Chapter 7
Clovis Script

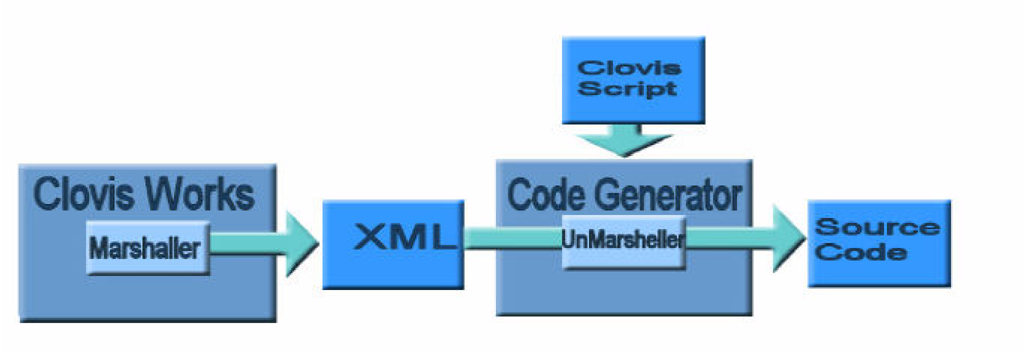
This section covers the following topics:

2.1 Clovis Script Overview
2.2 Clovis Script Syntax
2.3 Clovis Script Data Types and Variables
2.4 Clovis Scripts Operators
2.5 Clovis Script Statements
2.6 Clovis Script Functions
2.7 Clovis Script File I/O
2.8 Clovis Script XML Operation
2.9 Clovis Script Templates

2.1 Clovis Script Overview

Clovis Script is a 'C' language like scripting language to help you customize code generation. Clovis Script can be used to define the templates for source code files to be generated. These templates can be easily customized and you can decide your own output format for the source files. These templates must be saved with *.cs* extension and can be created using WordPad or the default editor of ClovisWorks.

ClovisWorks code generator accepts XML files and Clovis Script files as input, and marshall the XML parser to create source code files.



Following are some of the features of Clovis Script:

- Easy to use.
- Supports schema based xml reading.
- Powerful built-in support for template-bases text processing.

Contents - Index - Back

2.2 Clovis Script Syntax

The syntax of Clovis Script is similar to that of 'C' language. Clovis Script consists of functions having one or more expressions. The "main" function is the starting point for script execution.

2.3 Clovis Script Data Types and Variables

Following are the data types that can be used in Clovis Script.

- Number: integer/float/short/long
- String
- List
- Map
- Void
- Boolean

A variable has a name and a type. Clovis Script is not a type strict language. Variables need not be declared before use. Type of a variable is either explicitly given or deduced from assignment.

Example

```
int x;  //Declare x as int
x = 1;  //assign 1 to x
    Alternative, you can write:
x = 1;  //x is int with value 1.
```

Type of a variable can be changed after declaration. In the above example, x is an *integer*. Later in the script, if you declare:

`x = "Clovis"`

x will no longer be an integer. Its type will change to *string*. The assignment takes higher priority in determining the data type.

2.4 Clovis Scripts Operators

Equality, Relational and Logical Operators: These operators evaluate true or false for an expression.

Table 2-1 Equality Operators

Operator	Meaning	Usage
==	Equality expression	expression== expression
!=	Inequality expression	expression!= expression

The **equality operators** are defined for numbers and string expressions.:

Table 2-2 Relational Operators

Operator	Meaning	Usage
<	Less than expression	expression<expression

Contents - Index - Back

Table 2-2 Relational Operators

Operator	Meaning	Usage
< =	Less than or equal to expression	expression<=expression
>	Greater than expression	expression>expression
>=	Greater than or equal to expression	expression>=expression

The relational operators are defined only for numeric expressions.

Table 2-3 Logical Operators

Operator	Meaning	Usage
!	Logical NOT expression	expression! expression
||	Logical OR expression	expression || expression
&&	Logical AND expression	expression && expression

Table 2-4 Other Operators

Operator	Type	Meaning	Example
#	string	Length of String	string str;
#	list	Number of Elements in List	int x = #str;
#	map	Number of Elements in Map	
+	string	Contatenate two Strings	list x;
+	list	Add element to list	x = x + "1"
[]	array	Access given index	x[1] = "test";
[]	list	Access given index	
[]	map	Access given key	x["key"] = "value";

2.5 Clovis Script Statements

Statements are the fundamental units of execution in Clovis Script. The default order of execution of statements within a script is sequential. The following control flow statements allow you to change the order of execution. Syntax for all these statements are the same as that of 'C' language.

