

Übersicht

- Top-Down-Syntaxanalyse
- LL(k)
 - Definition
 - Beispiele und Eigenschaften
- Grammatikformungen und LL(k)
 - Grammatikformungen
 - Alternative Definition für LL(k)
 - Starke LL(k)-Grammatiken
- LL(k)-Parser
- Linksrekursion
- Rechtsreguläre kontextfreie Grammatiken (rrkfG)
- FIRST₁ und FOLLOW₁ für rrkfG
- RLL(1)-Parser (Tabellenversion)
- RLL(1)-Parser (Recursive descent)
- Behandlung von Syntaxfehlern
 - Fortsetzen bei LAST- und FOLLOW-Symbolen
 - Aufsetzen bei Anfangs- und Vorgänger-Symbolen

Lernziele

- Das Grundprinzip der Top-Down-Syntaxanalyse beherrschen
- Grundidee und wichtigste Aspekte von LL(k), insbesondere deren Umsetzung im Kontext von Parsern, benennen können
- Erklären können, wie man rechtsreguläre kontextfreie Grammatiken im Rahmen von tabellengesteuerten sowie Recursive-Descent-Parsern verwendet
- Die Grundideen der Behandlung von Syntaxfehlern benennen können

Top down-Parser

- Konstruiert Syntaxbaum von der Wurzel her
- Anfangssituation
- Aktivitäten
 - Produktion an aktuelles Baumfragment anbauen (nichtterminales Blatt durch rechte Seite ersetzen)
 - Übereinstimmung zwischen angebauten Symbolen und Eingabesymbolen verifizieren

Ist Linksparser

Baum
S

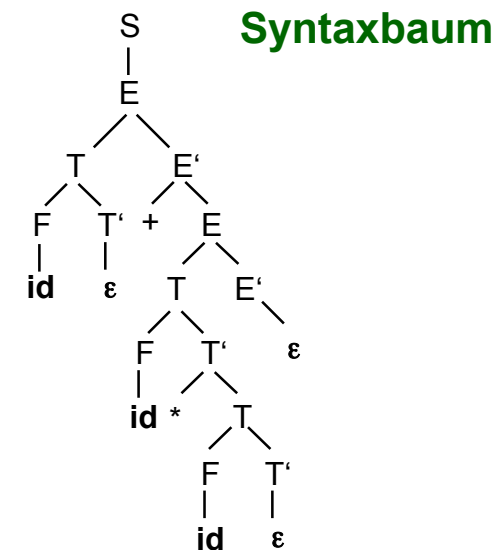
Resteingabe
id + id * id

Beispiel

- Geg.: kfG $G_2 = (\{S, E, E', T, T', F\}, \{+, *, (,), \text{id}\}, P, S)$ mit
 $P = \{S \rightarrow E, E \rightarrow TE', E' \rightarrow \varepsilon, E' \rightarrow +E, T \rightarrow FT',$
 $T' \rightarrow \varepsilon, T' \rightarrow *T, F \rightarrow (E), F \rightarrow \text{id}\}$

Eingabewort

id + id * id



Beobachtung

- Item-Kellerautomat K_G einer kfG G arbeitet im Prinzip wie ein Top-down-Parser
- Problem: Nichtdeterminismus bei (E)-Übergängen
(n Möglichkeiten bei n Alternativen einer Produktion)
- Abhilfe: Begrenzte Vorausschau in die restliche Eingabe

LL(k)-Grammatik

- Seien $G = (V_N, V_T, P, S)$ kfG, $k \in \mathbb{N}$
- G ist **LL(k)-Grammatik**, wenn gilt
 - Existieren zwei Linksableitungen
$$S \Rightarrow_{lm}^* uY\alpha \Rightarrow_{lm} u\beta\alpha \Rightarrow_{lm}^* ux$$
$$S \Rightarrow_{lm}^* uY\alpha \Rightarrow_{lm} u\gamma\alpha \Rightarrow_{lm}^* uy$$
 - und ist $k: x = k: y$
 - dann gilt $\beta = \gamma$

Intuitiv

- Auswahl der Alternative für Y bei festem Linkskontext u ist durch die k ersten Symbole der restlichen Eingabe eindeutig festgelegt

Beispiel 1

- kfG G_3 mit den Produktionen

STAT $\rightarrow$ **if id then** STAT **else** STAT **fi** |
 while id do STAT **od** |
 id := id

ist LL(1)-Grammatik

- **Beweisidee:**

Formaler Beweis, s.u.

- Erstes Symbol der Resteingabe legt auszuwählende Produktion eindeutig fest

Einfache LL(1)-Grammatik

- Sei G kfG ohne ε -Produktionen
- G heißt **einfache LL(1)-Grammatik**, wenn gilt:
Für jedes $X \in V_N$ beginnt jede der Alternativen (für X) mit einem anderen Terminalsymbol

Beispiel

- G_3 ist einfache LL(1)-Grammatik

Es gilt

- Nicht jede Grammatik ist (einfach) LL(1)

Beispiel 2

- kfG G_4 mit den Produktionen

STAT $\rightarrow$ **if id then** STAT **else** STAT **fi** |
 while id do STAT **od** |
 id := id |
 id: STAT |
 id (id)

ist offensichtlich nicht einfach LL(1), aber auch nicht LL(1)

- **Beweisidee:**
 - Für **id** als erstes Symbol der Resteingabe ist auszuwählende Produktion nicht eindeutig
- G_4 ist aber LL(2)-Grammatik, denn 2: x = „**id :=**“, 2: y = „**id:** “ und 2: z = „**id (**“ sind paarweise verschieden

Es gilt

- Es gibt Grammatiken, die nicht LL(k) für beliebiges k sind

Beispiel 3

- kfG G_5 mit den Produktionen

STAT $\rightarrow$ **if id then** STAT **else** STAT **fi** |

while id do STAT **od** |

 VAR := VAR |

id (IDLIST)

 (* Prozeduraufruf *)

VAR $\rightarrow$ **id** | **id** (IDLIST)

 (* indizierte Variable *)

IDLIST $\rightarrow$ **id** | **id**, IDLIST

ist nicht mehr LL(k) für beliebiges k

- **Beweisidee:**

- Wenn **id** (IDLIST) mehr als k Zeichen umfasst sind beide Produktionen „VAR := VAR“ und „**id** (IDLIST)“ mögliche Alternativen für STAT



