Chapter 5
Conversion of the Genesis 1:1-2:3 linguistic data between the XML database and the array in Visual Basic[103]

5.1 Introduction

In the electronic processing of language, one can concentrate either on the digital simulation of human understanding and language production, or on the most appropriate way to store and use existing knowledge. Both are valid and important. This thesis falls in the second category, assuming that it is important to capture the results of linguistic analyses in well-designed, exploitable, electronic databases. XML, for example, can be used to mark up free text, to create a well-structured textual database.[104] Since the data is separated from the manipulation and display thereof, the same data can be used for various purposes, and programs or queries can be created to suit the researcher's individual needs. This, however, necessitates the conversion of the data stored in XML format into a data structure, such as a threedimensional array[105], which can then be processed efficiently by a computer program[106].

This chapter will focus on the conversion of linguistic data of Genesis 1:1-2:3 between an XML data cube and a threedimensional array structure in Visual Basic 6 in order to eventually facilitate data access and manipulation. After a short reconsideration of the structures of the VB6 and XML databanks, conversion between the two will be discussed ("round-tripping"), as well as essential database functions (create, read, update and delete) that may be performed on the clause cube.

---

[103] This chapter is a revised and extended version of a short paper, "Round-tripping Biblical Hebrew linguistic data", read at the IRMA 2007 conference, Vancouver, British Columbia, Canada, May 19-23, 2007 (see Kroeze, 2007b).

[104] See Chapter 4.

[105] See Chapter 2.

[106] See Chapters 3, 6 and 7.

The XML document containing the text and mark-up of Genesis 1:1-2:3 may be regarded as a "*native XML database*" (i.e. "a database designed especially for storing XML"), while the VB6 program may be regarded as a "*content management system*" (i.e. "an application designed to manage documents and built on top of a native XML database") (Bourret, 2003). The native XML database stores the XML content, which consists of the original text (a phonetic version of the Hebrew text of Gen. 1:1-2:3) with all the added XML tags and mark-up (syntactic and semantic functions, etc.). The content management system is a database management system that operates on the data to allow editing and various views according to possible user needs. Although it is a very basic system, it does fulfil the basic requirements to qualify as a native XML database (cf. Vakali et al., 2005: 65, 67): the hierarchically-structured XML document serves "as the fundamental unit of logical storage", the schema serves as the "logical model for the XML document itself", and the XML file saved on the permanent storage device uses a sequential, text-oriented file structure as "underlying physical storage model".[107] A complete discussion on the XML clause cube may be found in Chapter 4 (cf. Kroeze 2006).

The hierarchical structure of the XML database is demonstrated by the extract shown in Figure 5.1, which partially repeats the contents of Figure 4.4 for the purpose of easy reference.

[107] According to Smiljanić et al. (2002: 17), however, a native XML database is **not** required to have the third property: it can be built on various types of databases or proprietary storage formats.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Genesis1v1-2v3>
    <clause>
        <clauseno>Gen01v01a</clauseno>
        <headers>
            <header>Level</header>
            <header>Phrase1</header>
            <header>Phrase2</header>
            <header>Phrase3</header>
            <header>Phrase4</header>
            <header>Phrase5</header>
        </headers>
        <level1>
            <leveldesc>Phon:</leveldesc>
            <phrase1>bre$it</phrase1>
            <phrase2>bara</phrase2>
            <phrase3>elohim</phrase3>
            <phrase4>et ha$amayim ve'et ha'arets</phrase4>
            <phrase5>-</phrase5>
        </level1>
        <level2>
            <leveldesc>Translation:</leveldesc>
            <phrase1>in the beginning</phrase1>
            <phrase2>he created</phrase2>
            <phrase3>God</phrase3>
            <phrase4>the heaven and the earth</phrase4>
            <phrase5>-</phrase5>
        </level2>
        <level3>
            <leveldesc>Phrase type:</leveldesc>
            <phrase1>PP</phrase1>
            <phrase2>VP</phrase2>
            <phrase3>NP</phrase3>
            <phrase4>NP</phrase4>
            <phrase5>-</phrase5>
        </level3>
        <level4>
            <leveldesc>SynF:</leveldesc>
            <phrase1>Adjunct</phrase1>
            <phrase2>Main verb</phrase2>
            <phrase3>Subject</phrase3>
            <phrase4>Object</phrase4>
            <phrase5>-</phrase5>
        </level4>
```

```
        <level5>
            <leveldesc>SemF:</leveldesc>
            <phrase1>Time</phrase1>
            <phrase2>Action</phrase2>
            <phrase3>Agent</phrase3>
            <phrase4>Product</phrase4>
            <phrase5>-</phrase5>
        </level5>
    </clause> ...
</Genesis1v1-2v3>
```

Figure 5.1. An extract of the Genesis 1:1-2:3 XML clause cube, which is representative of the hierarchy and structure of the file.

The platform independence of XML documents allows the marked-up text to be transported to other programs "capable of making sense of the tags embedded within it" (cf. Burnard, 2004). For this project, Visual Basic 6 (VB6) was chosen for this role because XML is essentially a hierarchical system that fits the threedimensional array data structure facilitated by VB6 perfectly. VB6 was chosen above Visual Basic.Net because it is easier to make an executable file for dissemination in the older version. It would, however, be relatively easy to transform the program(s) into a Visual Basic 2005 format since Visual Studio 2005 provides migration facilities. This would enable the use of pre-programmed classes, for example to extend, delete or edit the data in it. In this chapter, however, these CRUD (create, read, update and delete) functions had to be coded manually, since the size of arrays are static and do not allow automatic insertion and deletion of records (Crawford, 1999: 219).

When converted into VB6 the databank module consists of a threedimensional data structure. A multidimensional array is very suitable for a limited data set, such as the data in this project, due to its built-in indexing. Multidimensional online analytical products (MOLAP) "typically run faster than other approaches, primarily because it’s possible to index directly into the data cube’s structure to collect subsets of data" (Kay, 2004). The VB6 program discussed in this chapter and the following chapter may be regarded as a simple MOLAP tool.

The threedimensional array in VB6 contains the records of the 108 clauses found in Genesis 1:1-2:3. Each clause has five or less phrases. Each phrase has five levels of analysis. One level of analysis is added to record the verse number as primary key for reference and searching purposes (this will leave five unused data fields per clause, which may later be used for additional data). An array of 200 x 5 x 6 is used to implement this data structure. Although a size of 108 in the first dimension would be sufficient to hold all 108 clauses in the clause cube (cf. Figure 2.8), it was enlarged to 200 to allow room for appending more clauses' analyses, as discussed in section 5.3 of this chapter. If the array were populated manually with data (as was done in Chapter 2), the first clause could be coded as shown in Figure 5.2. The essential contents of Figure 2.8 is repeated in Figure 5.2 to enable readers to easily compare the two versions of the data that will be outcomes of the conversion processes discussed below.

