# Historical: SwiftScript Language Reference Manual

work-in-progress, $LastChangedRevision: 2611 $

Yong Zhao

## Table of Contents



# 1. Introduction

SwiftScript is a language for workflow specification in Data Grid environments, in which:

- Data lives in files, in a variety of different file system organizations and file formats;
- We want to be able to define and compose typed procedures that operate on such data; and
- We want to be able to execute these procedures on distributed resources.

SwiftScript addresses the challenges associated with such environments by defining:

- a language for describing operations on typed *data items*; and
- mechanisms for binding data items defined in this language to *datasets* stored on persistent storage.

The binding between data item and dataset is based on the XDTM (XML dataset typing and mapping) model [ref], which separates the declaration of the logical structure of datasets from their physical representation. The logical structure is specified via a subset of XML Schema, where a physical representation is defined by a mapping descriptor, which describes how each element in the dataset's SwiftScript representation can be mapped to a corresponding physical structure such as a directory, file, or database table.

This manual documents the XDTM-based SwiftScript, which uses a C-like syntax to represent XML Schema types and procedures. This C-like syntax is easier to read and write than XML, but can easily be mapped to XML.

# 2. Namespaces

Since Swift is to be used in Grid environments, the type definitions and procedure definitions can be shared across multiple virtual organizations, groups, and project development stages. Thus namespaces issue is important to address.

In general, every type definition and procedure definition has an associated namespace. When they are referenced from within another namespace, they must be referenced with their namespace specified explicitly, so as to avoid any confliction with types and procedures defined in the origin namespace.

If the namespace for a definition is not specified, it uses

'http://www.griphyn.org/vds/2006/08/nonamespace'

as the default namespace.

A namespace prefix can be defined to represent an XML-style namespace (in the form of a URI or URN), We follow the XML *prefix:localname* convention and use ':' as the separator between the namespace and the local name of a definition. Examples of namespace declaration can be found in Section ???.

# 3. Lexical structure

Lexical tokens follow the conventions of the C programming language. Specifically, there are five different tokens: identifiers, keywords, literals, operators, and other separators. White space (spaces, tabs, newlines) and comments are used to separate tokens and are ignored.

## 3.1. Comments

The characters # or // starts a comment, which terminates when a newline is encountered. The C style /* and */ pair are used for multi-line comments. For example:

```
// this is a single-line comment

# this is another single-line comment


/* multi-line comment line 1
   multi-line comment line 2 */
```

## 3.2. Identifiers

An identifier starts with an alphabetic character ('a'-'z', 'A'-'Z', '_'), after which there can be arbitrary number of letters or digits. Identifiers are case sensitive, meaning upper case letters are different from lower case ones. An identifier is used to represent the name of a variable, a procedure, a procedure argument, etc, which we'll talk in detail in later sections.

| | | |
|---|---|---|
| *identifier* | *::=* | *( letter \|'_') ( letter \| digit \| '_')** |
| *letter* | *::=* | *lowercase \| uppercase* |
| *lowercase* | *::=* | *'a ' .. 'z '* |
| *uppercase* | *::=* | *'A ' .. 'Z '* |
| *digit* | *::=* | *'0 ' .. '9 '* |

# 3.3. Keywords

Keywords are identifiers that are reserved for system use, and may not be used otherwise. We have reserved the following identifiers for type declarations and control statements:

**int float string**

**date boolean uri**

**any**

**true false null**

**namespace include type**

**if else**

**switch case default**

**while**

**foreach in step**

**repeat until**

# 3.4. Literals

Swift *literals* are constant values that are represented as strings in the program. The types and formats of literals are drawn from the set of atomic values defined by XML Schema. The type of a literal value is implicit from its context – from the type of the variable that its being assigned to or the type of the procedure parameter that it is being passed to, or the type of value that is expected in a specific position of a statement such as an *if*, *while*, or *switch*.

Some literal types can be identified without being enclosed in quotes; string literals and similar types based on strings must be enclosed in quotes.

