# THISTLE
# LANGUAGE REFERENCE MANUAL

Stephen R. Donaldson
stephen@codemagus.com

Code Magus Limited
23 Warnborough Road
Oxford, OX2 6JA, UK
www.codemagus.com

Copyright © 2002–2003, Code Magus Limited. All rights reserved.

# Contents


**1 Introduction** **1**

1.1 Features of Thistle . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 Operating Environment . . . . . . . . . . . . . . . . . . . . . . . . . . 2

**2 Elements of Thistle** **5**

2.1 comments . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

2.2 Reserved Words . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.3 Special Symbols . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.4 Identifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.5 Literals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

2.6 Tree Name Space . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9

2.7 Structure of Thistle Artefacts . . . . . . . . . . . . . . . . . . . . . . 17

2.8 Expressions and Operators . . . . . . . . . . . . . . . . . . . . . . . 19

**3 Executable Statements** **29**

3.1 Compond Statements . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.2 Assigment Statement . . . . . . . . . . . . . . . . . . . . . . . . . . 34

3.3 Transfer of Control: Method Invocation . . . . . . . . . . . . . . . . 34

3.4 Transfer of Control: The `return` Statement . . . . . . . . . . . . . 34

3.5 Conditional Execution . . . . . . . . . . . . . . . . . . . . . . . . . 35

3.6 Iteraction: The `for` Statement . . . . . . . . . . . . . . . . . . . . 35

3.7 Loops: The `while` Statement . . . . . . . . . . . . . . . . . . . . . 35

3.8 Loops: The `repeat` Statement . . . . . . . . . . . . . . . . . . . . 35

3.9 Interrupting Execution: The `check` Statement . . . . . . . . . . . . 36

3.10 Interrupting Execution: The `break` Statement . . . . . . . . . . . . 36

3.11 Choosing Name Space Scopes: The `with` Statement . . . . . . . . . 36

**4 Thistle Usecases** **37**

**5 Thistle Instances** **39**

**6 Thistle Interfaces** **41**

**7 Thistle System Objects** **43**

**Bibliography** **44**

# List of Figures

1.1 Thistle components and relationship to surrounding systems . . . . . 3

2.1 Thistle top level name space . . . . . . . . . . . . . . . . . . . . . . . 10
2.2 Resultant evaluated path and node values . . . . . . . . . . . . . . . . 12
2.3 . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

# List of Tables

2.1 Escape characters and their corresponding encoded characters . . . . 8
2.2 Thistle Operators: Precedence, Associativity, and Aridity . . . . . . . 20

# Chapter 1

# Introduction

Thistle is a programming language which has been designed for explicitly for scripting and it has a number of constructs which make it particularly suitable for scripting in the testing domain.

The language Thistle is based on the programming language Pascal[2, 1]. The inventors of Thistle have made a number of simplifications and generalisations on Pascal which have resulted in a language very suitable for scripting. Pascal was chosen as a basis for Thistle as it already had a number of attributes which makes it a good candidate for a scripting language.

## 1.1 Features of Thistle

The Thistle system comprises a compiler and run-time environment or interpreter. The technology is designed to make sure that implementations can exists on Unix platforms, on MicroSoft platforms and on OS/390 or z/OS. It should be clear from this document how character set encodings and endianness of the hosting system do not affect the Thistle scripts.

We assert that Thistle is a good scripting language for the following reasons:

Like Pascal, Thistle is a relatively small language, consists of few constructs and is easy to understand. We have further simplified the language by removing strong typing. Instead of the type-rich system of Pascal, Thistle only has a single elementary type which is the string (Chapter **??**).

The weak typeing of Thistle and hence the lack of compile-time type-checking is not a problem for Thistle as Thistle artefacts are always compiled on demand.

Thistle has no aggregation data types such as Pascal's arrays or records. The only aggregate data structure in Thistle is a tree-like data structure and there exists only one such tree. This is a generalisation of what is implicit in Pascal and many other programming languages. In Thistle we have made this explicit to the point where there is only one aggregated data structure which is a dynamic tree and to which all items belong, including elementary items. The root of this tree is called `thistle`.

Aggregate data structures such as records and arrays are also represented in this tree and hence hence there is no need for explicitly defining records or arrays.

This tree view is implicitly present in most programming languages and can be observed by considering the collection of activation records at run-time together with the scopes of objects, structures/records, classes, fucntions, procedures, methods and elementary items present at runtime.

The `thistle` tree also contains predefined system artefacts. The tree is initialised with these artefacts when the run-time system starts.

In Thistle, the `thistle` tree is initialised by the this run-time.

There has also been a conscious effort to make sure that the mapping to the hosting operating system's artefacts are handled by the run-time system and the naming conventions of the operating system's objects (such as files and directories) are not evident in the syntax of Thistle. Of course, Thistle scripts may still represent the names of such objects using the conventions of the hosting system for their own purposes. We do this so that Thistle artefacts can be moved from one user's system to another (of the same or different architecture—where that makes sense). Where this happens either because the scripts are moved or shared between users or projects with different local preferences, we want to make sure that the content and semantics of the scripts are not voided.

## 1.2 Operating Environment

Thistle is further isolated from the concrete representations and interfaces of the hosting system by the distance it maintains from the hosting system by interacting with it through a system of portals. Portals are also the means by which the scripts interact with the *System Under Test* or SUT when Thistle is used in a testing environment.

Figure 1.1 shows the relationship of the components of Thistle and its run-time system, its hosting system and a system under test. The Thistle system comprises a compiler and a run-time system. The Thistlecompiler has access to Thistle artefacts through a locally hosted artefact repository. There is a standard naming convention of artefacts as they are referred to within other Thistle artefacts. The mapping of this name space to a local name space of the hosting operating system is the responsibility of the Thistle system. This mapping is configurable and is the mechanism which makes sure that there is no binding of the name space within Thistle artefacts and the name space of the artefacts of the host system.

The run-time component prepares for execution of Thistle artefacts by building the `thistle` tree with pre-defined types (see Chapter 7). The initial artefact type that Thistle gives control to is the `package`. In the testing domain a `package` corresponds to a test pack.

A `package` can give control to `usecase` artefacts which are syntactically similar to `packages`. A `usecase` in turn can give control to other `usecases` and so on. Both aertfact types maintain state for local variables, parameters, etc, is maintained on the `thistle` tree.

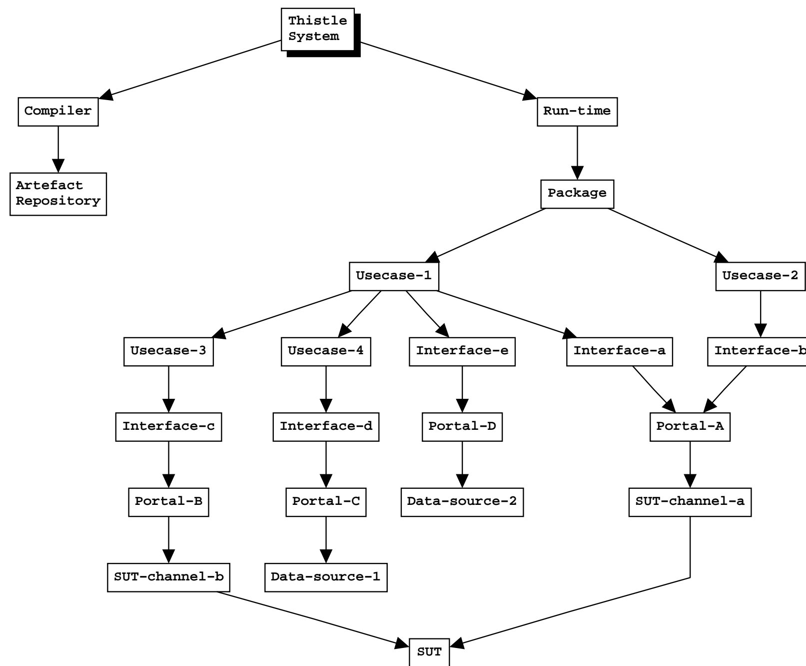



Figure 1.1: Thistle components and relationship to surrounding systems

The executing Thistle artefacts never interface directly with the host system, nor with any SUT. This is only ever achieved through an `interface`, with the actual manipulation of either the system component or the SUT being performed by a `portal`. Interfaces are either in-built or user-defined. For user-defined interaces, the interface is described to the Thistle system using an `interface` artefact. Regardless of whether the interface is built-in or user-defined, the interface is introduced into Thistle `packages` and `usecases` in the same manner, and for all practical purposes there is no disticntion between the types of `interface`.

