Vorbetrachtung

Die Aufgabe eines *Compilers* besteht aus der Übersetzung einer höheren Programmiersprache in eine maschinenverständliche Sprache, sowie aus der Organisation des benötigten Speicherplatzes. In Pascal bzw. Delphi erfolgt die Übersetzung direkt in Maschinensprache. Grob lässt sich der Übersetzungsvorgang in drei Schritte gliedern:

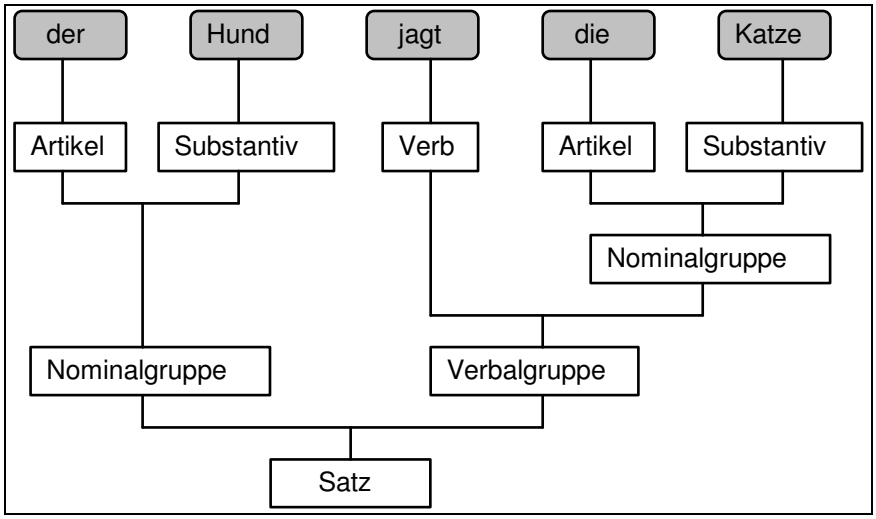
- der Quelltext wird vom ***Scanner*** gelesen,
- der Scanner-Output wird vom ***Parser*** auf syntaktische Korrektheit hin überprüft,
- sodass schließlich der ***Übersetzer*** die Übersetzung übernehmen kann;

zur Erfüllung dieser Aufgaben wird eine sogenannte Symboltabelle (***allocation table***) für die Speicher- und Adressorganisation angelegt.

Wir sehen, der Begriff *Compiler* ist fest mit dem Begriff der *Sprache* verknüpft! Da es sich bei Programmiersprachen um künstliche und nicht um natürliche Sprachen handelt, spricht man in diesem Zusammenhang von formalisierten oder auch *Formalen Sprachen*. Um uns den strukturellen Aufbau einer Sprache vor Augen zu führen, orientieren wir uns zunächst an einem Beispiel aus unserer natürlichen Sprache.

Beispiel: Sätze der deutschen Sprache wie z.B. *'der Hund jagt die Katze'* befolgen folgende strukturelle Regeln:

Dieser Typ von Sätzen lässt sich mittels der genannten Oberbegriffe analysieren. Die konkrete Ersetzung z.B. des Oberbegriffs 'Substantiv' durch *Hund* in der ersten Nominalgruppe ist nicht zwingend. Es ließe sich irgendein konkretes Substantiv unserer Sprache einsetzen. Wie dann allerdings der Sinn, die *Semantik*, des Satzes aussieht, ist eine andere Frage.



Dieses Beispiel zeigt schon deutlich, worauf es uns in den folgenden Ausführungen ankommt:
Eine Sprachenstruktur wird durch Gruppenbegriffsbildung zergliedert, die einzelnen Gruppen u.U. weiter 'verfeinert', bis schließlich durch konkrete Ersetzungen aus einer Menge möglicher Alternativen, dem konkreten Vokabular der Sprache, ein Satz der beschriebenen Sprachstruktur vorliegt.

So bezeichnet man allgemein die Menge der konkreten Ersetzungen, das Vokabular, als ***Terminales Alphabet T***, die Begriffsgruppen bzw. die Strukturklassen als ***Nicht-terminales Alphabet N*** und die Regeln für den strukturellen Zusammenhang beider Alphabete (*Syntax*) als ***Produktionen P***.

Bezogen auf deutschsprachige Sätze, wie im vorigen Beispiel gezeigt, könnte für das Terminalalphabet T gelten:

T = { der, die, das, Hund, Katze, Lehrer, Pferd, jagt, beißt, sieht, klaut }

und weiterhin

N = { Artikel, Substantiv, Verb, Nominalgruppe, Verbalgruppe, Satz }

mit den Produktionsregeln P, formuliert in der sogenannten **BNF (Backus-Naur-Form)**,

Satz	::=	Nominalgruppe Verbalgruppe	
Nominalgruppe	::=	Artikel Substantiv	
Verbalgruppe	::=	Verb Nominalgruppe	
Verb	::=	jagt | beißt | sieht | klaut	{ „|“ : „oder“! }
Artikel	::=	der | die | das	
Substantiv	::=	Hund | Katze | Lehrer | Pferd	

In diesem Sinne fassen wir zusammen und definieren:

> Eine ***Grammatik G*** ist ein Quadrupel **(T,N,P,S)**. Die Menge aller Worte aus Elementen von T, die aus dem Startsymbol S durch wiederholte Anwendung der Regeln aus P gebildet werden können, heißt ***Formale Sprache L(G)***.

Bemerkung:
Dem Begriff '*Worte'* in dieser Definition entsprechen in unserem konkreten Beispiel die '*Sätze'*. Die allgemeine Bezeichnung '*Worte'* kann man sich dadurch erklären, dass '*T'* und '*N'* als *Alphabete* in der Informatik bezeichnet werden, deren Elemente üblicherweise als '*Buchstaben'*. In dieser Hinsicht ist der theoretische Begriff *'Wort'* von seinem natürlich-sprachlichen Pendant zu unterscheiden. Der theoretische Begriff ist weit allgemeiner und umfassender.
Weiterhin muss die Bedeutung des '*Startsymbols S'* kurz erläutert werden:
Bei dem Prozess der Erzeugung von Worten einer vorliegenden Sprache L(G) liegt vor der Erzeugung natürlich noch kein Wort vor; man sagt daher: den Beginn einer Erzeugung stellt das sogenannte '*leere Wort S'* dar. In diesem formalen Sinn müsste für unser obiges Beispiel

S = Satz

gesetzt werden. In diesem Sinne stellt das Startsymbol die höchste Hierarchiestufe der Struktur dar.
Die Menge aller Produktionsregeln einer Grammatik beschreiben die '*Syntax*' der betrachteten Sprache.

Aufgabe 1: Analysiere die folgenden Produktionen ... ,
woraus besteht T, woraus N und was ist das „Startsymbol“?
Erzeuge dir ein paar gültige „Worte“ ... ,
worin besteht die Leistungsfähigkeit dieses Regelsystems?
Erweitere die Grammatik um die Klammerrechnung.

