Compilerbau CS1019

Th. Letschert

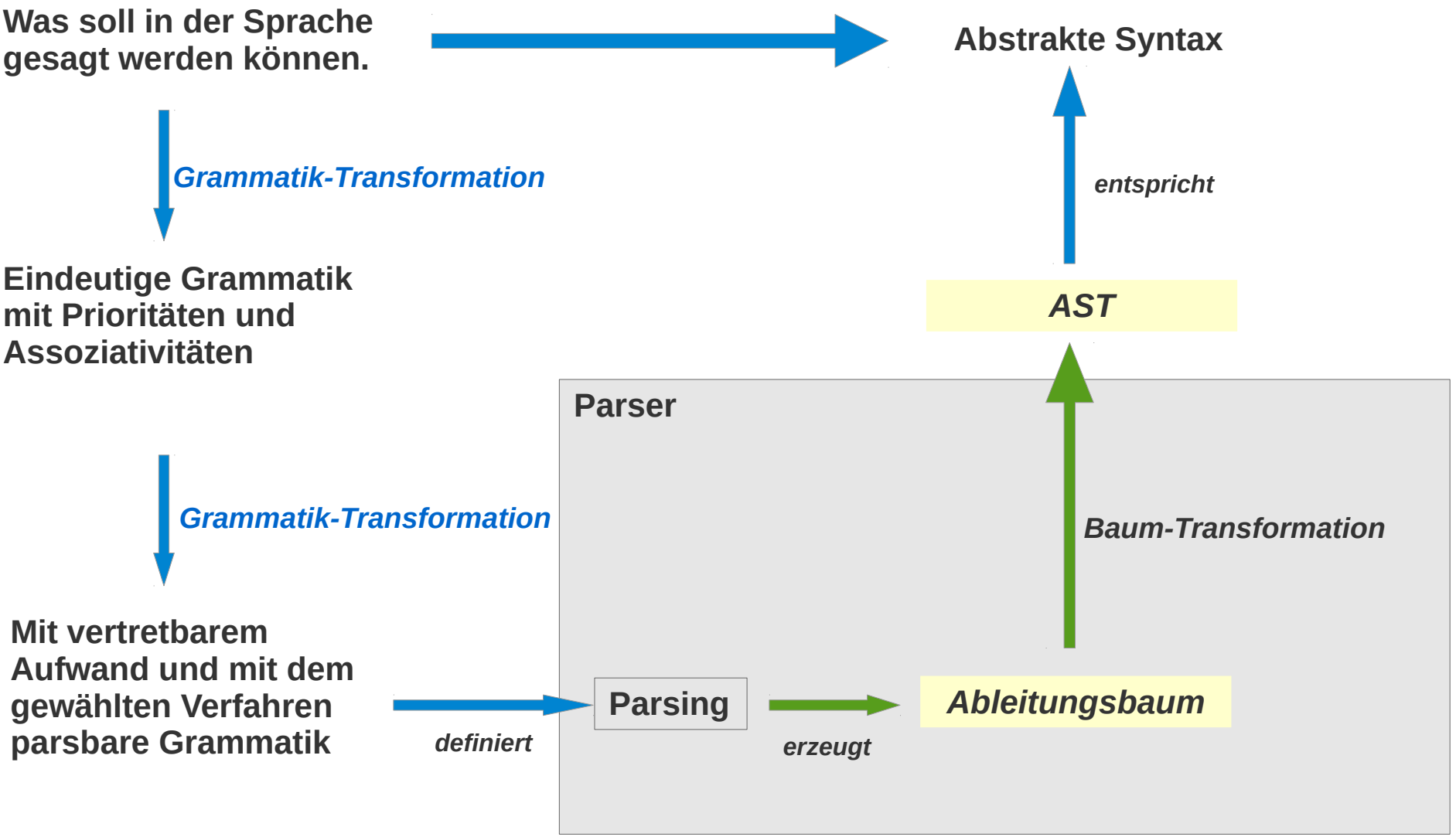
TH Mittelhessen Gießen

University of Applied Sciences

Parser

- Parser und Grammatiken
- Parsen: Top-Down / Bottom-up,
- Top-Down Prädikatives Parsen, Rekursiver Abstieg
- Grammatik-Transformation: Eindeutig parse-bar, mit Assoziativitäten, Präzedenzen
- Baum-Transformation: Ableitungsbaum nach AST

Übersicht


Was soll in der Sprache gesagt werden können.
Abstrakte Syntax
Grammatik-Transformation
entspricht
Eindeutige Grammatik mit Prioritäten und Assoziativitäten
AST
Parser
Grammatik-Transformation
Baum-Transformation
Mit vertretbarem Aufwand und mit dem gewählten Verfahren parsbare Grammatik
definiert
Parsing
erzeugt
Ableitungsbaum

Übersicht / Einordnung: Grammatik und Parsing

Grammatiken

- beschreiben Sprachen

Sprachen

- sind unendliche Mengen von Texten (die „Sätze“ / „Wörter“ der Sprache)

Sprache und Grammatik I

Sprache und Grammatik ist nicht das Gleiche:

- Jede Grammatik definiert eine Sprache (= Menge von korrekten Texten)
- Eine Sprache kann aber von vielen Grammatiken definiert werden.
  Unser primäres Interesse gilt den Sprachen, Grammatiken sind ein Hilfsmittel

Sprache und Grammatik II

Genau gesagt sind wir an zwei Aspekten einer Sprache interessiert

- Was sind die korrekten Texte?
- Welche Struktur haben die korrekten Texte?

Aber: Da die Grammatik

- die Struktur der korrekten Texte bestimmt,
- ist sie doch kein beliebig austauschbares Hilfsmittel bei der Definition einer Sprache.

Übersicht / Einordnung: Grammatik und Parsing

Grammatiken

- Grammatiken beschreiben Sprachen: Korrekte Texte und ihre Struktur
- Grammatiken definieren den Parsing-Prozess (Text => Ableitungsbaum)

Eine Grammatik-Definition hat zwei Ziele

Diese Ziele sind von einander weitgehend unabhängig und teilweise widersprüchlich:

- Erlaube es den Programmautoren das Gemeinte elegant und klar zum Ausdruck zu bringen
- Erlaube es dem Parser Programmtexte korrekt und mit vertretbarem Aufwand zu analysieren

Syntax und Abstrakte Syntax

Genau gesagt, sind wir auch mehr an der abstrakten, als der konkreten Syntax interessiert. Beispielsweise:

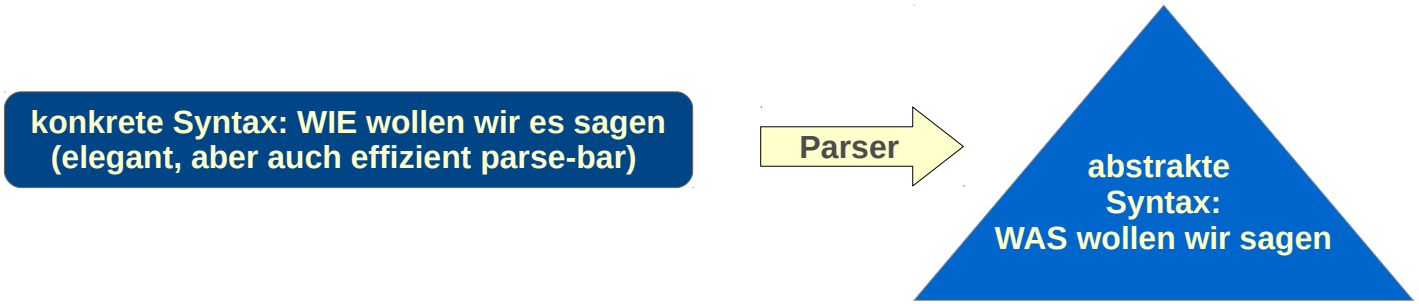
- Welches Zeichen genau für die Zuweisung gewählt wird, ist irrelevant
- Prioritäten und Assoziativitäten erlauben es uns mit weniger Klammer auszukommen, das dient der Bequemlichkeit, ist aber nicht essentiell für das was wie meinen.

Übersicht / Einordnung: Grammatik und Parsing

(Konkrete) Syntax und Abstrakte Syntax

Konkrete Syntax: Das Gesagte

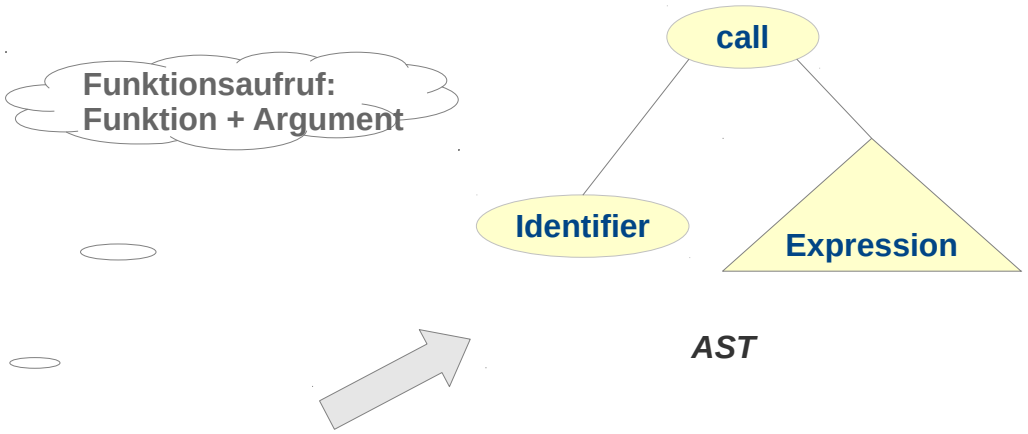
Abstrakte Syntax: Das Gemeinte



Übersicht / Einordnung: Grammatik und Parsing

Syntax und Abstrakte Syntax

Wir sind mehr an der abstrakten, als an der konkreten Syntax interessiert.



Das Wesentliche!

f(2+3)

Normal, aber eventuell schwer zu parsen

call f, 2+3;

Ungewohnt, aber leicht zu parsen

(f, 2+3)

Normal für Lisp- / Clojure-Fans, leicht zu parsen

Konkrete Syntax

Übersicht / Einordnung: Grammatik und Parsing

Eingeschränkte kontextfreie Grammatiken

- Für kontextfreie Grammatiken – im Gegensatz zu regulären Grammatiken – gibt es kein Standardverfahren zur Analyse / Erkennung
- Man unterscheidet Typen von kontextfreien Grammatiken und die für sie jeweils geeigneten Verfahren
- Parsing ist ein komplexer und aufwendiger Prozess, der vereinfacht werden kann, wenn die Grammatiken nur eine eingeschränkte Form haben
- Mit den eingeschränkten Formen von Grammatiken ist es unter Umständen schwieriger eine bestimmte Sprache zu definieren
- Oft definiert man eine Sprache mit einer uneingeschränkten Grammatik und transformiert sie dann in eine der eingeschränkten Formen

Praktisches Vorgehen

- Definiere abstrakte Syntax und Semantik der Sprache
- Wähle Parsing-Verfahren

  Beachte dabei Aufwand / typische Programmgröße / tolerierbare Laufzeit des Parsers ...
- Definiere die konkrete Syntax in einer Form, die zu dem gewählten Parsing-Verfahren passt. (Und zu den zukünftigen Programmautoren, den Nutzern der Sprache.)

Top-Down- / Bottom-up Verfahren

Parsing: Aufgabe / Ziel

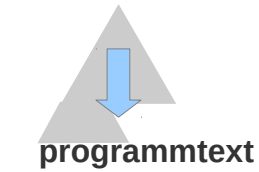
Finde den vorgelegten Text in der unendlichen Menge an Texten, die durch die Grammatiken definiert wird. Und:

Finde den (einen) Produktionsprozess, mit dem der Text erzeugt werden kann.

Top-Down-Parsing

Bei den Top-Down-Verfahren wird der Ableitungsbaum von der Wurzel (dem Startsymbol) aus aufgebaut. (Baumaufbau in *pre-order* Reihenfolge)

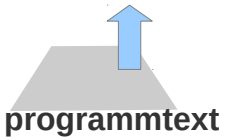
Ausgehend von der Wurzel werden Ableitungsbäume konstruiert und mit dem Text vergleichen.



Bottom-up-Parsing

Bei den Bottom-up-Verfahren wird der Ableitungsbaum von unten, vom Text aus, konstruiert. (Baumaufbau in *post-order* Reihenfolge)

Ausgehend von den Tokens (Symbolen) werden Produktionen gesucht, mit denen sie sich zu Nonterminalen „rückwärts ableiten“ lassen.



Beispiel: Parser für vollständig geklammerte Ausdrücke

Der Parser für die einfache Ausdruckssprache ist ein Top-Down-Parser.

Eine Expression ist eine Number oder eine Operation. Eine Number beginnt mit einer Ziffer, ...