The responsibilities of a portal (which is not a Thistle language defined artefact) are to manipulate the SUT, operating system, component, data source, etc. and to map the name-space of the component to the name-space of Thistle. In this manner, the Thistle artefacts of type `package` and `usecase` can directly manipulate the object. The featurs of the language and the portals together make it possible to manipulate objects (other applications, data sources, operating system objects, etc.) in a seamless, coherent manner. This does not mean that an understanding of the objects being manipulated is not required, but that the language framework and syntax does not have to be extended or re-learned because of an expanding collection of data sources, applications, and operating system components that can be interfaced to.

# Chapter 2

# Elements of Thistle

A program or script in Thistle is structured in a very similar way to Pascal programs and even though there is essentially one elementary un-structured type, the litrals of this type can have a form which implies an refined sub-type. Put another way, whilst every elementary item in Thistle is a string, literals which look like integers, characters, dates are simply representations of strings. There are also hexadecimal literals and compressed hexadecimal literals.

Consequently a data item which defines an elementary value is simply a string and the assignment:

```
Details.Account := 1048010481;
```

is indistinguishable from the following assignment:

```
Details.Account := '1048010481';
```

and in this case establishes the existance of the identifier if it does not already exist.

From the above fragments and the discussion on Thistle's geneology, the elements of Thistle are very similar to what one would expect from a contempory programming language and comprise reserved words, special symbols, identifiers, literals, comments, expressions and operators. Thistle scripts are free format and white spaces have no grammatical meaning except where they might appear within string literals.

## 2.1 comments

Comments in Thistle are completely have no effect on the meaning and are completely ignored by the interpretation of the script. Comments are introduced using the left brace ({) and continue up to and including the next right brace (}). Comments can span lines and can contain any characters except the right brace (}) which would end the comment. Consequently, comments in Thistle cannot be nested.

## 2.2 Reserved Words

Reserved words are sequences that have a special meaning in terms of directing the parsing of Thistle and the recognition of valid artefacts in terms of the Thistle syntax. The 'definition' of the words are dealt with in the discussion of the respective syntactical construct. The Thistle reserved words are:

```
and       array     begin    case      div
do        downto    usecase  delete    else
end       external  file     for       forward
function  goto      created  if        in
label     mod       exit     not       of
or        by        packed   procedure package
run       repeat    set      then      to
isodate   type      until    var       while
with      check     break    when      modified
int       real      string   accept    desc
date      target    note     interface
```

There are a few words listed above which do not correspond to current syntactical constructs of the language. These either correspond to features removed from the language or are reserved for future extensions to the language.

## 2.3 Special Symbols

There are a number of sepcial symbols made up of either one character or pairs of characters. These, together with the reserved words, comprise the operators and delimiters of the language.

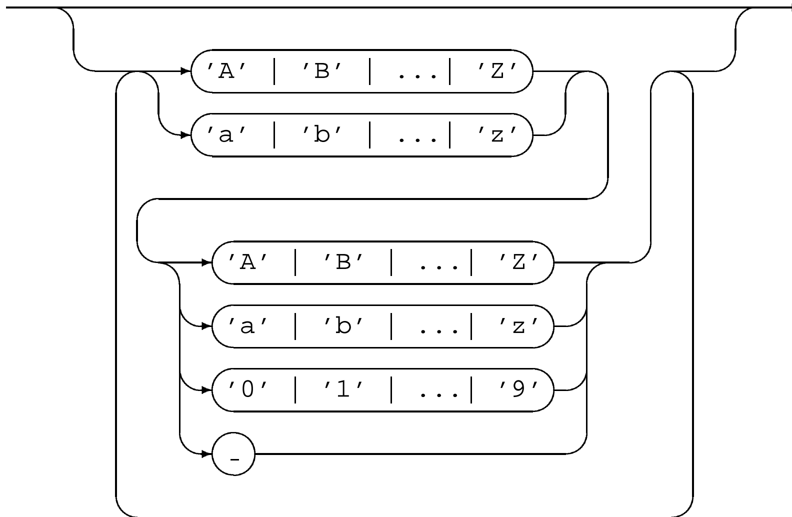
| | |
|---|---|
| `;` | See Section **??** |
| `:=` | See Section 3.2 and Section **??** |
| `<>` | See Section **??** |
| `>` | See Section **??** |
| `<` | See Section **??** |
| `<=` | See Section **??** |
| `>=` | See Section **??** |
| `=` | See Section **??** |
| `-` | See Section **??** |
| `:` | See Section **??** |
| { and } | See Section **??** |
| `<` and `>` | See Section **??** |
| `[` and `]` | See Section 2.6 |

## 2.4 Identifiers

Identifiers in Thistle are not quite the same as they are in Pascal. In Thistle identifiers are restricted names of nodes in the name space tree (see Section 2.6). Identifiers are

case sensitive, they start with a letter which can be followed by any number of letters or digits and the under-score character.

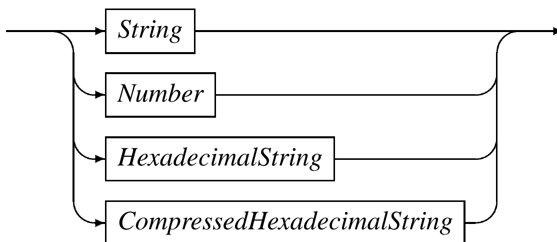
*Identifier*




## 2.5 Literals

Whilst all literals can thought of as strings, they can be expressed as numbers, string, hexadecimal strings, or compressed hexadecimal strings:

*Literal*



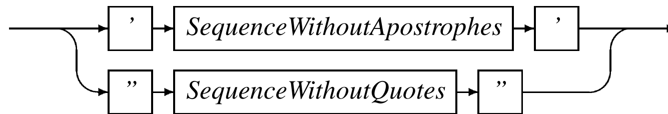

Thistleis an interpreted language and the resolution of values into concrete types does not need to occur until run-time. Consequently, as all input is ultimately user input, the type system exposed comprises a single type which is the string. The fact that a string may or may not be described by a further restricted domain such a a numeric string only needs to be checked and or considered at run-time. It is left up the use of a string to determine whether or not the value is appropriate (for example, an arithmetic operator requires a numeric value in its string operands). This does not mean that the implementation cannot store intermediate results in the most appropriate concrete format.

The generic string literal has the following format:

| Escape Character | Character encoded in string |
|---|---|
| a | alarm or bell |
| b | back space |
| f | forms feed |
| n | new line |
| r | carriage return |
| t | horizontal tab |
| v | vertical tab |
| ' | apostrophe |
| " | quotation marks |
| \ | backslash character itself |

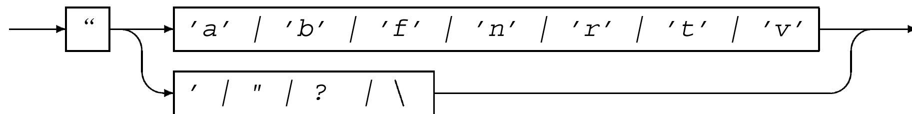
Table 2.1: Escape characters and their corresponding encoded characters

*String*




where *SequenceWithoutApostrophes* is a sequence of characters excluding apostrophes, and *SequenceWithoutQuotes* is a sequence of characters excluding quotation marks. Neither string can contain the newline character (i.e. strings cannot span source text lines), but they can contain escape characters, one of which represents the newline character. Also representable as escape characters are apostrophes and quotation marks. An escape sequence comprises the \ character followed by a character indicating the actual character to appear in the string:

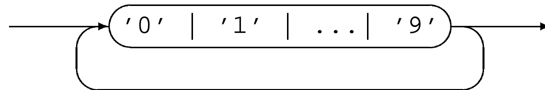
*EscapeSequence*




Where the esacpe characters and the encoded characters are explained in Table 2.1.

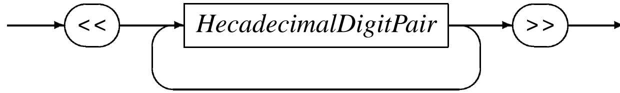
A number can be represented as a quoted string or as a sequence of digits without the quotation marks or apostrophes:

*Number*



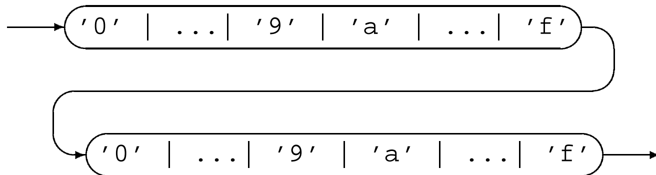

A *HexadecimalString* allows any sequence of character values to be encoded and has the following format:

*HexadecimalString*




Where

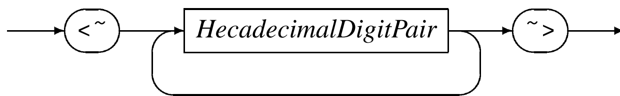
*HexadecimalDigitPair*




The uppercase equivalents of the hexadecimal digits digits `'a'` ... `'f'` may be used in place of their lower case counterparts.

