to a string containing the text `"false"`, and an integer number containing the value `53` will be converted to the integer constant `"53"`.

Unassigned variables cast to strings will result in an empty string. Binary data types are just set to the string type with no change in internal representation.

3.7.6 Binary data type conversions

When another data type is converted to a binary data type, there is guaranteed that the result is stored in a contiguous block of memory and in a format that maintains platform independence. In otherwords, if a

variable is converted to a binary data type and then back again on any other system, the value is guaranteed to be the same. Integers are always stored in big endian format when converted to a binary data type.

While a variable can be safely converted from one type to `data`, and then back to the original type with no loss of information; converting from one type to `data`, and then to a third type bypasses normal type conversion algorithms and will deliver undefined results.

3.8 Polymorphism

Polymorphism describes the ability in a language for the same piece of program code to operate on a variety of data types without any special provision.

As previously described, if you reference an undeclared variable, its type will be set appropriately according to the type of the resulting expression. This type, now associated with the variable, is however only a *fickle type*. This means that its type can change if it is subsequently set with an expression of a different type.

Use the `typeset` keyword, as in the following example, to prevent this behaviour:

```
myvar1 = myvar2 ++   # myvar2 is initiased as an integer with value 1
myvar3 = 5.3
myvar2 = myvar3      # myvar2 is now set to a float, with value 5.3
typeset int myvar2   # myvar2 is converted to an integer, with value 5
myvar2 = myvar3      # myvar2 remains as an integer
```

In the previous example, the ++ operator is used to increment the value of `myvar2` after assigning its current value to `myvar1`. But as `myvar2` currently has no value, the default value `0` is assumed before the operation (see: *3.6 Variable declaration and initialisers* on page 15). This value is then incremented and assigned to `myvar2`.

Because neither `myvar1` nor `myvar2` had been previously declared, they are fickle types. Although `myvar2` is at this point an integer type with value `1`, its type changes if a new value is assigned with a different type, such as when it is set to the value of `myvar3`. However, a variable declared using `typeset` will not have its type changed by such an assignment.

Polymorphism allows for methods that can handle any data type, and for collections containing multiple data types to be iterated by the same program code (see: *4 Stacks, arrays and structures (collections)* on page 26).

3.9 Pointers

All program code and data that make up a PROSE computer program is available to be referenced in a hierarchical tree structure - the *nexus* - where all objects are addressable using a unique path from the root node. Each data item also has an address in an internal pointer table, that can be used for directly locating an object. There is no correlation between pointers in PROSE and memory addresses known to the computer system.

Pointer variables can be set-up to point to any object in the nexus, and used as a quick reference to a variable or method, and for passing information between methods without the unnecessary duplication of data. Pointers to scalar variables are called *scalar pointers*.

Pointers have to be declared explicitly using the `typeset` keyword, there is no way of implicitly setting a pointer. Casting to a pointer will generate an error. To assign a pointer variable so that it references a foreign data item, you must set the location attribute, using the syntax `<name>:location = <expr>` or `<name>:loc = <expr>` where `<name>` is the name of the pointer variable and `<expr>` is an expression that evaluates to the object you wish to reference. Alternatively, an initialisation in the `typeset` declaration performs the same task.

Unlike C, for example, there is no special syntax for addressing the object that is pointed to. Pointers are used in expressions like ordinary variables, which manipulate the target data item not the pointer itself. Here is an example:

```
typeset pointer ptr   # pointers must have their type set explicitly

myvar1 = "glissando"
myvar2 = 25
ptr:loc = myvar1      # ptr now points to the same data as myvar1
print ptr             # this displays the contents of myvar1
ptr = myvar2          # myvar1 is overwritten with the value of myvar2
ptr:loc = myvar2      # ptr now points to the same data as myvar2
ptr = "glissando"     # myvar1 and myvar2 have now swapped contents
```

3.9.1 Pointer offsets

Pointer variables have an additional attribute, the *offset*, which has an integer value referenced using the syntax `<name>:offset` where `<name>` is the name of the pointer variable. The `offset` attribute modifies the behaviour of the pointer. This modification is specific to the target data type, but with all data types the `offset` defaults to `-1` which effectively turns it off. With scalar pointers, the `offset` attribute is ignored for all target data types with the exception of the Unicode string, when it specifies the starting character position; where `0` references the entire string, `1` the string without the first character, etc. For example:

```
typeset pointer ptr

ptr:loc = my_string = "string"
print (ptr:offset++, ptr), (ptr:offset++, ptr), (ptr:offset++, ptr)
```

The previous example will display `"string tring ring"`. The example also demonstrated *recursive expressions*, where `"string"` is a string constant assigned to the `my_string` variable which is subsequently assigned to the `ptr:loc` attribute; and *expression lists* where `ptr:offset++` is evaluated first (incrementing the offset) followed by the `ptr` reference itself which becomes the result of the expression (see: *5 Operators and expressions* on page 45).

PROSE makes extensive use of copy-on-write (CoW) algorithms. This means that data can be shared around using pointers and read from different locations without duplicating information until the copy is actually modified. Only then is the original data duplicated to the target location.

3.9.2 Declaring pointer target types

It is usually desirable to declare the type of data that a pointer references, i.e. the target data type. When such a declaration is made, an error is generated should an attempt be made to point to a different data type.

To specify the target data type use the form `typeset pointer <name> to <type>` where `<name>` is a comma-separated list of pointer variable names being declared and `<type>` the target data type.

Here is an example:

```
typeset pointer ptr to string  # only permit string references

composer = "Vivaldi"
year = 2004
ptr:loc = composer             # point to composer variable
ptr = "Bruch"
ptr:loc = year                 # this generates an error, as the year
                               # variable is not a string type
```

3.9.3 Pointer type masquerading

Another feature of PROSE pointers is the ability to masquerade as another data type. This means that references to the pointer variable behave as if the target data type is of one type, while in fact it is silently converted to another before being stored in the target object. This functionality is enabled when a pointer is declared using the form `typeset pointer <name> cast <type>` where `<name>` is a comma-separated list of pointer variable names being defined, and `<type>` the data type to masquerade as.

Here is an example of this feature in use, where a variable behaves exactly like an integer in all assignments and mathematical operations, but is stored in the back-end as a string:

```
typeset pointer ptr cast int   # masquerade as an integer

backend = "0"                  # set-up a string
ptr:loc = backend              # point to the string variable
print backend + 5              # casts 5 to string and appends
print ptr + 5                  # integer addition, NOT string append
```

Pointer masquerading is particularly useful in scenarios where the underlying storage type is subject to change but the interface must remain stable.

3.10 User-defined types

PROSE supports new numeric and string primitive types that can be variations on built-in primitive data types or new types entirely. These user-defined types behave in the same way as other numeric or string types in the language, and can have the same operations performed on them.

New types are introduced by the `typedef` keyword, and are assigned type names, complexity levels to assist with implicit type conversion, and methods for handling a variety of operations. These are all fully described in the *PROSE API Specification*.

New string types are assigned values using the same Unicode string constants as the built-in string type (see: *3.5.4 Character strings* on page 14), prefixed by a cast to the new type name. For example, if a new data type called `babelfish` was configured for language translation, you would make use of it in the following fashion:

```
typeset my_string = (babelfish)"My first test babelfish string"
my_string2 = (babelfish)"My second test babelfish string"
```

In the previous example, the string constants in the source file are first converted to Unicode strings, which are then passed to the `babelfish` API to be converted into whatever format is appropriate for the new string type. From then on, all operations on the variables `my_string` and `my_string2` will be passed to `babelfish` APIs. As with built-in string types, these variables do not need to be declared before they are used.

User-defined numeric types are usually assigned values using the form `[<expr>, <expr> ...]` where `<expr>` is an expression that evaluates to a primitive data type. These are known as *unit constants*. Any number of comma-separated expressions may be supplied within square brackets, and these are called *units*. The units are passed as arguments to the new data type API. If the expressions are numeric constants, they will be converted to the numeric type suggested by the constant first (see: *3.5.3 Numeric types* on page12), before being passed to the new API. These may also be unit constants.

For example, if a new data type called `interval` was configured for interval mathematics, and another called `complex` to represent complex numbers (with real and imaginery parts), they would be used as follows:

```
my_interval = (interval)[0.35, 0.36]  # initialise with two floats
my_complex = (complex)[5, 3]          # initialise real and imaginery
                                      # parts with two integers
```

Attribute names could be provided with the new types for accessing their component parts by name if required, for example `my_complex:real` and `my_complex:imaginery`.

In the previous example, the interval constant is converted to two C-type floating-point numbers before being passed to the interval type API. This would result in a loss of precision, and may not be desirable. In this case, the interval type API might also accept string constants in an assignment, illustrated below:

```
my_interval = (interval)["0.35", "0.36"]   # initialise with two strings
```

There is now no loss of precision when initialising the interval number. Assuming either the `interval` or `complex` APIs knew about the existance of the other and could perform a conversion from the one type to the other, it would be possible to use unit constants inside unit constants, e.g.

```
my_complex = (complex)[5, 3]
my_interval = (interval)[my_complex, (complex)[6, 3]]
```

Another kind of number that would have a use as a user-defined type is a version number, with major, minor and patch revision numbers. If such a type were configured, it could be initialised as follows:

```
my_version = (version)[0, 9, 1]
```

Casting a version number to a string type would result in a format like `"0.9.1"`, and the version number manipulated using increment or decrement operators on the attributes, for example following the expression `my_version:minor++`, the version number type would contain `[0, 10, 1]`.

User-defined constants may be given default types if required, so that constant expressions do not always require an explicit cast to the user-defined type. It is also possible to change the type of PROSE primitive constants if a new user-defined type provides similar or superior functionality (see: *6.4.3.2 Default constant data types* on page 56).

4 Stacks, arrays and structures (collections)

4.1 Introduction

The primitive data types discussed in the previous chapter can be combined together and contained within a larger group of data items, known as a *collection*. In other programming languages, these are known as arrays. They are used much like collective nouns are used in natural languages.

Arrays of varying types are common to many programming languages, but in PROSE any number of differing data types can be added to the same collection.

A PROSE collection contains objects. These objects are referenced by pointers. The data item itself is never stored in the collection, thus the collection is a typeless object and cannot compete in automatic type casting (see: *3.7.1 Implicit type conversion* on page 18).

When coupled with variable type polymorphism (see: *3.8 Polymorphism* on page 22), this makes it possible to iterate a collection using one variable that will change its type according to the type of the data item read. *Iteration* being the process whereby each item stored in a collection is read and processed in turn, one at a time.

Because a collection is simply an array of pointers that are organised in different ways, you can store primitive data types, other collections and even pointers to methods in a collection.

4.2 Collection identifiers

Collections are denoted using the `name[expr]` syntax, where `name` follows the same rules as a scalar variable name (see: *3.4 Scalar variable identifiers* on page 10). The square brackets following the name make the distinction between a primitive data type and a collection. The same namespace is used as for scalar variables, so it is not valid to have a scalar variable with the same name as a collection name.

The syntax of the expression `expr` enclosed within the square brackets is dependent upon the type of collection. This is called the *index*, and is used for referencing data items within the collection. When the index is omitted, the identifier references the entire collection.

4.3 Collection types

PROSE supports four types of collection, listed in Table 12 below.

`fifo`	FIFO stack (queue)
`filo`	FILO stack
`matrix`	matrix array
`tree`	tree structure

Table 12 - Collection types

These collection types are discussed in the following sections.

4.4 Default collection type

Because collections, just like scalar variables, can be referenced without a declaration (see: *3.6 Variable declaration and initialisers* on page 15), it is necessary to define a *default collection type*. This specifies which kind of collection should be initialised when a `name[]` is set without a prior declaration.

By default, undeclared collection names are initialised as tree structures. This behaviour can be changed using the `typeset -d collection <type>` command, where `<type>` is one of `fifo`, `filo`, `matrix` or `tree`.

4.5 FIFO and FILO stacks

A PROSE stack is a collection of objects that you can visualise are organised in a vertical arrangement. Data items are pushed onto the stack and pulled off the stack one at a time. When a value is assigned to the stack (much like assigning a value to a variable), an object is pushed onto the stack. By default, when the stack is referenced (much like reading the value of a variable), the object is pulled off the stack. This behaviour can be changed using the `freeze()` and `unfreeze()` methods, so you can peek at the object without pulling.

A FIFO (first-in first-out) stack is also commonly known as a *queue*. Data items are pulled in the same order in which they were pushed. This is illustrated in the following example:

```
typeset fifo myqueue[]

myqueue[] = 'first value'
myqueue[] = 2
myqueue[] = 'third value'
print myqueue[], myqueue[], myqueue[]
```

The previous code example will display:

```
first value   2    third value
```

A FILO (first-in last-out) stack behaves in the opposite fashion. Data items are pulled in the reverse order in which they were pushed. This is illustrated in the following example:

```
typeset filo mystack[]

mystack[] = 'first value'
mystack[] = 2
mystack[] = 'third value'
print mystack[], mystack[], mystack[]
```

The previous code example will display:

```
third value  2    first value
```

PROSE stacks are implemented as double-ended linked lists. This makes them efficient and flexible, but uses slightly more memory than a matrix array of similar dimensions.

4.5.1 Referencing individual objects in a stack

The PROSE stack types may also be treated like matrix arrays with a single dimension. This allows for individual items in the stack to be referenced by providing an expression between the square brackets that evaluates to an integer index, e.g. `stackname[i]` where `i` specifies the offset from the top of the stack. An offset of `0` references the topmost item.

