

Compiler: Parser

Prof. Dr. Oliver Braun

Letzte Änderung: 17.05.2017 11:06

Parsing

- ▶ Parsing ist die zweite Stufe im Compiler-Frontend
- ▶ Eingabe ist der Wörterstrom den der Scanner erzeugt
- ▶ der Parser versucht mit Hilfe eines grammatikalischen Modells eine syntaktische Struktur für das Programm herzuleiten
- ▶ wenn der Parser erkennt, dass die Eingabe ein gültiges Programm ist
 - ▶ erzeugt er ein konkretes Modell des Programms
- ▶ wenn die Eingabe nicht gültig ist, meldet er das Problem und die dazu gehörigen diagnostischen Informationen
- ▶ Parsing hat als Problemstellung viele Ähnlichkeiten mit dem Scanning
- ▶ die theoretische Grundlage für Parsertechnologien wurde sehr ausführlich als Teilgebiet der formalen Sprachtheorie untersucht

- ▶ wir benötigen eine Notation mit Hilfe derer wir die Syntax einer Sprache beschreiben und Programme dagegen prüfen können
- ▶ ein möglicher Kandidat sind reguläre Ausdrücke
- ▶ aber REs sind nicht mächtig genug die komplette Syntax zu beschreiben (Beispiel siehe folgende Folie)
- ▶ vielversprechender ist die kontextfreie Grammatik (*context-free grammar (CFG)*)

Warum keine regulären Ausdrücke?

- ▶ denken wir an das Problem arithmetische Ausdrücke mit Variablen und Operatoren zu erkennen

- ▶ der RE

$$[a\dots z]([a\dots z]|[0\dots 9])^*((+|-|\times|÷)[a\dots z]([a\dots z]|[0\dots 9])^*)^*$$

- ▶ enthält keinerlei Informationen über den Vorrang von Operatoren, z.B. bei $a + b \times c$
- ▶ wir können versuchen Klammern mit zu erkennen:
$$(\backslash|\epsilon)[a\dots z]([a\dots z]|[0\dots 9])^*((+|-|\times|÷)[a\dots z]([a\dots z]|[0\dots 9])^*)^*\backslash|\epsilon)$$
 - ▶ dieser RE kann ein Klammerpaar um einen Ausdruck erkennen, aber keine inneren Klammerpaare

Wirklich keine regulären Ausdrücke?

- ▶ nächster Versuch, Klammern im Abschluß:

$$\backslash(|\epsilon)[a\dots z]([a\dots z]|[0\dots 9])^*((+|-|\times|\div)[a\dots z]([a\dots z]|[0\dots 9])^*\backslash)|\epsilon))^*$$

- ▶ aber das würde $a + b) \times c)$ akzeptieren
- ▶ tatsächlich können wir keinen RE schreiben, der alle Ausdrücke mit korrekt angeordneten Klammerpaaren erkennen würde und die anderen nicht
- ▶ PCREs sind mächtiger, aber auch nicht ausdrucksstark genug

Kontextfreie Grammatik

- ▶ eine kontextfreie Grammatik G ist ein Quadrupel (T, NT, S, P) für das gilt
 - ▶ T ist die Menge von terminalen Symbolen oder Wörtern der Sprache $L(G)$. Terminale Symbole entsprechen den syntaktischen Kategorien die der Scanner ermittelt.
 - ▶ NT ist die Menge der nichtterminalen Symbolen die in den Produktionsregeln von G vorkommen. Nichtterminale Symbole sind syntaktische Variablen.
 - ▶ S ist ein nichtterminales Symbol das als Startsymbol dient.
 - ▶ P ist die Menge von Produktionsregeln von G . P hat die Form $NT \mapsto (T \cup NT)^+$

Backus-Naur-Form

- ▶ üblicherweise werden die Produktionsregeln einer CFG in Backus-Naur-Form (BNF) angegeben
- ▶ oft genutzt werden:
 - ▶ Extended BNF (EBNF) [ISO standard]
 - ▶ Augmented BNF (ABNF) [RFC]

Beispiele

- ▶ die Sprache der Schafe:

