

Imagen

for Windows/Linux

Language Reference

March 27, 2001. Document Revision 3

Netclime Inc.

P.O.Box 251666, LA, CA-90025
Tel (310) 571-3135 Fax (646) 514-0412

email: imagen@netclime.com

Introduction

This document describes in detail the structure of Imagen scripts as well as the actions taken by Imagen according to script content.

Imagen exports several routines for loading, manipulating and generating of images (defined in Imagen Runtime Library Reference). To allow easy and flexible use of these routines, a basic scripting language is provided. The script generally defines what routines in what order and with what parameters to be invoked.

The scripting language inherits small subset of the widely used, general-purpose programming language C. Some things are changed for easier usage. The result is also general purpose language: It does not depend on Imagen, it still only provides way to call a set of predefined routines (named here *external routines* or *external functions*).

Organization of this Document

The remainder of this document is divided into the following chapters:

- ◆ Language Elements
- ◆ Phases of Execution
- ◆ Appendixes

Language Elements

Lexemes

The Imagen language is built on top of several atomic elements named lexemes. Lexemes are then used to form the language constructs. At the point of the interpreter, the lexemes are **Keywords**, **Identifiers** and **Constants**. According to their representation in the source files, they are **Names**, **Punctuation**, **Integer Constants** and **String Literals**. See **Lexeme Encoding** for the source file representation. Here are described the lexemes at the point of the interpreter.

Keywords

Keywords are used to form the program structure according to the language syntax. For list of all keywords, refer to the **Keywords** tables in the Appendix. Some keywords form the structure of an expression. They are also named operators.

Keywords are encoded either as **Names** or as **Punctuation**.

Identifiers

Identifiers name user supplied objects like variables and functions. They are encoded as **Names** and must be different from any keyword.

Constants

Constants are integer or string. They have value that is the integer for integer constants, and string (character sequence) for string constants. They are encoded as **integer constants** and as **string literals**.

Data types

The language allows operating with the following data types:

<code>int</code>	Represents inequal number in the range -2^{31} to $2^{31}-1$.
<code>string</code>	Represents zero or more ASCII characters.

Additional data types cannot be defined.

There is also a pseudo-type `void` used to declare functions that do not return values.

Expressions

An expression represents a calculation. It has type and value obtained by evaluation.

Expressions are:

- ◆ IntegerExpression (for expressions of type `int`, i.e. evaluate to integer)
- ◆ StringExpression (for expressions of type `string`, i.e. evaluate to string)
- ◆ Constants (integer and string)
- ◆ Variables
- ◆ Function calls for functions that do not return `void`. The expression evaluates to the returned from the function value.
- ◆ Arithmetic, logical and string expressions denoted by arithmetic, logical and string operators.
- ◆ (Expression), that's expression enclosed by parenthesis.

Arithmetic Operators

- ◆ + IntegerExpression
Returns its argument. Exists only for symmetry with - IntegerExpression
`+3 → 3`
- ◆ - IntegerExpression
Subtracts its operand from zero. Example:
`-(2+3) → -5`
`--3 → 3`
- ◆ IntegerExpression + IntegerExpression
Adds the two integer expressions. Example:
`5 + 3 → 8`
- ◆ IntegerExpression - IntegerExpression
Subtracts the second operand from the first. Example:
`5 - 3 → 2`
- ◆ IntegerExpression * IntegerExpression

Multiplies the two integer expressions. Example:

5 * 3 → 15

◆ IntegerExpression / IntegerExpression

Divides the first operand by the second, truncating the result down to zero. Example:

5 / 3 → 1

11 / 4 → 2

◆ IntegerExpression % IntegerExpression

Divides the first operand by the second, and returns the remainder. Example:

5 % 3 → 2

6 % 3 → 0

7 % 3 → 1

8 % 3 → 2

9 % 3 → 0

Logical Operators

Logical operators return boolean values. A boolean represents either true or false condition and stands for the result of some comparison. There is no boolean type in the script and thus integers are used: 0 represents false condition, and all other values represent true condition. Note that although any non-zero value is considered true, the logical operators return true encoded only as 1.