4.5.2 Referencing multiple objects in a stack (wildcard index)

A wildcard index on a stack collection is one where the index `-1` is used. This has the effect of referencing every data item in the stack, as opposed to just the item at the top of the stack. Therefore, while `print mystack[]` pulls the topmost item and displays it, `print mystack[-1]` will pull all items and display them one at a time.

4.5.3 Stack methods

FIFO and FILO stacks are supplied with methods that extend their functionality. These methods are invoked using the syntax `stackname[].methodname()` where `stackname` is the name of the stack variable, and `methodname` the method you wish to call. Methods operate on the entire stack unless otherwise stated, in which case the index supplied between square brackets `[]` selects the context.

The most commonly used stack methods are summarised in Table 13 below.

`size()`	return number of items in the stack
`reverse()`	turn stack on its head
`push(e)`	push result of expression `e` on top of stack
`peek()`	return value on top of stack, but leave it there
`pull()` `pull(e)`	return value on top of stack, and remove it if expression `e` is supplied, pull the object that has been marked by `e` using the `marker()` method, dropping all items pushed on top of this
`freeze()`	further references `peek()` don't `pull()`
`unfreeze()`	further references will `pull()`
`marker(e)`	put a marker on the top-most object on the stack, optionally identified by expression `e` (any primitive data type)
`flush(e)`	flush stack down to (but not including) the item marked by expression `e`, if `e` is omitted the entire stack is flushed
`empty()`	return `true` if stack is empty, otherwise `return` false
`search(e)`	search stack for regular expression `e` and return index or `-1` if the expression is not matched
`next()`	find next cell during a search
`reset()`	if index is not supplied, reset search to top of stack, otherwise reset search to begin at the addressed data item
`sort()`	sort the stack
`rawOut()`, `rawIn()`	convert stack to and from a contiguous binary representation of the collection, for storing in flat files or in binary data types

Table 13 - Commonly used stack methods

This is not the entire list of available stack methods. A complete list and a full discussion of stack methods with examples are covered in the *PROSE API Specification*.

4.6 Matrix arrays

A PROSE matrix array is a collection of objects that are organised in a multi-dimensional tabular fashion. The smallest unit in a matrix is known as a *cell*. This is closest to the historical definition of an array within a computer programming language.

Data items are stored in and retrieved from the array using a list of comma-separated expressions enclosed in square brackets `[]` that evaluate to integer co-ordinates (the *index*), starting with `0`. The exact data required is retrieved after a simple mathematical calculation has occurred on the co-ordinates. This allows for very fast data manipulation, but memory requirements increase proportionally with the multiple of the dimensions.

A multi-dimensioned array can be referenced with less co-ordinates in the index than dimensions. In which case, the remaining unreferenced dimensions are assumed to be integer `0`.

4.6.1 Array sizing

As with all PROSE data types, matrix arrays do not need to be declared before they are used. If they are not declared, or they are declared without any dimensions, then they will be automatically sized (and if necessary resized) according to their use. Automatic sizing only occurs when a value is set in the array, not if a non-existant value is referenced.

A matrix array may be explicitly sized and resized using the `typeset matrix` command. Here, an integer defines the maximum number of cells in a dimension, which is one more than the highest co-ordinate that can be referenced. This does not result in any matrix initialisation activity, so does not carry a performance overhead. However, an array that has been explicitly sized will not be automatically resized if a value is set outside the bounds of the matrix. Instead, the value is discarded.

Here is an example:

```
typeset matrix my_array[]
my_array[5,2]="Marcato"   # my_array auto sized to 6 cols by 3 rows
my_array[8,9]=24          # now auto re-sized to 9 columns by 10 rows
typeset matrix my_array[6,10]   # explicitly re-sized to 6 by 10
my_array[8,9]=24          # auto re-sizing off, value is discarded
```

4.6.2 Cell data types

The `typeset` keyword can also be used to declare the type of a particular cell in the matrix. This has the same effect on the cell as declaring the type of a scalar variable up front, i.e. it forces the hand during automatic type casting (see: *3.7.1 Implicit type conversion* on page 18).

Here is an example:

```
typeset intm my_array[]       # force all assignments to be
                              # converted to a multi-precision int
typeset string my_array[0,0] # force all assignments to the first cell
                              # to be converted to a string (all other
                              # cells will be intm as above)
```

Using `typeset <type>` in the manner illustrated above with co-ordinates specified will force the creation of the addressed cell if it has not yet been set, and the newly created cell will be initialised to a default value (see: *3.6 Variable declaration and initialisers* on page 15). This syntax should not be confused with other languages where you are requesting an array of a particular data type. There is no such thing in PROSE as an integer array, for example, as collections are typeless. Instead, it is an array of data items. The type of data item is arbitrary and can be controlled on a cell-by-cell basis.

4.6.3 Addressing multiple cells (wildcard indices)

Multiple cells can be addressed if the integer co-ordinate `-1` is used. This has the effect of selecting all cells in that dimension. When setting a value in an array, all cells in the selected dimensions will be set to the same value. When reading a value from an array, the expression is evaluated as if it were a one-dimensional array containing all the objects in the selected dimensions. This construct is meaningless if an array has no cells.

Here are some examples:

```
typeset matrix my_array[10,20]  # set-up array 10 columns by 20 rows
typeset float my_array[0,-1]    # all cells in column 1 are floats
typeset string my_array[1,-1]   # all cells in column 2 are strings

my_array[0,-1] = 3.1415927      # set a value in all column 1 cells
my_array[1,-1] = "american pi"  # set a value in all column 2 cells
print my_array[-1,-1]           # display entire matrix
print my_array[]                # also displays entire matrix
```

It is also possible to use wildcard indices to copy rows and columns to other rows and columns, both within the same array or to another array. Here are some examples:

```
table[-1,2] = table[-1,0]       # copy first row to third row
table[-1,1] = table[1,-1]       # copy second column to second row
```

If wildcard indices are used and you are copying data from one matrix array to another, then it is possible that data will be silently discarded if the destination object is smaller than the source object, except in the case of an automatically resizable array in which case the destination will grow to fit the source data.

If all dimensions are indexed with `-1`, while the result is usually the same as an empty index it should not be confused as having an identical meaning. An empty index references an *entire collection* as one all-encompassing object. All indices set to `-1` references *every data item within the collection*.

4.6.4 Matrix methods

Matrix arrays are supplied with methods that extend their functionality. These methods are invoked using the syntax `matrixname[].methodname()` where `matrixname` is the name of the matrix array variable, and `methodname` the method you wish to call. Methods operate on the entire array unless otherwise stated, in which case the index supplied between square brackets `[]` selects the context.

The most commonly used array methods are summarised in Table 14 below.

`size()`	return total number of cells in the array (multiple of all dimensions), or number of cells in the indexed dimensions
`dimensions()`	return array variable listing matrix dimensions, use `dimensions().size()` to determine how many dimensions a matrix array has
`resize()`	resize array to list of dimensions supplied in the index
`empty()`	return `true` if indexed dimensions are empty, otherwise return `false`
`search(e)`	search matrix for regular expression `e` and return array variable containing co-ordinates of next cell found or `-1` if the search is exhausted (if no index is supplied the entire matrix is searched, otherwise just the selected cells or dimensions)
`next()`	find next cell during a search
`reset()`	if no index is supplied, reset search to first cell in matrix, otherwise reset search to begin at the addressed cell
`sort(`k_1 `...` k_n`)`	sort the array by the dimension selected in the index, and the cells listed in k_1 `...` k_n where these select the rows or columns that form the sort key
`rotate([d])`	where `d` is optional and if not supplied, rotate matrix by 90 degrees, otherwise rotate by `d` where valid values are multiples of 90 degrees and may be positive or negative
`list([d])`	return a single dimensioned matrix array containing a list of all data items in this matrix that matches the supplied index, where `d` defines the optional delimiter character used to separate the dimensions in the list
`rawout(), rawIn()`	convert array to and from a contiguous binary representation of the collection, for storing in flat files or in binary data types

Table 14 - Commonly used array methods

This is not the entire list of available array methods. A complete list and a full discussion of array methods with examples are covered in the *PROSE API Specification*.

4.7 Tree structures

A PROSE tree structure is a collection of objects organised in a hierarchical top-down fashion. The smallest unit in a tree structure is called a *node*. Nodes are connected horizontally on a *branch*, and connected vertically to their *parent node* (immediately above this branch) and to any number of *child nodes* (immediately below this branch). The topmost node is the *root*.

A list of comma-separated expressions supplied in square brackets `[]` (the *index*) identify the path from the root of the tree (at the top) to the node required. A *key* is any expression that can uniquely identify the node on a particular branch. The key may be any primitive data type, but will be converted to a Unicode string before use. The node contains the data item of interest. Here are some examples:

```
my_tree["orchestra"] = "London Symphony Orchestra"
my_tree["orchestra", "string choir", 1] = "Vln. 1"
my_tree["orchestra", "string choir", 2] = "Vln. 2"
print my_tree["orchestra", "*"]     # display nodes directly attached
                                    # to the "orchestra" branch
print my_tree["orchestra", "**"]    # display all nodes under the
                                    # "orchestra" branch (recursive)
print my_tree[]                     # display entire tree
```

4.7.1 Addressing multiple nodes (wildcard indices)

As illustrated in the previous example, the wildcard expressions `"*"` and `"**"` have a special meaning. Similar to the `-1` index in a matrix array, wildcard expressions in a tree index result in multiple nodes being overwritten if used in an assignment, or evaluate to a one-dimensional array containing all referenced objects. Unlike a matrix array, wildcard expressions cannot be used to set the value of nodes that do not already exist.

If the only key supplied is `"**"` then you are referencing every node in the tree. This is not the same as an empty index, which references the entire collection as one all-encompassing object, although the result may be identical.

If you prefix a wildcard expression with a backslash such as `"\*"` the asterisk loses its special meaning. A double backslash should prepend the asterisk `"\\*"` if you wish the value of the node to begin with the characters `"\*"`. The asterisk or backslash character in any other position in the tree index expression has no special meaning.

4.7.2 Data structures

PROSE tree structures can be used to mimic AWK-style associative arrays as well as naturally extending to represent hierarchical data. They can also be used in the absence of C-style structures to group data items together for convenience, such as in the following example:

```
typeset int c_struct[0, "numitems"], c_struct[0, "maxitems"]
typeset pointer c_struct[0, "next_ptr"], c_struct[0, "last_ptr"]

c_struct[0].numitems = 0
c_struct[0].maxitems = 1
c_struct[0].next_ptr:loc = c_struct[1]
```

The previous example is slightly convoluted, you would not ordinarily need to `typeset` every data item before using it. However, the syntax illustrates how tree structures become an extension to the nexus that can be addressed in the same way as any other object in the nexus. This is a very powerful feature, probably the most difficult concept in PROSE to understand, but very rewarding in the flexibility the feature offers. It is discussed in detail in later chapters.

4.7.3 Implementation

Each branch in a PROSE tree structure is implemented as a chained hash table, where the key is passed to a hash function that performs a mathematical algorithm in order to calculate a number that identifies one of many *buckets*. A bucket contains a linked list of all items on the branch that match the hashed key. The bucket is iterated until a successful comparison is found with the actual key (or no match is found). This is a standard approach to implementing hash tables.

The hash tables are complemented with an array that allows a branch both random and ordered access to the data nodes. Ordered access in this context means the nodes in a branch may be iterated without prior knowledge of any keys. The order of sequence matches the order in which the nodes were added to the branch.

The performance of hash tables depend upon the efficiency of the hash function, the randomness of distribution of hashed keys, and the size of the hash table. Once set, a hash table cannot be easily resized nor hash functions modified without a big performance overhead, as all the hash keys in the tree so far would need to be recalculated, and items moved into new buckets. However, a well constructed and optimised hash table is a very fast method of storing and retrieving data that would otherwise require CPU intensive search operations.

4.7.4 Comparison with matrix arrays

Do not confuse matrix arrays and tree structures in the way they store data. Tree structures carry the obvious benefit that data can be indexed by text strings, not just by integer co-ordinates. Matrix arrays however, will always be faster with both read and write operations, and if memory has no bounds or the density of the matrix is high, then it should be chosen in favour of a tree structure. If however, the data cannot be represented in tabular form, or needs to be indexed by text strings, or if implemented as a matrix array would contain a high percentage of empty cells, then a tree structure is a better approach.

Matrix arrays and tree structures are compared further in the following example:

```
my_array[5,3]=35.9    # this is NOT a matrix array, the default
                      # type of collection is a tree structure unless
                      # you use typeset -d collection <type>

delete my_array[]
typeset -d collection matrix
my_array[5,3]=35.9    # this IS a matrix array

typeset tree my_tree
my_tree[5,3]=35.9     # this is a tree structure

print my_array[5,3]   # displays cell at position 5 across, 3 down
print my_tree[5,3]    # displays node "3" on the branch underneath "5"
print my_tree.5.3     # another way of addressing a tree structure
```

In the previous example, the integers `5` and `3` remain as integers in the matrix array and are used as direct co-ordinates. In the tree structure, these integers are converted to strings and used as keys in the underlying hash tables.

4.7.5 Tree methods

Tree structures are supplied with methods that extend their functionality. These methods are invoked using the syntax `treename[].methodname()` where `treename` is the name of the tree variable, and `methodname` the method you wish to call. Methods operate on the entire tree unless otherwise stated, in which case the index supplied between square brackets `[]` selects the context.

The most commonly used methods are summarised in Table 15 below.