Start mit der Annahme: Es handelt sich um eine Produktion für das Startsymbol

```
def parse(text: String): ExpTree = {
  var pos: Int = 0

  def parseExp: ExpTree = text(pos) match {
    case '1' => parseNumber
    . . .
    case '(' => parseOperation
    case _ => throw new IllegalArgumentException
  }

  def parseNumber: Number = ...

  def parseOperation: Operation = {
    pos = pos+1 // skip '('
    val exp1 = parseExp
    val opSymbol = text(pos)
    pos = pos+1 // skip opSign
    val exp2 = parseExp
    pos = pos+1 // skip ')'
    opSymbol match {
      case '+' => Operation(exp1, Operator(opSymbol), exp2)
      case '-' => Operation(exp1, Operator(opSymbol), exp2)
      case '*' => Operation(exp1, Operator(opSymbol), exp2)
      case '/' => Operation(exp1, Operator(opSymbol), exp2)
      case _ => throw new IllegalArgumentException
    }
  }
  parseExp
}
```

Expression → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Expression Operator Expression*)
Operator → + | – | * | /
Startsymbol: *Expression*

Rekursiver Abstieg (*Recursive Descent*)

Das Verfahren

von oben wird rekursiver Abstieg genannt

Es kann allgemein eingesetzt werden:

- Für jedes Nonterminal generiere eine Funktion
- Vorschau: Die Funktion testet das aktuelle Zeichen der Eingabe (oder allgemein mehrere Zeichen)

  entscheidet dann, welche Produktion angewendet wurde, um den Text zu generieren, der mit diesem Zeichen beginnt und
  - prüft / vergleicht jedes Terminal der Produktion mit der aktuellen Textposition
  - ruft für jedes Nonterminal (rekursiv) die entsprechende Funktion auf

Voraussetzungen / Probleme

- Die Grammatik muss gewährleisten, dass
  - für jedes Nonterminal an Hand des (oder allgemeiner der) ersten Zeichen(s) eines Texts entschieden werden kann, mit welcher Produktion der Text aus dem Nonterminal abgeleitet wurde.
- Linksrekursion in der Grammatik für zu Endlos-Rekursion im Parsing.

  Linksrekursion macht die Grammatik also für jedes Top-Down-Verfahren ungeeignet.

Vorteil

Das Verfahren des rekursiven Abstiegs ist
- Sehr einfach implementierbar und
- sehr effizient: es handelt sich um ein Top-Down-Verfahren ohne Backtracking

Rekursiver Abstieg (*Recursive Descent*)

Für jedes Nichtterminalsymbol der Grammatik wird eine Prozedur definiert. Wenn

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

alle Regeln mit A auf der linken Seite sind, sieht die Prozedur A in Pseudocode folgendermaßen aus:

```
proc A()
  if lookahead in first({α1}follow(A)) then
    C1
  else if lookahead in first({α2}follow(A)) then
    C2
  ...
  else if lookahead in first({αn}follow(A)) then
    Cn
  else
    error
```

http://de.wikipedia.org/wiki/Rekursiver_Abstieg

Top-Down-Parsing, Rekursiver Abstieg und Backtracking

Top-Down-Parsing

- Erfordert Grammatiken ohne Linksrekursion
- Die Top-Down-Verfahren benötigen i.A. Backtrack-Algorithmen
- Backtracking ist unnötig, wenn an Hand des / der nächsten Text-Symbole entschieden werden kann, welche Produktion verwendet wurde.

Rekursiver Abstieg

- Spezielle Strategie zur Implementierung von Backtrack-freien Top-Down-Parsern mit beliebiger Vorausschau (beliebig viele nächste Symbole können in Betracht gezogen werden).
- Achtung: Der Begriff „rekursiver Abstieg“ wird gelegentlich für alle Parser verwendet, die mit einer rekursiven Funktion pro Nonterminal implementiert sind, also auch für solche mit Backtracking.

Prädikative Grammatik – Prädikativer Parser

- Eine Grammatik wird prädikativ genannt, wenn sie mit einem Top-Down-Parser ohne Backtracking geparst werden kann.
- Bei einer prädikativen Grammatik kann die richtige Produktion aus dem aktuellen Zeichen geschlossen werden.
- Diese Intuition kann präzisiert werden: First- und Follow-Mengen.

Arithmetische Ausdrücke parsen 1: Grammatiktransformation

Grammatik: eindeutig, korrekte Präzedenzen und Assoziativitäten

Eine Grammatik mit den üblichen Präzedenzen und Assoziativitäten für arithmetische Ausdrücke:

Exp	→	*Exp AddOp* Term | *Term*
Term	→	*Term MultOp* Faktor | *Faktor*
Faktor	→	*Number* | '(' *Exp* ')'
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Diese Grammatik

- **ist eindeutig**
- **Assoziativität korrekt: die Operatoren sind linksassoziativ (Linksrekursion in der Grammatik)**
- **Präzedenzen korrekt: Punktrechnung vor Strichrechnung (Term „weiter oben“ als Faktor)**

Allerdings: Sie sie enthält Linksrekursionen und damit ist ungeeignet für Top-Down Parsing

Arithmetische Ausdrücke parsen 1: Grammatiktransformation

Grammatiktransformation: Elimination der Linksrekursion

Um ein Parsing nach (einem bestimmten Verfahren) zu ermöglichen, kann die Grammatik transformiert werden: Die Grammatik wird in eine äquivalente aber leichter parsbare Grammatik transformiert.

Exp	→	*Exp AddOp* Term	*Term*
Term	→	*Term MultOp* Faktor	*Faktor*
Faktor	→	*Number*	'(' *Exp* ')'
AddOp	→	'+'	'-'
MultOp	→	'*'	'/'
Number	→	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'0'

Die Grammatik ist nicht geeignet für Top-Down-Verfahren allgemein und damit auch nicht für einen rekursiven Abstieg: Expression und Term haben Produktionen mit direkter Linksrekursion.

Expr	→	*Term* { *Expr1* }
Expr1	→	*AddOp Term*
Term	→	*Faktor* { *Term1* }
Term1	→	*MultOp Faktor*
Faktor	→	*Number*	'(' *Expr* ')'
AddOp	→	'+'	'-'
MultOp	→	'*'	'/'
Number	→	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'0'

Eine äquivalente Grammatik (in EBNF-Notation) ohne Linksrekursion. (Statt dessen mit Rechtsrekursion.) Diese Grammatik ist geeignet für Top-Down-Parsing.

Arithmetische Ausdrücke parsen 1: Grammatiktransformation

Erinnerung: Bedeutung der EBNF-Metazeichen

- **{ } bedeuted Liste**
- **[] bedeutet optional**

Exp	→	*Term* { *Expr1* }
Expr1	→	*AddOp Term*
Term	→	*Faktor* { *Term1* }
Term1	→	*MultOp Faktor*
Faktor	→	*Number* | '(' *Exp* ')'
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

ist äquivalent zu (Expansion der Meta-Zeichen {…}):

Exp	→	*Term TermList*
TermList	→	*ε* | *Expr1 TermList*
Expr1	→	*AddOp Term*
Term	→	*Faktor FactorList*
FactorList	→	*ε* | *Term1 FactorList*
Term1	→	*MultOp Faktor*
Faktor	→	*Number* | '(' *Exp* ')'
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Konkrete Syntax

Konkrete Syntax vs Abstrakte Syntax

- **Konkrete Syntax**
  Textuelle Struktur mit allen Informationen die für das korrekte Parsen notwendig sind
- **Abstrakte Syntax**
  Auf das wesentliche reduzierte Form der syntaktischen Struktur

Ableitungsbaum vs Baum der abstrakten Syntax *(abstract syntax tree)*

- **Ableitungsbaum: Datenstruktur die die konkrete Syntax repräsentiert**
- **AST: Datenstruktur die die abstrakte Syntax repräsentiert**

Parser

- **Produziert i.A. eines AST**
- **Hier wollen wir aber erst einmal einen Syntaxbaum „er-parsen“**

