Lektion 20

Soll die Top-Down-Analyse in der Praxis eingesetzt werden, tauchen bei vielen Grammatiken Probleme auf. Gegebene Grammatiken haben oft nicht die LL(1)-Eigenschaft, so dass die effiziente Analyse nicht möglich ist. Durch geeignete Grammatikumformungen können aber in vielen Fällen (nicht in allen) die Probleme gelöst werden. Sie werden dazu einige typische Problemfälle kennenlernen und wie damit umgegangen werden kann. Es wird auch vorgestellt, wie ein sog. Parser-Generator arbeitet, der zu einer Grammatik den Programmcode für die Syntaxanalyse erzeugt.

11.5 Grammatik-Umformungen

Die Top-Down-Analyse ist ein elegantes, einfach zu realisierendes und effizientes Syntaxanalyseverfahren. Allerdings zeigt sich bei der Anwendung in der Praxis sehr schnell, dass viele Grammatiken, die zunächst für eine Anwendung definiert werden, nicht die LL(1)-Eigenschaft haben und somit nicht direkt verwendbar sind.

Soll trotzdem die Top-Down-Analyse verwendet werden, kann versucht werden, die Grammatiken äquivalent so umzuformen, bis die LL(1)-Eigenschaft erreicht wird. Dies ist allerdings nicht immer möglich, es gibt Sprachen, für die keine LL(1)-Grammatik existiert.

Für einige in der Praxis häufiger auftretende Problemfälle gibt es allerdings Standardlösungen.

Typische Probleme bei der effizienten Top-Down-Analyse

- **Mehrdeutigkeit** der Grammatik. Häufig tritt dies in der Praxis bei Grammatiken für arithmetische oder logische Ausdrücke auf. In solchen Fällen ist dann meist eine Umformen in eine nicht mehrdeutige Grammatik möglich, indem die Präzedenz und Assoziativität der Operatoren in der Grammatik mit berücksichtigt wird.
- **Linksrekursion** in Produktionen: Linksrekursion bedeutet, dass das Symbol der linken Seite einer Produktion am Anfang der rechten Seite vorkommt (z.B. A→ Ab). Linksrekursion kann immer systematisch durch Umformungen aus einer Grammatik eliminiert werden (indem sie in geeigneter Weise in Rechtsrekursion umgewandelt wird).
- **Produktionen mit gleichem Anfangsstück**: Beginnen mehrere Produktionen für ein nichtterminales Symbol mit dem gleichen Anfangsstück auf der rechten Seite (z.B. A → **a**A | **a**Bc), dann lässt sich oft der gemeinsame Teil ausgliedern

(sog. Linksfaktorisierung), so dass die Entscheidung zwischen den Alternativen zu einem späteren Zeitpunkt stattfindet, an dem dann die Vorausschau ausreicht, um die Fälle unterscheiden zu können.

11.5.1 Umformung mehrdeutiger Grammatiken

Folgende Grammatik für arithmetische Ausdrücke wurde schon früher verwendet. Wie leicht zu sehen ist, ist diese Grammatik mehrdeutig.

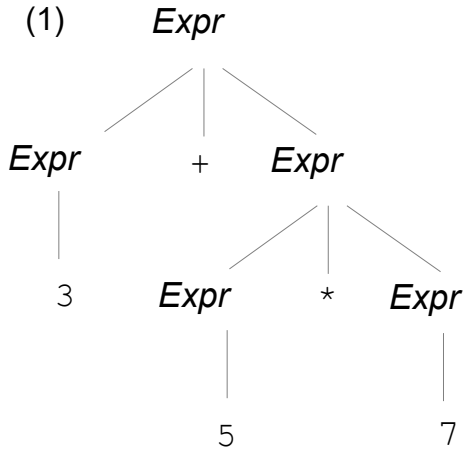
Beispiel 11.30 - Grammatik für arithmetische Ausdrücke

$$
\begin{aligned}
\mathit{Expr} \rightarrow\ & \mathit{Expr} + \mathit{Expr} \\
| \ & \mathit{Expr} - \mathit{Expr} \\
| \ & \mathit{Expr} * \mathit{Expr} \\
| \ & \mathit{Expr} \ / \ \mathit{Expr} \\
| \ & \mathit{Number} \\
| \ & (\ \mathit{Expr} \ ) \\
& \ldots
\end{aligned}
$$

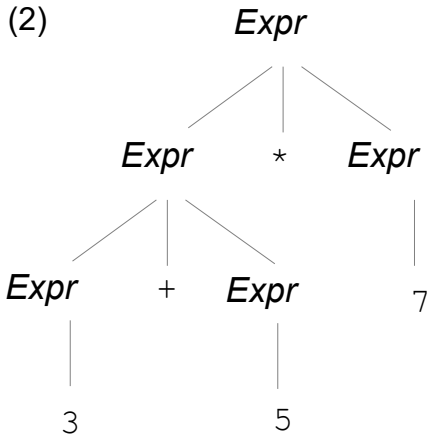
Wörter der Sprache sind z.B. `3 + 5 * 7` oder `3 - 2 - 1`.