Formale Beweise für die Beispiele

• Beweis für Beispiel 1:

- Für $\text{STAT} \Rightarrow_{\text{lm}}^* \text{u STAT} \alpha \Rightarrow_{\text{lm}} \text{u} \beta \alpha \Rightarrow_{\text{lm}}^* \text{ux}$ und
 $\text{STAT} \Rightarrow_{\text{lm}}^* \text{u STAT} \alpha \Rightarrow_{\text{lm}} \text{u} \gamma \alpha \Rightarrow_{\text{lm}}^* \text{uy}$ mit $1: x = 1: y$
- gilt $\beta = \gamma$
- Z.B. folgt aus $1: x = 1: y = \text{„id“}$, dass $\beta = \gamma = \text{„id} := \text{id“}$ (und analog für **if** und **while**)

• Beweis für Beispiel 2:

- Für $\text{STAT} \Rightarrow_{\text{lm}}^* \text{u STAT} \alpha \Rightarrow_{\text{lm}} \text{u id} := \text{id} \alpha \text{ [= u}\beta\alpha] \Rightarrow_{\text{lm}}^* \text{ux}$,
 $\text{STAT} \Rightarrow_{\text{lm}}^* \text{u STAT} \alpha \Rightarrow_{\text{lm}} \text{u id: STAT} \alpha \text{ [= u}\gamma\alpha] \Rightarrow_{\text{lm}}^* \text{uy}$,
 $\text{STAT} \Rightarrow_{\text{lm}}^* \text{u STAT} \alpha \Rightarrow_{\text{lm}} \text{u id (id)} \alpha \text{ [= u}\delta\alpha] \Rightarrow_{\text{lm}}^* \text{uz}$
- mit $1: x = 1: y = 1: z = \text{id}$
- gilt: β, γ und δ sind paarweise verschieden

• Beweis für Beispiel 3:

- Für $\text{STAT} \Rightarrow_{\text{lm}} \text{VAR} := \text{VAR} \Rightarrow_{\text{lm}} \text{id (IDLIST)} := \text{VAR} \Rightarrow_{\text{lm}}^* \text{id (id, id, ..., id)} := \text{id}$ und
 $\text{STAT} \Rightarrow_{\text{lm}} \text{id (IDLIST)} \Rightarrow_{\text{lm}}^* \text{id (id, id, ..., id)}$
mit $|\text{id (id, id, ..., id)}| \geq k$
- gilt: $k: (\text{id (id, id, ..., id)} := \text{id}) = k: (\text{id (id, id, ..., id)})$
- aber: „ $\text{VAR} := \text{VAR}$ “ $\neq$ „ id (IDLIST) “

Herstellung der LL(k)-Eigenschaft (durch Ausfaktorisierung)

- Ziel: zu gegebener kfG äquivalente LL(k)-Grammatik (mit demselben Sprachschatz)
- Technik „Ausfaktorisierung“: Zusammenfassung gemeinsamer Anfänge von Produktionen unter neuem Nichtterminal

Beispiel

- Betrachtet: kfG G_5' , die aus G_5 entsteht, wenn man
 - die Produktionen
$$\text{STAT} \rightarrow \dots \mid \text{VAR} := \text{VAR} \mid \text{id (IDLIST)}$$
ersetzt durch
$$\begin{aligned}\text{STAT} &\rightarrow \dots \mid \text{ZUWPROZ} \mid \text{id} := \text{VAR} \\ \text{ZUWPROZ} &\rightarrow \text{id (IDLIST) ZPREST} \\ \text{ZPREST} &\rightarrow := \text{VAR} \mid \varepsilon\end{aligned}$$
- Es gilt: G_5' ist LL(2)-Grammatik mit $L(G_5') = L(G_5)$

Es gilt aber

- LL(k)-Grammatiken sind eine echte Unterklasse der kontextfreien Grammatiken, d.h. es gibt kontextfreie Sprachen, für die es keine LL(k)-Grammatik gibt
- Beispiel: Sprache $\{a^n 0 b^n \mid n \geq 0\} \cup \{a^n 1 b^{2n} \mid n \geq 0\}$

Satz (Alternative Definition für LL(k))

*Intuitiv: Nächste für k
Eingabezeichen beseitigen
den Nicht-Determinismus*

- Sei $G = (V_N, V_T, P, S)$ kfG. G ist genau dann LL(k), wenn gilt
 - $A \rightarrow \beta, A \rightarrow \gamma \in P \wedge \beta \neq \gamma \Rightarrow \text{FIRST}_k(\beta\alpha) \cap \text{FIRST}_k(\gamma\alpha) = \emptyset$ für alle α mit $S \Rightarrow_{\text{lm}}^* uA\alpha$

Daraus

- Gute Kriterien für die Zugehörigkeit zu gewissen Teilklassen der LL(1)-Grammatiken
(da $\text{FIRST}_1(\beta\alpha) = \text{FIRST}_1(\beta) \oplus_1 \text{FOLLOW}_1(A)$ für alle α mit $S \Rightarrow_{\text{lm}}^* uA\alpha$)

Beachte:
Gilt nur für $k=1$, aber
nicht für beliebiges k

Verschiedene LL(1)-Kriterien

- Sei G beliebige kfG. G ist genau dann LL(1), wenn gilt (für alle $A \in V_N$)
 - $A \rightarrow \beta, A \rightarrow \gamma \in P \wedge \beta \neq \gamma \Rightarrow (\text{FIRST}_1(\beta) \oplus_1 \text{FOLLOW}_1(A)) \cap (\text{FIRST}_1(\gamma) \oplus_1 \text{FOLLOW}_1(A)) = \emptyset$
- Sei G ε -freie kfG (d.h. ohne Produktionen der Form $A \rightarrow \varepsilon$). G ist genau dann LL(1), wenn
 - Für jedes $A \in V_N$ mit $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$ gilt: $\text{FIRST}_1(\alpha_1), \dots, \text{FIRST}_1(\alpha_n)$ sind paarweise disjunkt
- Eine kfG G ist genau dann LL(1), wenn für alle Alternativen $A \rightarrow \alpha_1 \mid \dots \mid \alpha_n$ ($A \in V_N$) gilt
 - $\text{FIRST}_1(\alpha_1), \dots, \text{FIRST}_1(\alpha_n)$ sind paarweise disjunkt; höchstens ein $\text{FIRST}_1(\alpha_i)$ enthält ε
 - Aus $\alpha_i \Rightarrow_{\text{lm}}^* \varepsilon$ folgt: $\text{FIRST}_1(\alpha_j) \cap \text{FOLLOW}_1(A) = \emptyset$ für $1 \leq j \leq n, j \neq i$