`size()`	return total number of nodes in the tree, or in the indexed branches
`empty()`	return `true` if the indexed branches are empty (they have keys but no data items), otherwise return `false`
`children()`	return number of children underneath this indexed node
`search(e)`	search indexed branch for regular expression `e` and return the data item found (if no index is supplied the topmost branch at the root of the tree is searched)
`next()`	find next data item during a search
`reset()`	if no index is supplied, reset search to top of tree, otherwise reset search to begin at the indexed branch
`listOne()`	return a single dimensioned matrix array containing a list of all data items directly attached to the indexed branch
`listSub([d])`	return a single dimensioned matrix array containing a list of all data items attached to the indexed branch and below (recursively), where `d` defines the optional delimiter character used to separate the branches in the list
`setHash(p)`	set the hash function for the indexed branch, where `p` is a pointer to the method (only works if the branch has not yet been initialised)
`buckets([n])`	if `n` is not supplied, return the number of buckets in the indexed branch, otherwise set the number of buckets to `n` (only works if the branch has not yet been initialised)
`rawOut()`, `rawIn()`	convert tree to and from a contiguous binary representation of the collection, for storing in flat files or in binary data types

Table 15 - Commonly used tree methods

This is not the entire list of available tree methods. A complete list and a full discussion of tree methods with examples are covered in the *PROSE API Specification*.

4.8 Collection declaration and initialisers

Collections do not need to be declared before they are first used. However, a declaration is required if you wish to use a collection type that is not the default type (see: *4.4 Default collection type* on page 27).

A collection is declared using the `typeset` keyword. This will not overwrite any contents if the collection already contains data (use the `delete` keyword for this). The index provided in a collection declaration is only significant for a matrix array, where such an index is used to set a fixed size for the array (see: *4.6 Matrix arrays* on page 29). For any other collection type, the declaration index is ignored.

Here are some examples of basic collection declarations:

```
typeset fifo my_stack[]
typeset matrix my_autosize_array[]
typeset matrix my_fixed_array[10,10]
typeset tree my_tree[]
```

As with scalar variables (see: *3.6 Variable declaration and initialisers* on page 15), collections may also be initialised at the time of declaration. However, the syntax for initialising scalar variables is far from adequate for initialising collections, which in PROSE borrows much of its syntax from C.

The initialisers are evaluated at run-time, which means you can include variables in the initialisers if required. Collection initialisers may only be used with the `typeset` keyword, they cannot appear anywhere else in your program code. However, an initialiser does not overwrite previous contents that may already exist in the collection variable.

4.8.1 Initialising stacks and single dimensioned arrays

Stacks and single dimensioned arrays are initialised using a list of comma-separated expressions enclosed in `{ braces }`. An empty expression may be supplied if a data item is to be omitted from the initialisation.

An initialiser on a fixed size array declaration, or any initialiser on a collection that already has data does not need to provide the same number of expressions as items in the collection - any omitted will simply not be initialised. If an initialiser provides items beyond the bounds of a fixed size array, these items are discarded.

Here are some examples of initialisers on stacks and single dimensioned arrays:

```
typeset filo my_stack[] = { 'one', 2, 'three', 4 }
typeset matrix my_autosize_array[] = { 'debussy', , 'elgar' }
typeset matrix my_fixed_array[3] = { 'bach' }

typeset matrix my_autosize_array[] = { my_fixed_array[0], 'chopin' }
print my_autosize_array[]        # displays 'bach chopin elgar'
```

4.8.2 Initialising multi-dimensioned arrays

Multi-dimensioned arrays are initialised using a slightly modified syntax. Here, each dimension is enclosed within its own set of sub braces. Dimensions are also comma-separated, as follows:

```
typeset matrix orchestra[] = { { 'violin', 'viola', 'cello' },
                               { "flute", "clarinet", "english horn" } }
```

As with single dimensioned arrays, any initialiser data items that fall outside the bounds of a fixed size array will be discarded.

4.8.3 Initialising tree structures

Tree structures may have their nodes and branches initialised separately to their data items. Braces may be used in the index of a tree structure to identify multiple nodes for initialisation. The initialiser for the keys (inside the declarator index), is known as the *index initialiser*, while the initialiser for the data items in the expression is known as the *data initialiser*. The order in which items are listed in the initialiser construct is guaranteed to match the order in which the nodes are created on their relative branches.

There are two initialiser forms for tree structures, discussed in the following sections. The two forms are not mutually exclusive, therefore it is perfectly legal to use the index initialiser from the first form, and a data initialiser from the second, or vice versa.

4.8.3.1 First initialiser form (key text)

In the first initialiser form, the index initialiser and data initialiser follow the same format. The entire initialiser construct is enclosed within braces, inside which is a comma-separated list of branch constructors. A branch constructor takes the following form, where square brackets are used to indicate optional symbols:

```
node [ = { branch_constructor } ], ...
```

where `node` is the key (if this is an index initialiser) otherwise a data item (for data initialisers), and if this node is to have a child it is followed by the equals = sign and another node constructor (in its own pair of braces); and so on as required until all branches in the tree have been initialised.

This is best demonstrated in an example:

```
typeset tree orchestra[{
   "London Philharmonic Orchestra" =     # top-level node
         {    "Conductor",               # a first-level branch
              "Choir" = {  "Strings",    # a second-level branch
                           "Woodwind",
                           "Brass" }
         },
   "English Chamber Orchestra" =         # another top-level node
         {    "Conductor",               # a first-level branch
              "Choir" = {  "Strings",    # a second-level branch
                           "Woodwind",
                           "Plucked" }
         }
   }] = \
   \
   { 0 =                                 # assign value to top-level node
         {    "Raymond Leppard",         # and now a first-level branch
              0 = {        "violin1 violin2 viola cello",
                           "flute clarinet oboe",
                           "horn trumpet trombone tuba" }
         },
     1 =                                 # and the other top-level node
         {    "Paul Barritt",            # and a first-level branch
              0 = {        "violin cello double_bass",
                           "oboe bass_oboe uilleann_pipes",
                           "celtic_harp guitar" }
         }
   }
```

4.8.3.2 Second initialiser form (key symbols)

In the second initialiser form, the index initialiser is not identical to the data initialiser. Here, the nodes are treated like extensions to the nexus, and addressed as such (in dot notation) relative to the topmost node in the tree structure. The second form is identified by a construct that begins and ends with a `{[` brace and square bracket `]}`.

The index initialiser is a list of node keys (optionally expressing multiple levels in dot notation) presented as unquoted symbols delimited by new lines or semi-colons. If a data type is to be stipulated, then this is achieved with a further instance of the `typeset` command. If the `tree` data type is given, the construct of that named tree structure is copied into this new structure. Alternatively, an additional initialiser may be supplied inside the index construct for this sub-tree. This is akin to C structures.

The example below demonstrates the second form of a tree structure's index initialiser construct:

```
typeset tree subitems[{[ my_var1; my_var2 ]}]
typeset tree my_api[{[
   typeset int numitems, maxitems, mysub.counter
   typeset string description
   typeset matrix my_array[]
   typeset tree subitems[]     # use tree already defined for subitems[]
]}]
```

The data initialiser may be a comma-separated list of expressions in the order defined in the index initialiser that evaluate to the data item; alternatively it may be a comma, semi-colon or new line separated list of `name = expr` pairs, which are evaluated in the same fashion as a normal variable or collection assignment. In the case of the latter, the `typeset` command may also be used as in a normal assignment, and initialiser constructs for collections may also be specified if required.

The example below demonstrates the second form of a tree structure's data initialiser construct:

```
typeset tree my_api[] = {[
   numitems = 0, maxitems = 10, mysub.counter = 3
   description = "testing the interface"
   typeset matrix my_array[3,2] = { 5, 4, 3 }   # initialise the array
   subitems.my_var1 = 25M                       # one way of referencing a tree
   subitems["my_var2"] = 3.1415927M             # another way of referencing a tree
]}
```

The design of a tree structure's initialiser interface allows, for example, a method to pass an uninitialised tree to an API which stamps its structure on top allowing the caller to then populate it with all of the data that the API requires. This happens dynamically at run-time, allowing structures to change in newer APIs without breaking older code that uses the API. This allows for a great flexibility in the design of API interfaces.

4.9 Collection type conversion

Stacks, arrays and tree structures can be copied from one like variable to another, and between types. When a collection is assigned between variable names of the same collection type, a reference is made. The data items are not duplicated until a write operation occurs on either the source or target data (copy-on-write).

When a collection is subjected to implicit or explicit casting (see: *3.7.1 Implicit type conversion* on page 18 and *3.7.2 Explicit type conversion* on page 19), the collection is converted to the new type. Stacks can be cast to arrays and trees without data loss, as can arrays be cast to trees. Casting in the other direction will usually result in data loss (those data items which cannot be mapped).

When a collection is cast to a scalar variable a different kind of conversion needs to take place, as a list of potentially any number of objects now have to be represented by just one object. The same problem exists in reverse when a scalar variable is converted to become a collection.

4.9.1 Conversion from collections to boolean types

If a collection is cast to a boolean type, then the value is `false` if the collection contains no data items, otherwise it is `true`.

4.9.2 Conversion from collections to numeric types

This is not supported and the result is undefined.

4.9.3 Conversion from collections to string types

A conversion from a collection to a string type results in a string that contains a listing of the contents of a collection, in the same format as displayed using the `print` command or when using one of the collection's `list()` methods.

4.9.4 Conversion from collections to binary data types

A collection is represented in memory as a number of objects that may be stored in different locations in memory, and therefore cannot be easily represented by a single binary scalar type without a change to the underlying data representation. Therefore, when a collection is cast to a binary data type, it is necessary to represent the collection in a binary format that is guaranteed to be contained within one contiguous block of memory, which is the purpose of the collection's `rawOut()` method.

4.9.5 Conversion from any scalar type to a collection

Table 16 below summarises how any scalar type is converted to become a collection.

from scalar to `fifo` or `filo`	new stack initialised, scalar data item is pushed onto the stack
from scalar to `matrix`	new auto-size array initialised, scalar data item assigned to first cell (index `0`)
from scalar to `tree`	new tree initialised, scalar data item assigned to top-level node with the key of `0`
from binary data type to any collection type	the data is assumed to be a contiguous binary representation of a collection, as is returned by the collection's `rawOut()` method, and this process is reversed to reproduce the collection using the `rawIn()` method

Table 16 - Scalar to collection conversions

4.10 Pointers and collections

Pointer variables can be declared that point to objects in the nexus (see: *3.9 Pointers* on page 22). When applied to collections, the pointer will reference the data item that is addressed by the index. If the pointer variable itself is declared with an empty index `[]` then it is called a *collection pointer*, as the pointer references the collection, and may be used as if it were the collection.

Collection pointers allow for several references to one collection that can make changes to the same collection, avoiding the copy-on-write operations. This is illustrated in the example below:

```
typeset matrix my_array[] = {   { "pp", "p", "f", "ff" },
                                { "pianissimo", "piano",
                                  "forte", "fortissimo" } }

typeset pointer ptr[]       # ptr[] is a collection pointer
typeset pointer ptr2, ptr3  # scalar pointers

ptr[]:loc = my_array[]      # ptr[] will behave like my_array[]
ptr2:loc = my_array[0,0]    # ptr2 points to a data item
ptr3:loc = my_array[0,1]    # ...as does ptr3

print ptr[1,0], ptr[1,1], ptr2, ptr3
```

The previous example will display "`p piano pp pianissimo`". It is an error to try and set a collection pointer to reference a scalar variable, or a scalar pointer to reference a collection.

As with scalar pointers, the underlying object is overwritten if an assignment is made to the pointer. In the case of collection pointers, it is the collection itself that is overwritten, not the location of the pointer.

Consider the following example:

```
typeset stack my_stack[] = { 1, 2, 3 }     # first stack
typeset stack my_stack2[] = { 3, 4, 5 }    # second stack
typeset pointer ptr[]

ptr[]:loc = my_stack[]      # point to first stack
ptr[] = my_stack2[]         # overwrite first stack with second stack

my_stack[] = my_stack2[]    # this would have achieved the same effect
```

4.10.1 Collection pointer offsets

As with scalar pointers, the collection pointer has an *offset* attribute. This is referenced using the syntax `<name>[]:offset` where `<name>` is the name of the pointer variable. The offset modifies the behaviour in a type specific manner, but with all types an offset of `-1` effectively turns the feature off, leaving the collection pointer referencing the entire collection. When the offset is turned on, a pointer with an empty index no longer references the entire collection but a data item within the collection.

With stacks, the offset moves the pointer from the topmost item (offset `0`) down the stack. With matrix arrays, the offset moves the pointer from one cell to the next, spanning multiple dimensions if available.

When used on a tree structure, the offset moves the pointer from one node to the next on a single branch. This is the only collection where the index becomes significant, as you can address which branch you wish to iterate when setting the offset. Tree structures also expose the *key* attribute, referenced using the syntax `<name>[]:key` where `<name>` is the name of the collection or pointer variable. This allows you to determine both the key as well as the data item when pointing to an arbitrary node.

Here are some examples of using offsets with collection pointers:

```
typeset matrix prime[] = { 1, 2, 3, 5, 7, 9, 11, 13, 17 }
typeset tree library[{"Composers" = { "Purcell", "Handel", "Haydn" }}]
typeset pointer ptr[]

ptr[]:loc = prime[]
ptr[]:offset = 3                        # point to 4th cell
print ptr[], ( ptr[]:offset++, ptr[] ) # display 4th and 5th cells

typeset tree library[] = {0={ "Henry", "George Frideric", "Joseph" }}
ptr[]:loc = library[]
print ptr["Composers"]:key              # display "Composers"

ptr["Composers"]:offset = 2             # "Composers" branch, 3rd node
print ptr[]:key ", " ptr[]              # display key and data item
```

The previous example will display the following:

```
5 7
Haydn, Joseph
```

4.10.2 Declaring collection pointer target types

As with scalar pointers, it is usually desirable to declare the type of collection that is to be referenced by a collection pointer, when this is possible. This is achieved using the form `typeset pointer <name> to <collection>` where `<name>` is a comma-separated list of collection pointer names followed by empty indices `[]` to be declared, and `<collection>` the type of collection to be referenced. It is an error to try and set a collection pointer declared in this fashion to point to another collection type.

