Compilerbau in der Praxis

Compilerentwicklung anhand eines einfachen Beispiels

- von Stefan Tappertzhofen -

Inhalt

Teil 1 - Einführung

Zu diesem Tutorial

In diesem Tutorial werde ich versuchen die Komplexität der Compilerentwicklung anhand eines einfachen Compilers praxisorientiert zu reduzieren. Ziel der Serie „Compilerbau in der Praxis“ soll es sein einen rudimentären BASIC Compiler „eBas“ („educational Basic“) zu entwickeln, der in der Lage ist die Quellsprache BASIC in die Zielsprache Assembler umzuformen.

Der Entwicklung von Compilern sind in der Informatik eine Reihe von Fachbüchern gewidmet, die zum größten Teil für den Laien kaum zu verstehen sind. Das mag aber auch daran liegen, dass der Compilerbau nicht grade die einfachste Disziplin der Computerwissenschaften ist. Auf Grund dieser Tatsache muss ich auf eine Reihe von fundamentalen Informationen zu diesem Thema verzichten. Ich werde so zum Beispiel die genau Festlegung der Syntax, so wie Sie im „Compilerbau I“ Buch beschrieben ist nicht aufführen, sondern nur einen kurzen Überblick über die Syntax von eBas geben.

Syntax Definition von eBas

Bevor wir mit der eigentlichen Programmierung des Compilers beginnen müssen wir wissen was der Compiler alles verstehen muss. Der Compiler ist deswegen immer sehr stark an die Syntax und die Definitionen der Programmiersprache gebunden. Sie werden feststellen, dass eBas aus sehr einfachen „Bestandteilen“ zusammengesetzt ist.

Zuweisungen

$$[VAR] = [VAR|NUM]_1 ([OP]_1 [VAR|NUM]_2 (\ldots) [OP]_{n-1} [VAR|NUM]_n)$$

VAR = Variable
NUM = Zahl
OP = Operator [“+” | “-“ | “*” | “/” | “(“ | “)”]
[VAR|NUM] = Variable oder Zahl

Dieser Term besteht aus maximal *n* [VAR|NUM], wobei diese immer durch *n-1* [OP] verknüpft sind. Der Teil der Zuweisung hinter dem Gleichheitszeichen wird im Verlauf weiter als „Term“ bezeichnet

Anweisungen

(IF [TERM] = [TERM] THEN) [[PRINT STM]\[GOTO STM]\[ZUWEISUNG]]

Es handelt sich hierbei um die allgemeine Notation einer Anweisung. Optional kann diese Anweisung selektiv durchgeführt werden, wenn Sie in Form einer IF-Abfrage dargestellt wird. Wie Sie erkennen können besteht eine Anweisung aus dem PRINT oder GOTO Statement (= „STM“) oder einer Zuweisung.

Für eine einfache DO ... LOOP Schleife (wobei LOOP keine Abfrage von Gültigkeiten zum weiteren Ausführen der Schleife haben soll) benötigen wir eine weitere Notationsregel:

DO
[ANWEISUNG|ZUWEISUNG|DO-SCHLEIFE|LABEL]$_1$
(...)
[ANWEISUNG|ZUWEISUNG|DO-SCHLEIFE|LABEL]$_n$

LOOP

Aus dieser Syntaxregel kann man entnehmen, dass eine DO ... LOOP Schleife als Block geschrieben werden muss, wobei die *n* Zeilen des Blocks entweder jeweils aus Anweisungen, Zuweisungen, weiteren DO-SCHLEIFEN oder Labels (Sprungmarken) bestehen.

Zur Vollständigkeit müssen wir noch die Notation des PRINT und GOTO Befehls festlegen:

PRINT [TERM|STR]$_1$ ([OP]$_1$ [TERM|STR]$_2$ (…) [OP]$_{n-1}$ [TERM|STR]$_n$)

STR = String in der Form von "Dies ist ein String"
OP = Operator in der Form von „&“

GOTO [LABEL]

LABEL = Name der Sprungmarke (ohne Hash)

Labels und Definition von Variablen

#[Labelname]

Die Definition eines Labels (Sprungmarke) wird mit dem Raute Zeichen eingeleitet, worauf der eindeutige, nur einmal verwendete Labelname folgt (es darf sich auch nicht um den Namen einer Variable oder um ein Schlüsselwort handeln). Ein Labelname darf aus Zahlen, Buchstaben und dem „_“ zusammengesetzt sein, wobei ein Buchstabe immer zu Anfang stehen muss.

DIM [Variablenname]

Variablenname ist ein eindeutig festgelegter Name einer Variablen (es darf sich auch nicht um den Namen eines Labels, Variablen oder um ein Schlüsselwort handeln). Ein Variablenname darf aus Zahlen, Buchstaben und dem „_“ zusammengesetzt sein,

wobei ein Buchstabe immer zu Anfang stehen muss. Die Größe einer Variablen beträgt immer 32 Bit.

Einfaches Beispiel in eBas

Obwohl es in der einschlägigen Literatur nicht üblich ist Beispiele der Sprache zu zeigen, über die diskutiert wird will ich hier doch ein einfaches Beispiel zeigen:

```
PRINT "Hello World"


DIM i
i = i + 1
IF i = 5 THEN PRINT "i ist gleich " & i


DIM i
i = 1
DO
      i = i + 1
      IF i = 5 THEN GOTO Ende
LOOP
#Ende
PRINT "Ende"
```

Wie schon erwähnt ist die Sprache eBas sehr einfach und vermutlich in der Praxis kaum zu verwenden. Würde ich hier aber skizzieren wie man einen Compiler für eine moderne, wohlmöglich noch objektorientierte Sprache erstellt würde dies den Rahmen des Tutorials sprengen.

Im nächsten Teil werde ich mit der Entwicklung des Compilers beginnen. Dabei werde ich zunächst auf die Bestandteile wie Scanner und Parser eingehen.

Teil 2 – Einfacher Scanner und Parser

Wahl der Programmiersprache für den Compiler

Bevor wir mit der Programmierung anfangen können möchte ich noch kurz auf die Wahl die Programmiersprache eingehen, die ich hier verwende. Es handelt sich dabei um Visual Basic 6, da es besonders leicht zu programmieren und komfortabel ist. Anstatt wie es üblich ist Pseudo- Code zu verwenden ist es meiner Meinung nach schon ausreichend ein VB-Programm zu nennen.

Generell kann man aber mit jeder Programmiersprache, die Zeichenketten verarbeiten kann einen Compiler schreiben.

Ein einfacher Scanner für eBas

Damit der Compiler den Quellcode in die Zielsprache umsetzten kann benötigt er eine Methode, die den Quellcode in lesbare Symbole umwandelt. Ein Scanner verfügt über die jeweiligen Methoden und Funktionen um solche Symbole (oft auch „Tokens“ genannt) zu erstellen. Dabei ist es natürlich von der definierten Sprache her abhängig wie die Symbole des Quellcodes umgesetzt werden sollen. In unserem eBas Beispiel wäre beispielsweise das erkennen des Symbols „i++“ als Inkrementierung der Variablen *i* völlig nutzlos, da dies ja nicht in unseren Syntaxregeln enthalten ist.

Um den Scanner zu testen soll uns zunächst das Beispielprogramm:

```
DIM i
i = i + 1 * 2
```

genügen.

Ein einfacher Scanner für dieses Programm könnte so aussehen:

```
Option Explicit

Sub Main()

    On Error GoTo ErrHandle

    Dim Zeile As String
    Dim n As Integer, intLine As Integer
    Dim Tokens

    intLine = 1

    Open "C:\test.eb" For Input As #1
    Do

        Line Input #1, Zeile
        Tokens = Scan(Zeile)
```

```
        For n = 0 To UBound(Tokens)
            Debug.Print "Zeile " & intLine & ", Token " & (n + 1) & ": " _
                  & Tokens(n)
        Next n

        intLine = intLine + 1

    Loop Until EOF(1)
    Close #1

    Exit Sub

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"
    End
End Sub

Function Scan(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Scan = Split(strLine, " ")

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler beim Scannen"
    End
End Function
```

Im Direktfenster der Visual Basic IDE sollte nun folgendes stehen:

```
Zeile 1, Token 1: DIM
Zeile 1, Token 2: i
Zeile 2, Token 1: i
Zeile 2, Token 2: =
Zeile 2, Token 3: i
Zeile 2, Token 4: +
Zeile 2, Token 5: 1
Zeile 2, Token 6: *
Zeile 2, Token 7: 2
```

Der Scanner hat nun jedes einzelne, durch Leerzeichen getrennte Wort oder Symbol in ein Array gepackt. Nun kann es aber sein, dass zwischen dem Plus Zeichen und der Zahl 1 kein Leerzeichen steht. Hier beginnt die Fehleranfälligkeit unseres Scanners. Da wir in unserer Syntaxdefinition nichts zu Leerzeichen zwischen Operatoren und Argumenten [VAR|NUM] gesagt haben wird davon ausgegangen, dass diese ignoriert werden.

Das Ziel ist es also nun einen Scanner zu schreiben, der in der Lage ist auch ohne die optionalen Leerzeichen die nötigen Tokens zu erstellen. Im folgendem Beispiel werden Sie einen solchen Scanner kennen lernen:

```
Option Explicit

Function Scan(ByVal strLine As String) As String()

    On Error GoTo ErrHandle
```

```
Dim Symb() As String
Dim SymbNo As Integer, i As Integer, j As Integer
Dim ThisChar As String, strBuffer As String
Dim WasOperator As Boolean

ReDim Symb(Len(strLine))

WasOperator = False

For i = 1 To Len(strLine)

    ThisChar = Mid(strLine, i, 1)

    If IsOperator(ThisChar) Then

        If WasOperator = True Then
            MsgBox "Fehler: Unerwarteter Operator.", vbCritical, _
                "Syntaxfehler"
            End
        Else
            Symb(SymbNo) = ThisChar
            SymbNo = SymbNo + 1
            WasOperator = True
        End If

    ElseIf ThisChar = "(" Or ThisChar = ")" Then

            Symb(SymbNo) = ThisChar
            SymbNo = SymbNo + 1
            WasOperator = False

    ElseIf ThisChar = Chr(34) Then

        Symb(SymbNo) = Chr(34)

        For j = (i + 1) To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If ThisChar = Chr(34) Then
                i = j: Exit For
            Else
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            End If

        Next j

        Symb(SymbNo) = Symb(SymbNo) & Chr(34)
        SymbNo = SymbNo + 1

        WasOperator = False

    ElseIf IsNumeric(ThisChar) Then

        Symb(SymbNo) = ""

        For j = i To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If IsNumeric(ThisChar) Then
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            ElseIf ThisChar = " " Then
```

```
                    SymbNo = SymbNo + 1: i = j: GoTo Skip
                ElseIf IsOperator(ThisChar) Or ThisChar = ")" Or _
                  ThisChar = "(" Then
                    SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
                Else
                    MsgBox "Fehler: Unerwartetes Zeichen.", vbCritical, _
                        "Syntaxfehler"
                    End
                End If

            Next j

            SymbNo = SymbNo + 1

            WasOperator = False

        ElseIf ThisChar = " " Or ThisChar = vbTab Then

            ' NOP

        Else

            Symb(SymbNo) = ""

            For j = i To Len(strLine)

                ThisChar = Mid(strLine, j, 1)

                If IsOperator(ThisChar) Or ThisChar = ")" Or _
                  ThisChar = "(" Then
                    SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
                ElseIf ThisChar = " " Then
                    SymbNo = SymbNo + 1: i = j: GoTo Skip
                Else
                    Symb(SymbNo) = Symb(SymbNo) & Lcase(ThisChar)
                End If

            Next j

            SymbNo = SymbNo + 1
            i = j

Skip:

            WasOperator = False

        End If

    Next i

    If SymbNo > 0 Then ReDim Preserve Symb(SymbNo - 1)
    Scan = Symb

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler beim Scannen"
    End
End Function

Function IsOperator(ByVal Expression) As Boolean
    If Expression = "+" Or Expression = "-" Or Expression = "*" _
           Or Expression = "/" Or Expression = "=" Or _
```

```
            Expression = "&" Then IsOperator = True
End Function
```