```
Option Explicit
Public Clause(1 To 200, 1 To 5, 1 To 6) As String
Sub Main()
Clause(1, 1, 1) = "Gen01v01a"
Clause(1, 1, 2) = "bre$it"
Clause(1, 1, 3) = "in the beginning"
Clause(1, 1, 4) = "PP"
Clause(1, 1, 5) = "Adjunct"
Clause(1, 1, 6) = "Time"
Clause(1, 2, 1) = "-"
Clause(1, 2, 2) = "bara"
Clause(1, 2, 3) = "he created"
Clause(1, 2, 4) = "VP"
Clause(1, 2, 5) = "Main verb"
Clause(1, 2, 6) = "Action"
Clause(1, 3, 1) = "-"
Clause(1, 3, 2) = "elohim"
Clause(1, 3, 3) = "God"
Clause(1, 3, 4) = "NP"
Clause(1, 3, 5) = "Subject"
Clause(1, 3, 6) = "Agent"
Clause(1, 4, 1) = "-"
Clause(1, 4, 2) = "et ha$amayim ve'et ha'arets"
Clause(1, 4, 3) = "the heaven and the earth"
```

```
        Clause(1, 4, 4) = "NP"
        Clause(1, 4, 5) = "Object"
        Clause(1, 4, 6) = "Product"
        …
        End Sub
```

Figure 5.2. VB6 code that could be used to create a threedimensional array and populate one clause element with several layers of linguistic data.

A complete discussion of this structure may be found in Chapter 2 (cf. Kroeze, 2004a). The same underlying structure is used in this chapter to convert the data captured in the XML document into the VB6 array.

5.2 Conversion between VB6 and XML (round-tripping)

One of the advantages of an XML database is the separation of the data and the manipulation thereof. The same data can thus be used for various purposes, and programs or queries can be created to suit the researcher's individual needs. An XML document in itself is not very accessible for direct human inspection. Although it may be read in a simple word processor such as Notepad, the abundant use of tags poses an obstacle for human conception. One needs other software to process the data in such a repository efficiently, a tool to "bridge the gap between having a collection of structured documents and having a functional digital library" (Kumar et al., 2005: 118).[108] The VB6 program discussed in this chapter may be regarded as such a bridging tool. Another example is Petersen's (2004b) MQL query language that enables complex searches for patterns in annotated linguistic corpora such as the database of the Hebrew Bible developed by the Werkgroep Informatica (WI) at the Free University of Amsterdam.[109] However, according to Bourret (2003) "most native XML databases can only return the data as XML".

[108] <teiPublisher> is an open-source tool that aims to provide a customisable repository facilitating the dissemination of XML marked-up texts (see Kumar et al., 2005).

[109] Also, compare the description of XML-QL as a relational complete query language in Deutsch et al. (1999).

Another benefit of XML is that it provides an independent public standard and cross-platform compatibility (T. Sasaki, 2004: 19). Since XML provides a platform-independent organisation of data, conversion is often necessary to make the data accessible for algorithms that implement efficient retrieval and human-friendly interfaces (cf. Ramsay, *s.a.*). The conversion of data encoded in XML is often necessary to satisfy very specific needs identified by researchers. For example, if different linguistic layers are annotated in separate, but related, XML databanks, it is necessary to programmatically merge these data sets into Prolog facts in order to associate them in a single database (Witt, 2005: 68, 71). Conversion into a standardised format enables researchers to compare various annotated layers in order to discover relations that exist between them (cf. Bayerl et al., 2003: 165, 169). This type of data exploration activities will be discussed in Chapter 6.

If the XML data should be represented in a different, more human-readable, format, it should first be parsed by an application. In this experiment the data should be represented in an interlinear format which is more human-friendly to read. This necessitates the VB6 program to read the data into an array in order to be printed as a series of interlinear tables on the screen (cf. Chapter 3). By removing the XML tags the primary textual data is restored and the layers of analysis become much more comprehensible.

The next sections of this chapter describe the conversion of linguistic data of Genesis 1:1-2:3 between the XML data cube and a threedimensional array structure in Visual Basic 6 in order to facilitate data access and manipulation. The conversion from and to XML format is called round-tripping. Round-tripping is the circular process of storing XML data in a database and recreating the document from the database, a process which often results in a different document (Bourret, 2003). In this experiment round-tripping refers to the process of converting the Genesis 1:1-2:3 XML document to the threedimensional array structure in VB6 and saving it again in XML format. If no changes are done while the data reside in the array the second XML document should be an exact copy of the first (*ideal* round tripping - Smiljanić et al., 2002: 16). However, the array phase should facilitate updates to be made, which should be reflected in the resulting target XML document after conversion. These

CRUD facilities will be discussed towards the end of this chapter. The complete code and program may be viewed in Addendum L.

5.2.1 From XML to VB6

All data in an XML document is text (Bourret, 2003). The mark-up itself is also text only: "... markup consists of character strings carrying information about other character strings" (Huitfeldt, 2004). For a linguistic database this poses, of course, no problem since it also contains text data only. Therefore, in VB6, all the variables of the threedimensional array are also of type *string* only. The limitation of arrays that all the elements should be of the same type (string, integer, boolean, etc.), therefore, poses no problem. To strip the XML code from its tags a lot of string processing will be done (cf. Petroustos, 1999: 784-795).

An efficient way to prepare the Genesis 1:1-2:3 data for ideal round-tripping would be to ensure that empty elements (for example, where a clause has less than five noun phrases) are represented by a dash (-). The loop that reads the clause cube elements into the threedimensional array can then simply assume that the next line in the XML document will be the next element in the data structure. Not all phrases have syntactic or semantic functions and these missing elements may also be rendered by a dash. This simple implementation will be used in this experiment because this will also ensure that after ideal round-tripping the XML document is an exact copy of the original document. However, it is possible, in order to reduce file size and to save memory space, to represent null values by simply omitting these elements in the document. The conversion program will then have to evaluate the content of each line, using a selection structure (such as an if-statement) in order to ensure the correct placement in the array. This procedure causes another form of, and probably more, overhead.[110] On the VB6 side, empty elements could also be represented by zero-length string values in the array variables. To avoid problems

[110] The XML schema, discussed in Chapter 4, can only check the validity of data recorded in the XML file. Since absent elements are valid, another mechanism is needed to ensure correct conversion of such elements from the XML file into the threedimensional array.

during advanced array processing due to the null values the whole array may first be populated with dashes (as symbol of an empty element) which are then partly overwritten when the data is read in from the XML document. This will ensure that all empty elements (or yet unused spaces in the array reserved for new clauses to be appended) contain dashes.

Before the data is converted an algorithm is used to count the number of clauses appearing in the XML file, and the result is stored in variables called *countclauses* and *maxArray*. The last-mentioned variable is used to limit processing in the rest of the VB6 program to real data only (ignoring empty clause elements), and, therefore, its value should be adjusted when clauses are added or deleted during the array phase.

An extract of the code for this part of the program is shown in Figure 5.3. It is assumed that all variables have been declared.

