DX Language Reference Manual

1. Introduction

DX is a language which has features to create a xml file when given an input file with delimiters. The language is also useful for writing programs which need programming constructs such as if statement, while loops etc.

2. Lexical conventions

There are six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators. In general blanks, tabs, newlines, and comments as described below are ignored except as they serve to separate tokens.
At least one of these characters is required to separate otherwise adjacent identifiers, constants, and certain operator pairs. If the input stream has been parsed into tokens up to a given character, the next token is taken to include the longest
string of characters which could possibly constitute a token.

3 Comments

The characters // introduce a comment, which terminates with the new line. // has no special meaning inside comment line.

4. Identifiers (names)

An identifier is a sequence of letters and digits; the first character must be Alphabetic. The underscore ‘‘_’’ counts
as alphabetic. Upper and lower case letters are considered different.
ID : LETTER (LETTER | DIGIT | '_')* ;

5. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:
BEGIN
WHILE
addRecord
readRecord
createRecord
if
then
else
print
EOF
END

6 Constants

There are several kinds of constants, as follows:

6.1 Integer constants

An integer constant is a sequence of digits.
integer : digit+

6.2. string constants

A string is a sequence of ACSII characters surrounded by double quote

7. Separators

The following ASCII characters are separators:

{
}
;
,
(
)
[
]

8. Variables

A variable is declared using the syntax var variable_name. The variable name must begin with at least one alphabetic, then any combination of alphabetic, integer number and /or underscore '_' is optional.

Variablestatement : "var" ID (initialiser)?
It has an optional initialiser which would assign initial values to the var.

9. Built-in variables

The following identifiers are reserved for use as built-in variables, and may not be used otherwise:

INPUTFILE - filename of the input file which needs to be converted to XML
DELIM - field separator in the input file
ROOTELEM - root element of the output XML
RECNAME - record name in XML (XML element name)
RECATTRIB - record attributes (XML attribute name)
RECVAL - record value i.e the text value of the XML (XML value)
RECVAL[integer] - to get a particular attribute value

10. Expression

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first). Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

10.1 Primary expressions

10.1.1. identifier

An identifier is a primary expression. Its type is specified by its declaration.

10.1.2. integer constant

An integer constant is a primary expression.

10.1.3. string constant

A string is a primary expression.

10.1.4. expression

A parenthesized *expression* is a primary expression whose type and value are identical to those of the unadorned expression

10.2 Operators

10.2.1 Multiplicative operators

The multiplicative operators *, /, and % group lefttoright.

10.2.1.1 expression * expression

The binary * operator indicates multiplication.

10.2.1.2 expression / expression

The binary / operator indicates division.

10.2.2 Additive operators

The additive operators + and - group lefttoright.

10.2.2.1 expression + expression

The result is the sum of the expressions.

10.2.2.2 expression - expression

The result is the difference of the operands

10.2.3 Relational operators

The relational operators group lefttoright,
but this fact is not very useful; ‘‘a<b<c’’ does not mean what it seems
to.

10.2.3.1 expression < expression

10.2.3.2 expression > expression

10.2.3.3 expression <= expression

10.2.3.4 expression >= expression

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. Operand conversion is exactly the same as for the + operator

10.2.4 Equality operators

10.2.4.1 expression == expression

10.2.4.2 expression != expression

10.2.4.3 lvalue = expression

The value of the expression replaces that of the object referred to by the lvalue.

11 Statements

Except as indicated, statements are executed in sequence.

11.1 Expression statement

Most statements are expression statements, which have the form
expression ;
Usually expression statements are assignments or function calls.

11.2 Conditional statement

The four forms of the conditional statement are

if expression statement
if expression statement else statement

In all cases, the expression is evaluated, and if it is non-zero, the first substatement is executed. In the second if expression is zero, then the substatement in the else is executed.

11.3 While statement

The while statement has the form

while expression statement

The substatement is executed repeatedly so long as the value of the expression remains nonzero. The test takes place before each execution of the statement.

12 Scope rules

All components of the DX program must be part of one file and must be compiled at the same time. All variables are declared as global variables. Therefore, there are two kinds of scopes to consider: first, the lexical scope of the variable; and second, the scope of the dependency.

12.1 Lexical scope

DX can be considered similar to a scripting language, and also uses block structuring. Variables declared at any level are globally visible, as DX uses global scoping.