Das obige Beispiel hat keine Probleme mehr im Umgang mit Tabulatoren oder Leerzeichen. Außerdem werden Symbole die aus Buchstaben bestehen immer in Kleinbuchstaben gespeichert. Das ist übrigens nicht bei allen Scannern so, da Sprachen wie C oder JAVA durchaus Unterschiede zwischen Klein- und Großbuchstaben machen.

Das Prinzip der Funktion „Scan()“ ist relativ einfach zu skizzieren. Zunächst wird ein Array des Typs String erstellt, wobei dessen Größe gleich der Anzahl der Zeichen der Zeile ist. Denn genau dies ist die maximale Anzahl von Symbolen die in einer Zeile enthalten sein können.

Die FOR – Schleife wertet dann jedes einzelne Zeichen aus, wobei unterschieden wird, ob es sich um einen Operator, eine Zahl, das Leer- bzw. Tab- Zeichen, ein Hochkomma oder ein sonstiges Zeichen handelt. Handelt es sich um einen Operator wird der Operator als solches direkt in das Array eingefügt und der Index des aktuellen Elements inkrementiert. Handelt sich dagegen um eine Zahl oder ein anderes Zeichen wird solange eine weitere FOR – Schleife abgearbeitet bis das Ende des Symbols oder der Zahl durch Leerzeichen, Zeilenende oder einen Operator erreich wird. Erst jetzt wird der Wert in das Array gespeichert. Bei einem Hochkommata wird der Text ab dem beginnenden bis zu dem endenden Hochkommata gespeichert.

Zum Schluss der Routine wird noch das Array auf die passende Anzahl der gespeicherten Symbole reduziert. Der Scanner kann nach Belieben weiter ausgebaut und an die Bedürfnisse der Notationsregeln angepasst werden.

Ein einfacher Parser für eBas

Für die Zeile „DIM i“ benötigen wir eigentlich keine weitere Umformung der Symbole. Sobald aber mathematische Regeln wie Punkt-Vor-Strich oder Klammern zum tragen kommen benötigt man einen Parser, der diese Regeln beherrscht. Bevor wir also auf die Umsetzungen der einzelnen Symbole kommen wollen wir diese zunächst in die Richtige Reihenfolge bringen. Dazu benötigen wir einen Parser.

Um einen solchen Parser zu erstellen muss eine Rangfolge der Operatoren gegeben werden. Sofern man nicht seine eigenen mathematischen Regeln aufstellen will wird es geraten den Operatoren folgende Rangfolgen zu definieren:

Operator	Rangfolge
=	1
+ und - und &	2
* und /	3
(und)	4
Jedes andere Symbol	0

Je höherwertig ein Operator laut Rangfolge ist, desto „weiter vorne“ soll er später in den Tokens aufgelistet sein.

Demnach sollte zunächst eine Funktion erstellt werden an der man die Rangfolge eines Symbols feststellen kann. Des weiteren wird man ein Problem mit dem Vertauschen der Symbole im Symbol-Array bekommen (das Symbol-Array ist das Array aus Tokens das vom Scanner generiert wird). In der Informatik hat es sich als sehr sinnvoll erwiesen im Gegensatz zur gängigen Abarbeitung eines Terms die Umgekehrte Polnische Notation zu verwenden. Bei dieser Notation werden zunächst die Operanden und dann der Operator eingegeben:

Term:

*= 1 + 2 * 4*

Umgekehrte Polnische Notation:

*2 4 * 1 + =*

Gesprochen: 2 mit 4 multiplizieren und dieses Ergebnis mit 1 erhöhen.

Im Alltag findet sich eine Abart dieser Notation in den Syntaxregeln unserer Sprache:

Wäsche *waschen*
Essen *kochen*
Obst (und) Gemüse *kaufen*

Zugegebenermaßen erfordert die Umgekehrte Polnische Notation ein gewisses Umdenken, da man Sie aus der normalen mathematischen Umformung von Termen nicht gewohnt ist. Das obige Beispiel eines Terms mit Umgekehrter Polnischer Notation aber lässt sich für Computer besonders einfach umsetzten:

*2 4 * 1 + =*

```
PUSH 2
PUSH 4
MUL
PUSH 1
ADD
```

Dieser Pseudocode speichert zunächst die WORD Werte 2 und 4 in den Stack. Danach werden die letzten beiden WORD Werte des Stacks miteinander multipliziert, wobei dieses Ergebnis wieder in den Stack gespeichert wird. Nun muss nur noch die Zahl 1 hinzuaddiert werden, indem zunächst der WORD Wert 1 in den Stack gespeichert wird und dann die letzten beiden Stack Werte (also das Ergebnis der Multiplikation und der WORD Wert 1) miteinander addiert werden. Auch dieses Ergebnis wird wieder im Stack abgelegt.

Zugegebenermaßen kommt man hier schon leicht auf das Niveau der Codegenerierung, jedoch halte ich es sehr vom Vorteil, dass man weiß welchen Sinn die Umgekehrte Polnische Notation hat.

Das folgende Beispiel zeigt einen Scanner und Parser für eBas, wobei die Befehle erkannt, jedoch noch nicht umgesetzt werden:

mdlMain.Bas:

```
Sub Main()
```

```
    On Error GoTo ErrHandle

    Dim Zeile As String
    Dim n As Integer, intLine As Integer
    Dim Tokens

    intLine = 1

    Open "C:\test.eb" For Input As #1
    Do

        Line Input #1, Zeile

        Parse Zeile

        intLine = intLine + 1

    Loop Until EOF(1)
    Close #1

    Exit Sub

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"
    End
End Sub
```

mldScanner.Bas

```
Option Explicit

Function Scan(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Dim Symb() As String
    Dim SymbNo As Integer, i As Integer, j As Integer
    Dim ThisChar As String, strBuffer As String
    Dim WasOperator As Boolean

    ReDim Symb(Len(strLine))

    WasOperator = False

    For i = 1 To Len(strLine)

        ThisChar = Mid(strLine, i, 1)

        If IsOperator(ThisChar) Then

            If WasOperator = True Then
                MsgBox "Fehler: Unerwarteter Operator.", vbCritical, _
                    "Syntaxfehler"
                End
            Else
                Symb(SymbNo) = ThisChar
                SymbNo = SymbNo + 1
                WasOperator = True
            End If

        ElseIf ThisChar = "(" Or ThisChar = ")" Then
```

```
            Symb(SymbNo) = ThisChar
            SymbNo = SymbNo + 1
            WasOperator = False

    ElseIf ThisChar = Chr(34) Then

        Symb(SymbNo) = Chr(34)

        For j = (i + 1) To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If ThisChar = Chr(34) Then
                i = j: Exit For
            Else
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            End If

        Next j

        Symb(SymbNo) = Symb(SymbNo) & Chr(34)
        SymbNo = SymbNo + 1

        WasOperator = False

    ElseIf IsNumeric(ThisChar) Then

        Symb(SymbNo) = ""

        For j = i To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If IsNumeric(ThisChar) Then
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            ElseIf ThisChar = " " Then
                SymbNo = SymbNo + 1: i = j: GoTo Skip
            ElseIf IsOperator(ThisChar) Or ThisChar = ")" Or _
              ThisChar = "(" Then
                SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
            Else
                MsgBox "Fehler: Unerwartetes Zeichen.", vbCritical, _
                    "Syntaxfehler"
                End
            End If

        Next j

        SymbNo = SymbNo + 1

        WasOperator = False

    ElseIf ThisChar = " " Or ThisChar = vbTab Then

        ' NOP

    Else

        Symb(SymbNo) = ""

        For j = i To Len(strLine)
```

```
                ThisChar = Mid(strLine, j, 1)

                If IsOperator(ThisChar) Or ThisChar = ")" Or _
                  ThisChar = "(" Then
                    SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
                ElseIf ThisChar = " " Then
                    SymbNo = SymbNo + 1: i = j: GoTo Skip
                Else
                    Symb(SymbNo) = Symb(SymbNo) & LCase(ThisChar)
                End If

            Next j

            SymbNo = SymbNo + 1
            i = j

Skip:

            WasOperator = False

        End If

    Next i

    If SymbNo > 0 Then ReDim Preserve Symb(SymbNo - 1)
    Scan = Symb

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler beim Scannen"
    End
End Function

Function IsOperator(ByVal Expression) As Boolean
    If Expression = "+" Or Expression = "-" Or Expression = "*" _
           Or Expression = "/" Or Expression = "=" Or _
           Expression = "&" Then IsOperator = True
End Function
```

