

<input type="checkbox"/> Gr. 1, Dr. G. Kronberger	Name	Andreas Roither	Aufwand in h	6 h
<input type="checkbox"/> Gr. 2, Dr. H. Gruber				
<input checked="" type="checkbox"/> Gr. 3, Dr. D. Auer	Punkte		Kurzzeichen Tutor / Übungsleiter	/

## Einfacher Parser Generator

(24 Punkte)

Sie haben mittlerweile sicher festgestellt, dass es ziemlich eintönig – um nicht zu sagen *fad* – ist, für eine gegebene Grammatik mithilfe des „Kochrezepts“ für den rekursiven Abstieg einen Syntaxanalysator (*parser*) zu schreiben. Natürlich kann man diese Tätigkeit automatisieren: Man erhält dann ein Werkzeug, das im Compilerbau seit vielen Jahren zum Standardrepertoire gehört, einen so genannten *Parser-Generator*, das berühmteste Beispiel dafür ist sicher *yacc* (alias *bison*). Sie haben bald die Gelegenheit ein ähnliches Werkzeug, nämlich *Coco-2*, kennen zu lernen (am Do 18. 5. um 18 Uhr im HS 3, Einladung folgt).

Ihre Aufgabe besteht nun aber darin, selbst einen einfachen Parser-Generator namens *ParseGen* zu erstellen, der mit der Kommandozeile

```
ParseGen Grammar.syn Grammar.pas
```

aufgerufen werden kann. *ParseGen* muss dann für alle Regeln der Grammatik in der Eingabedatei *Grammar.syn* die entsprechenden Erkennungsprozeduren in Pascal erzeugen und diese in die Ausgabedatei *Grammar.pas* schreiben.

Die Syntax für die Eingabedatei, nennen wir sie *ParseGen*-EBNF, hat eine Erweiterung gegenüber der Wirth'schen EBNF und wird durch folgende Grammatik (hier noch in der "klassischen" EBNF-Notation formuliert) definiert:

```
Grammar = Rule { Rule } .
Rule    = ident '=' Expr '.' .
Expr    = Term | '<' Term { '|' Term } '>' .
Term    = Fact { Fact } .
Fact    = ident | '(' Expr ')' | '[' Expr ']' | '{' Expr '}' .
```

Grundsätzlich entspricht die oben beschriebene Syntax der "klassischen" EBNF, es gibt jedoch zwei Unterschiede (oben fett und rot dargestellt):

1. Bei Alternativen, die durch das Symbol '|' getrennt werden (z. B. in der Regel *Expr*), muss die *gesamte* Alternativenkette zwischen '<' und '>' gesetzt werden.
2. Als Terminalsymbole sind nur Bezeichner (*ident* in obiger Regel *Fact*) erlaubt, keine einzelnen Zeichen oder Zeichenketten mehr.

Darüber hinaus *müssen* die Bezeichner von Nonterminalsymbolen mit einem Großbuchstaben und die Bezeichner von Terminalsymbolen oder -klassen mit einem Kleinbuchstaben beginnen, um anhand des ersten Buchstabens die Art des jeweiligen Symbols ermitteln zu können.

*Beispiel:*

Folgende Tabelle stellt die Grammatik für einfache arithmetische Ausdrücke in beiden Notationen gegenüber (Unterschiede fett dargestellt).

in "klassischer" EBNF	in <i>ParseGen</i> -EBNF
Expr = Term { '+' Term } .	Expr = Term { <b>plus</b> Term } .
Term = Fact { '*' Fact } .	Term = Fact { <b>times</b> Fact } .
Fact = number   '(' Expr ')' .	Fact = < number   <b>leftPar</b> Expr <b>rightPar</b> > .