Here is an example:

```
typeset fifo my_stack[]
typeset filo my_stack2[]
typeset pointer ptr[] to fifo   # can point to a fifo only
ptr[]:loc = fifo                # this works
ptr[]:loc = filo                # this generates an error
```

4.10.3 Collection pointer masquerading

Unlike scalar pointers, it is not possible for a collection pointer to masquerade as another collection type, as the overhead of converting from one collection type to another is too cumbersome.

4.11 Attributes and collections

Object attributes are referenced in the same way for a collection variable as for a scalar, using the form `<collection>[<index>]:<attribute>` where `<collection>` is the name of the collection, `<index>` identifies the cell or dimension (wildcard), and `<attribute>` the name of the attribute. This identifies an attribute associated with the object in the indexed cell of the collection.

Attributes may also be associated with a cell or node itself, and which remain with the cell regardless of the data items that are stored in it. In matrix arrays, these are called *cell wall attributes* (node attributes in tree structures). They are referenced using the form <collection>[<index>]::<attribute>.

Cell wall attributes have a number of uses, for example, it is possible to associate methods with a cell wall that are invoked whenever the contents of the cell are modified, i.e. programmatical side-effects. This is the subject of a later chapter.

Dimensions (and branches) and the overall collection have attributes too. To reference dimension attributes, provide a wildcard index when using the double-colon :: syntax. To use collection attributes, omit the index.

Here are some examples:

```
my_tree["top","branch"]:my_attr = 1    # object attribute
my_tree["top","branch"]::my_attr = 1   # node attribute
my_tree["top","*"]::my_attr = 1        # branch attribute
my_tree[]::my_attr = 1                 # collection attribute

typeset matrix my_array[]
my_array[2,4,3]:my_attr = 1            # object attribute
my_array[2,4,3]::my_attr = 1           # cell wall attribute
my_array[2,4,-1]::my_attr = 1          # dimension attribute
my_array[]::my_attr = 1                # collection attribute
```

The double-colon syntax may actually be used on any object reference, which includes scalar variables. This is because scalar variables are objects that are attached to a tree structure - the nexus - inside nodes. These nodes have attributes that are referenced using the form <object>::<attribute> where <object> is the name of the object (e.g. variable name) and <attribute> the name of the attribute.

To improve performance, a unique memory storage location is not given to an attribute until it is changed from its default state. The default state of an attribute is defined by the type of the object which the attribute is attached to. This means that the attribute mechanism in PROSE is light-weight, and a programmer should not worry about PROSE using more memory than is appropriate for the size or complexity of the program module.

5 Operators and expressions

5.1 Introduction

In a natural language, nouns and collective nouns are cemented together with a variety of other words used in the construction of a sentence. In a programming language, these are operators and expressions. These allow you to not only identify data items, but to instruct the computer in their use.

5.2 Expressions and precedence

5.3 Mathematical operators

Mathematical operators may be applied to any data type. It is up to the data type API to determine how to interpret the operation, and may at its choosing, deem the operation an error. The operators and their various effects are summarised in Table 17 below. Except where noted, the result of the expression is the same as the result of the mathematical operation.

`x++` *result is x before the operation*	numeric types: increment by 1 string types: append `local.ofs` (a space by default) collections[-1]: increment operation on every indexed cell
`x--` *result is x before the operation*	numeric type: decrement by 1 string types: discard last word (up to last `local.ofs`) collections[-1]: decrement operation on every indexed cell
`++x` *result is x after the operation*	same effect as `x++`
`--x` *result is x after the operation*	same effect as `x--`
`-x, +x`	numeric types: arithmetic negation, plus
`x + y`	numeric types: mathematical addition string types: concatenation collections[]: combine the datasets from the two collections collections[-1]: add operation on every indexed cell, combine cells where they are unique to a collection
`x - y`	numeric types: mathematical subtraction string types: remove `y` from end of `x` if present collections[]: remove `y` dataset from `x` where it matches collections[-1]: subtract operation on every indexed cell
`x * y`	numeric types: mathematical multiplication collections[-1]: multiply operation on every indexed cell

`x / y`	numeric types: mathematical division collections[-1]: divide operation on every indexed cell
`x ^ y`	numeric types: raise `x` by the power of `y` collections[-1]: power operation on every indexed cell
`x % y`	numeric types: divide `x` by `y` and return the remainder collections[-1]: remainder operation on every indexed cell
`!x`	numeric types: bitwise not collections[-1]: bitwise not on every indexed cell
`x & y`	numeric types: bitwise and collections[]: combine the two datasets, `x` has precedence where both collections share cells collections[-1]: bitwise and on every indexed cell
`x | y`	numeric types: bitwise or collections[]: combine the two datasets, `y` has precedence where both collections share cells collections[-1]: bitwise or on every indexed cell
`x || y`	numeric types: bitwise exclusive-or (xor) collections[]: combine the two datasets, where both collections share cells, leave these index positions empty collections[-1]: bitwise xor on every indexed cell
`x << y`	numeric types: bitwise shift left (`x` moves by `y`) collections[-1]: bitwise shift left on every indexed cell
`x >> y`	numeric types: bitwise shift right (`x` moves by `y`) collections[-1]: bitwise shift right on every indexed cell
`x = y`	numeric types: numeric assignment string types: string assignment collections[]: for copying entire collections collections[-1]: copy only indexed cells
`x += y, x -= y,` `x*= y, x /= y, x ^= y,` `x %= y, x &= y,` `x |= y, x ||= y,` `x <<= y, x >>= y`	assignment with operation, e.g. `x += y` is shorthand for `x = x + y`

Table 17 - Mathemical operators

5.4 Comparison operators

This set of operators are used with conditional statements and expressions, as defined with the `test` and `if` keywords (see: *6.7 Conditional code* on page 63), and are summarised in Table 18 below.

`=`	equal to (the same effect is achieved if this operator is omitted)
`<`	less than
`<=`	less than or equal to
`>`	greater than
`>=`	greater than or equal to
`~=`	approximately or probably equal to
`~<`	probably less than
`~<=`	probably less than or equal to
`~>`	probably greater than
`~>=`	probably greater than or equal to
`in(<expr>, ...)`	equality with one of a list of comma-separated expressions
`~in(<expr>, ...)`	approximate or closest equality with one of a list of expressions
`like(<expr>, ...)`	similar to one of a list of comma-separated expressions

Table 18 - Comparison operators

All comparison operators evaluate to `true`, if the comparison succeeds, or `false` otherwise. Every data type in PROSE supports all operators. All user-defined data types must also support all operators, but are free to interpret the comparison in whatever type dependent fashion is appropriate.

The operators with no `<expr>` specified in Table 18 are always followed by a single secondary expression, the remaining operators may be followed by multiple secondary expressions separated by commas and enclosed in optional parentheses. The primary expression is defined in a statement dependent way. These are primary and secondary expressions in the context of conditional code (see: *6.7 Conditional code* on page 63).

The operators =, <, <=, >, >= are exact or definite comparisons, although when applied to inprecise floating-point numbers they can only be as accurate as the underlying type's mathematical precision allows.

The operators ~=, ~<, ~<=, ~>, ~>= are approximate or probable comparisons, and their interpretation varies, depending upon the data type being used. For example, the Unicode string approximate comparisons are identical to the exact comparisons except that uppercase and lowercase characters are assumed equal. This set of operators also helps the introduction of user-defined numeric types such as intervals and complex numbers (see: *3.10 User-defined types* on page 24).

The `in` operator is followed with multiple secondary expressions, comma-separated, and enclosed within optional parentheses. This allows for testing a data item for equality with one of many possible expressions. Once a match is found, no more expressions in the list are tested, and the comparison returns `true`.

The `~in` operator is similar to the `in` operator except that the tests performed on the multiple secondary expressions use ~= instead of = and test every expression to find the closest possible match. This relies on the underlying data type API supporting the `proximity()` method, that returns a number indicating the accuracy of an equality match, where an exact match is `0`. This is discussed further in the *PROSE API Specification*.

The `like` operator is similar to the `~in` operator except that the comparison is a type-specific similarity test. Every expression is tested to find the most similar match. For example, the Unicode string similarity test allows for *regular expressions* to be used for finding data that matches a particular pattern.

The `in`, `~in` and `like` operators use the variable `local.list` to store the index number of the list expression that caused the comparison to succeed, where `0` is the first expression, `1` the second expression etc. If the comparison failed, `local.list` yields `-1`.

5.4.1 Boolean comparisons

The built-in boolean type only supports definite equality comparisons. Greater than, less than, and probable comparisons make no sense with boolean types, nor does providing a list of possible matches - as the only possible matches are `true` and `false`. A comparison operator other than a definite equality used on a boolean type produces undefined results.

5.4.2 Numeric comparisons

Numeric type comparison is mathematical. The built-in PROSE numeric types do not differentiate between exact and probable comparison operators, which yield the same results. To support approximate or probable numeric comparisons, you must create a user-defined type such as an interval.

Numeric similarity comparisons with the `like` operator discards precision in the test. That is, the precision used in comparison will match the number with the least precision. Numbers are rounded toward the nearest whole number where precision needs to be lost to make such a comparison. Therefore, `3.142` is like `3.1415927`, `[1/3]` is like `0.333` and `[2/3]` is like `0.6667`. Normal equality tests would fail these comparisons.

Table 19 below illustrates some valid numeric comparison operations that may be used with conditional statements and expressions.

`3`	equal to `3`, same as `= 3`
`< 4.53M`	less than `4.53` (multi-precision float)
`>= [2/3]`	greater than or equal to two-thirds (rational)
`in(1, 3, 5, 7, 9)`	equal to one of the listed integers
`like(3.14, 3.142, 3.1416)`	ignore precision in the comparison, `local.list` will indicate which test matched more precisely

Table 19 - Example numeric comparison operators

5.4.3 String comparisons

Comparisons performed on the Unicode string type are lexicographical comparisons, that is based upon the ordering of the Unicode character set. An equality comparison evaluates to `true` if the strings are identical lengths, and the Unicode characters making up the two strings are identical and in the same order. Otherwise, a string is less than or greater than another under either of the following two circumstances:

- The strings are equal up to some character position, and at that first differing character position the Unicode value of the first string is less than or greater than the Unicode value of the second.

- The strings are equal up to some character position where the first string terminates (less than) or the second string terminates (greater than).

Approximate comparison operators are used on Unicode strings to ignore case differences in the lexicographical comparisons described above, that is an uppercase Unicode character will equate to its lowercase equivalent.

Similarity comparisons with the `like` operator on Unicode strings allow for the use of *regular expressions*. These can be used to identify strings that match a particular pattern.

Table 20 below illustrates some valid string comparison operations that may be used with conditional statements and expressions.

`"Caesar, Julius"`	lexicographically equal to `"Caesar, Julius"`, same as `= "Caesar, Julius"`
`< "Cromwell, Oliver"`	lexicographically less than `"Cromwell, Oliver"`, for example `"caesar, julius"` would fail when compared with this, but `"Caesar, Julius"` would succeed
`~= "Caesar, Julius"`	would match `"caesar, julius"` as well
`~< "Cromwell, Oliver"`	`"caesar, julius"` would now succeed when compared with this
`in("Caesar", "Clarence", "Cromwell")`	matches any of the listed strings (case sensitive), use `~in` for a case insensitive match

Table 20 - Example string comparison operators

5.4.4 Binary data comparisons

Only definite and approximate equality comparisons are supported on a binary data type. Any other comparison operator used on this type will produce undefined results.

The equality comparison = will compare the sizes and then each individual byte between the two data items. The result is `true` only if the data items are identical.

The approximate comparison ~= will compare the sizes, and then calculate a checksum on the two data items. The result is `true` if the sizes and checksums match. A *checksum* is a mathematical algorithm that can be performed on a set of data (in this case the bytes that make up the data item) and which results in a number that uniquely identifies a much smaller set of data. The checksums are guaranteed to match if the data is identical, and there is a high probability that the checksums won't match if the data is not identical. The approximate comparison therefore provides a reasonable certainty of accuracy in the result, without the computing overhead of a byte-by-byte comparison.

5.4.5 Collection comparisons

Only the comparison operators =, ~=, `in` and `~in` are supported in the context of a collection comparison, where any other comparison operator yields undefined results.

An equality comparison = between two collections determines if the collections are identical. For this comparison to return `true`, each of the following tests (in this order) must all be true:

- Collection types are identical
- Total number of objects in the collections are identical
- Dimensions (matrix array) are identical, key names (tree structure) are identical
- Data types, object sizes and their location in the collections are identical
- An equality comparison between every object in the collections returns `true`

An approximate equality comparison ~= between two collections performs the following tests:

- Collection types are identical
- Total number of objects in the collections are identical
- Dimensions (matrix array) are identical, key names (tree structure) pass approximate equality test
- An approximate equality comparison between every object in the collections returns `true`

The `in` and `~in` comparison operators when applied to collections perform equality and approximate equality comparisons respectively, as described above on multiple collections. The `like` operator performs similarity comparisons on all objects in the collections.

