ARBEITSBLATT ZU FORMALEN SPRACHEN

Aufgabe 1: Gegeben ist die folgende Formale Sprache L(G) mit G = (T, N, P, S). Die Produktionen lauten

ZUWEISUNG	::=	name zuweisungsoperator AUSDRUCK
AUSDRUCK	::=	TERM | TERM strichoperator AUSDRUCK
TERM	::=	FAKTOR | FAKTOR punktoperator TERM
FAKTOR	::=	name | zahl

a) Analysiere die Produktionen und bestimme damit die Terminale (T) und Nicht-Terminale (N) sowie das Startsymbol (S) der Grammatik G.
b) Erzeuge durch fortlaufende Anwendung der Produktionen beginnend mit ZUWEISUNG ein zwei gültige „Worte" – d. h. Zuweisungen – dieser Sprache und schreibe deren zugehörigen Ableitungsbaum auf.
c) Schaue Dir die Ableitungsbäume genauer an. Worin besteht die Leistungsfähigkeit dieses Regelsystems? **Hinweis:** Denke an die mathematisch korrekte Auswertung des Terms auf der rechten Seite der Zuweisung.

Aufgabe 2: Es sollen auch Klammerausdrücke wie der folgende erlaubt sein:

```
x := (a + 1) * c
```

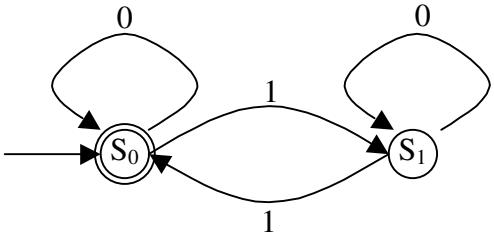
a) Erweitere die obige Grammatik um diese Möglichkeit (T, N, P). Achte dabei darauf, dass auch jetzt die mathematischen Gesetzmäßigkeiten (Klammer- vor Punkt- und Punkt- vor Strichrechnung) erfüllt sind.
b) Erzeuge ein gültiges „Wort" dieser Sprache und gebe auch hierzu den Ableitungsbaum an.

Aufgabe 3: Die erzeugende Anwendung der Regeln von L(G) stellt wie eben gesehen kein so großes Problem dar, die erkennende Anwendung der Regeln ist dagegen etwas komplexer.
a) Gebe wie im Unterricht gesehen den rekursiven Abstieg für die Erkennung des „Wortes" `x := a + b * 2` in tabellarischer Form an. Der Anfang ist unten gemacht.
b) Gebe auch den rekursiven Abstieg für das „Wort" `x := (a + 1) * c` der erweiterten Grammatik an.

Produktionsregel	**Zeichenkette**	**Aktion**	**Token**
		Der Scanner liefert das Token:	name
ZUWEISUNG	x	ist ok. Danach liefert der Scanner das Token:	zuweisungs-operator
	:=	ok und der Scanner liefert das Token	name
	Aufruf der Produktionsregel		
AUSDRUCK			
TERM			
FAKTOR			

ARBEITSBLATT ZU AUTOMATEN

Aufgabe 4: Gegeben ist folgender graphisch dargestellter globaler Automat (Startzustand = S_0):



a) Was leistet dieser Automat, d. h. welche Worte erkennt bzw. akzeptiert der Automat? **Hinweis:** Ein Wort wird akzeptiert, wenn sich der Automat nach Abarbeitung des Wortes in einem Endzustand befindet.
b) Gib die zum Automaten zugehörige Definition an [A = (S, Σ, s, F, R)].

Aufgabe 5: Gegeben ist der folgende globale (endliche) Automat A = (S, Σ, s, F, R) mit

$S = \{ S_0, S_1, S_2, S_3 \}$
$\Sigma = \{ a, b \}$
$s = S_0$
$F = \{ S_2 \}$
$R = \{$
$S_0 \times a \rightarrow S_1$
$S_1 \times a \rightarrow S_1$
$S_1 \times b \rightarrow S_2$
$S_2 \times a \rightarrow S_1$
$S_2 \times b \rightarrow S_2$
$S_i \times$ „anderes Zeichen“ $\rightarrow S_3$
$\}$

a) Zeichne den zum Automat zugehörigen Graphen.
b) Welche Worte erkennt bzw. akzeptiert der angegebene Automat?
c) Erweitere den Automaten um eine weiteren Zustand, so dass er auch die Worte ababcabc und aaacccbbb erkennt. **Hinweis:** Es werden Änderungen bei S, Σ, F und R nötig!