Strings can also be converted to boolean values by similar scheme: empty strings evaluate to false, and all others evaluate to true.

The logical operators are:

◆ !Expression

Logical *not* operator: Negates the boolean expression, i.e. true only if operand is false:

operand	result
false	true
true	false

Example:

!0 → 1 // i.e. false becomes true

!15 → 0 // and true becomes false

!"text" → 0

!"" → 1

More real example would be:

```
string text;
/* some code here */
if (!text) { /* the text is empty */ } else { /* it's not */ }
```

◆ Expression || Expression

Logical *or* operator: Returns true if at least one of the operands is true:

op. 1 \ op. 2	false	true
false	false	true
true	true	true

Example:

```
0 || 0 → 0
72 || "hello" → 1
x || !x → 1
```

◆ Expression && Expression

Logical *and* operator: Returns true if both operands are true:

op. 1 \ op. 2	false	true
false	false	false
true	false	true

Example:

```
0 && 0 → 0
72 && "hello" → 1
x && !x → 0
```

◆ Expression == Expression

Logical *equality* operator: Returns true if the operands are equal. Produces error if the operands have different types. Example:

```
5 == 5 → 1
2 == 3 → 0
"text" == "different text" → 0
"the same" == "the same" → 1
3 == "text" // produces error message, and terminates script
```

◆ Expression != Expression

Logical *inequality* operator: Returns true if the operands are different. Produces error if the operands have different types. Example:

```
5 != 5 → 0
2 != 3 → 1
"text" != "different text" → 1
"the same" != "the same" → 0
3 != "text" // produces error message, and terminates script
```

◆ IntegerExpression < IntegerExpression

Logical *less than* operator: Returns true if the left operand is less than the right one. Example:

```
10 < 15 → 1
15 < 10 → 0
10 < 10 → 0
```

`-5 < 8 → 1`

◆ `IntegerExpression > IntegerExpression`

Logical *greater than* operator: Returns true if the left operand is greater than the right one. Example:

`10 > 15 → 0`

`15 > 10 → 1`

`10 > 10 → 0`

`-5 > 8 → 0`

◆ `IntegerExpression <= IntegerExpression`

Logical *less than or equal* operator: Returns true if the left operand is less than or equal to the right one. Example:

`10 <= 15 → 1`

`15 <= 10 → 0`

`10 <= 10 → 1`

`-5 <= 8 → 1`

◆ `IntegerExpression >= IntegerExpression`

Logical *greater than or equal* operator: Returns true if the left operand is greater than or equal to the right one. Example:

`10 >= 15 → 0`

`15 >= 10 → 1`

`10 >= 10 → 1`

`-5 >= 8 → 0`

String Operators

There is only one string operator:

◆ `StringExpression + StringExpression`

String concatenation: Returns string that contains the left string concatenated with the right string. Example:

`"abc" + "def" → "abcdef"`

`"some " + "text" → "some text"`

`"" + "another text" → "another text"`

Scopes

The language allows defining new objects: variables and functions. Since placing all variables at one place obstructs creating larger scripts, scopes are introduced.

Scope is place where variables and functions are defined. Such defining adds the name of the object into the scope. It's illegal to add name that already exists in given scope again, but is legal among different scopes. This includes defining different objects with the same name: i.e. define variable and function to have one name.

There is "root" scope named *global scope*. Functions can be defined only in the global scope, while variables can be defined in other scopes too.

Every scope different from the global scope has parent scope. Whenever a name is not found in a scope, the name is searched in its parent scope. If the name is not found in that parent, its parent is checked, and so on, until the global scope is reached. If the name is not in the global scope, the name is considered "unknown", and error is issued.

Scopes are generated by functions and block statements.

Objects are placed in a scope named *current scope*. This is initially the global scope. During the **Loading Global Definitions** and **Executing the Script** phases, it can be changed to point at another scope.

