Syntaxanalyse – Ausgangspunkt und Ziel

- Ausgangspunkt: Kontextfreie Grammatik
- Im Normalfall BNF, manchmal EBNF

  BNF = Backus-Naur-Form = Produktionsregeln

  EBNF = erweiterte BNF (+ reguläre Ausdrücke)
- Prüfung der Korrektheit der Eingabe
- Darstellung der semantischen Bedeutung der Eingabe in anderer Form

  (Baumstruktur, Datenbank, anderes Datenformat)
- Erkennung der Semantik soll durch Syntaxanalyse unterstützt werden.

Syntaxanalyse – Einschränkungen

- Kontextfreie Grammatik
- Eindeutigkeit der Grammatik: es gibt genau einen Ableitungsbaum
  (es gibt unterschiedliche Ableitungen, je nachdem in welcher Reihenfolge man vorgeht. Bei eindeutiger Grammatik ist aber bei jeder Ableitung eines Nichtterminals immer genau eine Regel anwendbar.)
- Theorie verlangt die Existenz einer Ableitung.
- Compilerbau verlangt, dass die Ableitung effizient gefunden wird.
  (möglichst kein Backtracking)
- Es sollte leicht möglich sein, die Übersetzung an die Syntaxanalyse anzuhängen.
- Je nach verwendeter Methode gibt es mehr oder weniger große Einschränkungen in der Formulierung der Grammatik

Syntaxanalyse – Top Down Verfahren

- Ausgangspunkt: Eingabe, Startsymbol
- Zu jedem Zeitpunkt steht eine Satzform zu weiteren Ableitung an.
- Die Eingabe wird von links nach rechts verarbeit. L
- Die Satzform wird von links nach rechts verarbeitet. LL
- Bei Entscheidungen werden die k nächsten Eingabesymbole betrachtet (hier nicht 1) LL(k)
- Terminalsymbole müssen mit der Eingabe übereinstimmen. (match)
- Nichtterminale werden durch ihre Ableitung ersetzt. Bei der Auswahl der passenden Regel werden die k-ersten Anfangssymbole der rechten Seiten einer Regel benutzt (hier nur das 1. Symbol)
- Wenn die rechte Seite einer Regel leer ist, entscheidet das Folgesymbol der linken Seite (Erklärung kommt noch).

LL(1)-Verfahren

- **Programmierte Lösung: rekursiver Abstieg**

  match() vergleicht Terminalsymbole und steuert die Eingabe

  Nichtterminale werden durch Prozeduren implementiert.

  Entscheidungen werden über Lookahead-Symbol getroffen.

  Ableitungen werden durch Prozeduraufrufe realisiert.

  Folge von Symbolen = Anweisungsfolge.

- **Tabellengesteuerte Verfahren**

  Stack speichert die noch zu verarbeitenden Symbole

  Das Top-Stack Symbol wird verarbeitet.

  Ein Terminalsymbol wird mit der Eingabe gematcht.

  Ein Nichtterminalsymbol wird durch die rechte Seite einer Regel ersetzt.

  Über die Auswahl der Regel entscheidet eine Tabelle.

Beispiel für LL(1)-Verfahren

1) $E \rightarrow T\ R$
2) $T \rightarrow z$
3) $T \rightarrow (\ E\ )$
4) $R \rightarrow +\ T\ R$
5) $R \rightarrow$

Terminalsymbole: { z, +, (,)} + $
Nonterminalsymbole: { E, T, R}
Startsymbol: E

	z	+	**(**	**)**	**$**
E	1	-	1	-	-
T	2	-	3	-	-
R	-	4	-	5	5

Spalten: Eingabesymbole + $ = Ende
Zeilen: Nichtterminalsymbole

Schnittpunkte: anzuwendende Regel

Syntaxerkennung nach LL(1)

```
while true:
  if isDollar(tos):
    return isDollar(lookahead)
  else if isTerminal(tos):
    match(tos) or Synaxerror
  else:// isNonterminal(tos)
    nt = pop()
    push(apply(nt, lookahead))

match(symbol):
  if symbol == input[i]:
    i=i+1
  else:
    Syntaxerror

apply(ntsym, tsym):
    nr = tab[ntsym, tsym]
    if nr == -:
      Syntaxerror
    else:
      return umgedrehte rechte Seite
        von Regel nr
```

