

# **WebHare Application Portal / Lite**

## HareScript Language Reference

Date:	April 20th, 2006
Number of pages:	54
Version:	2.32
Target audience:	Template developers Module developers

## Table of Contents

---

1. Introduction .....	1
1.1 What you should already know.....	1
1.2 Prerequisites.....	1
2. Background information .....	2
2.1 Language history .....	2
2.2 What's new? .....	2
3. Language structure .....	3
3.1 HareScript code and comments .....	3
3.2 HareScript statements and blocks.....	3
3.3 Identifiers .....	4
3.4 Definitions .....	4
Global variables .....	5
Local variables .....	5
Visibility rules and name conflicts .....	5
Ordering of definitions.....	6
3.5 Functions and Macros .....	6
Argument lists .....	7
Attributes.....	7
External functions .....	8
Aggregate functions .....	8
3.6 Libraries .....	8
3.7 Main code .....	9
3.8 Column name lookup rules.....	10
4. Constants and initialisers .....	12
4.1 String constants and escape sequences.....	12
UTF-8 encoding .....	12
4.2 Numerical constants .....	13
Overriding a constant's type .....	13
Hexadecimal and binary constants.....	13
4.3 Boolean constants .....	14
4.4 Record initialisers .....	14
4.5 Array initialisers .....	14
4.6 Default values .....	15
5. Variable types .....	16
5.1 Integer.....	16
5.2 Boolean.....	16
5.3 String .....	16
5.4 Blob.....	17
5.5 Datetime .....	17
5.6 Money .....	18
5.7 Float.....	18
5.8 Array types.....	18
Automatic conversion of record arrays .....	19
5.9 Record .....	19
5.10 Table.....	19
NULL conversions.....	20
KEY lists.....	21
5.11 Variant .....	21
5.12 Function pointers .....	21
6. Operators .....	23
6.1 Assignment operator .....	23
6.2 Merge operators .....	23
6.3 Comparison operators .....	24
6.4 Arithmetic operators .....	25
6.5 Logical operators .....	25

6.6	Array subscript operator .....	26
6.7	Cell operator .....	26
6.8	Conditional operator .....	26
6.9	IN operator .....	26
6.10	LIKE operator.....	27
6.11	Bit operators .....	27
6.12	TYPEID operator .....	28
7.	Control statements.....	29
7.1	IF ... ELSE .....	29
7.2	FOREVERY .....	29
7.3	WHILE .....	30
7.4	FOR .....	31
7.5	RETURN.....	31
7.6	BREAK.....	32
7.7	CONTINUE .....	32
7.8	SWITCH, CASE, DEFAULT .....	32
8.	Source code commenting .....	34
8.1	Extractable comments .....	34
8.2	Comment examples.....	34
9.	SQL statements .....	36
9.1	SELECT .....	36
	Columns.....	36
	The FROM clause.....	37
	The WHERE clause .....	38
	The GROUP BY clause .....	39
	The HAVING clause .....	40
	The ORDER BY clause .....	40
	The LIMIT clause .....	41
9.2	INSERT.....	41
	Table inserts .....	41
	Array element inserts.....	41
	Record inserts.....	42
	Record cell inserts .....	42
9.3	UPDATE .....	42
9.4	DELETE.....	43
	Table deletes .....	43
	Array element deletes.....	43
	Record deletes.....	44
	Cell deletes .....	44
Appendix	.....	45
Appendix 1:	Listing of reserved words .....	46
Appendix 2:	Operator precedence .....	47
Appendix 3:	Deprecated HareScript features.....	48
Appendix 4:	ASCII table .....	50
	Control characters.....	50
	Printable characters.....	51

## 1. Introduction

---

With the release of the second version of WebHare® the HareScript® programming language has been completely renewed and updated, giving you even more power and flexibility in creating WebHare templates and modules.

This reference will explain the syntax of the HareScript language in detail. For information about the HareScript libraries (eg. all functions and macro's), you should refer to the [library references](http://www.webhare.net/library-references) on <http://www.webhare.net/>

This reference is not intended to teach you the basics of the HareScript language itself. If you have no prior experience with HareScript, you should probably refer to the HareScript manual first.

### 1.1 What you should already know

For optimal use of this reference you should already have some experience writing simple HareScript templates or scripts.

Previous experience with scripting languages like JavaScript, PHP or ASP and some knowledge of relational databases and the Structured Query Language (SQL) can be useful, but is not necessary to successfully use this reference.

### 1.2 Prerequisites

The information in this reference can be used with WebHare CMS, version 2.10 and up, and in all versions of WebHare Lite. Older versions of WebHare CMS will not support all language constructs described in this reference.

The implementation of the HareScript language does not differ between WebHare Lite and WebHare CMS, as they use the same HareScript compiler. However, WebHare Lite does not support connecting to external databases, and comes with fewer libraries than WebHare CMS.

## 2. Background information

---

This chapter will give some background information about the origins of the HareScript language. Although this information is not essential for using the language, it may be useful to understand why the language has been developed the way it is, and why some decisions were taken.

### 2.1 Language history

HareScript started in 1999 as a support language for WebHare's Word converter, as a means for the end-user to specify how design aspects such as navigation and table of contents would be created. In its first versions, it only supported *Select*, *Forever*, and a few other functions such as *Left* and *Length* (it didn't even have a *Right* function).

Over time, as more features were added to the language, the role of the HareScript inside WebHare grew. With WebHare CMS version 2.10, and the release of WebHare Lite, HareScript now offers many advanced features, such as TCP/IP support, database connectivity, and an optimising compiler. A large part of WebHare CMS itself, such as the web interface, has been exclusively written in HareScript.

### 2.2 What's new?

The HareScript language and libraries have undergone major changes, and the new HareScript environment will be shipped with WebHare CMS version 2.10 (both the Professional and Enterprise versions), and all versions of WebHare Lite. The complete list of changes is too numerous to list, so only the most important changes will be highlighted.

The most notable difference is the inclusion of a separate HareScript compiler, which offers the following advantages:

- Warnings can be generated for dangerous or possibly wrong code constructs.
- The code is optimised for faster execution.
- A smarter language parser is able to move much error detection from the run-time phase to the compilation phase, leading to more robust code.

The following improvements have been made to the core language:

- The virtual machine can now handle multiple database transactions simultaneously.
- HareScript can connect with various external databases (eg. ODBC, LDAP) but still allows the same simple syntax.
- New data types have been added, such as FLOAT and MONEY.
- Record and array manipulations have been made more powerful

The following additions have been made to the HareScript libraries:

- TCP/IP connectivity has been added, and libraries are already available to handle FTP, HTTP, SMTP, POP3 and other common Internet protocols.
- A documentation generator library (similar to Javadoc) has been added, and HareScript libraries can generate and handle their own documentation.
- The underlying file system and various operating system facilities can be directly managed.

Unfortunately, we've also had to deprecate some language features that we felt didn't properly fit in its overall design. A list of deprecated features can be found in [Appendix 3](#). These features are still supported by the current HareScript compiler, but will generate warnings when used, and may be removed in future versions.

### 3. Language structure

---

This section shows the basic building blocks of a HareScript file, and introduces some of the terms that will be used throughout this manual.

#### 3.1 HareScript code and comments

HareScript code is always placed between '<?wh' and '?>' tags, and is often embedded inside other code or text, such as HTML code.

Inside HareScript code, comments can be added by prefixing them with '/'\* or '//'. Any comment starting with '/'\* must be terminated by '\*'/', and any comment starting with //' automatically ends at the end of the line. However, if a HareScript close tag ('?>') appears anywhere inside a comment, it will terminate the comment *and* close that block of HareScript code.

Some of the above character sequences are interpreted differently when they are contained inside a *string constant*. A string constant is a sequence of characters contained between either single (') or double (") quotes, which is not part of a comment. In other words, a single or double quote appearing after a '/' character sequence, is not interpreted as a real string constant. Inside string constants, HareScript close tags ('?>') and comment starting characters ('/\*' and '//') are not recognized.

The following code fragments show the interactions between end tags, comments and string constants:

```
<?wh /* This is a valid way to start and end a comment */ ?>
<?wh /* This comment is ended implicitly by the next closing tag ?>

<?wh // The following statement is never executed: PRINT("Hello, World!"); ?>
<?wh // The following statement is executed: ?> <?wh PRINT("Hello, World!"); ?>

<?wh PRINT("// This text is printed, and not recognized as a comment"); ?>
<?wh PRINT("/* This text is printed, and not recognized as a comment */"); ?>
<?wh PRINT("The closing tag ?> is printed, and does not end this code block"); ?>

<?wh /* 'Even though quotes appear here, the following close tag DOES end this
comment block, as quotes inside a comment are ignored: ?>
```

It's not required to close the last block of HareScript code with a '?>'. Omitting the closing tag can be useful when writing libraries, as some editors may append empty lines to the end of a library. If these empty lines would appear outside a HareScript code block, they may appear as empty lines at the top of the HareScript output.

#### 3.2 HareScript statements and blocks

In HareScript, a semicolon (;) is used to terminate statements. Statements can be grouped together into blocks using curly braces ('{' and '}'). Grouping statements can be used to add structure to a block of code, and to use multiple statements with control statements such as IF and FOREVERY. The following code fragments demonstrates the use of semicolons and statement blocks:

```
/* The first PRINT is not executed, the second one is,
because the first semicolon terminates the IF statement */
IF (false) PRINT("Not displayed"); PRINT("Displayed");

/* Both PRINT statements are executed when curly braces are used */
IF (false) { PRINT("Not displayed"); PRINT("Not displayed either"); }

/* Even in statement blocks, the last statement must
be terminated by a semicolon: the following code is illegal: */
{ PRINT("Illegal code") }
```

```

/* A statement block 'counts' as a single statement, so it shouldn't
   be followed by a semicolon in a context where only one
   statement is permitted.
   This makes the following code legal: */
IF (true) { PRINT ("True"); } ELSE { PRINT ("False"); }

/* But the following code is illegal, because only one statement or block may
   appear between IF and ELSE: */
IF (true) { PRINT ("True"); }; ELSE { PRINT ("False"); }

```

A HareScript close tag is always interpreted as a semicolon. This is important to realize when a HareScript close tag is immediately used after a WHILE or IF statement. As the following example shows, you'll often want to use a semicolon block in that case:

```

<?wh
/* The following code is illegal, it looks to the compiler as:
   'IF; (false)', and the semicolon after the IF is illegal: */
IF ?><?wh (false) PRINT("false");

/* The following text is always printed, it looks to the compiler as:
   'IF (false); PRINT("This is printed"); - the semicolon terminates the IF */
IF (false) ?>This is printed<?wh ;

/* The following text is not printed, it looks to the compiler as:
   'IF (false) { PRINT ("This is printed"); } */
IF (false) { ?>This is printed<?wh }

```

### 3.3 Identifiers

An 'identifier' is a name for a variable, function, macro or cell in HareScript. All identifiers have the same common rules determining how they can be named, and which names are considered identical.

An identifier name may be up to 64 characters in length, and must start with either a letter or an underscore. An identifier name may contain only letters, numbers and underscores.

All identifier names are case-insensitive - the identifiers "myint" and "MyInt" are considered equal. The reserved keywords, listed in [Appendix 1](#), may never be used as an identifier name.

Cell names have fewer restrictions than identifier names, so it may be necessary to 'escape' a column name by using a string constant. This is only permitted in contexts where the HareScript compiler can unambiguously determine that a column is being referred, as shown in the following example:

```

// Access cell 'column data' from record rec;
STRING mystr := rec."column data";

// Select the cell 'column data', renaming it to coldata, from table MYTABLE
SELECT mytable."column data" AS coldata FROM mytable;

/* Illegal: the compiler isn't sure whether you're trying to
   create a column with the contents "column data", or selecting
   the column "column data" */
SELECT "column data" FROM mytable;

```

### 3.4 Definitions

```

[ PUBLIC | PRIVATE ] <type> <variablename> [ := value ]
[ , <variablename> [ := value ] ... ];

```

```
[ PUBLIC | PRIVATE ] TABLE <table-specification> <variablename> [ := value ];
```

A variable is defined in HareScript by specifying its type, its name, and optionally assigning it a value and an external visibility specifier.

The external visibility specifier, PUBLIC or PRIVATE, is optional and defaults to PRIVATE if not specified. This specifier is explained in the [libraries](#) section. The external visibility specifier may only be used for global variables. For tables, an [table specification](#) must be provided.

The initial value for a variable is also optional. Every type has its own [default value](#) that will be used when no value is specified at its point of definition. Except when defining TABLE variables, more than one variable of the same type can be defined in one statement by separating the definitions by a comma.