A *CompressedHexadecimalString* is a compressed representation of its uncompressed value and has the following format:

*CompressedHexadecimalString*




Whilst strings cannot span Thistle source code lines, string expressions with the same intended value can. This is achieved using the # string concatenation operator.

## 2.6 Tree Name Space

All names exposed, created and used by Thistle artefacts are stored within a single name space. This name space is organised as a tree structure with the root of the tree named `thistle`. The construction of structured types such as maps, arrays, and aggregates is achieved by maintaining sub-trees within the `thistle` name space.

Elementary items in Thistle artefacts appear in the same tree as leaf nodes. For example, `Details.Account` above, above without further qualification refers to:

```
thistle.Packages[0].Details.Account
```

Figure 2.1 shows the top of the `thistle` tree and shows the hierarchial view of the position of the node `Account`. The figure shows the structure given that the current package is the first in the current run-time of the Thistle execution environment. This also hints at the way in which arrays are mapped to `thistle` name space tree and is a generalisation of the way in which Thistle handles maps.

A *Variable* is a path in the name space tree. It comrises either a single node or a sequence of nodes or identifiers. The first identifier or node represents a point in the tree from which a search for the node or identifier should (for references) or will (for

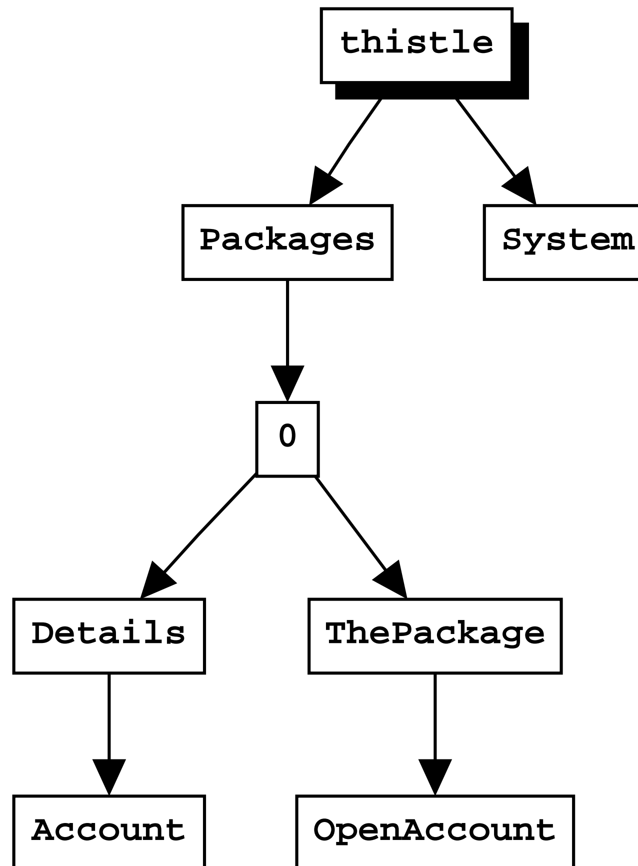
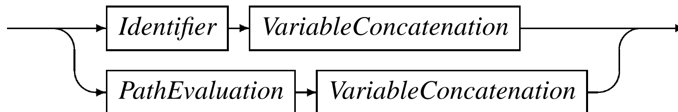



Figure 2.1: Thistle top level name space

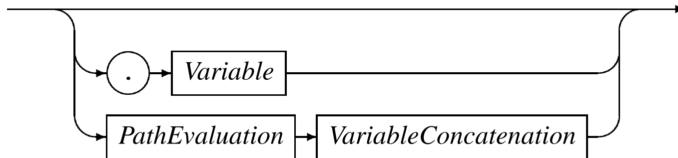
new nodes or definitions) be found. Any subsequent node will be found following the previous node and must be a direct descendant of that node, or will be a direct descendant of that node for a definition. A node need not be an *Identifer*, it could comprise an expression between `[` and `]` (referred to as a *PathEvaluation*) which will be evaluated at run-time and whose value is a further expansion of the path. The result of the evaluation can result in names of nodes which are not valid *Identifiers*. This is how arrays are defined in Thistle and which are implemented simply as the more generic map data structure. The content of the 'names' of such nodes are not restricted at all.

*Variable*



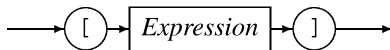

where

*VariableConcatenation*




and

*PathEvaluation*




The *Expression* may evaluate to string whose format is not a valid *Identifier* or sequence of *Identifers*. For example, the path:

```
TheUsecase[0].Details.Account['A B C'] := 1048010481;
```

refers to a variable which, after (and possibly before) execution of the assignment statement represents the path shown in Figure 2.2. The figure shows the names of the nodes and the relationship amongst the nodes. This example demonstrates how arrays (indexed by numeric values) and maps (indexed by strings) are treated in a generalised manner in Thistle which includes aggregates (`structs` in C or `records` in Pascal).

Because the height of the tree can grow significantly as a result of the combination of uses of aggregates, maps and arrays as well as Thistle `packages` and `usecases`, path names can become quite long especially if fully qualified from the `thistle` node. This can have the effect of making the script bodies quite dense and difficult to read and maintain. Thistle provides a mechanism that shortens names by supplying name search start points in the `thistle` tree. This mechanism has the effect maintaining nested open scopes in much the same way that Pascal opens scopes for local variables whilst keeping the previously open containing scope open. As with Pascal it is also possible to explicitly open a scope within a portion of the executable code using the `with` statement (see Section 3.11). For example, in the following code fragment, all the assignments of the literal `1048010481` are the same:

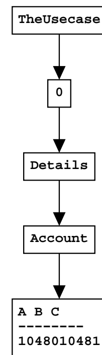


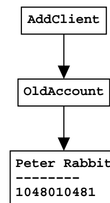

Figure 2.2: Resultant evaluated path and node values

```
TheUsecase[0].Details.Account := 1048010481;

AddClient.NewClient := 'TheUsecase[0].Details.Account';
[NewClient] := 1048010481;

AddClient.NewClient := 'TheUsecase[0]';
[NewClient]['Details.Account'] := 1048010481;
```

It should be clear from the above, that when used as a map data structure, the meaning of the period in a key string will add another layer to the name space tree. So for example, assume in the following that the nodes, except for the first, do not yet exist, then the assignment

```
AddClient.OldAccount['Peter Rabbit'] := 1048010481;
```

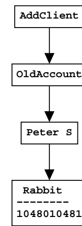
results in the creation of the following structure in the name space tree:




And the assignment

```
AddClient.OldAccount['Peter S. Rabbit'] := 1048010481;
```

results in the creation of the following structure:




A *Variable* is expected to exist wherever it is used except when the item is defined. In Thistle user *Variables* and hence the corresponding identifiers are created by the first occurance of the *Variable* in the artefact body and they do not have to be defined in the artefacts preamble. However, whenever a *Variable* is created, the point within the name space tree (i.e. the parent of the variable) has to be made explicit. In terms of the nodes of a *Variable* this means that the first node has to already exist and must be found by considering the current hierarchy of open scopes.

Scopes are opened implicitly by method or artefact invocation as well as explicitly by the script writer using the `width` statement (see Section 3.11). When a scope is opened because of an invocation (such as running a `package` or `usecase`), a node with the same name as the `package` or `usecase` is created in the name space tree. This node is where all local script defined *Variables* are located. For example, the following code fragment when executed just before the point where the `usecase` is about to return control to its caller results in the creation of sub-tree of the name space shown in Figure 2.3 (assuming that the contents of the cell `A1` in the sheet named in `SheetName` with the workbook whose name is in `BookName` is not the same as the value in the `AccountNumber` parameter and assuming the parameter `SheetName` contains `Sheet1`):

```
usecase CheckAccount(AccountNumber,BookName,SheetName);

   interface Portal.Excel : CodeMagus.excel;

   begin
     CheckAccount.checked := 'no';
     CheckAccount.myBook :=
           Portal.Excel.Connect(BookName);

     if CheckAccount.myBook.WorkSheets[SheetName].A[1]
            = AccountNumber then
        checked := 'yes';

     if checked = 'yes' then begin
        ReturnCode := 0;
        Reason := '';
        end
     else begin
        ReturnCode := 16;
        Reason := 'InvalidAct';
        end
   end.
```