Starke LL(k)-Grammatik

- Sei $G = (V_N, V_T, P, S)$ kfG
- G ist genau dann **starke LL(k)-Grammatik**, wenn gilt:
 - Sind $A \rightarrow \beta$ und $A \rightarrow \gamma$ verschiedene Produktionen aus P , dann ist
 - $(\text{FIRST}_k(\beta) \oplus_k \text{FOLLOW}_k(A)) \cap (\text{FIRST}_k(\gamma) \oplus_k \text{FOLLOW}_k(A)) = \emptyset$

Bemerkungen

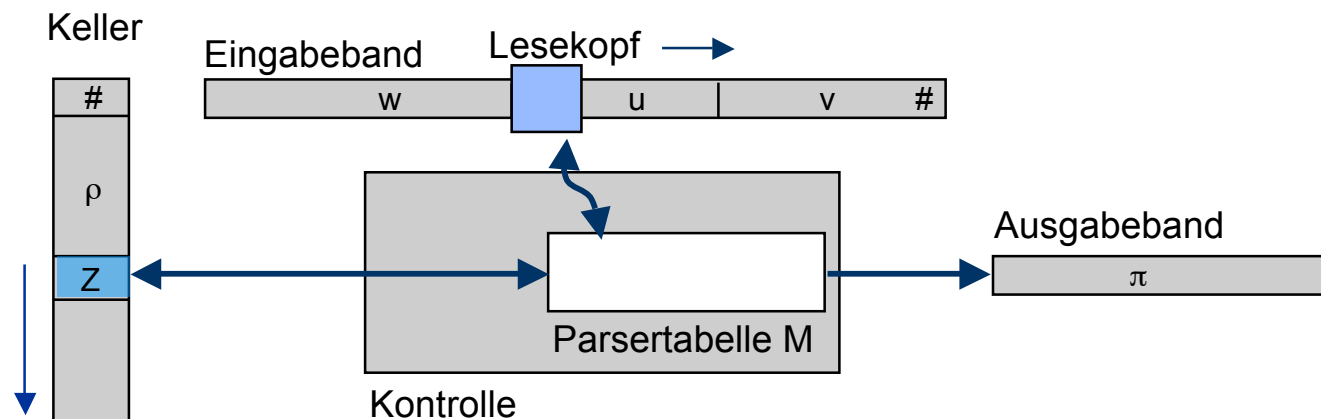
- Unterschied zu LL(k)
 - LL(k) berücksichtigt Ableitungen
 - Stark LL(k) berücksichtigt nur Produktionen (d.h. Kontext wird nicht berücksichtigt)
- Jede LL(1)-Grammatik ist stark
- Nicht jede LL(k)-Grammatik ist starke LL(k)-Grammatik
 - Beispiel: kfG $G = (\{S, A\}, \{a, b\}, \{S \rightarrow aAaa \mid bAba, A \rightarrow b \mid \varepsilon\}, S)$
 - G ist offensichtlich LL(2),
aber nicht stark LL(2)

Aber natürlich umgekehrt, d.h.
jede starke LL(k)-Grammatik
ist auch LL(k)

Für $A \rightarrow b$ und $A \rightarrow \varepsilon$ gilt:
 $(\text{FIRST}_2(b) \oplus_2 \text{FOLLOW}_2(A)) \cap (\text{FIRST}_2(\varepsilon) \oplus_2 \text{FOLLOW}_2(A)) =$
 $(b \oplus_2 \{aa, ba\}) \cap \{aa, ba\} =$
 $\{ba, bb\} \cap \{aa, ba\} = \{ba\} \neq \emptyset$

Struktur (eines Parser für (starke) LL(k)-Grammatiken)

- Aktueller Zustand Z legt Folgeaktion fest, d.h.
 - Nächstes Eingabesymbol lesen
 - Auf Ende der Analyse testen
 - Aktuelles Nichtterminal expandieren (anhand der Parser-Tabelle)



Im folgenden betrachtet

- Praktisch relevanter Fall: $k = 1$

Erzeugung der LL(1)-Parser-Tabelle

- Eingabe: LL(1)-Grammatik G , $FIRST_1$ und $FOLLOW_1$ für G
- Ausgabe: Parsertabelle M , die folgendermaßen aufgebaut wird

Algorithmus umfasst LL(1)-Test:
Wenn $M[X, a]$ im Laufe der
Besetzung von M zwei verschiedene
Einträge bekommt, ist die
Grammatik nicht LL(1)

- Für alle $X \rightarrow \alpha \in P$ und alle Terminalsymbole $a \in FIRST_1(\alpha)$: $M[X, a]$ auf $(X \rightarrow \alpha)$ setzen
- Falls $\text{eps}(\alpha) = \text{true}$, für alle $b \in FOLLOW_1(X)$: $M[X, b]$ auf $(X \rightarrow \alpha)$ setzen
- Alle übrigen Einträge von M auf *error* setzen

Beispiel

- Geg.: G_2 mit $P = \{S \rightarrow E, E \rightarrow TE', E' \rightarrow +E, E' \rightarrow \varepsilon, T \rightarrow FT', T' \rightarrow *T, T' \rightarrow \varepsilon, F \rightarrow (E), F \rightarrow \text{id}\}$
- $FIRST_1$ -Mengen: $FIRST_1(S) = FIRST_1(E) = FIRST_1(T) = FIRST_1(F) = \{ (, \text{id} \}$,
 $FIRST_1(E') = \{ +, \varepsilon \}$, $FIRST_1(T') = \{ *, \varepsilon \}$
- $FOLLOW_1$ -Mengen: $FOLLOW_1(S) = \{ \# \}$, $FOLLOW_1(E') = FOLLOW_1(E) = \{), \# \}$,
 $FOLLOW_1(T') = FOLLOW_1(T) = \{ +,), \# \}$, $FOLLOW_1(F) = \{ *, +,), \# \}$
- Parsertabelle M