### Global variables

A variable that is defined outside any statement block (a block delimited by '{' and '}') is considered a *global* variable. A *global* public variable is shared between libraries if both libraries are loaded (directly or indirectly) by a single script. For example, in the code below, there is only one variable 'i', even though library lib1 is loaded twice.

```
//Contents of library 1:
PUBLIC INTEGER i := 1;

//Contents of library 2:
LOADLIB "library 1";
i := i + 1;

//Contents of library 3:
LOADLIB "library 1";
i := i + 1;

//Contents of the HareScript that is being executed
LOADLIB "library 2";
LOADLIB "library 3";
LOADLIB "library 1";
PRINT ("i = " || i); //this will print '3'.
```

### Local variables

A variable that is defined inside a statement block (a block delimited by '{' and '}') is considered a *local* variable. It is only visible inside the block in which it was defined, and any blocks contained in that block. If a variable is defined inside a function, it is created for every call to the function. For example, in the following code, each call to function Faculty has its 'own' variable 'i' - they cannot see each other's variable and 'i' will never have a value higher than '1':

```
INTEGER FUNCTION Faculty(INTEGER value)
{
  INTEGER i;
  i := i + 1;
  PRINT ("i is now " || i || "\n");
  IF (value<=1) RETURN 1;
  RETURN value * Faculty(value-1);
}

INTEGER faculty_of_5 := Faculty(5);
```

### Visibility rules and name conflicts

It is illegal to define two variables with the same name inside the same statement block. However, it is allowed to define a local variable with a name that is already used in another statement block, or to define a local variable with a name that is already used by

a global variable, function or macro. When an expression refers to a variable, it will always use the 'closest' variable definition, as demonstrated in the following example:

```
INTEGER i := 1;
MACRO MyMacro()
{
  INTEGER i := 2;
  {
    INTEGER i := 3;
    PRINT("i is now " || i || "\n"); //prints '3'
  }
  PRINT("i is now " || i || "\n"); //prints '2'
}
PRINT("i is now " || i || "\n"); //prints '1' (the global variable 'i')
```

It is illegal to define a global identifier (variable, function or macro) with the same name as another global identifier defined or [exported](#) by the same library.

### Ordering of definitions

In contrast to functions and macros, all variables must be defined before they are first used. All variables receive their initial value, if any, at the point of their definition. If a variable is read before the code at its point of definition has been executed, the reader will see the variable's [default value](#), as shown in the following code:

```
///This will print '0', because variable i is not yet initialized
///at this point
PRINT (GetVarI() || "\n");

INTEGER i := 5;

///This will now print '5'
PRINT (GetVarI() || "\n");

INTEGER FUNCTION GetVarI()
{
  RETURN i;
}
```

## 3.5 Functions and Macros

```
[ PUBLIC | PRIVATE ] <type> [ AGGREGATE ] FUNCTION <name> ( [arguments] )
[attributes]
{ <code> }

[ PUBLIC | PRIVATE ] MACRO <name> ( [arguments] ) [attributes]
{ <code> }
```

Functions and macros allow you to reuse code and organise HareScript code into logical portions. Some functions and macros also allow you to interface with the 'external world', such as printing HTML code, opening a file, or committing a database transaction.

A function can optionally take one or more arguments, and must always return a value using the keyword [RETURN](#). A macro can also take arguments, but never returns a value.

All functions and macros must be defined at the 'top level', after any `LOADLIB` command. They cannot be defined inside another function or macro, or inside any statement block delimited by curly braces.

The external visibility specifier, `PUBLIC` or `PRIVATE`, is optional and defaults to `PRIVATE` if not specified. This specifier is explained in the [libraries](#) section.

The 'type' before the keyword `FUNCTION` specifies the type of data the function returns. The [argument lists](#) and [attributes](#) are explained in the following sections. The function

definition must follow immediately, between curly braces, unless the function has been marked as an [external function](#) in its attribute list.

The AGGREGATE keyword converts a function into an aggregate function.

Function names must follow the same rules as other [identifiers](#). It is not allowed to define a function or macro with the same name as any other global identifier defined or [exported](#) by the same library.

### Argument lists

A function or macro can take one or more arguments. For every argument a function takes, it must define a name and an expected type, and it can optionally define a 'default' value if the argument is omitted by using the *defaultsto* keyword. In an argument list, no arguments without a default value may follow an argument with a default value.

The following code gives some examples of legal and illegal argument lists:

```
//A function returning an integer, and taking two integers
INTEGER FUNCTION Multiply (INTEGER left, INTEGER right)
{ RETURN left * right; }

//A function returning a money value, and taking one or two money values
MONEY FUNCTION ExchangeMoney (MONEY originalamount
                             , MONEY rate DEFAULTSTO 2.20371)
{ RETURN originalamount * rate; }

//Illegal: all arguments must have a type and a name
MACRO badmacro(firstarg, INTEGER)

//Illegal: all arguments following a default value must also have
//a default value
INTEGER FUNCTION badfunc(INTEGER firstarg DEFAULTSTO 2, INTEGER secondarg)
```

### Attributes

A function can also have one or more attributes, which tell the compiler how to handle this function. Most attributes are only relevant for [external functions](#), but the *deprecated* attribute may also be useful for library designers.

The *deprecated* attribute will tell the compiler to give a warning whenever the marked function is used. The function can still be called normally, but the attribute can be used as an advance warning that the function will disappear in the future:

```
MACRO OldCode(INTEGER i) ATTRIBUTES(DEPRECATED "Please use NewCode")
{
}

/* This line will now give the warning:
   'OldCode' has been deprecated: Please use NewCode */
OldCode(1);
```

The *constant* attribute tells the compiler to assume that this function will not change global variables or the general 'state' of the HareScript system. In other words, as long as no global variable or external factor is changed, this function will always give the same results when called with the same arguments. An example of such a function would be a simple multiplication function, but *DeleteDiskFile* would definitely not be constant.

You're unlikely to ever need to specify the *constant* attribute for HareScript functions you write yourself, because the compiler will generally be able to figure out itself whether a function is constant. This attribute is mostly useful for [external functions](#).

The *skiptrace* attribute tells the compiler that this function may not appear in a stack trace or an error. This attribute is useful for functions that are called to generate errors (like *Abort*). External functions have this attribute implied.

The *external* attribute, optionally followed by a string constant specifying the module name, tells the compiler that the function is not a regular HareScript function. External functions are explained in the [following section](#).

The *invokesfunctionpointer* attribute can only be used in conjunction with the external attribute, and tells the compiler that the external function can call other HareScript functions.

The *terminates* attribute tells the compiler that this function terminates execution of the current script. It can only be used for macro's.

### External functions

External functions are written in a foreign language (such as C++), and are used to communicate with the 'outside world' or to access built-in features of the language. External functions can be built into the HareScript virtual machine (or an extended virtual machine, such as WebHare), or contained in an external library (a .DLL or .so file)

You cannot just call any function in any DLL using the external attribute, as the function needs to be prepared to support HareScript function calls. Details on how to do this can be found in the module development section of <http://www.webhare.net/>

For illustration, an example external function definition follows. Note that you will probably not be able to actually run this code in a HareScript library, as the example function names will not be available (you will get a 'function not registered' error).

```
//Define function 'GetExampleValue', which should already be
//available to the VM
INTEGER FUNCTION GetExampleValue() ATTRIBUTES(EXTERNAL);

//Load external module "windowsui" and lookup the MessageBox macro in it.
MACRO MessageBox(String text, String title) ATTRIBUTES(EXTERNAL "windowsui");
```

### Aggregate functions

An aggregate function is a special function that can be used inside a grouped SELECT expression. It must be of a special format: it must have exactly one parameter, that must either be an array type or type VARIANT.

Aggregate functions must be called with a non-array argument inside the select. For every record in a group, the arguments are collected into an array, and then the aggregate function is invoked with that array.

```
// Define an aggregate function SUM, that returns the integer sum of the array
argument
INTEGER AGGREGATE FUNCTION SUM(INTEGER ARRAY values)
{
  INTEGER total := 0;
  FOREVERY (INTEGER value FROM values)
    total := total + value;
  RETURN total;
}

// Invokes the SUM function with array [1, 2], the result is 3.
SELECT AS INTEGER SUM(x) FROM [[ x := 1 ], [ x := 2 ]];
```

## 3.6 Libraries

A library is a collection of functions, macros and variables, which can be re-used in other libraries and scripts. When a library is loaded, all PUBLIC identifiers it contains become available to the script that loads it. A library is referred to by its name, which consists of a namespace and a path, separated by a double colon, eg:

```
//Make all public variables, functions and macros in
//'money.whlib' available
LOADLIB "wh::money.whlib";
```

If you are writing a HareScript library, the symbols in libraries that you load won't be directly available to scripts that load your library. If you want this, you need to explicitly re-export any symbols you wish to make available to the scripts that load your library, as follows:

```
/*Make all public symbols in money.whlib available to our library,
and make FormatMoney and MoneyToInteger available to all libraries
that load our library */
LOADLIB "wh::money.whlib" EXPORT FormatMoney, MoneyToInteger;
```

An EXPORT can only be used to re-export PUBLIC variables and functions in the library from which you're exporting them. There's no way to gain access to the PRIVATE symbols of a library.

The following example will display the effects of EXPORT in more detail, by showing which variables are available to which library. Although this example shows only variables, the same rules apply to functions and macros:

```
//Library 1: defining a few variables
PUBLIC INTEGER i1;
PUBLIC INTEGER i2;

//Library 2: loading library 1, but only exporting i1
LOADLIB "library 1" EXPORT i1;
PRINT("Multiplication: " || i1 * i2); //okay: both i1 and i2 are available

//Library 3: loading library 2
LOADLIB "library 2";
PRINT("i1: " || i1); //okay: i1 from library 1 is available
//through library 2
PRINT("i2: " || i2); //illegal: i2 is not visible
```

### 3.7 Main code

All code that is outside any function and macro, including the initialisation of any global variables, is considered to be the *main code* of the script. This code is executed whenever the script is loaded as a library, or is run directly. Please note that if a script loads the same library twice, directly or indirectly, its main code is still executed only once.

The following example explains what is considered the 'main code' in every library. All print statements are executed only once, even though library 1 is loaded twice by the final script, library 3:

```
//Library 1 - the code that follows is considered the 'main code'
Print ("This is library 1\n");

//Library 2 - the code that follows is considered the 'main code'
LOADLIB "library 1";
Print("This is library 2\n");

//Library 3
LOADLIB "library 1";
LOADLIB "library 2";

//A function in library 3 - the code it contains is NOT part
//of the 'main code'
INTEGER FUNCTION Square(INTEGER i) { RETURN i * i; }
```

```
//The following code is considered the 'main code'  
Print ("This is library 3\n");
```

### 3.8 Column name lookup rules

Inside SELECT, UPDATE and DELETE expressions special rules apply when looking up the name of an identifier. These rules apply to all the expressions in this statement, with the exception of the FROM clause. In such an expression, an identifier can refer to a variable, a function or a column from one of the source expressions. The following lookup rules apply to identifiers in such expressions:

1. If the identifier is followed by an opening parenthesis, it is considered to refer to a function.
2. If the identifier appears after the cell operator ('.'), it is considered to be a cell inside the previously specified record. This record can also be one of the source expressions specified in a FROM clause.
3. If the identifier is prefixed with a VAR keyword, it is considered to refer to a local or global variable defined earlier.
4. If the identifier is prefixed with a COLUMN keyword, it is considered to refer to a column in one of the source expressions.
5. If no VAR or COLUMN keyword is present, the identifier is considered to refer to a local or global variable, if any variable by that name exists. If no identifier exists, the identifier is considered to refer to a column in one of the source expressions.

When a looked up column may exist in more than one source expression, the compiler generates an error. The compiler assumes that a record array contains all possible column names, so if two or more record arrays are specified as the source expression for a SELECT, all columns will have to be referred to explicitly.

Note that given the list above, the 'VAR' keyword is never required. When neither of the keywords VAR or COLUMN is used, the compiler will still select an existing variable instead of a cell. However, the usage of 'VAR' can clarify the code, and may suppress compiler warnings in some cases.

The following examples detail how the column name lookup works in various cases:

```
TABLE files < INTEGER id >;  
TABLE folders < INTEGER id, STRING name >;  
RECORD ARRAY ral;  
INTEGER id := 5;  
  
//'id' refers to the global variable 'id', NOT to one of the  
// sources being selected  
SELECT id FROM files, folders;  
  
//Two proper ways to select 'id' from table 'files'  
SELECT files.id FROM files;  
SELECT COLUMN id FROM files;  
  
//'test' will be looked up in 'ral', as 'files' won't contain the cell  
SELECT test FROM files, ral;
```

In the following examples, the lookup rules are insufficient to determine which source expression or variable to use, and the compiler will generate an error:

```
TABLE files < INTEGER id >;  
TABLE folders < INTEGER id, STRING name >;  
RECORD ARRAY ral, ra2;  
INTEGER id := 5;  
  
//Ambiguous, because both 'files' and 'folders' offer a column named 'id'
```

```
//The 'column' keyword ensures that variable 'id' is not considered
SELECT COLUMN id FROM files, folders;

//Column 'name' is ambiguous, because the compiler
//considers 'ra1' to contain all
//possible columns
SELECT name FROM folders, ra1;

//All columns are ambiguous, because more than
//one record array is specified
SELECT name, title FROM ra1, ra2;
```

The examples above only discuss SELECT, but the lookup rules also apply to UPDATE and DELETE statement. The only difference is that UPDATE and DELETE do not permit multiple source expressions, so some of the possible ambiguities do not apply there.

## 4. Constants and initialisers

---

Constants and initialisers are used to feed the basic data to the scripts. Every simple value (eg. 4, 2.20371) or string (eg. "Hello, World") is considered to be a constant. More complex data structures, such as records and arrays, can be built using both constant and dynamic data.