12.2. Scope of dependency

DX built-in variables and built-in functions have some dependencies on one another. DX built-in variables are designed to support this scope. Built in variables initialization is needed to execute the built-in functions. If a built-in variable has a default value, it becomes optional to initialize. The DX library contains built-in functions that can be used to build xml output. The increasing precedence of some built-in variable (see built-in variables for variable specification) and built-in functions is indicated below:

INPUTFILE
XMLFILE
DELIM
ROOTELEM
RECNAME
RECATTRIB
RECVAL
RECVAL[integer]

Lexer and Parser for DX (DxLexer.g)

```
class DXlexer extends Lexer;
options {
   testLiterals = false; // By default, don't check tokens against keywords
   k = 2;                // Need to decide when strings literals end
   charVocabulary = '\3'..'\377'; // Accept all eight-bit ASCII characters
}

tokens{

ADDRECORD = "addRecord";
CREATERECORD = "createRecord";
READRECORD = "readRecord" ;
ROOTELEM ;

}
PLUS   : '+' ;
MINUS  : '-' ;
TIMES  : '*' ;
DIV    : '/' ;
ASSIGN : '=' ;
SEMI   : ';' ;
COMMA  : ',' ;
LPAREN : '{' ;
RPAREN : '}' ;
LBRACKET : '[' ;
RBRACKET : ']' ;
GT  : '>' ;
GTE : ">=" ;
EQUALS : "==" ;
NOT_EQUALS  :  "<>" ;
LT     : '<' ;
LTE : "<=" ;



PARENS
options {
   testLiterals = true;
}
   : '(' | ')' ;

protected LETTER : ( 'a'..'z' | 'A'..'Z' ) ;
protected DIGIT  : '0'..'9' ;
```

```
ID
options {
   testLiterals = true;
}
   : LETTER (LETTER | DIGIT | '_')* ;

NUMBER : (DIGIT)+;




// Strings are "like this ""double quotes"" doubled to include them"
// Note that testLiterals are false so we don't have to worry about
// strings such as "if"
STRING :  '"'! ( '"' '"'! | ~('"'))*  '"'!;


WS  :   ( ' '
        | '\t'
        | '\n' { newline(); }
        | '\r'
        ) { $setType(Token.SKIP); }
    ;

COMMENT
: ("//") (' '..'~' | '\t')* WS { $setType(Token.SKIP); };




class DXParser extends Parser;
options {
   buildAST = true; // Enable AST building
   k = 2;           // Need to distinguish between ID by itself and ID ASSIGN
}

tokens {
  STATEMENTS;
  VAR_LIST;
  PARAMLIST;
  FUNCDECL;
  ARGLIST;
 }
```

```
startRule: "BEGIN" (stmts)+ "END"
          { #startRule = #([STATEMENTS], startRule); }
         ;
stmts  : exitStatement
         | returnStatement
         | ifStatement
         | while_stmt
         | printStatement
         //| ioStatement
         //| (IDENT (LPAREN|SEMI))=> procedureCallStatement
         | assignmentStatement
         | (endStatement)=> endStatement
         | variableStatement
         | funccall
         | createFunc
         | addRecFunc
         | assignmentState
         ;


builtinfunc :   createFunc
            |      addRecFunc
            |   readRecFunc
            ;

createFunc :    CREATERECORD^ "(" (STRING| expr )")";

addRecFunc  :         ADDRECORD^ "(" (STRING| expr )")";

readRecFunc :         READRECORD^ "(" (STRING| expr )? ")" ;

stmt_list
         : ( LPAREN (stmt_single | ifStatement | while_stmt)+ RPAREN)
         | stmt_single;

stmt_single
         :
         //assign_var
         //| assign_p1_m1
         //| create_rc_stmt
          printStatement
         | assignmentStatement
         | assignmentState
         | builtinfunc
         | SEMI;
```

```
variableStatement:
        "var" variableDeclarationList SEMI
        ;
variableDeclaration:
        ID (initialiser)?
        ;
variableDeclarationList:
        /*
         * SPEC:
         * variableDeclaration
         * | variableDeclarationList COMMA variableDeclaration
         */
        variableDeclaration (variableDeclarationTail)*
        ;

variableDeclarationTail:
        COMMA variableDeclaration
        ;

initialiser:
        ASSIGN expr
        ;

variableReference
  : ID
    ( LBRACKET expr RBRACKET
    | DOT ID
    )*
  ;

assignmentStatement
  : variableReference ASSIGN expr SEMI
  ;

funcDecl:
ID "("!paramlist ")"! variableDeclaration stmts END!
{ #funcDecl = #([FUNCDECL,"funcDecl"], #funcDecl); }
;

paramlist :
        (parameter (COMMA! parameter)*)?
        { #paramlist = #([PARAMLIST, "paramlist"], #paramlist); }
        ;

parameter :
```

```
        ID;

funccall:
  ID^ "("! (arglist)? ")"! SEMI!;


arglist:
(ID) (COMMA! (ID))*
{ #arglist = #([ARGLIST, "arglist"], #arglist); }
;

exitStatement
  : "exit" "when" expr
  ;

ifStatement :
        "if"^ expr "then"! stmt_list (options {greedy=true;} : "else"! stmt_list)? ;


while_stmt:
        ("WHILE"^ expr stmt_list)
   ;

 endStatement
   : "end" SEMI
   ;

printStatement
  : "print"^ "(" (expr | STRING) ")"  SEMI ;

returnStatement
  : "return" SEMI
  ;

assignmentState
        :  (accessval | "ROOTELEM"^ | "DELIM"^ | "RECNAME"^ |
"RECATTRIB"^ | "RECVAL"^ | "FILENAME"^ ) ASSIGN (expr| STRING
|builtinfunc) SEMI ;

eof
: "EOF";

expr
    //"if"^ expr "then"! expr (options {greedy=true;} : "else"! expr)?
   : ID ASSIGN^ expr
   | expr0
```

```
   | eof
   ;

expr0 : expr1 ((EQUALS^ |NOT_EQUALS^ |GT^ |GTE^ |LT^ |LTE^ ) expr1)* ;

expr1 : expr2 ( (PLUS^ | MINUS^) expr2 )* ;

expr2 : expr3 ( (TIMES^ | DIV^) expr3 )* ;

expr3
   : ID
   | "("! expr ")"!
   | NUMBER
   | MINUS^ expr3
   | keywords

   ;

keywords :  accessval|"ROOTELEM" | "DELIM" | "RECNAME" |
"RECATTRIB" | "RECVAL" | "FILENAME";

accessval : "RECVAL"^ LBRACKET NUMBER RBRACKET;
```