```
'Read XML file from disk into array
Public Sub Command1_Click()

'Initialise all array elements with empty element symbols
For iniArr1 = 1 To 200
 For iniArr2 = 1 To 5
  For iniArr3 = 1 To 6
   Clause(iniArr1, iniArr2, iniArr3) = "-"
  Next
 Next
Next

'Count number of clauses in the XML cube:

arrayMax = 0 'Reset total number of clauses in array
countclauses = 0 'Reset counter that counts number of clauses in XML file
filenum1 = FreeFile
Open "Gen1_InputV15_RT1.xml" For Input As #filenum1
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
```

```
While Not EOF(filenum1)
   Line Input #filenum1, tempLine
   countclauses = countclauses + 1
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
```

```
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
Wend
MsgBox ("There are " & countclauses & " clauses in the XML cube")
arrayMax = countclauses
Close #filenum1
```

```
'Populate array with data from XML file:
Open "Gen1_InputV15_RT1.xml" For Input As #filenum1

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

For count1 = 1 To arrayMax

   Line Input #filenum1, tempLine

   Line Input #filenum1, tempLine
   Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
       tempLine)
   Clause(count1, 1, 1) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine
   Line Input #filenum1, tempLine

   Line Input #filenum1, tempLine
   Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
       tempLine)
   Clause(count1, 1, 2) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

   Line Input #filenum1, tempLine
   Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
```

```
        tempLine)
    Clause(count1, 2, 2) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 3, 2) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 4, 2) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 5, 2) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Line Input #filenum1, tempLine
    Line Input #filenum1, tempLine

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 1, 3) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 2, 3) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 3, 3) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 4, 3) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 5, 3) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)
```

```
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 1, 4) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 2, 4) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 3, 4) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 4, 4) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 5, 4) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Line Input #filenum1, tempLine
Line Input #filenum1, tempLine

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 1, 5) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
    tempLine)
Clause(count1, 2, 5) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

Line Input #filenum1, tempLine
Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
```

```
        tempLine)
    Clause(count1, 3, 5) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 4, 5) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 5, 5) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Line Input #filenum1, tempLine
    Line Input #filenum1, tempLine

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 1, 6) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 2, 6) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 3, 6) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 4, 6) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

    Line Input #filenum1, tempLine
    Call DecodeXML(XMLstringBeginPos, XMLstringEndPos, XMLstringLength,
        tempLine)
    Clause(count1, 5, 6) = Mid(tempLine, XMLstringBeginPos, XMLstringLength)

 Line Input #filenum1, tempLine
 Line Input #filenum1, tempLine
Next
```

```
Close #filenum1
arrayflag = True
MsgBox ("XML cube Gen1_InputV15_RT1.xml converted to array in RAM")
count1 = 1
Call ShowArray

End Sub
```

```
' Function used to strip XML tags before inserting data into array
Public Sub DecodeXML(XMLstringBeginPos2 As Integer, XMLstringEndPos2 As
   Integer, XMLstringLength2 As Integer, templine2 As String)
 XMLstringBeginPos2 = InStr(templine2, ">") + 1
 XMLstringEndPos2 = InStrRev(templine2, "<")
 XMLstringLength2 = XMLstringEndPos2 - XMLstringBeginPos2
End Sub
```

Figure 5.3. VB6 code used to convert linguistic data from XML format into a threedimensional array.

Although validation of the XML file is usually done by means of a schema, it may again be done during the array state. In a subroutine the tags could first be stripped and selected data tested against a standardised list of valid entries (for example, syntactic and semantic functions). If the data does not conform to these values, an error message should be shown. The user should use this functionality to correct the data before any further processing takes place. In this project the validation of phrases, and syntactic and semantic functions has been done by means of a schema (see Chapter 4), and syntactic and semantic functions will again be validated during the advanced processing phase (see Chapter 6). Therefore, this procedure has been omitted from the version of the program available in Addendum L, and only an example of validation code for some syntactic functions is shown in Figure 5.4.

