

Parser

Compiler

Prof. Dr. Oliver Braun

Letzte Änderung: 27.01.2020 19:45

- 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

- 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 $NT \mapsto (T \cup NT)^+$

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

- die Sprache der Schafe:

$SheepNoise \mapsto \mathbf{baa} SheepNoise \mid \mathbf{baa}$

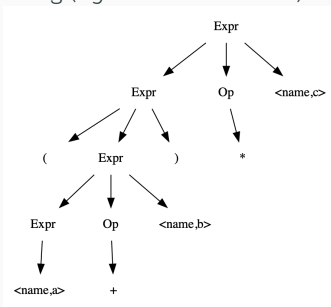
- eine Sprache von arithmetischen Ausdrücken mit Klammern

$1 \text{ Expr} \mapsto (\text{Expr}) \mid 2 \mid \text{Expr Op name} \mid 3 \mid \text{name Op} \mapsto + \mid 5 \mid - \mid 6 \mid * \mid 7 \mid /$

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)

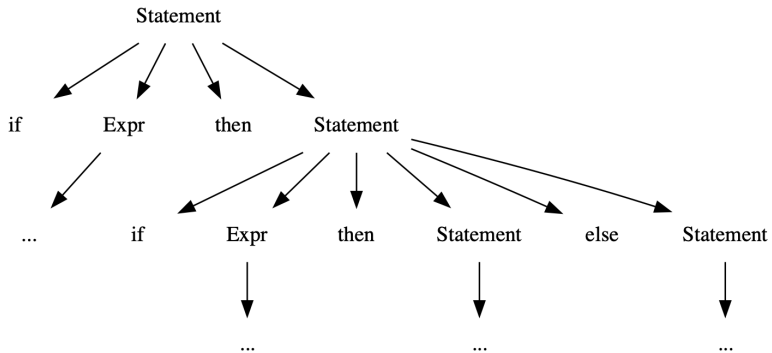
- die linkskanonische Ableitung (*leftmost derivation*) nutzt die Sequenz (2,1,2,3,4,6) und resultiert offensichtlich im selben Parsebaum

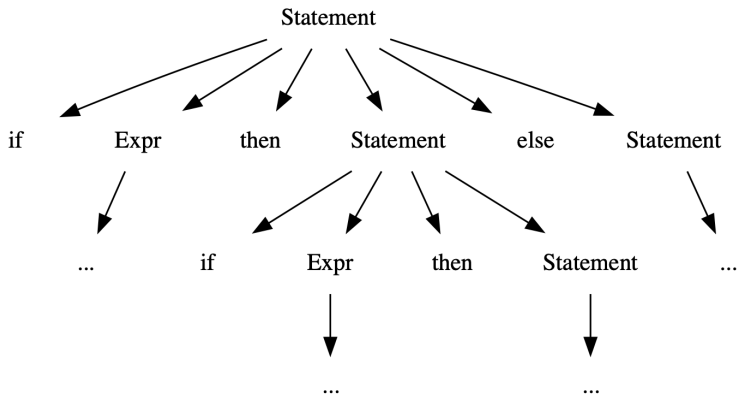
- für einen Compiler ist es wichtig, dass jeder Satz eine eindeutige (rechts- oder linkskanonische) Herleitung hat
- wenn es verschiedene Herleitungen für einen Satz gibt, heißt die Grammatik mehrdeutig
- eine solche Grammatik kann für einen Satz verschiedene Parsebäume erzeugen, die potentiell verschiedene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

Der Klassiker für eine mehrdeutiges Konstrukt

1 *Statement* \mapsto **if** *Expr* **then** *Statement* **else** *Statement* 2 | **if** *Expr* **then**
Statement 3 | **Assignment** 4 | ...*other statements*...