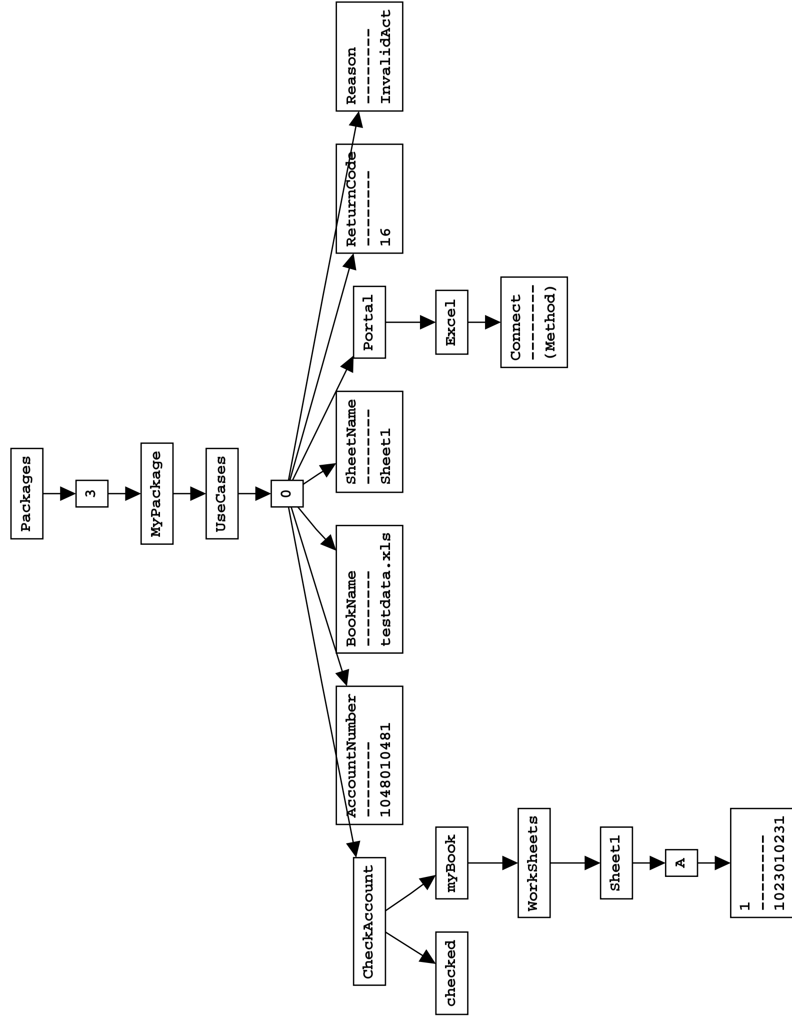



Figure 2.3:

We have described how interior nodes and leaves are defined, and that leaf nodes represent variables or methods. These are not the only types of nodes in the name space tree. In general a node in the tree can be one of the following types:

Leaf attribute These leaf nodes behave as the usual elementary variables of the language. As mentioned earlier, the type of the value of these nodes is always string as far as its semantics or observed behaviour is concerned. The implementation is free to represent the type in any suitable format, but when the value is extracted for external interpretation (for example to pass the value to a portal method), the string representation is expected to be regenerated (if not already stored as a string) and passed in this string representation.

In the following code fragment, the string assignment might keep the interal representation of the number as a string, but the assignment from the result of the expression in the second assignment could store the result as a number. Finally, the `if`-statement comparing the two values would perform the compare as though both items values were numeric and in this example, the boolean value evulates to true:

```
{ The result of the following assignment could result in a string
  representation of the number, for example as '01234'. }

sampleCase.numberA := 01234;

{ The result of the following assignment could result in the
  numeric 1234 being stored as the result of the expression. }

sampleCase.numberB := 1233+1;

{ Regardless of the internal representations of the numeric values, the
  boolean expression in the following if-statement evaluates to true. }

if (numberA = numberB) and (numberA = 1233+1) then
   System.Write(' The expression evaluated to true!');
```

Leaf method The values of these nodes are a callable method. This method can be invoked by placing the parenthesis behind the name of the method in a Thistle script. Optionally, any parameters to be passed to the script are listed in these parenthesis. In the following code fragment, the the first assignment invokes the method after evaluating the supplied parameter expressions and supplies the value returned by the method. In the second assignment, the value assigned is the method itself, and hence the third assignment has the same effect as the first assignment:

```
{ The first assignment evaluations the string expression argument and
   passes the resultant value through to the method for execution
   of the method by the Thistle run-time system. }

WorkSheet := Portal.Excel.Connect(myPath # 'WORKSHEET.XLS');

{ The second assignment copies the method itself without invoking
  the method. In this example, the third assignment is behaves in
  the same way as the assignment above and results in the invocation
  of the method. }
```

```
myUseCase.portalConnect := Portal.Excel.Connect;
portalConnect(myPath # 'WORKSHEET.XLS');
```

The above code fragment illsutrates that methods (whether they be procedures or functions are first class objects in Thistle and the following code is quite a valid:

```
usecase walktree(tree,apply);

   ...

   begin
      for node in tree do
         if System.NodeType(node) = 'SubTree' then
            walktree(node,apply);
         else
            apply(node);
   end;
```

And if the first call to `walktree` is:

```
walktree(myInteriorNode,printleaf);
```

Where `printleaf` is defined as:

```
usecase printleaf(leaf);

   ...

   begin
      System.WriteLn(' Value is ' # leaf);
   end;
```

Subtree nodes A subtree node is simply an interior node in the name space tree whose children are also in the name space tree within the Thistle execution environment. The the node types of children of these nodes can be of any type. In the above code fragments, the identifiers `System` and `myUseCase` represent interior subtree nodes.

Interface nodes An interface node is an interior node associate with an interface. These nodes represent the instantiated portals currently active and connected to the execution environment. These nodes represent the portal instance and any children of these nodes are interpreted as living outside of the run-time environment. The mapping of the children of these nodes to the system under test or the portal is the responibility of the portal. As far as Thistle is concerned these nodes are virtual in the that their interpretation exists outside of Thistle, but their behavour is indistinguishable from any other Thistle nodes except that they might expose some side effects of the portal.

In the following code fragment, the node `workbookA` is an interface node and the reference to the spreadsheet cell `workbook.WorkSheet.Sheet1.A`

1

is to a virtual node. In portals, updates to variables or invocation of methods can have side effects on other values. For example an application under test might respond to a transaction and update a variable which makes the transaction result available. Similarly, updating a cell on spreadsheet may cause a formula in

another cell, and hence another variable, to be updated. Not only is it possible to have such side effects in the children nodes of interface codes, it is also possible for a side effect to change the shape of the name space sub-tree below the interface node as the result of updating a variable or invoking a method under the interface node.

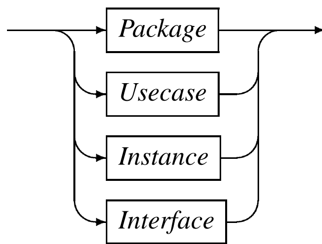
```
usecase openAccount(workBookPath);

   ...

   interface Portal.Excel : CodeMagus.Excel;

   begin

      ...

      openAccount.workbook := Portal.Excel.Connect(workBookPath);
      workbook.WorkSheet.Sheet1.A[1] := '6102045122080';

      ...

   end;
```

Nodes are created on demand and virtual nodes (below interface nodes) can be destroyed, apparantly automatically, by side effects of updates through portals. Nodes which are not the children of interface nodes and interface nodes can be explicitly deleted using the `delete` statement (See Section **??**). By being able to create nodes on demand and using the `delete` statement Thistle script code can maintain aspects of the `thistle` tree data structure.

## 2.7 Structure of Thistle Artefacts

Thistle artefacts usually contain a *Header*, a *Preamble* and a *Body*.

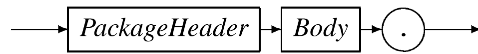
*Artefact*




A `package` is the top run unit in Thistleand corresponds to a program. From a subprogram point of view, a `package` cannot invoke other `packages`. In Thistle subprogram units do not have to be nested (and in the current implementation cannot be nested). The units that a Thistle `package` can invoke are methods or procedures and functions. There are a number of builtin or pre-defined methods and these all appear in the tree attached to the node `System` (see Chapter 7).

