Syntaktische Analyse

Contents

Begriffe

First(A)

First_1(*A*) enthält alle Terminale, die bei irgendeiner Ableitung von *A* an erster Stelle stehen.
! Wenn es ein Regel (A -> eps) gibt ----> # ist immer in First(A)

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.

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

a und b sind die verschiedene Produktionen von einem Nichtterminal. Hier werden nicht die verschiedene Nichtterminale verglichen!

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

Linksfaktorisierung

Idee: Gemeinsame Anfang nach eine Variable verschieben, und eps | Rest nach andere Regel

Grammar Engineering: Linksfaktorisierung

Grammatik mit Produktionen $X \to \gamma\alpha \mid \gamma\beta$ mit gemeinsamen Anfang γ ist nicht SLL, wenn Länge $|\omega|$,$\gamma \Rightarrow^* \omega$, unbeschränkt.[9]

Lösung: vertage Entscheidung zwischen α und β bis nachdem γ konsumiert wurde:

$$X \to \gamma X'$$
$$X' \to \alpha \mid \beta$$

Wobei X' ein neues Nichtterminal ist.

Beispiel zur Linksfaktorisierung

If-Produktionen von SIMPLE:

If → **if** *BraceExp Statement*
| **if** *BraceExp Statement* **else** *Statement*

haben gemeinsamen Anfang **if** *BraceExp Statement*.
Linksfaktorisierung ergibt:

If → **if** *BraceExp Statement If′*
If′ → ε | **else** *Statement*

Nach Linksfaktorisierung sind die Produktionen SLL.

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.

Was ist expect()?

Gute Sache: expect(Type) prüft Typ und verschiebt Token-Pointer

Vereinfachung: Gemeinsame Parse-Funktion für (fast) alle Terminale:

```
void expect(TokenType e)
{  if lexer.current == e then lexer.lex() else error()  }
```

Wie beweist man formal, dass Grammatik nicht SLL1 ist?

Gegeben sei die folgende Grammatik, die eine Untermenge von SGML beschreibt:

$$\begin{aligned} \mathit{SGML} &\rightarrow \texttt{<ident>}\ \mathit{Children}\ \texttt{</>} \\ \mathit{Children} &\rightarrow \varepsilon \mid \mathit{SGML}\ \mathit{Children} \end{aligned}$$

Das Startsymbol dieser Grammatik ist SGML.

(a) Begründen Sie formal, warum die obige Grammatik nicht in SLL(1)-Form ist. [3 Punkte]

Children: a |b , a -> eps, b !-> eps --> 2.Speziallfall für SLL

(a) Es gilt $\text{Follow}_1(\mathit{Children}) = \{\texttt{<}\}$. Wir prüfen die SLL(1)-Bedingung anhand der Produktionen von $\mathit{Children}$:

$$\begin{aligned} & \text{First}_1(\varepsilon \cdot \text{Follow}_1(\mathit{Children})) \cap \text{First}_1(\mathit{SGML} \cdot \mathit{Children} \cdot \text{Follow}_1(\mathit{Children})) \\ = \ & \text{Follow}_1(\mathit{Children}) \cap \text{First}_1(\mathit{SGML}) \\ = \ & \{\texttt{<}\} \cap \{\texttt{<}\} \\ \neq \ & \emptyset \end{aligned}$$

Aufgaben

First und Follow

ohne eps

Grammatik (=. id, * sind Terminale) :

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

First1(L) = {*, **id**}
First1(R) = {*, **id**}
First1(S) = First1(L) U First1(R) = {*, **id**}
Follow1(S) = {#}
Follow1(S) = {=, #}
Follow1(S) = {=, #}

mit eps

Grammatik:

```
A -> BC
B -> A | eps
C -> id
```

First(A) = {id}: A -> BC -> eps C -> id
First(B) = {id, #}: # wegen eps, id: B->A->BC->eps C -> C -> id
First(C) = {id}: trivial
Follow(A) = {id, #}: A -> BC -> AC ---> Follow(A) + # wegen Startsymbol
Follow(B) = {id}: nur C->id
Follow(C) = {id, #} : A -> BC -> AC -> BCC -> BC id, A -> BC#

Lambda Ausdrücke

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

```
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 (Follow(App)), 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 für die Indizberechnung

1. Falls es nur eine Produktion gibt: „egal“
2. Falls es nur Terminalsymbole gibt: diese 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

App	→	*Abstr AppList*	egal / {**id**, **(**, λ}
AppList	→	ε {**)**, **#**}	| *Abstr AppList* {**id**, **(**, λ}
Abstr	→	*Atom* {**id**, **(**}	| λ **id** . *Abstr* {λ}
Atom	→	*Var* {**id**}	| **(** *App* **)** {**(**}
Var	→	**id**	egal / {**id**}

Diese Grammatik ist SLL(1). Rechnen Sie nach, indem Sie die nötigen Indizmengen explizit aufstellen und in die dafür vorgesehenen Kästchen eintragen.

(c) Vervollständigen Sie den unten angegebenen rekursiven Abstiegsparser so, dass er die Variante des λ-Kalküls aus (b) in folgende Datenstruktur parst: [12 Punkte]