Visibility

The visibility of objects (functions or variables) represents where in the script they can be referred. The referring can be performed by using the name used to define the object, so the visibility specifies where in the script the same name can be used to refer to the specified object. Two rules apply:

Visible from Current Scope

As mentioned under **Scopes**, every object is placed in a scope. Since referring functions and variables can happen only inside statements (see **Statements**) and that statements also lie inside some specified scope so one can deduce that referring happens always from some given scope. For the object to be visible, it must be defined either in the same scope, or in parent of the referring scope.

Already defined

The object must be defined before being referred.

Variables

Variable is term for named data storage. It has name, data type and value. The name is user-supplied identifier. The data type is one of the data types above. The value is any element of the set of possible values for the data type, i.e. integer for `int` and character sequence for `string`.

Variables are defined by a *VarsDefStatement*. Variable definitions describe the type and name of the variable and optionally provide initial value. If initial value is not provided, a default is provided: 0 for `int` and "" for `string`.

Variables can also be created from input parameters as specified in the Imagen Operation and Usage document.

Functions

Functions represent tasks that have to be done by the script. A function has name, arguments (optional), and optionally returns data of some type.

Functions can be called by their name and providing the necessary arguments. Imagen searches its functions data bases and if finds a function with the same name, it passes it the arguments and calls it.

The functions data base initially contains only the external functions. New functions can be added by a *FunctionDefinition*. Function definitions describe the name of the function, its return type, arguments and a **block statement** containing what the function does.

Statements

Statements are instructions that Imagen executes. They exist only in functions and are executed only by calling the containing function. Every statement exists in some scope. This scope is always created by block statements. As exception block statements can also exist as part of function definitions.

variable definition statement

The variable definition statement is used to define new variable in the current scope. It has one of the forms:

```
DataType Identifier;
DataType Identifier = Expression;
```

Examples:

```
int j;
int foo=15;
string item;
```

More than one variable can be defined by separating names with commas:

```
int j, foo=15, last_y;
```

The strict definition is a *VarsDefStatement*.

The statement can be used to "define variable if not already defined in this scope". This can be done by a *VarsDefStatement* that has the optional **default** keyword added. In this case if the variable was not already defined, it is defined (and optionally initialized), just as if the **default** keyword was not used. If the variable was defined, its type is verified to match the new type. If types don't match an error is issued.

The **default** keyword is most appropriate when used for script parameters. For example placing the following statement somewhere in the global namespace:

```
default int param=7;
```

Gives default value of the variable, which can be overridden by a param=value clause stated in the command line or invoking URL. Please refer to Imagen Operation and Usage for more information.

assignment statement

The assignment statement is used to set the value of some variable to the result of an expression. It has the following form:

```
Identifier = Expression;
```

Examples:

```
x = 5;
y = 3*x + function(arg1, arg2);
```

The statement is executed by evaluating the expression and then assigning the result to the variable. The type of the expression must match the type of the variable.

block statement

The block statement represents list of statements to be executed in sequence. It creates new scope, with parent the current scope.

It has the following form:

```
{
  Statement // 0 or more times
}
```

if statement

The if statement executes given statement only if some condition is true. If the condition is false the statement is ignored. It can optionally accept second statement that will be executed only if the condition is false. It has one of the following forms:

```
if (Expression) Statement
if (Expression) Statement else Statement
```

Examples:

```
if (x > 0) z = y / x; else { z = 3; x = 1; }
```

This statement divides y by x and stores the result in z only if x is greater than zero. Otherwise z is set to 3.

while statement

The while statement provides ability to execute some *controlled* statement several times.

While given condition is true the controlled statement is executed. This is repeated until the condition becomes false (which mostly happens with the help of the controlled statement).

