Programming Languages and Translators

COMS W4115

Prof. Stephen Edwards

Department of Computer Science

Fall 2007

TweaXML

Language Reference Manual

Kaushal Kumar
kk2457@columbia.edu

Srinivasa Valluripalli
sv2232@columbia.edu

Introduction

The following document is a reference manual for TweaXML, a language to perform operations on XML documents.

Lexical Conventions

Tokens

Tokens are divided as follows, *Identifiers, Keywords, Constants, Operators, Other Separators.*

Blanks, horizontal and vertical tabs, newlines, form feeds and comments are considered as “white space”. They have no semantic meaning, and are used only as token separators.

The input is validated into various tokens that are available.

Comments

Comments are defined as a sequence of characters that begin with the sequence /* and end with the sequence */ . Comments cannot be nested and cannot occur inside a string or character literals.

Identifier

Identifiers are typically variables and they comprise of alphabets and numbers. Identifiers should essentially start with a character between [A-Z] or [a-z]. All succeeding characters can be characters from the sets, [A-Z], [a-z], [0-9] or an underscore (_).

Keywords

Following are the reserved keywords,

number boolean string file

open close read write
node countChildren containsNode
getValue getChild xmldoc if
else foreach return continue
break void

Constants

Number Constants

Being an XML document, there is no point in having separate data types for integer, float and double etc. “number” data type is provided to deal with all kind of data types.

String Constants

Every constant that is not a number is a *string* constant. Strings are enclosed in double quotes “ ”. The escape sequence ‘\n’ denotes a new line and it can be used in any string constants.

Punctuations

These are the punctuation symbols in TweaXML.

; statement end
, argument separator
“ ” string constant
{} function body or block delimiter
() function argument list

Data Types

TweaXML has four data types available,

number This data type represents all the numbers including integer and real present in the document.

string This data type is can hold multiple characters of variable length.

xmldoc This data type represents the XML documents

node Using this data type we can access the nodes in the documents.

Expressions

Expressions are all evaluated left to right. In case of logical expressions, if the evaluation of the first n elements of the expression is enough for evaluating the whole expression itself, the n+1th element will not be evaluated.

Arithmetic Expressions

Arithmetic expressions include operations such as +, -, *, / between *number* data types. For example, a + b * c - d

Conditional Expressions

Conditional Expressions are those involving comparison operators such as <, <=, >, >=, == and !=. These can be used in *if*, *while* and *foreach* loops. The associativity of these operators is from left to right. All of these operators are binary operators that work on two operands. Examples are a<b, b>=d.

Logical Expressions

Logical expressions return a boolean result i.e. either true or false. The supported operators are AND (&&), OR (||) and NOT (!).

All of the above expressions are evaluated from left to right and standard precedence rules are applicable throughout the language. ! has the highest precendence, followed by && and then by ||.

Operators

Unary Operators

Unary operators are applied on the expressions on their right.

Minus sign: - *expression*
The result of the minus sign **-** operator is the negative value of *expression*. *expression* must be of type **number**.

Logical negation: ! *expression*
The result of the logical negation operator *!* is the constant 1 of type **number** if *expression* value is 0; constant 0 of type **number** if *expression* value is otherwise. *Expression* must be of type **number.**

Arithmetic Operators

Multiplication –
expr1 * expr2 returns a value of type *number*. expr1 and expr2 must be of type **number** and this operator returns the product of the two values.

Division –
expr1 / expr2 returns a value of type *number*. Division by 0 is forbidden and will produce a runtime error. expr1 and expr2 must be of type **number** and this operator returns the division of the two values.

Addition –
expr1 + expr2 returns a value of type *number* if both expressions are of number type. But + can be performed also on strings to concatenate two strings.

Subtraction –
expr1 - expr2 returns a value of type *number*. expr1 and expr2 must be of type **number** and this operator returns the difference between the two

values.