	(	)	+	*	id	#
S	$(S \rightarrow E)$	error	error	error	$(S \rightarrow E)$	error
E	$(E \rightarrow TE')$	error	error	error	$(E \rightarrow TE')$	error
E'	error	$(E' \rightarrow \varepsilon)$	$(E' \rightarrow +E)$	error	error	$(E' \rightarrow \varepsilon)$
T	$(T \rightarrow FT')$	error	error	error	$(T \rightarrow FT')$	error
T'	error	$(T' \rightarrow \varepsilon)$	$(T' \rightarrow \varepsilon)$	$(T' \rightarrow *T)$	error	$(T' \rightarrow \varepsilon)$
F	$(F \rightarrow (E))$	error	error	error	$(F \rightarrow \text{id})$	error

Kellerinhalt

```
# [S → .E]
# [S → .E] [E → .TE']
# [S → .E] [E → .TE'] [T → .FT']
# [S → .E] [E → .TE'] [T → .FT'] [F → .id]
# [S → .E] [E → .TE'] [T → .FT'] [F → id.]
# [S → .E] [E → .TE'] [T → F.T']
# [S → .E] [E → .TE'] [T → F.T'] [T' → .*T]
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T]
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → .FT']
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → .FT'] [F → .id]
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → .FT'] [F → id.]
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → F.T']
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → F.T'] [T' → ε.]
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T] [T → FT']
# [S → .E] [E → .TE'] [T → F.T'] [T' → *.T]
# [S → .E] [E → .TE'] [T → FT']
# [S → .E] [E → T.E']
# [S → .E] [E → T.E'] [E' → ε.]
# [S → .E] [E → TE']
# [S → E.]
#
```

Eingabe

```
id * id #
id * id #
id * id #
id * id #
* id #
* id #
* id #
id #
id #
id #
#
#
#
#
#
#
#
#
#
#
```

Beispiel

Parserlauf für **id * id #**

Ausgabe

```
[S → E]
[E → TE']
[T → FT']
[F → id]
[T' → *T]
[T → FT']
[F → id]
[T' → ε]
[E' → ε]
```

	(	)	+	*	id	#
S	(S → E)	error	error	error	(S → E)	error
E	(E → TE')	error	error	error	(E → TE')	error
E'	error	(E' → ε)	(E' → +E)	error	error	(E' → ε)
T	(T → FT')	error	error	error	(T → FT')	error
T'	error	(T' → ε)	(T' → ε)	(T' → *T)	error	(T' → ε)
F	(F → (E))	error	error	error	(F → id)	error

Parsertabelle

Linksrekursion

- Sei $G = (V_N, V_T, P, S)$ kfG, $p \in P$, $A \in V_N$
 - p ist **direkt rekursiv** $\Leftrightarrow p$ hat die Form $A \rightarrow \alpha A \beta$
 - p ist **direkt linksrekursiv** (bzw. **rechtsrekursiv**) $\Leftrightarrow p$ hat die Form $A \rightarrow A \beta$ (bzw. $A \rightarrow \alpha A$)
 - A ist rekursiv $\Leftrightarrow$ es gibt Ableitung $A \Rightarrow^+ \alpha A \beta$
 - A ist **linksrekursiv** (bzw. **rechtsrekursiv**) $\Leftrightarrow$ es gibt Ableitung $A \Rightarrow^+ A \beta$ (bzw. $A \Rightarrow^+ \alpha A$)
 - G ist **linksrekursiv** $\Leftrightarrow G$ enthält mindestens 1 linksrekursives Nichtterminal

Satz

- Sei G kfG
 - G ist linksrekursiv $\Rightarrow G$ ist nicht LL(k) für jedes k
 - G ist LL(k)-Grammatik $\Rightarrow G$ ist nicht mehrdeutig

Elimination von Linksrekursion

- Transformation der Grammatik in eine, die nicht linksrekursiv und LL(k) ist
- Transformation ist immer möglich, hat aber Nachteile
 - Größe der Grammatik wächst stark
 - Struktur der Grammatik wird stark verändert

z.B. Rechtsrekursion
statt Linksrekursion

Beispiel

- Geg.: Varianten der Ausdrucksgrammatik
 - G_0 mit $P = \{E \rightarrow E+T \mid T, T \rightarrow T * F \mid F, F \rightarrow (E) \mid \text{id}\}$
 - G_2' mit $P = \{E \rightarrow TE', E' \rightarrow +E \mid \varepsilon, T \rightarrow FT', T' \rightarrow *T \mid \varepsilon, F \rightarrow (E) \mid \text{id}\}$
- Syntaxbäume für **id + id**

3 Nichtterminale
6 Produktionen

5 Nichtterminale
8 Produktionen

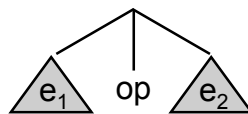
(für G_0)



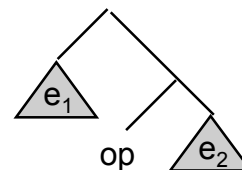
(für G_2')



Schematische Baumstrukturen (aus dem Beispiel)



(a)



(b)

- Dabei deutlich: Struktur (b) zur Behandlung der Semantik ungünstiger

Alternativen (zur Behandlung der Linksrekursion)

- Bäume auf wesentliche Struktur reduzieren („abstrakte Syntax“)
- Reguläre Ausdrücke auf rechten Seiten der Produktionen (*rechtsreguläre kfG* ➡)

Rechtsreguläre kontextfreie Grammatik (kurz: rrkfG)

- Idee: Reguläre Ausdrücke als Ersatz für Linksrekursion — z.B. in EBNF verwendet
- Definition: **rrkfG** $G = (V_N, V_T, p, S)$, wobei
 - V_N, V_T, S wie bei kfG
 - $p: V_N \rightarrow RA$ (= Menge der regulären Ausdrücke über $V_N \cup V_T$) — **Beachte:** p ist Funktion !

