Syntaktische Analyse

Contents

[Begriffe 1](#_Toc66900348)

[First(A) 1](#_Toc66900349)

[Follow(A) 2](#_Toc66900350)

[„#“ in First und Follow: 2](#_Toc66900351)

[Indizmenge 2](#_Toc66900352)

[SLL 2](#_Toc66900353)

[Allgemein 2](#_Toc66900354)

[Speziallfall 2](#_Toc66900355)

[FAQ 2](#_Toc66900356)

[Wofür braucht man Indizmenge? 2](#_Toc66900357)

[Warum dürfen die Indizmengen von verschieden Produktionen von einem Nichtterminal sich nicht überlappen? 3](#_Toc66900358)

[Warum soll eine Grammatik SLL(1) sein? 3](#_Toc66900359)

[Wie erstellt man ein Parser für die Grammatik (Rekursiver Abstieg)? 3](#_Toc66900360)

[Was macht lexer? 3](#_Toc66900361)

[Was ist lexer.current und lexer.lex()? 3](#_Toc66900362)

[Beispiele 3](#_Toc66900363)

[Für First und Follow: 3](#_Toc66900364)

[Grammatik für Lambda Ausdrücke aus Ü12: 4](#_Toc66900365)

[Indizmenge für Lambda Ausdrücke (aus Ü12): 4](#_Toc66900366)

[Grammatik 4](#_Toc66900367)

[Erklärung 4](#_Toc66900368)

[Vorgehensweise: 5](#_Toc66900369)

[Parser für Grammatik in Pseudocode: 5](#_Toc66900370)

# Begriffe

## First(A)

First\_1(*A*) enthält alle Terminale, die bei irgendeiner Ableitung von *A* an erster Stelle stehen.

## Follow(A)

Follow\_1(*A*) enthält alle Terminale, die bei irgendeiner Ableitung von *S* (dem Startsymbol) direkt hinter *A* stehen. Sie müssen sich also überlegen, welche Ableitungsschritte zu einem Vorkommen von *A* führen, und was dann direkt dahinter stehen kann.

## „#“ in First und Follow:

In beiden Fällen # enthalten ist, falls in irgendeiner Ableitung die entsprechende Stelle das Ende der Ableitung ist

## Indizmenge

Indizmenge I von A -> a ist mit k Token Lookahead genau   
First\_k(a Follow\_k (A)).

Diese komische Konstruktion mit Follow ist nur dazu da, um die Fälle a=epsilon zu kompensieren.

Wenn es nur eine Produktion für diesen Nichtterminal gibt, dann kann man als Indizmenge für diesen Nichtterminal einfach „**egal**“ schreiben.

## SLL

### Allgemein

Eine KFG ist genau dann eine SSL(k)-Grammatik, wenn für alle Paare von Produktionen (von einem Nichttermnal) A -> a | b, (a!=b) gilt:

First\_k(a Follow\_k(A)) SCHNITT First\_k(b Followk(A)) = Leer

(== verschiedene Produktionen von einem Nichtterminal haben vollständig disjunkte Indizmengen)

Sonst wäre die Unterscheidung mit k-Lookahead nicht möglich.

### Speziallfall

**(oft in Klausuren!)**

k=1, a !->\* epsilon und b !->\* epsilon (keine Eps möglich):

SLL <--> Fist(a) SCHNITT First(b) = leer

k=1, a ->\* epsilon und b !->\* epsilon (Eps nur bei A möglich):

SLL <--> Follow(A) SCHNITT First(b) = leer

# FAQ

## Wofür braucht man die Indizmenge?

Mit Hilfe der Indizmenge kann Compiler eine Entscheidung machen, welche Produktionen gewählt sein müssen.

## Warum dürfen die Indizmengen von verschieden Produktionen von einem Nichtterminal sich nicht überlappen?

Sonst gäbe es mehrere korrekte Entscheidungen für Compiler -> Codegeneration nicht eindeutig.

## Warum soll eine Grammatik SLL(1) sein?

Sonst wäre die Unterscheidung mit 1-Lookahead nicht möglich -> Kein effizienten Compiler möglich.

## Wie erstellt man ein Parser für die Grammatik (Rekursiver Abstieg)?

Es muss ein Parser für ein Nichtterminal implementiert werden.

Wichtigste Bausteine: lexer.current, lexer.lex(), switch(lexer.current), schleifen, default: error, manchmal Schleifen mit while(true) und break nach dem switch mit break und continue in verschiedenen Cases, return new <Classname>.