- Es gibt zwei Ableitungsbäume für `3 + 5 * 7`:

(1)

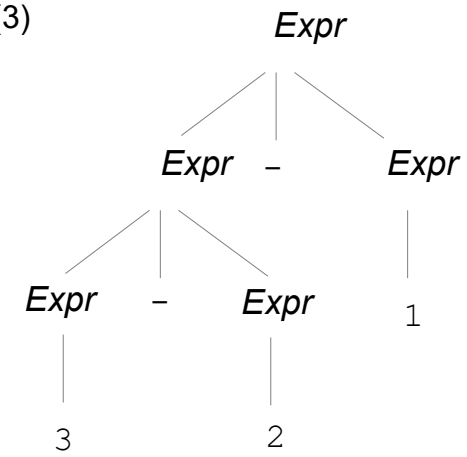


(2)

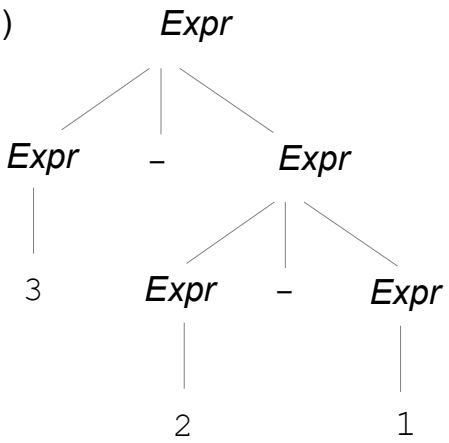


- Es gibt auch zwei Ableitungsbäume für `3 - 2 - 1`:

(3)



(4)



Bei beiden Ausdrücken gibt es einen "richtigen" Baum, der der üblichen mathematischen Leseweise entspricht, und einen "unerwünschten" Baum.

- Für `3 + 5 * 7` ist (1) der "richtige" Baum, da dies wie `3 + (5 * 7)` gelesen wird. Die Multiplikation hat höhere Präzedenz als die Addition, bei der Berechnung muss erst multipliziert werden, bevor addiert werden kann.
- Bei `3 - 2 - 1` ist (3) der "richtige" Baum, der der üblichen linksassoziativen Leseweise `(3 - 2) - 1` entspricht.

Eigenschaft 11.31 - Mehrdeutigkeit und LL(1)-Eigenschaft

Mehrdeutige Grammatiken haben niemals die LL(1)-Eigenschaft.

Auf einen Beweis dieser Eigenschaft wird hier verzichtet. Dass dies gilt, ist aber plausibel: Ist eine Grammatik mehrdeutig, dann gibt es für mindestens ein Eingabewort zwei unterschiedliche Ableitungsbäume. Bei einer Top-Down-Analyse für das Wort muss es dann die Situation geben, dass an einer Position des Eingabeworts bei gegebener Vorausschau zwei unterschiedliche Produktionen möglich sind und somit in der Vorausschautabelle ein Mehrfacheintrag vorliegen muss.

Inhärent mehrdeutige Sprachen

Viele (praktisch relevante) mehrdeutige Grammatiken können in äquivalente eindeutige Grammatiken umgeformt werden, aber es gibt auch kontextfreie Sprachen, für die es nur mehrdeutige Grammatiken gibt (sog. **inhärent mehrdeutige Sprachen**)!

Z.B. folgende Sprache ist kontextfrei und inhärent mehrdeutig:

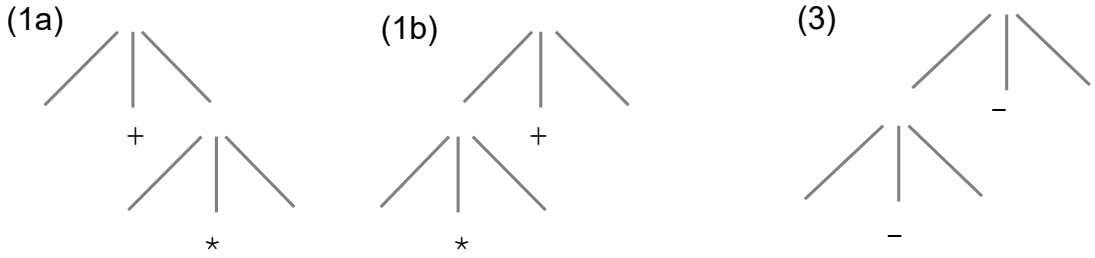
$$L = \{a^n b^n c^k d^k \mid n,k \geq 1\} \cup \{a^n b^k c^k d^n \mid n \geq 1, k \geq 1\}$$