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Konkrete Syntax

Konkrete Syntax (2-stufig)

Datenstruktur der Ableitungsbäume der Grammatik

```
Exp     → Term { Expr1 }
Expr1   → AddOp Term
Term    → Faktor { Term1 }
Term1   → MultOp Faktor
Faktor  → Number | '(' Exp ')'
AddOp   → '+' | '-'
MultOp  → '*' | '/'
Number  → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'
```

```
object Tokens {
  sealed abstract class Token
  case class  NumberToken(chars: String)  extends Token
  case class  LeftPToken(chars: String)   extends Token
  case class  RightPToken(chars: String)  extends Token
  case class  AddOpToken(chars: String)   extends Token
  case class  MultOpToken(chars: String)  extends Token
  case object EOFToken                    extends Token
  case class  ErrorToken(msg: String)     extends Token
}
```

```
object ParseTree {
  import Tokens._

  case class Exp (term: Term, exp1Lst: List[Exp1])
  case class Exp1 (addOp: AddOp, term: Term)
  case class AddOp(addOpToken: AddOpToken)

  case class Term(factor: Factor, term1List: List[Term1])
  case class Term1(multOp: MultOp, factor: Factor)
  case class MultOp(multOpToken: MultOpToken)

  sealed abstract class Factor
  case class NumberFactor(numberToken: NumberToken) extends Factor
  case class ParenthesisFactor(lpToken: LeftPToken, exp: Exp, rpToken: RightPToken) extends Factor
}
```

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Rekursiv absteigender Parser

Konzept

Eine Funktion pro syntaktischer Kategorie

```
Exp     → Term { Expr1 }
Expr1   → AddOp Term
Term    → Faktor { Term1 }
Term1   → MultOp Faktor
Faktor  → Number | '(' Exp ')'
AddOp   → '+' | '-'
MultOp  → '*' | '/'
Number  → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'
```

```
class Parser(scanner: Scanner) {

  import ParseTree._

  def parseExp(): Exp = ???

  def parseExp1(): Exp1 = ???

  def parseTerm(): Term = ???

  def parseTerm1(): Term1 = ???

  def parseFactor(): Factor = ???

}
```

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Rekursiv absteigender Parser

Konzept

Eine Funktion pro syntaktischer Kategorie

Unterscheidung der Produktionen an Hand der First- und Follow-Mengen

```
class Parser(scanner: Scanner) {

  import ParseTree._

  def parseExp(): Exp = ???

  def parseExp1(): Exp1 = ???

  def parseTerm(): Term = ???

  def parseTerm1(): Term1 = ???

  def parseFactor(): Factor = ???

}
```

Exp → *Term { Expr1 }*
Expr1 → *AddOp Term*
Term → *Faktor { Term1 }*
Term1 → *MultOp Faktor*
Faktor → *Number | '(' Exp ')'*
AddOp → '+' | '-'
MultOp → '*' | '/'
Number → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Prima: Nur ein Nonterminal mit alternativen Produktionen: Faktor! Und die Alternativen sind auch noch super leicht zu unterscheiden!

Für jedes Nichtterminalsymbol der Grammatik wird eine Prozedur definiert. Wenn

$$A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

alle Regeln mit A auf der linken Seite sind, sieht die Prozedur A in Pseudocode folgendermaßen aus:

```
proc A()
  if lookahead in first({α1}follow(A)) then
    C1
  else if lookahead in first({α2}follow(A)) then
    C2
  ...
  else if lookahead in first({αn}follow(A)) then
    Cn
  else
    error
```

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Rekursiv absteigender Parser

Konzept

Eine Funktion pro syntaktischer Kategorie

Unterscheidung der Produktionen an Hand der First- und Follow-Mengen

```
class Parser(scanner: Scanner) {

  import ParseTree._

  def parseExp(): Exp = ???

  def parseExp1(): Exp1 = ???

  def parseTerm(): Term = ???

  def parseTerm1(): Term1 = ???

  def parseFactor(): Factor = ???

}
```

Exp → *Term { Expr1 }*
Expr1 → *AddOp Term*
Term → *Faktor { Term1 }*
Term1 → *MultOp Faktor*
Faktor → *Number | '(' Exp ')'*
AddOp → '+' | '-'
MultOp → '*' | '/'
Number → '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Leider falsch: Die Listenkonstruktionen sind ebenfalls „versteckte“ Produktionen mit Alternativen

Für jedes Nichtterminalsymbol der Grammatik wird eine Prozedur definiert. Wenn

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$$

alle Regeln mit A auf der linken Seite sind, sieht die Prozedur A in Pseudocode folgendermaßen aus:

```
proc A()
  if lookahead in first({α1}follow(A)) then
    C1
  else if lookahead in first({α2}follow(A)) then
    C2
  ...
  else if lookahead in first({αn}follow(A)) then
    Cn
  else
    error
```

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Rekursiv absteigender Parser

Konzept

Die Listenkonstruktionen müssen beachtet werden:

Sie liefern weitere Alternativen: Eine Liste ist leer oder nicht leer.

Exp	→	*Term { Exp1 }*
Exp1	→	*AddOp Term*
Term	→	*Faktor { Term1 }*
Term1	→	*MultOp Faktor*
Faktor	→	*Number | '(' Exp ')'*
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Exp	→	*Term TermList*
TermList	→	*ε | Expr1 TermList*
Exp	→	*AddOp Term*
Term	→	*Faktor FactorList*
FactorList	→	*ε | Term1 FactorList*
Term1	→	*MultOp Faktor*
Faktor	→	*Number | '(' Exp ')'*
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

First- und Follow-Mengen

Bei einem Top-down-Parser muss aus dem (oder den) nächsten Symbol(-en) geschlossen werden, welche Produktion weiter zu verfolgen ist.

First- und Follow-Mengen sind ein Formalismus mit dem entsprechende Entscheidungen getroffen werden können.

Beispiel oben:

Exp	→	*Term TermList*
TermList	→	*ε* | *Expr1 TermList*
Expr1	→	*AddOp Term*
Term	→	*Faktor FactorList*
FactorList	→	*ε* | *Term1 FactorList*
Term1	→	*MultOp Faktor*
Faktor	→	*Number* | '(' *Exp* ')'
AddOp	→	'+' | '-'
MultOp	→	'*' | '/'
Number	→	'1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | '0'

Hier ε Produktion oder *Exp1 TermList* ? :

Bei einer ε Produktion muss das nächste Zeichen ein Zeichen sein, das *TermList* folgen kann. *TermList* ist das Ende von *Exp*. Nach *Exp* kann nur ein) kommen.

Bei einer *Exp1 TermList* Produktion muss das erste Zeichen ein erstes Zeichen von *Expr1* sein. Das kann nur ein *AddOp* sein.

Nachdenken über First- und Follow-Mengen.