## 3.4.1. Integer literals

An integer literal is a sequence of digits. (We may need to support octal and hexal integer literals too.)

| | | |
|---|---|---|
| *integer literal* | *::=* | *nonzerodigit digit * \| '0'* |
| *nonzerodigit* | *::=* | *'1' .. '9'* |

## 3.4.2. Float literals

A float literal has an integer part, a decimal point, a faction part, an **e,** and an optionally signed integer exponent. The integer part and the faction part both consist of a sequence of digits, where either (but not both) may be missing. The **e** together with the exponent may be missing too.

Every float literal is considered to be double-precision.

| | | |
|---|---|---|
| *float literal* | *::=* | *pointfloat \| exponentfloat* |
| *pointfloat* | *::=* | *[ intpart ] fraction \| intpart "."* |
| *exponentfloat* | *::=* | *( intpart \| pointfloat ) exponent* |
| *intpart* | *::=* | *digit +* |

| | | |
|---|---|---|
| *fraction* | *::=* | *"." digit +* |
| *exponent* | *::=* | *("e" \| "E") ["+" \| "-"] digit +* |

Examples of float literals are:

```
3.  .14  3.14  3.14e-6  2e100
```

### 3.4.3. Boolean literals

There are two boolean literals: **true** and **false**.

### 3.4.4. Date literals

A date literal is represented in quoted string conforming to ISO-8601 standard, for example:

```
"2005-09-25T11:30:00Z"
```

### 3.4.5. String literals

A string literal is a sequence of characters surrounded by two double quotes. The special string literal **null** is used to represent an uninitialized string.

### 3.4.6. XML literals

XML literals refers to verbatim XML documents. We use @ followed by a string representation of the XML document to denote such literals. For instance:

```
@“
<volume>
<image>b1.img</image>
<header>b1.hdr</header>
</volume>
”
```

### 3.4.7. URI literals

An URI literal is a string that conforms to the URI specification – IETF RFC 2396. (http://www.ietf.org/rfc/rfc2396.txt).

Example:

```
"http://www.griphyn.org/"
```

## 3.5. Operators and Separators

Operators are used in expressions for operations that involve one or more operands. Separators are for grouping and separation. The operators and separators are as follows:

| | | |
|---|---|---|
| Operators | () [ ] . | Procedure call, member reference |
| | = | Assignment operator |

| | | |
|---|---|---|
| | + - * / % | Arithmetic operators |
| | > < == != >= <= | Relational operators |
| | && \|\| ! | Boolean operators |
| Separators | { } | Block separator |
| | < > | Mapper declaration |
| | , : ; | Others |

# 4. Type Definitions

All data objects processed by Swift are typed. We distinguish between *primitive types* and *composite types.*

## 4.1. Primitive types

A primitive type is one of **int, float, boolean, date, string, uri**.

## 4.2. Composite types

A composite type is a type composed of primitive types. We support two kinds of type constructions: Arrays and Structs. We will talk more about these in the declaration section.

## 4.3. Arrays

An array is a data structure that contains zero or more elements that are all of the same type; this type is called the *element type* of the array.

Arrays are indexed by integer values, and they are 0-indexed following the C convention.

Currently only one-dimensional arrays are supported.

## 4.4. Structs

A struct is a data structure that can contain members of different types, where those types can be either primitive or composite types.

# 5. Datasets

Swift provides a logical programming model for data Grids. A SwiftScript program consists of procedure calls that operate on data items. Swift provides the level of abstraction such that operations can be specified on a data item without regard to its physical location or representation. Within the Swift logical space, a data item is called a *data object*, and its physical counterpart is called a *dataset*.

A *dataset* is a data item that has persistent physical storage. Datasets have both logical representations and physical representations. A dataset's logical structure is declared using a SwiftScript type definition, where its physical representation describes how the dataset is physically stored and cataloged on persistent storage.

A SwiftScript program specifies the operations on a dataset's logical structure. The physical dataset is accessed via a mapper, which translates between the physical, persistent structure of the dataset and its logical representation.