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

...versuche es erst selbst, dann kannst du weiterlesen!

Ich denke, du hast es herausbekommen ...

T = {name, zuweisungsoperator, strichoperator, punktoperator, zahl, klammerauf, klammerzu}
N = {ZUWEISUNG, AUSDRUCK, TERM, FAKTOR}
S = ZUWEISUNG

Das Terminalalphabet T müsste dir eigentlich zu denken geben! Dies ist nämlich wohl kaum ein Vokabular, wie du es dir i.a. vorstellen wirst - zumindest die Terminalen „name“ und „zahl“. Dennoch erscheint es sinnvoll, denn „name“ und „zahl“ sind im konkreten Fall endgültige Ersetzungen im Gegensatz zu den Nichtterminalen. Ähnliches gilt auch z.B. für „zuweisungsoperator“. Man bezeichnet diese Terminalen als sogenannte „***Tokens***“. Unter einem **Token** versteht man in Programmiersprachen den Namen, das Symbol für die entsprechende Zeichenkette. Ein Token ist somit die Codierung einer Zeichenkette. Daher ist zusätzlich zu den Produktionen P auch anzugeben, aus welchen Zeichen (ein Quelltext ist nichts anderes als eine Folge von Zeichen!) die Tokens bestehen bzw. bestehen dürfen. Eine der Aufgaben eines Scanners ist es gerade diese Tokens zu erkennen. Doch dazu später mehr.

Hast du auch die Leistungsfähigkeit des Systems erkannt? Verfolge dazu nochmals deine Beispiele wie auch das folgende, aber nicht in der „erzeugenden“ sondern dazu besser umgekehrt in „erkennender“ Richtung, wie es ein Compiler mit dem Quelltext zu machen hat. Mache dir dabei auch klar, dass der Parser grundsätzlich „ein Token voraus“ sein muss ...?! Warum wohl?

```
i  =  33 + n * 2;
```

			Der Scanner liefert das Token	*name*
ZUWEISUNG	:	**i**	ok und der Scanner liefert ...	*zuweisungsoperator*
		=	ok und der Scanner liefert ...	*zahl*
	und „Aufruf“ von ...			
AUSDRUCK	...			
TERM	...			
FAKTOR	:	**33**	ok und ...	*strichoperator*
	und zurück zu TERM,			
	da kein punktoperator vorliegt, zurück zu ...			
AUSDRUCK	:	**+**	ok und der Scanner ...	*name*
	und Rekursionsaufruf von ...			
AUSDRUCK2	und Aufruf von ...			
TERM		...		
FAKTOR	:	**n**	ok und ...	*punktoperator*
	und zurück zu TERM,			
TERM	:	*****	ok und ...	*zahl*
	da ein punktoperator vorlag, Rekursionsaufruf von ...			
TERM2	und Aufruf von ...			
FAKTOR	:	**2**	ok und ...	*semikolon*
	und zurück zu TERM2, wie auch zu TERM,			
	und zurück zu AUSDRUCK2, wie auch zu AUSDRUCK			
	und zurück zu ZUWEISUNG			
ZUWEISUNG	:	**;**	ok und der Scanner liefert...	*"leer"*

... und fertig! Das „Wort“ gehört zur beschriebenen Sprache ... und dir wird vielleicht aufgefallen sein, dass die Struktur der Produktionen P die wohlbekannte „Punkt vor Strich-Regel“ berücksichtigt! Da ein Parser erst auf ein Token reagieren kann, wenn es ihm vorliegt, muss nach dem Test des aktuellen Tokens schon das nächste „geholt“ werden - man sagt: der Parser muss ein Token voraus sein!

Aufgabe 2: Erstelle eine ähnliche Tabelle für das Beispiel: `x = (3 – y) * 4 * z + 1;`
Wozu dient das Token „leer“?

Bei der Übersetzung eines Programmtextes durch einen Compiler sind, wie Du gesehen hast, mehrere Aufgaben zu erfüllen: einerseits ist eine lexikographische Analyse des Textes vorzunehmen, d.h. Wort- und Symbolanalyse (*Scanner*), andererseits sind die vom *Scanner* gelieferten Informationen auf ihre syntaktische Korrektheit gemäß des Regelsystems der Sprache hin zu überprüfen (*Parser*). Erst anschließend kann die eigentliche Übersetzung in Maschinensprache erfolgen (*Übersetzer*).

Zwei unterschiedliche Vorgehensweisen des Scanners sind prinzipiell denkbar:

- Der Scanner liest immer nur das nächste Symbol und liefert das Token dem Parser zur Analyse.
- Der Scanner bearbeitet den gesamten Quelltext und baut dabei eine Folge von Tokens auf, die dem Parser als Grundlage für seine weitere Analyse dient.

Im ersten Fall arbeiten Scanner und Parser also „Hand in Hand“, im zweiten Fall erzeugt der Scanner eine vorläufige „Codierung“ des gesamten Quelltextes in eine Art „Zwischencode“, der z.B. temporär in einer linearen Liste festgehalten werden kann. Allerdings genügt es nicht, allein die Tokens zu liefern - wie im Beispiel der letzten Seite aufgeführt. Denn für eine Übersetzung ist es unerlässlich, außerdem die Information darüber festzuhalten, welcher konkrete Name (Variable) bzw. welche konkrete Zahl (Konstante) im Quelltext aufgeführt ist. Es ist ein Protokoll über die selbstdefinierten Programmobjekte zu führen! Dies geschieht in der schon erwähnten Symboltabelle, wo außer diesem Protokoll auch die Speicherverwaltung bzw. Speicherzuordnung dieser Programmobjekte organisiert wird. Diese selbstdefinierten Worte sind von den reservierten Worten der Sprache zu unterscheiden - in Pascal gehören zu den letzteren z.B. Worte wie BEGIN, WHILE, IF, THEN, ELSE etc.

Für unser Beispiel der Zuweisungen algebraischer Ausdrücke könnte eine solche *Symbol- und Tokentabelle* der terminalen Objekte wie folgt aussehen:

Zu Anfang der Tabelle werden die „festen“ Terminalsymbole unserer Beispielsprache aufgeführt. Nachfolgend erscheinen dann die benutzerdefinierten Terminalen in der Reihenfolge entsprechend ihres ersten Auftretens im Quelltext.