First- und Follow-Mengen

Unterscheidung der Produktionen in der Beispiel-Grammatik

Exp	→	*Term TermList*
TermList	→	*ε | Expr1 TermList*
Exp1	→	*AddOp Term*
Term	→	*Faktor FactorList*
FactorList	→	*ε | Term1 FactorList*
Term1	→	*MultOp Faktor*
Faktor	→	*Number | '(' Exp ')'*

Leere Produktion (Liste zu Ende), oder nicht-leere Produktion (Liste geht weiter): Unterschied an AddOp eindeutig erkennbar.

Leere Produktion (Liste zu Ende), oder nicht-leere Produktion (Liste geht weiter): Unterschied an MultOp eindeutig erkennbar.

Leicht an den ersten Zeichen zu unterscheiden.

Exp	→	*Term { Exp1 }*
Exp1	→	*AddOp Term*
Term	→	*Faktor { Term1 }*
Term1	→	*MultOp Faktor*
Faktor	→	*Number | '(' Exp ')'*

Allgemeine Strategie zum parsen einer Liste: Parse Exp1 / Term1 solange noch ein AddOp / MultOp kommt.

First- und Follow-Mengen: Formale Definition

First-Mengen von Symbolen: First(X)

Für jedes Symbol X einer Grammatik bestimmt die Menge First(X) die Menge der Terminale mit der eine Produktion von X beginnen kann.

Bestimme zu jedem Grammatik-Symbol $X \in (T \cup N)$ die Menge FIRST(X) (Menge aller Terminalzeichen, mit denen aus X abgeleitete Wörter beginnen können; enthält ε, falls ε aus X ableitbar)

1. Falls X $\in$ T, FIRST(X)=X
2. Falls $(X \to \varepsilon) \in P$, übernehme ε in FIRST(X)
3. Für jede Regel $(X \to Y_1 \ldots Y_k) \in P$ verfahre wie folgt:

   Falls $\varepsilon \in FIRST(Y_1) \wedge \cdots \wedge \varepsilon \in FIRST(Y_{i-1})$ für ein i, $1 \leq i < k$, übernehme alle Terminalzeichen aus FIRST(Y_i) in FIRST(X)

   Falls $\varepsilon \in FIRST(Y_1) \wedge \cdots \wedge \varepsilon \in FIRST(Y_k)$ übernehme ε in FIRST(X)

Quelle: M. Jäger, *Compilerbau*

First- und Follow-Mengen: Formale Definition

First-Mengen von Produktionen: First(X_1 X_2 … X_n)

Die First-Mengen von Symbolen interessiert uns aber eigentlich nicht so sehr. Wir wollen an Hand eines Symbols zwischen Produktionen unterscheiden können.

Der Begriff der First-Menge wird darum auf Produktionen (Satzformen) ausgedehnt.

Diese Ausdehnung ist offensichtlich. Für eine Produktion X_1 X_2 … X_n ist

- First(X_1 X_2 … X_n) = First(X_1)

  es sei denn X_1 ist ein Nonterminal, das zu ε abgeleitet werden kann, wenn ja, dann
- First(X_1 X_2 … X_n) = First(X_1) ∪ First(X_2)

  es sei denn X_2 ist ein Nonterminal, das zu ε abgeleitet werden kann, wenn ja, dann
- ...

First- und Follow-Mengen

Mit den First-Mengen kann ein Backtracking-freier Top-Down-Parser konstruiert werden.

Beispiel:

Expr1	→	*AddOp Term* | ε
AddOp	→	+ | –

Grammatik in EBNF

Expr1	→	*AddOp Term Expr1*
Expr1	→	ε
AddOp	→	+
AddOp	→	–

Grammatik in Basis-Form

Trifft der (Top-Down-) Parser auf Expr1, dann muss er zwischen den Produktionen

Expr1 → *AddOp Term*
Expr1 → ε

unterscheiden. Die beiden Produktionen kann man an Hand der First-Mengen ihres jeweils ersten Symbols unterscheiden.

First(*AddOp Term*) = First(*AddOp*) = { +, – }
First(ε) = { ε }

Wenn der Parser auf + oder – trifft, dann ist die erste Produktion zu wählen, ansonsten ist die Produktion zu Ende und die zweite, die ε-Produktion, ist zu wählen.

First- und Follow-Mengen

Trifft der Parser auf ein Zeichen,

- **das zu keiner First-Menge gehört**
- **und ist eine der Produktionen die ε-Produktion**

dann muss die ε-Produktion gewählt werden.

Um korrekte von nicht-korrekten Texten unterscheiden zu können, ist es sinnvoll zu testen, ob das nächste Zeichen tatsächlich nach einer ε-Produktion auftreten kann.

Die entsprechende Menge wird Follow-Menge genannt

Beispiel:

Expr1	→	*AddOp Term Expr1*
Expr1	→	*ε*
AddOp	→	+
AddOp	→	–

Follow(Expr1) = Menge aller Terminale die unmittelbar nach einer Expr1-Produktion folgen kann.

In der Grammatik kann kein Zeichen auf Exp1 folgen, d.h. Follow(Expr1) = { eof }
Wenn also nach einer Exp1-Produktion der Text nicht zu Ende ist, dann ist die Eingabe fehlerhaft.

First- und Follow-Mengen

First⁺ – Erweiterte Definition von First: Erstes oder nächstes Zeichen

Mit Hilfe der First- und Follow-Mengen kann definiert werden, wann eine Grammatik *Backtrack-frei* ist, d.h. ohne Backtracking top-down geparst werden kann.

$$\text{First}^+(A \rightarrow \alpha) = \begin{cases} \text{First}(\alpha) & \textit{falls } \varepsilon \notin \text{First}(\alpha) \\ \text{First}(\alpha) \cup \text{Follow}(A) & \textit{sonst} \end{cases}$$

Eine Grammatik ist ist Backtrack-frei, wenn für jedes Nonterminal A mit den Produktionen

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \alpha_n$$

gilt:

$$\text{First}^+(A \rightarrow \alpha_i) \cap \text{First}^+(A \rightarrow \alpha_j) = \varnothing \quad \text{Für alle } j \neq j \in \{ 1, \ldots n \}$$

Alle Produktionen eines Nonterminals A können an ihren ersten Zeichen, bzw. am ersten Zeichen nach A unterschieden werden.