### 4.1 String constants and escape sequences

String constants can be used to fill a [string](#) variable, print something, or pass a string parameter to a function or string operator. A string constant can contain an unlimited number of characters. For a string to be printable and usable in most HareScript functions, every character must either be a TAB, one of the 95 printable ASCII characters (character codes 32 to 126), or be part of a UTF-8 encoded Unicode character.

If an external editor is used to edit a HareScript file, and you wish to use any non-ASCII characters, such as 'á' or 'ç', you should make sure that this editor can properly create and edit UTF-8 encoded files. The java-based HareScript editor in WebHare Application Portal and the windows-based HareScript editor in WebHare Lite are always safe to use.

String constants must be enclosed in either single quotes or double quotes. A string constant may not span more than one line.

All occurrences of the enclosing quote character and the backslash character, in the string constant, must be escaped using an escape sequence. The following escape sequences are supported inside HareScript string constants:

<code>\a</code>	audible bell (ASCII code 7)
<code>\b</code>	backspace (ASCII code 8)
<code>\f</code>	formfeed (ASCII code 12)
<code>\n</code>	linefeed (ASCII code 10)
<code>\r</code>	carriage return (ASCII code 13)
<code>\t</code>	horizontal tab (ASCII code 9)
<code>\'</code>	escaped single quote
<code>\"</code>	escaped double quote
<code>\\</code>	escaped backslash
<code>\nnn</code>	octal ascii code
<code>\xnn</code>	hexadecimal ascii code

```
//A string containing four linefeeds
STRING linefeeds := "\n\n\n\n";

//A string containing ASCII characters 4, 12, 20
STRING asciis := '\004\014\024';

//A string containing backslashes, and the quote character
//used to delimit it
STRING escapes := 'Backslash: \\      Single quote: \'      Double quote \' "  ';
```

### UTF-8 encoding

HareScript expects its strings and files to follow the UTF-8 encoding standard. The UTF-8 encoding is a popular method for storing characters from the Unicode character set, because it allows many existing applications (which expect 8-bit characters) to easily support the full Unicode character set. UTF-8 is also a very efficient coding system when most text in a file is in the ASCII character set (such as HTML and HareScript), because the ASCII characters only require 8 bits per character to store.

## 4.2 Numerical constants

Numerical constants, such as '5', '2.20371' and hexadecimal numbers, can be used to fill variables of the various numeric types WebHare supports, or passed as parameters to a function or a numerical operator.

Numerical constants can be of type *integer*, *money* or *float*. For decimal numbers, a constant will always be of the smallest type that is able to represent it. For example, the constant '1000' will always be of type *integer* as '1000' can be stored by every numerical type, and *integer* is the smallest of those types. The constant '1.23456789' will always be of type *float*, since both *integer* and *money* are unable to store a value with 8 decimals.

HareScript considers *integer* to be the smallest numeric type, then *money*, and finally *float*. Each larger type is able to represent all values a smaller type can store, although the largest type, float, may not be able to represent every value exactly.

### Overriding a constant's type

In most cases, it doesn't matter which type HareScript assigns to a numerical constant, as all numerical operators and functions will convert a small type to a larger type when necessary. Eg, when you multiple '1000' with '1.23456789', '1000' is converted to type *float* before the multiplication takes place. A numerical type is never implicitly converted to a smaller type.

However, in some cases you may want to exactly specify the type of a constant, for reasons of clarity, or perhaps because you require a specific type to be used for a record cell. In these cases, you can add an 'i', 'f', or 'm' suffix to a numerical constant to specify its type, as detailed in the following example:

```
// Store the value '5' in a floating point value in a cell.
RECORD r;
INSERT CELL fl := 5f INTO r;

/* Store the integer value '8' into a money variable (the integer to money
conversion is done automatically - the suffix here is optional) */
MONEY m := 8i;

// Illegal: a money value cannot be stored into an integer variable
INTEGER i := 15m;

// Illegal: 1.23456789 is always of float type, and cannot be stored
// as an integer
INTEGER i := 1.23456789;

// Illegal: 1.23i cannot be of integer type
INTEGER i := 1.23i;
```

### Hexadecimal and binary constants

HareScript also permits the use of hexadecimal and binary constants, which may be easier to use when using any of the bit manipulation operators, such as BITAND or BITSHIFT. A hexadecimal constant must be prefixed with '0x', and a binary constant must be prefixed with '0b'. Hexadecimal and binary constants are interpreted as 2-complement signed integers, and can only be of type *integer*.

```
// The decimal value '31' in hex:
INTEGER thirty_one := 0x1F;

// The decimal value '20' in binary:
INTEGER twenty := 0b10100;

// The decimal value '-2' in hex:
INTEGER minus_two := 0xFFFFFEE;

// Illegal: the value '4294967296' is out of range for a hexadecimal value, even
// when immediately assigned to a MONEY variable:
MONEY out_of_range := 0x100000000;
```

### 4.3 Boolean constants

Boolean constants can be used to fill boolean variables, or passed as parameters to functions and operators expecting a boolean value. The only supported boolean constants are the keywords 'TRUE' and 'FALSE'.

Boolean constants are most commonly used to initialise a boolean value to a default, or to pass a simple 'switch' parameter to a function. They can also be used to add clarity to conditional expressions, or to create an 'infinite' loop.

The following examples show some of the most common uses of boolean constants:

```
// Define a boolean variable B and set it to 'true'
BOOLEAN b := TRUE;

// An example of a clarifying but unnecessary use of boolean constants:
BOOLEAN FUNCTION TestCondition() { ... }
IF (TestCondition() = TRUE) ...;

// An example of an 'infinite' loop
WHILE (TRUE)
{ ...
  IF (...) BREAK;
}
```

### 4.4 Record initialisers

Record initialisers simplify the process of building a complex record. Without a record initialiser, you would have to create a *record* first, and then use a series of INSERT CELL statements to add the cells one-by-one.

A record initialiser is contained inside square brackets ('[' and ']') and contains a comma-separated list of assignment statements to build the cells and their contents. It is not possible to build an empty or a non-existing record using a record initialiser.

The following source code shows how to use record initialisers:

```
// A record containing two integer cells, named A and B,
// containing values 2 and 3
RECORD r1 := [ a := 2, b := 3 ];

// A record containing a dynamically generated string, and a money cell
STRING mystr := "String";
RECORD r2 := [ str := "text:" || mystr, euro := 2.20371 ];

// An initialiser can also be used outside an assignment station
PRINT ("Number of cells: " || LENGTH ( [ a := 2, b := 3 ] ) );

// Illegal: multiple cells with the same name appear in the
// initialiser list
RECORD r3 := [ str := "Text", str := "Another Text" ];

// Illegal: record initialisers must define at least one cell
RECORD r4 := [ ];
```

### 4.5 Array initialisers

Array initialisers simplify the process of building an array. Without an array initialiser, you would have to create an array first, and then use a series of INSERT statements to add the elements one-by-one.

An array initialiser is contained inside square brackets ('[' and ']') and contains a comma-separated list of values to build the array. It is not possible to build an empty array using array initialisers (you'll need to use [default values](#) for that). All elements in the array must be convertible to the same type as the first element in the array.

The following source code shows how to use array initialisers:

```

// An array containing all values between 2 and 6
INTEGER ARRAY ar1 := [ 3,4,5 ];

// An array containing a constant and a dynamically generated string
STRING mystr := "String";
STRING ARRAY mystrarray := [ "text:" || mystr, "text2" ];

// An initialiser can also be used outside an assignment station
PRINT ("Number of elements: " || LENGTH ( [ 3,4,5 ] ) );

// An array of records
[ [ i := 2, j := 3 ], [ str := "string" ] ];

// Legal: all array elements can be converted to the first type (money)
[ 1.5, 2 ];

// Illegal: all array elements must be convertible to
// the first type (integer)
[ 1, 2.5 ];

```

## 4.6 Default values

Every HareScript type has a *default* value, which is assigned to a variable if it's [defined](#) without an initializer. The default value for any type can also be obtained by specifying the keyword `DEFAULT` followed by the type name. The following examples show some uses of default values:

```

//The following two statements are identical and both initialize i to zero
INTEGER i;
INTEGER i := DEFAULT INTEGER;

//Insert a cell with 'string array' type, but with no elements so far
INSERT CELL strarray := DEFAULT STRING ARRAY INTO myrecord;

//Check if 29th feburari of 2003 exists - variable isbaddate
//will contain TRUE.
//(makedate returns a default datetime if the parameters are
//out of range)
DATETIME mydate := MakeDate(2003,2,29);
BOOLEAN isbaddate := mydate = DEFAULT DATETIME;

```

For all numeric types, the default value is the zero value. For all other types, the default value is the 'smallest' possible value, ie. it will compare 'less than or equal to' any possible value for that type. The following table shows the default values for all HareScript types:

Type	Default value
Any array type	An empty array
Blob	A blob which is 0 bytes in size
Datetime	The day before 1-1-1
Float	0
Integer	0
Money	0
Record	A non-existing record, with no cells
String	An empty string
Table	Unbound. Using <code>DEFAULT TABLE</code> is not permitted.
Variant	None. Variants are never initialized and using <code>DEFAULT VALUE</code> is not permitted.

## 5. Variable types

---

HareScript offers various types for storage and manipulation of data, such as [Boolean](#), [Float](#) and [String](#). A few types, such as [Record](#), [Blob](#) and the [array types](#), can be used to store nearly infinite amounts of data. Function calls (also function pointers or callbacks) can be stored using a new [function pointer](#) type. HareScript also supports two 'special' types, [Table](#) and [Variant](#), which cannot be used in the language itself, but are used to communicate with external databases and functions.

### 5.1 Integer

Integers are used to store non-fractional values in the range  $-2,147,483,648$  ( $-2^{31}$ ) to  $2,147,483,647$  ( $2^{31}-1$ ).

An integer variable that is not explicitly initialised will contain the value 0 (zero). The following code shows examples of how to define integer variables:

```
// Definition of integer 'example1' with value 13
INTEGER example1 := 13;

// Definition of integer 'example2': a value of 0 is presumed
INTEGER example2;

/* Definition of integer 'example3', using an expression to set its value */
INTEGER example3 := file.id;
```

### 5.2 Boolean

Booleans are used to store '*truth*' values. A boolean variable contains either the value *true* or *false*.

A boolean variable that is not explicitly initialised will contain the value *false*. The following code shows examples of how to define boolean variables:

```
// Definition of boolean 'example1' with value TRUE
BOOLEAN example1 := TRUE;

/* Definition of boolean 'example2', using an expression
to set its value */
BOOLEAN example2 := file.name = "test";

// Definition of boolean 'example3', implicitly initialised to FALSE
BOOLEAN example3;
```

### 5.3 String

A string contains a series of characters. It can be empty (contain no characters at all), and may contain an unlimited number of characters, limited only by the amount of available memory. For efficiency reasons, it's recommended to keep strings to a short length (a few hundred characters) where possible, as performing operations on long strings can be relatively slow.

As strings consist of 8-bit characters, the data in strings should normally be UTF-8 encoded. PRINT and most Encoding functions (eg, EncodeHTML) will then take care of properly encoding the character sequences in the final output format. To use special characters in a string, you need to use [escape sequences](#).

A string variable that is not explicitly initialised will contain an empty string. The following code shows examples of how to define string variables:

```

// Definition of string 'example1' with initial value 'Johnny and Jane'
STRING example1 := "Johnny and Jane";

// Definition of string 'example2': string is presumed to be empty
STRING example2;

/* Definition of string 'example3', using an expression to set
   the value */
STRING example3 := "Name:" || file.name;

```

## 5.4 Blob

A blob type is used to store a reference to a file or other large object on disk, or inside a database table. The name blob is short for 'Binary Large Object'.

Blob variables can be read using blob functions, but can never be modified. To create a new blob from scratch, the stream functions and `MakeBlobFromStream` should be used.

Blobs are sometimes more useful than strings, as a blob usually takes up little memory until it is opened, and a string always consumes at least the memory it needs to store its own data. Another advantage of blobs is that some databases can optimise their handling of blobs in ways that they cannot optimise other objects.

A blob variable that is not explicitly initialised will contain a 0-byte blob. The following code shows examples of how to define blob variables:

```

// Definition of blob 'example1' with the default template
// from the Repository
BLOB example1 := FindFileByFullpath(1, "/templates/default.tpl").data;

// Definition of blob 'example2', containing the text Hello, World
INTEGER blobstr := CreateStream();
PrintTo (blobstr, "Hello, World\n");
BLOB example2 := MakeBlobFromStream(blobstr);

// Definition of blob 'example3', implicitly initialised
// with an empty blob
BLOB example3;

```

## 5.5 Datetime

The Datetime type is used to store date and/or time values. The type supports dates starting from January 1st, year 1, and can store time values with millisecond precision.

A datetime type can also contain an 'invalid' date, which is defined as a special value that is smaller than all other date and time values. This value is usually used to indicate that a date is not known, or an invalid date was received somewhere.

Although a datetime value contains both a date and a time part, in some contexts only a date or time value may be interesting. In those cases, it's customary to use the first day (January 1st, year 1) or midnight (00:00, or 12:00 am) for unused values.