A physical dataset is referenced via a *dataset handle*, which contains name, type, and mapping information. The name of the dataset handle uniquely identifies the dataset; the type information specifies the logical type the dataset

conforms to; and the mapping information comprises the name of a *mapping descriptor* and the necessary parameters to the mapper. A dataset handle builds the connection between a data object and its corresponding physical dataset.

The declaration of a dataset handle is defined in Section ???

# 6. Mapping

The process of mapping, as defined by XDTM, converts between a dataset's physical representation (typically in persistent storage) and a logical XML view of that data. SwiftScript programs operate on this logical view, and mapping functions implement the actions used to convert back and forth between the logical view and the physical representation.

Associated with each logical type is a mapping descriptor, which describes the implementation of the mapping functions and necessary mapping parameters to the implementation.

A mapping descriptor contains the following fields:

- name - name of the descriptor
- description- a brief description of the mapper
- type- name of the abstract type of the dataset to map
- implementation_class- java class that implements the mapping API
- parameters- parameters for the implementation class

The implementation of the mapper must conform to the mapper API, which is a standard interface defined between mappers and data sources.

# 7. Variables

A variable represents a storage location. Each variable has a name and an associated type that determines what values can be stored in the variable. The value of a variable is the value currently stored in the storage location allocated to the variable. The value of a variable can be initialized or changed through assignment.

A variable consists of a name and a value. A value is either a literal, or a reference to a data item.

## 7.1. Global variables

Global variables are the variables declared in the main body of a SwiftScript program. A global variable extends to any procedures and blocks defined in the program and can be referenced anywhere within the program. The syntax for declaring a global variable is not different from the others, it is just that its scope applies to the whole program.

## 7.2. Local variables

A local variable occurs in a block. A block is a section of code, which consists of one or more statements that can be grouped together. Examples of a block include an *if* statement, a *switch*, or a *while* statement, etc. Blocks can be nested with one block inside another.

A local variable can be declared, for instance, within the body of a compound procedure, or in a *while* or *switch* statement.

A local variable may also be declared within a *foreach* statement as an iteration variable.

## 7.3. Dataset-bound variable

When a variable is associated with a dataset, i.e. it holds the dataset handle of that dataset; it is also called a dataset-bound variable. A dataset-bound variable usually has an associated mapping specification, for details, please look at Section ???

## 7.4. Scopes

A scope defines the visibility of a variable. A global variable extends to any procedures and blocks defined in the program. For a local variable, if it is defined in a block, its scope is limited to that block. If it is defined at the beginning of a procedure, its scope extends to any blocks contained within the procedure, unless a contained block defines a variable with the same name.

## 7.5. Variable references

A variable reference is an expression that refers to a variable or its sub-elements. A simple example of a variable reference is an identifier. For an *array* variable, subscript can be used to reference an array element. For instance if *a* is an int array, then *a[2]* is a variable reference that refers to element 3 in the array. For a *struct* variable, member names can be used to refer to member variables in the struct. For instance, if *addr* is a struct, with string members: *street*, *city*, and *state*, then *addr.city* refers to its *city* member variable.

# 8. Procedure Definitions

Datasets are operated on by *procedures*, which take one or more typed data items as input, perform computations on those data item(s), and produce zero or more data items as output.

A SwiftScript procedure can be either an *atomic procedure* or a *compound procedure*. An *atomic* procedure definition specifies an interface to an executable program or service. A *compound procedure* composes calls to atomic procedures, other compound procedures, and/or control statements: it can be viewed as a named workflow template defining a graph of multiple nodes.

A procedure definition has the form

| | | |
|---|---|---|
| *procedure-definition* | *::=* | *procedure-declarator procedure-body* |

A procedure declarator declares the output formal parameters, the name, and the input parameters of the procedure being defined. This construct is used for all procedures, regardless of the form of their body declarations.

| | | |
|---|---|---|
| *procedure-declarator* | *::=* | '(' *output-parameter-list* ')' *procedure-name* '(' *input-parameter-list* ')' |
| *parameter-list* | *::=* | *parameter ( ',' parameter) ** |
| *parameter* | *::=* | *type identifier* |

Both *output-parameter-list* and *input-parameter-list* can be optional. When there is zero or one output parameter, the parentheses for *output-parameter-list* can be omitted.