```
Private Sub cmdCleanData_Click() 'Clean data - syntactic functions
Dim count11, count12, count13 As Integer
Dim arrsyn1(108, 5, 6) As String
For count11 = 1 To 108 'Copy array for validation purposes
 For count12 = 1 To 5
  For count13 = 1 To 6
   arrsyn1(count11, count12, count13) = Clause(count11, count12, count13)
  Next
```

```
 Next
Next

For count11 = 1 To 108 'Check syntactic functions
 For count12 = 1 To 5
  For count13 = 5 To 5 'Check only syntactic function dimension
  'Extend this scheme to include all possible syntactic functions
  'Here limited to those functions occurring in Gen 1:1-2:3
   If arrsyn1(count11, count12, 5) <> "Main verb" And _
   arrsyn1(count11, count12, 5) <> "Copulative verb" And _
    arrsyn1(count11, count12, 5) <> "Subject" And _
    arrsyn1(count11, count12, 5) <> "Object" And _
      arrsyn1(count11, count12, 5) <> "Object clause" And _
      arrsyn1(count11, count12, 5) <> "Object cluster" And _
      arrsyn1(count11, count12, 5) <> "IndObj" And _
      arrsyn1(count11, count12, 5) <> "Complement" And _
      arrsyn1(count11, count12, 5) <> "Copula-predicate" And _
      arrsyn1(count11, count12, 5) <> "Adjunct" And _
      arrsyn1(count11, count12, 5) <> "Disjunct" And _
      arrsyn1(count11, count12, 5) <> "Attribute" And _
      arrsyn1(count11, count12, 5) <> "Conj" And _
      arrsyn1(count11, count12, 5) <> "-" Then
          'User must clean data if following message is shown:
     MsgBox ("Synf " & arrsyn1(count11, count12, 5) & " in vs " & _
        arrsyn1(count11, 1, 1) & " is invalid")
   End If
   Next
 Next
Next
End Sub
```

Figure 5.4. Example of VB6 code that could be used to validate syntactic function elements during the array state.

To show the contents of the array, the elements of each clause are displayed in a series of textboxes and labels, simulating an interlinear rendering. The code used is shown in Figure 5.5.

```
 Public Sub ShowArray() 'Procedure used to display current clause on
   interface
 txtFind.Text = count1 'count1 is the array index of each clause
 txtC111.Text = Clause(count1, 1, 1)
```