A datetime variable that is not explicitly initialised will contain the Invalid date value (as given by `DEFAULT DATETIME`). The following code shows examples of how to define datetime variables:

```

/* Definition of datetime 'example1' with a value representing the current date
and time on the WebHare server */
DATETIME example1 := GetCurrentDatetime ();

/* Definition of datetime 'example2', with a value representing August
31st 2002, midnight*/
DATETIME example2 := MakeDate (2002, 08, 31);

```

The datetime type and associated functions assume takes leap years into account. The datetime type always calculates according to the Gregorian calendar rules, so calculations with datetime values before 1582 will not be historically accurate. The maximum upper range of the datetime type is somewhere after the 58000th century, which shouldn't be a problem in any practical application.

## 5.6 Money

Money types are used to store fractional values in the range -92,233,720,368,547.75808 to 92,233,720,368,547.75807. Money values support up to 5 decimals, but unlike floating point values, they have no accuracy loss when storing decimal values.

A money variable that is not explicitly initialised will contain the value 0 (zero). The following code shows examples of how to define money variables:

```
// Definition of money variable 'example1' with value 2.20371
MONEY example1 := 2.23071;

// Definition of money variable 'example2': a value of 0 is presumed
MONEY example2;
```

## 5.7 Float

Float types are used to store fractional or large values in the range  $-10^{308}$  to  $10^{308}$ , approximately. Floating-point values are not able to store most decimal values exactly. Instead, floating point values attempt to store the best possible approximation of a decimal value. This is not a limitation of HareScript, but a general 'problem' with floating point values. The approximation may differ slightly between HareScript and WebHare versions.

In most cases where floating-point values are used, this limitation is not a problem. However, floating points should never be used when loss of precision is unacceptable, e.g. when doing financial calculations. The [money type](#) is often better suited for such usage.

A floating-point variable that is not explicitly initialised will contain the value 0 (zero). The following code shows examples of how to define floating-point variables:

```
// Definition of money variable 'example1' with value 2.20371
FLOAT example1 := 2.3532458435253;

// Definition of money variable 'example2': a value of 0 is presumed
FLOAT example2;
```

## 5.8 Array types

An array is not a separate type, but a modifier that can be applied to existing types. It creates a list of elements of the original type, and allows you to access the elements by their number in the list (their *subscript*). The size of an array can be dynamically changed.

All elements in an array are numbered consecutively, beginning at 0 (zero). If an element is deleted from the array, all elements after it are shifted backwards, so that the elements in an array of 'n' elements are always numbered 0 to n-1.

All elements in an array must be of the same type. HareScript does not allow the creation of multidimensional arrays, or an array-of-arrays. However, it's perfectly acceptable to create an array of records, and store arrays in the cells of the individual records.

An array that is not explicitly initialised will contain no elements. The following code shows examples of how to define array variables:

```
// Create an array of four integers, initialising them to 1,2,3 and 4
INTEGER ARRAY intarray := [1,2,3,4];
```

```
// Create an empty array of strings
STRING ARRAY strarray;
```

### Automatic conversion of record arrays

HareScript will automatically convert a record array to a record, if a record array is used in a context where a record is required. This conversion is done by taking the first element of the record array and returning that record. An empty record array is converted to a non-existing record.

This automatic conversion allows the following code fragments to work without type-checking errors:

```
// Get any file matching the WHERE criteria
RECORD myfile := SELECT * FROM files WHERE parent=5 AND name="abc.txt";

// See if the folder with id '6' has any subfolders
IF (RecordExists(SELECT FROM folders WHERE parent=6)) ...;

// Get the ID of user 'sysop'
INTEGER sysop_user_id := (SELECT id FROM users WHERE name="sysop").id;
```

The automatic conversion is only permitted when evaluating a record array - it may not occur on the left side of an assignment operator. For example, the following code is illegal:

```
// Illegal: Try to overwrite the name cell of the first returned record
RECORD ARRAY allfiles := SELECT name FROM files;
allfiles.name := "Trying to set a new name";
```

## 5.9 Record

The Record type is used to store a collection of values, which can be of various types. Each value is uniquely referred to by its *cell name*. A record can store an unlimited number of cells, but a cell name may only be up to 64 characters in length. A record that does not contain any cells can be either *empty* or *non-existing*.

A non-existing record is created by using DEFAULT RECORD, by using a *record* variable that hasn't been initialised with any value, or as a result of a SELECT that did not find any matching records. Some functions, such as FindFile, also return a non-existing record if they were unable to find the requested data.

An empty record is created by the MakeEmptyRecord function, by a SELECT that did not select any columns, or by deleting all cells from a record.

The following example shows how to define a record variable:

```
/* Definition of record 'example1' using a SELECT statement */
RECORD example1 := SELECT * FROM FOLDERS WHERE FOLDERS.NAME = "news";

/* Definition of record 'example2', using a FIND function*/
RECORD example2 := FindFile (1);
```

## 5.10 Table

```
TABLE <table-field-list> <identifier> [ := <expression> ] ';'

<table-field-list> ::= '<' <table-fields> [ ';' KEY <table-field-names> ] '>'
<table-fields> ::= <table-field> [ ',' <table-fields> ]
<table-field> ::= <type-specifier> <column-name> [ AS <column-name> ]
```

```
[ NULL := <constant> ]  
<table-field-names> ::= <table-field-name> [ ',' <table-field-names> ]
```

The Table type is used to refer to a table inside an external database. Variables of this type cannot be passed to HareScript functions, but only to external functions. Inside HareScript expressions and statements, tables can only appear after an INTO or a FROM clause of a SQL statement.

A table variable does not represent an actual table, but merely a binding to a table inside a transaction. The normal way to associate table variables with an external database is to open a transaction, and then pass that transaction id and the table's name to a call to the *BindTransactionToTable()* function.

When defining a table variable, you also need to specify a list of column names and types that the table will contain. You can optionally use 'AS' to give a column a different name in HareScript than the name that is used to refer to the actual tables by the transaction driver.

This list will be used to check statements, to explain the table provider how to convert the database's native types to HareScript types, and to provide a column list for "SELECT \*" statements. For more information on how a table provider will interpret this column list, you will need to refer to the documentation for that table provider.

An example of how a HareScript might connect to the WebHare *files* table is presented below. Note that the column list in this example is not the complete list of columns in the *files* table:

```
LOADLIB "wh::dbase/whdb.whlib";  
  
//Define the structure of the FILES table  
TABLE files < INTEGER id, INTEGER parent, STRING name, STRING title>;  
  
//Connect to the database  
INTEGER webhare_transaction := OpenWHDBTransaction("sysop","secret");  
  
//Bind the transaction we just opened to the 'files' table  
files := BindTransactionToTable(webhare_transaction, "files");
```

## NULL conversions

HareScript does not directly support the SQL 'NULL value', a separate value for a variable of any type, which is different from all other values. When reading NULLs from an external table, the database driver will usually convert NULL values to the default value for a type. For example, a NULL integer value will be converted to '0' in the returned HareScript record arrays, and a NULL string value will be converted to an empty string.

This conversion is usually fine when reading data from an external database, but makes it harder to explicitly insert NULL values into the database, or to distinguish between the real value '0' and the NULL value in a table when both values are valid. To solve this problem, HareScript allows you to explicitly define a substitution value for the NULL value. HareScript will then convert any NULLs it see to the specified value, and will convert the specified value to NULL when it is used in any database query. You should ensure that this substitution value can never occur as actual data in the table.

The following code gives an example on how the NULL conversion can be used to write real NULLs to an external database:

```
//Bind table 'mydata' to some external database  
//(eg, an MS Access dbase via ODBC)  
TABLE mydata<INTEGER ref NULL -1, STRING name> := ...;  
  
//This will insert a NULL into the 'ref' field in the real table  
INSERT INTO mydata(ref, name) VALUES(-1, 'No reference');
```

```
//Selecting the NULL will return '-1' in 'ref'.  
RECORD inserted_data := SELECT * FROM mydata WHERE name = 'No reference';
```

## KEY lists

Key lists can be used to define which column(s) can be considered the primary key for a table. This is intended as an extra aid for some database drivers, which require a primary key in the data to be able to update and delete rows, and cannot accurately tell which fields would be a proper primary key.

None of the available database drivers for HareScript currently support an explicit specification of the primary keys, so using a KEY specification will not have any effect yet.

## 5.11 Variant

The Variant type is not a 'real' HareScript type, as it cannot be used to define a variable. External functions can accept and return expressions of type *variant*, which merely tells the HareScript compiler that the function accepts any type of argument. An example of such a function is *Length*, which accepts many different types, such as a *record array* or a *string*, as its argument.

You may also see the type name *variant* mentioned inside error or warning messages generated by the compiler when it tries to refer to a type, but it doesn't know the exact type of the variable yet.

## 5.12 Function pointers

Function pointers allow you to store a call to a function inside a variable. A function pointer is a very flexible way of re-using and selecting code inside a function. Although a SWITCH or IF statement can often do the job, a function pointer allows you to build code which will also work with new, unforeseen cases.

You can use two different names for the function pointer: either "FUNCTION PTR" or "MACRO PTR". These type names are considered equivalent by the HareScript compiler. All examples here will use the "FUNCTION PTR" type name.

A function pointer value is generated by the PTR keyword, which must be followed by the name of the function, and optionally, any arguments which will be passed to that function. If no arguments list is passed (not even '()'), a function pointer is generated which takes just as many arguments as the original function.

When declaring a function pointer, *argument placeholders* can be used to indicate that the arguments passed in a function pointer call should be passed to that function. An argument placeholder consists of a math symbol (#) followed by the argument number. The leftmost argument is #1.

Instead of using an argument placeholder, you can also specify a value for an argument. This value will then be passed to the function pointer when it is invoked. Specifying a value for an argument (often called *binding*) allows you to build function pointers which take a different number of arguments than the function they refer to.

A function pointer is called by simply referring to the variable (or record cell) as if it were a function. If the function pointer is declared to not accept any arguments, a call to it must still have an empty argument list, just like a normal function would require. A couple of examples of function pointers follow:

```
//A pointer to the Left function, which takes two values and  
//passes these to Left  
FUNCTION PTR MyLeftFunction := PTR Left(#1,#2);  
//This function pointer could be invoked as follows:  
Print("The left 3 characters of abcdef are: "  
  || MyLeftFunction("abcdef",3));  
  
//The same function pointer, now using the default argument list  
FUNCTION PTR MyLeftFunction := PTR Left;
```

```

//This function pointer could be invoked as follows:
Print("The left 3 characters of abcdef are: "
      || MyLeftFunction("abcdef",3));

//A pointer to the Left function, swapping both parameters
FUNCTION PTR MyLeftFunction := PTR Left(#2,#1);
//This function pointer could be invoked as follows:
Print("The left 3 characters of abcdef are: "
      || MyLeftFunction(3,"abcdef"));

//A pointer to the Left function, which always takes the left 3 charcters
FUNCTION PTR MyLeftFunction := PTR Left(#1,3);
//This function pointer could be invoked as follows:
Print("The left 3 characters of abcdef are: " || MyLeftFunction("abcdef"));

//An example of an incorrect pointer: missing one argument to Left
FUNCTION PTR MyLeftFunction := PTR Left(#1);
//An example of an incorrect pointer: no arguments at all,
//but Left requires 2
FUNCTION PTR MyLeftFunction := PTR Left();

```

Default argument values are necessarily disabled by function pointers. For example, the *ToInteger* function takes either two or three arguments. A function pointer to the *ToInteger* function would always require three arguments, unless one of the arguments is bound

## 6. Operators

---

Operators are used to build expressions and perform calculations. HareScript offers the following operators:

Category	Operators
<a href="#">Assignment operator</a>	:=
<a href="#">Merge operators</a>	, CONCAT
<a href="#">Comparison operators</a>	=, <, >, <=, >=, <>, !=
<a href="#">Arithmetic operators</a>	+, -, *, /, %
<a href="#">Logical operators</a>	AND, OR, XOR, NOT
<a href="#">Array subscript operator</a>	[]
<a href="#">Cell operator</a>	.
<a href="#">Conditional operator</a>	?:
<a href="#">IN operator</a>	IN, NOT IN
<a href="#">LIKE operator</a>	LIKE, NOT LIKE
<a href="#">Bit operators</a>	BITAND, BITOR, BITXOR, BITLSHIFT, BITRSHIFT, BITNEG
<a href="#">TYPEID operator</a>	TYPEID

The precedence rules for all operators can be found in [appendix 2](#).

### 6.1 Assignment operator

The assignment operator is used to assign a value to an existing variable, cell or array element. An assignment operator may only appear once in an expression, and has no resulting value.

When assigning a value to a cell, the assignment operator cannot create a new cell or change the type of an existing cell - the assigned value must be convertible to the type of the original value stored inside the cell.

The general syntax of the assignment operator is as follows:

```
// Assign "Hello, World" to (an earlier defined) string variable 's'
s := "Hello, World";

// Modify the contents of cell 'id' of record 'file'
file.id := 7;

// Replace element #3 inside integer array 'intarray' with element #2
intarray[3] := intarray[2];

// Illegal: the assignment operator cannot be used
// inside another expression
IF ( ( i := myvar) = 5) ...;
```

### 6.2 Merge operators

The string merge operator merges two strings, two integer values, or a string and an integer value together and returns a new string. It cannot be used to merge floating point, money or datetime values to a string - those types require the use of a formatting function first.