```
1  abstract class Expr {}  // expression
2
3  class Var extends Expr {
4    public Var(String name) ...
5  }
6  class Abstr extends Expr {  // abstraction
7    public Abstr(String varname, Expr body) ...
8  }
9  class App extends Expr {  // application
10   public App(Expr left, Expr right) ...
11 }
```

```
Expr parseAtom() {
  // gegeben
  ...
}

String expect(Token tok) {
  if (lexer.current != tok)
    error();
  lexer.lex();
  return lexer.string_value;
}

Expr parseApp() {
  Expr l = parseAbstr();
  return parseAppList(l);
}
```

Beispiellösung:

```
Expr parseAbstr() {
  switch (lexer.current) {
  case IDENT: case LPAREN:
    return parseAtom();
  case LAMBDA:
    expect(LAMBDA);
    String varname = expect(IDENT);
    expect(DOT);
    Expr body = parseAbstr();
    return new Abstr(varname, body);
  default:
    error();
  }
}

// Hier müssen die linken Hälften der Applikationen durchgefädelt werden
Expr parseAppList(Expr left) {
  switch(lexer.current) {
  case RPAREN: case EOF: // epsilon, da war gar keine Applikation
    return left;
  case IDENT: case LAMBDA: case LPAREN:
    Expr right = parseAbstr();
    left = new App(left, right);
    return parseAppList(left);
  default:
    error();
  }
}
```

JSON

Grammatik

$$
\begin{aligned}
\mathit{Value} &\rightarrow \underline{\texttt{string}} \mid \underline{\texttt{number}} \mid \mathit{Object} \\
\mathit{Object} &\rightarrow \underline{\{}\ \mathit{Members}\ \underline{\}} \\
\mathit{Members} &\rightarrow \mathit{Pair}\ \mathit{Members}' \\
\mathit{Members}' &\rightarrow \underline{,}\ \mathit{Members} \mid \epsilon \\
\mathit{Pair} &\rightarrow \underline{\texttt{string}}\ \underline{:}\ \mathit{Value}
\end{aligned}
$$

Parser

```
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;
      default: error();
    }
    break;
  }
  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);
}
```

Linksfaktorisierung

Gegeben sei die folgende Grammatik für Prolog-Fakten mit dem Startsymbol *Fact*:

$$
\begin{array}{rcl}
\mathit{Fact} & \rightarrow & \mathit{Term}\ \texttt{.} \\
\mathit{Term} & \rightarrow & \texttt{atom} \mid \texttt{atom (}\ \mathit{Termlist}\ \texttt{)} \mid \texttt{number} \mid \texttt{variable} \\
\mathit{Termlist} & \rightarrow & \mathit{Term}\ \mathit{Termlist}' \\
\mathit{Termlist}' & \rightarrow & \varepsilon \mid \texttt{,}\ \mathit{Term}\ \mathit{Termlist}'
\end{array}
$$

(a) Geben Sie die First_1- und Follow_1-Mengen für die Nichtterminale *Fact* und *Term* an. [4 Punkte]

(b) Geben Sie eine Linksfaktorisierung an, so dass die Grammatik SLL(1) ist. Es genügt geänderte bzw. neue Zeilen der Grammatik anzugeben. [2 Punkte]

Nicht SLL(1), da es die gleiche First-Mengen bei verschiedenen Produktionen vom „Term“ gibt.

(a)

$$
\begin{array}{rcl}
\text{First}_1(\mathit{Fact}) & = & \{\texttt{atom}, \texttt{number}, \texttt{variable}\} \\
\text{First}_1(\mathit{Term}) & = & \{\texttt{atom}, \texttt{number}, \texttt{variable}\} \\
\text{Follow}_1(\mathit{Fact}) & = & \{\texttt{\#}\} \\
\text{Follow}_1(\mathit{Term}) & = & \{\texttt{.}, \texttt{,}, \texttt{)}\}
\end{array}
$$

(b)

$$
\begin{array}{rcl}
\mathit{Term} & \rightarrow & \texttt{atom}\ \mathit{Atom}' \mid \texttt{number} \mid \texttt{variable} \\
\mathit{Atom}' & \rightarrow & \varepsilon \mid \texttt{(}\ \mathit{Termlist}\ \texttt{)}
\end{array}
$$

Prolog Parser

Grammatik

Fact -> Term **.**
Term -> **atom** Atom’ | **number** | **variable**
Atom’ -> **eps** | **(** Termlist **)**
Termlist -> Term Termlist’
Termlist’ -> eps | **,** Term Termlist’

Parser