```
 txtC112.Text = Clause(count1, 1, 2)
 txtC122.Text = Clause(count1, 2, 2)
 txtC132.Text = Clause(count1, 3, 2)
 txtC142.Text = Clause(count1, 4, 2)
 txtC152.Text = Clause(count1, 5, 2)
 txtC113.Text = Clause(count1, 1, 3)
 txtC123.Text = Clause(count1, 2, 3)
 txtC133.Text = Clause(count1, 3, 3)
 txtC143.Text = Clause(count1, 4, 3)
 txtC153.Text = Clause(count1, 5, 3)
 txtC114.Text = Clause(count1, 1, 4)
 txtC124.Text = Clause(count1, 2, 4)
 txtC134.Text = Clause(count1, 3, 4)
 txtC144.Text = Clause(count1, 4, 4)
 txtC154.Text = Clause(count1, 5, 4)
 txtC115.Text = Clause(count1, 1, 5)
 txtC125.Text = Clause(count1, 2, 5)
 txtC135.Text = Clause(count1, 3, 5)
 txtC145.Text = Clause(count1, 4, 5)
 txtC155.Text = Clause(count1, 5, 5)
 txtC116.Text = Clause(count1, 1, 6)
 txtC126.Text = Clause(count1, 2, 6)
 txtC136.Text = Clause(count1, 3, 6)
 txtC146.Text = Clause(count1, 4, 6)
 txtC156.Text = Clause(count1, 5, 6)
End Sub
```

Figure 5.5. VB6 code used to display one clause's linguistic analysis in a series of textboxes and labels on the interface.

When the user presses the "Read XML file from disk into array" button, the conversion is done and the first clause's data is displayed on the interface (see Figure 5.6). The array content is displayed, clause by clause, in a series of text boxes and labels to simulate an interlinear rendering, similar to the versions presented in Chapters 3 and 4.

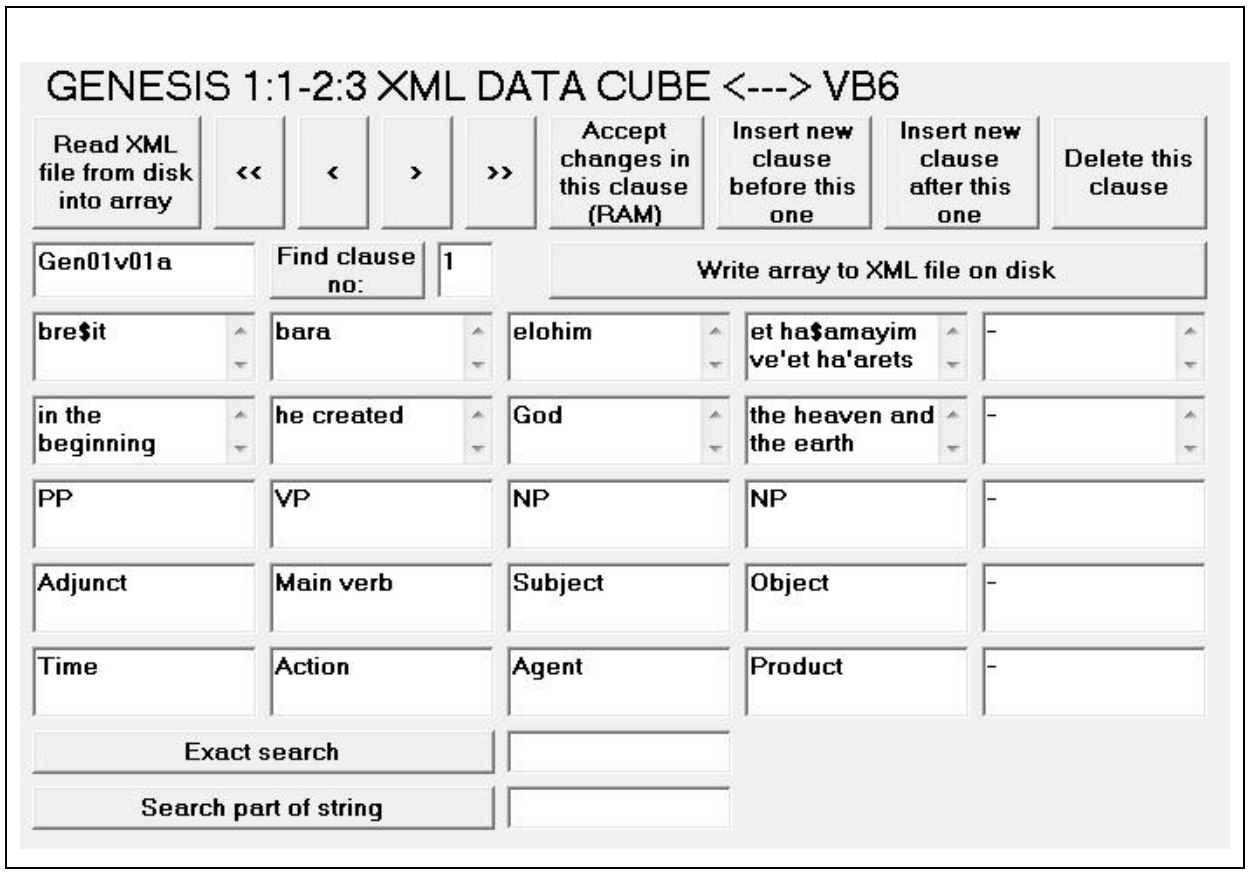

Figure 5.6. The end-result after converting data from the XML clause cube into a threedimensional array in VB6.

The code in Figure 5.7 is used to scroll through the data. The user may use the next (>) and previous (<) buttons to view the data clause by clause, or they may go directly to the first (<<) or last element (>>). The program also enables rolling over from the last element to the first and *vice versa*.

```
Private Sub btnFirst_Click(Index As Integer) 'Move to the first element in
   the array

count1 = 1
Call ShowArray

End Sub
```

```
Private Sub btnPrev_Click(Index As Integer) 'Move to the previous element
   in the array
```

```
If count1 = 1 Then
 count1 = (arrayMax + 1)
End If
count1 = count1 - 1
Call ShowArray

End Sub
```

```
Private Sub btnNext_Click(Index As Integer) 'Move to the next element in
   the array

If count1 = arrayMax Then
 count1 = 0
End If
count1 = count1 + 1
Call ShowArray

End Sub
```

```
Private Sub btnLast_Click(Index As Integer) 'Move to the last element in
   the array

count1 = arrayMax
Call ShowArray

End Sub
```

Figure 5.7. VB6 code used to scroll through the clause cube data.

The code in Figure 5.8 is used to display a required clause, the array index of which is shown in the clause number textbox.

```
Private Sub btnFind_Click() 'Show clause of clause index shown in textbox
   "Find clause no"

count1 = txtFind.Text
 If count1 > arrayMax Or count1 < 1 Then
  MsgBox ("Invalid clause no")
  Exit Sub
 Else
```

```
 Call ShowArray
 End If

End Sub
```

Figure 5.8. VB6 code used to display a required clause using its array index.

The code in Figure 5.9 is used to facilitate exact searches; for example, the user may enter "Gen01v07a" and click on the "Exact search" button to move directly to the eighteenth clause. If a parameter is used that appears more than once all clauses containing the parameter are shown one-by-one, paused by a message box. The parameter must match the searched item exactly, but it is not case sensitive.

```
Private Sub btnSearch_Click() 'Exact search

count1 = 1
flagSrch = 0
If txtSearch.Text = "" Then
 Exit Sub
End If

For countSrch1 = 1 To arrayMax
 For countSrch2 = 1 To 5
  For countSrch3 = 1 To 6
   If StrComp(Clause(countSrch1, countSrch2, countSrch3), txtSearch.Text,
      1) = 0 Then
    flagSrch = 1
    count1 = countSrch1
    Call ShowArray
    MsgBox ("Click OK to search next")
   End If
  Next
 Next
Next

If flagSrch = 0 Then
 MsgBox ("Not found")
 Else
  MsgBox ("End of data cube reached")
```

```
End If

End Sub
```

Figure 5.9. VB6 code used to perform exact searches.

Users who are not acquainted with the sets of phrase types, semantic and syntactic functions used, may need a facility to do "fuzzy" searches. The "Search part of string" button enables one to type any part of a string to be searched within the elements; for example, one may enter "Ben" to find instances of the semantic function of Beneficiary; however, in addition to the required clauses, other clauses containing the Hebrew word *ben* in the phonological rendering will also be shown. The code in Figure 5.10 is used to do searches on parts of strings in the clause cube.