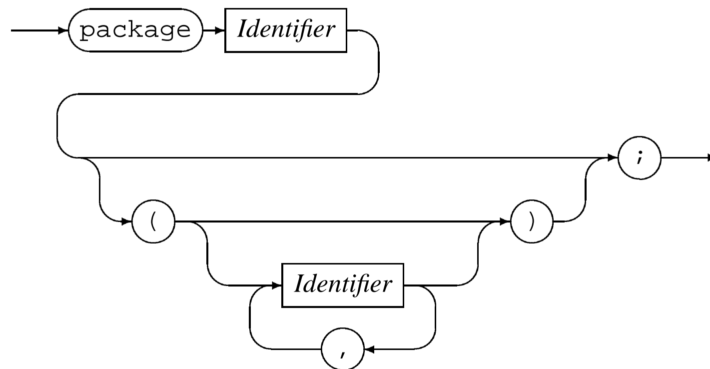
A *Package* has the following structure:

*Package*




where

*PackageHeader*



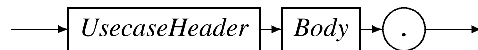

and

*Body*

(see Section 3.1 for details of the compound statement).

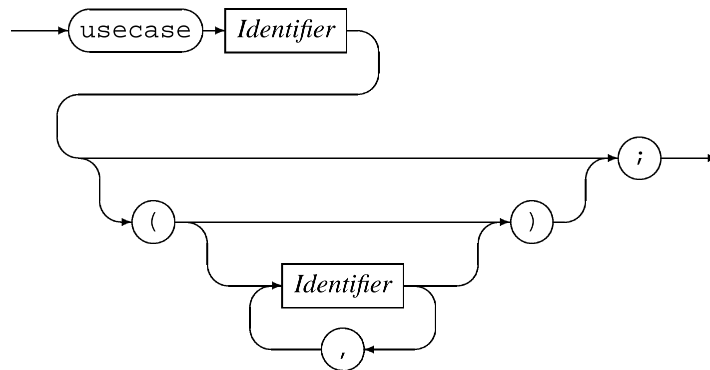
A *Usecase* is very similar to a *Package*, the only syntactical distinction is in the artefact type:

*Usecase*




where

*UsecaseHeader*



A `usecase` is an artefact which defines a method. This method is defined in a separate file which has an internal name which maps to file system path names in an implementation defined manner. For example, the definition of `Portal.Excel` in:

```
interface Portal.Excel : CodeMagus.excel;
```

defines `Portal.Excel` under the current instance of the artefact in which it occurs. The definition of the interface to this portal is taken from the external description using the path `CodeMagus.excel`. In this example, `CodeMagus` maps to an implemenation defined path name which refers to the local operating systems file system. The last node of such external artefact references always maps to the actual file name on the host system (with or without an extension and with or without any mandatory case translation). In this example, `CodeMagus.excel` might refer to the file

```
C:\Eresia\Thistle\CodeMagus\excel.tid
```

on the Microsoft platforms. This same string might map to

```
/home/testing/thistle/CodeMagus/excel.tid
```

on Unix platforms and to

```
ERESIA.CML.TIDLIB(EXCEL)
```

on MVS, OS/390 and z/OS platforms.

See Chapter 6 for details of the external definitions of interfaces.

An instance is very similar to both a `package` and a `usecase` and is not intended to be executed. The `instance` is a recorded as an audit log of the execution of a test.

## 2.8 Expressions and Operators

We mentioned earlier that the only elementary type was the string and that regardless of the apparent type of a literal, it had the semantics of a string. Expressions in Thistle operate on strings an return results which in turn are strings. This not to say that an operator does not require it operands to be within a specific domain. The numeric operators for example, require that their operands have nummeric values. Additionally, some operators, apart from not being defined over certain values may also change their meaning depending on the domain of the values of the operands. In Pascal, a similar overloading of operators applies to, for example, the relational operatores such as `<`, `=`, `>`, etc. where the `<` has a different meaning depending on the type of their operands.

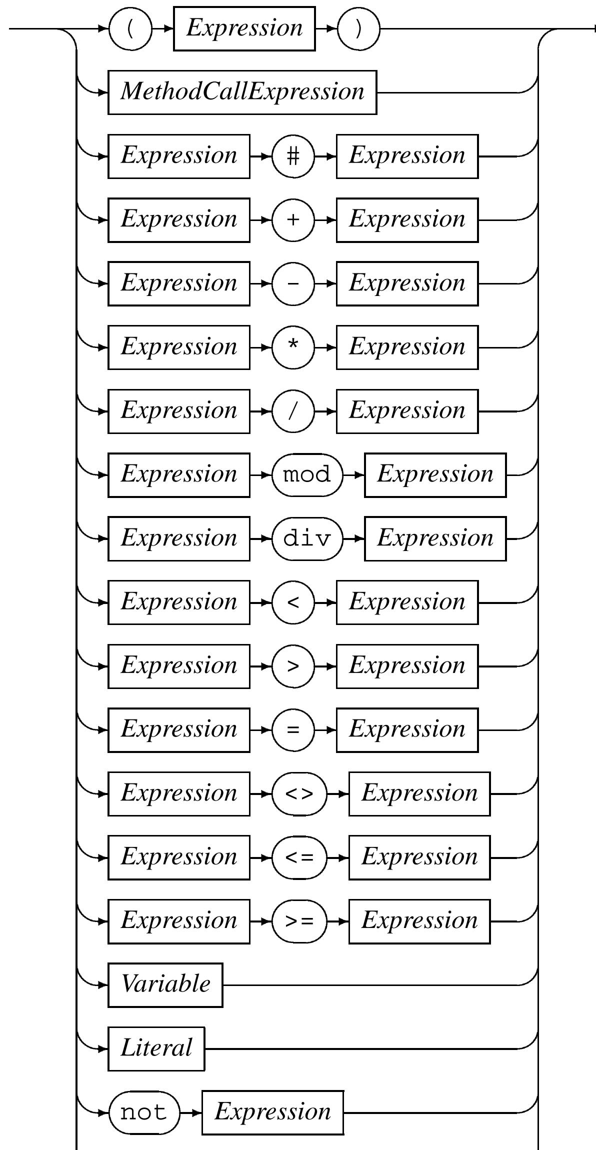
Table 2.2 lists the Thistle operators, their precedence, associativity, and whether or not they are monadic or dyadic.

As described below, the assignment operator is not a real operator as it does not result in the evaualtion of a result, but rather causes a side effect. This is different from assignment operator as found in languages such as C and C++ [**?**, **?**] in which the operator has both a side effect and a value (in C and C++ the value of the result of the assignment operator is the value of the right-hand side sub-expression). Indeed in Thistle as in Pascal, the assigment operator does not feature in the recursive definition of expressions. An expression is composed of sub-expressions and elementary items with the operators symbols and is defined recursively as follows:

| Operator | Precedence | Associatvity | Aridity | Type |
|---|---|---|---|---|
| `:=` | 0 | n/a | dyadic | Assignment |
| `<>` | 1 | left | dyadic | Relational |
| `>=` | 1 | left | dyadic | Relational |
| `<=` | 1 | left | dyadic | Relational |
| `=` | 1 | left | dyadic | Relational |
| `>` | 1 | left | dyadic | Relational |
| `<` | 1 | left | dyadic | Relational |
| `+` | 2 | left | dyadic | Arithmetic |
| `-` | 2 | left | dyadic | Arithmetic |
| `#` | 2 | left | dyadic | Concatenation |
| `or` | 2 | left | dyadic | Boolean |
| `*` | 3 | left | dyadic | Arithmetic |
| `/` | 3 | left | dyadic | Arithmetic |
| `div` | 3 | left | dyadic | Arithmetic |
| `and` | 3 | left | dyadic | Boolean |
| `mod` | 3 | left | dyadic | Arithmetic |
| `not` | 4 | left | monadic | Boolean |
| `-` | 4 | left | monadic | Arithmetic |

Table 2.2: Thistle Operators: Precedence, Associativity, and Aridity

*Expression*



**Assignment symbol** This symbol is not really an operator in the ordinary sense in that it does not produce a result which is a function of its left and right 'operands'. Instead, the evaluation of the assignment, whether it be in an assignment statement or an if-statement, causes a side effect which results in the update (and possibly creation) of the variable which appears on the left-hand side of the assignment symbol (See Section 3.2).

**Relational Operators** These operators compare their left and right sub-expressions and return a Boolean result which reflects the type of comparisson and the actual operator used. Unless both left and right sub-expressions are numeric and in the case where both are elementary, the relational operator performs a string compare in which the ordering is lexicographical. When doing so, the local colating sequence of the hosting encvironment is used.

If need be, once evaluated sub-expressions that result in elementary items will be converted to their string representation before the comparrison is performed.

Howver, when both sub-expressions are numeric, the compares are performed in a machine in dependent manner with the ordering being the regular real number ordering.

These type conversion rules apply to all the relational operators, and such conversions are implicitly performed based on the current types mplied by the values of the operators left and right sub-expressions.