Grundidee: Umwandlung mehrdeutiger Grammatiken für Ausdrücke

In der Mathematik (und entsprechend in Programmiersprachen) gibt es Präzedenzregeln für den Vorrang von Operatoren ("Punkt-vor-Strich-Rechung"), die sicherstellen, dass die Leseweise für einen Ausdruck eindeutig ist. Dies wird auch der Ansatzpunkt für die Umformung der Grammatiken sein, um die Mehrdeutigkeit zu eliminieren. Die Grammatik kann so verändert werden, dass die Präzedenzregeln in den Produktionen berücksichtigt sind, so dass Ableitungsbäume für die "falsche" Leseweise nicht mehr möglich sind.

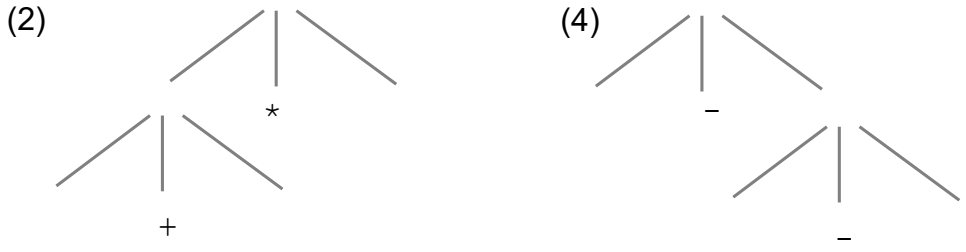
Wir betrachten, welche Teilstrukturen in einem Ableitungsbaum für Ausdrücke mit unterschiedlichen Operatoren "erlaubt" und welche "falsch" sind.

- "Erlaubte" Teilstrukturen eines Ableitungsbaums:



  - Operationen mit höherer Präzedenz müssen (sofern keine Klammern gesetzt sind), im Baum unter den Operationen mit niedrigerer Präzedenz vorkommen, da die Auswertung von unten nach oben erfolgt, siehe (1a), (1b).
  - Sind Operationen auf gleicher Präzedenzsstufe linksassoziative (wie beispielsweise (+, - , *, /), dann darf ein Operator gleicher Präzedenzstufe nur im linken Teilbaum vorkommen, siehe (3).

- "Falsche" Teilstrukturen eines Ableitungsbaums:



  - Bei (2) befindet sich der Operator mit niedriger Priorität (+) unter dem mit höher Priorität (∗) und würde dann fälschlicherweise zuerst ausgewertet.
  - Bei (4) werden beide Operatoren (-) mit gleicher Priorität nicht linksassoziativ, sondern rechtsassoziativ zusammengefasst, was auch ein falsches Ergebnis liefern würde.

Das folgende Beispiel zeigt, wie die Grammatik für Ausdrücke so erweitert werden kann, dass nur die "richtigen" Strukturen entstehen können. Dahinter steckt folgender Ansatz:

- In den Ableitungsbäumen sind die Operatoren mit niedriger Priorität weiter oben und die mit höher Priorität ggf. weiter unten.
- Linksassoziative Operatoren auf gleicher Präzedenzstufe können im linken Teilbaum folgen, aber nicht im rechten Teilbaum.

Beispiel 11.32 - Eindeutige Grammatik für Ausdrücke

$$
\begin{aligned}
Expr \quad \rightarrow \quad & Expr + Term \\
| \; & Expr - Term \\
| \; & Term
\end{aligned}
$$

Term → *Term* * *Factor*
| *Term* / *Factor*
| *Factor*

Factor → *number*
| (*Expr*)

Diese Grammatik ist nicht mehr mehrdeutig (aber aus anderen Gründen hat sich trotzdem noch nicht die LL(1)-Eigenschaft, siehe später).

- Priorität der Operatoren (* vor +) wird berücksichtigt. Durch die neu eingeführten nichtterminalen Symbole *Term* und *Factor* werden verschiedene Präzedenzstufen abgebildet.

  "Oben" im Baum bei *Expr* sind die Operatoren + und - mit niedriger Präzedenz, weiter unten bei *Term* kommen dann die Operatoren * und / mit höherer Priorität.