Relational Operators

Greater than - expr1 > expr2
Lesser than - expr1 < expr2
Greater than or equal to - expr1 >= expr2
Lesser than or equal to - expr1 <= expr2

All the above operators return 1 if true and 0 if false. Both expr1 and expr2 should be of type number or string (a combination is not allowed).

In case of number, expr1 > expr2 and expr1 < expr2 will return true (1) and false (0) respectively, if expr1 is numerically greater than expr2. Similarly, expr1 >= expr2 and expr1 <= expr2 will return true (1) and false (0) respectively, if expr1 is numerically greater than or equal to expr2.

In case of strings, expr1 > expr2 and expr1 < expr2 will return true (1) and false (0) respectively, if expr1 comes before expr2 alphabetically. Similarly, expr1 >= expr2 and expr1 <= expr2 will return true (1) and false (0) respectively, if expr1 comes before or is same as expr2 alphabetically.

Equality Operators

Equality - expr1 == expr2
Inequality - expr1!= expr2

The above operators return 1 if true and 0 if false. Both the expressions expr1 and expr2 should be of type number or string (a combination is not allowed).

Logical Operators

AND - expr1 && expr2
Logical AND returns 1 if only both expressions are true and returns false for all other cases.

OR - expr1 || expr2
Logical OR returns 1 if both the expressions are true or either of the expressions is true. It returns 0 if both expressions are false.

NOT - !(expr)
Logical NOT returns 1 if the expression is false and 0 if the expression is true.

Operator Precedence

( )
!
* /
+ -
< > <= >= == !=
&& ||
=
,

All binary operators associate left-to-right whereas the unary operators associate right-to-left.

Variable Declaration

The *number* and *string* variables are declared similar to Java.

number n1, n2;
string s1;

Assignment Statements

Variables can be assigned with either expressions or constants.

```
number n1 = expression;
number n2 = 9;
```

Conditional Statements

The conditional statements in this language are *if-else*,

```
if ( condition)
{
        statement 1;
        statement 2;
}
else
{
        statement 1;
        statement 2;
}
```

Loop Statements

There are two looping statements, *while* and *foreach*.

```
while ( condition)
{
        statement 1;
        statement 2;
}
```

The while block will be executed only if 'condition' evaluates to a value other than 0.

```
foreach ( expression )
{
```

```
    statement 1;
    statement 2;
}
```

'foreach' block will be executed for each value assumed by the expression. In case of nodes of a xml, getchild function returns a collection of nodes. 'foreach' loop statement can be used to loop over those nodes.

Write Statement

Write statements can be used to write the output to a file. The syntax is:
write(file, expression)

Jump statement

Jump statements are used to jump to a different set of statements altogether. They affect the control flow immediately without any condition check as in iterative or conditional statements.

return : used in functions to return a single value or void.
continue : used to continue with the next iteration in a while loop construct.
break : used to break out of a while loop immediately.

Fuctions

There are two types of functions defined in TweaXML: language defined functions and user defined functions.

Main Keywords related to the use of functions:

return

As soon as a return statement is executed, the function returns to its caller. All user defined functions need to have at least one return statement. However, return statement can be omitted in the main() function.

main

main() is a language-defined function which is the execution point in TweaXML programs. In other words, the program cannot be executed without a main function.

Void

If a return type is void, there is no return value of the function. If void is used as an argument, there is no argument needed to call the function.

Function declarations:

A function declaration has four sub parts, namely: a return type, a function name, a list of arguments, a body which consists of statements enclosed by curly brackets.

<return type> <function name> (argument list) {body}

A return type can be any of the basic types available, which are number, string, xmldoc and void. A user can choose an arbitrary name for a user-defined function and the name should follow the definition given before for identifiers. An argument list can contain a number of arguments separated by a comma. An argument should be declared as a type followed by its name.