mdlParser.Bas:

```
Option Explicit

Sub Parse(ByVal strLine As String)

    On Error GoTo ErrHandle

    Dim Symb
    Dim SymbNo As Integer
    Dim i As Integer

    If Trim(strLine) <> "" Then

        Symb = Scan(strLine)

        Select Case LCase(Symb(0))
            Case "dim"
                ' Hier Programmteil DIM
            Case "if"
                ' Hier Programmteil IF ... THEN
            Case "goto"
```

```
                ' Hier Programmteil GOTO
            Case "do"
                ' Hier Programmteil DO
            Case "loop"
                ' Hier Programmteil LOOP
            Case "print"
                ' Hier Programmteil PRINT
            Case Else
                If UBound(Symb) > 0 Then
                    If Symb(1) = "=" Then
                        ParseTerm strLine
                    ElseIf Mid(Symb(0), 1, 1) = "#" Then
                        ' Hier Labelinitialisierung
                    Else
                        MsgBox "Syntaxfehler.", vbCritical, "Fehler"
                    End If
                End If

        End Select

    End If

    Exit Sub

ErrHandle:
        MsgBox Err.Description, vbCritical, "Fehler"

End Sub

Sub ParseTerm(ByVal Expression As String)

    Dim Symb
    Dim SymbIndex As Integer, i As Integer

    If Trim(Expression) <> "" Then

        Symb = ReservePolishNotation(Expression)

        ' Hier Auswerung des Terms

    End If

End Sub

Function GetPrecedence(ByVal Symb As String) As Byte

    Select Case LCase(Symb)
        Case "="
            GetPrecedence = 1
        Case "+", "-", "&"
            GetPrecedence = 2
        Case "*", "/"
            GetPrecedence = 3
        Case "(", ")"
            GetPrecedence = 4
        Case Else
            GetPrecedence = 0
    End Select

End Function

Function ReservePolishNotation(ByVal strLine As String) As String()
```

```
    On Error GoTo ErrHandle

    Dim i As Integer, SymbNo As Integer
    Dim Stack As New clsStack
    Dim Symb() As String
    Dim Tokens

    Stack.ClearStack

    Tokens = Scan("( " & strLine & " )")

    If UBound(Tokens) > 0 Then

        ReDim Symb(UBound(Tokens))

        For i = 0 To UBound(Tokens)

            Select Case Tokens(i)

                Case "("
                    Stack.Push Tokens(i)

                Case ")"
                    Do While Stack.Peek <> "("
                        Symb(SymbNo) = Stack.Pop
                        SymbNo = SymbNo + 1
                    Loop

                    If Stack.Peek = "(" Then Stack.Pop

                Case "+", "-", "*", "/", "="
                    Do
                        If Stack.StackSize = 0 Or Stack.Peek = "(" Or _
                          GetPrecedence(Stack.Peek) < _
                          GetPrecedence(Tokens(i)) Then
                            Stack.Push Tokens(i): Exit Do
                        Else
                            Symb(SymbNo) = Stack.Pop: SymbNo = SymbNo + 1
                        End If
                    Loop

                Case Else
                    Symb(SymbNo) = Tokens(i): SymbNo = SymbNo + 1

            End Select

        Next i

        ReDim Preserve Symb(SymbNo - 1)

        ReservePolishNotation = Symb

    End If

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"

End Function
```

clsStack.cls:

```
Option Explicit

Dim Stack(255) As String
Public StackSize As Byte

Sub ClearStack()
    StackSize = 0
End Sub

Function Peek() As String
    Peek = Stack(StackSize)
End Function

Function Pop() As String
    Dim Buffer As String
    Buffer = Stack(StackSize)
    StackSize = StackSize - 1
    Pop = Buffer
End Function

Sub Push(ByVal Expression As String)
    StackSize = StackSize + 1
    Stack(StackSize) = Expression
End Sub
```

Die Klasse Stack dient zum Ordnen und Zwischenspeichern der Argumente des Terms, was besonders wichtig bei Klammern und Punkt-Vor-Strich Rechnung ist. Die Funktion „ReservePolishNotation“ bedient sich der Funktion „Scan“ und ordnet dann die Argumente des Terms in die korrekte Reihenfolge, wobei diese sich nach der Rangordnung der Funktion „GetPrecedence“ richtet. Klammern werden wie man sehen kann durch die „ReservePolishNotation“ Funktion aufgelöst, so dass das Resultat keine Klammern mehr enthält.

Das Unterprogramm „Parse“ wird für jede Zeile aufgerufen. Dabei wird die Zeile zunächst in Symbole mit der Funktion „Scan“ aufgeteilt und dann geschaut welches das erste Symbol ist, da dieses entweder ein Befehl, eine Sprungmarke oder eine Zuweisung einer Variablen sein kann. Letzteres wird daran erkannt, ob das zweite Symbol ein Zuweisungszeichen („=“) ist. In diesem Fall wird die gesamte Zeile der Funktion „ParseTerm“ übergeben, die mittels der Funktion „ReservePolishNotation“ den Term auflöst und berechnen soll.

Die Code Generierung erfolgt durch diesen Scanner und Parser noch nicht. Diesem Thema werden sich die nächsten Teile der Serie „Compilerbau in der Praxis“ annehmen.

Teil 3 – Vom Parsen zur Codeerzeugung

Der eBas Compiler als „Crosscompiler“

Da sich dieses Tutorial besonders an Neulinge der Compilerprogrammierung richtet soll auch der Compiler von eBas besonders einfach aufgebaut sein. Dabei kommt es mir mehr darauf an das Prinzip eines Compilers zu erklären, als genau zu wissen wie man Maschinencode erzeugt. Deswegen ist der eBas Compiler als sog. Crosscompiler konzipiert, der aus der Zielsprache eBas ein Assembler Programm erstellt, das dann von einem Assembler (in diesem Falle der Freeware Assembler NASM) übersetzt werden kann.

In der Praxis gibt es eine Reihe von Crosscompilern, die vor allem aus der einen Hochsprache in eine andere Hochsprache übersetzten. Oft findet man so beispielsweise Crosscompiler die C Quellcode in Basic Quellcode oder umgekehrt übersetzten.

Der DIM Befehl

Bevor wir mit der eigentlichen Codegenerierung anfangen kümmern wir uns um das Handling von Variablen. Dazu reicht es nicht nur den DIM Befehl zu programmieren, wir müssen auch Funktionen schreiben die uns mitteilen, ob ein Symbol eine Variable ist oder nicht.

Alle Variablen werden nach dem DIM Befehl in ein besonderes Array des Compilers gespeichert:

```
Type tVars
    Name As String
End Type

Global Variables() As tVars
```

Zugegeben könnte man für die Bedürfnisse von eBas das Array Variables auch vom Typ String deklarieren, aber sobald man schon über die Implementation von mehreren Datentypen nachdenkt würde sich sofort ein Array mit einem besonderen Datentyp empfehlen.

Die Größe des Arrays ist dynamisch und kann daher mit dem REDIM Befehl in VB variiert werden.

Als nächstes wird eine einfache Prozedur entwickelt, die es ermöglicht eine neue Variable zu erstellen:

```
Sub AddNewVariable(ByVal VarName As String)
    On Error GoTo ErrHandle

    ReDim Variables(UBound(Variables) + 1)

Resm:

    Variables(UBound(Variables)).Name = LCase(Trim(VarName))
    Exit Sub
```

```
ErrHandle:

    ReDim Variables(1)
    GoTo Resm
End Sub
```

Leider kann man nicht davon ausgehen, dass es ein Benutzer sich immer an die Konventionen der Programmiersprache hält. Deswegen muss eine Funktion entwickelt werden, die prüft ob ein Variablenname erlaubt ist oder nicht. Aus der Definition von eBas wissen wir welche Variablennamen erlaubt sind. So modifierziert sich die obige Funktion ein wenig und eine neue Funktion zum testen des Variabelennamens kommt hinzu:

```
Sub AddNewVariable(ByVal VarName As String)
    On Error GoTo ErrHandle

    If Not VaildVarname(VarName) Then
        MsgBox "Variablenname nicht erlaubt.", vbCritical, "Fehler"
        Exit Sub
    End If

    ReDim Variables(UBound(Variables) + 1)

Resm:

    Variables(UBound(Variables)).Name = LCase(Trim(VarName))
    Exit Sub

ErrHandle:

    ReDim Variables(1)
    GoTo Resm
End Sub

Function VaildVarname(ByVal VarName As String) As Boolean

    Dim i As Integer
    Dim Res As Boolean
    Dim KeyWords As String

    KeyWords = " dim goto print do loop if then "

    Res = True

    If Len(Trim(VarName)) > 0 Then

        For i = 0 To UBound(Variables)
            If Variables(i).Name = VarName Then
                Res = False
                GoTo Skip
            End If
        Next i

        If InStr(1, VarName, KeyWords, vbTextCompare) > 0 Then
            Res = False: GoTo Skip
        End If

        If InStr(1, VarName, "+") > 0 Then Res = False
        If InStr(1, VarName, "/") > 0 Then Res = False
```

```
        If InStr(1, VarName, "-") > 0 Then Res = False
        If InStr(1, VarName, "*") > 0 Then Res = False
        If InStr(1, VarName, ".") > 0 Then Res = False
        If InStr(1, VarName, "\") > 0 Then Res = False
        If InStr(1, VarName, "=") > 0 Then Res = False
        If InStr(1, VarName, "?") > 0 Then Res = False
        If InStr(1, VarName, "ß") > 0 Then Res = False
        If InStr(1, VarName, "!") > 0 Then Res = False
        If InStr(1, VarName, "§") > 0 Then Res = False
        If InStr(1, VarName, "$") > 0 Then Res = False
        If InStr(1, VarName, "%") > 0 Then Res = False
        If InStr(1, VarName, "&") > 0 Then Res = False
        If InStr(1, VarName, "(") > 0 Then Res = False
        If InStr(1, VarName, ")") > 0 Then Res = False
        If InStr(1, VarName, "[") > 0 Then Res = False
        If InStr(1, VarName, "]") > 0 Then Res = False
        If InStr(1, VarName, "{") > 0 Then Res = False
        If InStr(1, VarName, "}") > 0 Then Res = False
        If InStr(1, VarName, "@") > 0 Then Res = False
        If InStr(1, VarName, "^") > 0 Then Res = False
        If InStr(1, VarName, "~") > 0 Then Res = False
        If InStr(1, VarName, "#") > 0 Then Res = False
        If InStr(1, VarName, "'") > 0 Then Res = False
        If InStr(1, VarName, "µ") > 0 Then Res = False
        If InStr(1, VarName, ";") > 0 Then Res = False
        If InStr(1, VarName, ",") > 0 Then Res = False
        If InStr(1, VarName, ":") > 0 Then Res = False
        If InStr(1, VarName, "<") > 0 Then Res = False
        If InStr(1, VarName, ">") > 0 Then Res = False
        If InStr(1, VarName, "|") > 0 Then Res = False
        If InStr(1, VarName, "°") > 0 Then Res = False
        If InStr(1, VarName, "´") > 0 Then Res = False
        If InStr(1, VarName, "`") > 0 Then Res = False

    Else
        Res = False
    End If

Skip:
    VaildVarname = Res
End Function
```

Codegenerierung aus einem Term

Als nächstes beschäftigen wir uns mit dem Parsen eines Terms. Das ist schon relativ kompliziert, da hier zum erstenmal auch direkt Code generiert wird. Zunächst brauchen wir deshalb eine globale Variable, die den generierten Code enthält:

```
Global OutPut As String
```

Nun möchte Ich dem Leser noch kurz etwas zu der hier vorgestellten Idee der Codegenerierung aus einem Term mitteilen. Nehmen wir an wir müssten den Term:

$$x = 1 + 2 * x$$

kompilieren. Als Ausgabe aus dem Parser erhalten wir:

*1 2 x * + = (x)*

Besonders einfach, wenn auch nicht effizient wäre es nun die jeweiligen Zahlen oder Variablen in den Stack des PCs beim Abarbeiten zu speichern und dann eine Operation (hier: Multiplikation und Addition) auf die letzten beiden Stack Werte anzuwenden und das Ergebnis wieder in den Stack zu schreiben. Zugegeben ist - wie ich schon erwähnt habe - das Prinzip nicht wirklich effizient, es erleichtert uns aber ungemein die Arbeit. Nach diesem Tutorial sollten Sie ferner in der Lage sein hierfür ein Alternativkonzept zu entwickeln.