Index	String	Token	Adresse	...
1	=	zuweisop	–	
2	+	strichop	–	
3	–	strichop	–	
4	*	punktop	–	
5	/	punktop	–	
6	(	kla.auf	–	
7	)	kla.zu	–	
8	;	semicolon	–	
9	i	name	xxx1	
10	33	zahl	xxx2	
11	n	name	xxx3	
12	2	zahl	xxx4	

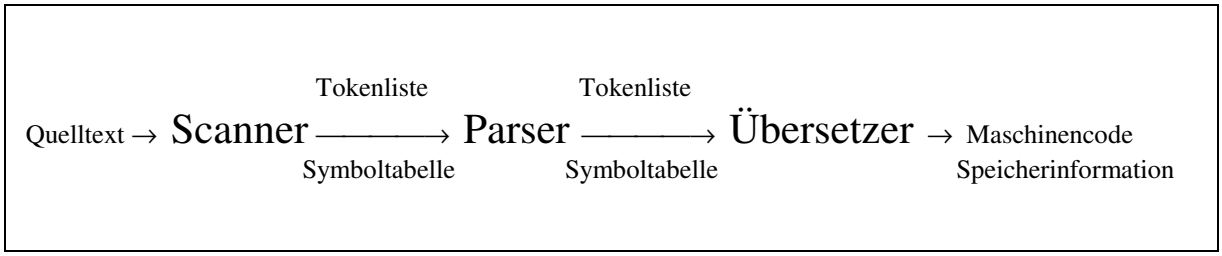
Als Zwischencode des obigen Zuweisungsbeispiels genügt damit eine lineare Liste, die allein aus den Index-Nummern der zugehörigen Tokens in der Tabelle besteht. Diese Liste wird als ***Tokenliste*** des Quelltextes bezeichnet. Die Tokens selbst brauchen wegen der Eindeutigkeit der Indizes nicht mit aufgeführt zu werden.

`i = 33 + n * 2;` würde codiert als: 9, 1, 10, 2, 11, 4, 12, 8 .

Beachte: Der untere Abschnitt der benutzerdefinierten Symbole enthält neben Index, Zeichenkette und Token noch weitere Informationen wie z.B. die zugehörige Speicheradresse, die diesen Variablen bzw. Zahlen zuzuordnen sind. Denken wir schon weiter, nämlich dass auch Labels oder Prozedurnamen zu protokollieren sind, so sind auch für diese Programmobjekte Einsprungadressen festzuhalten ...

Aufgabe 3: Erstelle für deine Zuweisungsbeispiele wie auch für das Beispiel von *Aufgabe 2* eine Symboltabelle und gib den „Zwischencode“ als Tokenliste an.

Die von einem Compiler zu leistenden Aufgaben fassen wir im folgenden Diagramm zusammen:



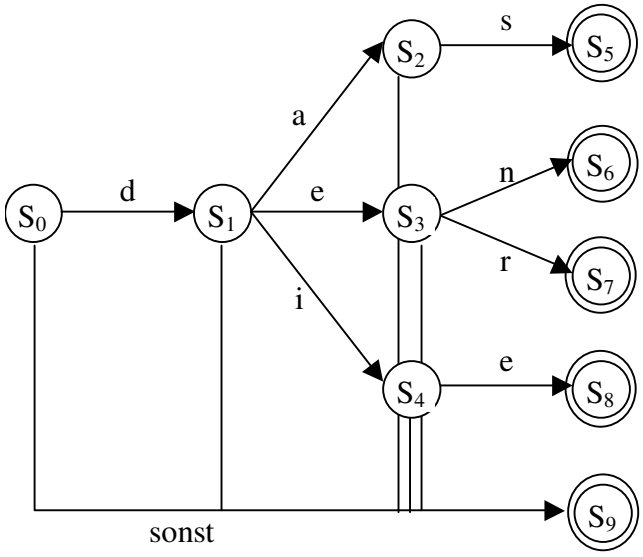
Wir werden uns nun mit den einzelnen Phasen dieser Quelltextumsetzung genauer befassen; d.h. wir werden die einzelnen Arbeitsschritte eines Compilers unter algorithmischen Gesichtspunkten näher durchleuchten. Damit werden wir in der Lage sein, letztendlich einen eigenen Compiler einer von uns definierten Sprache schreiben zu können.

[Eine interessante Anwendung dieser Thematik wäre auch die Entwicklung eines Rollenspiels oder Adventures oder einer Sprache zur Konstruktion geometrischer Objekte - dir aus früheren Zeiten sicherlich unter der Bezeichnung „Konstruktionsbeschreibung“ bekannt ... oder vieles mehr!]

Der Scanner als *endlicher Automat*

Befassen wir uns also zunächst mit der Arbeitsweise eines Scanners. Soll ein Text lexikographisch analysiert werden (*gescanned* werden), so lassen sich die notwendigen Schritte anschaulich durch einen sogenannten „*erkennenden Automaten*“ darstellen. Wir betrachten dazu ein Beispiel:

Es sollen die Artikel 'der', 'die', 'das' und 'den' erkannt werden. Ein zugehöriger „*Automat*“ erhält als Eingaben einzelne Zeichen und soll als Ausgabe entsprechende Meldungen machen. Graphisch lässt sich der Vorgang wie folgt veranschaulichen:



Ausgehend von dem Startzustand 'S_0' wird bei gelesenem Zeichen 'd' in den Zustand 'S_1' übergegangen, von da z.B. bei gelesenem Zeichen 'e' in den Folgezustand 'S_3' usw. . Die Zustände 'S_5', 'S_6', 'S_7' und , 'S_8' sind sogenannte *Endzustände*, die zu einem gültigen Abbruch der 'Artikelerkennung' führen. Andere Eingaben müssten einen Abbruch des Erkennungsvorgangs (Zustand , 'S_9') mit entsprechender Fehlermeldung bewirken.

Ein Automat ohne diesen Fehlerzustand wird als „partiell“ definiert bezeichnet. Bei Hinzunahme eines Fehlerzustandes mit entsprechenden Übergängen spricht man von einem „global“ definierten

Automaten. Da aber in der Regel die Graphen von Automaten recht komplex aufgebaut sind, verzichtet man der Übersichtlichkeit halber häufig auf die explizite Aufnahme des Fehlerzustandes in das Diagramm.

Wir definieren:

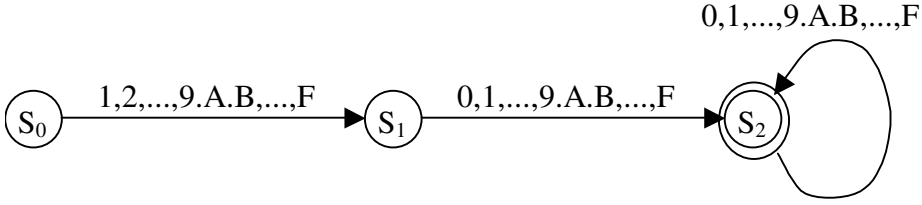
> Ein ***(endlicher) Automat A*** ist charakterisiert durch eine endliche Menge von Zuständen 'S', einem Anfangszustand 's' und einer nicht leeren Menge 'F' von Endzuständen, wobei die Zustandsübergänge 'R' eingabeabhängig sind - gemäß vorliegendem Alphabet 'Σ'. Die Zustandsübergänge lassen sich daher als Abbildung R: S x Σ → S (lies: „S kreuz Sigma“) beschreiben, der Automat selbst also kurz durch das Fünf-Tupel:
>
> $$\boldsymbol{A = (S, \Sigma, s, F, R)}\ .$$
>
> In seinen Endzuständen kann (muss nicht) eine Ausgabe erfolgen.