The statement has the following form:

```
while (Expression) Statement
```

Example:

```
int s=0, i=5;
while (i<=10) { s=s+i; i=i+1; }
```

This code fragment will execute until the expression $i \leq 10$ becomes false. That is until i becomes greater than 10. As consequence the variable s will hold the sum of the numbers from 5 to 10 inclusively after the loop completes.

do-while statement

The do-while statement provides ability to execute some *controlled* statement several times. The only difference from the while statement is that it first executes the controlled

statement, and then checks if it should continue executing it. Thus the controlled statement is always executed at least once.

The statement has the following form:

```
do BlockStatement while (Expression);
```

Example:

```
int s=0, i=5;
do { s=s+i; i=i+1; } while (i<=10);
```

function call statement

Function call statements have the following form:

```
FuncName ();
FuncName (Expression {, Expression } ); // the {} denote zero or more repeats
```

These statements allow executing function call expressions. The value returned from the function (for non-void functions) is ignored.

Examples:

```
func (5);
func ("text", alpha * beta);
```

return statement

The return statement ends the execution of a function and provides its return value (if such). If the function does not return a value (it is of type void) the statement has the form:

```
return;
```

If the function returns a value, the statement contains expression which evaluation is considered the return value from the function:

```
return Expression;
```

Phases of Execution

Processing of Imagen Script consists of the following operations:

- ◆ Preprocessing
- ◆ Loading Global Definitions
- ◆ Executing the Script

Preprocessing

The preprocessing phase is responsible to load the required source files and to execute the preprocessor directives found there. The result is list of lexemes passed to the next phase. Preprocessing of single source file consists of the following:

- ◆ Extracting lexemes

The source file is read and broken into array of lexemes according to the rules under **Lexeme Encoding**

- ◆ Handle preprocessor directives

The lexemes are scanned for preprocessor directives. These directives allow some modifications of the script before executing as well as giving some out-of-the-language information to the interpreter (like what language to be used).

Preprocessor directives can appear anywhere in the script. They are found by searching for the the pound sign (#). Directives are scanned from the beginning of the file to its end, and whenever one is found, it's effects are immediately applied. See **Preprocessor Directives** below for details.

Lexeme Encoding

There are four forms that represent a lexeme in a script: name, punctuation, integer constant and string literal. The lexemes are separated by white-space characters and comments with the exception of punctuation. Punctuation symbols do not need whitespace or comments to be separated from neighbouring lexemes.

Names

A name consists of letter or underscore followed by zero or more underscores or alphanumeric (letter or decimal digit) characters. **Names** stand for human readable names of objects like built-in language operators (**if**, **return**, etc.) or user supplied names of variables and functions.

Examples:

```
name
one23
x12
_anotherName
return
```

Punctuation

Lexemes encoded as punctuation are sequences of non-letter, non-digit, non-whitespace characters. There is fixed set of such sequences listed in the **Keywords encoded as punctuation** table.

Integer Constants

An integer constant consists of one or more decimal digits. Integer constants provide integral numbers into the script.

Example:

```
0
874023
0123
00033
```

String Literals

A String Literal represents sequence of zero or more characters. The characters are enclosed by double quotation marks ("). To represent the double quotation mark inside string literal an escape sequence is used. Escape sequence begins with backslash and continues with escape character. For example \" and \\ encode " and \ respectively. Refer to the **Escape sequences** table for listing of all supported escape sequences.

Preprocessor Directives

There are several preprocessor directives handled during preprocessing. Each directive begins with # and continues with preprocessor keyword that names the directive. After the keyword follow zero or more directive parameters. After a directive is processed, it is removed from the lexemes array. That's preprocessor directives are never passed to the interpreter.

The directives are:

- ◆ Version

The version directive has the following syntax:

```
# version LeStringLiteral
```

This directive must appear at the beginning of each script. The string literal represents the version that the script expects from Imagen. This directive is intended for future enhancements to the script and external routines sets that are incompatible with current versions.

Currently the string must be "1.0".

Future versions of Imagen that change the language syntax and/or change the external routines available should require different version string. Such versions may still support the older behavior by switching to the old language syntax and external routines when old version string is encountered.