Beispiel (rrkfG für arithmetische Ausdrücke)

- $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, \text{id}\}, \{S \rightarrow E, E \rightarrow T((+|-)T)^*, T \rightarrow F((*/|) F)^*, F \rightarrow (\langle E \rangle | \text{id})\}, S)$

Begriffe

Als Terminalsymbole werden hier zur besseren Lesbarkeit die Klammern $\langle$ und $\rangle$ verwendet

- Sei G rrkfG
 - Ableitungsrelation** $\Rightarrow_{lm}$ auf RA („leitet regulär direkt links ab“)
 - (a) $wX\beta \Rightarrow_{lm} w\alpha\beta$ (mit $\alpha = p(X)$)
 - (b) $w(r_1 | \dots | r_n)\beta \Rightarrow_{lm} w r_i\beta$ (für $1 \leq i \leq n$)
 - (c) $w(r)^*\beta \Rightarrow_{lm} w\beta$
 - (d) $w(r)^*\beta \Rightarrow_{lm} wr(r)^*\beta$
 - Definierte Sprache:** $L(G) =_{\text{def}} \{w \in V_T^* \mid S \Rightarrow_{lm}^* w\}$
(wobei $\Rightarrow_{lm}^*$ reflexiv-transitive Hülle von $\Rightarrow_{lm}$)

Beispiel

- Geg.: $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, \text{id}\}, \{S \rightarrow E, E \rightarrow T((+|-)T)^*, T \rightarrow F((*/|) F)^*, F \rightarrow (\langle E \rangle \text{id})\}, S)$
- Reguläre Linksableitung zum Wort **id + id * id**

$$\begin{aligned}
 & S \xRightarrow{\text{lm}} E \xRightarrow{\text{lm}} T((+|-)T)^* \xRightarrow{\text{lm}} F((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} (\langle E \rangle \text{id}) ((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} \text{id} ((*/|) F)^*((+|-)T)^* \\
 & \xRightarrow{\text{lm}} \text{id} ((+|-)T)^* \xRightarrow{\text{lm}} \text{id} (+|-)T((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + T((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + F((*/|) F)^*((+|-)T)^* \\
 & \xRightarrow{\text{lm}} \text{id} + (\langle E \rangle \text{id}) ((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + \text{id} ((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + \text{id} (*|/) F ((*/|) F)^*((+|-)T)^* \\
 & \xRightarrow{\text{lm}} \text{id} + \text{id} * F((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + \text{id} * (\langle E \rangle \text{id}) ((*/|) F)^*((+|-)T)^* \\
 & \xRightarrow{\text{lm}} \text{id} + \text{id} * \text{id} ((*/|) F)^*((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + \text{id} * \text{id} ((+|-)T)^* \xRightarrow{\text{lm}} \text{id} + \text{id} * \text{id}
 \end{aligned}$$

Beobachtung

- Expansion (= Fall (a)) nicht mehr kritisch, stattdessen aber die Fälle (b), (c) und (d)

RLL(1)-Parser

- Parser für rrkfG, der bei Vorliegen der regulären Linkssatzform
 - $w(r_1 \mid \dots \mid r_n)\beta$ Entscheidung für richtige Alternative, und bei
 - $w(r)^*\beta$ Entscheidung für Fortsetzung bzw. Beendigung der Iteration mithilfe des nächsten Symbols der Resteingabe treffen kann

Übertragung einiger Begriffe auf rrkfG

- Seien r, r_i ($1 \leq i \leq n$) reguläre Ausdrücke

- **Direkter regulärer Unterausdruck**

- r_i : von $(r_1 \mid \dots \mid r_n)$ und $(r_1 \dots r_n)$
 - r : von $(r)^*$ und r selbst

- r_1 ist **regulärer Unterausdruck** von r_2 , wenn

- r_1 direkter regulärer Unterausdruck von r_2
 - r_1 regulärer Unterausdruck eines direkt regulären Unterausdrucks von r_2

- Sei $G = (V_N, V_T, p, S)$ rrkfG

- **Erweitertes kontextfreies Item** $[X \rightarrow \beta.\alpha\gamma]$, wobei

- $X \in V_N$ mit $p(X) = \beta\alpha\gamma$
 - $\alpha, \beta, \gamma \in (V_N \cup V_T \cup \{ (,), *, |, \epsilon \})^*$
 - α ist regulärer Unterausdruck von $\beta\alpha\gamma$

Ist erforderlich, da die rechte Seite einer Produktion neben relevanten Symbolen auch Meta-Symbole der regulären Ausdrücke enthält

Intuitiv:

Punkt nicht vor beliebigen Symbol(folg)en, sondern nur vor regulären Unterausdrücken

Beispiel

- Geg.: Regulärer Ausdruck $\text{id} + \text{id} * F ((*|/) F)^* ((+|-)T)^*$

- Direkt reguläre Unterausdrücke: $\text{id}, +, *, F, ((*|/) F)^*, ((+|-)T)^*$

- Reguläre Unterausdrücke: $(*|/) F, (*|/), F, *, /, (+|-)T, (+|-), T, +, -, \dots$

- Erweiterte kontextfreie Items von $E \rightarrow T((+|-)T)^*$:

$E \rightarrow .T((+|-)T)^*, E \rightarrow T.(+|-)T^*, E \rightarrow T.(+|-)T^*, E \rightarrow T((+|-)T)^*, E \rightarrow T((+|-)T)^*, E \rightarrow T((+|-)T)^*, E \rightarrow T((+|-)T)^*,$
 $E \rightarrow T((+|-)T)^*.$