Gehen Sie für die Lösung dieser Aufgabe in folgenden vier Schritten vor:

1. Erstellen Sie für die oben angegebene Grammatik *Grammar* der *ParseGen*-EBNF einen lexikalischen Analysator (in Form der beiden Prozeduren *NewCh* und *NewSy*) und – gemäß Kochrezept für den rekursiven Abstieg – einen Syntaxanalysator, so dass Sie Eingabedateien, welche Grammatiken in der Notation *ParseGen*-EBNF enthalten (z. B. die Grammatik *Expr* in der linken Hälfte der oberen Tabelle) analysieren können.
2. Überlegen Sie, welche Pascal-Anweisungen Sie für die jeweiligen syntaktischen Konstruktionen in den Grammatikregeln erzeugen müssen. Hierbei werden Sie feststellen, dass gewisse Informationen, die Sie dafür benötigen, nicht oder noch nicht vorhanden sind. So wird z. B. im Kochrezept folgendes vorgeschlagen:

```

... = { a | b | C } c    WHILE (sy = aSy) OR (sy = bSy) OR (sy = dSy) DO BEGIN
C   = d ...              ...
                           END; (*WHILE*)
                           IF sy <> cSy THEN BEGIN success := FALSE; Exit END;
                           NewSy;

```

Sie werden sich wohl oder übel darauf beschränken müssen, etwas in der Art

```

WHILE (sy = ??? ) DO BEGIN
    ...
END; (*WHILE*)
IF sy <> cSy THEN BEGIN success := FALSE; Exit END;
NewSy;

```

zu erzeugen, und es dem Anwender von *ParseGen* überlassen, die fehlenden Vergleiche (in der *WHILE*-Bedingung die *???*) in den erzeugten Pascal-Quelltexten händisch einzusetzen.

3. Erweitern Sie die oben angegebene Grammatik *Grammar* mit Attributen und semantischen Aktionen (im wesentlichen *Write*- und *WriteLn*-Anweisungen, die Pascal-Codestücke erzeugen).
4. Bauen Sie die Attribute und die semantischen Aktionen aus Punkt 3 in Ihr Pascal-Programm aus Punkt 1 ein – und Sie haben die fertige Lösung.

Um Ihnen die Aufgabe zu verdeutlichen, finden Sie im Moodle-Kurs in der ZIP-Datei *ForParseGen* (unter *Diverse Materialien*) zwei Beispiele für Grammatikdateien (*EBNF.syn* und *Expr.syn*) als Eingabedateien sowie die Ergebnisse (in Form von *pas*-Dateien), die (so oder so ähnlich) von Ihrem Werkzeug *ParseGen* erzeugt werden sollen.

Wenn Sie *ParseGen* fertig gestellt haben, können Sie versuchen, mithilfe von *ParseGen* den Quelltext für den Parser von *ParseGen* selbst aus einer entsprechenden Datei *ParseGen.syn* herzustellen. Diesen Prozess nennt man im Compilerbau übrigens *Bootstrapping*.