<i>SheepNoise</i>	\mapsto	baa <i>SheepNoise</i>
		baa

- ▶ eine Sprache von arithmetischen Ausdrücken mit Klammern

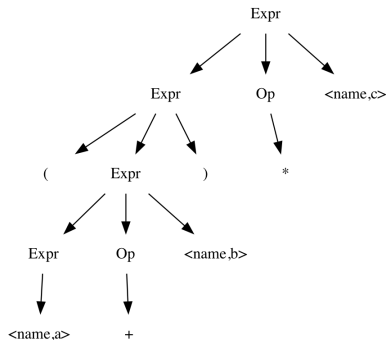
1	<i>Expr</i>	\mapsto	(<i>Expr</i>)
2			<i>Expr Op name</i>
3			name
4	<i>Op</i>	\mapsto	+
5			-
6			*
7			/

Arithmetische Ausdrücke in Haskell

siehe <https://github.com/ob-cs-hm-edu/compiler-ParserArithExpr>

Eine Beispiel-Herleitung

- ▶ mit dem Startsymbol *Expr* können wir den Satz $(a + b) * c$ mit einer rechtskanonischen Ableitung (*rightmost derivation*) herleiten, durch die Sequenz (2,6,1,2,4,3)



- ▶ die linkskanonische Ableitung (*leftmost derivation*) nutzt die Sequenz (2,1,2,3,4,6) und resultiert offensichtlich im selben Parsebaum

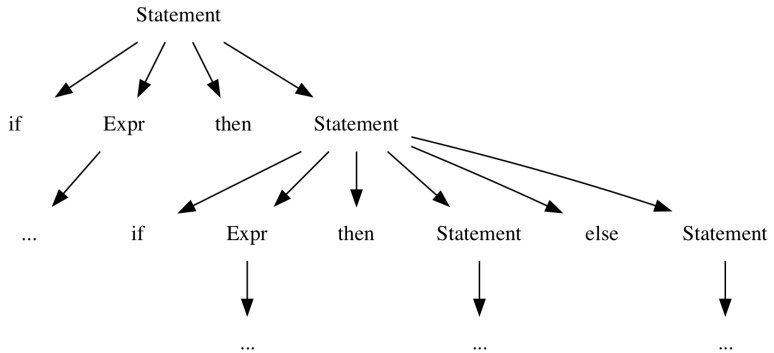
Mehrdeutige Grammatik

- ▶ für einen Compiler ist es wichtig, dass jeder Satz eine eindeutige (rechts- oder linkskanonische) Herleitung hat
- ▶ wenn es verschiedene Herleitungen für einen Satz gibt, heißt die Grammatik mehrdeutig
- ▶ eine solche Grammatik kann für einen Satz verschiedene Parsebäume erzeugen, die potentiell verschiedene Bedeutungen eines Programmes bedeuten können

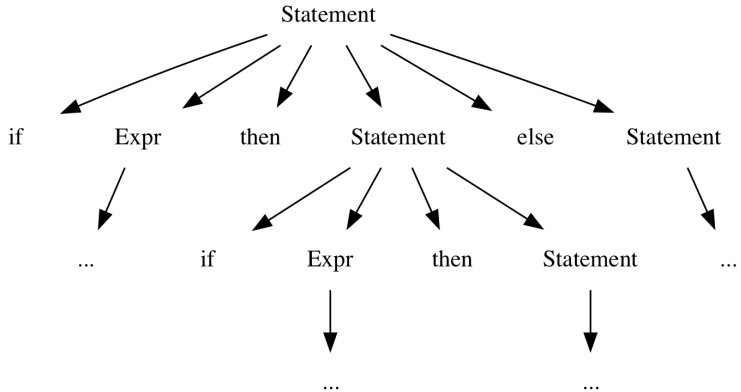
Der Klassiker für eine mehrdeutiges Konstrukt

1	<i>Statement</i>	\mapsto	if <i>Expr</i> then <i>Statement</i> else <i>Statement</i>
2			if <i>Expr</i> then <i>Statement</i>
3			<i>Assignment</i>
4			...other statements...