The procedure-body is different for atomic procedure and compound procedure:

| | | |
|---|---|---|
| *procedure-body* | *::=* | *atomic-procedure-body* \| *compound-procedure-body* |

## 8.1. Atomic procedure body

An atomic procedure defines an interface to an external executable program or Web Service, and specifies how data items passed as input and output parameters are mapped to and from application program or service arguments and results. While the header of an atomic procedure specifies the name of the procedure, and the inputs and outputs to the procedure, the body of such an atomic procedure specifies how to set up its execution environment and how to assemble the call to the procedure. Thus, it is in the body of an atomic procedure that *mapping operations* may appear to access components of any physical dataset that is dataset-bound to data items passed as procedure parameters.

| | | |
|---|---|---|
| *atomic-procedure-body* | *::=* | *procedure-type* '{' *invocation-config* '}' |
| *procedure-type* | *::=* | "**app**" \| "**service**" |

The body can specify the invocation of either an application or a Web Service, where *procedure-type* specifies the type of the procedure.

## 8.1.1. Application procedure body

An application procedure defines the interface to an application program that should be invoked, typically by a POSIX exec() primitive.

A program procedure body maps the SwiftScript arguments to the information needed to ultimately invoke an application through the POSIX interface, which involves setting arguments and environment variables, and passing back a return code (via an exit value).

Provisions for handling file descriptors (stdin, stdout, stderr) are provided in the body.

(TODO: environment variable and other configuration handling, probably using Profile)

In addition, we define the mapping from logical types to physical representations, via mapping functions.

| | | |
|---|---|---|
| *invocation-config* | *::=* | *application-name application-argument** ';' |
| *application-argument* | *::=* | *mapping-expression* \| *stdio-argument* |
| *mapping-expression* | *::=* | *mapping-function-call* \| *expression* |
| *mapping-function-call* | *::=* | '@' *function-name* '(' *expression* ')' |
| *stdio-argument* | *::=* | "**stdin**" ' =' *mapping-expression* \|<br>"**stdout**" '=' *mapping-expression* \|<br>"**stderr**" '=' *mapping-expression* |

Since *@filename(f)* is commonly used for getting the name of a file *f*, we introduce a shortcut for this specification, where *filename* along with the parentheses can be omitted. In this case, it can be specified as either *@f* or *@(f)*.

## 8.1.2. Service procedure body

A Web Service body specifies the URL of the WSDL description, the port type and operation to invoke, and soap message mappings.

| | | |
|---|---|---|
| *invocation-config* | *::=* | *wsdlURI port-type operation soap-message-mapping** |
| *wsdlURI* | *::=* | "**wsdlURI**" '**=**' *string-literal* ';' |
| *port-type* | *::=* | "**portType**" '=' *string-literal* ';' |
| *operation* | *::=* | "**operation**" '=' *string-literal* ';' |

| | | |
|---|---|---|
| *soap-message-mapping* | *::=* | ( “**request**” \| “**response**” ) *message-element-name* ‘=’<br>( ‘{’ *message-part-mapping** ‘}’ ) \| *mapping-expression* ‘;’ |
| *message-part-mapping* | *::=* | *message-part-name* ‘=’ *mapping-expression* ‘;’ |

(TODO: WSRF service specification)

## 8.2. Compound procedure body

A compound procedure body is a block of one or more SwiftScript statements, which are executed in an order determined by their data dependencies.

The body is comprised of *procedure-statement-sequence*, which is just a sequence of statements:

| | | |
|---|---|---|
| *compound-procedure-body* | *::=* | ‘{’ *procedure-statement-sequence* ‘}’ |
| *procedure-statement-sequence* | *::=* | *statement** |

# 9. Expressions

An expression consists of operands and operators that follow a certain sequence.

## 9.1. Primary Expressions

There are several kinds of primary expressions:

**Literals**

A literal is a value that has an associated type. We have already discussed literals in Section .

**Variables**

A variable also needs to have an associated type. Variables have been described in section .