# Übung 6

## Aufgabe 1

### Lösungsidee

Zuerst wird ein lexikalischer Analysator erstellt, mit dessen Hilfe wird das überprüfende File analysiert. Dabei wird bei besonderen Zeichen ( Abfrage in Case Statement ) das aktuelle Symbol auf einen Enum Typ gesetzt. Das aktuelle Symbol wird dann beim Parser verwendet um eine Syntaxanalyse zu ermöglichen. Der Parser wird durch Rekursiven Abstieg realisiert. Durch Aufrufen der WritePas Funktion wird der Pascal Syntax des generierten Parsers (bzw. den Parser Prozeduren) die Ausgabedatei schrittweise aufbaut und raus geschrieben.

```
1  (* Lex                                03.05.17 *)
2  (* Lexikalischer Analysator (scanner)UE6 *)
3
4  UNIT Lex;
5  INTERFACE
6      TYPE
7          SymbolCode = (errorSy, (* error symbol *)
8                          leftParSy, rightParSy,
9                          eofSy, periodSy, equalsSy,
10                         leftCompSy, rightCompSy, optSy,
11                         leftCurlySy, rightCurlySy, identSy);
12
13  VAR
14      sy: SymbolCode;
15      syLnr, syCnr : INTEGER;
16      identStr : STRING;
17
18  PROCEDURE InitScanner(srcName: STRING; VAR ok: BOOLEAN);
19  PROCEDURE NewCh;
20  PROCEDURE NewSy;
21
22  IMPLEMENTATION
23  CONST
24      EF = Chr(0);
25  VAR
26      srcLine: STRING;
27      ch: CHAR;
28      chLnr, chCnr : INTEGER;
29      srcFile: TEXT;
30
31  PROCEDURE InitScanner(srcName: STRING; VAR ok: BOOLEAN);
32  BEGIN
33      Assign(srcFile, srcName);
34      {$I-}
35      Reset(srcFile);
36      {$I+}
37      ok := IOResult = 0;
```

```
37     IF ok THEN BEGIN
39         srcLine := '';
40         chLnr := 0;
41         chCnr := 1;
42         NewCh;
43         NewSy;
44     END;
45 END;

47 PROCEDURE NewCh;
48 BEGIN
49     IF chCnr < Length(srcLine) THEN BEGIN
50         chCnr := chCnr + 1;
51         ch := srcLine[chCnr];
52     END
53     ELSE BEGIN (* new line *)
54         IF NOT Eof(srcFile) THEN BEGIN
55             ReadLn(srcFile, srcLine);
56             Inc(chLnr);
57             chCnr := 0;
58             (* da leerzeichen überlesen werden wird in newsy gleich der
59                nächste char eingelesen *)
60             ch := ' ';
61         END
62         ELSE BEGIN
63             Close(srcFile);
64             ch := EF;
65         END;
66     END;
67 END;

69 PROCEDURE NewSy;
70 VAR
71     code: INTEGER;
72 BEGIN
73     WHILE ch = ' ' DO BEGIN
74         NewCh;
75     END;
76     syLnr := chLnr;
77     syCnr := chCnr;

79     CASE ch OF
80         EF: BEGIN
81             sy := eofSy;
82         END;
83         '(': BEGIN
84             sy := leftParSy;
```

```
85     NewCh;
      END;
87   ')' : BEGIN
      sy := rightParSy;
89     NewCh;
      END;
91   ' ': BEGIN
      sy := periodSy;
93     NewCh;
      END;
95   '<': BEGIN
      sy := leftCompSy;
97     NewCh;
      END;
99   '>': BEGIN
      sy := rightCompSy;
101     NewCh;
      END;
103   '|': BEGIN
      sy := optSy;
105     NewCh;
      END;
107   '{': BEGIN
      sy := leftCurlySy;
109     NewCh;
      END;
111   '}': BEGIN
      sy := rightCurlySy;
113     NewCh;
      END;
115   '=' : BEGIN
      sy := equalsSy;
117     NewCh;
      END;
119   'a' .. 'z', 'A' .. 'Z': BEGIN
      identStr := '';
121     WHILE ch IN ['a' .. 'z', 'A' .. 'Z', '_'] DO BEGIN
      identStr := Concat(identStr, ch);
123     NewCh;
      END;
125     sy := identSy;
      END;
127 ELSE
      sy := errorSy;
129 END;
      END;
131 END. (* Lex *)
```