Um mit dem Stack diese Operationen vornehmen zu können müssen wir vier Assembler Prozeduren schreiben, die dann bei der Kompilierung immer mit eingebunden werden:

```
_asm_mul:
      pop cx
      pop bx
      pop ax
      xor dx, dx
      mul bx
      push ax
      push cx
      ret

_asm_div:
      pop cx
      pop bx
      pop ax
      xor dx, dx
      div bx
      push ax
      push cx
      ret

_asm_add:
      pop cx
      pop bx
      pop ax
      add ax, bx
      push ax
      push cx
      ret

_asm_sub:
      pop cx
      pop bx
      pop ax
      sub ax, bx
      push ax
      push cx
      ret
```

Das Prinzip der Funktionen ist schnell erklärt. Wenn man mit dem Assembler Befehl CALL beispielsweise die Prozedur _asm_mul aufrufen will wird zunächst die Adresse des nächsten Befehls nach dem CALL Befehl (also die Rücksprungadresse) in den Stack geladen. Der oberste Stackwert ist beim starten der Prozedur also die Rücksprungadresse. Diese wird in CX zwischengespeichert, da wir noch weiter mit dem Stack arbeiten wollen. Als nächstes werden die beiden Stackwerte, die mit einander multipliziert werden soll eingelesen (das geschieht in umgekehrter Reihenfolge wegen des LIFO (Last in, first out) Prinzips). Man speichert die

beiden Werte in AX und BX und kann nun die Rechenoperation anwenden, wobei das Ergebnis nach jeder Rechenoperation automatisch in AX steht. AX wird nun vor der Rücksprungadresse, die wir uns in CX gemerkt haben, in den Stack gepusht. Durch PUSH CX wird dann vor dem RET Befehl noch sichergestellt, dass die Prozedur wieder dahin springt wo sie aufgerufen wurde. Das Ergebnis befindet sich also nach dem Rücksprung im obersten Stack Element.

Demnach kann man den parsergenerierten Term:

$$1\ 2\ x\ *\ +\ =\ (x)$$

wie folgt in Assembler umsetzten:

```
push word 1
push word 2
push word [x]
call _asm_mul
call _asm_add
pop ax
mov word [x], ax
```

Auf diese einfache Art und Weise ist es nun möglich jeden Term der auf den vier Grundrechenarten beruht zu verarbeiten.

Praktisch sind folgende Funktion in der Lage diese Operationen auszuführen:

mdlASM.bas

```
Global Const ASM_ADD = "_asm_add:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "add ax, bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_SUB = "_asm_sub:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "sub ax, bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_MUL = "_asm_mul:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "xor dx, dx" & vbCrLf & _
                vbTab & "mul bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_DIV = "_asm_div:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
```

```
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "xor dx, dx" & vbCrLf & _
                vbTab & "div bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf
```

mdlMain.bas

```
Sub Main()

    On Error GoTo ErrHandle

    Dim Zeile As String
    Dim n As Integer, intLine As Integer
    Dim Tokens

    intLine = 1

    Open "C:\test.eb" For Input As #1
    Do

        Line Input #1, Zeile

        Parse Zeile

        intLine = intLine + 1

    Loop Until EOF(1)
    Close #1

    MakeASMFile

    Exit Sub

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"
    End
End Sub

Sub MakeASMFile()

    On Error Resume Next

    Dim i As Integer

    Open "c:\test.asm" For Output As #1

        Print #1, "org 100h"

        Print #1, "mov ax, cs"
        Print #1, "mov ds, ax"
        Print #1, "mov es, ax"

        Print #1, OutPut

        Print #1, "mov ah, 4ch"
        Print #1, "int 21h"
```

```
        Print #1, "hang:"
        Print #1, vbTab & "jmp short hang"

        Print #1, ASM_ADD
        Print #1, ASM_SUB
        Print #1, ASM_MUL
        Print #1, ASM_DIV

        For i = 0 To UBound(Variables)
            If Variables(i).Name <> "" Then Print #1, Variables(i).Name _
                                                    & " db 0"
        Next i

    Close #1

End Sub


MdlParser.bas

Sub Parse(ByVal strLine As String)

    On Error GoTo ErrHandle

    Dim Symb
    Dim SymbNo As Integer
    Dim i As Integer

    If Trim(strLine) <> "" Then

        Symb = Scan(strLine)

        Select Case LCase(Symb(0))
            Case "dim"
                If UBound(Symb) = 1 Then
                    AddNewVariable Symb(1)
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If
            Case "if"
                ' Hier Programmteil IF ... THEN
            Case "goto"
                ' Hier Programmteil GOTO
            Case "do"
                ' Hier Programmteil DO
            Case "loop"
                ' Hier Programmteil LOOP
            Case "print"
                ' Hier Programmteil PRINT
            Case Else
                If UBound(Symb) > 0 Then
                    If Symb(1) = "=" Then
                        ParseTerm strLine
                    ElseIf Mid(Symb(0), 1, 1) = "#" Then
                        ' Hier Labelinitialisierung
                    Else
                        If Symb(0) <> "" Then MsgBox  _
                               "Syntaxfehler.", vbCritical, "Fehler"
                    End If
                End If

        End Select
```

```
    End If

    Exit Sub

ErrHandle:
        MsgBox Err.Description, vbCritical, "Fehler"

End Sub

Sub ParseTerm(ByVal Expression As String)

    Dim Symb
    Dim SymbIndex As Integer, i As Integer
    Dim SaveToVar As String

    If Trim(Expression) <> "" Then

        Symb = ReservePolishNotation(Expression)

        SaveToVar = Symb(0)

        For i = 1 To UBound(Symb)

            If IsNumeric(Symb(i)) Then
                OutPut = OutPut & "push word " & Symb(i) & vbCrLf
            ElseIf IsVariable(Symb(i)) Then
                OutPut = OutPut & "push word [" & Symb(i) & "]" & vbCrLf
            ElseIf Symb(i) = "+" Then
                OutPut = OutPut & "call _asm_add" & vbCrLf
            ElseIf Symb(i) = "-" Then
                OutPut = OutPut & "call _asm_sub" & vbCrLf
            ElseIf Symb(i) = "/" Then
                OutPut = OutPut & "call _asm_div" & vbCrLf
            ElseIf Symb(i) = "*" Then
                OutPut = OutPut & "call _asm_mul" & vbCrLf
            ElseIf Symb(i) = "=" Then
                If IsVariable(SaveToVar) Then
                    OutPut = OutPut & "pop ax" & vbCrLf & _
                                      "mov word [" & SaveToVar & "], ax" _
                                      & vbCrLf
                Else
                    MsgBox "Erwarte Variable", vbCritical, "Fehler"
                End If
            Else
                If Symb(i) <> "" Then MsgBox _
                            "Unbekannter Bezeichner", vbCritical, "Fehler"
            End If

        Next i

    End If

End Sub

Function GetPrecedence(ByVal Symb As String) As Byte

    Select Case LCase(Symb)
        Case "="
            GetPrecedence = 1
        Case "+", "-", "&"
            GetPrecedence = 2
        Case "*", "/"
            GetPrecedence = 3
```

```
        Case "(", ")"
            GetPrecedence = 4
        Case Else
            GetPrecedence = 0
    End Select

End Function

Function ReservePolishNotation(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Dim i As Integer, SymbNo As Integer
    Dim Stack As New clsStack
    Dim Symb() As String
    Dim Tokens

    Stack.ClearStack

    Tokens = Scan("( " & strLine & " )")

    If UBound(Tokens) > 0 Then

        ReDim Symb(UBound(Tokens))

        For i = 0 To UBound(Tokens)

            Select Case Tokens(i)

                Case "("
                    Stack.Push Tokens(i)

                Case ")"
                    Do While Stack.Peek <> "("
                        Symb(SymbNo) = Stack.Pop
                        SymbNo = SymbNo + 1
                    Loop

                    If Stack.Peek = "(" Then Stack.Pop

                Case "+", "-", "*", "/", "="
                    Do
                        If Stack.StackSize = 0 Or Stack.Peek = "(" Or _
                          GetPrecedence(Stack.Peek) < _
                          GetPrecedence(Tokens(i)) Then
                            Stack.Push Tokens(i): Exit Do
                        Else
                            Symb(SymbNo) = Stack.Pop: SymbNo = SymbNo + 1
                        End If
                    Loop

                Case Else
                    Symb(SymbNo) = Tokens(i): SymbNo = SymbNo + 1

            End Select

        Next i

        ReDim Preserve Symb(SymbNo - 1)

        ReservePolishNotation = Symb

    End If
```

```
    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"

End Function
```