Eingabe	Stack	Bemerkung
z + (z + z) $	E $	Regel 1
z + (z + z) $	T R $	Regel 2
z + (z + z) $	z R $	match
+ (z + z) $	R $	Regel 4
+ (z + z) $	+ T R $	match
(z + z) $	T R $	Regel 3
(z + z) $	(E) R $	match
z + z) $	E) R $	Regel 1
z + z) $	T R) R $	Regel 2
z + z) $	z R) R $	match
+ z) $	R) R $	Regel 4
+ z) $	+ T R) R $	match
z) $	T R) R $	Regel 2
z) $	z R) R $	match
) $	R) R $	Regel 5
) $	) R $	match
$	R $	Regel 5
$	$	**accept**

LL(1) – Tabelle

Die Tabelle steuert die Entscheidung welche Regel bei der Ableitung eines Nichterminalsymbols zu verwenden ist. Die Entscheidung hängt von dem nächsten Eingabesymbol ab (lookahead-Symbol)

Notwendige Bedingungen:

Eine Regelanwendung kann nur erfolgreich sein, wenn das lookahead-Symbol gleich einem der möglichen Anfangsterminale der rechten Seite ist.
(First-Symbole)

Wenn die rechte Seite einer Regel leer ist, kann die Regelanwendung nur dann erfolgreich sein, wenn das lookahead-Symbol gleich einem der Terminalsymbole ist, die bei einer gültigen Ableitung hinter dem Symbol der linken Seite stehen können.
(Follow-Symbole)

Die Tabellenkonstruktion führt nur dann zu einem brauchbaren Verfahren, wenn in jeder Zelle höchstens eine Regelnummer steht.
(LL(1)-Bedingung)

Ermittlung der Firstmenge

Die Funktion **first(*folge*)** ordnet einer **Folge von (beliebigen) Symbolen** eine M**enge von Terminalsymbolen** zu. In der Ergebnismenge kann auch das **leere Symbol (ε)** enthalten sein.

Ein Nichtterminalsymbol **N** habe mehrere Regeln der Form $N \rightarrow R_i$. **T** steht für ein Terminalsymbol. α steht für eine Folge von Terminal und Nichtterminalsymbolen (kann auch leer sein).. Eine Symbolfolge heißt ***nullable***, wenn sie die Firstmenge ε enthält.

Für die Ermittlung der Firstmenge gelten die folgenden Regeln:

- **first(Tα) = {T}**
- **first(ε) = {ε}**
- **first(N) = U_i first(R_i),** wobei R_i eine rechte Seite von N-Regeln ist.
- **first(Nα) =**
  first(N) \ ε ∪ first(α), nullable(N)
  first(N), sonst

Ermittlung der Followmenge

Die Funktion **follow(N)** ordnet einem **Nichtterminalsymbol N** eine **Menge von Terminalsymbolen** zu. In der Ergebnismenge kann auch das **Endesymbol ($)** gehören (aber nicht das leere Symbol ε).

Ein Nichtterminalsymbol N habe mehrere Regeln der Form $N \rightarrow R_i$. T steht für ein Terminalsymbol. α und β stehen für eine Folge von Terminal und Nichtterminal-symbolen.

Für die Ermittlung der Followmenge von N gelten die folgenden Regeln:

- **Wenn N das Startsymbol ist, so gilt $ in follow(N)**
- **Wenn N auf der rechten Seite einer Regel der Form α N β gefolgt von der Symbolfolge β steht, so gilt: first(β) \ ε in follow(N).**
- **Wenn X $\rightarrow \alpha$N oder X $\rightarrow \alpha$Nβ mit nullable(β), so gilt follow(X) $\subseteq$ follow(N).**

LL(1)-Tabellenkonstruktion

Die Tabelle enthält für jedes Nichtterminalsymbol der Grammatik eine Zeile (in der ersten Zeile steht das Startsymbol) und für jedes Terminalsymbol und für das Endezeichen $ eine Spalte.