```
Private Sub cmdSearchPart_Click() 'Fuzzy search

count1 = 1
flagSrch = 0
If txtSearchPart.Text = "" Then
 Exit Sub
End If

For countSrch4 = 1 To arrayMax
 For countSrch5 = 1 To 5
  For countSrch6 = 1 To 6
   pos = InStr(1, Clause(countSrch4, countSrch5, countSrch6),
      txtSearchPart.Text, 1)
   If pos > 0 Then
    flagSrch = 1
    count1 = countSrch4
    Call ShowArray
    MsgBox ("Click OK to search next")
   End If
  Next
 Next
Next

If flagSrch = 0 Then
 MsgBox ("Not found")
```

```
Else
 MsgBox ("End of data cube reached")
End If

End Sub
```

Figure 5.10. VB6 code used to perform searches on parts of strings.

After discussing conversion from the array in VB6 back to an XML file in the following paragraph, more CRUD functionalities will be discussed. These procedures should also take place while the cube resides in the computer's RAM (random access memory) during its array phase.

5.2.2 From VB6 to XML

The conversion of the content of the threedimensional array in VB6 into the XML clause cube is more or less the reversal of the above process. Assuming that no updates have been done, it is of course not necessary to do validation again, but string processing will again be used to convert the variables to lines of text wrapped in applicable XML tags. The structure of the XML schema must strictly be adhered to in order to create a file that can again be read into VB6 using the same algorithm. In order to keep the original data intact the current date and time may be added to the name of the output file so that a different XML file is created each time when the button "Write array to XML file on disk" is pressed. If one wants to accept and store edited data permanently, the output file should have the same name as the input file. In the empirical experiment of this chapter, a copy of the XML clause cube was used for this purpose (see Gen1_InputV15_RT1.xml in Addendum L). Figure 5.11 shows the code that is used to write the linguistic data from the array into the XML clause cube on disk.

```
Public Sub Command3_Click() 'Write array to XML file on disk

If arrayflag = False Then
 MsgBox ("Array is empty - not saved")
 Exit Sub
End If

filenum2 = FreeFile

'Create unique output file name (optional):
'outputname = "Gen1V15_Output_" & Format(Now, "yyyymmddhhmmss") & ".xml"

outputname = "Gen1_InputV15_RT1.xml"

Open outputname For Output As #filenum2
Print #filenum2, "<?xml version='1.0' encoding='UTF-8' standalone='yes' ?>"
Print #filenum2, "<?xml-stylesheet type='text/css'
   href='Gen1XMLdb03c.css'?>"
Print #filenum2, "<Genesis1v1-2v3>"

For count1 = 1 To arrayMax
 Print #filenum2, "   <clause>"
 Print #filenum2, "      <clauseno>" & Clause(count1, 1, 1) & "</clauseno>"
 Print #filenum2, "      <headers>"
 Print #filenum2, "         <header>Level</header>"
 Print #filenum2, "         <header>Phrase1</header>"
 Print #filenum2, "         <header>Phrase2</header>"
 Print #filenum2, "         <header>Phrase3</header>"
 Print #filenum2, "         <header>Phrase4</header>"
 Print #filenum2, "         <header>Phrase5</header>"
 Print #filenum2, "      </headers>"
 Print #filenum2, "      <level1>"
 Print #filenum2, "         <leveldesc>Phon:</leveldesc>"
 Print #filenum2, "         <phrase1>" & Clause(count1, 1, 2) &
    "</phrase1>"
 Print #filenum2, "         <phrase2>" & Clause(count1, 2, 2) &
    "</phrase2>"
 Print #filenum2, "         <phrase3>" & Clause(count1, 3, 2) &
    "</phrase3>"
 Print #filenum2, "         <phrase4>" & Clause(count1, 4, 2) &
    "</phrase4>"
 Print #filenum2, "         <phrase5>" & Clause(count1, 5, 2) &
    "</phrase5>"
```

```
Print #filenum2, "      </level1>"
Print #filenum2, "      <level2>"
Print #filenum2, "         <leveldesc>Translation:</leveldesc>"
Print #filenum2, "         <phrase1>" & Clause(count1, 1, 3) &
   "</phrase1>"
Print #filenum2, "         <phrase2>" & Clause(count1, 2, 3) &
   "</phrase2>"
Print #filenum2, "         <phrase3>" & Clause(count1, 3, 3) &
   "</phrase3>"
Print #filenum2, "         <phrase4>" & Clause(count1, 4, 3) &
   "</phrase4>"
Print #filenum2, "         <phrase5>" & Clause(count1, 5, 3) &
   "</phrase5>"
Print #filenum2, "      </level2>"
Print #filenum2, "      <level3>"
Print #filenum2, "         <leveldesc>Phrase type:</leveldesc>"
Print #filenum2, "         <phrase1>" & Clause(count1, 1, 4) &
   "</phrase1>"
Print #filenum2, "         <phrase2>" & Clause(count1, 2, 4) &
   "</phrase2>"
Print #filenum2, "         <phrase3>" & Clause(count1, 3, 4) &
   "</phrase3>"
Print #filenum2, "         <phrase4>" & Clause(count1, 4, 4) &
   "</phrase4>"
Print #filenum2, "         <phrase5>" & Clause(count1, 5, 4) &
   "</phrase5>"
Print #filenum2, "      </level3>"
Print #filenum2, "      <level4>"
Print #filenum2, "         <leveldesc>SynF:</leveldesc>"
Print #filenum2, "         <phrase1>" & Clause(count1, 1, 5) &
   "</phrase1>"
Print #filenum2, "         <phrase2>" & Clause(count1, 2, 5) &
   "</phrase2>"
Print #filenum2, "         <phrase3>" & Clause(count1, 3, 5) &
   "</phrase3>"
Print #filenum2, "         <phrase4>" & Clause(count1, 4, 5) &
   "</phrase4>"
Print #filenum2, "         <phrase5>" & Clause(count1, 5, 5) &
   "</phrase5>"
Print #filenum2, "      </level4>"
Print #filenum2, "      <level5>"
Print #filenum2, "         <leveldesc>SemF:</leveldesc>"
Print #filenum2, "         <phrase1>" & Clause(count1, 1, 6) &
   "</phrase1>"
Print #filenum2, "         <phrase2>" & Clause(count1, 2, 6) &
   "</phrase2>"
```