mdlScanner.bas

```
Function Scan(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Dim Symb() As String
    Dim SymbNo As Integer, i As Integer, j As Integer
    Dim ThisChar As String, strBuffer As String
    Dim WasOperator As Boolean

    ReDim Symb(Len(strLine))

    WasOperator = False

    For i = 1 To Len(strLine)

        ThisChar = Mid(strLine, i, 1)

        If IsOperator(ThisChar) Then

            If WasOperator = True Then
                MsgBox "Fehler: Unerwarteter Operator.", vbCritical, _
                    "Syntaxfehler"
                End
            Else
                Symb(SymbNo) = ThisChar
                SymbNo = SymbNo + 1
                WasOperator = True
            End If

        ElseIf ThisChar = "(" Or ThisChar = ")" Then

                Symb(SymbNo) = ThisChar
                SymbNo = SymbNo + 1
                WasOperator = False

        ElseIf ThisChar = Chr(34) Then

            Symb(SymbNo) = Chr(34)

            For j = (i + 1) To Len(strLine)

                ThisChar = Mid(strLine, j, 1)

                If ThisChar = Chr(34) Then
                    i = j: Exit For
                Else
                    Symb(SymbNo) = Symb(SymbNo) & ThisChar
                End If

            Next j
```

```
            Symb(SymbNo) = Symb(SymbNo) & Chr(34)
            SymbNo = SymbNo + 1

            WasOperator = False

        ElseIf IsNumeric(ThisChar) Then

            Symb(SymbNo) = ""

            For j = i To Len(strLine)

                ThisChar = Mid(strLine, j, 1)

                If IsNumeric(ThisChar) Then
                    Symb(SymbNo) = Symb(SymbNo) & ThisChar
                ElseIf ThisChar = " " Then
                    SymbNo = SymbNo + 1: i = j: GoTo Skip
                ElseIf IsOperator(ThisChar) Or ThisChar = ")" Or _
                  ThisChar = "(" Then
                    SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
                Else
                    MsgBox "Fehler: Unerwartetes Zeichen.", vbCritical, _
                        "Syntaxfehler"
                    End
                End If

            Next j

            SymbNo = SymbNo + 1

            WasOperator = False

        ElseIf ThisChar = " " Or ThisChar = vbTab Then

            ' NOP

        Else

            Symb(SymbNo) = ""

            For j = i To Len(strLine)

                ThisChar = Mid(strLine, j, 1)

                If IsOperator(ThisChar) Or ThisChar = ")" Or _
                  ThisChar = "(" Then
                    SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
                ElseIf ThisChar = " " Then
                    SymbNo = SymbNo + 1: i = j: GoTo Skip
                Else
                    Symb(SymbNo) = Symb(SymbNo) & ThisChar
                End If

            Next j

            SymbNo = SymbNo + 1
            i = j

Skip:

            WasOperator = False
```

```
        End If

    Next i

    If SymbNo > 0 Then ReDim Preserve Symb(SymbNo - 1)
    Scan = Symb

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler beim Scannen"
    End
End Function

Function IsOperator(ByVal Expression) As Boolean
    If Expression = "+" Or Expression = "-" Or Expression = "*" _
           Or Expression = "/" Or Expression = "=" Or _
           Expression = "&" Then IsOperator = True
End Function
```

mdlTools.bas

```
Sub AddNewVariable(ByVal VarName As String)
    On Error GoTo ErrHandle

    If Not VaildVarname(VarName) Then
        MsgBox "Variablenname nicht erlaubt.", vbCritical, "Fehler"
        Exit Sub
    End If

    ReDim Variables(UBound(Variables) + 1)

Resm:

    Variables(UBound(Variables)).Name = LCase(Trim(VarName))
    Exit Sub

ErrHandle:

    ReDim Variables(1)
    GoTo Resm
End Sub

Function IsVariable(ByVal VarName As String) As Boolean

    Dim i As Integer

    For i = 0 To UBound(Variables)
        If Variables(i).Name = VarName Then
            IsVariable = True
            Exit For
        End If
    Next i

End Function

Function VaildVarname(ByVal VarName As String) As Boolean

    On Error GoTo ErrHandle
```

```
    Dim i As Integer
    Dim Res As Boolean
    Dim KeyWords As String

    KeyWords = " dim goto print do loop if then "

    Res = True

    If Len(Trim(VarName)) > 0 Then

        For i = 0 To UBound(Variables)
            If Variables(i).Name = VarName Then
                Res = False
                GoTo Skip
            End If
        Next i

        If InStr(1, VarName, KeyWords, vbTextCompare) > 0 Then
            Res = False: GoTo Skip
        End If

        If InStr(1, VarName, "+") > 0 Then Res = False
        If InStr(1, VarName, "/") > 0 Then Res = False
        If InStr(1, VarName, "-") > 0 Then Res = False
        If InStr(1, VarName, "*") > 0 Then Res = False
        If InStr(1, VarName, ".") > 0 Then Res = False
        If InStr(1, VarName, "\") > 0 Then Res = False
        If InStr(1, VarName, "=") > 0 Then Res = False
        If InStr(1, VarName, "?") > 0 Then Res = False
        If InStr(1, VarName, "ß") > 0 Then Res = False
        If InStr(1, VarName, "!") > 0 Then Res = False
        If InStr(1, VarName, "§") > 0 Then Res = False
        If InStr(1, VarName, "$") > 0 Then Res = False
        If InStr(1, VarName, "%") > 0 Then Res = False
        If InStr(1, VarName, "&") > 0 Then Res = False
        If InStr(1, VarName, "(") > 0 Then Res = False
        If InStr(1, VarName, ")") > 0 Then Res = False
        If InStr(1, VarName, "[") > 0 Then Res = False
        If InStr(1, VarName, "]") > 0 Then Res = False
        If InStr(1, VarName, "{") > 0 Then Res = False
        If InStr(1, VarName, "}") > 0 Then Res = False
        If InStr(1, VarName, "@") > 0 Then Res = False
        If InStr(1, VarName, "^") > 0 Then Res = False
        If InStr(1, VarName, "~") > 0 Then Res = False
        If InStr(1, VarName, "#") > 0 Then Res = False
        If InStr(1, VarName, "'") > 0 Then Res = False
        If InStr(1, VarName, "µ") > 0 Then Res = False
        If InStr(1, VarName, ";") > 0 Then Res = False
        If InStr(1, VarName, ",") > 0 Then Res = False
        If InStr(1, VarName, ":") > 0 Then Res = False
        If InStr(1, VarName, "<") > 0 Then Res = False
        If InStr(1, VarName, ">") > 0 Then Res = False
        If InStr(1, VarName, "|") > 0 Then Res = False
        If InStr(1, VarName, "°") > 0 Then Res = False
        If InStr(1, VarName, "´") > 0 Then Res = False
        If InStr(1, VarName, "`") > 0 Then Res = False

    Else
        Res = False
    End If

Skip:
    VaildVarname = Res
```

```
    Exit Function
ErrHandle:
    VaildVarname = Res
End Function
```

mdlTypes.bas

```
Type tVars
    Name As String
End Type

Global Variables() As tVars
Global OutPut As String
```

clsStack.cls

```
Dim Stack(255) As String
Public StackSize As Byte

Sub ClearStack()
    StackSize = 0
End Sub

Function Peek() As String
    Peek = Stack(StackSize)
End Function

Function Pop() As String
    Dim Buffer As String
    Buffer = Stack(StackSize)
    StackSize = StackSize - 1
    Pop = Buffer
End Function

Sub Push(ByVal Expression As String)
    StackSize = StackSize + 1
    Stack(StackSize) = Expression
End Sub
```

Teil 4 – Die restlichen Befehle

GoTo...

Der GoTo Befehl kann sowohl rückwärts und vorwärts zu einem Label, einer sogenannten Sprungmarke springen. Dazu muss zunächst eine Routine geschrieben werden, womit solche Sprungmarken umgesetzt werden können.

```
Type tLabels
    Name As String
End Type

Global Label() As tLabels
Global Gotos() As tLabels
```

Für die Label und für die einzelnen Sprünge muss in diesem Fall ein extra Array angelegt werden, da man in der Variable Gotos die jeweiligen Sprünge schreibt um später nach Ablauf des Programms zu prüfen ob alle Sprünge auch zu einem Label führen, das in der Label Liste „Label“ steht.

```
Sub AddNewLabel(ByVal VarName As String)
    On Error GoTo ErrHandle

    Dim f As Boolean

    f = VaildName(VarName)

    If Not f Then
        MsgBox "Sprungmarkenname nicht erlaubt.", vbCritical, "Fehler"
        Exit Sub
    End If

    ReDim Label(UBound(Label) + 1)

Resm:

    Label(UBound(Label)).Name = LCase(Trim(VarName))
    Exit Sub

ErrHandle:

    ReDim Label(1)
    GoTo Resm
End Sub

Sub AddNewGoto(ByVal Name As String)
    On Error GoTo ErrHandle

    ReDim Gotos(UBound(Gotos) + 1)

Resm:

    Gotos(UBound(Gotos)).Name = LCase(Trim(Name))
    Exit Sub

ErrHandle:
```

```
    ReDim Gotos(1)
    GoTo Resm
End Sub


Sub CheckGotos()

    On Error Resume Next

    Dim i As Integer, s As Integer
    Dim f As Boolean

    If UBound(Gotos) >= 0 Then

        If UBound(labels) >= 0 Then

            For i = 0 To UBound(Gotos)

                f = False

                For s = 0 To UBound(Label)
                    If Gotos(i).Name = Label(s).Name Then
                        f = True
                        Exit For
                    End If
                Next s

                If f = False Then MsgBox "Sprungmarke '" & Gotos(i).Name &
_
                    "' nicht gefunden!", vbCritical, "Fehler"

            Next i

        Else
            MsgBox "Keine Sprungmarken vorhanden!"
        End If

    End If

    Exit Sub

End Sub

Function VaildName(ByVal VarName As String) As Boolean

    On Error GoTo ErrHandle

    Dim i As Integer
    Dim Res As Boolean
    Dim KeyWords As String

    KeyWords = " dim goto print do loop if then "

    Res = True

    If Len(Trim(VarName)) > 0 Then

        For i = 0 To UBound(Variables)
            If Variables(i).Name = VarName Then
                Res = False
                GoTo Skip
            End If
```

```
        Next i


        For i = 0 To UBound(Label)
            If Label(i).Name = VarName Then
                Res = False
                GoTo Skip
            End If
        Next i

        If InStr(1, VarName, KeyWords, vbTextCompare) > 0 Then
            Res = False: GoTo Skip
        End If

        If InStr(1, VarName, "+") > 0 Then Res = False
        If InStr(1, VarName, "/") > 0 Then Res = False
        If InStr(1, VarName, "-") > 0 Then Res = False
        If InStr(1, VarName, "*") > 0 Then Res = False
        If InStr(1, VarName, ".") > 0 Then Res = False
        If InStr(1, VarName, "\") > 0 Then Res = False
        If InStr(1, VarName, "=") > 0 Then Res = False
        If InStr(1, VarName, "?") > 0 Then Res = False
        If InStr(1, VarName, "ß") > 0 Then Res = False
        If InStr(1, VarName, "!") > 0 Then Res = False
        If InStr(1, VarName, "§") > 0 Then Res = False
        If InStr(1, VarName, "$") > 0 Then Res = False
        If InStr(1, VarName, "%") > 0 Then Res = False
        If InStr(1, VarName, "&") > 0 Then Res = False
        If InStr(1, VarName, "(") > 0 Then Res = False
        If InStr(1, VarName, ")") > 0 Then Res = False
        If InStr(1, VarName, "[") > 0 Then Res = False
        If InStr(1, VarName, "]") > 0 Then Res = False
        If InStr(1, VarName, "{") > 0 Then Res = False
        If InStr(1, VarName, "}") > 0 Then Res = False
        If InStr(1, VarName, "@") > 0 Then Res = False
        If InStr(1, VarName, "^") > 0 Then Res = False
        If InStr(1, VarName, "~") > 0 Then Res = False
        If InStr(1, VarName, "#") > 0 Then Res = False
        If InStr(1, VarName, "'") > 0 Then Res = False
        If InStr(1, VarName, "µ") > 0 Then Res = False
        If InStr(1, VarName, ";") > 0 Then Res = False
        If InStr(1, VarName, ",") > 0 Then Res = False
        If InStr(1, VarName, ":") > 0 Then Res = False
        If InStr(1, VarName, "<") > 0 Then Res = False
        If InStr(1, VarName, ">") > 0 Then Res = False
        If InStr(1, VarName, "|") > 0 Then Res = False
        If InStr(1, VarName, "°") > 0 Then Res = False
        If InStr(1, VarName, "´") > 0 Then Res = False
        If InStr(1, VarName, "`") > 0 Then Res = False

    Else
        Res = False
    End If

Skip:
    VaildName = Res
    Exit Function
ErrHandle:
    VaildName = Res
    GoTo Skip
End Function
```