Der Inhalt einer Zeile ergibt sich aus den Regeln für das entsprechende Nichtterminalsymbol.

Das Nichtterminalsymbol habe mehrere (mit *i* numerierte) Regeln N -> R_i.

Für jedes Terminalsymbol T (nicht für ε) aus first(R_i) wird der Verweis auf die Regel *i* in der Spalte des Terminalsymbols T eingetragen.

Gilt ε in first(R_i), so wird die Regel *i* in die Spalte der Symbole aus follow(R_i) eingetragen.

Freibleibende Tabellenfelder bezeichnen Syntaxfehler.

Wird nach diesen Regeln ein Tabellenfeld doppelt besetzt, so ist die Tabellenkonstruktion fehlgeschlagen.

Die Grammatiken, bei denen die LL(1)-Konstruktion funktioniert, nennt man LL(1)-Grammatiken.

LL(1)-Bedingungen

Eine Grammatik ist in LL(1), wenn sie den folgenden Bedingungen für jedes Nichtterminalsymbol N genügt:

- Die Firstmengen verschiedener Regeln eines Nichterminalsymbols N sind untereinander disjunkt.

- Enthält die Firstmenge einer rechten Seite eines Nichterminalsymbols N das leere Symbol ε, so ist follow(N) mit den Firstmengen aller rechten Seiten von N disjunkt.

Beispiel non LL(1) - Grammatik

1) $E \rightarrow E + T$
2) $E \rightarrow T$
3) $T \rightarrow z$
4) $T \rightarrow (E)$

Terminalsymbole: { z, +, (,) } und $

Nonterminalsymbole: { E, T }

Startsymbol: E

	z	+	(	)	$
E	1/2	-	1/2	-	-
T	3	-	4	-	-

für E:

first(E+T) = { z, (}

first(T) = { z, (}

für T

first(z) = { z }

first((E)) = { (}

Grammatik für Scheme

Eine *Scheme Ausdruck* ist entweder ein *Atom* oder eine *Liste*.

Eine *Liste* ist eine *eingeklammerte Folge* von *Scheme Ausdrücken*.

Ein *Atom* ist ein *Wort* oder eine *Zahl*.

Beispiele: `(+ alfa 4),  (define (a x) (+ x 1)),  345, alfa`

- Schreiben Sie formal die Grammatik auf.
- Konstruieren Sie eine LL(1)-Tabelle.
- Führen Sie die Syntaxanalyse für ein Beispiel durch.
- Frage am Rande: wie sind in Scheme Wörter definiert?

JavaCC

```
InstructionNode ifInstruction() :
{    // Deklaration
     Node block;
     Node cond;
     IfInstructionNode node;
}
{    // Syntax
  (
     <IF> cond=expression() <COLON> block=block()
          { node = new IfInstructionNode(cond, block);}
     ( <ELSEIF> log=expression() <COLON> block=block())
          { node.addElseIfBranch(cond, block);
     )*
     ( <ELSE> <COLON> block=block())
          { node.addElseBranch(block);
     )?
  )
  { return node; }
}
```

Rekursiver Abstieg – eine LL(1) Variante

- Tabellengesteuerte Methoden legen die Verwendung von Parsergeneratoren nahe.

- Das gibt es auch im Fall von LL(1): javacc

- Tabellengesteuerte Verfahren benötigen in jedem Fall eine Kopplung zur Programmiersprache.

- Zusätzlich haben Tabellengesteuerte Verfahren häufig die Möglichkeit, die Nachteile von LL(1) durch besondere Eingriffsmöglichkeiten in den Parserablauf (z.B. gezielte Vergrößerung des Lookahead) abzumildern.

- Ein sinnvolle Alternative ist die durchgängige Programmierung des Parsers nach der Methode des **rekursiven Abstiegs**.