```
 Print #filenum2, "         <phrase3>" & Clause(count1, 3, 6) &
    "</phrase3>"
 Print #filenum2, "         <phrase4>" & Clause(count1, 4, 6) &
    "</phrase4>"
 Print #filenum2, "         <phrase5>" & Clause(count1, 5, 6) &
    "</phrase5>"
 Print #filenum2, "      </level5>"
 Print #filenum2, "   </clause>"
Next

Print #filenum2, "</Genesis1v1-2v3>"
Close #filenum2
MsgBox ("Array converted to XML and saved as " & outputname)

End Sub
```

Figure 5.11. VB6 code used to save clause cube data from the threedimensional array into permanent XML-formatted storage.

The first version of the XML file for this study was, actually, created in a similar way. The original data was written as code in a module of a VB6 program that creates and populates a threedimensional array with the clause cube data (see Chapter 2). Empty elements were not marked by a dash or other symbol, implying that the array contained null values in those variables. To fill up the array with symbols representing empty values, a for-loop was used, first of all, to populate the whole array with dashes, after which parts of the array were overwritten by those elements that do exist. The array was then converted into an XML file using the same set of code as the lines discussed above (the structure of the XML file is discussed in detail in Chapter 4).

5.3 Editing the data in the clause cube

Reading the data requires a procedure that displays the clause cube data as a set of twodimensional tables (see 5.2.1 above). For read-only purposes, this functionality, combined with the search functions discussed above, should be sufficient. However, it is very likely that some users would need the opportunity to add more clauses to

the clause cube, to edit existing data, or even to delete records. Array-like functionalities make these types of operations relatively easy, especially if predefined functions exist which may be called, such as those available in collections or arraylists. However, in this experiment, basic original code was written to facilitate full CRUD since arrays' sizes in VB6 are static and cannot grow or shrink automatically.[111] This approach may be regarded as an example of "creative programming techniques" that may be used to overcome the limitations of simple arrays (cf. Crawford, 1999: 219).[112]

In order to add clauses to the database, the size of the primary dimension of the cube had to be enlarged to make room for the required number of extra clauses. This was done by changing the declaration of the threedimensional array. In this program, the size was changed from 108 to 200, thus making space for 92 more clauses.

New records may be inserted either before or after the current clause. When the user identifies the location after which another clause should be inserted (see the "Insert new clause after this one" button on Figure 5.6), all the clause elements following this location in the computer's memory should be moved one place down to free the current set of variables for a new clause's data to be recorded. If the new clause must be inserted before the current one (see the "Insert new clause before this one" button on Figure 5.6), the current clause must also be moved one position down the array.

The code used to create space for a new clause record preceding the one currently displayed on the interface is shown in Figure 5.12.

---

[111] "In most programming languages, conventional arrays have a fixed size–they cannot grow or shrink dynamically to conform to an application's execution-time memory requirements" (Deitel & Deitel, 2006: 1321). Dynamic arrays in VB6 is not an option since only the last dimension may be changed without losing existing data. Even if the Preserve keyword is used to maintain a multidimensional array's contents when it is resized (using the ReDim statement), only the last dimension may be changed (Petroustos, 1999: 769).

[112] Grow and shrink functionalities are facilitated by means of array lists in Visual Basic 2005, a fully object-oriented language (Deitel & Deitel, 2006: 1321; MacDonald, 2006: 207; Foxall, 2006: 77). Visual Basic 6 offers the use of collections instead (Crawford, 1999: 219-224).

```
Private Sub Command4_Click() 'Insert new clause before current one into
   array

arrayMax = arrayMax + 1

For countAddRec = (arrayMax - 1) To count1 Step -1
 countAddRec2 = countAddRec + 1
 For count4 = 1 To 5
  For count5 = 1 To 6
   Clause(countAddRec2, count4, count5) = Clause(countAddRec, count4,
      count5)
  Next
 Next
Next

For count4 = 1 To 5 'Clear new element in array
 For count5 = 1 To 6
   Clause(count1, count4, count5) = "-"
 Next
Next

Call ShowArray

End Sub
```

Figure 5.12. The VB6 code used to make space for a new clause record to precede the current one.

Figure 5.13 shows the code used to insert a new, empty clause record set following the current one.