Table 21 below illustrates some valid collection comparison operations that may be used with conditional statements and expressions.

`my_collection`	identical to the collection named `my_collection`, same as = `my_collection`
`~= my_collection`	all data items approximately equal to those in the collection named `my_collection`

Table 21 - Example collection comparison operators

5.4.6 Pointer comparisons

Pointers behave exactly the same as the data type they are pointing to when used in comparisons (see: *3.9 Pointers* on page 22 and *4.10 Pointers and collections* on page 41). It is legal to compare pointers with pointers, as well as data items with pointers.

Use the pointers' `location` (or `loc`) attribute if you wish to determine if two pointers are referencing the same object, where only the equality operator is supported (any other comparison operator on a `loc` yield undefined results).

Table 22 below illustrates some valid pointer comparison operations:

`ptr`	data item pointed to by the `ptr` pointer variable passes the equality test, same as = `ptr`
`ptr[]`	collection pointed to by the `ptr[]` variable passes the collection equality test, same as = `ptr[]`
`< ptr`	data item pointed to is less than
`~> ptr`	data item pointed to is approximately (or probably) greater than
`in(ptr1, ptr2)`	equal to one of the data items pointed to by `ptr1` and `ptr2`
`ptr:loc`	scalar pointer location is identical, same as = `ptr:loc`
`ptr[]:loc`	collection pointer location is identical, same as = `ptr[]:loc`

Table 22 - Example pointer comparison operators

5.5 Logic operators

This set of operators are used with conditional statements and expressions, as defined with the `test` and `if` keywords (see: *6.7 Conditional code* on page 63). Logic operators prefix comparison operators, and are summarised in Table 23 below.

`<cmp> and <cmp>`	`true` if both comparisons evaluate to `true`
`<cmp> or <cmp>`	`true` if one or other comparison, or both, evaluate to `true`
`<cmp> xor <cmp>`	`true` if one or other condition, but not both, evaluate to `true`
`not <cmp>`	`true` if comparison evalutes to `false` (may also prefix `and`, `or`, `xor` operators)

Table 23 - Logic operators

The use of `(` parentheses `)` is also permitted in any position where a logic operator is valid. These group logical operations together. Without parentheses, the `not` operator has a higher precedence than `and`, which has a higher precedence than `or` and `xor` (which have an equal precedence).

Therefore the pseudo-expression:

```
<cmp1> and not <cmp2> or <cmp3> xor <cmp4>
```

is equivalent to:

```
( <cmp1> and ( not <cmp2> ) ) or <cmp3> xor <cmp4>
```

The latter is also more readable, and the PROSE programmer is encouraged to use parentheses habitually to make code logic as clear as possible for others to read.

6 Statements and keywords

6.1 Introduction

Statements in a programming language are like sentences in a natural language. A programming language that only allows for the manipulation of data storage locations, no matter how powerful the expressions, is incomplete without the ability to make compound expressions, iterate data items, draw comparisons between data, and run conditional program code depending upon the results.

This chapter examines the powerful set of statements and keywords available in PROSE that perform tasks or operations that cannot be expressed in any other way, or are commonly required and provide a convenient standard interface that adds a degree of expressiveness to the language.

6.2 Regular statements and code blocks

A regular statement in PROSE is one or more completed expressions, where an expression is a list of commands or operators that are evaluated to form a *result*. Statements are terminated by a newline character or semi-colon at the end of an expression. Statements may span multiple lines. If the line break is to occur at a point in the expression that might be a valid end of expression, the statement must include a continuation character, which is a backslash \ immediately before the line break. Here is an example:

```
print "this statement spans
  multiple lines because we are currently
  in double-quotes, so the expression
  has not yet ended"

print "this statement spans " \
  "multiple lines, but not because the " \
  "expressions have not ended, but because " \
  "we are using the continuation character"
```

Although not required, it is good form in PROSE to surround multi-line expressions in parentheses, if this adds clarity to the code. This can also be used to reduce the number of continuation characters used, demonstrated below:

```
print ( "this statement spans multiple lines, "
   "not because there is only one string expression, "
   "and not because we are using the continuation character, "
   "but because all of the string expressions are "
   "contained within one outer set of parentheses" )
```

The semi-colon `;` character is used to break up a single line into multiple statements. The comma `,` is used to create expression lists, that is a list of expressions in a single statement. The result of the last expression in the list becomes the result of the entire expression list:

```
a = 35; b = 36       # two statements on one line
c = (++a, ++b)       # increment a and b, result of the expression list
                     #    is the result of the last expression (37)
```

Code blocks are collections of statements contained within { braces }. They are used to maintain a context for a section of code, such as the statements to execute when a particular condition is met, and are discussed further in the following sections of this chapter. Code blocks can always be used in place of a single regular statement, and vice versa. The number of code blocks currently opened is known as the *encapsulation level*, and this is tracked by PROSE compilers and can help identify structural errors in program code.

6.3 Compound statements

A compound statement is where two regular statements have been joined onto one line without a semi-colon separating them. This cannot be done arbitrarily, but where this is supported to make code more succinct, it will be explicitly stated as such in this specification.

6.4 Data type management

6.4.1 Type query (typeof)

The `typeof` expression in the form `typeof(<name>)` evaluates to a number which uniquely identifies the data type of variable `<name>`, where the parentheses are optional. Data types may not have the same numeric identifier from one platform to another so make no assumptions regarding the result of this expression.

The expression may also be used in any position where a data type is expected, including with the `typeset` and `typedef` keywords, as well as in explicit casts as illustrated in the example below:

```
typeset int myvar1
typeset (typeof myvar1) myvar2  # myvar2 to have same type as myvar1
myvar3 = (typeof myvar2) 35     # 35 is cast to same type as myvar1
```

When used with pointers, the `typeof` keyword determines the type of the target data type. There is no numeric identifier for a pointer itself. Here is an example of a function that accepts a pointer argument and then sets a local variable based on the target type:

```
function borrowType(pointer ptr)    # pointer argument, any type
{
    typeset (typeof ptr) localvar   # localvar set to same type
    localvar = 5                    # this ensures that 5 will be
                                    # cast to the appropriate type
    return localvar                 # returns same type as argument
}
end function
```

It is also possible to request that a pointer masquerades as another dynamic type while storing the data as a fixed type, for example:

```
typeset pointer ptr cast typeof(myvar1) # masquerade as myvar1's type
ptr:loc = myvar2                        # but store in myvar2
```

6.4.2 Type declaration (typeset)

The `typeset` keyword is used to provide an optional declaration for variables as well as a set of initialisation constructs (see: *3.6 Variable declaration and initialisers* on page 15 and *4.8 Collection declaration and initialisers* on page 36). Additionally when used for declaring pointer variables, the `to` and `cast` modifiers may be used to change the behaviour of the pointers, declaring the target pointer type or enabling pointer masquerading (see: *3.9 Pointers* on page 22 and *4.10 Pointers and collections* on page 41).

Additional options may be specified in a `typeset` declaration that further modify the behaviour of the declared variables. These options, defined in Table 24 below, must appear immediately following the `typeset` keyword before the type name. If multiple options are to be supplied, use only a single hyphen - followed by each option letter (with no spaces or commas).

`-g`	global (module-level) context, these variables will be accessible outside of the current function or method
`-s`	static context, even if this `typeset` declaration is parsed several times (e.g. recursive function calls), these variables will be stored in only one location and accessible across all invocations
`-v`	volatile context, the contents of these variables are subject to change by external code or libraries without using standard PROSE APIs, and therefore the contents of these variables must not be cached

Table 24 - Type declaration options

6.4.3 Setting default bahaviour for typing (typeset -d and typeset -gd)

It is possible to modify the default behaviour of various data type management functions using the form `typeset -d <default> <assign>` where `<default>` is the name of the default behaviour to change, and `<assign>` sets the new behaviour. Table 25 below summarises the supported values for `<default>`, which are then discussed in further sections below.

`options`	default declarator options
`constant`	default data types for constants
`nosupport`	default behaviour when a declarator option is unsupported

Table 25 - Type defaults

The scope of the default behaviour is the same as that of a type declaration. To set a default with global (module-level) scope, use `typeset -gd` instead of `typeset -d`.

6.4.3.1 Default declarator options

Data types may support declarator options which modify their behaviour. By default, PROSE supports options on a number of primitive data types (see: *3.6.1 Declarator options* on page 16). The default behaviour for these types may be modified using the form `typeset -d options <type> (<options>)` where `<type>` is the name of the data type and `<options>` a comma-separated list of default options to assign.

Here is an example of modifying the default type for integer declarations:

```
j = -101M                          # set j to multi-precision integer

typeset int i = 75                 # declare 64-bit signed integer
print (int)j                       # cast j to 64-bit signed integer

typeset int(8) i = 75              # declare 8-bit unsigned integer
print (int(8))j                    # cast j to 8-bit unsigned integer

typeset -d options int(8)          # change default to 8-bit unsigned
typeset int i = 75                 # declare 8-bit unsigned integer
print (int)j                       # cast j to 8-bit unsigned integer
```

Changing the default declarator option for a primitive data type only applies to variables that are explicitly declared or cast. The data type of a variable that has not been declared before use is determined based upon the type of the resulting assignment (see: *3.6 Variable declaration and initialisers* on page 15). To change the behaviour of implicit type conversion, one must change the interpretation of primitive constants, as described below.

6.4.3.2 Default constant data types

The primitive constants (see: *3.5.1 Constants* on page 11) may be changed from their default data types to other types using the form `typeset -d constant <expr>=<type>` where `<expr>` is one of the default constant expressions listed in Table 26 below, and `<type>` the data type (with options in parentheses if required) to set as the default type for this constant.

`0`	fixed precision integer constants, defaults to `int(-64)`
`0M`	multi-precision integer constants, defaults to `intm`
`[0]`	single unit constants, defaults to `rational`
`[0/0]`	double slashed unit constants, defaults to `rational`
`0.0`	fixed precision floating-point constants, defaults to `float(-64)`
`0.0M`	multi-precision floating-point constants, defaults to `floatm`
`""`	double quote constants, defaults to `string`
`''`	single quote constants, defaults to `string`
`` `` ```	angled quote constants, defaults to `string`
`[0, ... ]`	multiple comma-separated unit constants, no defaults
`[0/ ...]`	multiple slashed unit constants, no defaults

Table 26 - Default constant expressions

In its most simple form, this feature allows for integers or floating-point numbers to default to a different precision, rather than the minimum signed 64-bits. Here is an example:

```
typeset -d constant 0 = int(8)
print -1            # displays 255 because the default integer
                    # constant has now been set to unsigned 8-bits

typeset -d constant 0.0 = float(32)
                    # floating point constants reduced to a
                    # minimum of 32-bits precision
```

Another advantage of this feature is that it becomes easy in PROSE to work with multi-precision numerics all of the time if required, without having to postfix every constant with the letter `M`, as illustrated in the following example:

```
typeset -d constant 0 = intm
typeset -d constant 0.0 = floatm(256)
                    # all integer and floating-point constants
                    # will now be multi-precision, floats will have
                    # a minimum precision of 256-bits

a = 3.14159265358979323846 * 367100.31108
                    # this arithmetic will now be handled entirely
                    # by the GNU MP library, and a is floatm
```

Rational numbers make use of two different types of default constant expressions, `[0]` and `[0/0]`. User defined types generally make use of constants of the form `[0,0]` or `[0,0,0]` etc. where the number of comma-separated units inside the constant might default to different types, e.g. `typeset -d constant [0,0] = complex` would set the default type of any two unit constants to the user-defined `complex` type, while `typeset -d constant [0,0,0] = version` would not clash with the `complex` type but in addition a three unit constant defaults to the user-defined `version` type (see: *3.10 User-defined types* on page 24).

Units themselves are interpreted as primitive constants, so if the integer constant type has been changed then this also affects unit constants that make use of integers, e.g. the integers in the rational number constant `[1/3]` will be the same default type as integers `1` and `3` used outside of the construct. It is not possible to set the default type of a unit constant based upon the types of the units inside, i.e. following a `typeset -d constant [0,0] = complex` all two-unit constants will default to the `complex` type, including `[3,4]`, `[3.1, 5]` and `["3.8", "21"]`.

Multiple unit constants with slashes separating the units may also be set to different types, e.g. rational number constants may be changed, or a new type may be associated with the constant of the form `[1/3/10]` for example, using `typeset -d constant [0/0/0] = new_type`.

String constants inside the three kinds of quotation mark by default are handled by the same type, but may be separated if required. Therefore a new type `babelfish` might be set as the default for string constants in single quotes, a new type `evaluate` for string constants in angled quotes, while the standard `string` type remains for those in double quotes, as demonstrated below:

```
typeset -d constant "" = string     # this is the default anyway
typeset -d constant '' = babelfish
typeset -d constant `` = evaluate
```

6.4.3.3 Unsupported declarator options

If an unsupported declarator option is used with a type declaration, e.g. a requested precision that cannot be met; there are several possible actions that can occur. These are summarised in Table X below.

`<type>()=<type2>`	use alternative type
`<type>()=OTHER`	use the most similar supported option instead
`<type>()=ERROR`	generate an error

Table 27 - Options for nosupport conditions

By default, fixed-precision numeric types with arbitrary precision equivalents will fall back on the MP type. For example, if `float(128)` is requested but the maximum fixed precision that the platform can represent is 64 bits, the data type will be set to `floatm`. This behaviour may also be explicitly requested e.g. `typeset -d nosupport float()=floatm`.