Rekursiver Abstieg - Vorbereitung

Als Scannerschnittstelle verwenden wir die Methoden

```
interface LL1Scanner {
  /**
   * Liefert das aktuelle Lookahead-Symbol
   */
  int lookahead()

  /**
   * Matched ein Symbol und liest das nächste Symbol von der
   * Eingabe.
   * @param zu prüfendes Terminalsymbol
   * @throws SyntaxException wenn das Symbol nicht mit dem
   * Lookahead-Symbol übereinstimmt
   */
  void match(symbol)
}
```

Terminale werden durch match(sym) dargestellt.
Nichterminale werden durch Parsermethoden repräsentiert.
Ableitungen werden zu Methodenaufrufen.
Entscheidungen werden durch Vergleich mit lookahead() getroffen.

Rekursiver Abstieg - Methodik

- Terminalsymbole sind durch Konstanten kodiert.
- Für jedes **Nichtterminal** schreiben wir eine eigene Methode.
- Der Auswahl der richtigen Regel erfolgt durch Vergleich mit **lookahead().**
- Die Folge der Symbole der rechten Seite einer Regel ergibt eine Folge von Anweisungen.
- Für jedes Terminalsymbol rufen wir **match(tsym)** auf.
- Für jedes **Nichtterminalsymbol** rufen wir dessen Methode auf.

Rekursiver Abstieg - Beispiel

```
void expression() {                          // expression -> term termRest
    term(); termRest();
}

void term() {
    if (lookahead() == ZAHL)                 // term -> zahl
        match(ZAHL);
    else if (lookahead() == KLAUF) {         // term -> ( expression )
        match(KLAUF); expression(); match(KLZU);
    }
    else
        throw new SyntaxException();
}

void termRest() {
    if (lookahead() == PLUS) {               // termRest -> + term termRest
        match(PLUS); term(); termRest();
    }
    if (lookahead()==ENDE||lookahead()==KLZU)      // termRest ->
         /* nichts */
    else
        throw new SyntaxException();
}
```

(anstelle von if kann man auch switch verwenden)

Rekursiver Abstieg – Bemerkungen

- Rekursiver Abstieg lässt sich **schematisch** aus der Syntaxbeschreibung herleiten.
- Rekursiver Abstieg funktioniert genau dann, wenn die Syntax **LL(1)** ist.
- Rekursiver Abstieg lässt sich leicht durch **programmierte Aktionen** für die Übersetzung erweitern.
- Rekursiver Abstieg in der dargestellten Form nutzt nicht alle Kontrollstrukturen (Wiederholung durch **Rekursion**)
- Rekursiver Abstieg lässt sich besser bei einer in **EBNF** formulierten Grammatik einsetzen (die Nachteile von LL(1) werden dabei etwas ausgeglichen)
- Rekursiver Abstieg lässt programmierte **Optimierungen** zu.

EBNF

- EBNF = BNF + reguläre Ausdrücke
- [...] für 0-1 mal, manchmal auch (...)?
- { ... } für 0-n mal, manchmal auch (...)*
- (...) für Schachtelung
- | für Alternative

Die Umsetzungsregeln ergeben sich daraus, dass man jede EBNF formal in eine BNF umwandeln kann.

Vereinfacht: [...] in eine Fallunterscheidung, { ... } in eine Iteration.

Eine Besonderheit ist, dass man jetzt genau überlegen muss, von welchen Bedingungen Fallunterscheidungen und Wiederholungen abhängen.

EBNF-Grammatik für Ausdrücke

```
expression → term ( + term )*
term → z | ( expression )
```

```
void expression() {
    term();
    while (lookahead() == PLUS) {     // { + term }
        match(PLUS); term();
    }
}

void term() {
    if (lookahead() == ZAHL)
        match(ZAHL);
    else if (lookahead() == KLAUF) {
        match(KLAUF); expression(); match(KLZU);
    }
    else
        throw new SyntaxException();
}
```

Unwandlung von EBNF in BNF

Abschließend soll kurz dargestellt werden, dass EBNF letztlich nur eine abgekürzte BNF ist. EBNF Formeln können nämlich einfach umgefwandelt werden:

`( Ausdruck )` wird zu

```
Klammer ::= Ausdruck
```

`{ Ausdruck }` wird zu

```
Wiederholung ::= ;
Wiederholung ::= Wiederholung Ausdruck ;
                 // oder: Ausdruck Wiederholung
```

`[ Ausdruck ]` wird zu

```
Option ::= ;
Option ::= Ausdruck ;
```