.. kann in diesen Parsebaum resultieren:



.. oder in diesen

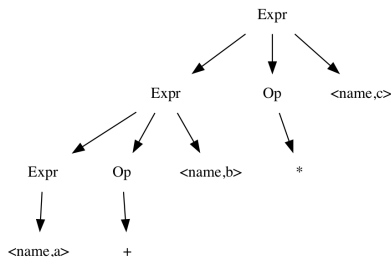


else ohne Mehrdeutigkeit

1	<i>Statement</i>	\mapsto	if <i>Expr</i> then <i>Statement</i>
2			if <i>Expr</i> then <i>WithElse</i> else <i>Statement</i>
3			<i>Assignment</i>
4			...other statements...
5	<i>WithElse</i>	\mapsto	if <i>Expr</i> then <i>WithElse</i> else <i>WithElse</i>
6			<i>Assignment</i>

Bedeutung in Struktur kodieren

- ▶ Parsen von $a + b * c$ resultiert mit den o.a. Regeln in



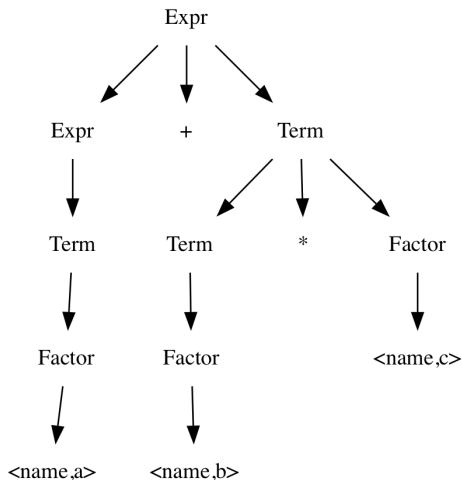
- ▶ ein naheliegender Ansatz den Ausdruck auszuwerten ist den Baum “Postorder” zu durchlaufen
- ▶ aber das resultiert in $(a + b) * c$ und nicht $a + b * c$, da wir bislang keinen Vorrang der Operatoren in der Grammatik formuliert haben

Operator Vorrang hinzufügen

0	<i>Goal</i>	\mapsto	<i>Expr</i>
1	<i>Expr</i>	\mapsto	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	\mapsto	<i>Term</i> * <i>Factor</i>
5			<i>Term</i> / <i>Factor</i>
6			<i>Factor</i>
7	<i>Factor</i>	\mapsto	(<i>Expr</i>)
8			num
9			name

Parsen mit den neuen Regeln

Durch die Sequenz (0,1,4,6,9,9,3,6,9) bekommen wir jetzt den Parsebaum:



Top-Down Parsing

- ▶ ein Top-Down Parser beginnt an der Wurzel des Parsebaumes und erweitert ihn systematisch nach unten
- ▶ er wählt Nichtterminale am unteren Rand des Baumes aus und erweitert ihn indem er Kindknoten hinzufügt die den Regeln entsprechen
- ▶ der Prozess wird solange fortgesetzt bis entweder
 1. der untere Rand des Baumes nur terminale Symbole enthält **und** der Eingabestrom zuende ist, oder
 2. ein klarer Mismatch zwischen dem unteren Rand und dem Eingabestrom entstanden ist.
- ▶ im ersten Fall war der Parser erfolgreich
- ▶ im zweiten Fall
 - ▶ könnte der Parser in einem früheren Schritt eine falsche Regel ausgewählt haben. Er kann durch *Backtracking* dort hin zurück und mit einer anderen Regel weiter machen
 - ▶ wenn das Backtracking auch nicht zum Erfolg geführt hat, ist die Eingabe ungültig

Top-Down Parsing ...

- ▶ Top-Down Parsing ist für eine große Teilmenge der CFGs, die ohne Backtracking auskommt, effizient
- ▶ es gibt Transformationen die *in vielen Fällen* eine beliebige Grammatik in eine Grammatik umwandeln kann, die ohne Backtracking aus kommt
- ▶ es gibt zwei verschiedene Ansätze um Top-Down Parser zu bauen:
 1. Handgeschriebene rekursiv-absteigende Parser (*hand-coded recursive-descent parsers*), und
 2. generierte LL(1) Parser

Linksrekursion eliminieren

- ▶ eine linkskanonischer Top-Down Parser kann in eine Endlosschleife geraten, wenn die Grammatik *Linksrekursion* enthält, z.B.

$$\frac{}{Fee \mapsto Fee \alpha}$$
$$\frac{}{Fee \mapsto Fee \beta}$$

- ▶ diese können wir aber einfach eliminieren durch beispielsweise

$$\frac{}{Fee \mapsto \beta Fee'}$$
$$\frac{}{Fee' \mapsto \alpha Fee'}$$
$$\frac{}{Fee' \mapsto \epsilon}$$