Aufgabe 6: Du kennst hoffentlich (nicht??? übles Laster!!!) die Funktionsweise eines Zigarettenautomaten. Mögliche Eingaben sind 50 Cent („F“), 1 € („E“), 2 € („Z“) und der Geldrückgabeknopf („G“). Der akzeptierende Endzustand S_{Danke} soll erreicht sein, wenn der Betrag passend bezahlt wurde (momentan 3 €). Wurde der Betrag überschritten oder der Geldrückgabeknopf gedrückt, so soll der Fehlerzustand $S_{Geldrückgabe}$ erreicht werden.
a) Zeichne den Graphen des zugehörigen endlichen partiell definierten Automaten und gebe seine Definition an.
b) Erweitere den Automaten so, dass erst ein akzeptierender Endzustand erreicht wird, wenn der Benutzer eine der Sorten Camel („C“), HB („H“) oder Lucky Strike („L“) gewählt hat.

ARBEITSBLATT ZUM SCANNERAUTOMAT

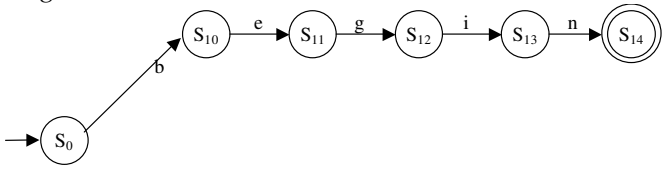
Wir wollen im folgenden einen Compiler für eine Mini-Programmiersprache schreiben, welche durch die folgende Grammatik (T, N, P, S) beschrieben wird:

```
T   = {  prog, start, ende, punkt, semikolon, name, zahl, punktoperator, strichoperator,
         zuweisungsop, klammerauf,, klammerzu }
N   = {  PROGRAMM, BLOCK, ZUWEISUNG, AUSDRUCK, TERM, FAKTOR }
P   = {  PROGRAMM  ::= prog name semikolon BLOCK punkt
         BLOCK     ::= start ZUWEISUNG ende
         ZUWEISUNG ::= name zuweisungsop AUSDRUCK ,
         AUSDRUCK  ::= TERM | TERM strichoperator AUSDRUCK ,
         TERM      ::= FAKTOR | FAKTOR punktoperator TERM ,
         FAKTOR    ::= name | zahl | klammerauf AUSDRUCK klammerzu ,
       }
S   =    PROGRAMM
```

Für die lexikographische Analyse des *Scanners* und der Festsetzung der *Tokens* gelten dabei folgende Regeln:

	Token	*Zeichenkette*	
P' = {	prog	::= p r o g r a m	
	start	::= b e g i n	
	ende	::= e n d	
	punkt	::= **.**	
	semikolon	::= **;**	
	name	::= buchstabe { zeichen }	„{...}“ optional und beliebig oft
	buchstabe	::= a | b | c | ... | y | z	
	zeichen	::= buchstabe | ziffer	
	ziffer	::= 0 | 1 | 2 | ... | 8 | 9	
	zahl	::= (strichoperator) ziffer { ziffer }	„(..)“: optional, aber nur 1 mal
	strichoperator	::= + | −	
	punktoperator	::= * | /	
	zuweisungsop	::= **:** =	
	klammerauf	::= (	
	klammerzu	::=)	
}			

Aufgabe 7: Zeichne einen partiellen Automaten, der sämtliche Schlüsselwörter (*program, begin, end*) sowie alle reservierten Zeichenketten (*punkt, semikolon, strichoperator, punktoperator, zuweisungsop, klammerauf, klammerzu*) akzeptiert. Das Schlüsselwort *begin* wird bereits erkannt.

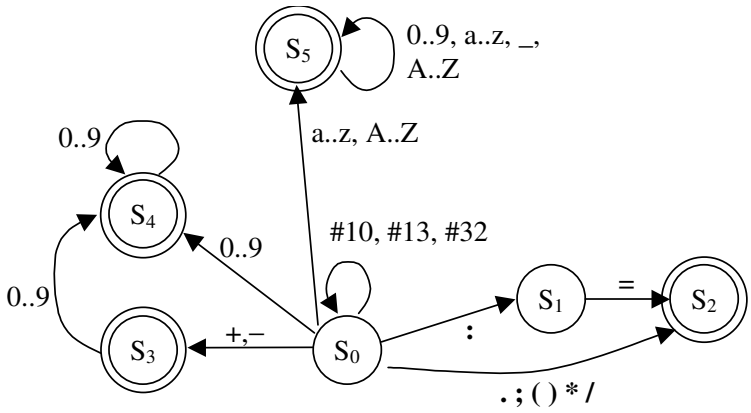


Aufgabe 8: Wie müsste man den Automaten erweitern, damit er auch das Token *zahl* akzeptiert? Achte auf das eventuelle Vorzeichen!