```
void parseFact() {
    parseTerm();
    if (token.getType() != dot) {
        error();
    }
    nextToken();
}

void parseTerm() {
    switch (token.getType()) {
    case atom:
        nextToken();
        parseAtom'();
        break;
    case number:
    case variable:
        nextToken();
        break;
    default:
        error();
    }
}

void parseAtom'() {
    switch (token.getType()) {
    case dot:
    case comma:
    case rp:
        break;
    case lp:
        nextToken();
        parseTermlist();
        if (token.getType() != rp) {
            error();
        }
        nextToken();
        break;
    default:
        error();
    }
}

void parseTermlist() {
    parseTerm();
    parseTermlist'();
}

void parseTermlist'() {
    switch (token.getType()) {
    case rp:
        break;
    case comma:
        nextToken();
        parseTerm();
        parseTermlist'();
        break;
    default:
        error();
    }
}
```

Relationen zwischen First und Follow (Teilmengen)

Aufgabe 11 (First_1- und Follow_1-Mengen) [3 Punkte]

Gegeben sei die folgende Produktion einer Grammatik G:

$$S \rightarrow AB$$

Es gilt

$$\text{First}_1(A) \subseteq \text{First}_1(S).$$

Geben Sie zwei weitere gültige Teilmengenbeziehung zwischen den Mengen

$$\begin{array}{ccc} \text{First}_1(S) & \text{First}_1(A) & \text{First}_1(B) \\ \text{Follow}_1(S) & \text{Follow}_1(A) & \text{Follow}_1(B) \end{array}$$

an. Triviale Teilmengenbeziehungen, wie $\text{First}_1(S) \subseteq \text{First}_1(S)$, sind dabei ausgeschlossen.

Beachten Sie, dass die Grammatik G weitere Produktionen mit den Nichtterminalen S, A und B enthalten kann.

Beispiellösung:

$$\begin{aligned} \text{First}_1(B) &\subseteq \text{Follow}_1(A) \\ \text{Follow}_1(S) &\subseteq \text{Follow}_1(B) \end{aligned}$$

SGML

Aufgabe

(b) Entwickeln Sie für die folgende, linksfaktorisierte SGML-Grammatik einen rekursiven Abstiegsparser mit AST-Aufbau in Pseudocode. [16 Punkte]

$$
\begin{aligned}
\textit{SGML} &\rightarrow \texttt{<}\ \textbf{ident}\ \texttt{>}\ \textit{ChildrenAndEnd} \\
\textit{ChildrenAndEnd} &\rightarrow \texttt{<}\ \textit{OpenOrClose} \\
\textit{OpenOrClose} &\rightarrow \texttt{/}\ \texttt{>} \\
&\quad\ |\ \ \textbf{ident}\ \texttt{>}\ \textit{ChildrenAndEnd}\ \textit{ChildrenAndEnd}
\end{aligned}
$$

Der Parser von *SGML* soll ein Objekt der Klasse `SGML` zurückgeben. Diese ist folgendermaßen definiert:

```
1  class SGML {
2    public String tag;
3    public List<SGML> children;
4    public SGML(String tag, List<SGML> children) { ... }
5  }
```

Hinweis zum AST-Aufbau:
In der Produktion *OpenOrClose* → **ident** > *ChildrenAndEnd* *ChildrenAndEnd* parst das erste Vorkommen von *ChildrenAndEnd* alle Kinder des gerade geöffneten Tags, das zweite Vorkommen parst die restlichen Kinder des umgebenden Tags.

Lexer-Schnittstelle:
Die globale Variable `token` enthält immer das aktuelle Token. Tokens besitzen die folgenden Methoden:

- `getType()` gibt den Token-Typ zurück
- `getIdent()` gibt einen String zurück, der den Namen eines **ident**-Tokens enthält.

Folgende Token-Typen sind definiert:

`IDENT`	für einen Bezeichner **ident**
`LT`	für das Kleiner-Zeichen **<**
`GT`	für das Größer-Zeichen **>**
`SLASH`	für den Schrägstrich **/**

Die globale Methode `nextToken()` setzt `token` auf das nächste Token. Brechen Sie bei Parsefehlern durch Aufruf der globalen `error()`-Methode ohne Fehlermeldung ab.

Parser

```
void expect(TokenType tt) {
   if (token.getType() != tt) {
      error();
   }
   nextToken();
}

SGML parseSGML() {
   expect(LT);

   if (token.getType() != IDENT) {
      error();
   }
   String tag = token.getIdent();
   nextToken();

   expect(GT);

   List<SGML> children = parseChildrenAndEnd();

   return new SGML(tag, children);
}

List<SGML> parseChildrenAndEnd() {
   expect(LT);

   return parseOpenOrClose();
}

List<SGML> parseOpenOrClose() {
   switch (token.getType()) {
   case SLASH:
      nextToken();

      expect(GT);

      return new ArrayList<SGML>();
   case IDENT:
      String tag = token.getIdent();
      nextToken();

      expect(GT);

      List<SGML> children = parseChildrenAndEnd();

      SGML elementHere = new SGML(tag, children);

      List<SGML> siblings = parseChildrenAndEnd();
      siblings.add(0, elementHere);

      return siblings;

   default:
      error();
      return null;
   }
}
```