Aufgabe

Eliminieren Sie die Linksrekursion

0	<i>Goal</i>	\mapsto	<i>Expr</i>
1	<i>Expr</i>	\mapsto	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	\mapsto	<i>Term</i> * <i>Factor</i>
5			<i>Term</i> / <i>Factor</i>
6			<i>Factor</i>
7	<i>Factor</i>	\mapsto	(<i>Expr</i>)
8			num
9			name

Lösung

0	<i>Goal</i>	\mapsto	<i>Expr</i>
1	<i>Expr</i>	\mapsto	<i>Term Expr'</i>
2	<i>Expr'</i>	\mapsto	$+ \textit{Term Expr}'$
3			$- \textit{Term Expr}'$
4			ϵ
5	<i>Term</i>	\mapsto	<i>Factor Term'</i>
6	<i>Term'</i>	\mapsto	$* \textit{Factor Term}'$
7			$/ \textit{Factor Term}'$
8			ϵ
9	<i>Factor</i>	\mapsto	(\textit{Expr})
10			num
11			name

Backtrack-Free Parser

- ▶ das Hauptproblem das zu ineffizientem, linkskanonischem Top-Down Parsen führen kann, ist Backtracking
- ▶ Backtracking kann vermieden werden, wenn der Parser immer die “richtige” Regel auswählt
- ▶ für die vorherige Grammatik, kann der Parser beides, das fokusierte Symbol und das nächste Eingabesymbol, in Betracht ziehen, um die nächste Regel auszuwählen
- ▶ das nächste Eingabesymbol heißt **lookahead symbol**
- ▶ wir können also sagen, dass eine Grammatik frei von Backtracking ist mit einem Symbol lookahead
- ▶ Eine solche Grammatik heißt auch *Predictive Grammar*

Linksfaktorisierung zum Eliminieren von Backtracking

- ▶ erweitern wir unsere Grammatik durch die folgenden Regeln

11	<i>Factor</i>	↦	name
12			name [<i>ArgList</i>]
13			name (<i>ArgList</i>)
15	<i>ArgList</i>	↦	<i>Expr</i> <i>MoreArgs</i>
16	<i>MoreArgs</i>	↦	, <i>Expr</i> <i>MoreArgs</i>
17			ε

- ▶ mit einem Lookahead von name kann der Parser nicht entscheiden ob er Regel 11, 12 oder 13 nehmen soll
- ▶ durch Linksfaktorisierung können wir die Regel ändern in:

11	<i>Factor</i>	↦	name <i>Arguments</i>
12	<i>Arguments</i>	↦	[<i>ArgList</i>]
13			(<i>ArgList</i>)
14			ε

Top-Down rekursiv-absteigende Parser

- ▶ Backtracking-freie Grammatiken eignen sich zum einfachen und effizienten Parsen mit rekursiv-absteigenden Parsern
- ▶ ein rekursiv-absteigender Parser wird durch eine Menge sich gegenseitig rekursiv aufrufender Prozeduren, eine für jedes nichtterminale Symbol der Grammatik
- ▶ gegeben seien die drei folgenden Regeln:

2	$Expr'$	\mapsto	$+$	$Term$	$Expr'$
3			$-$	$Term$	$Expr'$
4			ϵ		

- ▶ um Instanzen von $Expr'$ zu erkennen, wird eine Prozedur `EPrime()` implementiert
 - ▶ die eine Regel gem. dem Lookahead Symbol auswählt
 - ▶ und in Abhängigkeit davon `NextWord()` und die entsprechende Prozedur aufruft

Table-driven LL(1) Parsers

- ▶ mit Tools können automatisch effiziente Top-Down Parser für Backtracking-freie Grammatiken generiert werden
- ▶ die erzeugten Parser heißen LL(1) Parser weil
 - ▶ sie die Eingabe von links nach rechts verarbeiten,
 - ▶ eine linkskanonische Ableitung konstruieren und
 - ▶ ein Lookahead von **1** Symbol nutzen.
- ▶ Grammatiken die nach einem LL(1)-Schema arbeiten, heißen **LL(1) Grammatiken** und sind, per Definition, frei von Backtracking
- ▶ die am meisten verbreitetste Implementierungstechnik nutzt einen *table-driven skeleton parser*