Jeder Parser wird als eine Methode implementiert, die entweder ein Objekt von passendem Typ zurückliefert oder error() aufruft.

Name: „**Rekursiver Abstieg“**

## Was macht lexer?

Lexer ist ein „Lexical-Analyzer“-Objekt .

## Was ist lexer.current und lexer.lex()?

lexer.current ist current Token Type.

lexer.lex() setzt lexer.current auf das nächste Token.

# Beispiele

## Für First und Follow:

Grammatik:

S -> L**=** R| R

L -> \*R | id

R -> L

(=. id, \* sind Terminale)

First1(L) = {\**,* **id**}

First1(R) = {\**,* **id**}

First1(S) = First1(L) U First1(R) = {\**,* **id**}

Follow1(S) = {#}

Follow1(S) = {=, #}

Follow1(S) = {=, #}

## Grammatik für Lambda Ausdrücke aus Ü12:

Grammatik:

L --> Abstr

Abstr --> lambda id . Abstr | App

App --> App Atom | Atom

Atom --> Var | (Abstr)

Das Startsymbol der Grammatik soll *L* heißen, entweder nutzen wir diesen Namen für das

niedrigstpriore Syntaxelement oder lassen es wie hier auf das Element für Abstraktion ableiten.

Die Abstraktion hat die niedrigste Priorität, damit kommt die dazugehörige Regel in der Grammatik am Anfang.

Ganz allgemein gilt: In der Regel für das Syntaxelement der Priorität *n* nutzen wir nur die

Nichtterminale der Syntaxelemente mit Priorität *n* oder *n* + 1.

## Indizmenge für Lambda Ausdrücke (aus Ü12):

Grammatik:

App --> Abstr AppList [egal]

AppList --> eps [ ) , # ] | Abstr AppList [lambda, id, (]

Abstr --> Atom [id, (] | lambda id . Abstr [lambda]

Atom --> Var [id] | (App) [ ( ]

Var --> id [egal]

Indizmenge in [ ]

Erklärung:

App: egal, nur 1 Produktion (alternativ [id, (, lambda])

AppList: eps: schaue, was nach dem App kommen kann, da Applist nur links bei App vorkommt. **Abstr -> alle Indizies von Abstr**

Abstr: **Atom -> alle Indizies von Atom**, zweite Produktion beginnt mit Terminalsymbol

Atom: in Var kommt immer id, zweite Produktion beginnt mit Terminalsymbol

Var: egal, nur 1 Produktion (alternativ [id])

### Vorgehensweise:

1. Falls es nur eine Produktion gibt: „egal“
2. Falls es nur Terminalsymbole gibt: giese Terminalsymbole
3. Gehe von unten nach oben, berechne einfache Indizmengen, die Terbinalsymbole als 1. Zeichen haben (hier wäre es Var, Atom, lambda bei Abstr)
4. Indiz von einem Nichtterminal ist die Vereinigung von allen Indizen seinen Produktionen
5. Epsilon bei X: berechne First(Follow(X))

## Parser für Grammatik in Pseudocode:

*Value* --> **string** | **number** | *Object*

*Object* --> **{** *Members* **}**

*Members* --> *Pair Members*’

*Members*‘ --> **,** *Members* | *eps*

*Pair* --> **string :** *Value*

JSONValue parseValue() {

String s;

**switch** (lexer.current.type) {

**case** STRING:

s = lexer.current.text;

lexer.lex();

**return new** JSONString(s);

**case** NUMBER:

s = lexer.current.text;

lexer.lex();

**return new** JSONNumber(s);

**case** LCURLY:

**return** parseObject();

**default**: error();

}

}

JSONObject parseObject() {

**switch** (lexer.current.type) {

**case** LCURLY: lexer.lex();

JSONObject o = **new** JSONObject(parseMembers());

**if** (lexer.current.type != RCURLY) error();

lexer.lex();

**return** o;

**default**: error();

}

}

List<Pair> parseMembers() {

List<Pair> l = **new** LinkedList<Pair>();

l.add(parsePair());

**while** (**true**) {

**switch** (lexer.current.type) {

**case** COMMA: lexer.lex();

l.add(parsePair());

**continue**;

**case** RCURLY: **break**; // break switch

**default**: error();

}

**break**; // break while

}

**return** l;

}

Pair parsePair() {

**if** (lexer.current.type != TokenType.STRING) error();

JSONString s = **new** JSONString(lexer.current.text);

lexer.lex();

**if** (lexer.current.type != TokenType.COLON) error();

lexer.lex();

JSONValue v = parseValue();

**return new** Pair(s, v);

}