Main.java

```
import java.io.*;
import antlr.CommonAST;
import antlr.collections.AST;
import antlr.debug.misc.ASTFrame;

public class Main {
  public static void main(String args[]) {
    try {
      DataInputStream input = new DataInputStream(System.in);

      // Create the lexer and parser and feed them the input
      DXlexer lexer = new DXlexer(input);
      DXParser parser = new DXParser(lexer);
      parser.startRule(); // "file" is the main rule in the parser

      // Get the AST from the parser
      CommonAST parseTree = (CommonAST)parser.getAST();
```

```
    // Print the AST in a human-readable format
    System.out.println(parseTree.toStringList());

    // Open a window in which the AST is displayed graphically
    ASTFrame frame = new ASTFrame("AST from the DX parser", parseTree);
    frame.setVisible(true);

   } catch(Exception e) { System.err.println("Exception: "+e); }
  }
}
```

DX program1 written than be parsed using the above grammar (test.dx)

```
BEGIN
INPUTFILE = input.txt";
XMLFILE = “output.xml”
ROOTELEM = "address_book";
DELIM = ";" ;
RECNAME = "Address_book_entry" ;        // sets the XML record name

RECATTRIB = "First_name;Last_name;city;email;phone_no";

createRecord (RECATTRIB)                // sets the value of the XML tags
WHILE EOF
{
      RECVAL = readRecord();            // reads record from the input file
      if RECVAL[1] > 1  then            // filtering input
            addRecord (RECVAL)          // adds record to the output XML
      else
            print ( "did not add ") ;
}

RECVAL = "Tom ;Smith; Tampa; tomsmith@yahoo.com; 8133832844";
// can add any record manually not specified in input file
print ( "DONE – XML output in file output.xml");

END
```

A simple DX program2 doing some non-XML related operations

```
BEGIN
      var m = 10 , n = 12 , t = 0;
```

```
        WHILE ( m > 0 )
        {
                if n > m then
                {
                        t =  m ;
                        m = n;
                        n = t;
                }
                m = m - n;
        }
        print (n);
END
```