Keine direkt regulären Unterausdrücke sind z.B. $* F, + \text{id} * F, (+|-)$

Keine regulären Unterausdrücke sind z.B. $F ((*|/) F, ((+|$

FIRST₁- und FOLLOW₁-Berechnung für rrkfG

- Wie früher
 - Berechnung der ε -Produktivität
 - Berechnung der ε -freien first-Funktion
- **ε -Produktivität** eps
 - $\text{eps}(\varepsilon) =_{\text{def}} \text{true}$
 - $\text{eps}(a) =_{\text{def}} \text{false}$ für $a \in V_T$
 - $\text{eps}(X) =_{\text{def}} \text{eps}(r)$ falls $p(X) = r$ für $X \in V_N$
 - $\text{eps}(r^*) =_{\text{def}} \text{true}$
 - $\text{eps}((r_1 \mid \dots \mid r_n)) =_{\text{def}} \bigvee_{i=1..n} \text{eps}(r_i)$
 - $\text{eps}((r_1 \dots r_n)) =_{\text{def}} \bigwedge_{i=1..n} \text{eps}(r_i)$
- **ε -freie FIRST-Funktion** $\varepsilon\text{-ffi}$
 - $\varepsilon\text{-ffi}(\varepsilon) =_{\text{def}} \emptyset$
 - $\varepsilon\text{-ffi}(a) =_{\text{def}} \{a\}$ für $a \in V_T$
 - $\varepsilon\text{-ffi}(X) =_{\text{def}} \varepsilon\text{-ffi}(r)$, falls $p(X) = r$ für $X \in V_N$
 - $\varepsilon\text{-ffi}(r^*) =_{\text{def}} \varepsilon\text{-ffi}(r)$
 - $\varepsilon\text{-ffi}((r_1 \mid \dots \mid r_n)) =_{\text{def}} \bigcup_{1 \leq i \leq n} \varepsilon\text{-ffi}(r_i)$
 - $\varepsilon\text{-ffi}((r_1 \dots r_n)) =_{\text{def}} \bigcup_{1 \leq j \leq n} \{x \in \varepsilon\text{-ffi}(r_j) \mid \bigwedge_{1 \leq i < j} \text{eps}(r_i)\}$

Beispiel

- Geg.: $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, \text{id}\}, p, S)$
mit $p = \{S \rightarrow E, E \rightarrow T((+|-)T)^*,$
 $T \rightarrow F((*/|) F)^*, F \rightarrow (\langle E \rangle \mid \text{id})\}$
- Es gilt:
 - $\text{eps}(X) =_{\text{def}} \text{false}$ für alle $X \in V_N$
 - $\varepsilon\text{-ffi-Mengen} = \text{FIRST}_1\text{-Mengen}$
 $FI_1(S) = FI_1(E) = FI_1(T) = FI_1(F) = \{\langle, \text{id}\}$

Unterschied FIRST₁- und FOLLOW₁-Mengen regulärer (Unter-)Ausdrücke

- FIRST₁-Mengen unabhängig vom Kontext
- FOLLOW₁-Mengen i.a. verschieden für verschiedene Vorkommen
(in erweiterten regulären Items)
 - (1) $FO_1([S' \rightarrow .S]) =_{\text{def}} \{\#\}$
 - (2) $FO_1([X \rightarrow \beta (r_1 | \dots | r_i | \dots | r_n) \gamma]) =_{\text{def}} FO_1([X \rightarrow \beta .(r_1 | \dots | r_i | \dots | r_n) \gamma])$ für $1 \leq i \leq n$
 - (3) $FO_1([X \rightarrow \beta (r_1 \dots r_i r_{i+1} \dots r_n) \gamma]) =_{\text{def}} \varepsilon\text{-ffi}(r_{i+1})$, falls $\text{eps}(r_{i+1}) = \text{false}$
 - (4) $FO_1([X \rightarrow \beta (r_1 \dots r_i r_{i+1} \dots r_n) \gamma]) =_{\text{def}} \varepsilon\text{-ffi}(r_{i+1}) \cup FO_1([X \rightarrow \beta (r_1 \dots r_i . r_{i+1} \dots r_n) \gamma])$, sonst
 - (5) $FO_1([X \rightarrow \beta (r_1 \dots r_{n-1} . r_n) \gamma]) =_{\text{def}} FO_1([X \rightarrow \beta .(r_1 \dots r_{n-1} r_n) \gamma])$
 - (6) $FO_1([X \rightarrow \beta (.r)^* \gamma]) =_{\text{def}} \varepsilon\text{-ffi}(r) \cup FO_1([X \rightarrow \beta .(r)^* \gamma])$
 - (7) $FO_1([X \rightarrow .r]) =_{\text{def}} \bigcup_{Y \in V_N} FO_1([Y \rightarrow \beta .X \gamma])$

Beispiel

Nur zur Verdeutlichung

- Geg.: $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, \text{id}\}, \{S \rightarrow E, E \rightarrow (T((+|-)T)^*), T \rightarrow (F((*)/ F)^*), F \rightarrow (\langle E \rangle | \text{id})\}, S)$
 - $FO_1([S \rightarrow .E]) = \{\#\}$
 - $FO_1([E \rightarrow (T((+|-)T)^*)]) = \text{[(5)]}$
 - $FO_1([E \rightarrow .(T((+|-)T)^*)]) = \text{[(7)]}$
 - $FO_1([S \rightarrow .E]) \cup FO_1([F \rightarrow \langle .E \rangle]) = \text{[(1),(3)]}$
 - $\{\#\} \cup \{\rangle\} = \{\#, \rangle\}$
 - $FO_1([T \rightarrow (F((*)/ F)^*)]) = \text{[(5)]}$
 - $FO_1([T \rightarrow .(F((*)/ F)^*)]) = \text{[(7)]}$
 - $FO_1([E \rightarrow (T((+|-)T)^*)]) \cup FO_1([E \rightarrow T((+|-).T)^*]) = \text{[(4), li., (5)]}$
 - $\{+, -\} \cup \{\#, \rangle\} \cup FO_1([E \rightarrow T((+|-)T)^*]) = \text{[(6)]}$
 - $\{\#, \rangle\} \cup \{+, -\} \cup FO_1([E \rightarrow (T((+|-)T)^*)]) = \{\#, \rangle, +, -\}$

RLL(1)-Grammatik

- Eine rrkfG $G = (V_N, V_T, p, S)$ ist **RLL(1)-Grammatik**, wenn gilt:
 - Für alle erweiterten kontextfreien Items $[X \rightarrow \beta \cdot (r_1 | \dots | r_i | \dots | r_n) \gamma]$ und $i \neq j$:
 $(FI_1(r_i) \oplus_1 FO_1([X \rightarrow \beta \cdot (r_1 | \dots | r_i | \dots | r_n) \gamma])) \cap (FI_1(r_j) \oplus_1 FO_1([X \rightarrow \beta \cdot (r_1 | \dots | r_i | \dots | r_n) \gamma])) = \emptyset$
 - Für alle erweiterten kontextfreien Items $[X \rightarrow \beta \cdot (r)^* \gamma]$:
 $FI_1(r) \cap FO_1([X \rightarrow \beta \cdot (r)^* \gamma]) = \emptyset$ und $\text{eps}(r) = \text{false}$