.. kann in diesen Parsebaum resultieren:

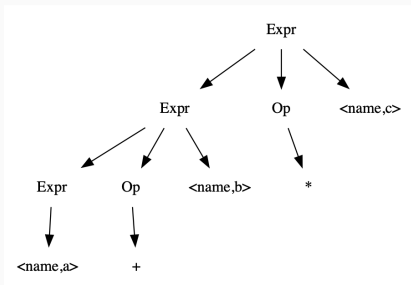




1 *Statement* \mapsto **if** *Expr* **then** *Statement 2* | **if** *Expr* **then** *WithElse* **else**
Statement 3 | *Assignment 4* | ...other statements... 5 *WithElse* \mapsto **if** *Expr*
then *WithElse* **else** *WithElse 6* | *Assignment*

Bedeutung in Struktur kodieren

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



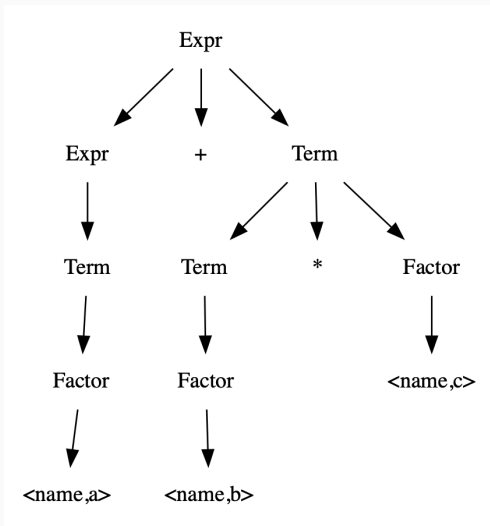
- ein naheliegender Ansatz den Ausdruck auszuwerten ist den Baum “Postorder” zu durchlaufen
- aber das resultiert in $(a + b) * c$ und nicht $a + b * c$, da wir bislang keinen Vorrang der Operatoren in der Grammatik formuliert haben

Operator Vorrang hinzufügen

0 $Goal \mapsto Expr$ 1 $Expr \mapsto Expr + Term$ 2 | $Expr - Term$ 3 | $Term$ 4 $Term \mapsto Term$
* $Factor$ 5 | $Term / Factor$ 6 | $Factor$ 7 $Factor \mapsto (Expr)$ 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
 - a) der untere Rand des Baumes nur terminale Symbole enthält **und** der Eingabestrom zuende ist, oder
 - b) 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

- 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

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

$$Fee \mapsto Fee \alpha \mid \beta$$

- diese können wir aber einfach eliminieren durch beispielsweise

$$Fee \mapsto \beta Fee' \quad Fee' \mapsto \alpha Fee' \mid \epsilon$$

Eliminieren Sie die Linksrekursion

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

0 Goal \mapsto Expr 1 Expr \mapsto Term Expr' 2 Expr' \mapsto + Term Expr' 3 | - Term Expr' 4
| \in 5 Term \mapsto Factor Term' 6 Term' \mapsto * Factor Term' 7 | / Factor Term' 8 | \in 9
Factor \mapsto (Expr) 10 | num 11 | name

- 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 fokussierte 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

$$\begin{aligned} 11 \text{ Factor} &\mapsto \text{name } 12 \mid \text{name } [\text{ArgList}] 13 \mid \text{name } (\text{ArgList}) 15 \text{ ArgList} \\ &\mapsto \text{Expr MoreArgs } 16 \text{ MoreArgs} \mapsto , \text{Expr MoreArgs } 17 \mid \epsilon \end{aligned}$$

- 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:

$$11 \text{ Factor} \mapsto \text{name Arguments } 12 \text{ Arguments} \mapsto [\text{ArgList}] 13 \mid (\text{ArgList}) 14 \mid \epsilon$$

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:

$$2 \text{ Expr}' \mapsto + \text{Term Expr}' \quad 3 \mid - \text{Term Expr}' \quad 4 \mid \epsilon$$

- um Instanzen von Expr' zu erkennen, wird eine Prozedur $\text{EPrime}()$ implementiert

- die eine Regel gem. dem Lookahead Symbol auswählt

- 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 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

- 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

- 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

- 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
 - Left-to-right scan,
 - Reverse rightmost derivation, und

Praktische Parsing-Probleme

- 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

- 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

- 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

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

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

- 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`

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

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`

- 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