Analog zu den Funktionen für die Speicherung der Variablen werden die Sprünge und Sprungmarken auch in ein dynamisches Array geladen. Der Unterschied hierbei ist allerdings, dass automatisch nach Abarbeitung der jeweiligen Funktionen direkt in den Ausgabestream der Assemblercode für die Sprünge geschrieben wird.

```
Sub Parse(ByVal strLine As String)

    On Error GoTo ErrHandle

    Dim Symb
    Dim SymbNo As Integer
    Dim i As Integer

    If Trim(strLine) <> "" Then

        Symb = Scan(strLine)

        Select Case LCase(Symb(0))
            Case "dim"
                If UBound(Symb) = 1 Then
                    AddNewVariable Symb(1)
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If
            Case "if"
                eBas_If_Statement Symb
            Case "goto"
                If UBound(Symb) = 1 Then
                    AddNewGoto Symb(1)
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If

                OutPut = OutPut & "jmp near " & Symb(1) & vbCrLf

            Case "do"
                ' Hier Programmteil DO
            Case "loop"
                ' Hier Programmteil LOOP
            Case "print"
                eBas_Print_Statement Symb
            Case Else
                If UBound(Symb) > 0 Then
                    If Symb(1) = "=" Then
                        ParseTerm strLine
                    ElseIf Mid(Symb(0), 1, 1) = "#" Then
                        MakeLabel Symb
                    Else
                        If Symb(0) <> "" Then MsgBox "Syntaxfehler.",
vbCritical, "Fehler"
                    End If
                Else

                    If Mid(Symb(0), 1, 1) = "#" Then
                        MakeLabel Symb
                    ElseIf Symb(0) <> "" Then
                        MsgBox "Unbekannter Befehl: " & Symb(0) & ".",
vbCritical, "Fehler"
                    End If

                End If
```

```
        End Select

    End If

    If WaitingForIfEnd = True Then
        OutPut = OutPut & "_if_" & Ifs & ":" & vbCrLf
        Ifs = Ifs + 1
        WaitingForIfEnd = False
    End If

    Exit Sub

ErrHandle:
        MsgBox Err.Description, vbCritical, "Fehler"

End Sub

Sub MakeLabel(ByVal Symb)

    If UBound(Symb) = 0 Then
        AddNewLabel Mid(Symb(0), 2)
    Else
        MsgBox "Syntax Fehler", vbCritical, "Fehler"
    End If

    OutPut = OutPut & Mid(Symb(0), 2) & ":" & vbCrLf

End Sub
```

Eigentlich könnte man direkt aus dem Goto und der Labelmarke den Assemblercode erzeugen, jedoch sollte der Compiler auch auf fehlerhafte Sprünge aufmerksam machen. Deshalb muss zur Labelinitialisierung MakeLabel aufgerufen werden. Der Goto Befehl sollte ebenfalls zur Fehlerbehandlung die Funktion AddNewGoto ausführen. Zum Schluss der Abarbeitung des gesamten Codes sollte der Compiler außerdem noch CheckGotos aufrufen. Hier werden alle Sprünge auf deren Korrektheit getestet.

Damit wäre auch der Goto Befehl implementiert.

Do...Loop

Beim Do...Loop Befehl machen wir es uns einfach. Wir bauen aus einem Goto...#Label Konstrukt einen Loop Befehl. Eine Start- und Endbedingung bauen wir nicht ein. Do...Loop Schleifen können aber verschachtelt sein und deswegen muss man sich merken welche Loop Schleife aktuell verwendet wird und welche die übergeordnete Schleife ist.

```
Type tDo
    Last As Integer
    This As Integer
End Type

Global DoLoop(255) As tDo, DoIndex As Integer
Global ThisDo As Integer, LastDo As Integer

Sub AddNewDo()

    ThisDo = ThisDo + 1
    DoIndex = DoIndex + 1
```

```
    DoLoop(DoIndex).Last = DoLoop(DoIndex - 1).This
    DoLoop(DoIndex).This = ThisDo

    OutPut = OutPut & "__do_" & DoLoop(DoIndex).This & ":" & vbCrLf

    Exit Sub

End Sub

Sub CloseDo()
    OutPut = OutPut & "jmp short __do_" & DoLoop(DoIndex).This & vbCrLf
    DoIndex = DoLoop(DoIndex).Last
End Sub
```

Wenn das DO Schlüsselwort gefunden wurde wird die Routine AddNewDo ausgeführt. Diese erstellt eine neue Sprungmarke mit einer durchlaufenden Nummer. Außerdem wird die letzte Do Schleife gespeichert.

Mit LOOP wird eine Schleife geschlossen und die CloseDo Funktion aufgerufen. Diese erstellt einen Sprung zur letzten Sprungmarke und wählt die davor liegende Do Schleife aus um die Verschachtelung zu ermöglichen.

Kompletter Source Code

Damit wäre der Compiler für eBas fertig gestellt. Es wird natürlich kein Anspruch auf fehlerfreie Syntax und Bugfreies Ausführen erhoben.

Hier folgt der komplette Sourcecode:

mdlTypes.bas:

```
Type tVars
    Name As String
End Type

Type tLabels
    Name As String
End Type

Type tDo
    Last As Integer
    This As Integer
End Type

Global Variables() As tVars
Global OutPut As String
Global StringOutPutBuffer As String
Global StringCount As Integer
Global Ifs As Integer
Global WaitingForIfEnd As Boolean
Global Label() As tLabels
Global Gotos() As tLabels
Global DoLoop(255) As tDo, DoIndex As Integer
Global ThisDo As Integer, LastDo As Integer
```

clsStack.cls:

```
Option Explicit

Dim Stack(255) As String
Public StackSize As Byte

Sub ClearStack()
    StackSize = 0
End Sub

Function Peek() As String
    Peek = Stack(StackSize)
End Function

Function Pop() As String
    Dim Buffer As String
    Buffer = Stack(StackSize)
    StackSize = StackSize - 1
    Pop = Buffer
End Function

Sub Push(ByVal Expression As String)
    StackSize = StackSize + 1
    Stack(StackSize) = Expression
End Sub
```

mdlMain.bas:

```
Option Explicit

Sub Main()

    On Error GoTo ErrHandle

    Dim Zeile As String
    Dim n As Integer, intLine As Integer
    Dim Tokens

    intLine = 1

    AddNewVariable "ebas"
    AddNewLabel "start"

    Open "C:\test.eb" For Input As #1
    Do

        Line Input #1, Zeile

        Parse Zeile

        intLine = intLine + 1

    Loop Until EOF(1)
    Close #1

    CheckGotos

    Open "c:\test.asm" For Output As #1
```

```
        Print #1, ASM_BEGIN

        Print #1, OutPut

        Print #1, ASM_END

        Print #1, StringOutPutBuffer

        For n = 0 To UBound(Variables)
            Print #1, Variables(n).Name & " DB " & 0
        Next n

        Print #1, "_str_crlf DB 13,10,'$'"
        Print #1, "Buffer TIMES 10 DB '$'"

        Print #1, ASM_PRINTNUMBER1
        Print #1, ASM_PRINTNUMBER2
        Print #1, ASM_PRINT
        Print #1, ASM_ADD
        Print #1, ASM_SUB
        Print #1, ASM_DIV
        Print #1, ASM_MUL


    Close #1

    Exit Sub

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"
    End
End Sub
```