The operands of the relational operators do not have to be elementary items and may be aggregates (or interior nodes with children). In these cases, the orderings are not total and are evaluated by tree-walks of the evaluated left and right sub-expressions.

The following descibes the meaning of the operators where both left and right sub-expressions evaluate to elementary items.

- < The left sub-expression is evaluated and compared to the evaluated right sub-expression and if the left sub-expression appears strictly earlier than the right sub-expression in the ordering indicated by the types of the sub-expressions as determined by the typing rules, then the operator evaluates to true; otherwise the operator evaluates to false.

  In the the following fragment were executed, the code would execute the method `System.WriteLn`:

  ```
  local.lower := 'Apple';
  local.higher := 'Peers';
  if local.lower < local.higher then
     System.WriteLn('Apples are less than Peers');
  ```

- > The left sub-expression is evaluated and is compared to the evaluated right sub-expression and if the left sub-expression appears strictly later than the right sub-expression in the ordering indicated by the types of the sub-expressions as determined by the typing rules, then the operator evaluates to true; otherwise the operator evaluates to false.

  If the following fragment results in the invocation of the method `System.WriteLn`.

  ```
  local.lower := 1;
  local.higher := 2;
  ```

```
if not (local.lower > local.higher) then
   System.WriteLn('One really is less than two!!');
```

= The left sub-expression is evaluated and compared to the evaluated right sub-expression and the operator evaluates to true of the operands are determined to be equal. As with the other relational operators, depending on whether the operands can be converted to numeric values, an implicit conversion could be performed. For example, in the following fragment, the operands of the equals-operator are compared and the result of the operation is true causing the method `System.WriteLn` to be invoked.

```
local.first_number := 0001;
local.second_number := '1';
if local.lower = local.higher then
   System.WriteLn('The numbers are equal!');
```

<> The left and right sub-expressions are evaluated and compared. The result evaulates to true if the operands are not equal. The following two fragments both cause the method `System.WriteLn` to be invoked; or neither causes the method to be invoked:

```
if local.lower = local.higher then
   System.WriteLn('The numbers are equal!');

if not (local.lower <> local.higher) then
   System.WriteLn('The numbers are equal!');
```

<= The left sub-expression and right sub-expressions are evaluated and the results compared. The operator evaluates to true if the left sub-expression appears earlier than the right sub-expression in the appropriate ordering or the operands compare equal.

>= The left sub-expression and right sub-expressions are evaluated and the results compared. The operator evaluates to true if the left sub-expression appears later than the right sub-expression in the appropriate ordering or the operands compare equal.

The relational operators are also defined in the case where both left and right sub-expressions do not evaluate to elementary items. For example, when one of the operands evaluates to an aggregate or sub-tree. In this case the operator is evaluated by walking the sub-tree comparing sub-trees and leaf nodes (or elementary items) based on the items names within the sub-trees.

In the description of the operators for the case where the operands are not both elementary items, the following code fragment is assumed to have created the context in which the relational operators are evaluated in the examples below:

```
local.A.a := 'a';
local.A.b := 'b';
local.A.c := 'c';
local.B.a := 'a';
local.B.b := 'b';
local.B.c := 'c';
local.C.a := 'a';
local.C.b := 'b';
local.D.a := 'a';
```

```
local.D.x := 'y';
```

< The left and right sub-expression operands are evaulated and if resultant left operand sub-tree is imbedded in the right operand sub-tree then the operator evaluates to true; otherwise the operator evaluates to false. For example, in the following code fragment, the method `Is.Imbedded` is invoked, but the method `Is.NotImbedded` is not invoked:

```
if local.A < local.C then
   Is.Imbedded(local.A, local.C);

if local.A < local.B then
   Is.NotImbedded(local.A, local.C);
```

If there is either an additional item (aggregate or elementary) in the evaluated left sub-tree value which does not appear in the evaluated right sub-tree value, or an elementary value in the left sub-tree is not less than the corresponding (has the same name and parents, and so on) elementary value in the right sub-tree, then the comparrison evalues to false. Further, if the evaluated left and right sub-tree operands are identical in structure and they have the same elementary leaf item values, then the operator evaluates to false.

\> The left and right sub-expression operands are evaluated and if the resultant right operand sub-tree is imbedded in the left operand sub-tree then the operator evaulates to true; otherwise the operator evaluates to false. For example, in the following code fragment, the method `Is.Imbedded` is invoked, but the method `Is.NotImbedded` is not invoked:

```
if local.C > local.A then
   Is.Imbedded(local.A, local.C);

if local.C > local.D then
   Is.NotImbedded(local.C, local.D);
```

If there is either an additional item (aggregate or elementary) in the evaluated right sub-tree value which does not appear in the evaluated left sub-tree value, or an elementary value in the left sub-tree is not less than the corresponding (has the same name and parents, and so on) elementary value in the right sub-tree, then the comparrison evalues to false. Further, if the evaluated left and right sub-tree operands are identical in structure and they have the same elementary leaf item values, then the operator evaluates to false.

= The left and right sub-expressions are evaluated and if the resultant trees are identical in structure and the elementary leaf items have the same values, then the operator evaluates to true; otherwise it evaluates to false. In the following example, the method `Is.Equal` is invoked, but the method `Is.NotEqual` is not:

```
if local.A = local.B then
   Is.Equal(local.A, local.B);

if local.A = local.C then
   Is.NotEqual(local.A, local.C);

if local.C = local.D then
```

```
    Is.NotEqual(local.C, local.D);
```

`<>` The left and right sub-expression operands are evaluated and if either the left operand sub-tree is imbedded in the right operand sub-tree or the right operand sub-tree is imbedded in the left operand sub-tree, but the two sub-trees are not idendical then the operator evaluates to true; otherwise it evaluates to false. In the following example, the method `Is.Imbedded` is invoked, but the method `Is.NotEqual` is not:

```
if local.A <> local.C then
   Is.Imbedded(local.A, local.C);

if local.A <> local.B then
   Is.NotEqual(local.A, local.B);
```

Note that the following two if-statements do not have the same meaning given that the relation operators do not always define a total ordering:

```
if local.A <> local.C then
   Is.CheckOperands(local.A, local.C);

if not (local.A = local.C) then
   Is.CheckOperands(local.A, local.C);
```

`<=` The left and right operand sub-expressions are evaluated and if the resultant left sub-tree is imbedded in the resultant right sub-tree (as defined above for the < operator) or the two sub-trees compare equal under the definition of the = operator as described above, then the <= operator evaluates to true; otherwise it evaluates to false.

`>=` The left and right operand sub-expressions are evaluated and if the resultant right sub-tree is imbedded in the resultant left sub-tree (as defined above for the > operator) or the two sub-trees compare equal under the definition of the = operator as described above, then the >= operator evaluates to true; otherwise it evaluates to false.

**Arithmetic Operators** The arithmetic operators take numeric operands and produce a numeric result. The numeric operands can be obtained by string operators and the context at the time of of the evailation of of the operator (i.e. at run-time) determines any implicit conversions which might take place. Such conversions are automatic and have not semantic surprises as all elementary types in Thistle are considered strings regardless of any internal optimised representations.

All the arithmetic operators are dyadic, except the unary negation operator which has the same symbol as the dyadic subtraction operator. Also, unlike the relational operators, the arithmetic operators can only operate on elementatry numeric items.

`+` The left and right sub-expressions are evaluated. Both are required to form a numeric result or a string result which contains valid numerics. For example, in the following fragment the Boolean expression avaluates to true and the method `System.WriteLn` is invoked:

```
if '0002'+2 = 4 then
   System.WriteLn('Computes!');
```

`-` If used as the unary negation operator, the right sub-expression is evaluated and the result is expected to be numeric. The result of rvaluating the operator returns the negation of the evaluated numeric value. In the following example the values of `local.a` and `local.b` end up having the same value, namely `-10`:

```
local.a := -10;
local.b := 10;
local.b := -local.b;
```

Note that in this example, the first assignment does not involve the negation operator and the minus sign is part of the number being assigned to `local.a`.

When used as dyadic binary operator, the operation performed is subtraction. In the following example the value that `local.b` ends up with is also `-10`:

```
local.a := 100;
local.b := 90;
local.b := local.b-local.a;
```

`*` The left and right sub-expressions are evaluated and the resultant values are expected to be numeric. The operator evaluates the product of the left and right numeric sub-expression values by by multiplyig the left sub-expression value by the right sub-expression value. In this example, `local.b` ends up with a value of `110`:

```
local.a := 10;
local.b := 11;
local.b := local.b*local.a;
```

`/` The left and right sub-expressions are evaluated and the resultant values are expected to be numeric. The operator evaluates the quotient of the left and right numeric sub-expression values by by dividing the left sub-expression value by the right sub-expression value. In this example, `local.b` ends up with a value of `5.5`:

```
local.a := 11;
local.b := local.a/2;
```

`div` The left and right sub-expressions are evaluated and the the resultant values are expected to be numeric. The `div` operator performs and generalised integer division where the result of the division retrurns a integer value. In this example, `local.b` ends up with a value of `5`:

```
local.a := 11;
local.b := local.a div 2;
```

The operation is generalised in the sense that the operands do not have to have integer values.

