FCW 1x Übung zu Formale Sprachen, Compiler- und Werkzeugbau 1 — WS 2016/17, Übung 1

Abgabetermin: in der KW 43

☐ **Gr. 1,** Dr. H. Dobler
☐ **Gr. 2,** Dr. G. Kronberger

Name _____ **Aufwand in h** _____

Punkte _____ **Übungsleiter** _____

1. Grammatiken – Grundbegriffe (8 Punkte)

Die folgende Grammatik *G*(*DataStat*) beschreibt in vereinfachter Form den Aufbau der *DATA*[1] Anweisung der Programmiersprache Fortran 77:

```
DataStat       = "DATA" DataDecl DataDeclRest.
DataDeclRest   = ε | DataDeclRest ["," ] DataDecl.
DataDecl       = DataNameList "/" DataValueList "/".
DataNameList   = DataName | DataName "," DataNameList.
DataName       = id | DataDoList.
DataValueList  = DataValue | DataValueList "," DataValue.
DataValue      = ( (num | id) ["*" (id | ("+"|"-"|ε) num | str)] ) |
                 ( ["+"|"-"] num | str ).
DataDoList     = "(" ( id "(" id {"," id} ")" | DataDoList )
                 DataDoListRest ")".
DataDoListRest = ε |
                 DataDoListRest "," (id ( "(" expr { "," expr} ")"
                                        | "=" expr "," expr (ε | expr)
                                        )
                                    | DataDoList
                                    ).
```

a) Bestimmen Sie die Mengen V_T und V_N.

b) Geben Sie den/die kürzesten Satz/Sätze an, den/die man mit dieser Grammatik bilden kann.

c) Ermitteln Sie alle rekursiven Nonterminalsymbole. Geben Sie für jedes dieser Nonterminalsymbole an, ob es direkt- oder indirekt- und links-, zentral- oder rechtsrekursiv ist.

d) Transformieren Sie die gegebene Grammatik in das Regelsystem der formalen Sprachen. Welche Grammatikschreibweise halten Sie für lesbarer? Begründen Sie Ihre Antwort.

e) Zeichnen Sie den Syntaxbaum für folgenden Satz (verwenden Sie dazu die gegebene Grammatik *G*(*DataStat*)):

```
DATA id, id / num * str /, ( id(id), id = expr, expr ) / num * num /
```

Gibt es mehrere Syntaxbäume für diesen Satz? (Mit Begründung!)

2. Konstruktion einer Grammatik (4 Punkte)

Konstruieren Sie eine Grammatik (in der Schreibweise des Regelsystems der formalen Sprachen) für die Menge aller ungeraden ganzen Dezimalzahlen mit optionalem Vorzeichen. Die Zahlen dürfen keine führenden Nullen enthalten. Geben Sie Ihre Grammatik nun auch in Wirth'scher EBNF mit möglichst wenig Regeln an.

[1] Eine genauere Erklärung dieser Anweisung finden Sie am Ende des Übungszettels.

3. Oo-Implementierung von Grammatiken (4 + 6 + 2 Punkte)

Studieren Sie die oo Implementierung von Grammatiken in *FormalLanguagesForStudents* aus dem *moodle*-Kurs. Das UML-Klassendiagramm dafür finden Sie in der VL-Präsentation für den FS-Teil.

a) In der Vorlesung wurden Algorithmen auf Grammatiken besprochen, insbesondere jener zur Beseitigung leerer Alternativen. Implementieren Sie diesen Algorithmus in Form einer Funktion

```
Grammar *epsilonFreeGrammarOf(Grammar *g);
```

Testen Sie Ihre Implementierung mit der Beispielgrammatik im Foliensatz der VL (auf S. 26).

b) Die Sprache *L* einer Grammatik *G* mit dem Statzsymbol *S*, also *L*(*G*(*S*)), ist nichts anderes als die Menge aller terminalen Ketten σ, die sich aus *S* mit den Regeln aus der Grammatik ableiten lassen. Eine Kette wird durch ein Objekt der Klasse *Sequence* repräsentiert. Entwickeln Sie eine Klasse *Language*, die eine Menge solcher Ketten speichert und eine Funktion

```
Language *generateLanguage(Grammar *g, int maxLen);
```

die alle Sätze bis zur Länge *maxLen* einem *Language*-Objekt erzeugt.

c) Wenn Sie in die Klasse *Language* noch eine Methode

```
bool Language::hasSentence(Sequence *s) const;
```

einbauen, haben Sie einen einfachen Mechanismus für die Syntaxanalyse.

Testen Sie Ihre Implementierungen von b) und c) mit folgender Grammatik *G*(*S*):

```
S  → aB  |  bA
A  → a   |  aS  |  bAA
B  → b   |  bS  |  aBB
```

indem Sie alle Sätze σ dieser Grammatik erzeugen, für die $|\sigma| \leq 6$ gilt und beantworten Sie folgende Fragen: Weisen diese Sätze eine besondere Eigenschaft auf? Haben alle Sätze dieser Grammatik diese Eigenschaft? Kann man diese Eigenschaft schon aus der Grammatik ableiten?