The string merge operator is probably the most-used operator in HareScript. Its general syntax is as follows:

```
//Assign "Hello, World" to string 'str'
STRING hello := "Hello," || " World";
```

```
//Append the number '5' to an existing string
hello := hello || '5';

//Assign "25" to string 'twentyfive'
STRING twentyfive := 2 || 5;
```

The array merge operator **CONCAT** merges two arrays of the same type together, and returns a new combined array. **CONCAT** does not re-order the elements in the individual arrays, and does not eliminate duplicate elements. It is used as follows:

```
//Returns [ 1, 2, 2, 3, 4]
INTEGER ARRAY j := [1, 2] CONCAT [2, 3, 4];

//Illegal to merge a string array to an integer array
STRING ARRAY s := j CONCAT [ 'abc', ' def' ];

//Illegal: merging a single element to an array:
INTEGER i := 4
INTEGER ARRAY k := [ 1,2,3 ] CONCAT i;

//Legal: first convert the element to a single element array,
//and then merge it
INTEGER ARRAY k := [ 1,2,3 ] CONCAT [ i ];
```

### 6.3 Comparison operators

The comparison operators are used to compare two values of the same type, or two values of a numerical type. Eg, the comparison operators can also compare money values to floating point values, integers to money values, etcetera.

It is not possible to compare two values of a blob, record or table type, or to compare two values of any array type. When comparing boolean values, 'false' is considered smaller than 'true'.

Strings are compared based on the ordering of characters in the [ASCII table](#). For instance, "abcdef" < "pqrst" evaluates to TRUE, because 'a' comes before 'p'. "ABCDE" < "abcde" evaluates to TRUE, because 'A' comes before 'a'. "abcd" < "PQ" evaluates to FALSE, because "P" comes before "a".

A comparison operator always returns a boolean value, according to the following truth table:

```
A = B    TRUE if A equals B
A >= B   TRUE if A is greater than or equals B
A <= B   TRUE if A is less than or equals B
A > B    TRUE if A is greater than B
A < B    TRUE if A is less than B
A <> B   TRUE if A is not equal to B
A != B   TRUE if A is not equal to B
```

The following example code demonstrates the comparison operators:

```
//evaluates to TRUE
BOOLEAN b := 3 >= 1;

//evaluates to FALSE
BOOLEAN b := "Dog" < "Cat";

//evaluates to TRUE
DATETIME today := GetCurrentDatetime();
DATETIME start := MakeDate(1974,08,31);
BOOLEAN b := start != today;
```

## 6.4 Arithmetic operators

The arithmetic operators are used to perform basic arithmetic operations on two values of a numeric type. If the two types used in an arithmetic operation differ, the operator automatically upgrades the smaller of the two types. The resulting type of an arithmetic operation is the same as the largest type involved in the operation.

The addition, subtraction, multiplication and division operators can be applied to all numerical types (Integer, Float and Money). The modulus operator can only be applied to integer values.

If the result of a division requires more precision than is available in the final result type, it is rounded towards zero.

A + B	Addition	returns the sum of A and B
A - B	Subtraction	returns the difference between A and B
A * B	Multiplication	returns the product of A and B
A / B	Division	returns the quotient of A and B, rounded towards zero
A % B	Modulus	returns the remainder of A divided by B

## 6.5 Logical operators

The logical operators perform logical computation on boolean values. The NOT operator is a unary operator, and must be followed by a boolean expression. The AND, OR and XOR operators are binary operators, and must appear in between boolean expressions. All logical operators return a boolean value.

The following truth table applies to the logical operators:

a AND b	logical AND	evaluates to TRUE if both A and B are TRUE
a OR b	logical OR	returns TRUE if either A or B is TRUE
a XOR b	exclusive OR	returns TRUE if either A or B is TRUE, but not both
NOT a	logical NOT	returns FALSE if A is TRUE

The following code gives examples on how the logical operators can and cannot be used:

```
// Stores TRUE in boolean 'B'
BOOLEAN b := FALSE OR TRUE;
// Stores FALSE in boolean 'B'
BOOLEAN b := NOT (TRUE XOR FALSE);
// Illegal: logical operators only work with boolean values
BOOLEAN c := TRUE AND 5;
```

The AND and OR operators are short-circuiting, which means that they won't evaluate their second parameter if the first parameter is sufficient to determine their end value. This permits the following code:

```
/* Safe, as the expression 'myrec.id' won't be evaluated unless
   'RecordExists(myrec)' returns TRUE, so there is no risk of
   'Non-existing record'
   errors */
IF (RecordExists(myrec) AND myrec.id > 2) ...;

/* Unsafe, as both expressions would be evaluated,
   and the second evaluation can
   cause an error if the first evaluation returned FALSE */
BOOLEAN did_record_exist := RecordExists(myrec);
BOOLEAN is_id_greater_than_2 := myrec.id > 2;
IF (did_record_exist AND is_id_greater_than_2) ...;
```

## 6.6 Array subscript operator

```
//Retrieving a value from an array
<array value> '[' <index> ']'

//Updating a value in an array
<array variable> '[' <index> ']' ':=' <new value> ';' ;'
```

The array subscript operator allows direct access to an element inside an array. It can be used to read or to update an element in array.

The 'index' must be equal to or larger than 0 (zero), and smaller than the number of elements in the array. Trying to access a non-existing element causes a run-time error. To create elements, you must either use an [array initialiser](#), or an INSERT statement.

## 6.7 Cell operator

```
//Retrieving a cell from a record
<record value> '.' <cell name>

//Updating a cell in a record
<record value> '.' <cell name> ':=' <new value> ';' ;'
```

The cell subscript operator allows direct access to a cell inside a record. It can be used to read or to update cells in a record.

The specified cell name must already have been created in the record. Trying to access a non-existing cell causes a run-time error. Cells can be created using various statements and expressions, including a record initialiser and the INSERT CELL statement.

## 6.8 Conditional operator

```
<boolean expression> ? <result if true> : <result if false>
```

The conditional operator evaluates its first parameter, and if it evaluates to *true*, it returns its second parameter. Otherwise, it returns its third parameter. The first parameter must evaluate to a boolean value, and the second and third parameter must evaluate to the same type.

The conditional operator is short-circuiting in normal expressions, just like the AND and OR operators. After evaluating the first parameter, it will only evaluate the second or third parameter, but will never evaluate both. Although the effect of the conditional operator is comparable to an IF statement, it can be used in a context where an IF statement is impractical:

```
// Use a different WHERE expression, depending on a pre-set boolean flag
BOOLEAN get_all_files := ...;
SELECT * FROM files WHERE get_all_files ? TRUE : files.parent = parent_id;

// Stores '10' into 'i', because '8 < 5' is false.
INTEGER i := 8 < 5 ? 100 : 10;

// Illegal, the type of the second and third parameter may not differ
10 > 100 ? 15 : "string";
```

## 6.9 IN operator

```
<value> IN <array value>
<value> NOT IN <array value>
```

The 'IN' operator (or 'is element of' operator) is used to check whether a certain value exists in an array. The operator returns a boolean value, which is true if the requested element indeed exists in the specified array. The 'NOT IN' operator operates in the exact opposite manner, returning the boolean value true when the requested element does not exist.

The array must be an array of the same type as the value that is being looked for.

## 6.10 LIKE operator

The 'LIKE' operator offers wildcard pattern matching, and returns true when a given value matches a specified pattern. In the pattern, a '?' is used to signify 'any' character, a '\*' indicates 'any amount of any character', and any other character indicates that this character must appear as-is in the given value. The pattern matching is always performed case-sensitively. The 'NOT LIKE' operator can be used to obtain the logical NOT of the return value of the 'LIKE' operator.

The following code gives some examples of this pattern-matching:

```
// Stores 'true', as the text matches the pattern
BOOLEAN b := "hello.txt" LIKE "*.txt";

// Stores 'true', as every '?' is matched by a character
BOOLEAN b := "1234567" LIKE "12?45?7";

// Stores 'false', as a '?' must match at least one character
BOOLEAN b := "abc?def" LIKE "abcdef";

// Stores 'false', as pattern mathing is done case-sensitively
BOOLEAN b := "ABCDEF*" LIKE "abc*";

// Stores 'false', as the text does match the pattern
BOOLEAN b := "hello.txt" NOT LIKE "*.txt"
```

## 6.11 Bit operators

The bit operators can be used to perform operations at the bit level on integer values. The BITNEG operator is a unary operator, and must be followed by an integer expression. All other bit operators are binary operators, and must appear in between integer expressions. All bit operators return an integer value.

The following table defines the effect of the bit operators:

<b>BITNEG a</b>	Returns the value 'a' with all bits inverted (all 0s become 1s, and vice versa).
<b>a BITOR b</b>	Returns a value in which every bit is set that is set in either 'a' or 'b'.
<b>a BITAND b</b>	Returns a value in which every bit is set that is set in both 'a' and 'b'.
<b>a BITXOR b</b>	Returns a value in which every bit is set that is set in either 'a' or 'b', but not in both.
<b>a BITLSHIFT i</b>	Returns a value with all bits in 'a' shift 'i' positions to the left. The newly inserted bits will all be zero. If 'i' is smaller than 1, nothing happens. If 'i' is larger than 31, 0 (zero) is returned.
<b>a BITRSHIFT i</b>	Returns a value with all bits in 'a' shift 'i' positions to the right. The newly inserted bits will all have the same value as the most significant bit in the original value (a negative value will never become positive). If 'i' is smaller than 1, nothing happens. If 'i' is larger than 31, either -1 or 0 is returned, depending on the original value.

The following code gives examples on how the logical operators can and cannot be used:

```
// Stores -6 in integer 'i'
INTEGER i := BITNEG 5;
```

```
// Stores -2 in integer 'i'
INTEGER i := -8 BITRSHIFT 2;
// Illegal: bit operators only work with integer values
INTEGER i := 7 BITAND 2.5;
```

## 6.12 TYPEID operator

```
TYPEID(<type>)
TYPEID(<expression>)
```

The 'TYPEID' operator is used to obtain the *type number* of a HareScript type or a HareScript value. This function is mostly used to detect the type of a cell inside a record, when you have no other way of knowing the cell's type (eg, after a call to the *UnpackRecord* function).

In most practical applications, this function will not be that useful, as you will usually know which types you stored inside a record. The TYPEID operator is however sometimes useful when debugging HareScript applications, or when writing 'general' conversion functions, such as the record printing functions in library *wh::devsupport.whlib*. Some database drivers use TYPEIDs to communicate the HareScript types used in returned record arrays.

You should not expect the returned integer values to have any special meaning, or to remain constant between different versions of the HareScript compiler. It is not possible to request the type id of a *TABLE* variable.

More information about the above functions and libraries can be found on <http://www.webhare.net/>. A simple example of the TYPEID operator follows:

```
RECORD myfile := FindFileByFullpath(1, "/templates/default.tpl");
IF ( TYPEID(myfile.id) = TYPEID(INTEGER))
  PRINT("Myfile.id contains an integer value!");
```

## 7. Control statements

---

Control statements alter the flow of code execution, and determines whether, and how often, statements are executed. HareScript offers all the control statements you would expect in a programming language.

### 7.1 IF ... ELSE

```
IF ( <condition> ) <statement if true> [ ELSE <statement if false> ]
```

The IF statement is one of the most important features of many languages, including HareScript. It allows for conditional execution of code fragments.

The IF statement takes a boolean expression, and if it evaluates to true, executes the statement that follows it. Optionally, an ELSE clause can be specified to indicate a statement that should be executed if the boolean expression evaluates to false.

A block of statements can be used after the IF or ELSE clause by placing them between curly braces. An example of a few possible uses of IF and ELSE are:

```
IF (i>5)
  PRINT("i is greater than 5");
ELSE
  PRINT("i is less than 5");

IF (i<2)
{
  PRINT("i is less than 2");
  MyMacroToBeCalledIfIIIsLessThan2();
}

IF (i<5)
  PRINT("i is less than 5");
ELSE IF (i>5)
  PRINT("i is greater than 5");
ELSE
  PRINT("i is equal to 5");
```

An IF statement can be nested inside another IF statement. There is no limit to the number of nested IF statements you can use.

### 7.2 FOREVERY

```
FOREVERY ( [type] <value variable> FROM <array> ) <statement>
```

The FOREVERY statement loops through all elements in an array, assigns the current value to a variable, and executes the specified statement for each element. By prefixing the variable name with a type, a local variable is defined for the duration of the FOREVERY loop.

Modifications to the value variable have no effect on the array through which the FOREVERY loop runs. As FOREVERY makes a copy of the array it uses before it starts the loop, any modifications to the array itself will not affect the FOREVERY loop either.

Inside the statement or statement block following the FOREVERY statement, the current element counter (the currently used element) can be retrieved by prefixing the value variable name with a pound sign ('#'). This value starts at 0, and will be incremented at every loop iteration. It will thus always be less than the number of elements in the array.