mdlTools.bas:

```
Sub AddNewVariable(ByVal VarName As String)
    On Error GoTo ErrHandle

    Dim f As Boolean

    f = VaildName(VarName)

    If Not f Then
        MsgBox "Variablenname nicht erlaubt.", vbCritical, "Fehler"
        Exit Sub
    End If

    ReDim Variables(UBound(Variables) + 1)

Resm:

    Variables(UBound(Variables)).Name = LCase(Trim(VarName))
    Exit Sub

ErrHandle:

    ReDim Variables(1)
    GoTo Resm
End Sub

Sub AddNewLabel(ByVal VarName As String)
```

```
    On Error GoTo ErrHandle

    Dim f As Boolean

    f = VaildName(VarName)

    If Not f Then
        MsgBox "Sprungmarkenname nicht erlaubt.", vbCritical, "Fehler"
        Exit Sub
    End If

    ReDim Label(UBound(Label) + 1)

Resm:

    Label(UBound(Label)).Name = LCase(Trim(VarName))
    Exit Sub

ErrHandle:

    ReDim Label(1)
    GoTo Resm
End Sub

Sub AddNewGoto(ByVal Name As String)
    On Error GoTo ErrHandle

    ReDim Gotos(UBound(Gotos) + 1)

Resm:

    Gotos(UBound(Gotos)).Name = LCase(Trim(Name))
    Exit Sub

ErrHandle:

    ReDim Gotos(1)
    GoTo Resm
End Sub

Sub AddNewDo()

    ThisDo = ThisDo + 1
    DoIndex = DoIndex + 1

    DoLoop(DoIndex).Last = DoLoop(DoIndex - 1).This
    DoLoop(DoIndex).This = ThisDo

    OutPut = OutPut & "__do_" & DoLoop(DoIndex).This & ":" & vbCrLf

    Exit Sub

End Sub

Sub CloseDo()
    OutPut = OutPut & "jmp short __do_" & DoLoop(DoIndex).This & vbCrLf
    DoIndex = DoLoop(DoIndex).Last
End Sub

Function IsVariable(ByVal VarName As String) As Boolean

    On Error GoTo ErrHandle
```

```
    Dim i As Integer

    If UBound(Variables) >= 0 Then

        For i = 0 To UBound(Variables)
            If Variables(i).Name = VarName Then
                IsVariable = True
                Exit For
            End If
        Next i

    End If

    Exit Function
ErrHandle:
    IsVariable = False

End Function

Sub CheckGotos()

    On Error Resume Next

    Dim i As Integer, s As Integer
    Dim f As Boolean

    If UBound(Gotos) >= 0 Then

        If UBound(labels) >= 0 Then

            For i = 0 To UBound(Gotos)

                f = False

                For s = 0 To UBound(Label)
                    If Gotos(i).Name = Label(s).Name Then
                        f = True
                        Exit For
                    End If
                Next s

                If f = False Then MsgBox "Sprungmarke '" & Gotos(i).Name &
_
                    "' nicht gefunden!", vbCritical, "Fehler"

            Next i

        Else
            MsgBox "Keine Sprungmarken vorhanden!"
        End If

    End If

    Exit Sub

End Sub

Function VaildName(ByVal VarName As String) As Boolean

    On Error GoTo ErrHandle

    Dim i As Integer
    Dim Res As Boolean
```

```
Dim KeyWords As String

KeyWords = " dim goto print do loop if then "

Res = True

If Len(Trim(VarName)) > 0 Then

    For i = 0 To UBound(Variables)
        If Variables(i).Name = VarName Then
            Res = False
            GoTo Skip
        End If
    Next i


    For i = 0 To UBound(Label)
        If Label(i).Name = VarName Then
            Res = False
            GoTo Skip
        End If
    Next i

    If InStr(1, VarName, KeyWords, vbTextCompare) > 0 Then
        Res = False: GoTo Skip
    End If

    If InStr(1, VarName, "+") > 0 Then Res = False
    If InStr(1, VarName, "/") > 0 Then Res = False
    If InStr(1, VarName, "-") > 0 Then Res = False
    If InStr(1, VarName, "*") > 0 Then Res = False
    If InStr(1, VarName, ".") > 0 Then Res = False
    If InStr(1, VarName, "\") > 0 Then Res = False
    If InStr(1, VarName, "=") > 0 Then Res = False
    If InStr(1, VarName, "?") > 0 Then Res = False
    If InStr(1, VarName, "ß") > 0 Then Res = False
    If InStr(1, VarName, "!") > 0 Then Res = False
    If InStr(1, VarName, "§") > 0 Then Res = False
    If InStr(1, VarName, "$") > 0 Then Res = False
    If InStr(1, VarName, "%") > 0 Then Res = False
    If InStr(1, VarName, "&") > 0 Then Res = False
    If InStr(1, VarName, "(") > 0 Then Res = False
    If InStr(1, VarName, ")") > 0 Then Res = False
    If InStr(1, VarName, "[") > 0 Then Res = False
    If InStr(1, VarName, "]") > 0 Then Res = False
    If InStr(1, VarName, "{") > 0 Then Res = False
    If InStr(1, VarName, "}") > 0 Then Res = False
    If InStr(1, VarName, "@") > 0 Then Res = False
    If InStr(1, VarName, "^") > 0 Then Res = False
    If InStr(1, VarName, "~") > 0 Then Res = False
    If InStr(1, VarName, "#") > 0 Then Res = False
    If InStr(1, VarName, "'") > 0 Then Res = False
    If InStr(1, VarName, "µ") > 0 Then Res = False
    If InStr(1, VarName, ";") > 0 Then Res = False
    If InStr(1, VarName, ",") > 0 Then Res = False
    If InStr(1, VarName, ":") > 0 Then Res = False
    If InStr(1, VarName, "<") > 0 Then Res = False
    If InStr(1, VarName, ">") > 0 Then Res = False
    If InStr(1, VarName, "|") > 0 Then Res = False
    If InStr(1, VarName, "°") > 0 Then Res = False
    If InStr(1, VarName, "´") > 0 Then Res = False
    If InStr(1, VarName, "`") > 0 Then Res = False
```

```
    Else
        Res = False
    End If

Skip:
    VaildName = Res
    Exit Function
ErrHandle:
    VaildName = Res
    GoTo Skip
End Function
```

mdlScanner.bas:

```
Option Explicit

Function Scan(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Dim Symb() As String
    Dim SymbNo As Integer, i As Integer, j As Integer
    Dim ThisChar As String, strBuffer As String
    Dim WasOperator As Boolean

    ReDim Symb(Len(strLine))

    WasOperator = False

    For i = 1 To Len(strLine)

        ThisChar = Mid(strLine, i, 1)

        If IsOperator(ThisChar) Then

            If WasOperator = True Then
                MsgBox "Fehler: Unerwarteter Operator.", vbCritical, _
                    "Syntaxfehler"
                End
            Else
                Symb(SymbNo) = ThisChar
                SymbNo = SymbNo + 1
                WasOperator = True
            End If

        ElseIf ThisChar = "(" Or ThisChar = ")" Then

                Symb(SymbNo) = ThisChar
                SymbNo = SymbNo + 1
                WasOperator = False

        ElseIf ThisChar = Chr(34) Then

            Symb(SymbNo) = Chr(34)

            For j = (i + 1) To Len(strLine)

                ThisChar = Mid(strLine, j, 1)

                If ThisChar = Chr(34) Then
                    i = j: Exit For
```

```
            Else
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            End If

        Next j

        Symb(SymbNo) = Symb(SymbNo) & Chr(34)
        SymbNo = SymbNo + 1

        WasOperator = False

    ElseIf IsNumeric(ThisChar) Then

        Symb(SymbNo) = ""

        For j = i To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If IsNumeric(ThisChar) Then
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            ElseIf ThisChar = " " Then
                SymbNo = SymbNo + 1: i = j: GoTo Skip
            ElseIf IsOperator(ThisChar) Or ThisChar = ")" Or _
              ThisChar = "(" Then
                SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
            Else
                MsgBox "Fehler: Unerwartetes Zeichen.", vbCritical, _
                    "Syntaxfehler"
                End
            End If

        Next j

        SymbNo = SymbNo + 1

        WasOperator = False

    ElseIf ThisChar = " " Or ThisChar = vbTab Then

        ' NOP

    Else

        Symb(SymbNo) = ""

        For j = i To Len(strLine)

            ThisChar = Mid(strLine, j, 1)

            If IsOperator(ThisChar) Or ThisChar = ")" Or _
              ThisChar = "(" Then
                SymbNo = SymbNo + 1: i = (j - 1): GoTo Skip
            ElseIf ThisChar = " " Then
                SymbNo = SymbNo + 1: i = j: GoTo Skip
            Else
                Symb(SymbNo) = Symb(SymbNo) & ThisChar
            End If

        Next j

        SymbNo = SymbNo + 1
        i = j
```

```
Skip:

            WasOperator = False

        End If

    Next i

    If SymbNo > 0 Then ReDim Preserve Symb(SymbNo - 1)
    Scan = Symb

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler beim Scannen"
    End
End Function

Function IsOperator(ByVal Expression) As Boolean
    If Expression = "+" Or Expression = "-" Or Expression = "*" _
           Or Expression = "/" Or Expression = "=" Or _
           Expression = "&" Then IsOperator = True
End Function
```