Lex.pas

```
1  (* Parser                                03.05.17 *)
2  (* Syntax Analysator (scanner) UE6      *)
3
4  UNIT Parser;
5  INTERFACE
6
7      VAR
8          success: BOOLEAN;
9
10     PROCEDURE S;
11     PROCEDURE InitParser(outputFileName: STRING; ok: BOOLEAN);
12
13 IMPLEMENTATION
14     USES
15         Lex;
16
17     VAR
18         outputFile : TEXT;
19         tab : STRING;
20
21     TYPE
22         Mode = (printTitle, printHead, printEnd,
23                 printCurlyBegin, printCurlyEnd,
24                 printCompBegin, printOpt, printCompEnd,
25                 printIsNotSy, printNonTerminal, printTerminal);
26
27     (* Init parser with the file to write to*)
28     PROCEDURE InitParser(outputFileName: STRING; ok: BOOLEAN);
29     BEGIN
30         Assign(outputFile, outputFile);
31         {$I-}
32         Rewrite(outputFile);
33         {$I+}
34         ok := IOResult = 0;
35     END;
36
37     (* tab control *)
38     PROCEDURE IncTab;
39     BEGIN
40         tab := tab + ' ';
41     END;
42
43     PROCEDURE DecTab;
44     BEGIN
45         Delete(tab, Length(tab)-1, 2);
46     END;
```

```
49  (* Check sy;
    returns false if sy is not expected sy *)
51  FUNCTION SyIsNot(expectedSy: SymbolCode): BOOLEAN;
    BEGIN
53      success := success AND (sy = expectedSy);
    SyIsNot := NOT success;
55  END;

57  (* write pascal syntax to outputfile *)
    PROCEDURE WritePas(m: Mode; msg: STRING);
59  BEGIN
    CASE m OF
61      printTitle: BEGIN
        WriteLn(outputFile, '(* PARSER Generated *)');
63      END;
    printHead: BEGIN
65        tab := ' ';
        WriteLn(outputFile, 'PROCEDURE ', msg, ';');
67        WriteLn(outputFile, 'BEGIN');
        END;
    printEnd: BEGIN
69        DecTab;
        WriteLn(outputFile, 'END;');
71      END;
    printCurlyBegin: BEGIN
73        WriteLn(outputFile, tab, 'WHILE sy = .... DO BEGIN');
75        IncTab;
        END;
    printCurlyEnd: BEGIN
77        DecTab;
        WriteLn(outputFile, tab, 'END;');
79      END;
    printCompBegin: BEGIN
81        WriteLn(outputFile, tab, 'IF sy = .... THEN BEGIN');
83        IncTab;
        END;
    printOpt: BEGIN
85        DecTab;
        WriteLn(outputFile, tab, 'END ELSE');
87      END;
    printCompEnd: BEGIN
89        IncTab;
        WriteLn(outputFile, tab, 'success := FALSE');
91      END;
    printIsNotSy: BEGIN
93        WriteLn(outputFile, 'FUNCTION SyIsNot(expectedSy: Symbol):',
95          'BOOLEAN;');
```

```

        WriteLn(outputFile, 'BEGIN');
        WriteLn(outputFile, ' success:= success AND (sy = expectedSy);');
        WriteLn(outputFile, ' SyIsNot := NOT success;');
        WriteLn(outputFile, 'END;');
        WriteLn(outputFile);
    END;
printNonTerminal: BEGIN
    WriteLn(outputFile, tab, msg, '; IF NOT success THEN EXIT;');
    END;
printTerminal: BEGIN
    WriteLn(outputFile, tab, 'IF SyIsNot(', msg, 'Sy) THEN EXIT;');
    WriteLn(outputFile, tab, 'NewSy;');
    END;
END;
END;

(*===== PARSER =====*)
PROCEDURE Seq;    FORWARD;
PROCEDURE Stat;   FORWARD;
PROCEDURE Fact;   FORWARD;

PROCEDURE S;
BEGIN
    success := TRUE;
    Seq; IF NOT success OR SyIsNot(eofSy) THEN BEGIN
        WriteLn('----- Error -----');

        WriteLn('Error in line ', syLnr, ' at position ', syCnr)
    END
    ELSE
        WriteLn('Finished writing to output file');
        WriteLn('Sucessfully parsed');
    Close(outputFile);
END;

PROCEDURE Seq;
BEGIN
    WriteLn('Creating output..');
    WritePas(printTitle, '');
    WritePas(printIsNotSy, '');

    WHILE sy <> eofSy DO BEGIN
        IF SyIsNot(identSy) THEN EXIT;
        WritePas(printHead, identStr);
        NewSy;
        IF SyIsNot(equalsSy) THEN EXIT;
        NewSy;