`mod` The left and right sub-expressions are evaluated and the resultant values are expected to be numeric. The `mod` operator returns the remainder from the division. The operator is defined as:

```
local.q := local.a div local.b;
local.remainder := local.a-local.q*local.b;
```

The `mod` operator is generalised in the same sense as the `div` operator. In the following example, `local.b` ends up with a value of `1`:

```
local.a := 11;
local.b := local.a mod 2;
```

**Boolean Operators** The operands of Boolean operators have restrictions which are similar to the arithmentic operators in that the operand sub-expressions must also evaluate to elementary numeric items. However, in order to be valid Boolean values the numeric values are expected to be in $\{0, 1\}$. The Boolean false value is represented by the integer zero and the Boolean true value is represented by the integer one.

Fundamentally different from the arithmetic operators, howver, and a departure from the usual Pascal semantics, the order of evaluation of the conjucntion `and`-operator and the disjunction `or`-operator is the same as in C and C++ [**?**, **?**]. That is, the left and right sub-expressions are evaluated in a left-to-right manner with the the right sub-expression being evaluated conditionally on the value of the left sub-expression.

`and` The left sub-expression operand is evaluated and if the operand evaluates to true, the right sub-expression operand is evaluated and the result is the value of the right sub-expression. If the left sub-expression evaluates to false, then the result of the evaluation of the operator is false.

In the following, the method `Never` is never invoked:

```
local.false := 0;
if local.false and Never('Mind!') then
   System.WriteLn('Should never write this!');
```

`or` The left sub-expression operand is evaluated and if the operand evaluates to false, the right sub-expression operand is evaluated and the result is the value of right sub-expression. If the left sub-expression evaluates to true, then the result of the evaluation of the operator is true.

In the follwing, the method `Always` is alwayes invoked:

```
local.false := 0;
if local.false or Always('Invoked!') then
   System.WriteLn('Should conditionally write this!');
```

`not` The `not` Boolean operator is a unary operator which evaluates its right sub-expression operand which is expected to evaluate to a Boolean value. The result of evaulating the `not` operator is to negate the Boolean value of the evalated right sub-expression.

For example, in the following the `local.a` ends up with the value of one (true) and `local.b` ends up with the value of zero (false):

```
local.false := 0;
local.true := 1;
local.a := not local.false;
local.b := not local.true;
```

**The String Concatenation Operator** The string concatenation operator (#) can be applied to any elementary leaf item or literal. The string representations of the resultant evaluated left and right sub-expressions are concatenated together to form the result string.

For example, the following takes a integer part and concatenates an decimal point followed by decimal digits:

```
local.amount := local.integer # '.' # local.decimalDigits;
```

# Chapter 3

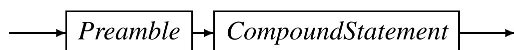
# Executable Statements

Interface definition artefacts in Thistle contain no executable statements, they simply provide definitions to the run-time system regarding the location of the corresponding portal, and the protocol for the run-time environment to interact with the portal.

Howver, being a scripting language most artefacts include some logic or code in the form of executable statements. Portals also provide logic, but apart from the interface definition, this is done at a level which is hosted by the execution evironment of the run-time system (or one of its components or connected systems). In anycase, such portal code is provided by the provider of the access to the channel covered by the portal and not by the user of Thistle.

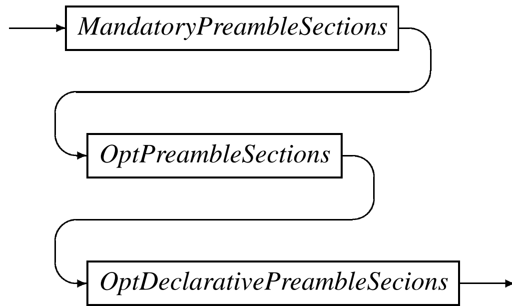
Each Thistle language artefact type or element thereof defines its methods (functions or procedures) using the same grammar. This grammar too is modeled on the executable statements of Pascal and is described in this chapter. In Chapter **??** the `usecase` and `package` artefact types were introduced. In their respective definitions both artefacts comprised a header followed by a *Body*. It is the body of an artefact type which describes the executable statements of the artefact:

*Body*




where the *Preamble* to an artefact provides certain documentation regarding the artefact. A *Preamble* comprises a number of sections some of which are mandatory and some of which are optional:

*Preamble*




*MandatoryPreambleSections* provide for information about the creator of the artefact, some description of the artefact, the date the artefact was created and the target to which the artefact applies:

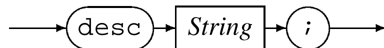
*MandatoryPreambleSections*

Note that the order in which these sections appear is important, and they must be provided in the order indicated.

*Creator*

The creator string is a regular Thistle string literal as described in Section 2.5.

*Description*




The *Description* section provides a mechanism for assiging an comment to the artefact which is formally part of the description of the artefact. The description is a regular Thistle string literal as described in Section 2.5.

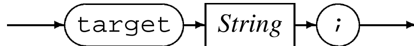
*Date*

The *Date* section is provided so that a date can be associated with the artefact. This date is interpreted as the date the artefact was created. *ISODate* has the ISO date and time format:

`yyyy-mm-ddThh:mm:ss`

Where the portion before the `T`-character is the date and the portion after the `T` character is the time stamp. In the date portion, *yyyy* is the four digit year, *mm* is two digit the month number, and *dd* is the two digit day of the month. In the format of the time-stamp, the *hh* is the hour of the day accordin to the twentry four hour clock format, *mm* is the two digit minutes passed the hour and *ss* is the two digits passed the minute.

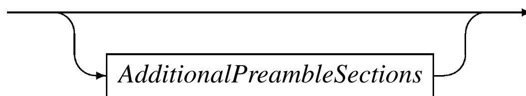
*Target*



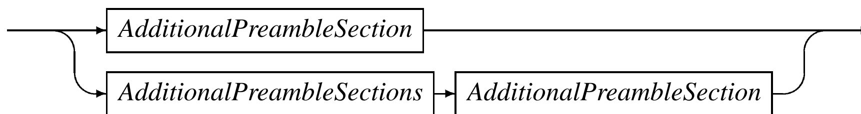

The *Target* preamble section is a mandatory comment field which indicates the target *system under test* to which the artefact applies. The target is a regular Thistle string literal as described in Section 2.5.

*OptPreambleSections* provide additional optional data regarding the artefact. Each section can appear any number of times, in any order, or not at all. These sections are provided as additional structured comments so that standardised comments can be included as part of every Thistle artefact.

*OptPreambleSections*

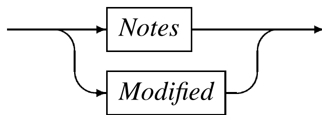



*AdditionalPreambleSections*

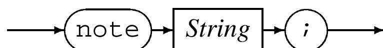



Where

*AdditionalPreambleSection*




*Notes*




The *Notes* secion is designed so that any additional commentary can be include as part of the artefact. For example, if the artefacts are being version controlled through a CVS [**?**] repository, then you might choose to described your CVS `$Log:$` entries as *Notes* strings.

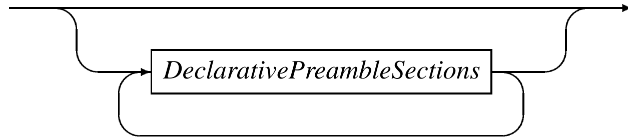
*Modified*

The *Modified* optional preamble section is provided as a means by which anyone modifying the artefact can record the name of the user who modified the artefact. A *Notes* optional preamble section can be used to record the details of the modification.

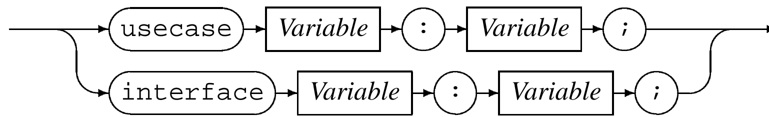
The *OptDeclarativePreambleSections* are not documentary preamble sections. These sections are used to associate an externally defined artfeact with the current Thistle artefact and to define the position in the `thistle` tree that the externally defined artefacts are to be located. Examples of such artefacts are the portal (intriduced using the

interface declarative section) and usecase introduced using a declaraitvesection of the same name (examples of these declarative sections appeared in Chapter **??**).