Entsprechend dieser Definition gilt für unser „Artikel-Beispiel“:

$S = \{ S_0, S_1, \ldots, S_9 \}$,
$\Sigma = \{ a, d, e, i, n, r, s \}$,
$s = S_0$,
$F = \{ S_5, S_6, S_7, S_8, S_9 \}$,
$R = \{ \quad S_0 \times d \rightarrow S_1,$
$\quad S_1 \times a \rightarrow S_2,\ S_1 \times e \rightarrow S_3,\ S_0 \times i \rightarrow S_4,$
$\quad S_2 \times s \rightarrow S_5,$
$\quad S_3 \times n \rightarrow S_6,\ S_3 \times r \rightarrow S_7$
$\quad S_4 \times e \rightarrow S_8,$
$\quad$ ... <und noch die Fehlersituationen> $\quad \}$.

Aufgabe 4: Dir ist sicherlich ein Zigarettenautomat bekannt. Nach dem Einwurf von 3 € kann ein Wahlknopf gedrückt werden. Zulässige Münzen sind 50 Cent-, 1 €- und 2 €-Münzen. Erstelle einen realisierenden endlichen Automaten und definiere S, Σ, s, F und R.

Aufgabe 5: Was leistet der folgende graphisch dargestellte Automat?



Wenden wir uns nun aber dem konkreten Beispiel einer „Mini-Sprache“ zu: im folgenden Abschnitt wird eine zunächst recht einfache Sprache definiert, anhand der die Scanner- und Parser-Algorithmen behandelt werden. Neben dem Verständnis dieser Algorithmen wird es insbesondere darauf ankommen, zu erkennen, wie einfach die Erweiterung dieser Sprache zu einem durchaus schon recht komplexen Werkzeug sein kann!

Eine „Mini-Programmiersprache“

Die Grammatik (T,N,P,S) einer fiktiven Programmiersprache könnte wie folgt aussehen:

```
T  = { typ, blockauf, blockzu, semikolon, name, zahl, strichoperator, punktoperator, zuweisungsop, klammerauf,
       klammerzu }

N  = { METHODE, BLOCK, ZUWEISUNG, AUSDRUCK, TERM, FAKTOR }

P  = {  METHODE      ::= typ name klammerauf klammerzu BLOCK,
        BLOCK        ::= blockauf ZUWEISUNG blockzu,
        ZUWEISUNG    ::= name zuweisungsop AUSDRUCK semikolon,
        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'  = { typ             ::= v o i d | i n t ,
        blockauf        ::= { ,
        blockzu         ::= } ,
        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       ::= )
      }
```

Ein gültiges Mini-Programm, ein „Wort“ dieser Sprache, wäre z.B.:

```
void main() {
  otto = (x - ( 4 * 11 - (otto / 13) ) ) ;
}
```

Aufgabe 7: Wieso ist das folgende Programm fehlerhaft? **Tipp:** es sind drei verschiedene Fehlertypen!

```
void noch_Mist() {
  otto = 4 * otto
}
```

... noch nicht weiterlesen, erst selber denken!

Zwei der Fehler waren wohl nicht schwer zu erkennen:

- das „Underline“ (_) ist in der definierten Sprache noch nicht zugelassen, also erweitern ...
- wie auch die Verwendung von Großbuchstaben, ebenso erweitern!
- Schwieriger ist die Feststellung des dritten Fehlers. Wir spielen die Analyse der Zuweisung innerhalb des Blocks einmal durch:

```
otto := 4 * otto
```

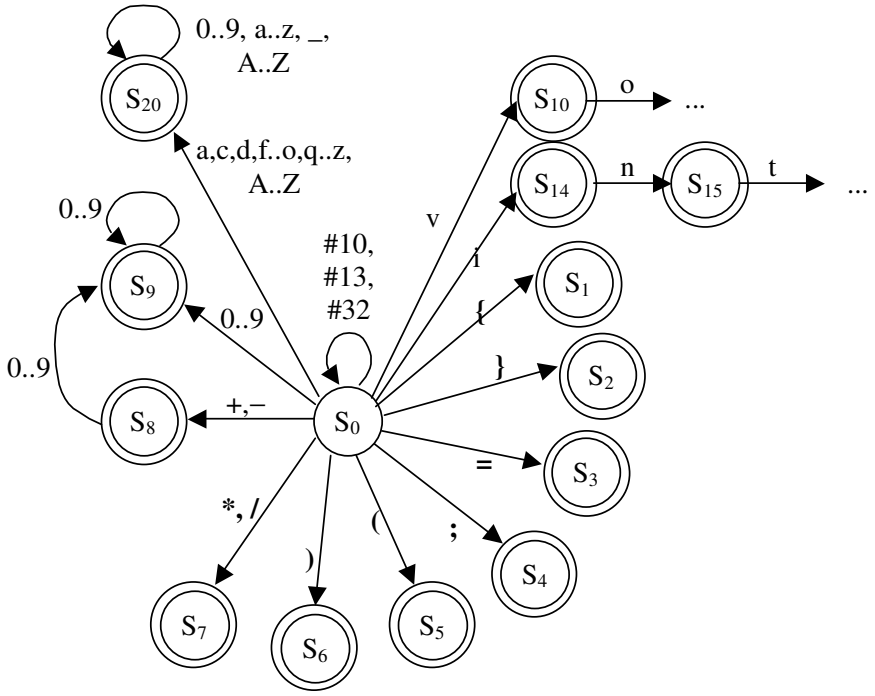
... in BLOCK	nach *blockauf* liegt folgendes Token vor			*name*
ZUWEISUNG:		**otto**	ok, nun das nächste Token	*zuweisungsop*
		=	ok, nun ...	*zahl*
	und Aufruf von ...			
AUSDRUCK1:	und Aufruf von ...			
TERM1:	und Aufruf von ...			
FAKTOR1:		**4**	ok, nun das nächste Token	*punktoperator*
	und zurück zu TERM1			
TERM1:		*****	ok, ein punktoperator, nun	*name*
	und daher Rekusionsaufruf von ...			
TERM2:	und Aufruf von ...			
FAKTOR2:		**otto**	ok, nun das nächste Token	*blockzu*
	und zurück zu TERM2			
TERM2:	kein punktoperator, also zurück zu...			
TERM1:	kein punktoperator, also zurück zu...			
AUSDRUCK1:	kein strichoperator, d. h. nichts mehr zu tun, also zurück zu...			
ZUWEISUNG:	**}**	**Fehler**: ***semikolon*** erwartet!!!		