Hintergrundinformation zur Aufgabe 1

Die *DATA*-Anweisung der Programmiersprache Fortran 77 wird zur Initialisierung von Variablen, Feldern, Feldelementen und Zeichenketten benutzt. Die Anweisung ist nicht ausführbar und kann deshalb nur im Deklarationsteil eines Fortran-Programms verwendet werden. Struktur vereinfacht dargestellt:

```
DATA var_list / val_list / { , var_list / val_list / }
```

Wobei *var_list* für durch Komma getrennte Listen von Variablen und *val_list* für durch Komma getrennte Listen von Werten steht. Die Werte sind entweder Konstanten oder Gebilde der Form *r*c* sind, wobei *r* die Anzahl der Wiederholungen (*repetitions*) der Konstante *c* angibt.

Im folgenden Beispiel werden die Variablen *A* – *L* deklariert und dann mit Initialwerten belegt:

```
BLOCKDATA SETUP
INTEGER A, B, C
REAL    I, J, K, L
COMMON  /AREA1/ A, B, C
COMMON  /AREA2/ I, J, K, L
DATA    A, B, C, I, J, K, L / 0, 1, 2, 10.0, -20.0, 30.0, -40.0 /
END
```

1 Grammatiken - Grundbegriffe

Dieser Teil der Dokumentation behandelt die Aufgabe 1 der ersten Übung.

1.1 Die Mengen V_T und V_N

V_N = { DataStat, DataDeclRest, DataDecl, DataNameList, DataName, DataValueList, DataValue, DataDoList, DataDoListRest }

Alle Nichtterminalsymbole befinden sich links in der Grammatik, wobei das Nichtterminalsymbol *DataStat* das Satzsymbol ist.

V_T = { “DATA”,“,”, “/”, “(”, “)”,“=”, “*”, “+”, “-”, “=”, expr, id, num, str }

Alle Terminalsymbole kommen nicht auf der linken Seite der Grammatik vor und können nicht weiter abgeleitet werden. Das Symbol “ϵ” ist ein Metasymbol, dass die leere Kette repräsentiert und ist weder ein Nichtterminalsymbol oder ein Terminalsymbol.

$V = V_T \cup V_N$

Die Menge V ist das Alphabet der Grammatik und ist die Vereinigung der Menge der Nichtterminalsymbole und der Menge der Terminalsymbole. Da das Symbol ϵ ein Metasymbol ist, ist es nicht Teil der Grammatik und auch kein Teil des Alphabets der Grammatik.

1.2 Kürzeste Sätze der Grammatik

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” num “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” num “/”

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” id “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” id “/”

DataStat $\overset{L}{\Rightarrow}$ “DATA” DataDecl DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataNameList “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” DataName “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValueList “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” DataValue “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” str “/” DataDeclRest
$\overset{L}{\Rightarrow}$ “DATA” id “/” str “/”

Das Nichtterminalsymbol *DataValue* kann in drei Varianten abgeleitet werden, *id*, *num* und *str*. Die Satzlänge bleibt bei jeder der drei Ableitungen des Nichtterminalsymbols *DataValue* gleich.

1.3 Rekursionen der Nichtterminalsymbole

Nichtterminalsymbol	direkt/indirekt rek.	links/rechts rek.
DataDeclRest	direkt rek.	links rek.
DataDecl	-	-
DataNameList	direkt rek.	rechts rek.
DataName	-	-
DataValueList	direkt rek.	links rek.
DataValue	-	-
DataDoList	direkt rek.	zentral rek.
DataDoList	indirekt rek.	zentral rek.
DataDoListRest	indirekt rek.	zentral rek.

Die beiden indirekten Rekursionen der Nichtterminalsymbole *DataDoList* und *DataDoListRest* ergeben sich aus der Tatsache, dass wenn es eine indirekte Rekursion gibt, es auch eine zweite Rekursion geben muss. Ich gehe davon aus, dass ich alle Rekursionen gefunden habe, wobei angemerkt sei, dass es sehr schwer ist Rekursionen aus einer Grammatik auszulesen.

1.4 Transformation in das Regelsystem der formalen Sprachen

DataStat → DATA DataDecl DataDeclRest
DataDeclRest → ϵ | DataDeclRest , DataDecl | DataDeclRest DataDecl
DataDecl → DataNameList / DataValueList /
DataNameList → DataName | DataName , DataNameList
DataName → id | DataDoList
DataValueList → DataValue | DataValueList , DataValue
DataValue → num DerefValue | id DerefValue | NumOrStr
NumValue → num | + num | - num
NumOrStr → NumValue | str
DerefValue → ϵ | * id | * NumOrStr
DataDoList → (IdList DataDoListRest) | (DataDoList DataDoListRest)
CommaId → ϵ | , id CommaId
IdList → id (id CommaId)
DataDoListRest → ϵ | DataDoListRest , DoListRestOpt
ComaExpr → ϵ | , expr CommaExpr
ExprList → (expr CommaExpr)
OptExpr → ϵ | expr
EqualExpr → = expr , expr OptExpr
DoListRestOpt → id ExprList | id EuqlExpr | DataDoList

Um diesen Punkt der Aufgabe 1 zu lösen wurden die Optionen und Schleifen von innen nach außen aufgelöst und in eigene Nichtterminalsybole ausgelagert. Das ist notwendig, da das Regelwerk der formalen Sprachen diese Konstrukte nicht kennt und die Konstrukte Optionen und Schleifen in Oder Konstrukte umgewandelt werden müssen. Bsp.: [num, str] $\equiv \epsilon$ | num | str

Ich halte die Schreibweise der Wirth'schen EBNF für lesbarer, da in dieser Schreibweise mehr Konstrukte wie Optionen und Schleifen zur Verfügung stehen und dadurch die Grammatik durch weniger Regeln definiert werden kann, was die Grammatik übersichtlicher macht.