```



```

145     Stat; IF NOT success THEN EXIT;
146     IF SyIsNot(periodSy) THEN EXIT;
147     NewSy;
148     WritePas(printEnd, '');
149     WriteLn(outputFile);
150 END;
151 END;
152
153 PROCEDURE Stat;
154 BEGIN
155     Fact; IF NOT success THEN EXIT;
156     WHILE (sy = identSy) OR (sy = optSy) OR (sy = leftCompSy) OR
157           (sy = leftCurlySy) OR (sy = leftParSy) DO BEGIN
158         Fact; IF NOT success THEN EXIT;
159     END;
160 END;
161
162 PROCEDURE Fact;
163 BEGIN
164     CASE sy OF
165         identSy: BEGIN
166             (* term or non-term symbol check *)
167             IF identStr[1] IN ['A'..'Z'] THEN
168                 WritePas(printNonTerminal, identStr)
169             ELSE
170                 WritePas(printTerminal, identStr);
171             NewSy;
172         END;
173         optSy: BEGIN
174             WritePas(printOpt, '');
175             WritePas(printCompBegin, '');
176             NewSy;
177         END;
178         leftCompSy: BEGIN
179             NewSy;
180             WritePas(printCompBegin, '');
181             Stat; IF NOT success THEN EXIT;
182             IF sy <> rightCompSy THEN BEGIN success := FALSE; EXIT; END;
183             WritePas(printOpt, '');
184             WritePas(printCompEnd, '');
185             NewSy;
186         END;
187         leftCurlySy: BEGIN
188             NewSy;
189             WritePas(printCurlyBegin, '');
190             Stat; IF NOT success THEN EXIT;
191             IF sy <> rightCurlySy THEN BEGIN success := FALSE; EXIT; END;
192             WritePas(printCurlyEnd, '');

```

```
193         NewSy;  
194         END;  
195     leftParSy: BEGIN  
196         NewSy;  
197         Stat; IF NOT success THEN EXIT;  
198         IF sy <> rightParSy THEN BEGIN success := FALSE; EXIT; END;  
199         NewSy;  
200     END;  
201 END;  
202 BEGIN  
203     tab := ' ';  
END.
```

Parser.pas

Der Parser analysiert das Input File solange bis entweder ein Fehler gefunden wurde oder erfolgreich ohne Fehler sy auf eofSy ( End of file symbol ) gesetzt wurde. Damit die Ausgabe Datei nicht komplett ohne Formatierung ist, wird ein tab String verwendet der für den notwendigen Abstand sorgt.

```
(* Test    03.05.17 *)
2 (* Parser + Lexi.  *)

4 PROGRAM Test;
  USES
6   Lex, Parser;

8  VAR
  ok : BOOLEAN;
10  inFileName, outFileName: STRING;

12 BEGIN
  (* Param check *)
14  IF (ParamCount < 2) THEN
  BEGIN
16    WriteLn('Wrong input, try again: ');

18    Write('syn File name > ');
    ReadLn(inFileName);

20    Write('out File name > ');
    ReadLn(outFileName);
22  END
24  ELSE BEGIN
    inFileName := ParamStr(1);
26    outFileName := ParamStr(2);
  END;

28  InitScanner(inFileName, ok);
30  InitParser(outFileName, ok);

32  IF ok THEN S
  ELSE
34    WriteLn('File error')
  END.
```

Test.pas

In Test.pas wird der Lexikalische Analysator und der Parser initialisiert nachdem die Kommandozeilenargumente überprüft wurden. Danach wird die Prozedur S aufgerufen um die Syntaxanalyse und das generieren des Pascal Codes zu initiieren.