Dies war schon recht aufwendig, oder? Aber wir haben es herausgefunden: der dritte Fehler lag in dem fehlenden Semikolon.

Bevor wir uns nun mit sinnvollen Erweiterungen der Mini-Sprache befassen, sollten wir uns erst einmal auf eine algorithmische und damit programmierbare Lösung konzentrieren. Auf Dauer finde ich - und du sicherlich auch - die manuelle Analyse reichlich aufwendig und unangenehm. Also ‘ran an die Algorithmen ...

Der Scanner unserer Mini-Sprache

... aber halt! Für eine algorithmische Beschreibung des Scan-Vorgangs ist es wichtig, uns als erstes den zugehörigen ***endlichen (erkennenden) Automaten*** klarzumachen.
Betrachte dazu das folgende Diagramm, wobei wir uns aus Platzgründen auf eine Teil-Graphik beschränken, d.h. auf einen Ausschnitt der möglichen Zustände:



Aufgabe 9: Zeichne den partiellen Automaten vollständig, d.h. mit allen seinen Zuständen, auf ein DIN-A4-Blatt.

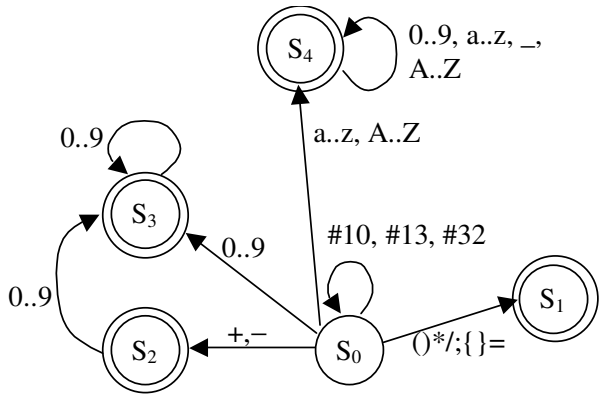
Für den partiellen Automat A = (S, Σ, s, F, R) gilt:

S = { S_0, ... , S_{20} }
Σ = { '0'..'9', 'a'..'z', 'A'..'Z', '_', '{','}', '=', '–', '+', '*', '/', '(', ')', #10, #13, #32 }
s = S_0
F = { S_1, S_3, ... , S_{20} } = S \ { S_0 }
R = ... s.o. ...

Fällt dir eigentlich auf, was an diesem Automaten, der zunächst durchaus als naheliegend erscheint, unpraktikabel ist? Was wäre, wenn du den Sprachumfang um weitere Tokens, wie z.B. if, else oder while oder switch, case, etc., erweitern wolltest? ...

Richtig, die Zustandszahl „explodiert"! Wir suchen daher nach einem Automaten, der „cleverer" ist, der nur eine „Minimalzahl" an notwendigen Zuständen besitzt. Aber wie können wir die Zustandszahl gering halten? Eine Minimierung kann oftmals nur durch zusätzliche Informationen erreicht werden! Denke dazu doch einmal an das erste Kapitel zurück, wo wir die sogenannte „*Symbol- und Tokentabelle*" am Beispiel entwickelt haben! Wenn wir also eine Tabelle der reservierten Wörter unserer Sprache hätten, so kann jedes reservierte Wort, das aus „normalen" ASCII-Zeichen besteht (Sonderzeichen sind hier auszuschließen), zunächst vom Scanner genauso behandelt werden, als wäre es ein „name". Dazu hat der Scanner natürlich die sequentiell gelesenen Zeichen zu einer Zeichenkette aufzubauen. Vor Zuordnung des Tokens müsste nun allerdings in der Tabelle nachgeschaut werden, ob diese Zeichenkette in der Tabelle schon aufgeführt ist: wenn ja, kann die dortige Information genommen werden, wenn nein, ist die Tabelle entsprechend zu erweitern. Auf jeden Fall ist gewährleistet, dass das passende Token der Zeichenkette zugeordnet werden kann.

Betrachte dazu den folgenden, nun vollständigen und globalen Automaten, der für unsere bisherige Mini-Sprache mit nur sechs Zuständen einschließlich eines Fehlerzustandes auskommt. Auch bei Spracherweiterungen wird sich die Zustandsanzahl, wenn überhaupt, nur unwesentlich vergrößern. Beachte weiterhin, dass die Zuordnung der Endzustände oftmals genau einen Scan-Schritt früher erfolgt als in dem letzten Automaten. Erreicht werden kann dies, da der variierte Scanner einen logisch zusammenhängenden String aufbaut, unabhängig davon, ob ein reserviertes Wort, ein kombiniertes Symbol (z.B. **:=**), eine Zahl oder ein Name vorliegt. Für den Anfangszustand S_0 z.B. bedeutet dies, dass S_0 für die Zeichen '(', ')', '{', '}', '=', ';', '*' und '/' schon ein Endzustand ist. Erst nach dem Scan-Prozess wird in Abhängigkeit der erzeugten Zeichenkette das zugehörige Token aus der Symbol- und Tokentabelle bestimmt.



Eine erste algorithmische Fassung eines Scanners könnte wie folgt lauten:

Scan	I: *Quelltext*	: String;
	I/O: ...? ...	: ...? ...
lokale Objekte:	...? ...	: ...? ...

- initialisiere ...
- Wiederhole folgendes

Scan_Symbol		
lokale Objekte:	*Endzustand*	: boolean;
	Zustand	: int;
	Zeichen	: char;
	Zeichenkette	: String; ...? ...

- Initialisiere *Endzustand* mit false, Zu*stand* mit S_0, *Zeichenkette* mit ''
- Wiederhole folgendes:
  - ***LiesZeichen***(*Quelltext*, *Zeichen*, ...)
  - falls *Zeichen* existiert,
    dann - falls *Zeichen* kein „Kontrollzeichen" ist,
    dann erweitere die aktuelle *Zeichenkette* um dieses *Zeichen*,
    - leite entsprechend dem eingelesenen *Zeichen* in den Folgezustand über, merke dir u.U. das aktuelle *Zeichen* für die folgende Analyse und setze u. U. *Endzustand* oder halte einen aufgetretenen Lesefehler fest,

  bis kein weiteres *Zeichen* mehr für das aktuelle Symbol zu akzeptieren ist/werden kann.
  [d.h.: *Endzustand* erreicht oder *Fehler* oder es gab kein *Zeichen* mehr]
- Falls *Fehler*, dann Fall: Token „***nicht_zulaessig***",
  sonst falls *Zeichenkette* leer, dann ist das Ende des *Quelltextes* erreicht,
  sonst ordne über die Symboltabelle der *Zeichenkette* das passende Token zu.

bis das Ende des Quelltextes erreicht wurde.