Arithmetische Ausdrücke parsen 2: Konkrete Syntax parsen

Rekursiv absteigender Parser – 1

erzeugt Ableitungsbaum

Exp → *Term { Expr1 }*
Expr1 → *AddOp Term*
Term → *Faktor { Term1 }*
Term1 → *MultOp Faktor*
Faktor → *Number | '(' Exp ')'*

```
class Parser(scanner: Scanner) {
  import Tokens._
  import ParseTree._

  def parseExp1List(): List[Exp1] = {
    scanner.lookahead() match {
      case EOFToken      => Nil
      case t@AddOpToken(_) =>
        scanner.move()
        Exp1(AddOp(t), parseTerm()) :: parseExp1List()
      case _ => Nil
    }
  }

  def parseTerm1List(): List[Term1] = {
    scanner.lookahead() match {
      case EOFToken      => Nil
      case t@MultOpToken(_) =>
        scanner.move()
        Term1(MultOp(t), parseFactor()) :: parseTerm1List()
      case _ => Nil
    }
  }

  def parseExp(): Exp = {
    val term: Term = parseTerm()
    val exp1List: List[Exp1] = parseExp1List()
    Exp(term, exp1List)
  }
```

```
class Scanner(input: String) {

  ...
  private var actToken: Token = nextToken()

  def lookahead(): Token = actToken

  def move(): Unit =
    actToken = nextToken()

}
```

Scanner mit einem Zeichen Vorschau

Rekursiv absteigender Parser – 2

erzeugt Ableitungsbaum

Exp → *Term* { *Expr1* }
Expr1 → *AddOp Term*
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
def parseExp1(): Exp1 = {
  scanner.lookahead() match {
    case ao@AddOpToken(_) =>
      scanner.move()
      Exp1(AddOp(ao), parseTerm())
    case x => throw new Exception(s"Error when parsing Exp1: expected addop found $x")
  }
}

def parseTerm(): Term = {
  val factor: Factor = parseFactor()
  val term1List: List[Term1] = parseTerm1List()
  Term(factor, term1List)
}

def parseTerm1(): Term1 = {
  scanner.lookahead() match {
    case mo@MultOpToken(_) =>
      scanner.move()
      Term1(MultOp(mo), parseFactor())
    case x => throw new Exception(s"Error when parsing Tem1: expected multop found $x")
  }
}
```

Rekursiv absteigender Parser – 3

erzeugt Ableitungsbaum

Exp → *Term* { *Expr1* }
Expr1 → *AddOp Term*
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
  def parseFactor(): Factor =
     scanner.lookahead() match {
       case nt@NumberToken(_) =>
         scanner.move()
         NumberFactor(nt)
       case lp@LeftPToken(_) =>
         scanner.move()
         val exp: Exp = parseExp()
         scanner.lookahead() match {
           case rp@RightPToken(_) =>
             scanner.move()
             ParenthesisFactor(lp, exp, rp)
           case x => throw new Exception(s"Error when parsing Factor: expected ) found $x")
         }
       case x => throw new Exception(s"Error when parsing Factor: expected number or ( found $x")
     }

}
```

Rekursiv absteigender Parser – 3

Exp → *Term* { *Expr1* }
Expr1 → *AddOp Term*
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

erzeugt Ableitungsbaum

Beispiel:

2 + 3 * 4

```
val text = scala.io.StdIn.readLine()
val scanner = new Scanner(text)
val tree = new
Parser(scanner).parseExp()
println(tree)
```

Exp(Term(NumberFactor(NumberToken(2)),List()),
List(Exp1(AddOp(AddOpToken(+)), Term(NumberFactor(NumberToken(3)),
List(Term1(MultOp(MultOpToken(*)),NumberFactor(NumberToken(4))))
))))

Der Ableitungsbaum wird korrekt erzeugt, aber wer will ihn wirklich sehen?

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

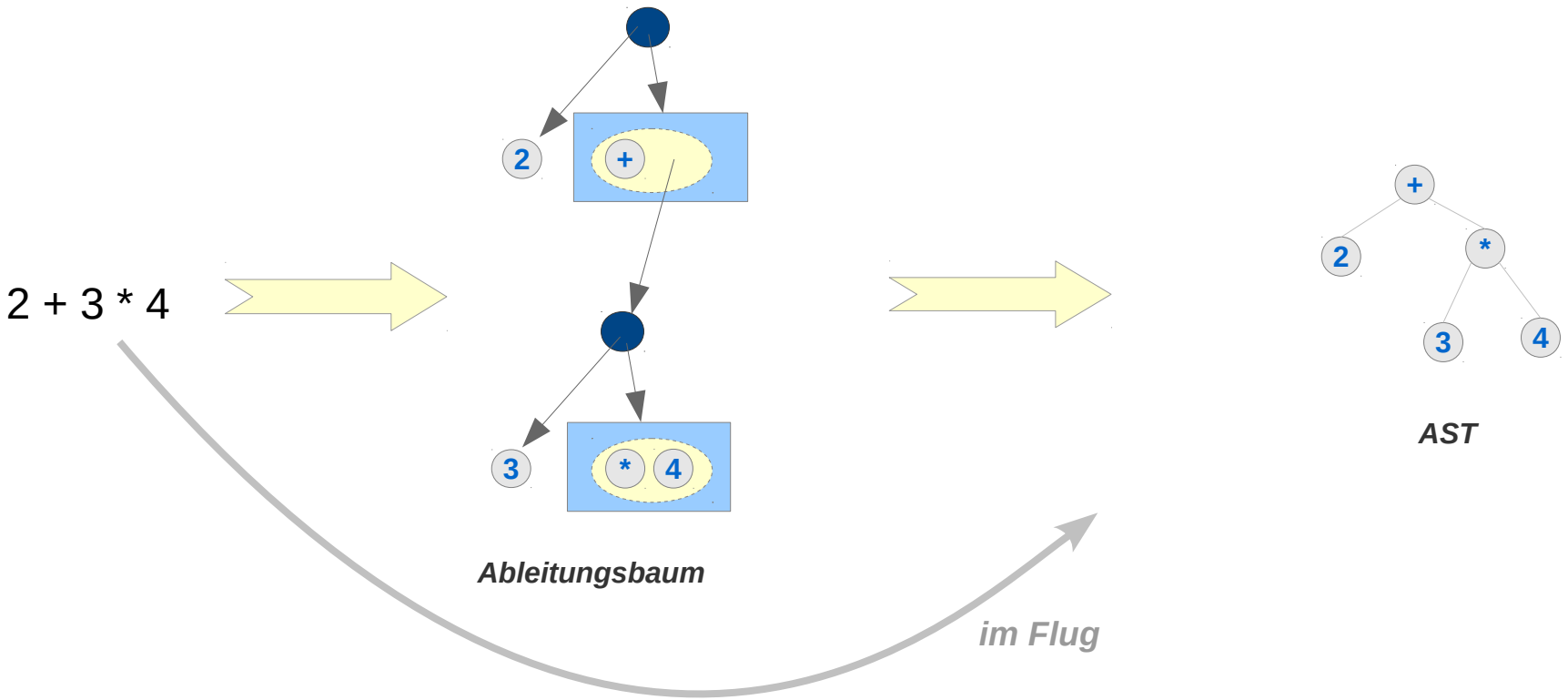
Transformation des Ableitungsbaums zum AST

Wir sind für die weitere Verarbeitung nicht am Ableitungsbaum interessiert. Er enthält zu viele Informationen, die nur für das korrekte und eindeutige Parsing interessant waren.