*OptDeclarativePreambleSections*



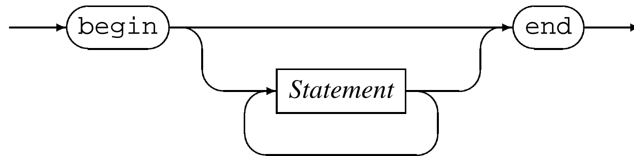

*DeclarativePreambleSections*




Whether an externally defined usecase or an interface to a portal is being introduced to the current Thistle artefact the interpretation of the *Variable*s on the left and right hand side of the semi-colon remains unchanged. The *Variable* on the left-hand side pf the semi-colon is the name in the thistle name space at which the Thistle run-time system is to attach the externally defined artefact for use within the current artefact. More often than not, this would cause the creation of the node on the left-hand side of the semi-colon.

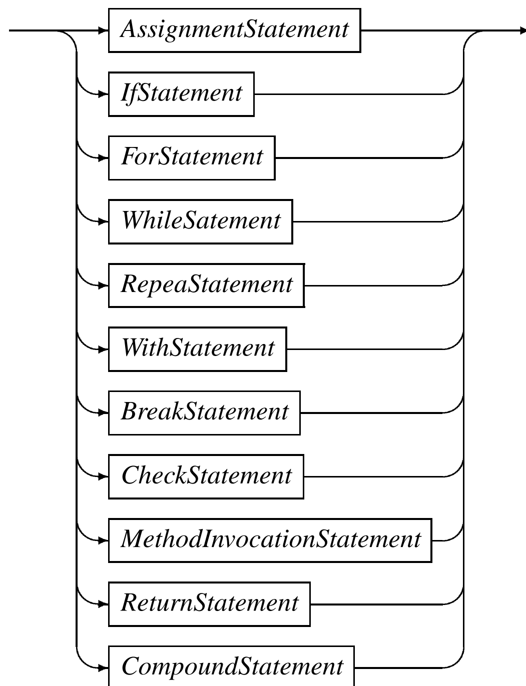
The *Variable* on the right-hand side of the semi-colon is not in the thistle name space, but is a *Variable* in the name space of the hosting run-time system. An implemantaion would map this *Variable* to the hosting systems fike system name space (as described in Chapter **??**). As mentioned in that chapter, it is the responsibity of the implementation to map the *Variable* name into the lcoal systems name space in such a manner as to mask any local-only naming conventions from the content of the artefact (see the examples in Chapter **??**).

*CompoundStatement*




A statement can be any one of the execuatble statement allowed in an artefact *Body*:

*Statement*




The Thistle statements are taken from Pascal [**?**]. Two notable and conscious omissions are the `goto` statement and the `case` statement. Addititionally, Thistle introduces statements which make the language suitable for scripting language in a testing environment, most notably the *BreakStatement* and the *CheckStatement*.

As mentioned in the previous chapter all Thistle variables can be brought into exictance by demand and do not have to be declared. This is unlike Pascal, and necessaattes a means of destroying createdvariables and sub-trees of the `thistle` name space. Thistle has a *DeketeStatement* which is analogous to the Pascal `dispose` predefined procedure.

There is no counterpart in Pascal to the *ReturnStatement*. The *ReturnStatement* in Thistle is very similar the C or C++ `return` statement.

In the following sections, each of the Thistle statements is described giving its grammar and suitable examples. It might be instructive to give a full example of a small `package`:

```
package CISCreateBusClients;

{ Preamble }

   Created by 'Han Solo';
   Description 'CIS Package';
   Date 2003-01-08T22:01:05;
   Target 'ALPHA';

   usecase CreateNew : MyPlace.CreateNewClient1;
   interface Portal.Excel : CodeMagus.Excel;
```

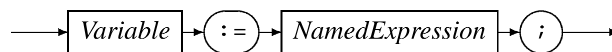
```
begin

   CISCreateBus_Nedbank.XLS1 := Portal.Excel.Connect('tspread.xls');

   for index := 1 to 1 do
      CreateNew( Fred.WorkSheet.Sheet1.A[index] );

end.
```

From the heading, the artefact in this example is a `package`, its name is `CISCreateBusClients` and that this `package` has no parameters.

## 3.1 Compond Statements

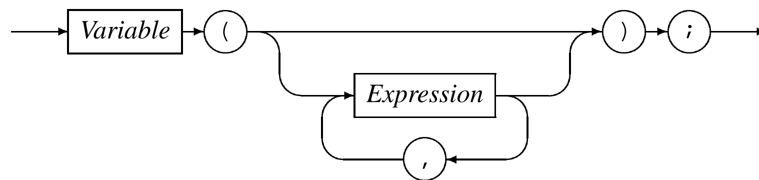
## 3.2 Assigment Statement

*AssignmentStatement*



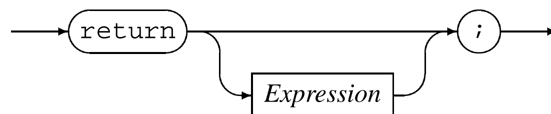

## 3.3 Transfer of Control: Method Invocation

*MethodInvocationStatement*



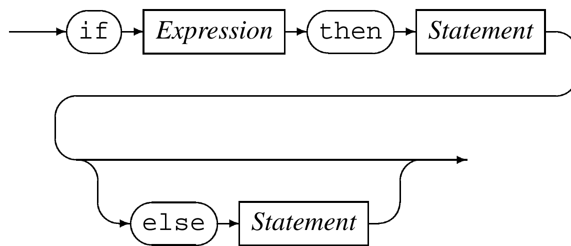

## 3.4 Transfer of Control: The `return` Statement

*ReturnStatement*



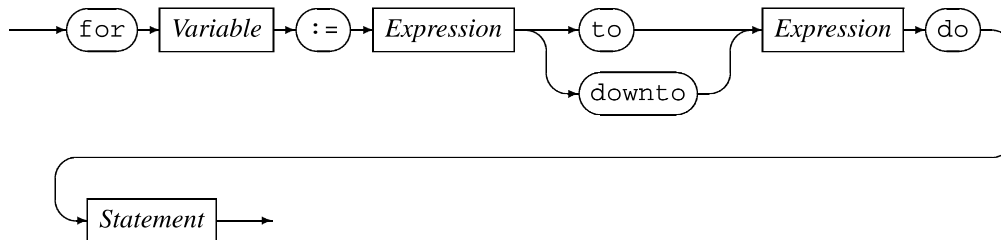
## 3.5 Conditional Execution

*IfStatement*




## 3.6 Iteraction: The `for` Statement

*ForStatement*

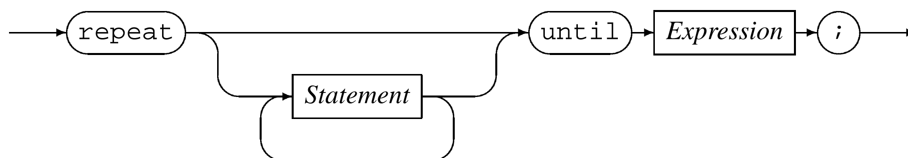



## 3.7 Loops: The `while` Statement

*WhileStatement*

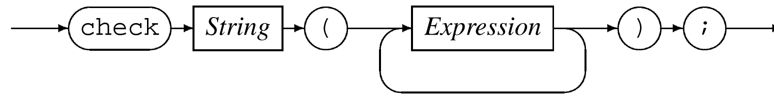
## 3.8 Loops: The `repeat` Statement

*RepeatStatement*



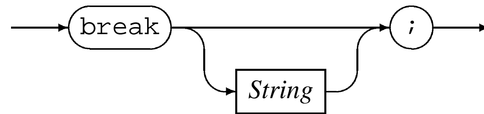
## 3.9 Interrupting Execution: The `check` Statement

*CheckStatement*



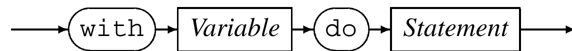

## 3.10 Interrupting Execution: The `break` Statement

*BreakStatement*




## 3.11 Choosing Name Space Scopes: The `with` Statement

*WithSattement*



## Chapter 4

# Thistle Usecases

# Chapter 5

# Thistle Instances

# Chapter 6

# Thistle Interfaces

## Chapter 7

# Thistle System Objects

# Bibliography


[1] Specification for computer programming language — Pascal. Technical Report ISO 7185-1982, 1982.

[2] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual and Report*. Springer-Verlag, third edition, iso standard pascal edition, 1985. Revised by Andrew B. Mickel and James F. Miner.