- Operationen mit gleicher Priorität werden linksassoziativ zusammengefasst. Durch die linksrekursiven Produktionen

  Expr → *Expr* + *Term*

  und

  Term → *Term* * *Factor*

  (entsprechend bei den anderen Regeln mit Operatoren) ist es möglich, dass der gleichartige Operator wieder im linken Teilbaum vorkommt, aber nicht im rechten.

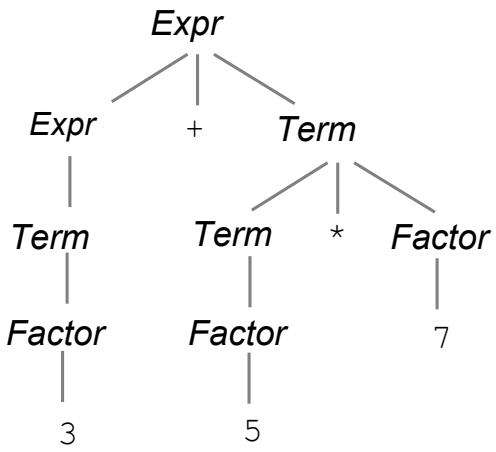
- Da es in einem Ausdruck nicht auf jeder Präzedenzstufe einen Operator geben muss, gibt es jeweils auch die "Kurzschluss-Produktionen"

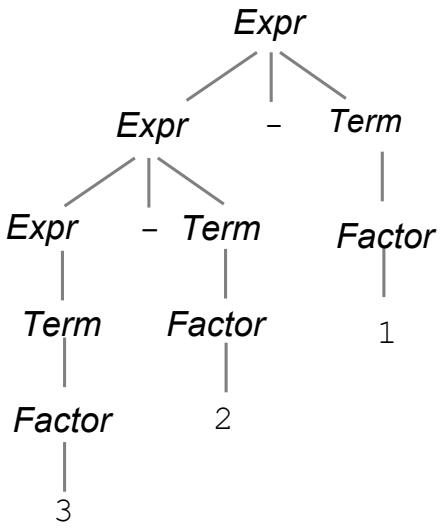
  Expr → *Term*

  und

  Term → *Factor.*

Damit ist dann für `3 + 5 * 7` bzw. `3 - 2 - 1` jeweils nur noch ein Ableitungsbaum möglich, der auch der mathematisch richtigen Leseweise entspricht:





Aufgabe 11.33 - Erweiterte Grammatik für Ausdrücke

Die im vorigen Beispiel 11.32 gebildete Grammatik für arithmetischer Ausdrücke mit den Operatoren `+`, `-`, `*` und `/` soll nun noch um einen Potenzoperator `^` erweitert werden. Z.B. `3^5` soll 3^5 bedeuten. Der Potenzoperator `^` soll höhere Präzedenz als der Multiplikationsoperator `*` haben und rechtsassoziativ sein. D.h.

```
5*4^3^2
```

sollte als

```
5 * (4 ^ (3 ^ 2))
```

verstanden werden. Die erweitertet Grammatik soll eindeutig sein und die Priorität der Operatoren entsprechend berücksichtigen.

11.5.2 Elimination von Linksrekursion

Neben der Mehrdeutigkeit gibt es noch ein zweites, häufig vorkommendes Problem für LL(1)-Grammatiken: die Linksrekursion. Linksrekursion bedeutet, dass das Nichtterminal der linken Seite am Anfang der rechten Seite vorkommt.
Wir betrachten hier nur den Fall direkter Linksrekursion, z.B.

$\underline{A} \rightarrow \underline{A}...,$

in analoger Weise tritt das Problem auch bei indirekter Linksrekursion auf:

$\underline{A} \rightarrow B$ ...
$B \rightarrow C$ …
$C \rightarrow \underline{A}$ …

Eigenschaft 11.34 - Linksrekursion und LL(1)-Eigenschaft

Grammatiken mit Linksrekursion haben niemals die LL(1)-Eigenschaft.

Das Problem wird an folgendem sehr simplen Beispiel deutlich.

Beispiel 11.35 - Grammatik mit Linksrekursion

$A \rightarrow Ab$
$\quad | \ a$

Die Sprache dieser Grammatik ist $L(G) = \{ab^n \mid n \geq 0\}$, als regulärer Ausdruck ab*.
Die Vorausschautabelle sieht so aus:

	a	b	$
A	A→ Ab A→ a	-	-

Wird ein Wort ab… analysiert, dann kann anhand des Zeichens a als Vorausschau nicht entschieden werden, ob die Produktion A→Ab oder A→a richtig ist, da $a \in$ First(Ab) und auch $a \in$ First(a) gilt.