A PROSE implementation is at liberty to select a higher number of bits if required to meet the needs of the requested precision, e.g. `int(-40)` may very well be represented in a signed 64-bit integer. Therefore PROSE programmers are ill-advised to rely on predictable overflow behaviour, and should instead use bit masks if an exact number of bits are required.

It is also reasonable to request that another, most similar, declarator option is used in place of one that is not supported. For example, you may wish for a request `int(64)` to result in only an unsigned 32-bit integer on platforms that cannot normally represent unsigned 64 bits. This behaviour is requested using the form `typeset -d nosupport int()=OTHER`.

To generate an error if an option is unsupported, use `typeset -d nosupport int()=ERROR`.

6.4.4 Type definition (typedef)

A new type is defined by providing a type name, and the data type function entry point. The entry point may be a relative name, or an absolute name anchored to the root of the nexus. A type may also be redefined using the `typedef` command. Here are some examples:

```
typedef evaluate api.support.type.evaluate() # define new type
typeset -d constant `` = evaluate            # set-up a constant type
my_var = `my first evaluate string`          # and use it

typedef int api.support.type.new_int()       # redefine int type
```

If a data type API is replaced, all operations on existing variables of that type will be passed to the new API too. It must therefore be compatible with the type it is replacing, and understand the underlying data representation.

Defining new data types is discussed at greater length in the *PROSE API Specification*.

6.5 Data manipulation

There are two basic types of structure in the PROSE programming language. They are the building blocks that every data type relies on. These are objects and attributes (see: *3.2 Variable names and attributes* on page 10).

Every time a variable is referenced, an object is manipulated on the nexus. All data items in scalar variables and collections are examples of such objects, as are functions and methods. An object therefore contains the data types previously discussed in this specification. Referencing by object is actually satisfied by an internal reference by attribute. That is, a default attribute name matching the name of the object defines the contents of the object, as returned when the object name is referenced. Therefore, when the variable `my_var1` is assigned a value, this value is actually assigned to the attribute `my_var1` in the object of the same name, and would therefore have the same effect as assigning to a variable name `my_var1:my_var1`.

Objects then, contain attributes, which can be individually addressed and as a whole define the object. Attributes have names and values, those values can be any PROSE data type, and can be used in the same way as variables. Objects belong to different *classes*, where a class is a definition of what attributes must be defined, and which may optionally be defined. Object classes are defined in a *schema*. For example, every object in PROSE must support the `description` attribute, which takes a Unicode string and may be used for documenting the purpose or use of the object.

The only difference between referencing an object and its underlying attributes is in the API implementation. The underlying data type API uses attributes to define the object, and therefore manipulation of the underlying attributes allows for greater control of the object, but requires knowledge of how the data type is defined at a more fundamental level. For example, assigning to `my_var1:my_var1` will never change the data type, which is defined by a different attribute in the `my_var1` object. Data type APIs can also set traps on attributes so that they may have specific side-effects. The use of attributes is discussed in greater detail in a later chapter.

The statements defined by PROSE for manipulating data can be used on both objects and attributes. They are summarised in Table 28 below.

`copy <expr> to <expr>`	copy an object or attribute from one location to another
`move <expr> to <expr>`	move an object or attribute from one location to another
`swap <expr> with <expr>`	swap the locations of two objects or attributes
`delete <expr>`	delete an object or attribute
`print <expr>`	display contents of an object or attribute
`push <expr>` `push -m <expr>`	push object onto the local stack (use `-m` to push a marker onto the stack)
`pull <expr>` `pull -m <expr>`	pull object from the local stack (use `-m` to pull a marker from the stack, dropping all items pushed on top of the marker)
`peek <expr>`	peek at the item on top of the local stack without pulling

Table 28 - Data manipulation statements

For the purpose of this chapter, you may assume that attributes are just extended variables. However, it is safer not to use attributes that have not been specifically documented, as their side-effects may not be obvious.

6.5.1 Copying data (copy-to)

The `copy-to` statement is identical in effect to a variable or collection assignment and takes the form `copy <source> to <dest>` where `<source>` is a list of one or more comma-separated object names to copy from and `<dest>` a list of one or more comma-separated object names to copy to.

As with variable assignments, the copy operation is a copy-on-write algorithm, which means that the data will only be copied by reference until a write operation occurs to either copy, at which point the data that is changing is branched off into its own copy. As with collection assignments, wildcards may be specified in the collection indices, which can evaluate to a one-to-one, one-to-many, many-to-one or many-to-many operation.

The following example demonstrates the use of the `copy-to` statement on scalar objects:

```
my_string1 = "Oberon and Titania"
copy my_string1 to my_string2         # my_string2 = my_string1
copy "another test string"            # possible because constants are
             to my_string3            # also objects

typeset matrix my_array[3,3]
copy my_array[] to my_array2[]        # copy entire array
copy my_array[0,-1] to my_array2[]    # copy first row only
copy my_array[1,1] to my_array2[2,1]  # copy one cell to another

copy (s1, s2) to (s3, s4)             # s3 = s1, s4 = s2
copy (s1, s2) to s3                   # s3 = s1, s3 = s2 (pointless)

typeset fifo my_stack[]
copy (s1, s2, s3) to my_stack[]       # copy several items onto
                                      # a stack in one go
```

6.5.2 Moving data (move-to)

The syntax of the `move-to` statement is identical to the `copy-to` statement. The operation is similar, except that data is moved from one location to another. The source object becomes unassigned after a move operation. This is similar to a `copy-to` followed by a `delete` on the source, but can be optimised further by the data type API.

6.5.3 Swapping data (swap-with)

The `swap-with` statement is identical in syntax again to the `copy-to` keyword. However, a `swap-with` operation takes the data in two objects and moves them into each other's location, effectively swapping the values or cells.

Wildcard indices are permitted when swapping between collections. Data will be lost if the number of cells are not identical on both sides of the `swap-with`.

6.5.4 Deleting data (delete)

The `delete` statement takes a comma-separated list of objects. The `delete` operation unassigns a variable and releases any memory that it used. The data in that variable or collection is permanently lost.

An empty collection index will delete the entire collection. A wildcard index will delete a number of referenced cells. A single index will delete the contents of a single cell.

The `delete` keyword is short-hand for invoking an object's `delete()` method.

6.5.5 Displaying data (print)

The `print` keyword provides a generic interface for displaying any kind of object in the nexus. The way the data type is displayed is entirely type-specific and defined by the data type API. The `print` keyword provides a means of tying together the `print()` methods from one or more data types listed in an expression.

6.6 Local stack operations

The local stack is a temporary data storage area with the same context as a local variable. Objects may be pushed onto this stack, and pulled off whenever required. The stack is flushed whenever a function or method returns. This is not related to any internal stacks, and is provided purely for speed and convenience. It will often be faster to use the local stack for a commonly requested variable, as this can be optimised by the compiler for fast access.

6.6.1 Pushing data onto the local stack (push)

The `push` statement takes a comma-separated list of objects to push onto the local stack. Collections with wildcard indices are supported for pushing multiple objects onto the stack in one go. For example:

```
my_string = "test string"
push my_string, "another test"

typeset filo my_stack[]
my_stack[] = "Mozart"
my_stack[] = "Beethoven"
push my_stack[-1]           # same as push my_stack[], my_stack[]
```

The `-m` modifier used with `push` statement identifies the data item as a marker object. This behaves like any other object pushed onto the stack, but may be specifically requested in a later `pull` regardless of how many items have been pushed on top.

6.6.2 Pulling data from the local stack (pull)

The `pull` keyword may be used in expressions, and takes objects off the local stack that have been previously pushed. Without arguments, the `pull` keyword evaluates to the data item much like referencing a variable, and then the data item is dropped. Alternatively, supply a comma-separated list of variables that should be assigned each respective `pull` from the stack. For example:

```
print pull                      # pulls, displays, and drops data
my_string = pull                # pulls and assigns to a variable
pull my_string, my_string2      # pulls and assigns to two variables
```

The `-m` modifier used with the `pull` keyword pulls a previously marked object from the local stack. All items pushed onto that marked object are discarded. Consider the following example:

```
push "Cantus Inaequalis", "Cantus Insolitus"
push -m 0
push "In Caelum Fero", "Cantus Iteratus"
pull -m 0
pull playstring     # playstring contains "Cantus Insolitus"
```

6.6.3 Peeking at data without pulling (peek)

The `peek` keyword uses the same syntax as `pull`. The `peek` operation is identical to `pull` except that the data item remains on the local stack. If a list of variables are provided with the `peek` keyword, then all variables will be set to the same value.

6.7 Conditional code

PROSE provides a rich set of statements, expressions and keywords to support code that will only be executed under certain defined conditions. This is called *conditional code*, where a condition is described using a combination of expressions, comparisons and logic operators, which are summarised in Table 29 below.

`test <expr>` `when <condition>` `else when <condition>` `else` `end test`	conditional statements with a single primary expression
`if <expr> is <condition>` `else if <expr> is <condition>` `else` `end if`	conditional statements with multiple primary expressions
`test <expr> is <condition>` `if <expr> is <condition> ?` `  <match expr> ?` `  <nomatch expr>`	conditional expressions
`test -s <expr>` `{` `  when <condition>` `  when <condition>` `  else` `}` `end test` `break`	multiway branching

Table 29 - Summary of conditional code constructs

The most basic condition is a simple comparison between two data items. This requires a *primary expression*, that is the first data item in the test; a *comparison operator* (is the data equal, or greater than etc.); and the *secondary expression*, that is the data item you wish to compare with.

It is always the responsibility of underlying data type APIs to determine how to perform comparisons. When the primary expression and secondary expression are not of the same data type, PROSE will not perform any automatic type conversion before comparison. Instead, a method is called on the primary expression to perform the comparison, and if this does not know how to handle the data type of the secondary expression then a method is called on the secondary expression to perform the comparison. PROSE may be requested to perform a type conversion by the underlying data type API if it cannot perform a comparison without such a cast (see: *3.7.1 Implicit type conversion* on page 18). To force a type conversion before a comparison, use an explicit cast.

Conditions are usually implemented whereby multiple primary, comparison and secondary expressions are evaluated and related to one another using *logic operators*, for example '`and`' to specify that both comparisons must be true, '`or`' to specify that either comparison (or both) must be true. While supporting this type of operation, and unlike other common programming languages, PROSE also provides the ability to work with multiple comparison and secondary expressions in relation to a single primary expression, and it is this construct that is discussed first.

6.7.1 Single primary expressions (test/when statements)

When testing the result of a single primary expression, one must identify the data that is to be tested. This is handled using the *test statement*, which takes the form `test <expr>`. The expression `<expr>` is evaluated to determine a result. The result is stored for the duration of the code block, and may be compared at any time with the result of any number of secondary expressions using a *when statement*, which takes the form `when <condition>`, where `<condition>` is a collection of comparison operators, secondary expressions and logic operators to evaluate for comparison. The logic operator may be omitted in the case of a comparison for equality. The regular statement (or code block) immediately following a `when` statement is executed only if the condition is satisfied, otherwise it is skipped over and program control continues at the next statement.

Once conditional code has been executed following successful logic in a `when` statement, program control follows on immediately to the next statement. If this is another `when` statement, it is possible that another condition is met. In order to prevent multiple `when` statements from executing, use the *else-when statement* in its place which takes the same form as a when statement, but is only evaluated if no previous when statement since the last test has been satisfied. As a catch-all, if the *else statement* is used it is executed if no previous `when` or `else-when` logic was matched. The `else` statement and its conditional code may be joined to make a compound statement.

The *end-test statement* may be used once a test result is no longer required. The result is then discarded. The use of `end-test` is optional but helps the readability of the code. PROSE compilers can also be requested to display warnings or errors if an `end-test` is not found at the right encapsulation level, making it easier to identify structural problems in the program.

A `when` statement may follow a `test` statement to form a compound statement (i.e. without a new line or semi-colon separating the two statements), if required for brevity.

The following example demonstrates the use of `test`, `when`, `else-when` and `else` statements:

```
test composer_name
when "Karl Jenkins"; print "The Armed Man"
else when not "Mozart" and not "Beethoven"; print "Unknown"
else {
  test library_id = Get_database_id()
  status = 0
  when = 0 { print "*** bad id ***"; status = -1 }
  when > 0 { print "Welcome back, " Person[library_id] }
  end test
}
end test
```

In the previous example, the first `test` keyword evaluates the expression `composer_name`, which in this case is simply a string variable. The result of this expression (the contents of the variable) is stored ready for comparison by a `when` statement.

The `when` and `else-when` statements that follow the test compare the stored result of the previous `test` with another expression. If the comparison is successful, the respective `print` command is executed (the next regular statement, in this case it is on the same line as the `when` statement, separated by a semi-colon).

The final `else` block is executed if none of the previous `when` or `else-when` statements matched. This is a new code block, so the new `test` statement within stores the result of its expression `library_id = Get_database_id()` without overwriting the result of the previous `test` statement. This result will be whatever value the function `Get_database_id()` returns, which is also stored in the `library_id` variable.

The statement `status = 0` after the second `test` sets a new variable. It is placed here to demonstrate that `when` statements do not need to immediately follow a `test` statement. Likewise, it is not necessary for `when` statements to be grouped together, although it usually makes your program code more legible if they are.

The final two `when` statements use code blocks instead of single statements if the result of `Get_database_id()` matches their respective conditions. Notice that the last `when` statement will be evaluated regardless of whether the first `when` statement met the condition or not. As the two conditions are mutually exclusive, this serves the same purpose as an `else-when`, but does require an additional comparison that would not be required if an `else-when` was used in its place.