mdlParser.bas:

```
Option Explicit

Sub Parse(ByVal strLine As String)

    On Error GoTo ErrHandle

    Dim Symb
    Dim SymbNo As Integer
    Dim i As Integer

    If Trim(strLine) <> "" Then

        Symb = Scan(strLine)

        Select Case LCase(Symb(0))
            Case "dim"
                If UBound(Symb) = 1 Then
                    AddNewVariable Symb(1)
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If
            Case "if"
                eBas_If_Statement Symb
            Case "goto"
                If UBound(Symb) = 1 Then
                    AddNewGoto Symb(1)
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If

                OutPut = OutPut & "jmp near " & Symb(1) & vbCrLf

            Case "do"
                If UBound(Symb) = 0 Then
```

```
                    AddNewDo
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If
            Case "loop"
                If UBound(Symb) = 0 Then
                    CloseDo
                Else
                    MsgBox "Syntaxfehler", vbCritical, "Fehler"
                End If
            Case "print"
                eBas_Print_Statement Symb
            Case Else
                If UBound(Symb) > 0 Then
                    If Symb(1) = "=" Then
                        ParseTerm strLine
                    ElseIf Mid(Symb(0), 1, 1) = "#" Then
                        MakeLabel Symb
                    Else
                        If Symb(0) <> "" Then MsgBox "Syntaxfehler.",
vbCritical, "Fehler"
                    End If
                Else

                    If Mid(Symb(0), 1, 1) = "#" Then
                        MakeLabel Symb
                    ElseIf Symb(0) <> "" Then
                        MsgBox "Unbekannter Befehl: " & Symb(0) & ".",
vbCritical, "Fehler"
                    End If

                End If

        End Select

    End If

    If WaitingForIfEnd = True Then
        OutPut = OutPut & "_if_" & Ifs & ":" & vbCrLf
        Ifs = Ifs + 1
        WaitingForIfEnd = False
    End If

    Exit Sub

ErrHandle:
        MsgBox Err.Description, vbCritical, "Fehler"

End Sub

Sub MakeLabel(ByVal Symb)

    If UBound(Symb) = 0 Then
        AddNewLabel Mid(Symb(0), 2)
    Else
        MsgBox "Syntax Fehler", vbCritical, "Fehler"
    End If

    OutPut = OutPut & Mid(Symb(0), 2) & ":" & vbCrLf

End Sub

Sub AddNewString(ByVal Name As String, ByVal Expression As String)
```

```
    StringOutPutBuffer = StringOutPutBuffer & Name & " DB '" & Expression &
"$'" & vbCrLf
End Sub

Sub eBas_Print_Statement(ByVal Symb)

    On Error GoTo ErrHandle

    Dim i As Integer
    Dim Exp As Boolean

    Exp = True

    For i = 1 To UBound(Symb)

        If IsVariable(Symb(i)) = True Or IsNumeric(Symb(i)) Then

            If Exp = True Then

                If IsNumeric(Symb(i)) = True Then
                    OutPut = OutPut & "mov bx, " & Symb(i) & vbCrLf
                Else
                    OutPut = OutPut & "mov bx, [" & Symb(i) & "]" & vbCrLf
                End If

                OutPut = OutPut & "mov di, Buffer" & vbCrLf & _
                    "call _print_number" & vbCrLf

                Exp = False
            Else
                MsgBox "Unerwartetes Symbol"
            End If

        Else

            If Left(Symb(i), 1) = Chr(34) And Right(Symb(i), 1) = Chr(34)
Then

                If Exp = True Then
                    AddNewString "_str_" & StringCount, Mid(Symb(i), 2,
Len(Symb(i)) - 2)
                    OutPut = OutPut & "mov di, _str_" & StringCount &
vbCrLf & _
                        "call _print" & vbCrLf
                    Exp = False
                Else
                    MsgBox "Unerwartetes Symbol"
                End If

            ElseIf Symb(i) = "&" Then
                If Exp = True Then
                    MsgBox "Unerwartetes Symbol"
                Else
                    Exp = True
                End If
            ElseIf Symb(i) = "ebcrlf" Then
                If Exp = False Then
                    MsgBox "Unerwartetes Symbol"
                Else
                    OutPut = OutPut & "mov di, _str_crlf" & vbCrLf & _
                        "call _print" & vbCrLf
                    Exp = False
                End If
```

```
            Else
                MsgBox "Syntax Fehler"
            End If

        End If

    Next i

    If Exp = True Then MsgBox "Unerwartetes Zeilenende"

    Exit Sub

ErrHandle:
    MsgBox "Unerwarteter Fehler."

End Sub

Sub eBas_If_Statement(ByVal Symb)

    On Error GoTo ErrHandle

    Dim i As Integer
    Dim Buffer
    Dim t As String

    ' IF (...) = (...) THEN (...)

    If UBound(Symb) >= 5 Then

        For i = 1 To UBound(Symb)

            If LCase(Symb(i)) = "then" Then
                i = i + 1
                Exit For
            Else
                If UBound(Symb) = i Then
                    MsgBox "THEN erwartet"
                Else
                    t = t & Symb(i) & " "
                End If
            End If

        Next i

        ParseTerm t, True

        t = ""

        For i = i To UBound(Symb)
            t = t & Symb(i) & " "
        Next i

        WaitingForIfEnd = True

        Parse t

    Else
        MsgBox "Unerwartetes Zeilenende"
    End If

    Exit Sub

ErrHandle:
```

```
    MsgBox "Unerwarteter Fehler"

End Sub

Sub ParseTerm(ByVal Expression As String, Optional IsIf As Boolean = False)

    Dim Symb
    Dim SymbIndex As Integer, i As Integer
    Dim SaveToVar As String

    If Trim(Expression) <> "" Then

        Symb = ReservePolishNotation(Expression)

        SaveToVar = Symb(0)

        For i = 1 To UBound(Symb)

            If IsNumeric(Symb(i)) Then
                OutPut = OutPut & "push word " & Symb(i) & vbCrLf
            ElseIf IsVariable(Symb(i)) Then
                OutPut = OutPut & "push word [" & Symb(i) & "]" & vbCrLf
            ElseIf Symb(i) = "+" Then
                OutPut = OutPut & "call _asm_add" & vbCrLf
            ElseIf Symb(i) = "-" Then
                OutPut = OutPut & "call _asm_sub" & vbCrLf
            ElseIf Symb(i) = "/" Then
                OutPut = OutPut & "call _asm_div" & vbCrLf
            ElseIf Symb(i) = "*" Then
                OutPut = OutPut & "call _asm_mul" & vbCrLf
            ElseIf Symb(i) = "=" Then
                If IsIf = False Then

                    If IsVariable(SaveToVar) Then
                        OutPut = OutPut & "pop ax" & _
                                          vbCrLf & "mov word [" & SaveToVar
& "], ax" _
                                          & vbCrLf
                    Else
                        MsgBox "Erwarte Variable", vbCritical, "Fehler"
                    End If
                End If
            Else

            End If

        Next i

    End If

    If IsIf = True Then
        OutPut = OutPut & "pop ax" & vbCrLf & "pop bx" & _
            vbCrLf & "cmp ax, bx" & vbCrLf & "jne _if_" & Ifs & _
            vbCrLf
    End If

End Sub

Function GetPrecedence(ByVal Symb As String) As Byte

    Select Case LCase(Symb)
        Case "="
            GetPrecedence = 1
```

```
        Case "+", "-", "&"
            GetPrecedence = 2
        Case "*", "/"
            GetPrecedence = 3
        Case "(", ")"
            GetPrecedence = 4
        Case Else
            GetPrecedence = 0
    End Select

End Function

Function ReservePolishNotation(ByVal strLine As String) As String()

    On Error GoTo ErrHandle

    Dim i As Integer, SymbNo As Integer
    Dim Stack As New clsStack
    Dim Symb() As String
    Dim Tokens

    Stack.ClearStack

    Tokens = Scan("( " & strLine & " )")

    If UBound(Tokens) > 0 Then

        ReDim Symb(UBound(Tokens))

        For i = 0 To UBound(Tokens)

            Select Case Tokens(i)

                Case "("
                    Stack.Push Tokens(i)

                Case ")"
                    Do While Stack.Peek <> "("
                        Symb(SymbNo) = Stack.Pop
                        SymbNo = SymbNo + 1
                    Loop

                    If Stack.Peek = "(" Then Stack.Pop

                Case "+", "-", "*", "/", "="
                    Do
                        If Stack.StackSize = 0 Or Stack.Peek = "(" Or _
                          GetPrecedence(Stack.Peek) < _
                          GetPrecedence(Tokens(i)) Then
                            Stack.Push Tokens(i): Exit Do
                        Else
                            Symb(SymbNo) = Stack.Pop: SymbNo = SymbNo + 1
                        End If
                    Loop

                Case Else
                    Symb(SymbNo) = Tokens(i): SymbNo = SymbNo + 1

            End Select

        Next i

        ReDim Preserve Symb(SymbNo - 1)
```

```
        ReservePolishNotation = Symb

    End If

    Exit Function

ErrHandle:
    MsgBox Err.Description, vbCritical, "Fehler"

End Function
```

mdlASM.bas:

```
Global Const ASM_END = "mov ah, 4ch" & vbCrLf & _
                       "int 21h" & vbCrLf

Global Const ASM_BEGIN = "org 100h" & vbCrLf & _
                         "mov ax, cs" & vbCrLf & _
                         "mov es, ax" & vbCrLf & _
                         "mov ds, ax" & vbCrLf

Global Const ASM_ADD = "_asm_add:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "add ax, bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_SUB = "_asm_sub:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "sub ax, bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_MUL = "_asm_mul:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "xor dx, dx" & vbCrLf & _
                vbTab & "mul bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_DIV = "_asm_div:" & vbCrLf & _
                vbTab & "pop cx" & vbCrLf & _
                vbTab & "pop bx" & vbCrLf & _
                vbTab & "pop ax" & vbCrLf & _
                vbTab & "xor dx, dx" & vbCrLf & _
                vbTab & "div bx" & vbCrLf & _
                vbTab & "push ax" & vbCrLf & _
                vbTab & "push cx" & vbCrLf & _
                vbTab & "ret" & vbCrLf

Global Const ASM_PRINTNUMBER1 = "_print_number: " & vbCrLf & _
```

```
                        "mov si, di" & vbCrLf & _
                        "mov ax, bx" & vbCrLf & _
                        "or ax, ax" & vbCrLf & _
                        "jnz .s1" & vbCrLf & _
                        "mov byte [si], 48" & vbCrLf & _
                        "inc si" & vbCrLf & _
                        "jmp short .fertig" & vbCrLf & _
                        ".s1:" & vbCrLf & _
                        "mov cx, 5" & vbCrLf & _
                        "mov bx, 10000" & vbCrLf & _
                        "xor di, di" & vbCrLf & _
                        ".nochmal:" & vbCrLf & _
                        "xor dx, dx" & vbCrLf & _
                        "div bx" & vbCrLf & _
                        "or di, di" & vbCrLf & _
                        "jnz .s4" & vbCrLf & _
                        "or al, al" & vbCrLf & _
                        "jz .s5" & vbCrLf & _
                        ".s4:" & vbCrLf & _
                        "add al, 48" & vbCrLf & _
                        "mov [si], al" & vbCrLf & _
                        "inc si" & vbCrLf & _
                        "inc di" & vbCrLf & _
                        ".s5:" & vbCrLf
Global Const ASM_PRINTNUMBER2 = "push dx" & vbCrLf & _
                        "mov ax, bx" & vbCrLf & _
                        "xor dx, dx" & vbCrLf & _
                        "mov bx, 10" & vbCrLf & _
                        "div bx" & vbCrLf & _
                        "mov bx, ax" & vbCrLf & _
                        "Pop ax" & vbCrLf & _
                        "loop .nochmal" & vbCrLf & _
                        ".fertig:" & vbCrLf & _
                        "mov byte [si], '$'" & vbCrLf & _
                        "mov di, Buffer" & vbCrLf & _
                        "call _print" & vbCrLf & _
                        "ret" & vbCrLf

Global Const ASM_PRINT = "_print:" & vbCrLf & _
                  "mov dx, di" & vbCrLf & _
                  "Push cs" & vbCrLf & _
                  "Pop ds" & vbCrLf & _
                  "mov ah, 9" & vbCrLf & _
                  "int 21h" & vbCrLf & _
                  "ret" & vbCrLf
```

Schlussresüme

Dieses kurze Tutorial konnte nur einen sehr kleinen Einblick in die Welt des Compilerbaus werfen. Ziel war es vor allem einen kleinen Compiler zu schreiben womit auch Neulinge ohne große Vorkenntnisse einen Einstieg in die Compilerentwicklung erhalten. Wenn man mehr über dieses Thema erfahren will kommt man jedoch nicht um die einschlägige Fachliteratur herum.

Literaturhinweise

Explizit verweise ich in dem Tutorial auf keine Quellen. Das Wissen ist zum größten Teil in langer Zeit gelesen und selbst „beigebracht“ worden. Allerdings möchte ich zumindest einige Bücher nennen, die mir auf dem Weg zum Verständnis eines Compilers weitergeholfen haben:

- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, “Compilerbau Teil 1”, Oldenbourg Verlag, Dezember 1999
- Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, “Compilerbau Teil 2”, Oldenbourg Verlag, Januar 1999
- Niklaus Wirth, „Grundlagen und Techniken des Compilerbaus“, Oldenbourg Verlag, Januar 1996

Niklaus Wirth, „Compilerbau“, Teubner Verla