Allgemeiner Fall der Linksrekursion

Wir betrachten den Fall eines Nichtterminals X einer kontextfreien Grammatik G = (N, T, P, S) mit einer linksrekursiven Produktion und einer Produktion ohne Linksrekursion:

(1) $X \rightarrow Xu$ linksrekursiv, mit $u \in (N \cup T)^*$,
(2) $X \rightarrow v$ v beginnt nicht mit X, $v \in (N \cup T)^*$
…

Bei der Berechnung der First-Mengen (Algorithmus 11.12) ergibt sich bei $X \rightarrow v$:

Alle Zeichen von First(v) werden in First(X) aufgenommen,
d.h. $First(v) \subseteq First(X)$

Wenn nun die Vorausschautabelle gemäß Definition 11.22 bestimmt wird, ergibt sich:

- Produktion $X \rightarrow v$ wird bei allen Zeichen aus First(v) eingetragen
- Produktion $X \rightarrow Xu$ wird bei allen Zeichen von First(Xu) eingetragen. Es gilt aber
  $First(X) \subseteq First(Xu)$ (Berechnung 11.10 First-Menge für Wörter)
  und
  $First(v) \subseteq First(X)$ (s.o.)
  Somit wird diese Produktion auch bei allen Zeichen aus First(v) eingetragen, d.h. für alle Zeichen aus First(v) gibt es Mehrfacheinträge in der Vorausschautabelle.

Linksrekursion ist ein generelles Problem für die LL(1)-Eigenschaft von Grammatiken. Es ist aber ein relativ "angenehmes" Problem:

- Linksrekursion kann leicht erkannt werden (einmal genau Hinschauen reicht bei üblichen Beispielaufgaben).
- Linksrekursion kann immer systematisch durch Umformung der Grammatik eliminiert werden.

Problemlösung: Linksrekursion durch Rechtsrekursion ersetzende

Die Grundidee für die Elimination der Linksrekursion ist einfach:

- Eine Linksrekursion beschreibt immer eine Wiederholung von Teilen.
- Eine Wiederholung kann in einer Grammatik auch durch Rechtsrekursion formuliert werden und Rechtsrekursion ist für die LL(1)-Eigenschaft unproblematisch.

Beispiel 11.36 - Umwandlung von Links- in Rechtsrekursion

Die linksrekursive Grammatik

$A \rightarrow Ab \mid a$

definiert die Sprache ab*. Eine Wiederholung von beliebig vielen b kann auch durch eine rechtsrekursive Produktion formuliert werden, wenn man ein Hilfssymbol R für b* einführt.

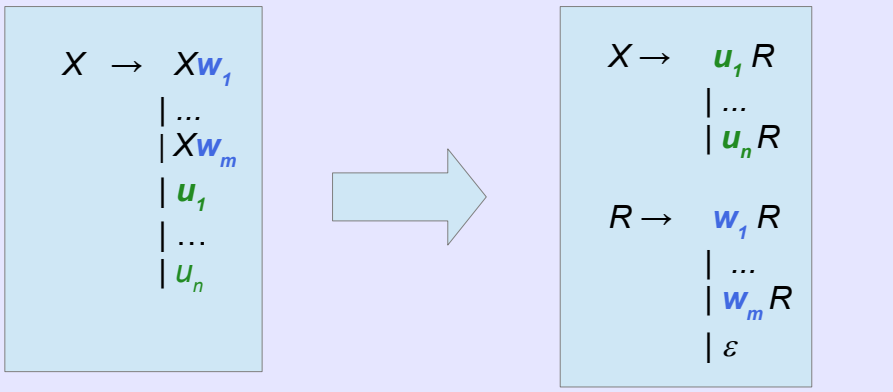
$A \rightarrow aR$
$R \rightarrow bR \mid \varepsilon$

Diese Grammatik hat nun die LL(1)-Eigenschaft.

Im allgemeinen Fall kann es zu einem Nichtterminal X mehrere linksrekursive Produktionen $X \rightarrow Xw_i$ geben und mehre Produktionen ohne Linksrekursion $X \rightarrow u_j$.

Umwandlungsregel 11.37 - Elimination von Linksrekursion

Die Wörter w_i und u_j in folgender Transformationsregel können beliebige Folgen von terminalen oder nichtterminalen Symbolen sein. In den Folgen u_j steht X nicht am Anfang.



Produktionen für X mit/ohne Linksrekursion

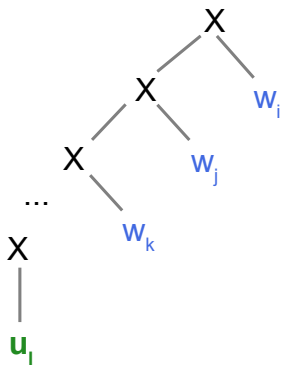
äquivalente Produktionen ohne Linksrekursion (*R* ist neuer Nichtterminal)

Diese Umwandlungsregel kann man sich leicht herleiten, wenn man EBNF als Zwischenschritt verwendet.