A block of statements can be used after the FOREVERY statement by placing them between curly braces. The following code demonstrates a few uses for the FOREVERY statement:

```
//Print the names of all files contained in the folder
//with id 'parentfolderid'
RECORD ARRAY subfiles := SELECT name
                          FROM files
                          WHERE parent = parentfolderid;

FOREVERY(RECORD subfile FROM subfiles)
  PRINT(subfile.name || "\n");

//Add '1' to every integer in the integer array
INTEGER ARRAY intarray := [ 1,2,3,4,5,6,7 ];
INTEGER current_int;
FOREVERY(current_int FROM intarray)
{
  intarray[#current_int] := current_int + 1;
}
```

A FOREVERY statement can be nested inside another FOREVERY statement. There is no limit to the number of nested FOREVERY statements you can use. Inside a statement block following a FOREVERY statement, you can use [CONTINUE](#) to stop the current iteration and run the loop for the next element in the array, or use [BREAK](#) to abort the FOREVERY statement entirely.

### 7.3 WHILE

```
WHILE ( <boolean expression> ) <statement>
```

The WHILE statement tells HareScript to execute the nested statement(s) repeatedly, as long as the WHILE expression evaluates to TRUE.

The value of the expression is checked each time at the beginning of the loop. As long as the expression evaluates to TRUE, execution will continue. Even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time HareScript runs the statements in the loop is one iteration).

A block of statements can be used after the WHILE statement by placing them between curly braces. The following code demonstrates the WHILE statement:

```
BOOLEAN finished := FALSE;
//Loop until finished is TRUE, and print
//"Not finished yet" for every iteration
WHILE (NOT finished)
{
  PRINT("Not finished yet!\n");
  IF (Check1())
    finished:=TRUE;
}

//Another way to write the above loop is
WHILE (TRUE)
{
  PRINT('Not finished yet!\n");
  IF (Check1())
    BREAK;
}

/* The above loop always prints "Not finished yet!" at least once. If we don't
   want that, we might also be able to rewrite the loop as following */
WHILE (NOT Check1())
  PRINT("Not finished yet!\n");
```

A WHILE statement can be nested inside another WHILE statement. There is no limit to the number of nested WHILE statements you can use. You can use the [BREAK](#) statement to end the execution of the statement(s), or use the [CONTINUE](#) statement to stop the execution of the current statement(s) and start the next iteration.

## 7.4 FOR

```
FOR ( <definition | assignment> ; <test expression> ; <step expression> )
  statement
```

The FOR statement tells HareScript to execute the nested statement(s) repeatedly, as long as the *test expression* evaluates to TRUE. The *step expression* is executed after every loop iteration.

The value of the expression is checked each time at the beginning of the loop. As long as the expression evaluates to TRUE, execution will continue. Even if this value changes during the execution of the nested statement(s), execution will not stop until the end of the iteration (each time HareScript runs the statements in the loop is one iteration).

A block of statements can be used after the FOR statement by placing them between curly braces. The following code demonstrates the FOR statement:

```
//Print the numbers 1 to 10
FOR (INTEGER i := 1; i <= 10; i := i + 1)
  PRINT (i || "\n");

//Loop through the specified array, checking every other
//element whether its equal
//to '2', and returning the position of that element (it will return '4')
INTEGER ARRAY testdata := [ 1, 2, 3, 3, 2, 1 ];
INTEGER position;
FOR (position := 0; position < Length(testdata); position := position + 2)
{
  IF (testdata [position] = 2)
    BREAK;
}
```

A FOR statement can be nested inside another FOR statement. There is no limitation to the number of nested FOR statements you can use. You can use the [BREAK](#) statement to end the execution of the statement(s) and leave the FOR loop. In this case, the test expression and the step expression are not re-evaluated. You can also use the [CONTINUE](#) statement to stop the execution of the current statement(s) and start the next iteration. If you use CONTINUE, the step expression will be rerun, and the test expression re-evaluated, before starting another iteration of the loop.

## 7.5 RETURN

```
RETURN [value];
```

The RETURN statement is used to both end the current [MACRO or FUNCTION](#), and to provide the return value for a function. The RETURN statement can also be used to abort running the [main code](#) for a library. The RETURN statement is required inside a FUNCTION, and must then specify a return value. The RETURN statement may never be used with a value inside a MACRO or the main code.

The following example demonstrates valid and invalid uses of the RETURN statement:

```
INTEGER FUNCTION Square(INTEGER i)
{
  RETURN i*i; //this is the value that will be returned by a call to Square()
  Print("This statement is never executed");
}
```

```

}
MACRO Test(INTEGER i)
{
  IF (i = 5)
    RETURN 123; //Illegal: a MACRO may never return a value

  RETURN; //This return statement is optional, because a macro automatically
0      //returns when the end of its code is reached
}

```

## 7.6 BREAK

```
BREAK;
```

The **BREAK** statement aborts the current [FOR](#), [FOREVER](#) or [WHILE](#) loop, and resumes execution at the statement that follows the statement or statement block that was part of the last loop.

A **BREAK** statement may not be used outside a **FOR**, **FOREVER** or **WHILE** loop, and will only break out of one such loop at a time. Eg, when two **WHILE** loops are nested, a **BREAK** statement inside the inner loop will return to the outer loop.

For examples on using the **BREAK** statement, please refer to the examples of the loop statements mentioned above.

## 7.7 CONTINUE

```
CONTINUE;
```

The **CONTINUE** statement ends the execution of the current [FOR](#), [FOREVER](#) or [WHILE](#) loop, and resumes with the next element or iteration of that loop. If there were no more elements or iterations to perform, **CONTINUE** ends the current loop.

A **CONTINUE** statement may not be used outside a **FOR**, **FOREVER** or **WHILE** loop, and will only end one such loop at a time. Eg, when two **WHILE** loops are nested, a **CONTINUE** statement inside the inner loop will only start the next iteration of that inner loop, and will not affect the outer loop.

For examples on using the **CONTINUE** statement, please refer to the examples of the loop statements mentioned above.

## 7.8 SWITCH, CASE, DEFAULT

```

SWITCH ( <expression> )
{
  [ CASE <value> [, <value> ...] { <statements> } ]
  [ CASE <value> [, <value> ...] { <statements> } ... ]
  [ DEFAULT { <statements> } ]
}

```

The **SWITCH** statement executes a group of statements depending on the result of an expression. It first evaluates the given expression, and then looks through all the **CASE** statements if it contains a value matching the result of the expression. If one of the **CASE** statement matches, the group of statements following that **CASE** is executed.

The values following a **CASE** statement must be constant values.

If none of the **CASE** values match the result of the expression, and a **DEFAULT** statement is present, the code following that statement is executed. The **DEFAULT** statement, if any, must appear as the last option inside the **SWITCH** statement. Each possible value may occur only once inside a **SWITCH** statement.

The following code examples detail the workings of the SWITCH statement:

```
//Prints 'even' or 'odd' if the integer is in the range 1-3
INTEGER i := AskUserForValue();
SWITCH (i)
{
  CASE 1,3 { print ("i is odd"); }
  CASE 2   { print ("i is even"); }
  DEFAULT { print ("i is out of range"); }
}

//Prints 'digit' if the number is in range 0-9, does nothing otherwise
SWITCH (i)
{
  CASE 0,1,2,3,4,5,6,7,8,9 { print ("i is a digit"); }
}

//Illegal: duplicate case
SWITCH (i)
{
  CASE 0,1,2 { print ("i <= 2"); }
  CASE 2,3,4 { print ("i >= 2"); }
}
```

## 8. Source code commenting

---

Creating and maintaining documentation for your libraries is a difficult and time consuming task, especially if development is still happening and changes are frequent. HareScript comes with built-in support for generating source code documentation by specifying short descriptions of functions and their parameters in your source code. You can then extract this documentation using the [HareScript processing library](#) (wh::filetypes/archiving.whlib).

To allow the processor to automatically generate documentation for your files, you need to create special comments for it to pickup. The processor picks up any comment which starts with `/**` or `///` for documentation generation (that is, an extra `'` or `/`, depending on whether the comment is a block comment or single line comment).

To fully document your file, one comment should be inserted at the top of every file (after the WebHare open tag `<?wh`, but before any `LOADLIBS`), and should document the name and the purpose of the library. Next, a comment should appear in front of every public function, macro and global variable you wish to document.

### 8.1 Extractable comments

An extractable comment consists of the following fields. A field, and any optional parameters, must be followed by the text that will be assigned to this field.

The comment parser considers all fields optional, as how they will be used will depend on your documentation template. The following table lists the supported fields and their most likely usage:

<code>@short</code>	A short title
<code>@long</code>	A longer description
<code>@example</code>	Example of how to use a function or macro
<code>@author</code>	Author of a library or function
<code>@copyright</code>	Copyright statement
<code>@version</code>	Library or function version
<code>@deprecated</code>	Reason for a function's deprecation
<code>@return</code>	Description of the return value
<code>@param &lt;paramname&gt;</code>	Description of a parameter's contents
<code>@cell return.&lt;cell&gt;</code>	Description of a cell contained in the return value
<code>@cell &lt;param&gt;.&lt;cell&gt;</code>	Description of a cell contained in a paramter
<code>@see</code>	A list of functions that are related to this function

You can also use text markup inside your comments. DocGen does not require or support any markup itself, so whatever you define will depend on your documentation template. The following table lists the fields supported by our example templates:

<code>@bold</code>	Make the next word bold
<code>@italic</code>	Make the next word italic
<code>@link &lt;location&gt;</code>	Make the next word a hyperlink

These fields can also be applied to more than one word by wrapping the field inside curly braces, eg: `@{link http://www.webhare.net/ WebHare information website}`

### 8.2 Comment examples

You can find a lot of comment examples in the HareScript source code itself, which we use as the base for the function documentation on the [WebHare support site](http://www.webhare.net/) (<http://www.webhare.net/>). For example, we start every library with a comment similar to the following:

```
<?wh
/** @short File and disk I/O
```

```

@long Functions to create blobs and to manage files and
    directories on the filesystem hosting WebHare
@copyright B-Lex Information Technologies 1999-2005.
    All rights reserved.
@author B-Lex Information Technologies */
LOADLIB ".....";

```

Functions can also be documented. The comment for a function should appear right before the function itself:

```

/** @short Read the contents of a directory on disk
    @long Reads a directory, optionally filtering the contents
        with a mask. The UNIX globbing rules will apply when
        using this mask (eg, "*.*)" only matches files with an
        extension, not all files as it would do on DOS/NT).
        This function also returns the "." and ".." directory
        entries, if they exist.
    @param directory Path to directory to read
    @param mask Mask of files to read (* to read all files)
    @return A record array of found files and directories
    @cell return.name Name of the directory entry
    @cell return.type Type of the directory entry
        (0=file, 1=directory, 2=link)
    @cell return.modified Last modification time
    @cell return.size Size of the file, in bytes
    @cell return.unixpermissions Unix file permissions. 0 on Windows */

PUBLIC RECORD ARRAY FUNCTION ReadDiskDirectory(STRING directory, STRING mask)

```

## 9. SQL statements

---

In HareScript, manipulation of tables in an external database, records and (record) arrays is mostly done through four statements, SELECT, INSERT, UPDATE and DELETE. These statements try to follow the syntax of their SQL counterparts as close as possible, but some limitations have been imposed to make the statements work well in the HareScript environment.

On the other hand, you can use the full HareScript syntax inside all of these statements. This allows you to call normal HareScript functions and use the built-in HareScript operators anywhere inside the SQL syntax.

### 9.1 SELECT

```
SELECT <select-type> FROM <sources>
      [ WHERE <clause> ]
      [ GROUP BY <group-by-list>
        [ HAVING <expression> ] ]
      [ ORDER BY <orderings> ]
      [ LIMIT <select limit> ]
```

The SELECT expression is used to retrieve data from database tables, and to filter or rearrange information in record arrays. This expression can combine a lot of complex operations such as column creations, column renaming, sorting, filtering, and joining data from various data sources, inside a single operation.

The HareScript SELECT expression is similar to the SQL SELECT statement. It does not yet offer all features of a SQL select. It does however extend a SQL select in various ways, such as allowing to select the results of a function, reprocessing already selected data, or joining tables together from different external databases.

The different components of the SELECT expression will be discussed separately, with examples showing how to use each part.

#### Columns

```
<select-type> ::= [ DISTINCT ] <columns>
                | AS <type> [ DISTINCT ] <expression>
<columns> ::= <select-column> [, <columns> ]
<select-column> ::= <name> := <expression>
                  | <expression> [ 'AS' <column-name> ]
                  | [ <source-name> '.' ] '*'
```

The AS part of a SELECT determines what type the returned data will have. If that type is not specified, the default is *record array*.

When the AS part is present, the returned data is generated from the specified expression or column. With an array return type, an array with the expressions generated from all matching records is given back. With non-array types, the calculated value of the expression for the first matching record (obeying ordering requirements) is returned. If the select had no results, the default value of the return-type is given back.

When no 'AS <type>' clause is specified, the column list determines which cells will appear in the record array returned by the expression. The column list can be left empty, in which case only empty records will appear in the returned array - this is only useful for counting the number of records matching certain criteria.

The specification of the record to create is given by specifying a list of source records and cells generated by an expression.

- A source record (the asterisk (\*) syntax) selects all cells from the specified source. The source-name may be omitted if only one source is present. The cells have are names according to their name within the source, and cannot be renamed.
- Single cell expressions must be explicitly named by using an SQL-style "AS" expression, or by using an assignment expression. The only exception is when the expression refers to a cell within a source expression (with the syntax *column-name* or the syntax *source-name '.' column-name*). In that case the column-name is used as name for that expression.