- ◆ Include

The include directive has the following syntax:

```
# include LeStringLiteral
```

This directive informs the preprocessor to include the file pointed by the file name stored in the string at the place of the include directive. The file name is relative to the directory where the including source file is located.

The including consists of full preprocessing of the file: the lexemes are extracted from it, and then the preprocessor directives are handled as described here. At the end, the resulting lexemes are inserted at the place of the directive.

Note that the file name is OS dependent, but with the only extension: For platforms that use the backslash (\) for directory separator, the slash (/) can still be used. This avoids the use of double backslashes (since \ begins escape sequences).

Loading Global Definitions

The lexemes accumulated in the **Preprocessing** phase are parsed for global declarations (*GlobalDeclaration*). Global declarations constitute of variable and function definitions, and both define new objects into the global scope.

Executing the Script

Since the previous phase collected all global scope objects, the script can be executed. This is realized by searching for and executing of a function called `main`. If no such function is found, an error is issued.

Appendixes

Keywords

Preprocessor Keywords

The keywords recognized by the preprocessor are:

encoded as names

The keywords that are encoded as names are:

`include version`

encoded as punctuation

The keywords that are encoded as punctuation are:

`#`

Language Keywords

The keywords recognized by the language interpreter are:

encoded as names

The keywords that are encoded as names are:

`default do else for if int return string void while`

encoded as punctuation

The keywords that are encoded as punctuation are:

`! && || = == != < > <= >= + - * / % () { } , ;`

Operator Precedence

This table describes the precedence of operators, which affects their grouping. The operators are sorted by precedence, where the top-most have highest precedence, while the bottom-most have lowest precedence. Operators in the same group have equal precedence.

All operators have left associativity.

Symbol	Type of Operation
<i>highest precedence</i>	
()	Expression

+ - !	Unary
* / %	Multiplicative
+ -	Additive
== != < > <= >=	Relational
&&	Logical AND
	Logical OR
<i>lowest precedence</i>	

The operator precedence allows resolving ambiguities found in complex expressions. For instance:

$$a + b * c$$

could be interpreted as both $(a + b) * c$ or as $a + (b * c)$. The rule is that operators with higher precedence take their operands first, and for the other operators they along with their operands appear as single operand. In the example, the multiplication sign (*) has higher precedence than the addition (+), so the expression will be evaluated as

$$a + (b * c).$$

It's possible, for expression to contain operands with the same priority. In that case, their associativity rules the precedence among them. Since all operators are with left associativity, it's the left operator that takes precedence. For example:

$$(8/2/2) \text{ becomes } ((8/2)/2) \text{ which equals } 2, \text{ not } (8/(2/2)) \text{ which equals } 8.$$

If you need different precedence than the supplied, you can use the parenthesis. I.e.

$$(a + b) * c \text{ will not be changed to } a + (b * c)$$

Escape Sequences

The escape sequences handled by Imagen are the same as the escape sequences introduced in the C programming language, with the exception of octal and hexadecimal escapes:

Escape Sequence	Represents
\a	Bell (alert)
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\?	Literal question mark
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash

Backus-Naur Form (BNF) Reference

The BNF forms used in this document have the following meaning:

1. Bold text represents literal characters.
2. Italic text stands for some defined syntactic category.
3. Syntactic category is defined by the form: *category* ::= *definition*
4. Square brackets ([]) surround optional items.
5. Curly brackets ({ }) surround items that can repeat zero or more times.
6. A vertical line (|) separates alternatives.
7. Parenthesis (()) range the extent of |.
8. An ellipsis (. . .) between | stands for range of alternatives.

Example:

```
UpperAlpha ::= A | B | ... | X | Y | Z
LowerAlpha ::= a | b | ... | x | y | z
Alpha ::= UpperAlpha | LowerAlpha
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
Number ::= Digit { Digit }
AlphaNumeric ::= Alpha | Digit
Name ::= ( Alpha | _ ) { AlphaNumeric | _ }
```

Thus UpperAlpha is any upper English letter, and Name is anything that begins with English letter or underscore and continues with zero or more letters, digits or underscores.