Eindeutige
Alternativen-
auswahl

Abbruch oder Fortsetzung
der Iteration eindeutig

Beispiel

- Geg.: $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, \text{id}\}, \{S \rightarrow E, E \rightarrow T((+|-)T)^*, T \rightarrow F((*/|) F)^*, F \rightarrow (\langle E \rangle | \text{id})\}, S)$
- G_e ist RLL(1):
 - $(FI_1(+) \oplus_1 FO_1([E \rightarrow T \cdot (+|-)T]^*)) \cap (FI_1(-) \oplus_1 FO_1([E \rightarrow T \cdot (+|-)T]^*)) = \{+\} \cap \{-\} = \emptyset$
 - $(FI_1(*) \oplus_1 FO_1([T \rightarrow F \cdot (*|/) F]^*)) \cap (FI_1(/) \oplus_1 FO_1([T \rightarrow F \cdot (*|/) F]^*)) = \{*\} \cap \{/ \} = \emptyset$
 - $(FI_1(\langle) \oplus_1 FO_1([F \rightarrow \cdot (\langle E \rangle | \text{id})])) \cap (FI_1(\text{id}) \oplus_1 FO_1([F \rightarrow \cdot (\langle E \rangle | \text{id})])) = \{\langle\} \cap \{\text{id}\} = \emptyset$
 - $FI_1((+|-)T) \cap FO_1([E \rightarrow T \cdot ((+|-)T)^*]) = \{+, -\} \cap \{\#, \rangle\} = \emptyset$ und $\text{eps}((+|-)T) = \text{false}$
 - $FI_1((*/|) F) \cap FO_1([T \rightarrow F \cdot ((*|/) F)^*]) = \{*, /\} \cap \{\#, \rangle, +, -\} = \emptyset$ und $\text{eps}((*/|) F) = \text{false}$



Beispiel (Anweisungs-Grammatik)

- Geg.: Grammatik G mit

Anw	→	If_Anw While_Anw Repeat_Anw Proz_Aufruf Wertzuweisung
If_Anw	→	if Bed then An_Folge {fi else An_Folge fi}
While_Anw	→	while Bed do An_Folge od
Repeat_Anw	→	repeat An_Folge until Bed
Proz_Aufruf	→	call name (Ausdr_Folge)
Wertzuweisung	→	name := Ausdr
Ausdr_Folge	→	Ausdr (, Ausdr)*
An_Folge	→	Anw (; Anw)*
Progr	→	An_Folge

- Relevante FIRST₁- und FOLLOW₁-Mengen

$FI_1(\text{If_Anw}) = \{\mathbf{if}\}$, $FI_1(\text{While_Anw}) = \{\mathbf{while}\}$, $FI_1(\text{Repeat_Anw}) = \{\mathbf{repeat}\}$, $FI_1(\text{Proz_Aufruf}) = \{\mathbf{call}\}$

$FI_1(\text{Wertzuweisung}) = \{\mathbf{name}\}$, $FI_1(\text{Ausdr_Folge}) = \{\mathbf{Ausdr}\}$

$FI_1(\text{Progr}) = FI_1(\text{An_Folge}) = FI_1(\text{Anw}) = \{\mathbf{if, while, repeat, call, name}\}$

$FO_1([\text{Ausdr_Folge} \rightarrow \mathbf{Ausdr}.(, \mathbf{Ausdr})^*]) = \{\}$, $FO_1([\text{An_Folge} \rightarrow \text{Anw}.(; \text{Anw})^*]) = \{\mathbf{fi, else, od, until, \#}\}$

- G ist RLL(1)

1) Da kein Nichtterminal ϵ produziert, zu zeigen: FI_1 für verschiedene Alternativen disjunkt (offensichtlich!)

2) a) $FI_1(, \mathbf{Ausdr}) \cap FO_1([\text{Ausdr_Folge} \rightarrow \mathbf{Ausdr}.(, \mathbf{Ausdr})^*]) = \{, \} \cap \{\}$ = $\emptyset$ und $\text{eps}(, \mathbf{Ausdr}) = \text{false}$

b) $FI_1(; \text{Anw}) \cap FO_1([\text{An_Folge} \rightarrow \text{Anw}.(; \text{Anw})^*]) = \{;\} \cap \{\mathbf{fi, else, od, until, \#}\} = \emptyset$

und $\text{eps}(, \text{Anw}) = \text{false}$



RLL(1)-Parser für rrkfG (Tabellenversion)

• Parsertabelle M

- Repräsentiert partielle Abbildung $m: It_G \times V_T \rightarrow It_G \cup \{\text{error}\}$
- Enthält nur Zeilen für Items, in denen
 - Eine Alternative ausgewählt werden muss, d.h. Form $[X \rightarrow \dots(r_1 \mid \dots \mid r_n)\dots]$
 - Eine Iteration bearbeitet werden muss, d.h. Form $[X \rightarrow \dots(r)^*\dots]$

• Arbeitsweise

- Anfangskonfiguration: $([S' \rightarrow \cdot S], w\#)$
- Oberstes Item ρ legt fest, ob Tabelle konsultiert werden muss; falls ja
 - $M[\rho, a] \neq \text{error}$: Folge-Item (gemäß Tabelle)
 - $M[\rho, a] = \text{error}$: Syntaxfehler
- Endkonfiguration: $([S' \rightarrow S \cdot], \#)$
- Sonstige Übergänge
 - $\delta([X \rightarrow \dots a \dots], a) = [X \rightarrow \dots a \dots]$ [Lesen]
 - $\delta([X \rightarrow \dots Y \dots], \epsilon) = [X \rightarrow \dots Y \dots] [Y \rightarrow \cdot p(Y)]$ [Expansion]
 - $\delta([X \rightarrow \dots Y \dots] [Y \rightarrow p(Y) \cdot], \epsilon) = [X \rightarrow \dots Y \dots]$ [Reduktion]
- „Korrekturen“ und Optimierung durch Modifikation der Übergangstabelle, z.B.