Zum Testen wird eine .syn Datei ohne Fehler verwendet und eine .syn Datei die einen Fehler enthält. Hier das erfolgreiche Parsen von Expr.syn.

```
1 Expr = Term { plus Term } .
3 Term = Fact { times Fact } .
5 Fact = < number | leftPar Expr rightPar > .
```

Expr.syn

```
(* PARSER Generated *)
2 FUNCTION SyIsNot(expectedSy: Symbol):BOOLEAN;
BEGIN
4   success:= success AND (sy = expectedSy);
   SyIsNot := NOT success;
6 END;

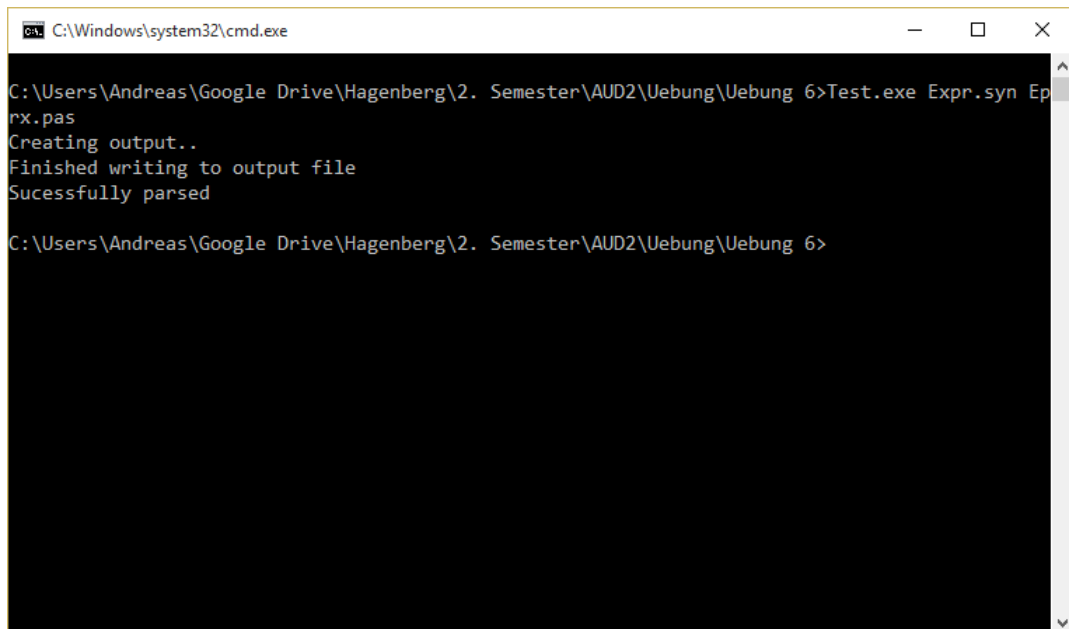
8 PROCEDURE Expr;
BEGIN
10  Term; IF NOT success THEN EXIT;
   WHILE sy = .... DO BEGIN
12    IF SyIsNot(plusSy) THEN EXIT;
    NewSy;
14    Term; IF NOT success THEN EXIT;
   END;
16 END;

18 PROCEDURE Term;
BEGIN
20  Fact; IF NOT success THEN EXIT;
   WHILE sy = .... DO BEGIN
22    IF SyIsNot(timesSy) THEN EXIT;
    NewSy;
24    Fact; IF NOT success THEN EXIT;
   END;
26 END;

28 PROCEDURE Fact;
BEGIN
30  IF sy = .... THEN BEGIN
    IF SyIsNot(numberSy) THEN EXIT;
32    NewSy;
   END ELSE
34  IF sy = .... THEN BEGIN
    IF SyIsNot(leftParSy) THEN EXIT;
36    NewSy;
    Expr; IF NOT success THEN EXIT;
38    IF SyIsNot(rightParSy) THEN EXIT;
```

```
    NewSy;  
40  END ELSE  
    success := FALSE  
42 END;
```

output.pas



```
C:\Windows\system32\cmd.exe  
C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD2\Uebung\Uebung 6>Test.exe Expr.syn Expr.pas  
Creating output..  
Finished writing to output file  
Sucessfully parsed  
C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD2\Uebung\Uebung 6>
```