Offengeblieben sind im Scanner-Algorithmus ...
a) ... die genaue Parameterliste von *Scan*,
b) ... die Beschreibung aller benötigten lokalen Variablen von *Scan*,
c) ... wie dort die Übergänge in die Folgezustände realisiert werden,
d) ... wie die Symboltabelle aufgebaut ist, einschließlich der konkreten Tokenzuordnung,
e) ... wie die Tokenliste realisiert werden kann,
f) ... wie die Hilfsroutine „*LiesZeichen*“ arbeitet.

zu a) Soll der Scanner nicht „Hand-in-Hand“ mit dem Parser arbeiten, sondern einen vollständigen „Zwischencode“, eine Tokenliste, erzeugen, sowie eine Symboltabelle aufbauen bzw. ergänzen, auf deren Basis der Parser seine Analyse anschließen kann, so sind diese zwei Objekte von der Routine *Scan* aufzubauen: die Tokenliste und die Symboltabelle. Diese sind als Datenfelder im Scanner global zu definieren, da beide Objekte die zentrale Rolle im Hintergrund darstellen... Weiterhin ist es sinnvoll, eine Meldung nach außen zu geben, ob der Scan-Prozess erfolgreich verlaufen ist oder nicht. Ein Misserfolg kann verschiedene Gründe haben: der Scanner findet ein unzulässiges Zeichen oder die Symboltabelle war nicht genügend groß ausgelegt oder ... Es wird davon ausgegangen, dass die Textdatei existiert und geöffnet vorliegt! Der vollständige Funktionskopf ist:

```
public int scan (String quelltext)
```

zu b) Die lokalen Variablen von *Scan* ergeben sich aus der genauen Analyse der internen Arbeitsweise. Führen wir uns vor Augen, wie z.B. die Teilkette ‘39*...‘ gescant wird, so kann die Zahl 39 erst dann erkannt sein, wenn der Operator ‘*’ gelesen wurde. Der Operator ist aber eigentlich erst beim nächsten Aufruf von *Scan_Symbol* zu bearbeiten. Dann ist das Token *operator* zu liefern! Diese Schwierigkeit bekommen wir in den Griff, wenn wir eine eigene Klasse entwickeln, welche den Quelltext Zeichen für Zeichen zurückgibt, welche aber auch über eine Methode verfügt, ein Zeichen zu wiederholen, falls es zuviel gelesen wurde. Dies ist der Fall, wenn wir in einen Endzustand gelangen, nachdem wir z. B. '39*' gelesen haben. Siehe dazu auch die Klasse ZeichenLeser.

zu c) Hier bietet sich eine switch-Anweisung an:

```
switch (zustand) {
            case 0: if (zeichen == ' ' ||
                        zeichen == (char)9 ||
                        zeichen == '\n' ) {
                      zustand = 0;
                      zeichenkette = "";
                    } else if (zeichen == '(' || zeichen == ')' ||
                               zeichen == '{' || zeichen == '}' ||
                               zeichen == '*' || zeichen == '/' ||
                               zeichen == ';' || zeichen == '=') {
                      zustand = 1;
                    } else if (zeichen == '+' || zeichen == '-' ) {
                      zustand = 2;
                    } else if (zeichen >= '0' && zeichen <= '9') {
                      zustand = 3;
                    } else if ( (zeichen >= 'a' && zeichen <= 'z') ||
                                (zeichen >= 'A' && zeichen <= 'Z') ) {
                      zustand = 4;
                    } else {
                      fehlerNr = 1;
                    }
                    break;
            case 1: endzustand = true;s1:
            case 2: if (zeichen >= '0' && zeichen <= '9') {
                      zustand = 3;
```

```
                        } else {
                          endzustand = true;
                        }
                        break;
... usw. ... siehe dazu auch die Unit Scanner ...
} // end of case
```

zu d) Die Symboltabelle kann als Array [0..MAX] über den Datentyp Token (eigene Klasse) deklariert werden. In einem Token sollten ähnlich zu dem Beispiel von Seite 4 mindestens der Tokentyp und die zugehörige Zeichenkette abgelegt sein. Diese Tabelle ist mit allen reservierten Wörtern der Sprache, Sonderzeichen, Operatoren usw. zu initialisieren. Die Tokenzuordnung erfolgt mittels einer internen Methode: die Tabelle ist zu durchlaufen; gegebenenfalls wird die Zeichenkette gefunden oder die Tabelle wird entsprechend ergänzt, falls noch Platz für die Einträge vorhanden ist. Der passende Tokenindex wird zur Erweiterung der Tokenliste exportiert. (siehe dazu auch die Klasse Symboltabelle)

zu e) Die Tokenliste als reine Indexliste der Symboltabelle realisieren wir mittels einer linearen Liste, also dynamisch. Die bekannte Datenstruktur List erweist sich hier als recht nützlich (siehe dazu auch die Klassen List sowie Tokenliste). Denkbar wäre auch die Verwendung einer Schlange. Aus praktischen Gründen - gemeint sind Java-Outputs - sehen wir aber davon ab. Ebenfalls erweist es sich als praktisch ein spezielles Token zur End-Markierung dieser Liste zu definieren. Wir verwenden dazu das Token *leer*. Dieses Token ist grundsätzlich das letzte Token der Liste!

zu f) Die Aufgabe der Mehode *LiesZeichen* der Klasse ZeichenLeser sollte inzwischen klar sein. Die Methode gibt lediglich das aktuelle Zeichen des Quelltextes zurück und wandert mit der Position ein Zeichen weiter. Falls das Ende des *Quelltext* schon erreicht war, so sollte stattdessen ein definiertes leeres Zeichen, z.B. #255 - das üblicherweise nie benötigt wird, ausgegeben werden.

Aufgabe 10: Schaue Dir das Projekt im Ordner Compiler an

Klasse Compiler:	Zuammenfassung des Scanners, der Symboltabelle und der Tokenliste
Klasse Token:	Klasse für einen Eintrag der Symboltabelle
Klasse Tokenliste:	Definition der Tokenliste
Klasse Symboltabelle:	Definition der Symboltabelle
Klasse ZeichenLeser:	Klasse zum zeichenweise Lesen eines Quelltextes
Klasse Scanner:	Die eigentliche Scannerroutine (Realisation des endl. Automaten)

Ergänze den Scanner (Symboltabelle) nun um folgende Eigenschaften:

- auch deutsche Umlaute sollen in Namen akzeptiert werden
- Kommentare, markiert durch /* ... */ sollen überlesen werden
- der Operator % soll erkannt werden. (hier schauen wir schon etwas voraus...)