Gesucht:

- **Ableitungsbaum ~> AST**
- **am Besten „im Flug“, also während des Parsens**

Beispiel:



Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST

Eine Reduktion des Ableitungsbaums auf das Wesentliche

```
class AST {
  sealed abstract class Exp
  case class Number(d: Int) extends Exp
  case class Add(e1: Exp, e2: Exp) extends Exp
  case class Sub(e1: Exp, e2: Exp) extends Exp
  case class Div(e1: Exp, e2: Exp) extends Exp
  case class Mul(e1: Exp, e2: Exp) extends Exp
}
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen

Einige Funktionen können problemlos umgeschrieben werden
es werden nur Informationen weg gelassen

z.B. parseFactor

```
def parseFactor(): Factor =
    scanner.lookahead() match {
      case nt@NumberToken(_) =>
        scanner.move()
        NumberFactor(nt)
      case lp@LeftPToken(_) =>
        scanner.move()
        val exp: Exp = parseExp()
        scanner.lookahead() match {
          case rp@RightPToken(_) =>
            scanner.move()
            ParenthesisFactor(lp, exp, rp)
          case x => throw new Exception(s"Error ...")
        }
      case x => throw new Exception(s"Error ...")
    }
```

```
def parseFactor(): Exp =
  scanner.lookahead() match {
    case NumberToken(v) =>
      scanner.move()
      Number(v.toInt)
    case lp@LeftPToken(_) =>
      scanner.move()
      val exp: Exp = parseExp()
      scanner.lookahead() match {
        case rp@RightPToken(_) =>
          scanner.move()
          exp
        case x => throw new Exception(s"Error when parsing Factor: expected ) found $x")
      }
    case x => throw new Exception(s"Error when parsing Factor: expected number or ( found $x")
  }
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen

Bei anderen muss eine völlig andere Struktur erzeugt werden
Vor allem müssen statt der Listen die entsprechenden Bäume erzeugt werden

```
def parseExp1List(): List[Exp1] = {
  scanner.lookahead() match {
    case EOFToken         => Nil
    case t@AddOpToken(_) =>
      scanner.move()
      Exp1(AddOp(t), parseTerm()) :: parseExp1List()
    case _ => Nil
  }
}
```

```
def parseExp1List() = ???
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen

Listen zu Bäumen, Beispiel Exp1

Exp → *Term* { ***Expr1*** }
Expr1 → ***AddOp Term***
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
def parseExp1List(): List[(AddOpToken, Exp)] = ???
```

```
def parseExp1List(): List[Exp1] = {
  scanner.lookahead() match {
    case EOFToken         => Nil
    case t@AddOpToken(_) =>
      scanner.move()
      Exp1(AddOp(t), parseTerm()) :: parseExp1List()
    case _ => Nil
  }
}
```

Exp1 ist ein Paar aus AddOps und Termen.
Terme werden im AST nicht von anderen Ausdrücken unterschieden.

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen

Listen zu Bäumen, Beispiel Exp1

Exp → *Term* { ***Expr1*** }
Expr1 → ***AddOp Term***
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
def parseExp1List(): List[Exp1] = {
  scanner.lookahead() match {
    case EOFToken        => Nil
    case t@AddOpToken(_) =>
      scanner.move()
      Exp1(AddOp(t), parseTerm()) :: parseExp1List()
    case _ => Nil
  }
}
```

```
def parseExp1List(): List[(AddOpToken, Exp)] = {
  scanner.lookahead() match {
    case EOFToken      => Nil
    case t@AddOpToken(_) =>
      scanner.move()
      (t, parseTerm()) :: parseExp1List()
    case _ => Nil
  }
}
```

wird in parseExp benutzt. Die Nutzung muss noch angepasst werden.

```
def parseExp(): Exp = {
  val term: Exp = parseTerm()
  val exp1List: List[(AddOpToken, Exp)] = parseExp1List()
  ??? // Exp(term, exp1List)
}
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen

Listen zu Bäumen, Beispiel Exp1

Exp → *Term* { ***Expr1*** }
Expr1 → ***AddOp Term***
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
def parseExp1List(): List[(AddOpToken, Exp)] = {
  scanner.lookahead() match {
    case EOFToken      => Nil
    case t@AddOpToken(_) =>
      scanner.move()
      (t, parseTerm()) :: parseExp1List()
    case _ => Nil
  }
}
```

```
def parseExp(): Exp = {
  val term: Exp = parseTerm()
  val exp1List: List[(AddOpToken, Exp)] = parseExp1List()
  ??? // Exp(term, exp1List)
}
```

Die Liste wird zu einem Baum „aufgefaltet“.

```
def parseExp(): Exp = {
  val term: Exp = parseTerm()
  val exp1List: List[(AddOpToken, Exp)] = parseExp1List()
  exp1List.foldLeft(term)( (acc, x) => x match {
    case (AddOpToken("+"), t) =>     Add(acc, t)
    case (AddOpToken("-"), t)=>    Sub(acc, t)
  })
}
```

Die gleiche Transformation / Anpassung kann für die Term1-Listen durchgeführt werden.

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen – 1

Exp → *Term* { *Expr1* }
Expr1 → *AddOp Term*
Term → *Faktor* { *Term1* }
Term1 → *MultOp Faktor*
Faktor → *Number* | '(' *Exp* ')'

```
class ParserDerivationTree(scanner: Scanner) {
  import Tokens._
  import DerivationTree._

  def parseExp1List(): List[Exp1] = {
    scanner.lookahead() match {
      case EOFToken      => Nil
      case t@AddOpToken(_) =>
        scanner.move()
        Exp1(AddOp(t), parseTerm()) :: parseExp1List()
      case _ => Nil
    }
  }

  def parseTerm1List(): List[Term1] = {
    scanner.lookahead() match {
      case EOFToken      => Nil
      case t@MultOpToken(_) =>
        scanner.move()
        Term1(MultOp(t), parseFactor()) :: parseTerm1List()
      case _ => Nil
    }
  }

  def parseExp(): Exp = {
    val term: Term = parseTerm()
    val exp1List: List[Exp1] = parseExp1List()
    Exp(term, exp1List)
  }

  def parseExp1(): Exp1 = {
    scanner.lookahead() match {
      case ao@AddOpToken(_) =>
        scanner.move()
        Exp1(AddOp(ao), parseTerm())
      case x => throw new Exception(s"Error when parsing Exp1: expected addop found $x")
    }
  }
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

AST im Fluge erzeugen – 2

Exp → *Term { Expr1 }*
Expr1 → *AddOp Term*
Term → *Faktor { Term1 }*
Term1 → *MultOp Faktor*
Faktor → *Number | '(' Exp ')'*

```
  def parseTerm(): Term = {
     val factor: Factor = parseFactor()
     val term1List: List[Term1] = parseTerm1List()
     Term(factor, term1List)
   }

   def parseTerm1(): Term1 = {
     scanner.lookahead() match {
       case mo@MultOpToken(_) =>
         scanner.move()
         Term1(MultOp(mo), parseFactor())
       case x => throw new Exception(s"Error when parsing Tem1: expected multop found $x")
     }
   }

   def parseFactor(): Factor =
     scanner.lookahead() match {
       case nt@NumberToken(_) =>
         scanner.move()
         NumberFactor(nt)
       case lp@LeftPToken(_) =>
         scanner.move()
         val exp: Exp = parseExp()
         scanner.lookahead() match {
           case rp@RightPToken(_) =>
             scanner.move()
             ParenthesisFactor(lp, exp, rp)
           case x => throw new Exception(s"Error when parsing Factor: expected ) found $x")
         }
       case x => throw new Exception(s"Error when parsing Factor: expected number or ( found $x")
     }

}
```

***Hinweis**: In der Regel sind Ausdrücke die anspruchsvollsten Konstrukte, wenn es ums Parsen geht!*
Oder: Alles andere ist normalerweise einfacher.

