Abgabetermin: Mi in der KW 19

☐ **Gr. 1,** Dr. G. Kronberger **Name** _____ **Aufwand in h** _____
☐ **Gr. 2,** Dr. H. Gruber
☐ **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 *ParseGen*-EBNF
Expr = Term { '+' Term } .	Expr = Term { **plus** Term } .
Term = Fact { '*' Fact } .	Term = Fact { **times** Fact } .
Fact = number | '(' Expr ')' .	Fact = **<** number | **leftPar** Expr **rightPar >** .

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
   C = d ...

   ```
   WHILE (sy = aSy) OR (sy = bSy) OR (sy = dSy) DO BEGIN
     …
   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*.