Stellt man sich die möglichen Ableitungsbäume vor (siehe rechts), erkennt man, dass die Ausgangsgrammatik durch folgende EBNF-Produktion zusammengefasst werden kann:

$X \rightarrow (u_1 \mid \ldots \mid u_n)\ (w_1 \mid \ldots \mid w_m)^*$

Es kommt also genau einer der u_l-Teile am Anfang,



danach können beliebig viele w_j-Teile in beliebiger Reihenfolge folgen.

Der Wiederholungsteil (w_1 | … | w_m)* kann, wir oben mit *R* angegeben, rechtsrekursiv beschrieben werden.

Aufgabe 11.38 - Elimination von Linksrekursion

Wandeln Sie folgende Grammatik in eine nicht linksrekursive Grammatik um.

S → AB

A → aBa
| AaB
| bb
| AaA

B → ba

11.5.3 Linksfaktorisierung

Ein weiteres typische Problem bei der Top-Down-Analyse sind gleiche Anfangsstücke der rechten Seite bei mehreren Produktionen zum gleichen Nichtterminal.

Beispiel 11.39 - Gemeinsame Anfangsstücke

Folgender Teil einer Grammatik beschreibt Anweisungen einer Programmiersprache.

Stmt	→	*id* = *Expr* ;	*(1)Zuweisung an Variable*
		| *id* **[** *Expr* **]** = *Expr* ;	*(2)Zuweisung an Arrayelement*
		| **if** **(** *Expr* **)** *Stmt*	
		| **while** **(** *Expr* **)** *Stmt*	
		...	

Es gibt Wertzuweisungen an Variablen, an Elemente eines Arrays, Fallunterscheidungen und While-Schleifen. *id* stellt hier ein Symbol für einen Bezeichner einer Variable oder eines Arrays dar ("Identifier"). Bezeichner und Schlüsselwörter (**if**, **while**) sollen jeweils als ein Symbol (Token) betrachtet werden.

Durch ein Symbol Vorausschau kann bei Eingabesymbol *id* nicht entschieden werden, ob Produktion (1) oder (2) gewählt werden soll.

Linksfaktorisierung

Eine Lösung ist hier einfach durch Faktorisierung ("Ausklammern") der gemeinsamen Anfangsstücke möglich.

Stmt →	*id R*	
|	**if** **(** *Expr* **)** *Stmt*	
|	**while** **(** *Expr* **)** *Stmt*	

$$
\begin{array}{lll}
 & \ldots & \\
R \rightarrow & & = \ Expr \ ; \\
 & | & [\ Expr \ ] = Expr \ ;
\end{array}
$$

Aufgabe 11.40 - Linksfaktorisierung

Gegeben ist folgende Grammatik, die nicht die LL(1)-Eigenschaft hat.

$$
\begin{array}{lll}
S \rightarrow & & aSa \\
 & | & bAb \\
 & | & bAc \\
A \rightarrow & & ab \\
 & | & aAb
\end{array}
$$

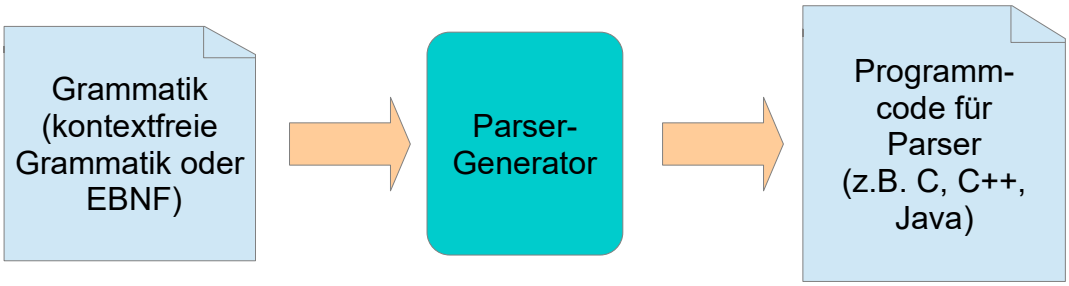
Welche Probleme sind erkennbar?