- if
- if - else

Contents - Index - Back

- for
- while
- switch
- break
- continue

Block Statements

The braces '{' and '}' can be used to group several statements and variable declarations into a compound statement or a block. A block is syntactically equivalent to a single statement. The default order of evaluation for statements in a block is sequential.

2.6 Clovis Script Functions

A function is a user defined operation. A function has a *name*, an *argument list*, a *return type*, and a *body.*

The list of operands for a function is called the argument list. An individual operand is called an argument. The argument list determines how many, and what type of, parameters a function requires. The argument list is enclosed in parenthesis and the arguments are separated by commas.

The type of result of a function is it's return type. If a function does not return any value it's return type is void. The result of a function, if any, is returned by using the return statement.

The body of a function contains statements or statement blocks.

An example of a function declaration

```
int add(int a, int b)
        {
            return a + b
        }
```

Utility Functions

Following are some of the utility functions you can use in Clovis Script:

print: Prints the arguments (comma separated) in the log file.

Examples:

```
            print "Hello World";
            print "Number of entries = ", x;
```

load: Described in Clovis Script XML Operation.

2.7 Clovis Script File I/O

Clovis Script makes use of *File Objects* for file I/O. It supports open, write and close operations. The following example explains all File I/O functions.

```
 fileObj = new File; //Create File object.
fd = fileObj.open(filePath);//Open file given by filePath
 if (fd != null) {
     fileObj.write(fd,content);//Write content to file
     fileObj.close()fd;
 }
```

Contents - Index - Back

2.8 Clovis Script XML Operation

Clovis Script provides schema (xsd) based XML operations. JAXB classes for the schema are required to use XML functionality of Clovis Script.
The "**load**" function is used to read an xml. Following is the syntax of the *load* function:

```
        load <package-name>, <xml-file path>
```

After executing the above statement, xml will be loaded in a variable. This variable is implicitly defined. The name of this variable is generated after replacing all dots ('.') in package name with underscores ('_').
For example, for the package "**com.foo**", variable name will be **com_foo**.

For example, consider the following xsd:

```
<?xml version="1.0" encoding="UTF-8"?>
        <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
            <xs:element name="Ports">
                <xs:complexType>
                    <xs:sequence>
                        <xs:element name="PortNo" type="xs:int"
                           minOccurs="0" maxOccurs="unbounded"/>
                    </xs:sequence>
                </xs:complexType>
            </xs:element>
        </xs:schema>
```

If JAXB classes for the above example is generated in the package "com.foo", following statements will read "ports.xml" and print PortNo in the log file.

```
//Load ports.xml according to package com.foo
    load com.foo, ports.xml;
    for (i = 0; i < #com_foo.Ports.PortNo; i++) {
       print "PortNo read from XLS = ", com_foo.Ports.PortNo[i];
    }
```

Intermediate objects that are read from the XML can be saved in a variable. Above example can also be written as shown below:

```
//Load ports.xml according to package com.foo
    load com.foo, ports.xml;
    portObj = com_foo.Ports;
    for (i = 0; i < #portObj.PortNo; i++) {
        print "PortNo = ", portObj.PortNo[i];
    }
```

2.9 Clovis Script Templates

Templates are a very useful feature of Clovis Script. Templates provides a way to define the format of a string, in which values can be provided later. This is very useful during code-generation as the format of code with unknowns can be written first, and then code can be generated by providing these unknowns to the templates.

Following is the syntax for defining a template. In the format unknowns are written as @<*unknown-name*>.

```
   string <template-name> = <<{
            <free-format-string>
        }>>;
```

The example of printing port numbers can be simplified using template as shown below:

```
//Define template
        string portTemplate = <<{
            PortNo read from XLS = @portNo
        }>>;
        //Load ports.xml according to package com.foo
        load com.foo, ports.xml;
        for (i = 0; i < #com_foo.Ports.PortNo; i++) {
            portNo = com_foo.Ports.PortNo[i];
            print portTemplate;
        }
```

In this template, (*portTemplate*) the portNo is an unknown, When this template is used, value of portNo is filled up from current context. There is no limit of unknowns in a template.

List Templates

To provide repetitive code generation from a list of unknowns, list-based template can be used. This template expands the format for each entry in the list.

Following is the syntax for list-based template:

```
            @@<list-object>{{
                <free-format-string>
            }}
```

In the above template, @<list-object> will give i'th element in i'th iteration. The example of printing port numbers can be further simplified using list-based template as shown below.

```
        //Define template
        string portTemplate == <<{
            @@portNoList{{
                PortNo read from XLS = @portNoList
            }}
        }>>;
        //Print the port nos
        print portTemplate;
```