The names of the cells in the select list must all be unique within that list. An error is generated when the same name is encountered twice (if possible compile-time, otherwise run-time). Single cell expressions can be renamed by using an SQL-style "AS" expression, or by using an assignment expression.

Inside a select-column expression, the [column name lookup rules](#) apply - any name that does not refer to a variable is considered to be a cell from one the source expressions.

If the DISTINCT keyword is present, all the duplicates in the generated output values are removed. When used, all the selected values must evaluate to a type that can be supplied to a [comparison operator](#). This implies that for example integers, strings and booleans are valid types, but blobs and arrays aren't.

```
// Select column 'a' to an integer array, returns [1, 1, 2]
SELECT AS INTEGER ARRAY a FROM [[a := 1], [a := 1], [a := 2]];

// Select distinct values of column 'a' to an integer array, returns [1, 2]
SELECT AS INTEGER ARRAY DISTINCT a FROM [[a := 1], [a := 1], [a := 2]];

// Selects column 'a' to a record array, returns [[ a := 1]]
SELECT a FROM [[a:=1, b:=1], [a:=2, b:=2]] WHERE b = 1;

// Select source record from source 'a', returns [[a := 1, b := 3]]
SELECT a.* FROM [[a := 1, b := 3]] AS a, [[c := 1]] AS b

// Error, cell 'b' in source record conflicts with cell 'b' from expression
SELECT a.*, 5 AS b FROM [[a := 1, b := 3]] AS a
```

## The FROM clause

```
<sources> ::= <select-source> [, <sources> ]
<select-source> ::= <expression | table> [ AS <source-name> ]
```

The *from* clause specifies one or more table names or record array expressions, from which records should be selected. You can mix tables from different external databases, or tables and record arrays inside the *from* clause, the HareScript engine will work out how to combine the results from various data source. The table names and record arrays are referred to as 'source expressions'. Commas are required to separate multiple source expressions.

A source expression can have a name. If present, this name must be unique in the FROM-clause. The name can be used to select columns from this particular source using the cell-get operator ('.'). For tables, this name is the table name. For record array expressions, a name is only given when the expression is a simple variable, and then is the the same as the name of that variable. Otherwise, the source remains unnamed. The name of a source can be overridden using the 'AS' keyword.

The SELECT expression, when executed, passes only a single record to its WHERE clause, ORDER BY clauses, etcetera. When you're selecting from multiple source expressions, the *Cartesian product* of the records in these sources is passed to the other clauses. The following example explains the generation of this Cartesian product:

```

RECORD ARRAY source1 := [ [ id := 2 ], [ id := 5 ] ];
RECORD ARRAY source2 := [ [ name := "a" ], [ name := "b" ]
                        , [ name := "c" ] ];

RECORD ARRAY results := SELECT source1.id, source2.name
                        FROM source1, source2;

/* Results will now contain six records, and the following code
   will print: "2a,2b,2c,5a,5b,5c," */
FOREVERY (RECORD result FROM results)
  Print(result.id || result.name || ",");

```

The WHERE clause, if any, is applied to the records generated by the Cartesian product. This can be used to create a *join* between tables, by requiring two cells in the different sources to have the same value. The example below uses such a join to combine the name of a file with the name of its parent folder:

```

// Build two arrays, representing two folders (folder1 and folder2), in which
// folder1 contains file1, and folder2 contains file2.
RECORD ARRAY folders := [ [ id := 1, name := "folder1" ],
                        [ id := 2, name := "folder2" ] ];
RECORD ARRAY files := [ [ parent := 1, name := "file1" ],
                       [ parent := 2, name := "file2" ] ];

// Join files and folders, resolving the duplicate cell name using 'AS'
RECORD ARRAY results := SELECT folders.name AS foldername,
                          files.name AS filename
                        FROM folders, files
                        WHERE folders.id = files.parent;

// Results will now contain two records, and the following code will print
// 'file1 is in folder1' and 'file2 is in folder2'
FOREVERY (RECORD result FROM results)
  PRINT(result.filename || ' is in ' || result.foldername || '\n');

```

If a record array expression is specified as a source, the position of the current record within that record array can be obtained by prefixing the name of the source with a pound sign ('#').

### The WHERE clause

The optional WHERE clause can be used to specify a boolean expression which every source record must match to be selected. A SELECT may only contain one WHERE expression, but multiple conditions can be combined using the AND operator.

Inside a select-column expression, the [column name lookup rules](#) apply - any name that does not refer to a variable is considered to be a cell from one of the source expressions. The usual [short-circuiting rules](#) do apply inside the WHERE clause of a SELECT.

A where clause can contain simple comparison operators, but is also permitted to contain complex operations or function calls. However, it's recommended to specify simple operations as simple as possible, and to 'AND' various operations together. This allows the HareScript compiler to better optimise your SELECT expressions.

The following examples show some typical uses of the WHERE clause:

```

//Select all files in folder '15'
SELECT * FROM files WHERE parent=15;

//Select all subfolders of folder '8' whose name starts with 'data'
SELECT * FROM folders WHERE parent=8 AND name LIKE 'data*';

//Select all users who have more privileges than basic users
SELECT * FROM users WHERE access > 0;

```

To ensure high performance, as much as possible of the WHERE clause is sent to the database. Typically, the only parts of the clause that can be optimized in this way are:

1. Joined together with the AND-operator.
2. Have one of the following forms:
  - column comparison-operator column
  - column comparison-operator value
  - value comparison-operator column
  - column LIKE value
  - [ NOT ] column (only for boolean columns)
  - [ NOT ] constant value

Furthermore, expressions within the conditional operator can also be optimized, but only when its first expression (the condition) does not contain any reference to the selected tables or record arrays.

If the compiler finds an optimizable expression after an unoptimizable part, it will issue a warning. If possible, the optimizable part should be placed before the unoptimizable. Failing to do this could result in a performance hit.

The following examples illustrate what kind of expressions are optimizable:

```
//All the parts can be optimized.
SELECT * FROM files, folders WHERE files.id=15 AND files.parent = folders.parent

//The parts are not joined together with AND.
SELECT * FROM files WHERE (files.parent = 0 OR files.id = 16)

//All the parts within the conditional operator can be optimized
INTEGER boolval := TRUE;
SELECT * FROM users WHERE (boolval ? users.id = 5 : users.id = 4)

//Second part will not be optimized, because the first part can't be optimized
SELECT * FROM users WHERE IsCool(users.name) AND users.id < 100;
```

## The GROUP BY clause

```
<group-by-list> ::= <group-by> [ , <group-by-list> ]
<group-by> ::= [ <source-name> '.' ] <column-name>
```

When the GROUP BY clause is present, or when an aggregation function (MIN, MAX, COUNT, etc.) is used within the SELECT or ORDER BY clauses, the SELECT will operate in grouping mode.

In this mode, all records are organized into groups. All records that have the same value for the columns specified in the *group-by-list* (the grouped columns) are put into a separate group. If the GROUP BY clause is not present at all, all the records are put into one single group. So, inside a group, all records have the same values for the grouped columns.

The second effect is that the ORDER BY and the selected columns (or expressions) switch from operating on single records to operating on groups (note: the WHERE clause doesn't switch, and will continue to operate on records).

Because the grouped columns always have the same value for every record in a group, those columns can be used directly inside the ORDER BY clause, HAVING clause and the selected columns.

The other, non-grouped columns can only be used inside the arguments of an aggregation function (because they can vary from record to record). The argument to that function is evaluated for every record in the group, and the resulting values are collected in an array. This array is used as argument to the aggregation function.

The columns passed to a GROUP BY must evaluate to a type that can be supplied to a [comparison operator](#). This implies that integers, strings and booleans are valid types to use, but blobs and arrays aren't.

```
//Select the units with users and the number of users inside that unit
SELECT unit, COUNT(*) FROM users GROUP BY unit;

//Error, the id column cannot be used because it is not grouped.
SELECT unit, id FROM users GROUP BY unit;

//Select per unit the highest access any user has
SELECT unit, MAX(access) FROM users GROUP BY unit;

//Select the maximum access of all users (return as integer)
SELECT AS INTEGER MAX(access) FROM users
```

### The HAVING clause

The HAVING clause has the same function as the WHERE clause; the difference is that the WHERE clause operates before the records are grouped, while the HAVING clause operates after grouping has been performed.

Thus, the WHERE clause can be used to filter individual records, and the HAVING clause can be used to filter groups.

This means that aggregate functions can be used within the HAVING clause, but not in the WHERE clause.

```
//Select the units with users and the number of users inside that unit
SELECT unit, COUNT(*) FROM users GROUP BY unit;

//Select the units with users and the number of users inside that unit (exclude
unit 1)
SELECT unit, COUNT(*) FROM users GROUP BY unit HAVING unit != 1;

//Select the units with users and the number of users inside that unit (don't count
the sysop with id 1)
SELECT unit, COUNT(*) FROM users WHERE id != 1 GROUP BY unit;

//Select the units with more than 4 users
SELECT unit, COUNT(*) FROM users GROUP BY unit HAVING COUNT(*) > 4;
```

### The ORDER BY clause

```
<orderings> ::= <ordering> [ , <orderings> ]
<ordering> ::= <ordering clause> [ ASC | DESC ]
```

The optional ORDER BY clause can be used to specify a sorting order for the output records. Without an ORDER BY clause, the ordering of the final results of a SELECT will depend on the source expressions. If the source data only consists of one or more record arrays, their ordering will be retained. However, if a table appears in the list of source expressions, the ordering of records is indeterminate.

The expressions passed to an ORDER BY must evaluate to a type that can be supplied to a [comparison operator](#). This implies that integers, strings and booleans are valid types to use for sorting, but blobs and arrays aren't.

An ordering clause can be followed by the keyword ASC or DESC, to indicate an ascending or descending sorting order. If neither keyword is specified, an ascending sorting order is used.

Multiple sorting expressions can be passed to an ORDER BY expression by separating them with commas. The sorting order will be determined by evaluating the sorting expressions from left-to-right, ie. the second and third ordering criteria will only apply if two records would be given an equal position based on the first criterium.

Inside a sorting expression, the [column name lookup rules](#) apply - any name that does not refer to a variable is considered to be a cell from one of the source expressions. The following code gives a few examples on how ORDER BY can be used:

```
//Get all files from folder 27, sorting them by name, case-insensitively
SELECT * FROM files WHERE parent=27 ORDER BY ToUppercase(name);

//Sort an existing record array by name or title,
//depending on an earlier choice
RECORD ARRAY results;
BOOLEAN sort_by_name := AskUserHowToSort();
results := SELECT * FROM results ORDER BY (sort_by_name ? name : title);

//Get the file which would be at the 'end' of a list of files
//sorted by name;
RECORD lastfile := SELECT * FROM files ORDER BY name DESC LIMIT 1;
```

### The LIMIT clause

The optional LIMIT clause can be used to specify a maximum number of results to return. The LIMIT clause will be applied to the results after processing any WHERE and ORDER BY clauses. If a LIMIT smaller than or equal to zero is specified, no results are returned.

## 9.2 INSERT

The INSERT statement has four different, related forms. It can be used to manipulate tables, (record) arrays, and to add cells to an existing record. The four different forms will be discussed separately in the following sections.

### Table inserts

```
INSERT INTO <table> ( <cells> ) VALUES ( <cell values> );
<cells> ::= <cell name> [ ,<cells> ]
<cell values> ::= <value> [ ,<cell values> ]
```

The table insert is used to add a single record to a database table. The *table* variable must refer to a table that has been bound using the *BindTransactionToTable()* function.

The INSERT statement requires a comma-separated list of cell names to set, and a comma-separated list of expressions to fill these cells. The number of cell names must match the number of cell values.

It is not possible to specify a position at which the record should be inserted into the table. The table provider, and the external database to which the provider connects, may impose additional restrictions on the values being inserted. If an error is reported by the database, most table providers won't report it immediately, but postpone it until the transaction causing the error is committed.

A simple example of inserting a file into the WebHare database follows. This example assumes the table *files* has already been properly bound:

```
//Insert a file named 'test.txt' into folder 5, owned by the sysop (user 1)
INSERT INTO files(parent, owner, name) VALUES(5, 1, "test.txt");
```

### Array element inserts

```
INSERT <element> INTO <array> AT ( <position> | END );
```

The array insert statement is used to insert a single element into an array. The element must be of the same type of the elements already in the array, or be of a numeric type that can be upgraded to that array's type. The array insert statement can be applied to all array types, including *record arrays*, but cannot be used for tables.

The statement must also specify the position at which the new element must be inserted. The new element will be placed at this position. The already existing element, and all elements following it, will be shifted one position. If the new element's position is specified as 'AT END', the element will be appended to the end of the array.