It is legal for a `when` statement to follow a `test` statement on the same line, without a semi-colon separating them. This is an abbreviated form of the construct, demonstrated below:

```
test tuning when true; print "You're in tune!"
else print "Please tune your instrument now ..."
```

6.7.2 Single primary results (test-is expression)

It is sometimes necessary to store the result of a comparison in a variable for later reference. This requires a conditional expression rather than a statement, and is called the *test-is expression*. This takes the form `test <expr> is <condition>` where `<expr>` serves the same purpose as the expression on a `test` statement, and `<condition>` the same meaning as that on a `when` statement.

The result of a `test-is` expression is `false` if the comparison failed, or `true` if the comparison succeeded. Consider the following example (parentheses added for clarity, but are optional):

```
good_result = test getResult() is ( > 50 and not > 100 )
print good_result
```

A `test-is` expression sets the result for a future `when` statement, just as a `test` statement does. This allows you to store the result of a `test` and perform conditional code with secondary logic based on the same initial test result, in a succinct fashion:

```
good_result = test getResult() is ( > 50 and not > 100 )
print good_result

when > 95 and not > 100
        print "an outstanding achievement!"
when > 100
        print "you exceeded the top mark - you cheat!"
end test
```

6.7.3 Delayed evaluation (test -d)

The `test-when` and `test-is` constructs may be modified by a `-d` option. In this case the primary expression is not evaluated once, but everytime a `when` statement is encountered. Here is an example:

```
test -d c++
when 0; print "zero"
when 1; print "one"
when 2; print "two"
```

In the previous example, assuming `c` began as an uninitialised variable (or value `0`), then all three `when` expressions will evaluate to `true`, and all three `print` statements will be executed. This is because `test -d` has requested that the primary expression `c++` is evaluated each time, which tests the current value and then increments. The same effect would not have been achieved if `else-when` statements were used, or if the `-d` option was omitted.

6.7.4 Multiple primary expressions (if-is statements)

Often, testing a single primary expression against a variety of conditions is adequate. However, when the logic processing is complicated it becomes necessary to test multiple primary expressions.

This is achieved using the *if-is statement*, which takes the form `if <expr> is <condition>` where `<expr>` is a primary expression and `<condition>` a single comparison operator and secondary expression. If followed by a logic operator, such as '`and`', a further `<expr> is <condition>` may be specified, and so on in a recursive fashion. As with the `when` statement, the logic operator may be omitted in the case of comparison for equality.

When the condition is met, the regular statement immediately following the `if-is` is executed. Further `else-if-is` statements can be provided which will match other conditions, and a final `else` as a catch-all. Once conditional code is executed, the program continues at the statement immediately following the last `else-if-is` or `else`. This branching is very similar to the previous descriptions of `else-when` statements. Unlike the `test` statement, an `if-is` statement does not retain results that can be tested later. It is therefore invalid for an `else-if-is` or `else` statement to be orphaned from an `if-is` statement.

The *end-if* statement may be used after the last `if-is`, `else-if-is` or `else` to indicate that there are no further statements relating to this collection of tests. The use of `end-if` is optional but helps the readability of the code. PROSE compilers can also be requested to display warnings or errors if an `end-if` is not found at the right encapsulation level, making it easier to identify structural problems in the program.

The following example, although a bit contrived, demonstrates the use of multiple primary expressions:

```
if ( sky is grey
  and it is raining
  and fred is complaining )
{
  welcome(fred)
  print ( "Come inside, " ) fred ( " and get yourself warm" )
}

else if sky is blue
  print "Look!  A sunny day!"

else if rating = get_rating() is > PG
{
  thunder(loud)
  lightning(bright)
  print "I'll scare you all with wondrous night!"
}

else print "BOO!"

end if
```

6.7.5 Conditional expressions (if-is expressions)

The *if-is expression* can be used in place of any normal expression. It evaluates to one expression or another based upon a condition of the form `if <expr> is <condition> ? <match expr> ? <nomatch expr>` where `<expr>` and `<condition>` are the same as in an `if-is` statement, `<match expr>` is the expression to return if the condition is met, otherwise `<nomatch expr>` is returned. The delimiter character is a question `?` mark that separates the test and the conditional expressions. Either `<match expr>` or `<nomatch expr>` may be omitted. If `<nomatch expr>` is omitted, the second question mark may also be omitted.

The result of the entire expression is the result of whichever conditional expression was evaluated. If no conditional expression is evaluated, the result is the same as referencing a variable that does not exist - i.e. a default value is returned (see: *3.6 Variable declaration and initialisers* on page 15).

Here are some examples:

```
my_result = ( if score is < 50 ? "poor" ? "average" )
print ( "Your result is " ) my_result
print "Try harder next time"
print ( "You have " ) exam_count ( " exam" ) \
         ( if exam_count is 1 ? ? "s" ) \
         " remaining"
```

6.7.6 Multiway branching (test -s statement)

It is possible in PROSE to use `test` and `when` statements to create the effect of a C-type switch/case construct. This is a type of construct that allows you to execute conditional code based upon a control expression. This is demonstrated in the following (incomplete) example:

```
test -s month
{
   when "Jan" or "January";  print "01"
   when "Feb" or "February"; print "02"
   when "Mar" or "March";    print "03"
   when "Apr" or "April";    print "04"
   when "May";               print "05"

            # ... add in all the other months too

  else print "??"
}
end test
```

The first difference you'll notice in the previous example is the use of the `-s` option with the `test` keyword. This modifies a `test` statement so that it behaves like a *switch*. In this context, it must be followed by a code block (the *test body*), which may only contain one or more `when` statements and one final `else` statement (optional). These behave as previously defined, with one exception: if a condition on a `when` statement is met and the resulting conditional code executed, program control resumes at the next statement immediately after the `test` body is closed. No further `when` statements are evaluated.

The advantage of the `test -s` scenario is two-fold. For the programmer, and unlike C, a multiway branch construct is not cluttered with many `break` keywords. For the compiler, the code may be optimised further because it is not possible for a control expression to match more than one block of conditional code.

The `-s` option on the `test` keyword does not prevent the normal behaviour of `test`, i.e. the result of the expression is still retained and can be accessed with additional `when` statements if required. So if it necessary to branch your code further once a condition has been met, this can still be achieved without losing the benefit of code optimisation, as follows:

```
test -s month
{
   when "Jan" or "January";        print "01"
   when "Feb" or "February";       print "02"

   when in ( "Mar", "March", "Apr", "April" )
   {
       when "Mar" or "March";      print "03"
       else when "Apr" or "April"; print "04"

       print " (rainy days)"
   }
}
end test
```

However, this is better written as:

```
test -s month
{
   when "Jan" or "January";           print "01"
   when "Feb" or "February";          print "02"
   when in ( "Mar", "March", "Apr", "April" )
   {
       test local.list  when 1 or 2;  print "03"
                        when 3 or 4;  print "04"

       print " (rainy days)"
   }
}
end test
```

The previous example uses the `in` operator and the `local.list` variable (see: *5.4 Comparison operators* on page 47).

6.7.7 Multiway branching and the break keyword

The `break` keyword can be used for moving program execution past the end of the current `test` body. An integer following the keyword may be used to specify the number of `test` bodies to break out of if there are several levels of encapsulation.

It is therefore possible to produce the effect of multiway branching without the `-s` option on the `test` keyword, as demonstrated below:

```
test month
{
   when "Jan" or "January"   { print "01"; break }
   when "Feb" or "February"  { print "02"; break }
   # etc.
}
end test
```

This however prevents the compiler from creating fully optimised code and should be avoided.

6.7.8 Logic switching

Because defining a single test expression with the `test` statement is separated from the comparison logic in the `when` statement, it is possible in PROSE to modify the initial `test` expression while still evaluating a list of comparisons. This is known as *logic switching*, and is demonstrated in the following example:

```
test application_request
when "desktop";        test get_desktop_acl()
else when "office";    test get_office_acl()
else when "network";   test get_network_acl()
else when "firewall";  test get_firewall_acl()
else                   test ""

when "granted";        print application_request " granted"
when "special";        test get_special_permissions()
when "admin";          print application_request " administrator"
when "super-user";     print application_request " super-user"
else                   print application_request " DENIED"
end test
```

The previous example illustrates how another `test` statement appearing as the conditional code for a `when` statement will affect future `when` statements. This only works if the `test` statement does not appear in its own code block, because `test` results are only in scope at the encapsulation level in which the `test` is performed, and therefore can be the only statement executed in the conditional code. Take particular note of the statement `test get_special_permissions()` which may very well return `"admin"` or `"super-user"`, just as `get_network_acl()` or `get_firewall_acl()` could return the same, and achieve identical results.

6.8 Iterative code

Program code that repeats its execution while cycling through a number of data items, or keeps looping until a particular condition is met, is known as *iterative code*. PROSE provides an extensive set of iterative code constructs, summarised in Table 30 below.

`for <expr>` `  to <expr>` `  until <expr> is <condition>` `  while <expr> is <condition>` `  step <expr>` `end for`	loop constructs with defined start-points and end-points
`for <expr>` `  in <expr>` `end for`	loop constructs for iterating objects in a collection
`loop` `  until <expr> is <condition>` `  while <expr> is <condition>` `repeat` `end loop`	loop constructs with no defined start-point, and an end-point that might be at the head or tail of the loop
`break` `continue`	keywords used to amend control flow within a loop

Table 30 - Summary of iterative code constructs

The statements listed in Table 30 may be separated onto new lines as illustrated without the use of a continuation character, and may also appear on a single line without semi-colons.

All `for` and `loop` constructs are followed by a code block or regular statement, known as the *loop body*, followed by an *end-for statement* or *end-loop statement* respectively. The use of `end-for` or `end-loop` is optional, but recommended as it adds clarity to the code and allows for additional error checking by the compiler.

The loop body is the code that is executed for each iteration of the loop. If there is no loop body, use a semi-colon instead of the regular statement. Alternatively, the loop body may be completely omitted, in which case the `end-for` or `end-loop` statement is mandatory.

6.8.1 Iteration with defined start-points and end-points

This section discusses looping constructs that have a well-defined start and end point. The *start-point* is one or a number of variable assignments or a `typeset` declaration list with initialisers - that are evaluated once - before the first loop iteration (any variable data type is legal). The *end-point* defines how the loop will terminate, or how many iterations of the loop will occur. Loop constructs may also require a *step expression*, which describes what steps are to be performed between each loop iteration.

6.8.1.1 The for-to statement

The `for-to` statement is used for looping constructs that require a single add, subtract, multiply or divide operation between each iteration. The statement takes the form `for <assign> to <value> step <op><value>` where `<assign>` is one or more comma-separated variable assignments or a `typeset` declaration list with initialisers (these variables are called the *control variables*), `<value>` is one or more comma-separated end-points defined by value (the *end values*) where a successful definite equality comparison with the corresponding control variable flags the last iteration of the loop, and the optional `step <op><value>` defines the mathematical operation to perform on each control variable between each iteration of the loop (the *step value*).

There may be multiple comma-separated step values supplied. The mathematical operation specified by `<op>` may be empty or `+` for add, `-` to subtract, `*` to multiply and `/` to divide.

The simplest use of the `for-to` statement is where `<assign>` has one variable assignment, `<value>` is an expression that evaluates to a single value, and `step <value>` is omitted. Under these conditions, the control variable will have integer `1` added to it at the end of each iteration (the default step operation when the `step` keyword has been omitted), and the loop will continue until the contents of the control variable matches the end value, when the last loop iteration occurs, and then program control carries on past the loop body. This is demonstrated in the following example:

```
for i = 1 to 100
    print i
end for
```

The previous example will display each integer between `1` and `100` inclusive. At first glance it appears that you can add another statement between the `for` and `end-for` without using a code block, but this would result in a compile-time error. The only correct way of adding multiple statements to a loop body is as demonstrated below:

```
for i = 1 to 100
{
    print i
    print i + .5
}
end for
```

The last example will display all decimal numbers between `1` and `100.5` at `.5` intervals. This could also have been achieved using the `step` keyword, as follows:

```
for i = 1 to 100.5 step .5
    print i
end for
```

This also demonstrates how an undeclared variable might have its type changed implicitly during a loop. Here, `i` is initialised with an integer constant `1`, but is cast to a floating-point type after the first iteration when the add `.5` operation occurs. It will then remain a floating-point type for the duration of the construct (see: *3.7.1 Implicit type conversion* on page 18). To force `i` to remain a particular type, it must have either been previously declared using the `typeset` keyword, or the start-point expression should contain a declaration and initialiser as illustrated below (parentheses are optional):

```
for (typeset int i = 1) to 100.5 step .5
    print i
end for
```

Here, `i` is forced to remain an integer. This means that the step value `.5` will have to be cast to an integer each loop iteration, which results in a rounding-up to the value of `1`. The above example will therefore display all integer values from `1` to `101`. Because the `typeset` declaration is only evaluated once before loop iteration begins, it is possible to change the type of variable `i` at any time during the loop, for example performing a `typeset float i` when `i` reaches `50` will result in a display of integers between `1` and `50`, then followed in `.5` steps thereafter to `100.5`. Changing the type of the control variable mid-loop does make your code harder to read and less intuitive, but is permitted. Compilers must therefore cast the step value each loop iteration (esp. if the cast results in a loss of precision).

The control variable might be any data type, because every data type must support add, subtract and comparison operators. Here is another example:

```
start = [1/3]; end = [8/3]
for rt = start to end step start
{
    print "Next number in sequence:"
    print "\t" rt
}
end for
```

The previous example demonstrates the use of variable names in place of constants, and will print the rational number sequence between `[1/3]` and `[8/3]` inclusive. The last loop iteration occurs when a definite equality comparison between the control variable `rt` and the end value contained in the variable `end` evaluates to `true` (see: *5.4 Comparison operators* on page 47).