Bottom-Up Parsing

- ▶ Bottom-Up Parser erzeugen den Parsebaum indem sie an den Blättern starten und sich nach oben zur Wurzel arbeiten
- ▶ der Parser erzeugt für jedes Wort das der Scanner liefert ein Blatt
- ▶ um eine Ableitung zu erzeugen, fügt der Parser an der oberen Grenze eine Schicht von Nichtterminalen über die Blätter
- ▶ der Parser sucht an der oberen Grenze nach einer Zeichenkette die zur rechten Seite einer Produktionsregel $A \mapsto \beta$ passt
- ▶ wenn er β findet, erzeugt er einen Knoten für A und verbindet die Knoten die β repräsentieren mit A als Kindknoten
- ▶ das Vorgehen nennen wir **Reduktion**, weil es die Anzahl der Knoten an der oberen Grenze reduziert
- ▶ das Ersetzen von β durch A an der Position k wird geschrieben $\langle A \mapsto \beta, k \rangle$ und heißt ein **Handle**

Bottom-Up Parsing...

- ▶ der Bottom-Up Parser wiederholt diesen einfachen Prozess
- ▶ er findet ein Handle $\langle A \mapsto \beta, k \rangle$ an der oberen Grenze
- ▶ er ersetzt das Vorkommen von β bei k mit A
- ▶ dieser Prozess wiederholt sich bis entweder
 1. er die gesamte Grenze zu einem einzigen Knoten ersetzt, der das Startsymbol der Grammatik repräsentiert oder
 2. er kein Handle findet.
- ▶ im ersten Fall hat der Parser eine Herleitung gefunden und, wenn er bereits den gesamten Eingabestrom verbraucht hat, ist erfolgreich
- ▶ im zweiten Fall meldet der Parser einen Fehler
- ▶ in vielen Fällen kann der Parser aber trotz Fehler weiter machen (error recovery) und so in einem Lauf möglichst viele Fehler finden

Beziehung zwischen dem Parsen und der Herleitung

- ▶ der Bottom-Up Parser arbeitet vom fertigen Satz zum Startsymbol
- ▶ die Herleitung beginnt mit dem Startsymbol und arbeitet bis zum fertigen Satz
- ▶ der Parser findet die Herleitung also rückwärts
- ▶ der Scanner ermittelt die Wörter von links nach rechts
- ▶ ein Bottom-Up Parser sucht nach der von rechtskanonischen Ableitung
- ▶ für eine Herleitung
 $Goal = \gamma_0 \mapsto \gamma_1 \mapsto \gamma_2 \mapsto \dots \mapsto \gamma_{n-1} \mapsto \gamma_n = \textit{sentence}$
findet der Parser $\gamma_i \mapsto \gamma_{i+1}$ bevor er $\gamma_{i-1} \mapsto \gamma_i$ findet

LR(1) Parser

- ▶ die rechtskanonische Ableitung ist eindeutig, wenn die Grammatik keine Mehrdeutigkeiten enthält
- ▶ für eine große Klassen eindeutiger Grammatiken ist γ_{i-1} direkt durch γ_i und ein kleines bißchen Lookahead bestimmt
- ▶ für solche Grammatiken können wir einen effizienten Algorithmus zum Finden von Handles konstruieren
- ▶ die Technik dazu heißt LR-Parsing
- ▶ ein LR(1)-Parser liest den Eingabestrom von links nach rechts um eine rechtskanonische Ableitung rückwärts herzuleiten
- ▶ der Name LR(1) bezieht sich auf
 - ▶ **L**eft-to-right scan,
 - ▶ **R**everse rightmost derivation, und
 - ▶ **1** symbol of lookahead.

Praktische Parsing-Probleme

Error Recovery

- ▶ ein Parser sollte so viele Syntaxfehler wie möglich in einem Durchgang finden
- ▶ dazu benötigen wir einen Mechanismus mit dem der Parser nach einem Fehler wieder in einen Zustand kommen kann aus dem er weiter parsen kann
- ▶ ein üblicher Ansatz ist ein oder mehrere Wörter zu wählen mit denen der Parser den Eingabestrom wieder mit seinem internen Zustand synchronisieren kann
- ▶ wenn der Parser einen Fehler findet, verwirft er solange Eingabesymbole, bis er ein solches Synchronisierungswort findet