Formen Sie die Grammatik in eine LL(1)-Grammatik um

11.6 Ausblick: Parser-Generatoren

Soll in der Praxis eine Sprache implementiert werden, z.B. eine Programmiersprache oder eine sog. *Domain Specific Language* (DSL, also eine Spezialsprache für ein spezielles Anwendungsgebiet), werden die lexikalische und syntaktische Analyse üblicherweise nicht von Hand implementiert, sondern durch Scanner- und Parsergeneratoren aus Spezifikationen der Sprache generiert. Moderne Tools (z.B. ANTLR oder javacc) beinhalten gleichzeitig den Scanner- und den Parsergenerator, so dass die Integration zwischen beiden Teilen automatisch erfolgt. Die oben angesprochenen Tools erlauben es auch, die Grammatik bequem mittels EBNF zu spezifizieren.

Funktionsweise eines Parser-Generators



z.B. Top-Down-Parser mittels rekursivem Abstieg

Merkmale des Parser-/Scanner-Generators ANTLR

- Die Spezifikation der lexikalischen Einheiten und der Grammatik erfolgt in einheitlicher Weise gemeinsam in einer Datei. Für die Grammatik kann EBNF verwendet werden.
- Es wird Code für einen Top-Down-Parser mit rekursivem Abstieg generiert
- Es kann auch mit einem Lookahead k>1 gearbeitet werden, entweder lokal in einzelnen Grammatikregeln oder für die gesamte Grammatik. Der Generator bietet eine Reihe von Erweiterungen, so dass auch viele Sprachen, die nicht die LL(1)-Eigenschaft haben, bequem verarbeitet werden können.
- Es kann Scanner/Parser-Code in verschiedenen Programmiersprachen generieren werden, z.B. Java, C++, C#, Python, JavaScript, Swift und Go.

Beispiel 11.41 - ANTLR-Grammatik für THULogo

Die in früheren Beispielen erwähnt Programmiersprache THULogo wurde mit Hilfe von ANTLR implementiert. Die komplette Beschreibung für Scanner und Parser ist hier zu sehen.

```
grammar THULogo;
options { k=1; }            // 1 Zeichen Lookahead
program:
        'program' STRING (definition | statement)* EOF ;
definition:
        'to' ID '(' ( VAR )*  ')' statement*  'end';
statement:
        ('fd' | 'forward') expression
        | ('tr' | 'turnright') expression
        | ('tl' | 'turnleft') expression
        | 'clear'
        | ('pu' | 'penup')
        | ('pd' | 'pendown')
        | 'setpencolor' expression expression expression
        | 'setbackcolor' expression expression expression
        | 'moveto' expression expression
        | 'write' STRING
        | 'wait' expression
        | 'output' expression
        | 'make' VAR expression
        | 'repeat' expression block
        | 'if' condition block
        | 'show' STRING expression
        | 'read' STRING VAR
        | ID '(' expression* ')' ;
block:
        '[' statement* ']'            ;
condition:
        andCondition ('or' andCondition)* ;
```

```
andCondition:
        negCondition ('and' negCondition )* ;
negCondition:
        ('not')? cmpCondition;
cmpCondition:
        expression ( compareOp expression )? ;
compareOp:
        '<' | '<=' | '=' | '>' | '>=' | '<>' ;
expression:
        term ( ('+' | '-' ) term )*;
term:
        unopExpression ( ( '*' | '/' ) unopExpression )? ;
unopExpression:
        ('-')? factor ;
factor :
        NUMBER   | VAR | ID '(' expression* ')'
        | '(' expression ')' ;
VAR:
        ':' ID ;
ID:
        ('a'..'z'|'A'..'Z'|'_')
           ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
NUMBER:
        ('0'..'9')+  ( '.' ('0'..'9')+ )? ;
STRING:
        '"' ~('\n'|'\r'|'"')* '"' ;
COMMENT:
        ';' ~('\n'|'\r')* '\r'? '\n' { $channel=HIDDEN;};
WS:
        ( ' ' | '\t' | '\r' | '\n' ) {$channel=HIDDEN;}      ;
```

Auch wenn Sie die Details der ANTLR-Notation nicht verstehen, ist erkennbar, wie mit einer überschaubaren Anzahl von Zeilen eine realistische Programmiersprache definiert werden kann.