The step value is used to form an add operation unless it begins with a `-`, `*` or `/` operator, in which case it will form a subtract, multiply or divide operation. The following `for` statements are identical:

```
for num = 3 to 0 step -1      # subtract operation on integer type
end for

incr = -1
for num = 3 to 0 step incr    # add operation (adding a negative)
end for

incr = 1
for num = 3 to 0 step -incr   # subtract operation
end for
```

The `for` statements in the previous example would produce identical results because adding a negative number to an integer type produces the same result as subtracting the absolute (positive) equivalent, i.e. `3 - 1 = 3 + -1`.

This is not the case with string types, where an add operation appends a string and a subtract operation removes it. This is demonstrated in the following examples:

```
for str = "1" to "111" step "1"
    print str       # displays "1", then "11", then "111"
end for

for str = "111" to "1" step "-1"
    print str       # displays "111", then "111-1", then "111-1-1"
                    # and the sequence continues indefinitely
end for

for str = "111" to "1" step -"1"
    print str       # displays "111", then "11", then "1"
end for
```

The loop will only complete when the end value is matched exactly. Therefore, be particularly careful when using the divide operator and testing for floating-point numbers. Also be sure that the start-point is never beyond the end-point in the sequence, otherwise the sequence will not complete and will loop forever.

It is also legal with the `for-to` statement to provide multiple start-points and multiple end-points. This is achieved by supplying a list of comma-separated assignments, or a single `typeset` with multiple declarators and initialisers. The following formats illustrate this behaviour (parentheses are optional):

```
for (var1 = val1, var2 = val2)
        to end_val1
        step step_val1

# equivalent: for (var1 = var2 = val1)
# equivalent: for (typeset <type> var1 = val1, var2 = val2)

for (var1 = val1, var2 = val2)
        to (end_val1, end_val2)
        step step_val1

for (var1 = val2, var2 = val2)
        to (end_val1, end_val2)
        step (step_val1, step_val2)
```

Where multiple assignments have been provided in a `for-to` statement, as illustrated above, then there are multiple control variables that will each have a start-point, an end-point, and a step value. This is short-hand for writing loops within loops.

The first control variable var_1 defines the behaviour of the outermost loop; its start-point val_1, end-point end_val_1, and step value $step\_val_1$. The second control variable var_2 defines the behaviour

of the next inner loop; its start-point will be `val2` unless assigned using the syntax `var1 = var2 = val1` in which case its start-point is `val1`. The second control variable has an end-point of `end_val2` unless this is not defined, in which case the end-point will be the last one defined, i.e. `end_val1`. The step value for the second control variable is `step_val2` unless this is not defined, in which case the step value will be the last one defined, i.e. `step_val1`.

The following `for-to` statements are therefore identical, and print all cells in a two-dimensional matrix called `my_matrix[]` that has 6 columns and 6 rows:

```
for (i = 0, j = 0) to 5
    print my_matrix[i, j]
end for

for i = 0 to 5
    for j = 0 to 5
        print my_matrix[i, j]
    end for
end for
```

To display the contents of a matrix that has ten available indices in the second dimension, and six in the first, the following two example loops are identical in behaviour:

```
for (i = 0, j = 0) to (5, 9)
    print my_matrix[i, j]
end for

for i = 0 to 5
    for j = 0 to 9
        print my_matrix[i, j]
    end for
end for
```

To display the contents of only every other cell in the second dimension, here again are two identical examples:

```
for (i = 0, j = 0) to (5, 9) step (1, 2)
    print my_matrix[i, j]
end for

for i = 0 to 5
    for j = 0 to 9 step 2
        print my_matrix[i, j]
    end for
end for
```

The following example finds all empty columns in a table and deletes them, moving the next columns over the top, effectively squashing the table:

```
function squashTable(matrix table[])
{
    push table[].size() - 1
    for i = 0 to peek - 1
    {
        test table[i,-1].empty()
        when true
        {
           for j = i + 1 to peek
              move table[j,-1] to table[i,-1]
           end for
        }
        end test
    }
    end for
}
end function
```

6.8.1.2 The for-until statement

The `for-until` statement allows for looping constructs that continue until a conditional expression returns `true`. It takes the form `for <assign> until <expr> is <condition> step <step_expr>` where `<assign>` is the same as the `for-to` statement, `<expr> is <condition>` is the same as the `if-is` statement (see: *6.7.4 Multiple primary expressions (if-is statements)* on page 66) and `step <step_expr>` is optional. If `step <step_expr>` is supplied, then `<step_expr>` is a comma-separated list of operations that may be used for adjusting the values of the control variables.

Unlike with the `for-to` statement, the `for-until` statement does not have short-hand for creating loops within loops. Multiple control variables in a `for-until` statement are simply treated as a list of initialisers at the head of a single loop.

Here is an example:

```
maxnum = 10
for i = 0 until i is maxnum step i++
    print i         # displays numbers 0 through to 9
end for
```

Here is another example, using multiple control variables. Parentheses are optional but added for clarity:

```
for (i = 0, j = 10)
   until (i is > i_max or j is > j_max)
   step (i++, j++2)
{
     test table[i, j] when true
         print "found at " i, j
     end test
}
end for
```

6.8.1.3 The for-while statement

The `for-while` statement is identical in use to the `for-until` statement, except that the loop continues for as long as the condition is `true`.

Here is a simple example:

```
maxnum = 10
for i = 0 while i is < maxnum step i++
    print i         # displays numbers 0 through to 9
end for
```

Here is an example shell sort algorithm written using the `for-while` statement:

```
function shellSort(matrix array[])
{
     push array[].size()
     for (gap = peek / 2) while (gap is > 0) step (gap /= 2)
          for (i = gap) while (i is < peek) step (i++)
          {
             push array[i]
             for (j = i - gap)
                while (j is >= 0 and array[j] is > peek)
                step (j -= gap)
             {
                    move array[j] to array[j + gap]
             }
             end for
             array[j + gap] = pull
          }
          end for
     end for
}
end function
```

6.8.2 Iteration through a collection (for-in)

The `for-in` statement provides a convenient construct for processing multiple data items within a collection, by means of iterating the collection indices. With stacks and matrix arrays, these are the integer co-ordinates of cells that contain data (cells that are uninitialised are skipped). When used with a tree structure, it is possible to iterate keys in single or multiple branches.

The statement takes the form `for <varlist> in <collection>` where `<varlist>` is a comma-separated list of control variables and `<collection>` is the name of the collection to iterate with an index provided between square `[]` brackets.

6.8.2.1 Iterating a stack or matrix

When iterating a collection that uses integer co-ordinates in the index, the control variables will be assigned numbers that can be used in the index. The first control variable iterates the first dimension, the second control variable the second dimension, and so on. Uninitialised cells in the matrix are skipped. If a cell has been assigned a type but not a data item, it will be included.

The index used with the collection name on the `for-in` statement selects the entire collection or a portion of the collection. If the index is empty, the entire collection is iterated. If a wildcard index is used, it selects the columns or rows to be iterated. Where a specific cell is addressed, the control variable associated with this dimension will always contain this value for every iteration of the loop.

Here are some examples:

```
for i in array[]           # iterate first dimension of array
    print array[i]         # display contents of first dimension
end for

for i in array[]           # iterate first dimension
    print array[i,0]       # display first cell of second dimension
end for

for (i, j) in array[]      # iterate two dimensions
    print array[i,j]       # display all cells, one at a time
end for

for (i, j) in array[-1,2] # iterate every column of third row
    print array[i,j]       # j will always be 2
end for
```

6.8.2.2 Iterating a tree structure

When iterating a tree structure the control variables will be assigned values that can be used as keys for looking up items in the collection. The first control variable iterates nodes in the first indexed branch, and second control variable iterates nodes in the second branch etc.

The index used with the collection name on the `for-in` statement identifies which branches to iterate, and which should remain static. If the index is empty, only the nodes of the topmost branch are iterated. If wildcard indices are used, the `"*"` indicates a branch that is to be iterated such that the control variable for the index contains each key in turn, and `"**"` indicates all sub-branches are to be iterated such that the control variable for the index contains keys listed in the same format as displayed by the `listSub()` method (but in this case cannot be used in a subsequent index lookup). Where a specific key is addressed, the corresponding control variable will always contain that value.

Here are some examples:

```
for key in my_tree[]            # keys in top-level branch
    print key, my_tree[key]     # display key and value
end for

for (key1, key2) in my_tree[]   # keys in top two levels
end for

for (key1, key2, key3)
    in my_tree["composer", "*", "**"]
{
             # key1 will always contain "composer"
             # key2 the keys at the second level
             # key3 all remaining keys from third level and below
             # (although key3 can't then be used in an index)
    print key1, key2 " ="
    print "\t" my_tree[key1, key2]
    print key3
}
end for
```

The wildcard index `"**"` has limited use, as the control variable associated with this index cannot be used for determining the value contained within the addressed node, as the format is not appropriate for an address index. Therefore, to iterate nodes at an arbitrary depth within a tree structure, it is necessary to use pointers and recursive functions, as illustrated in the following example:

```
function displayBranch(pointer root[] to tree)
{
   for (parent in root[])
   {
      print key " = " root[parent]
      test root[parent].children() when > 0
         displayBranch(root[parent])
      end test
   }
   end for
}
end function
```

6.8.3 Iteration with floating end-points

This section discusses the `loop` and `end-loop` statements, as well as the `repeat`, `while-is` and `until-is` statements used in the context of a `loop`. These are looping constructs that check for loop termination at the head or tail-end of the construct, and do not contain control variables or step expressions.

6.8.3.1 The loop construct

The most basic loop is one that continues forever without any termination clause. This is described using a `loop` keyword, followed by the loop body, followed by the optional `end-loop` statement. Here is a simple example, that displays a message repeatedly until forceably terminated:

```
loop
{
    print "loops forever"
    print "...and forever"
}
end loop
```

6.8.3.2 The loop-until construct

This loops until a condition is met. The termination clause takes the form `until <expr> is <condition>` where `<expr> is <condition>` is the same as the `if-is` statement (see: *6.7.4 Multiple primary expressions (if-is statements)* on page 66).

The termination clause may immediately follow the `loop` keyword at the head of the loop body, or if used at the end of the loop body it must follow the `repeat` keyword.

The following example displays a sequence of asterisks beginning with one and ending with ten:

```
loop until s.size() is 10
{
    s += "*"
    print s
}
end loop
```

The following example calculates a factorial, the loop body is only one regular statement so it has been placed on the same line as the `loop` keyword, separated by a semi-colon:

```
function factorial(int n)
{
    typeset int result = i = 1

    loop; result *= ++i
    repeat until i is n
    end loop

    return result
}
```

6.8.3.3 The loop-while construct

This loops while a condition is true. It takes the same form as the `loop-until` construct. Here are the `loop-until` examples re-written using `loop-while`:

```
loop while s.size() is < 10
{
    s += "*"
    print s
}
end loop
```

And the factorial example:

```
function factorial(int n)
{
    typeset int result = i = 1

    loop; result *= ++i
    repeat while i is < n
    end loop

    return result
}
```

6.8.4 Control flow

6.8.4.1 Iteration and the break keyword

The `break` keyword may be used inside a `for` or `loop` construct to break out of that loop. If inside nested loops, an integer may be supplied after the `break` keyword to specify how many nested loops to break out of. Program execution continues following the end of the loop construct broken out of.

The following example displays the contents of a single-dimensioned matrix array but terminates the loop prematurely if an empty cell is found:

```
for i = 0 to array[].size() - 1
{
    test array[i].empty()
    when true; break
    end test

    print ("array[") i ("] = ") array[i]
}
end for
```

6.8.4.2 The continue keyword

The `continue` keyword may be used inside a `for` or `loop` construct to skip the rest of the loop body and continue with the next loop maintenance task, such as checking for loop termination or incrementing a control variable.

The following example is a modification of the break example, where the contents of a single-dimensioned matrix array are displayed, but empty cells are skipped:

```
for i = 0 to array[].size() - 1
{
    test array[i].empty()
    when true; continue
    end test

    print ("array[") i ("] = ") array[i]
}
end for
```

7 Appendix

7.1 References

[R1] Harbison III, Samuel P. and Steele Jr., Guy L. *C: A Reference Manual, Fifth Edition*, Prentice Hall, Upper Saddle River, NJ 07458, USA, 2002, ISBN 0-13-089592-X.

[R2] Loudon, Kyle. *Mastering Algorithms with C*, O'Reilly & Associates Inc., Sebastopool, CA 95472, USA, 1999, ISBN 1-56592-453-3.

[R3] Levine, John R.; Mason, Tony and Brown, Doug. *lex & yacc*, O'Reilly & Associates Inc, Farnham, Surrey, GU9 7HS, England, 1995, ISBN 1-56592-000-7.

[R4] Howes, Timothy A.; Smith, Mark C. and Good, Gordon S. *Understanding and Deploying LDAP Directory Services*, Macmillan Technical Publishing, USA, 1999, ISBN 1-57870-070-1.

[R5] Gosling, James; Joy, Bill and Steele, Guy. *Java Language Specification, Second Edition*, Sun Microsystems Inc, USA, 1996.

[R6] Unicode Consortium, The. `http://www.unicode.org`

[R7] Free Software Foundation Inc. *GNU multiple precision arithmetic library*, *version 4.1.2*. `http://www.swox.com/gmp`