**Member accesses**

A member access expression is an expression that accesses a member of a struct variable. It is a variable expression followed by a dot, and then followed by the name of a struct member. It has the type of the named member of the struct.

| | | |
|---|---|---|
| *member-expression* | *::=* | *primary* ‘.’ *identifier* |

Example:

```
addr.city
```

**Element accesses []**

An element access expression is an expression that accesses an element of an array. It is a primary expression followed by square brackets, containing a subscript expression. It has the type of the element type. It is also called subscription.

| *element-expression* | *::=* | *primary* ‘[’ *expression* ‘]’ |
|---|---|---|

Example:

```
itemNumbers[5]
```

**Procedure calls ()**

A procedure call expression is an invocation of a procedure. It is in the form of parenthesized list of comma separated expressions, for actual output parameters; followed by a primary expression, for function name; and then followed by parenthesized list of comma separated expressions, for actual input parameters. Output parameters should have associated types explicitly defined. The acutal paramters can be optional. When there is only one output parameter specified, the parentheses can be optional.

| *procedure-call* | *::=* | ‘(’ *output-param-list* ‘)’ *primary* ‘(’ *input-param-list* ‘)’ |
|---|---|---|
| *output-param-list* | *::=* | *typed-parameter** |
| *typed-parameter* | *::=* | *(type )? identifier* |
| *input - param - list* | *::=* | *positional-parameters ( ‘,’ keyword-parameters )?* |
| *positional - parameters* | *::=* | *expression ( ‘,’ expression )** |
| *keyword - parameters* | *::=* | *keyword-item ( ‘,’ keyword-item )** |
| *keyword - item* | *::=* | *identifier* ‘=’ *expression* |

Example:

```
File out = myproc1 ( 100, optional_arg = “v1” );
```

**Parenthesized Expressions**

A parenthesized expression is a primary expression enclosed in parentheses. The presence of parentheses does not affect its type, or value. Parentheses are used solely for grouping, to achieve a specific order of evaluation.

| *parenthesized_expression* | *::=* | ‘(’ *expression*‘)’ |
|---|---|---|

Example:

```
int i = (a + b) * 5;
```

# 9.2. Operators

Operators in an expression indicate what kind of operations to apply to the operands. Currently we support an assignment operator, arithmetic operators and relational operators.

## 9.2.1. Assignment Operators

The assignment operator = assigns the value of the right operand to the left operand. The left operand must be a variable reference.

## 9.2.2. Arithmetic Operators

Currently we support arithmetic operators + - * / %

### 9.2.3. Relational Operators

The relational operators ==, !=, <, >, <= and >= are comparison operators, and the result of the comparisons evaluates to either **true** or **false**. For instance, *x==y* evaluates to true is x is equal to y, and false otherwise.

### 9.2.4. Boolean Expressions

A boolean expression is an expression that evaluates to either true or false. There are three boolean operators: && || ! for AND, OR, and NOT operations respectively.

The controlling conditional expression of an if-statement, while-statement, or repeat-statement is a boolean expression.

# 10. Statements

## 10.1. Namespace Statement

The namespace statement MUST appear at the very beginning of a SwiftScript program, and the namespace must be unique. It serves similar purpose as a Java package definition, so that the type definitions and procedure definitions defined in this namespace would not collide with others defined outside. The syntax for namespace definition is as follows:

"**namespace**" (*prefix*)? '"' *uri* '"' (';')?

*prefix* is the abbreviation of the namespace denoted by *uri*. If prefix is ommitted, then the namespace is regarded as the default namespace. If a default namespace is not defined in the program, it assumes the value

"http://www.griphyn.org/vds/2006/08/nonamespace"

Some examples:

```
namespace"http://www.griphyn.org/"

namespacefmri"http://www.fmridc.org/"
```

For the definitions that follow the namespace statement, they all belong to the default namespace unless otherwise specified.

## 10.2. Include Statements

An include statement is used to include type definitions defined in an external XML Schema document, or to include another program defined in SwiftScript, so that the type definitions and procedure definitions can be used directly within the current SwiftScript program.