The following code gives a few examples on how to use the array insertion statement:

```
//Insert '5' into the middle of the array, making the array [ 1,2,5,2,1 ]
INTEGER ARRAY ids := [ 1,2,2,1 ];
INSERT 5 INTO ids AT length(ids)/2;

//Append '9' to the end of the array
INSERT 9 INTO ids AT END;
```

### Record inserts

```
INSERT INTO <record array> ( <cells> ) VALUES ( <cell values> ) <position>;
<cells> ::= <cell name> [ ,<cells> ]
<cell values> ::= <value> [ ,<cell values> ]
<position> ::= AT ( position | END )
```

The record insert statement is used to combine the creation of a record and its insertion into an array into a single statement. The syntax of the record insertion statement is similar to that of the table insertion statement, with the addition of a position indicator, as shown below:

```
RECORD ARRAY myarray;
INSERT INTO myarray(id, name) VALUES(3,"test.txt") AT END;
```

### Record cell inserts

```
INSERT CELL <cell name> := <cell value> INTO <record>;
```

The cell insert statement adds a cell to a record. This record can be either a normal record or a non-existing record. If a non-existing record is passed to INSERT CELL, it is automatically converted to a normal record.

The cell insert statement cannot be used to overwrite an existing cell. You need to use a cell assignment if you want to replace an existing cell, or use [cell deletion](#) statement first.

A few examples of the cell insertion statement follow:

```
RECORD filerec := [ id := 2, name := "test.txt" ];

//Add the cell 'title' to testfile
INSERT CELL title := "Title of " || filerec.name INTO filerec;

//Illegal: trying to replace an existing cell
INSERT CELL name := "anothertest.txt" INTO filerec;
```

## 9.3 UPDATE

```
UPDATE ( <table> | <record array> ) SET <set expressions> [ WHERE <clause> ];
<set expressions> ::= <set expression> [ , <set expressions> ]
<set expression> ::= <column> := <new value>
```

The UPDATE statement is used to look for all records inside a record array or table matching a specified criterium, and then updating the specified cells inside this record. The list of columns to update is specified as a comma-separated list of assignments.

An optional WHERE clause can be specified to limit which records will be updated. The statement will then evaluate the WHERE clause for these expressions before updating them, and only update them if the WHERE clause evaluated to TRUE.

Inside the WHERE clause and the list of assignment expressions, the [column name lookup rules](#) apply - any name that does not refer to a variable is considered to be a cell in the table or record array being updated.

When updating a table, the table provider, and the external database to which the provider connects, may impose additional restrictions on the values being updated. If an error is reported by the database, most table providers won't report it immediately, but postpone it until the transaction causing the error is committed.

The following code gives a few examples on how UPDATE can be used:

```
//Move all files in folder 5 to folder 7
UPDATE files SET parent := 7 WHERE parent = 5;

//Add a '.bak' extension to all files in the files table
UPDATE files SET name := name || ".bak";

//Negate the 'id' field of all records whose name field starts with an 'a'
UPDATE files SET id := - id WHERE name LIKE "a*";
```

## 9.4 DELETE

The DELETE statement, like the [INSERT](#) statement, has four different, related forms. It can be used to manipulate tables, (record) arrays, and to remove cells from an existing record. The four different forms will be discussed separately in the following sections.

### Table deletes

```
DELETE FROM <table> [ WHERE <clause> ];
```

The table delete statement is used to delete all records in the table matching a specified criterium. The optional WHERE clause can be specified to limit which records will be deleted. The statement will only delete records for which the WHERE clause evaluated to TRUE.

When deleting records from a table, the table provider, and the external database to which the provider connects, may impose additional restrictions on the records being deleted. If an error is reported by the database, most table providers won't report it immediately, but postpone it until the transaction causing the error is committed.

If the WHERE clause is not specified, all records from the specified table will be deleted. A few examples of the DELETE statement are given below:

```
//Delete all files in folder 8
DELETE FROM files WHERE parent=8;

//Delete all users from the user table
DELETE FROM users;
```

### Array element deletes

```
DELETE FROM <array> ( AT <position> | ALL );
```

The array delete statement is used to remove elements from an array. The array delete statement can be applied to all array types, including *record arrays*, but cannot be used for tables.

A single element can be deleted by using the AT keyword, followed by the position of the element to delete. The element at the specified position will be removed, and the elements following it will be moved one position backwards to fill the remaining space.

You can also delete all elements an array contains by using the 'ALL' keyword. The following code gives some examples for using the array delete statement.

```
//Remove '5' from the middle of the array, making the array [ 1,2,2,1 ]
INTEGER ARRAY ids := [ 1,2,5,2,1 ];
DELETE FROM ids AT length(ids)/2;

//Empty the array 'ids'
DELETE FROM ids ALL;
```

### Record deletes

```
DELETE FROM <record array> [ WHERE <clause> ];
```

The record array delete statement is used to delete all records in an array matching the specified criterium. The optional WHERE clause can be specified to limit which records will be deleted. The statement will only delete records for which the WHERE clause evaluated to TRUE.

If the WHERE clause is not specified, all records from the specified array will be deleted. A few examples of the DELETE statement are given below:

```
RECORD ARRAY ra := [ [ id := 4, name := "test.txt" ]
                    , [ id := 7, name := "hi" ] ];

//Delete the file 'test.txt' from the record array above
DELETE FROM ra WHERE id=4;

//Empty the record array
DELETE FROM ra;
```

### Cell deletes

```
DELETE CELL <cellname> FROM <record>;
```

The cell delete statement removes the cell with the specified name from a record. If no cell with the specified name exists, the delete command is ignored.

A few examples of the cell deletion statement follow:

```
RECORD filerec := [ id := 2, name := "test.txt" ];

//Remove the 'name' cell from filerec
DELETE CELL name FROM filerec;

//Ignored: removing non-existing cell title from filerec
DELETE CELL title FROM filerec;
```

## Appendixes

---

This manual contains the following appendixes:

- Appendix 1: [Listing of reserved words](#)
- Appendix 2: [Operator precedence](#)
- Appendix 3: [Deprecated HareScript features](#)
- Appendix 4: [ASCII table](#)

## Appendix 1: Listing of reserved words

The following keywords are reserved, and may not be used as a function, macro, variable or cell name. Some already have a defined use in the HareScript language, others are reserved for future extensions:

Aggregate	DateTime	Index	Record
All	Default	Inner	Ref
And	DefaultsTo	Insert	Return
Array	Delete	Integer	Select
As	DeleteCell	Intersect	Set
Asc	Desc	Into	SetCell
At	Distinct	Join	String
Attributes	Else	Key	Switch
BitAnd	End	Like	Table
BitLShift	Except	Limit	True
BitNeg	Export	Loadlib	Typed
BitOr	False	Macro	Union
BitRShift	FixedPoint	Money	Unique
BitXor	Float	New	Update
Blob	For	Not	Using
Boolean	Forevery	Null	Values
Break	From	Offset	Var
By	Full	Only	Variant
Case	Function	Or	Where
Cell	Goto	Order	While
Column	Group	Outer	Xor
Concat	Having	Private	
Continue	If	Ptr	
Cross	In	Public	

## Appendix 2: Operator precedence

The following table lists the precedence of operators. Operators have the same precedence as other operators in their group, and higher precedence than operators in lower groups. Operators in the same group are evaluated left-to-right.

TYPEID	Get type of
[ ]	Array subscript (list element)
.	Record cell (record element)
+	Unary positive
-	Unary negative
NOT	Logical NOT
BITNEG	Binary negation
*	Multiplication
/	Division
%	Modulus
+	Addition
-	Subtraction
BITLSHIFT	Binary left shift
BITRSHIFT	Binary right shift
BITAND	Binary AND
BITOR	Binary OR
BITXOR	Binary XOR
	String merge
CONCAT	Array merge
=	Is equal to
>=	Is greater than or equal to
<=	Is less than or equal to
>	Is greater than
<	Is less than
<>	Is not equal to
!=	Is not equal to
LIKE	Matches wildcard pattern
IN	Is contained in
AND	Logical AND
OR	Logical OR
XOR	Exclusive OR
SELECT	Array or table select statement
:=	Assignment

### Appendix 3: Deprecated HareScript features

This appendix lists HareScript features that are deprecated in the current compiler, and how they should be rewritten to suit this and future versions of the compiler.

#### Legacy LoadLibs

The WebHare publishing module in versions before WebHare CMS v2.10, had a quite different syntax for referring to libraries. The following table helps you to convert legacy-style LOADLIBs:

Pre-v2.10 LoadLib	v2.10 LoadLib
system/datetime.whlib	wh::datetime.whlib
~<site>/<path>/<libname>	site::<site>/<path>/<libname>
<module>/<libname>	module::<module>/<libname>
/<path>/libname	currentsite::<path>/<libname>

#### Optional location for INSERT

In previous HareScript versions, "AT 0" at the end of an array INSERT statement was optional - if no "AT" clause was found, insertion at the beginning was assumed. HareScript now always requires an "AT" clause for array INSERTs.

#### Array DELETES

The syntax used to remove array elements in previous HareScript versions would be ambiguous if it were extended to support deletion of array elements inside records. Also, it was felt that the syntax of an array DELETE differed too much from a similar array INSERT statement. The following code fragment shows the change:

```
//Original, deprecated delete element syntax:
DELETE <array name> [ <element number> ];
//New, recommended delete element syntax:
DELETE FROM <array variable> AT <element number>;

//Original, deprecated delete all elements syntax:
DELETE <array name>;
//New, recommended delete all elements syntax:
DELETE FROM <array name> ALL;
```

#### Inserting cells into a record

Previous HareScript versions offered the SetCell keyword to both create and update cells inside a record. The SetCell syntax was considered confusing, because it looked like a function call but it wasn't in reality, as a normal function can't change the original variable that is passed to it as a parameter. SetCell also allowed users to accidentally change the type of a cell. The following code fragment shows the change:

```
//Original, deprecated SETCELL syntax
SETCELL (<record variable>, "<cell name>", <value>);

//New, recommended cell creation syntax
INSERT CELL <cellname> := <value> INTO <record variable>;

//New, recommended cell modification syntax
<record variable>.<cellname> := <value>;
```

#### Removing cells from a record

Previous HareScript versions offered the DeleteCell keyword to remove from a record. The DeleteCell syntax was considered confusing for much of the same reasons SetCell was. The following code fragment shows the change:

```
//Original, deprecated DELETECELL syntax
DELETECELL (<record variable>, "<cell name>");

//New, recommended cell deletion syntax
DELETE CELL <cellname> FROM <record variable>;
```

## Appendix 4: ASCII table

HareScript uses the [Unicode](#) standard for character codes. The first 128 characters in the Unicode standard are the same as the ASCII character set.

The ASCII character set consists of printable characters (the character codes in range 32 to 126) and control characters (codes 0 to 31, and 127).

### Control characters

This table lists the ASCII control characters, including their name, description, and octal values. The octal values are required to include the ASCII control characters inside a string constant. For example, the 'bell' control character can be embedded inside a string by using the escape code `'\007'`.

Please note that the description associated with each ASCII character does *not* indicate how HareScript handles these characters when printing, but the characters may have the specified effect when sent to some devices, such as printers and terminals.

Decimal	Octal	Name	Description
000	000	NUL	(Null character)
001	001	SOH	(Start of Header)
002	002	STX	(Start of Text)
003	003	ETX	(End of Text)
004	004	EOT	(End of Transmission)
005	005	ENQ	(Enquiry)
006	006	ACK	(Acknowledgment)
007	007	BEL	(Bell)
008	010	BS	(Backspace)
009	011	HT	(Horizontal Tab)
010	012	LF	(Line Feed)
011	013	VT	(Vertical Tab)
012	014	FF	(Form Feed)
013	015	CR	(Carriage Return)
014	016	SO	(Shift Out)
015	017	SI	(Shift In)
016	020	DLE	(Data Link Escape)
017	021	DC1	(XON) (Device Control 1)
018	022	DC2	(Device Control 2)
019	023	DC3	(XOFF)(Device Control 3)
020	024	DC4	(Device Control 4)
021	025	NAK	(Negative Acknowledgement)
022	026	SYN	(Synchronous Idle)
023	027	ETB	(End of Trans. Block)
024	030	CAN	(Cancel)
025	031	EM	(End of Medium)
026	032	SUB	(Substitute)
027	033	ESC	(Escape)
028	034	FS	(File Separator)
029	035	GS	(Group Separator)
030	036	RS	(Request to Send)(Record Separator)
031	037	US	(Unit Separator)
127	177	DEL	(Delete)

## Printable characters

This table lists the printable ASCII characters - the characters that can be directly used inside string constants. Some characters, such as the backslash and the quote characters, may still require an escape sequence inside a string constant.

032		056 8	080 P	104 h
033	!	057 9	081 Q	105 i
034	"	058 :	082 R	106 j
035	#	059 ;	083 S	107 k
036	\$	060 <	084 T	108 l
037	%	061 =	085 U	109 m
038	&	062 >	086 V	110 n
039	'	063 ?	087 W	111 o
040	(	064 @	088 X	112 p
041	)	065 A	089 Y	113 q
042	*	066 B	090 Z	114 r
043	+	067 C	091 [	115 s
044	,	068 D	092 \	116 t
045	-	069 E	093 ]	117 u
046	.	070 F	094 ^	118 v
047	/	071 G	095 _	119 w
048	0	072 H	096 `	120 x
049	1	073 I	097 a	121 y
050	2	074 J	098 b	122 z
051	3	075 K	099 c	123 {
052	4	076 L	100 d	124
053	5	077 M	101 e	125 }
054	6	078 N	102 f	126 ~
055	7	079 O	103 g	