Abbildung 1: Console Output

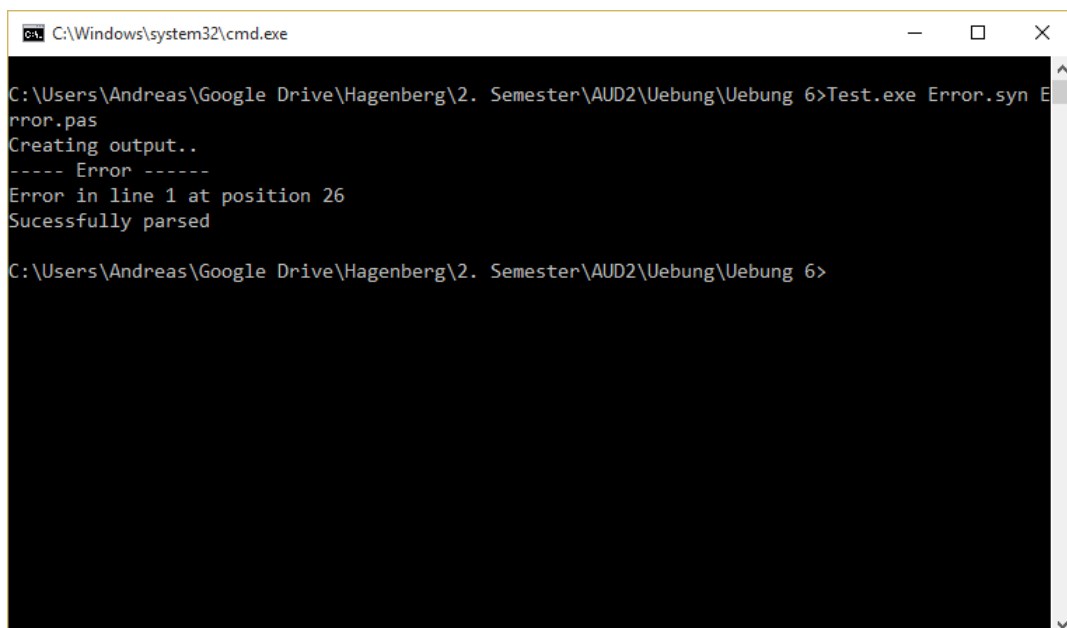
Das Parsen der Error.syn Datei mit dem eingebauten Fehler war nicht erfolgreich. Das Ausgabe File ist durch den Fehler beim Parsen unvollständig.

```
1 Expr = Term { plus Term } .
3 Term = Fact { times Fact } .
5 Fact = < number | leftPar Expr rightPar > .
```

Error.syn

```
1 (* PARSER Generated *)
FUNCTION SyIsNot(expectedSy: Symbol):BOOLEAN;
3 BEGIN
    success:= success AND (sy = expectedSy);
5 SyIsNot := NOT success;
END;
7
PROCEDURE Expr;
9 BEGIN
    Term; IF NOT success THEN EXIT;
11 WHILE sy = .... DO BEGIN
    IF SyIsNot(plusSy) THEN EXIT;
13 NewSy;
    Term; IF NOT success THEN EXIT;
15 END;
```

Error.pas



```
C:\Windows\system32\cmd.exe
C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD2\Uebung\Uebung 6>Test.exe Error.syn Error.pas
Creating output..
Error in line 1 at position 26
Sucessfully parsed
C:\Users\Andreas\Google Drive\Hagenberg\2. Semester\AUD2\Uebung\Uebung 6>
```

Abbildung 2: Console Output