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

Der Parser unserer Mini-Sprache

Nachdem der Scanner den Quelltext in eine Tokenliste zwischencodiert und die Symboltabelle um die benutzerdefinierten Namen und Konstanten erweitert hat, steht einer Syntaxanalyse nun nichts mehr im Wege. Dazu schauen wir uns nochmals die Produktionsregeln unserer Mini-Sprache an.

```
P  = {  METHODE     ::= typ name klammerauf klammerzu BLOCK,
        BLOCK       ::= blockauf ZUWEISUNG blockzu ,
        ZUWEISUNG   ::= name zuweisungsop AUSDRUCK semikolon,
        AUSDRUCK    ::= TERM | TERM strichoperator AUSDRUCK ,
        TERM        ::= FAKTOR | FAKTOR punktoperator TERM ,
```

```
        FAKTOR          ::= name | zahl | klammerauf AUSDRUCK klammerzu ,
    }
```

Am Rande bemerkt: dir ist vielleicht aufgefallen, dass das Regelsystem P' vom Scanner bearbeitet wurde ... Aber nun zurück zum Parser. Seine Aufgabe besteht nun darin, die in P definierte Struktur aus Terminalen und Nichtterminalen Komponenten unter Berücksichtigung der Tokenliste und unter Verwendung der Symboltabelle abzuarbeiten. Eine algorithmische Fassung des Parsers könnte wie folgt lauten:

Parse	I: *tokenliste*	: Tokenliste
	I/O: *fehlerNr*	: int
lokale Objekte:	*aktuellesToken*	: Token
- initialisiere *fehlerNr* mit 0, *aktuellesToken* mit dem ersten Token der Liste - Falls *aktuellesToken* = *leer* dann Fall: leere Tokenliste und *FehlerNr* := 1 {unexpected end of file} sonst Aufruf der Startregel ***methode ()***		

... also recht einfach! Na ja, es fehlt natürlich noch eine ganze Menge: wir müssen die geschachtelte Struktur der Produktionsregeln „nachspielen". Diese Schachtelung von Regeln ist in ihrer Implementierung nichts anderes als eine entsprechende Schachtelung von Prozeduren. Das Prozedurgerüst sieht wie folgt aus:

```
public int parse() {}
  public int methode() {}
    public int block() {}
      public int zuweisung() {}
        public int ausdruck() {}
          public int term() {}
            public int faktor() {}
```

Du siehst, auch dies ist einfach! Interessant ist nun der Aufbau der verschiedenen Prozedurrümpfe. Das Prinzip dürfte dir klar sein: der Aufbau der jeweiligen Produktionsregel ist zu checken, also die dort auftretenden Tokens sind mit denen der Tokenliste zu vergleichen und für eine nichtterminale Komponente hat der entsprechende Prozeduraufruf zu erfolgen.

Wichtig im Aufbau der Prozedurrümpfe ist noch folgender Gesichtspunkt: damit der Parser die Tokenliste mit der Struktur seiner Grammatik vergleichen kann, muss selbstverständlicherweise auch schon ein Token für den Vergleich vorliegen. Wie der Scanner bei der Zwischencode-Erzeugung in der Regel ein Zeichen voraus war, so muss der Parser ebenfalls immer ein Token voraus sein.

Schaue dir einmal exemplarisch den Prozedurrumpf der Startregel S an:

```
  public int methode() {
    int fehlerNr = 0;
    if (aktuellesToken != Token.typ) {
      return 2;
    } else {
      aktuellesToken = tokenliste.naechstesToken();
      if (aktuellesToken != Token.name) {
        return 3;
      } else {
        aktuellesToken = tokenliste.naechstesToken();
        if (aktuellesToken != Token.klammerauf) {
          return 4;
        } else {
          aktuellesToken = tokenliste.naechstesToken();
          if (aktuellesToken != Token.klammerzu) {
            return 5;
```

```
          } else {
            aktuellesToken = tokenliste.naechstesToken();
            fehlerNr = block();
            if (fehlerNr != 0){
              return fehlerNr;
            } else {
              return 0;
            }
          }
        }
      }
    }
  }// end of methode, der Startregel
```

Als erstes wird nachgeprüft, ob das aktuelle Token *typ* ist. Falls nicht, dann liegt ein Fehler vor, sonst kann das nächste Token aus der Tokenliste gelesen werden. Falls dieses kein *name* ist, dann liegt ein neuer Fehler vor. Andernfalls kann das nächste Token aus der Liste genommen werden. Dann muss überprüft werden, ob es eine *klammerauf* ist. Ist dies auch in Ordnung und ist das nächste Token aus der Liste auch noch eine *klammerzu*, dann kann die Methode *block* aufgerufen werden. Kehrt man ohne Fehler aus dieser Prozedur zurück, dann ist dies durch den Rückgabewert 0 zu bestätigen, andernfalls trat ein Fehler im Block auf, der unverändert zurückgegeben wird.

Beachte: Jedes mal, wenn ein Token abgearbeitet ist, wird direkt schon das nächste Token geholt. Weiterhin: nach Rücklauf aus einer Prozedur wird die Syntaxkontrolle der aktuellen Produktion nur dann fortgesetzt, wenn bisher kein Fehler aufgetreten ist!

Aufgabe 11: Bevor du dir den Quelltext der Klasse Parser ansiehst, schreibe die beiden Methodenrümpfe der Methoden *block* und *zweisung* zunächst selbst.

Aufgabe 12: Schaue dir die neue Version des Compilers (inkl. Parser) an. Kontrolliere deine Lösung zu Aufgabe 11 anhand des Parser-Quelltextes. Teste das Programm aus.

Aufgabe 13: Innerhalb eines Blocks dürfen auch mehrere Anweisung nacheinander folgen. 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 Klasse Parser.

Aufgabe 14: Wir erweitern die Sprache um eine Blockanweisung, welche mehrere Anweisungen in einen Block zusammenfasst (Java: `{ ... }`) Die Produktionen der Grammatik sehen damit wie folgt aus (Lösung aus Aufgabe 13 mitberücksichtigt):

```
P      = {   METHODE   ::= typ name klammerauf klammerzu BLOCK,
             BLOCK     ::= start ANWFOLG ende ,
             ANWFOLG   ::= ANWEISUNG (ANWFOLG ) ,
             ANWEISUNG::= ZUWEISUNG | BLOCK ,
             ZUWEISUNG::= name zuweisungsop AUSDRUCK semikolon,
             AUSDRUCK  ::= TERM | TERM strichoperator AUSDRUCK ,
             TERM      ::= FAKTOR | FAKTOR punktoperator TERM ,
             FAKTOR    ::= name | zahl | klammerauf AUSDRUCK klammerzu ,
        }
```

Erweitere die Klasse Parser insofern, dass auch diese Blockanweisungen akzeptiert werden. (In der Token-Klasse und Symboltabelle-Klasse 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 eine Kontrollstruktur zur Fallunterscheidung sowie einer Schleifen-Struktur 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.
Um einen Vergleich zu erkennen ändern sich im erkennenden Automaten die Übergänge an dem Zustand S_0. Außerdem kommen zwei neue Zustände hinzu.