Aufgabe 9: Der Automat soll nun auch das Token *name* akzeptieren. Allerdings sind bereits die Zeichen „p“, „e“ und „b“ im Startzustand belegt durch die Übergänge nach *program, end* und *begin*. Dennoch soll die Programmiersprache *name*n zulassen, welche mit den Buchstaben „p“, „e“ oder „b“ anfangen. Erweitere entsprechend den Automaten.

ARBEITSBLATT ZUM SCANNER-MODUL

Wir haben gesehen, dass sich der Scannerautomat mit Hilfe einer Symboltabelle stark vereinfachen lässt:



Schaue Dir das Projekt im Ordner Compiler an:

Unit Compiler: Zuammenfassung des Module Scanner, Symboltabelle und Tokenliste
Unit U_adt: Der Datentyp TLinList und andere bekannte Listentypen
Unit U_Types: Definition der Symboltabelle und der von TLinList abgeleiteten Tokenliste
Unit Scanner: Die eigentliche Scannerroutine (Realisation des endl. Automaten)

Aufgabe 10: Ergänze in der Unit Scanner den Programmtext für die Realisation des oben abgebildeten endlichen Automaten. Verwende dazu eine Case-Anweisung, welche je nach aktuellem Zustand und je nach aktuell gelesenem Zeichen verzweigt.

Aufgabe 11: Ergänze den Scanner (Unit Scanner und eventuell auch die Unit U_Types) nun um folgende Eigenschaften:

- Auch deutsche Umlaute sollen in Namen akzeptiert werden.
- Kommentare, markiert durch geschweifte Klammern, sollen überlesen werden.
- Der Operator `mod` soll als weitere Strichoperator erkannt werden.

Welche Auswirkung haben diese Punkte für die Grammatik der Mini-Sprache?

ARBEITSBLATT ZUM PARSER-MODUL

Du erinnerst dich noch an die Grammatik unserer Minisprache:

```
T   = {  prog, start, ende, punkt, semikolon, name, zahl, punktoperator, strichoperator,
         zuweisungsop, klammerauf,, klammerzu }
N   = {  PROGRAMM, BLOCK, ZUWEISUNG, AUSDRUCK, TERM, FAKTOR }
P   = {  PROGRAMM  ::= prog name semikolon BLOCK punkt
         BLOCK     ::= start ZUWEISUNG ende
         ZUWEISUNG ::= name zuweisungsop AUSDRUCK ,
         AUSDRUCK  ::= TERM | TERM strichoperator AUSDRUCK ,
         TERM      ::= FAKTOR | FAKTOR punktoperator TERM ,
         FAKTOR    ::= name | zahl | klammerauf AUSDRUCK klammerzu ,
       }
S   =    PROGRAMM
```

Aufgabe 12: Erstelle die Parserprozedur für die Produktionsregel BLOCK und ZUWEISUNG. Überprüfe anschließend dein Ergebnis am Compiler-Projekt.

Aufgabe 13: Das Token *semikolon* hinter dem Programmnamen (Siehe Startregel *S*) wird in Pascal auch zur Trennung zweier Befehle verwendet. Erweitere zunächst die Grammatik der Mini-Sprache um die beiden nichtterminalen Komponenten:

ANWFOLG sowie ANWEISUNG!

Die ZUWEISUNG ist damit nur noch eine spezielle ANWEISUNG!
Erweitere entsprechend die Unit Parser.pas.

Aufgabe 14: Wir erweitern die Sprache um eine Blockanweisung, welche mehrere Anweisungen in einen Block zusammenfasst (DELPHI: `begin ... end`).
Die Produktionsregel für die ANWEISUNG lautet dementsprechend:

ANWEISUNG::= ZUWEISUNG | BLOCK ,

Erweitere die Unit Parser.Pas insofern, dass auch diese Blockanweisungen akzeptiert werden. (In der Datei U_Types.Pas ist nichts zu tun, da keine neuen Symbole hinzugekommen sind! Auch der Scanner-Automat hat sich nicht verändert.)