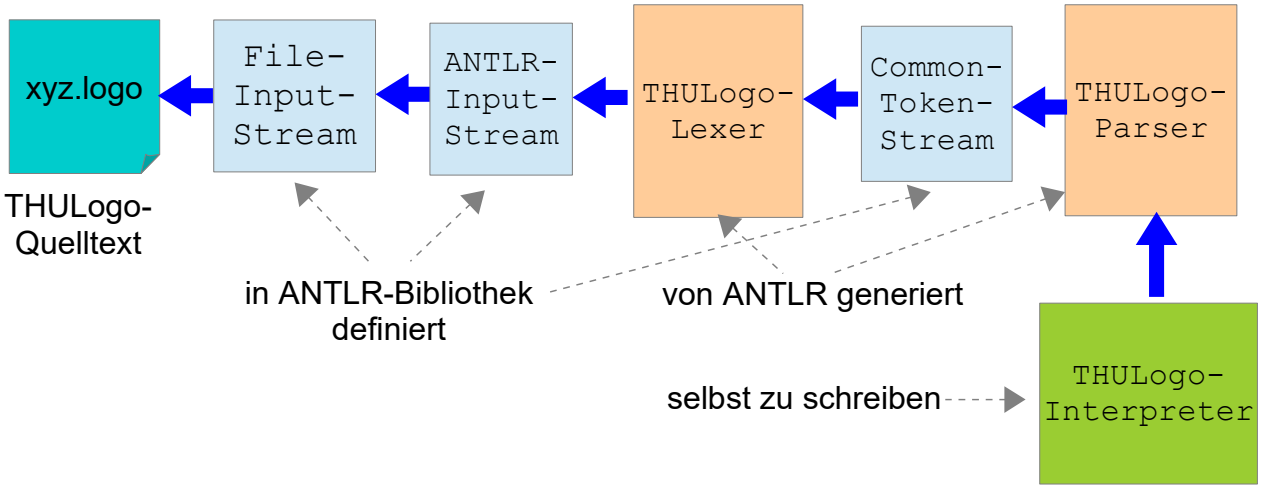
Da für die Grammatikspezifikation EBNF verwendet werden kann, sehen die Grammatikregeln für die nichtterminalen Symbole (klein geschrieben, z.B. `program`, `statement`) und die Definitionen für die lexikalischen Einheiten (= Token, terminale Symbole / große geschrieben, z.B. `VAR`, `ID`, `NUMBER`, …) gleichartig aus, da für beide Zwecke reguläre Ausdrücke verwendet werden.

Es ist auch zu sehen, dass die Präzedenz der arithmetischen und logischen Operatoren in der THULogo-Grammatik durch unterschiedliche "Stufen" bei den nichtterminalen Symbolen berücksichtigt ist, so wie in Abschnitt 11.5.1 vorgestellt.

Da EBNF verwendet werden kann, lassen sich bei den Produktionen für arithmetische oder logische Ausdrücke die Probleme mit Linksrekursion vermieden, da Wiederholungen direkt mit * angegeben werden können und nicht durch Links- oder Rechtsrekursion ausformuliert werden müssen.

Verwendung des generierten Scanners und Parsers

Aus der Sprachspezifikation werden von ANTLR zwei Klassen für den Lexer (Scanner) und den Parser generiert. Mit weiteren Hilfsklassen aus der Java- und ANTLR-Bibliothek kann dann mit wenigen Zeilen Code die komplette lexikalische und syntaktische Analyse realisiert werden:



Der verbindende Code zum Aufruf des generierten Lexers und Parsers im HSULogo-Interpreter sieht folgendermaßen aus.

```
String filename = ...
InputStream fin = new FileInputStream(filename);
ANTLRInputStream input = new ANTLRInputStream(fin);
THULogoLexer lexer = new THULogoLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
THULogoParser parser = new THULogoParser(tokens);

parser.program();     // 'program' ist das Startsymbol
```

Es wird die oben dargestellte Verarbeitungskette aus Scanner und Parser aufgebaut. dann wird zum Start der Analyse die Methode für das Startsymbol *program* aufgerufen.

Für eine Weiterverarbeitung des analysierten Codes bietet ANTLR mehrere Möglichkeiten:

- Es kann automatisch ein abstrakter Syntaxbaum (AST) aufgebaut werden, der als Ergebnis der Analyse geliefert wird. Die weitere Verarbeitung kann dann auf Basis des AST erfolgen.
- Es können in der Grammatikspezifikation sog. *semantische Aktionen* mit angegeben werden. Das sind Programmstücke, die in den generierten Code übernommen werden, so dass sie ausgeführt werden, sobald entsprechende Teil der Grammatik erkannt wurden.

Zusammenfassung zu Lektion 20

Diese Fragen sollten Sie nun beantworten können

- Welche Eigenschaften einer Grammatik führen in praktischen Anwendung oft dazu, dass sie nicht die LL(1)-Eigenschaft hat?
- Wie kann eine mehrdeutige Grammatik für Ausdrücke umgeformt werden, so dass die Priorität der Operatoren berücksichtigt wird uns sie nicht mehr mehrdeutig ist?
- Wie kann Linksrekursion aus einer Grammatik eliminiert werden?
- Was versteht man unter Linksfaktorisierung bei Grammatiken? Wozu wird sie verwendet?
- Wie funktioniert prinzipiell ein Parser-Generator?