Weiterhin benötigen wir für die Fallunterscheidung die Schlüsselwörter "`if`" und "`else`".

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

Natürlich ist es mal wieder nicht nur mit dem Scannen getan. Der Parser soll nun auch die if-else-Anweisung erkennen. Dabei soll der else-Teil optional gehalten werden. diese Einschränkung spiegelt sich in der Erweiterung der Produktionen oben genannter Grammatik wie folgt wieder:

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

Aufgabe 16: Erweitere den Parser um die Erkennung einer Bedingungs-Anweisung (BEDANW), welche wie die ZUWEISUNG und BLOCK-Anweisung eine spezielle Unterart von ANWEISUNG ist:

Aufgabe 17: Zu guter letzt soll auch eine Wiederholung zugelassen sein (SOLANGEANW). Erweitere die Grammatik der Mini-Sprache um die nichtterminale Komponente SOLANGEANW, den festen Teil der Symboltabelle um das Token 'wiederhole' (mit Zeichenkette `while`) und implementiere die Erkennung einer Wiederhole-Schleife in der Parser-Klasse.

Der Übersetzer unserer Mini-Sprache

Der Scanner analysierte den Quelltext mit Hilfe der Symboltabelle. Diese wurde eventuell um Variablen und Zahlen erweitert. Resultat des Scanners war die Tokenliste, die das Programm codiert durch die Indizes der Symboltabelle in Kurzform wiedergibt. Prinzipiell ist eine Übersetzung während des Parsings möglich und aus lauzeittechnischen Überlegungen auch sinnvoll. Dennoch wollen wir den Übersetzungsprozess vom Parsing abkoppeln, um die Besonderheiten besser zu verstehen.
Der Codierer muss im Prinzip – genau wie der Parser – die Grammatik durchlaufen. Ebenfalls genau wie der Parser muss auch der Codierer stets ein Token voraus sein, um die Verzweigung in den Produktionsregeln korrekt vornehmen zu können. Anders als der Parser kann beim Codierer allerdings kein Syntaxfehler mehr festgestellt werden, da dies schon im Parseprozess aufgefallen wäre.
Die Aufgabe des Übersetzers besteht also darin, die in P definierte Struktur aus Terminalen und Nichtterminalen Komponenten nochmals unter Berücksichtigung der Tokenliste und unter Verwendung der Symboltabelle abzuarbeiten. Eine algorithmische Fassung des Übersetzers könnte wie folgt lauten:

Code	I:	*Tokenliste*	: TLinList	
lokale Objekte:		*AktTokenIndex*	: TSymbolIndex	
		OTK, ODK	: TStack	{ für Dreiadressformat }
		HCount, MCount	: integer	{ Hilfsvariablenzähler, Sprungmarkenzähler }
- initialisiere *HCount* und *MCount* mit 0, *OTK*, *ODK* als leere Stacks, *AktTokenIndex* mit dem ersten TokenIndex der Tokenliste - Aufruf der Startregel ***S (AktTokenIndex)***				

... auch recht simpel. Allerdings muss auch hier die Grammatik – oder besser gesagt die einzelnen Produktionsregeln – durch Prozeduren nachgebildet werden. Auch dies ist im Prinzip nichts anderes als beim Parser, wie die folgende Prozedur der Startregel S zeigt:

```
procedure TCoder.S (var AktuellerTokenIndex: TSymbolIndex);
var m: string;
    i: integer;
    v: string;
    LastTokenIndex: TSymbolIndex;
begin {of S, der Startregel}
  AktuellerTokenIndex:= Tokenliste.NextTokenIndex;
  m:= Symboltabelle.SymbolString(AktuellerTokenIndex);
  Aliprogramm.add(m,'START','0','');
  AktuellerTokenIndex:= Tokenliste.NextTokenIndex;
  AktuellerTokenIndex:= Tokenliste.NextTokenIndex;
  BLOCK(AktuellerTokenIndex);
  Aliprogramm.add('','EOJ','','   Ende der Anwweisungen');

  for i:= 1 to maxSymbole do
  begin
    if (Symboltabelle.Symbol(i)=name) and
       (Symboltabelle.Symbol(i-1)<>nicht_zulaessig)
    then Aliprogramm.add(Symboltabelle.Symbolstring(i),'DS','F','zwei Bytes');

  end;
  Aliprogramm.add('','END',m,'   Ende des Programms');
end; {of S, der Startregel}
```

Dadurch, dass keine Fehler abgefangen werden müssen, entfällt die if-then-else-Kaskadierung des Parsers. Stattdessen muss sich der Codierer um eventuelle Sprungmarken und neue Variablen kümmern. Ich erläutere das Verfahren an obiger Prozedur:

Bei Aufruf der Startregel S wurde das Token *prog* (`program`) bereits gelesen. Deshalb holt sich der Übersetzer direkt das nächste Token, welches das Token *name* ist. Die lokale Variable *m* erhält aus der Symboltabelle die entsprechende Zeichenkette. Anschließend wird das ALI-Programm mit der Anweisung `xxx START 0` begonnen.
Im Programm folgt das Semikolon (`;`), welches überlesen wird.
Im Programm folgt das `begin`, welches auch überlesen wird.
Jetzt kann die der folgende BLOCK übersetzt werden.
Wurde der BLOCK übersetzt, so folgt die ALI-Anweisung `EOJ`.
Anschließend müssen noch sämtliche Variablen in ALI deklariert werden. Dazu wird die Symboltabelle durchlaufen und alle Token *name* (bis auf den Programmnamen) in die ALI-Anweisung

`xxx DS F` übersetzt.

Abgeschlossen wird das ALI-Programm mit der Anweisung `END xxx` , wobei der zuvor gesicherte Programmname Verwendung findet.
Diese Prozedur war noch relativ einfach. Problematisch wird es bei der Produktionsregel ZUWEISUNG. Dort muss nämlich die Zuweisung, welche i. a. nicht im Dreiadressformat vorliegt, in dieses konvertiert werden. Dafür werden die beiden Stacks ODK und OTK genau wie im Algorithmus zur Zerlegung ins Dreiadressformat verwendet (Schaue dir den Algorithmus notfalls noch einmal an).

Aufgabe 18: 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 19: 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 20: Am Beispiel der wenn-dann-sonst-Anweisung kannst du dir klarmachen, wie Kontrollstrukturen mit Hilfe des Übersetzers in ALI-Assembler übersetzt werden können.
In Aufgabe 17 hast du die Grammatik bereits um eine Wiederhole-Anweisung erweitert. Erstelle auch die zugehörige Übersetzer-Routine.

Aufgabe 21: 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 22: 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 ( oder VERGLEICH )*
VERGLEICH ::= AUSDRUCK vergloperator AUSDRUCK
```