An include statement is of the form:

"**include**" '"' *include-file-name* '"'

Since the definitions in the included file may have a different namespace from the one in the current program, it is necessary to explicitly specify the namespace for those definitions when they are used in the current program.

# 10.3. Type Definitions

A type definition is usually used at the beginning of a program, to define the structure of a new type, which can later be used to declare a variable. Type definitions have the form:

| | | |
|---|---|---|
| *type-definition* | *::=* | "**type**" *type-name type-specifier* ';' |

Where the *type-name* is a unique identifier and the *type-specifier* is either an already defined type, such as primitive types, or a struct declaration.

## 10.3.1. Type Specifiers

The type specifiers are

"**int**"

"**float**"

"**string**"

"**boolean**"

"**date**"

"**uri**"

*struct-declaration*

## 10.3.2. Struct Declarations

A struct declaration is of the form:

| | | |
|---|---|---|
| *struct-declaration* | *::=* | '{' *type-declaration-list* '}' |

The *type-declaration-list* is a sequence of type declarations for the members of the struct.

| | | |
|---|---|---|
| *type-declaration-list* | *::=* | *type-declaration** |

A type declaration is of the form:

| | | |
|---|---|---|
| *type-declaration* | *::=* | *type-specifier declarator-list* ';' |

The *declarator-list* is a comma-separated sequence of declarators. Each declarator can be an identifier, or an array declarator, which is an identifer followed by [ ], with an optional array size designated by an integer literal.

| | | |
|---|---|---|
| *declarator-list* | *::=* | *identifier* \|<br>*identifier* '[' *integer_literal?* ']' |

| | | |
|---|---|---|

For example, an order with an order number, a description, and a sequence of item numbers can be specified as follows:

```
type order {
    int orderNumber;
    string description;
    int itemNumbers[];
}
```

# 10.4. Declaration Statements

A declaration statement declares a variable, or a physical dataset in the form of a *dataset handle*.

## 10.4.1. Local Variable Declaration

A local variable declaration declares one or more local variables.

| | | |
|---|---|---|
| *local-variable-declaration* | *::=* | *type local-variable-declarator-list ';'* |
| *local-variable-declarator-list* | *::=* | *local-variable-declarator ( ',' local-variable-declarator )** |
| *local-variable-declarator* | *::=* | *identifier ( '=' local-variable-initializer )?* |
| *local-variable-initializer* | *::=* | *expression \| array-initializer \| range-initializer* |
| *array-initializer* | *::=* | *'[' expression ( ',' expression )* ']'* |
| *range-initializer* | *::=* | *'[' expression ':' expression ( ':' expression ) ? ']'* |

Some examples:

```
int x, y=2;
String s = "hello";
floatf[] = [1.0, 2.0, 3.0];
intp[] = [1 : 9 : 2];  // numbers 1 3 5 7 9
```

Note a range initializes an array with a series of values with a fixed step, with a default step 1.

## 10.4.2. Dataset Declaration

A physical dataset is referenced by a *dataset handle*, which contains name, type and mapping information of the dataset.

| | | |
|---|---|---|
| *dataset-declaration* | *::=* | *type dataset-name '<' mapping-description '>' ';'* |
| *dataset-name* | *::=* | *identifier* |
| *mapping-descrition* | *::=* | *mapping-descriptor ( ';' mapping-parameter-list )?* |
| *mapping-descriptor* | *::=* | *identifier* |
| *mapping-parameter-list* | *::=* | *mapping-parameter ( ',' mapping-parameter )** |
| *mapping-parameter* | *::=* | *identifier '=' mapping-expression* |

A sample dataset declaration is shown as follows:

```
Imageimg1<image_mapper; location="/home/archive/images/image1.jpg">;

Imageimg2<simple_mapper; prefix=@img1, suffix=".2">;
```

As a dataset handle is no more than a variable holding a dataset, we can also call it a *dataset-bound variable.*

# 10.5. Expression Statements

Most statements are expression statements, they take the form:

*expression* ';'

Usually expression statements are assignments, or procedure calls.