Im folgenden wollen wir unsere Minsprache um ein wenn-dann-sonst-Konstrukt erweitern. Dazu muss unser Compiler jedoch auch boolsche Ausdrücke für die jeweiligen Bedingungen erkennen. Der Einfachheit halber beschränken wir uns auf boolsche Ausdrücke der Form

AUSDRUCK vergleichsop AUSDRUCK

mit den Vergleichsoperatoren '<', '<=', '>', '>=', '=' und '<>'. Der feste Teil der Symboltabelle erweitert sich somit um diese sechs Vergleichsoperatoren als Token, welche allesamt den Tokennamen "vergleichsop" erhalten.

Aufgabe 15: Überarbeite den Scannerautomat (Datei Scanner.Pas) und die Symboltabelle (Datei U_Types.pas) so, dass der Scanner alle Vergleichsoperatoren der oben genannten Art sowie die Schlüsselworte "`if`", "`then`" und "`else`" erkennt.

Die Parseränderungen spiegeln sich in der folgenden Grammatik wieder:

```
        ANWEISUNG::= ZUWEISUNG | BLOCK | BEDANW ,
        BEDANW     ::= wenn VERGLEICH dann ANWEISUNG ( sonst ANWEISUNG ) ,
und     VERGLEICH  ::= AUSDRUCK vergleichsop AUSDRUCK
```

Aufgabe 16: Überarbeite den Parser entsprechend.

ARBEITSBLATT ZUM ÜBERSETZER-MODUL

Aufgabe 17: Öffne das Projekt Compiler inklusive des Übersetzers (Unit Coder.Pas) und schaue dir die Übersetzung der Produktionsregel ZUWEISUNG genauer an. Schaue dir auch an, was in der Prozedur Werte_aus vorgeht. Erweitere anschließend den Übersetzer (Prozedur `werte_aus`) so, dass der bereits im Scanner berücksichtigte Operator `mod` auch in ALI übersetzt werden kann. Hinweis: a mod b = a - (a div b) * b.

Aufgabe 18: Erweitere die Grammatik und damit die vom Compiler akzeptierte Programmiersprache um die beiden folgenden Befehle (Produktionen):

LIESANW ::= lies klammerauf name klammerzu
SCHREIBANW::= schreib klammerauf AUSDRUCK klammerzu

Diese Erweiterung hat Auswirkungen auf die Units:

U_Types.pas: Der feste Teil der Symboltabelle wird um die Token *lies* (Zeichenkette `readln`) und *schreib* (Zeichenkette `writeln`) erweitert.

Scanner.pas: Keine Auswirkungen, da der Scanautomat dank der Symboltabelle unverändert bleibt.

Parser.pas : Die Produktionsregel ANWEISUNG wird ergänzt, die obigen Produktionsregeln kommen hinzu.

Coder.pas: Realisation der Produktionsregeln ähnlich der Produktionsregel ZUWEISUNG.

Aufgabe 19: Erweitere den Compiler um die Möglichkeit einer while-Anweisung. Diese Änderung hat Auswirkungen auf:

U_Types.pas: Der feste Teil der Symboltabelle wird um die Token *solange* (Zeichenkette `while`) und *tue* (Zeichenkette `do`) erweitert.

Scanner.pas: Keine Auswirkungen, da der Scanautomat dank der Symboltabelle unverändert bleibt.

Parser.pas : Die Produktionsregel ANWEISUNG wird ergänzt, die Produktionsregel
SOLANGEANW::= solange VERGLEICH tue ANWEISUNG
kommt hinzu.

Coder.pas: Realisation der Produktionsregel ähnlich wie if-then-else.

Aufgabe 20: Erweitere den Compiler dahingehend, dass nicht nur einfache Vergleiche der Art AUSDRUCK vergleichsop AUSDRUCK möglich sind, sondern auch Bedingungsausdrücke welche sich durch die boolsche Verknüpfung von einzelnen Vergleichen durch AND und OR ergeben. Eine Grammatikerweiterung wäre z. B. wie folgt möglich (beachte auch die Auswirkungen auf Produktionen, die von VERGLEICH Gebrauch machen):

BEDAUSDR ::= BEDTERM (oder BEDTERM)* ** heißt beliebig oft*
BEDTERM ::= VERGLEICH (und VERGLEICH)*
VERGLEICH ::= AUSDRUCK vergleichsop AUSDRUCK