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

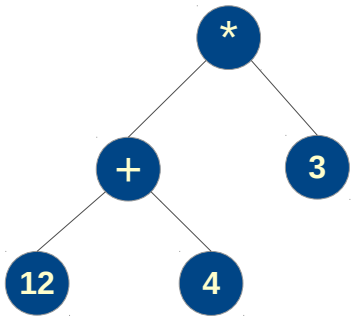
AST im Fluge erzeugen – 3

Exp → *Term { Expr1 }*
Expr1 → *AddOp Term*
Term → *Faktor { Term1 }*
Term1 → *MultOp Faktor*
Faktor → *Number | '(' Exp ')'*

(12 + 4) * 3

```
val text = scala.io.StdIn.readLine()
val scanner = new Scanner(text)
val tree = new ParserAst(scanner).parseExp()
println(tree)
```

Mul(Add(Number(12),Number(4)),Number(3))




```
object AST {
  sealed abstract class Exp
  case class Number(d: Int) extends Exp
  case class Add(e1: Exp, e2: Exp) extends Exp
  case class Sub(e1: Exp, e2: Exp) extends Exp
  case class Div(e1: Exp, e2: Exp) extends Exp
  case class Mul(e1: Exp, e2: Exp) extends Exp
}
```

Arithmetische Ausdrücke parsen 3: AST statt Ableitungsbaum

Zusammenfassung

Eine Grammatik die

- **eindeutig ist und**
- **die gewollten Präzedenzen und Assoziativitäten zum Ausdruck bringt, kann komplex werden und viele „semantisch irrelevante“ Konstrukte enthalten**

Um parsebar zu sein muss eine Grammatik transformiert werden.

- **Z.B. Links-Rekursionen müssen eliminiert werden, um Top-Down-Parsing möglich zu machen**
- **dabei werden entstehen typischerweise Listenkonstruktionen**

Die beim Parsen der so entstandenen (konkreten) Grammatik erzeugt Ableitungsbäume die

- **komplex sind und**
- **enthält viele „semantisch irrelevante“ Konstrukte**

Die „semantisch irrelevanten“ Konstrukte

- **müssen eliminiert werden:**
  AST statt Ableitungsbaum erzeugen
- **am besten direkt durch den Parser**

Prädikative Top-Down-Parser: LL(1)

Mit Hilfe der First$^+$ - Mengen kann entschieden werden, ob eine Grammatik für Backtrack-freies (prädikatives) Top-Down-Parsing geeignet ist.

Wenn ja, dann kann dieser Parser auf unterschiedliche Art implementiert werden. Gängig sind:

- **Recursive-Descent-Parser**
- **LL(1)-Parser**

LL(1)-Parser

Ein LL(1)-Parser basiert auf der Einsicht, dass

- **das nächste Eingabesymbol und**
- **die First$^+$-Menge,**

das Verhalten des Parsers in jedem Schritt bestimmen. Daraus kann ein tabellengesteuertes Verfahren generiert werden.

Der entsprechende Parser wird LL(1)-Parser genannt:

LL(1) :

- **Left-to-Right Verarbeitung der Eingabe;**
- **wobei eine Links-Ableitung erzeugt wird und**
- **1 Zeichnen Vorausschau notwendig ist**

LL(1)-Parser werden in der Regel aus einer passenden Grammatik mit einem Parser-Generator generiert.

Top-Down-Parsing

Ohne Backtracking (Prädikativ)

- Rekursiver Abstieg:
  - Eine Parsing-Funktion pro Nonterminal
  - Beliebige Zahl (meist und am besten 1) an Vorausschau-Zeichen
- LL(k): Tabellen-gesteuert, fixe Zahl k an Vorausschau-Zeichen

Mit Backtracking

– Rekursives Backtrack-Parsen
  - Eine Parsing-Funktion pro Nonterminal
  - Keine Vorausschau, Backtracking bei Failure

– Parser-Kombinatoren

  Systematische programmgesteuerte Konstruktion von rekursiven Backtrack-Parsern

Bottom-Up-Parser

Bottom-Up-Parser (auch: Shift-Reduce-Parser / LR-Parser)

Prinzip

- **Der Text wird schrittweise, Symbol für Symbol gelesen,**
- **Nach jedem Schritt wird geprüft, ob der gelesene Text zu einer Produktion passt,**
- **Wenn ja, wird sie durch das entsprechende Nonterminal ersetzt.**

Bei dem Verfahren wird ein Stack mit zwei Operationen genutzt:

- **Shift: Schiebe das nächste Eingabesymbol auf den Stack**
- **Reduce: Reduziere eine Sequenz von Symbolen auf dem Stack zu einem Nonterminal**

Bottom-Up-Parser werden hier nicht weiter behandelt.

- **Sie werden in der Regel von Parser-Generatoren erzeugt.**
- **Sie sind für anspruchsvolle Parsing-Aufgaben geeignet (schnelle Analyse komplexer Grammatiken.)**
- **und sollten Thema einer fortgeschritten Veranstaltung sein.**

Parsing – Zusammenfassung

- Syntax-Analyse ist das älteste und am besten erforschte Teilgebiet der theoretischen Informatik – aber immer noch ein hoch interessantes lebendiges Forschungsgebiet
- Die Ergebnisse werden in guten und praktischen Tools verwendet
  - Reguläre Ausdrücke als eingebettete Teilsprache vieler Programmiersprachen
  - Parser / Scanner
  - Parser- / Scanner-Generatoren
- „Richtige Programmiersprachen“ werden in der Regel von handgeschriebenen Varianten eines *Recursive-Descent*-Parsers geparst
- Für „kleine Sprachen“ eignen sich aber die Tools sehr gut
- Grammatiken definieren eine konkrete Syntax, deren Bedeutung ist groß, sollte aber auch nicht überschätzt werden
  - für die Weiterverarbeitung ist die abstrakte Syntax wesentlich interessanter
  - Programmiersprachen definieren sich in erster Linie über ihre abstrakte Syntax und deren Bedeutung
  - Die konkrete Syntax sollte intuitiv nutzbar sein und mit einem Blick auf den Anwender *und* den Parser (die Tools mit denen er erstellt wird) definiert werden.
  - Das gilt natürlich nicht, wenn die Syntax der Sprache vorliegt und und unverändert zu verarbeiten ist

Parser – Überblick

***Achtung**: Nicht jedes Verfahren kann jede Grammatik parsen.*