The body contains statements executed when the function is called is followed.
For example,

```
number sample(number x, number y)
{
        return x+y;
}
```

This is the declaration of function 'sample', whose return type is a number and which takes a number type variable as the first argument and a number variable as the second argument. Then, the arguments are local variables within sample function body. Since the return type of sample

function is of type number, sample function should have at least one return statement which returns a number value. In the example, x+y is returned.

Function calls

User-defined functions can be called in any function including the calling function itself, allowing recursive calls. A function invocation should match exactly the name, number and type of arguments as for the function declaration. No name overloading is allowed. Also, to call a function, the function should be declared in advance. When a function is called with a list of arguments, the arguments are evaluated before calling the function from left to right. When the function returns a value, the returned value can be retrieved. However, this is optional. Thus, the function call follows the notation below.

(<variable> =)? <function name> (argument list);

Scope

A program should be compiled at once which means that all source code should be in one file. Thus, there is only one kind of scope in TweaXML, the lexical scope of an identifier that is the region of a program in which the identifier is recognized.

Lexical Scope

Identifiers in different name space do not interfere with each other. In TweaXML, there are two separate name spaces available: variables name space and functions name space. To clarify the explanation of the lexical scope, the meaning of a block should be defined. A block is a set of statements grouped by curly brackets, {}. Thus, a block of statements is executed like a single statement syntactically.
An identifier can be declared once in a block within a name space.

Identifiers can be used several times in its block. If the same name of an identifier is declared outside the block, the identifier in the outer block will be undermined in the current block. Thus, TweaXML follows static scoping, which means the life of an identifier begins with its declaration and ends at the end of its block.

Language defined Functions:

open(file, "filename")

This function opens a file to read. It takes two arguments: filename should be the location of the file on the disk and it should be in xml file format. It opens this file and assigns it to the file variable given in the first argument. This file variable can be used to read the nodes or write the output.

read(file)

This function reads the input xml file and returns the root node of the xml file. It will give a runtime error if the file is not in xml format. The file should be opened before reading, otherwise it will give runtime error.

contains(node, "node-name")

This function returns true if the xml-node "node" contains "node-name" node, otherwise it will return false.

getChild(node, "node-name")

This function returns all the child nodes of node "node", with the name "node-name". It can be used then to iterate using foreach looping statement.

getValue(node)

This function returns the value of the node "node".

write(output, expression)

This function will write the given expression in the output file "output". This expression can be "\n" for inserting a new line, "\t" for inserting a tab and any other expression which evaluates to a string or a number.

Sample Program:

Input file: (input.xml)

```
<customer-records>
    <customer>
        <name>John</name>
        <order-number>1</order-number>
        <amount>100</amount>
    </customer>
    <customer>
        <name>Jack</name>
        <order-number>2</order-number>
        <amount>50</amount>
    </customer>
    <customer>
        <name>Harry</name>
        <order-number>3</order-number>
        <amount>150</amount>
    </customer>
</customer-records>
```

Program:

```
main(){
    // declaring FILE variable.
    file input;
    file output;

    // opening the file to read
    open(input, "input.xml");
    // opening a file to write in a comma separated format.
```

```
        open(output, "output.csv");

        // gives the root node of the xml document.
        node rootNode = read(input);

        // variable to calculate the total amount.
        number totalAmount = 0;

        if(contains(rootNode, "customer-records/customer/amount"))
        {
                // looping over all the nodes with name "customer-records/customer/amount"
                FOREACH(NODE nextNode = getChild(rootNode,
                                                "customer-records/customer"))
                {
                        write(output, getValue(getChild(nextNode, "name")));
                        write(output,getValue( getChild(nextNode, "order-number"));
                         write(output, getValue(getChild(nextNode, "amount")));
                        write(output,"\n");
                        totalAmount = totalAmount + getValue(getChild(nextNode, "amount")));
                }
                write(output, "Total Amount: ");
                write(output, totalAmount);
        }
}
```

This program will read the input file (given above), and outputs the data in a comma separated csv file. Additionally, it will add the values of all the "amount" nodes and print the added value at the last line of the output file.