# 10.6. Selection Statements

A selection statement selects one of a number of possible statements for execution, based on the value of a boolean expression.

| | | |
|---|---|---|
| *selection-statement* | *::=* | *if-statement* \| *switch-statement* |

## 10.6.1. The if statement

The if statement selects a statement for execution based on the value of a boolean expression.

| | | |
|---|---|---|
| *if-statement* | *::=* | "**if**" '(' *boolean-expression* ')' '{' *statement** '}'<br>*(* "**else**" '{' *statement** '}' *)?* |

## 10.6.2. The switch statement

The switch statement selects one of many statement lists for execution based on the value of the switch expression.

| | | |
|---|---|---|
| *switch-statement* | *::=* | "**switch**" '(' *expression* ')' *switch-block* |
| *switch-block* | *::=* | '{' *switch-section** '}' |
| *switch-section* | *::=* | *switch-label statement** |
| *switch-label* | *::=* | "**case**" *constant-expression* ':' \| *(* "**default**" ':' *)* |

# 10.7. Loop statements

A loop statement repeatedly executes some statements in the loop body. It can be of one of the following statements:

| | | |
|---|---|---|
| *loop-statement* | *::=* | *foreach-statement* \| *while-statement* \| *repeat-statement* |

### 10.7.1. The foreach statement

The foreach statement iterates over the elements of a collection, and executes the embedded statement for each of the elements.

| | | |
|---|---|---|
| *foreach-statement* | *::=* | "**foreach**" *type? identifier ( ',' index-identifier )?*<br>"**in**" *expression* "**step**" *int-literal* '{' *statement** '}' |

The *type* and *identifier* of a foreach statement declare the iteration variable of the statement. if the *identifier* is defined before the *foreach* statement, then *type* is optional. The type of the *expression* in the foreach statement must be a collection type. The *step* controls how far off the iteration jumps forward to another element, and the *index* variable is an integer variable to track the current position of the iteration.

### 10.7.2. The while statement

The while statement executes an embedded statement zero or more times conditionally based on a boolean expression.

| | | |
|---|---|---|
| *while-statement* | *::=* | "**while**" '(' *boolean-expression* ')'<br>'{' *embedded-statement* '}' |
| *embedded-statement* | *::=* | *statement** |

A while statement is executed as follows:

- First the *boolean-expression* is evaluated.
- If it is evaluated to true, control is transferred to the embedded statement. When control reaches the end point of the embedded statement, control goes back to the beginning of the while statement.
- If the boolean expression yields false, control is transferred to the end point of the while statement.

### 10.7.3. The repeat statement

The repeat statement executes an embedded statement zero or more times conditionally based on a boolean expression.

| | | |
|---|---|---|
| *repeat-statement* | *::=* | "**repeat**"'{' *embedded-statement* '}'<br>"**until**" '(' *boolean-expression* ')' ';' |

The repeat statement is slightly different from the while statement in that control goes to the embedded statement first, and the boolean expression is evaluated, if true, then control goes to the end point of the repeat statement, otherwise, control goes back to the embedded statement.

## 10.8. The break statement

The statement

“**break**” ‘;’

causes termination of the smallest enclosing loop, or switch statement; control passes to the statement following the terminated statement.

## 10.9. The continue statement

The statement

“**continue**” ‘;’

causes control to pass to the loop continuation portion of the smallest enclosing loop statement; that is to the end of the loop.

# 11. Examples

For detailed examples, please refer to the User Guide document in the Swift public release.

# 12. Extensions to consider

- More atomic types, such as those defined in XML Schema

- Type inference: if the type of a formal parameter to a procedure can be inferred from its definition, then the type does not need to appear in the procedure signature.

For example, if you write

(c) myfunction (a,b)

{

tmp=combineImages(a,b)

c=invertImage(tmp)

}

as long as you have function prototypes for combineImages and invertImage,

you can infer from the program the types for the variables a,b and c and

hence the prototype for myfunction... and so on for the entire program.

- Literal XML snippets instead of quoted XML to avoid quoting problem.

- Blocks within a procedure, with new scopes for declaring variables

- Ability to invoke an XPath to extract a value from a document