Language Syntax Summary in BNF

Here are described all language syntax elements by using Backus-Naur Forms. According to this summary Imagen script is described by the *ImagenProgram* syntactic category at the bottom.

Character Classes

```
AnyCharacter ::= <<any ASCII character>>
UpperAlpha ::= A | B | ... | X | Y | Z
LowerAlpha ::= a | b | ... | x | y | z
Alpha ::= UpperAlpha | LowerAlpha
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
AlphaNumeric ::= Alpha | Digit
```

Lexeme Encoding Classes

```
Name ::= ( Alpha | _ ) { AlphaNumeric | _ }
Punctuation ::= ! | && | || | = | == | != | < | > | <= | >= | + | - | * |
                / | % | ( | ) | { | } | , | ;
LeInteger ::= Digit { Digit }
EscapeId ::= a | b | f | n | r | t | v | ? | ' | " | \
NormalStrChar ::= <<any ASCII character except \ and ">>
```

```

InputChar ::= \ EscapeId | NormalStrChar
LeStringLiteral ::= " { InputChar } "

```

Preprocessor Directives

```

PreprocessorDirective ::= # { VersionDirective | IncludeDirective }
VersionDirective ::= version LeStringLiteral
IncludeDirective ::= include LeStringLiteral

```

Lexemes

```

Identifier ::= Name
Keyword ::= Punctuation | default | do | else | for | if | int | return |
string | void | while
IntegerConstant ::= LeInteger
StringConstant ::= LeStringLiteral
Constant ::= IntegerConstant | StringConstant
VarName ::= Identifier
FuncName ::= Identifier

```

Types

```

DataType ::= int | string
ExtendedType ::= DataType | void

```

Expressions

```

Expression ::=
    OrExpression
OrExpression ::=
    AndExpression [ || OrExpression ]
AndExpression ::=
    RelationalExpression [ && AndExpression ]
RelationalExpression ::=
    AdditiveExpression [ RelationalOperator AdditiveExpression ]
AdditiveExpression ::=
    MultiplicativeExpression [ AdditiveOperator AdditiveExpression ]
MultiplicativeExpression ::=
    UnaryExpression [ MultiplicativeOperator MultiplicativeExpression ]
UnaryExpression ::=
    [ UnaryOperator ] PrimaryExpression
PrimaryExpression ::=
    Constant | VarName | FunctionCall | ( Expression )

RelationalOperator ::= == | != | < | > | <= | >=
AdditiveOperator ::= + | -

```

*MultiplicativeOperator ::= * | / | %*

UnaryOperator ::= + | - | !

FunctionCall ::= FuncName ([Expression { , Expression }])

Statements

*Statement ::= BlockStatement | IfStatement | WhileStatement |
DoWhileStatement | VarsDefStatement | AssignStatement |
FunctionCallStatement*

BlockStatement ::= { { Statement } }

IfStatement ::= if (Expression) Statement [else Statement]

WhileStatement ::= while (Expression) Statement

DoWhileStatement ::= do BlockStatement while (Expression) ;

VarDefPart ::= Identifier [= Expression]

VarsDefStatement ::= [default] DataType VarDefPart { , VarDefPart } ;

AssignStatement ::= VarName = Expression ;

FunctionCallStatement ::= FunctionCall ;

Global declarations

GlobalDeclaration ::= VarsDefStatement | FunctionDefinition

*FunctionDefinition ::=
ExtendedType Identifier ([FormalArgument { , FormalArgument }])
BlockStatement*

Imagen Program

ImagenProgram ::= { GlobalDeclaration }

TBDs

- ◆ Behavior of (-5/-3) and (-5%-3)
- ◆ Handling of Ctrl-Z
- ◆ Null statements (;;;)