Semikolons finden

- ▶ in Sprachen die ein Semikolon verwenden um Anweisungen von einander zu trennen, braucht der Parser nur alles bis zum nächsten Semikolon verwerfen
- ▶ in einem rekursiv-absteigenden Parser kann der Code einfach die Wörter bis dahin ignorieren
- ▶ bei einem LR(1)-Parser ist das etwas komplexer
- ▶ bei einem table-driven Parser muss der Compiler dem Parsergenerator sagen können wo er synchronisieren kann
 - ▶ das kann durch “Fehlerproduktionsregeln” erreicht werden — eine Produktionsregel bei der die rechte Seite ein reserviertes Wort enthält, dass die Fehlersynchronisation anzeigt und ein oder mehrere Synchronisationstokens

Unäre Operatoren

- ▶ es ist nicht ganz einfach unäre Operatoren zur Expression-Grammatik hinzuzufügen
- ▶ fügen wir z.B. den unären Absolutbetragsoperator $||$ mit höherem Vorrang als die binären Operatoren und geringerem Vorrang als Klammern hinzu

Unäre Operatoren — Beispiel

0	<i>Goal</i>	\mapsto	<i>Expr</i>
1	<i>Expr</i>	\mapsto	<i>Expr</i> + <i>Term</i>
2			<i>Expr</i> - <i>Term</i>
3			<i>Term</i>
4	<i>Term</i>	\mapsto	<i>Term</i> * <i>Factor</i>
5			<i>Term</i> / <i>Factor</i>
6			<i>Value</i>
7	<i>Value</i>	\mapsto	<i>Factor</i>
8			<i>Factor</i>
9	<i>Factor</i>	\mapsto	(<i>Expr</i>)
10			num
11			name

- ▶ diese Grammatik erlaubt beispielsweise nicht |||x zu schreiben

Kontextsensitive Mehrdeutigkeit

- ▶ das Benutzen eines Wortes um mehrere Bedeutungen zu repräsentieren, kann syntaktische Uneindeutigkeit hervorrufen
- ▶ ein Beispiel gab es in einigen frühen Programmiersprachen, wie z.B. Fortran, PL/I und Ada
- ▶ diese Sprachen nutzen runde Klammern für beides
 - ▶ Zugriff auf Element eines Arrays über den Index
 - ▶ Parameterlisten von Prozeduren und Funktionen
- ▶ bei `foo(i, j)` konnte der Compiler also nicht feststellen ob `foo` ein zweidimensionales Array oder eine Prozedur/Funktion ist
- ▶ der Scanner klassifizierte `foo` einfach nur als `name`

Ein Ansatz um das Problem zu lösen

umschreiben der Grammatik, so dass Funktionsaufruf und Array-Referenz eine einzige Produktion sind

- ▶ das Problem ist dann in einen späteren Schritt der Übersetzung verschoben
- ▶ es kann dann mit Hilfe von Informationen aus Deklarationen gelöst werden
- ▶ der Parser muss eine Repräsentation konstruieren, die alle später notwendigen Informationen enthält
- ▶ der spätere Schritt schreibt dies dann noch einmal um

Ein zweiter Ansatz um das Problem zu lösen

der Scanner kann die Bezeichner gemäß ihrer deklarierten Typen klassifizieren

- ▶ das setzt eine Zusammenarbeit zwischen Scanner und Parser voraus
- ▶ solange die Sprache die “define-before-use”-Regel befolgt, ist das problemlos machbar
- ▶ die Deklaration wird dann ja vor dem Scannen des Ausdrucks bereits geparst
- ▶ der Parser kann seine interne Symboltabelle dem Scanner zur Verfügung stellen um Bezeichner in verschiedene Klassen einzuteilen, z.B. `variable-name` und `function-name`

Links vs. Rechtsrekursion

- ▶ Top-Down Parser brauchen rechtsrekursive Grammatiken, Bottom-Up Parser können mit beiden arbeiten
- ▶ der Compilerbauer muss auswählen
- ▶ verschiedene Faktoren beeinflussen die Entscheidung:

Stacktiefe im allgemeinen kann Linksrekursion mit geringeren Stacktiefen zurecht kommen

Assoziativität Linksrekursion erzeugt auf natürliche Weise Linksassoziativität, Rechtsrekursion erzeugt Rechtsassoziativität