```
Private Sub Command7_Click() 'Insert new clause after current one in array

arrayMax = arrayMax + 1
count1 = count1 + 1

For countAddRec = (arrayMax - 1) To (count1) Step -1
 countAddRec2 = countAddRec + 1
```

```
 For count4 = 1 To 5
  For count5 = 1 To 6
   Clause(countAddRec2, count4, count5) = Clause(countAddRec, count4,
count5)
  Next
 Next
Next

For count4 = 1 To 5 'Clear new element in array
 For count5 = 1 To 6
   Clause(count1, count4, count5) = "-"
 Next
Next

Call ShowArray

End Sub
```

Figure 5.13. The VB6 code used to make space for a new clause record to follow the current one.

After the existing records have been moved down the array, old, redundant data in the freed space is overwritten with dashes and the new, empty record is shown on the screen. The user may now enter the new clause data here. After the user has typed the new information on the usual interface, he/she may press a button ("Accept changes in this clause (RAM)") to save the new data into the array in the RAM. If he/she is satisfied that all the information is correct, he/she should press another button ("Write array to XML file on disk") to save the information to the target XML-file (see Figure 5.6 above).

Tagging mistakes may be corrected by directly changing the information shown on the display and by saving the updates both to the RAM for immediate use and to the XML file for permanent storage. The same code is used to save new or updated data to the RAM (see Figure 5.14).

```
Private Sub Command2_Click() 'Accept changes in this clause (RAM)
   Clause(count1, 1, 1) = txtC111.Text
   Clause(count1, 1, 2) = txtC112.Text
   Clause(count1, 2, 2) = txtC122.Text
   Clause(count1, 3, 2) = txtC132.Text
   Clause(count1, 4, 2) = txtC142.Text
   Clause(count1, 5, 2) = txtC152.Text
   Clause(count1, 1, 3) = txtC113.Text
   Clause(count1, 2, 3) = txtC123.Text
   Clause(count1, 3, 3) = txtC133.Text
   Clause(count1, 4, 3) = txtC143.Text
   Clause(count1, 5, 3) = txtC153.Text
   Clause(count1, 1, 4) = txtC114.Text
   Clause(count1, 2, 4) = txtC124.Text
   Clause(count1, 3, 4) = txtC134.Text
   Clause(count1, 4, 4) = txtC144.Text
   Clause(count1, 5, 4) = txtC154.Text
   Clause(count1, 1, 5) = txtC115.Text
   Clause(count1, 2, 5) = txtC125.Text
   Clause(count1, 3, 5) = txtC135.Text
   Clause(count1, 4, 5) = txtC145.Text
   Clause(count1, 5, 5) = txtC155.Text
   Clause(count1, 1, 6) = txtC116.Text
   Clause(count1, 2, 6) = txtC126.Text
   Clause(count1, 3, 6) = txtC136.Text
   Clause(count1, 4, 6) = txtC146.Text
   Clause(count1, 5, 6) = txtC156.Text
End Sub
```

Figure 5.14 The VB6 code used to save new or edited clause data to the RAM.

If the user wants to delete a whole clause's data, he/she should be able to press a button to activate a procedure that removes the data of the clause currently shown on the display (see the "Delete this clause" button on Figure 5.6 above). The related set of clause variables is removed by moving all the following clauses' data one position up in the primary dimension, and by clearing the last element's data that is now duplicated in the second-last position (see Figure 5.15). These changes should also be saved to the RAM and the target XML file.

```
Private Sub Command5_Click() 'Delete this clause

For countDelRec = count1 To (arrayMax - 1)
 countDelRec2 = countDelRec + 1
 For count2 = 1 To 5
  For count3 = 1 To 6
   Clause(countDelRec, count2, count3) = Clause(countDelRec2, count2,
      count3)
  Next
 Next
Next

For count2 = 1 To 5 'Clear last element in array
 For count3 = 1 To 6
  Clause(arrayMax, count2, count3) = "-"
 Next
Next

If count1 = arrayMax Then
 count1 = count1 - 1
End If

arrayMax = arrayMax - 1

Call ShowArray

End Sub
```

Figure 5.15. The VB6 code used to delete a clause record.

The procedure used to save all current data in the array from the RAM to the target XML file for permanent storage and recovery has already been discussed in 5.2.2 (see Figure 5.11).

Adding these CRUD functions to the program significantly enhances its functionalities by facilitating basic database procedures to create, delete and maintain data. Even though it could not be called "ideal round-tripping" anymore (since the contents of the source and target XML files differ), in practice, this scenario is preferable for an

environment where the database is populated, corrected and expanded. In a situation where end-users should not be able to change the data, these functionalities should, of course, not be offered. The CRUD functionality currently is the only way to extend the database, manually and clause-by-clause, to larger parts of the Hebrew Bible. A more elegant solution would be to import existing data, but this need creates new challenges that fall outside the scope of this thesis.

5.4 Conclusion

In this chapter, a synthesis was found between two separate concepts discussed in earlier chapters of the thesis. After a review of the essential concepts of building a clause cube, either by using a threedimensional array in VB6, or a hierarchically structured XML file, a method was proposed to convert the linguistic data between these two formats. "Ideal" round-tripping was implemented by means of string processing to either strip or wrap the primary data in XML tags. This enabled the transformation of the data from permanent storage into a temporary threedimensional array in the computer's RAM, and *vice versa*. Round-tripping enables one to overcome the limitations of both the array and XML phases. Using only an array does not allow permanent storage, while viewing the data simply by using an XML style sheet does not allow advanced processing. By using both phases the data is stored elegantly and permanently as an XML file, while some of the functionalities of array processing, like searching and scrolling through the multidimensional clause representations, discussed in Chapters 2 and 3, were re-introduced into the consolidated system. Slicing and dicing will also be integrated into the text-mining functions discussed in Chapter 6.

Various viewing and searching functions have been discussed. In addition, create, update and delete functionalities were added to enable users to populate and edit the clause cube while it is in the array state and to save these updates both to the RAM and on permanent storage in XML format. One may conclude that these technologies are suitable for the efficient storage, transfer and processing of linguistic data. Since all essential database functionalities are now possible, the created software may be

regarded as a humanities-oriented information system. In the following chapters, advanced processing and visualisation of this data will be discussed.