- $[X \rightarrow \dots(\dots \mid r_i \mid \dots)\dots] \Rightarrow [X \rightarrow \dots(\dots \mid r_i \mid \dots) \cdot \dots]$
 - $[X \rightarrow \dots (r) \cdot \dots] \Rightarrow [X \rightarrow \dots (r)^* \dots]$
 - $[X \rightarrow \dots (r_1 \dots r_n) \dots] \Rightarrow [X \rightarrow \dots (r_1 \dots r_n) \cdot \dots]$
- } Falls bei Lesen/Reduktion „illegale Items“ entstehen
— Optimierung

Erzeugung der RLL(1)-Parsertabelle

- Geg.: RLL(1)-Grammatik G , $FIRST_1$ und $FOLLOW_1$ für G
- Vorgehensweise
 - Für alle Items der Form $[X \rightarrow \dots(r_1 \mid \dots \mid r_n)\dots]$ setze
 - $M([X \rightarrow \dots(r_1 \mid \dots \mid r_n)\dots], a) = [X \rightarrow \dots(\dots \mid .r_i \mid \dots)]$ für $a \in FIRST_1(r_i)$ falls $\epsilon \notin FIRST_1(r_i)$
 - $M([X \rightarrow \dots(r_1 \mid \dots \mid r_n)\dots], a) = [X \rightarrow \dots(\dots \mid .r_i \mid \dots)]$ für $a \in FIRST_1(r_i) \cup FOLLOW_1([X \rightarrow \dots(r_1 \mid \dots \mid r_n)\dots])$, sonst
 - Für alle Items der Form $[X \rightarrow \dots(r)^*\dots]$ setze
 - $M([X \rightarrow \dots(r)^*\dots], a) = [X \rightarrow \dots(r)^*\dots]$ falls $a \in FIRST_1(r)$
 - $M([X \rightarrow \dots(r)^*\dots], a) = [X \rightarrow \dots(r)^*\dots]$ falls $a \in FOLLOW_1([X \rightarrow \dots(r)^*\dots])$
 - Setze alle noch nicht gefüllten Einträge auf *error*

Beispiel

- Geg.: $G_e = (\{S, E, T, F\}, \{+, -, *, /, \langle, \rangle, id\}, \{S \rightarrow E, E \rightarrow T((+|-)T)^*, T \rightarrow F((*/|)F)^*, F \rightarrow (\langle E \rangle | id)\}, S)$
- Es war: $FI_1(S) = FI_1(E) = FI_1(T) = FI_1(F) = \{\langle, id\}$, $FO_1([E \rightarrow T((+|-)T)^*]) = \{\#, \rangle\}$, $FO_1([T \rightarrow F((*/|)F)^*]) = \{\#, \rangle, +, -\}$
- Parsertabelle

Beachte: Komprimierung, d.h. direkter Übergang von $[E \rightarrow T((+|-)T)^*]$ in $[E \rightarrow T((+|-)T)^*]$ unter $+$; analog für $-$, $*$ und $/$

	+	-	#	⟩	*	/	⟨	id
$[E \rightarrow T((+ -)T)^*]$	$[E \rightarrow T((+ -)T)^*]$	$[E \rightarrow T((+ -)T)^*]$	$[E \rightarrow T((+ -)T)^*]$	$[E \rightarrow T((+ -)T)^*]$	error	error	error	error
$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	$[T \rightarrow F((*/)F)^*]$	error	error
$[F \rightarrow \langle E \rangle id]$	error	error	error	error	error	error	$[F \rightarrow \langle E \rangle id]$	$[F \rightarrow \langle E \rangle id]$

Recursive descent RLL(1)-Parser

- Programm, das sich direkt aus einer RLL(1)-Grammatik und ihren $FIRST_1$ - und $FOLLOW_1$ -Mengen ableiten lässt
- Geg.: (V_N, V_T, p, S) mit $V_N = \{X_0, \dots, X_n\}$, $S = X_0$, $p = \{X_0 \rightarrow \alpha_0, \dots, X_n \rightarrow \alpha_n\}$
- Außerdem sei $FiFo([X \rightarrow \dots \beta \dots]) =_{\text{def}} FI_1(\beta) \oplus_1 FO_1([X \rightarrow \dots \beta \dots])$

Hauptprogramm eines recursive descent RLL(1)-Parsers

```

program parser;
  var nextsymbol: symbol;
  proc scan; (* liest nächstes Eingabesymbol in nextsym *)
  proc error (meldung: string); (* gibt Fehlermeldung aus und stoppt Parserlauf *)
  proc accept; (* meldet Ende der Analyse und stoppt Parserlauf *)
  proc X0; begin progr([X0 → .α0]) end;
  proc X1; begin progr([X1 → .α1]) end; ....
  proc Xn; begin progr([Xn → .αn]) end;
begin
  scan;
  X0;
  if nextsym = „#“ then accept else error (...) fi
end
  
```

*Deklaration von Prozeduren;
je eine pro Nichtterminal*

*Eigentliches
Hauptprogramm*

Behandlung der regulären Ausdrücke

- Nichtterminale $Y \in V_N$
 - $\text{progr}([X \rightarrow \dots Y \dots]) =_{\text{def}} Y$
- Terminale $a \in V_T$
 - $\text{progr}([X \rightarrow \dots a \dots]) =_{\text{def}} \text{if nextsym} = a \text{ then scan else error(„a erwartet“) fi}$
- Komposition
 - $\text{progr}([X \rightarrow \dots (\alpha_1 \alpha_2 \dots \alpha_k) \dots]) =_{\text{def}} \text{progr}([X \rightarrow \dots (\alpha_1 \alpha_2 \dots \alpha_k) \dots]); \text{progr}([X \rightarrow \dots (\alpha_1 \alpha_2 \dots \alpha_k) \dots]); \dots \text{progr}([X \rightarrow \dots (\alpha_1 \alpha_2 \dots \alpha_k) \dots]);$
- Alternative
 - $\text{progr}([X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]) =_{\text{def}}$
case nextsym **in**
 FiFo($[X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]$): $\text{progr}([X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]);$
 FiFo($[X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]$): $\text{progr}([X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]); \dots$
 FiFo($[X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots]$): $\text{progr}([X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots])$
 otherwise $\text{progr}([X \rightarrow \dots (\alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_{k-1} \mid \alpha_k) \dots])$ **endcase**
- Stern und Plus
 - $\text{progr}([X \rightarrow \dots (\alpha)^* \dots]) =_{\text{def}} \text{while nextsym in } Fl_1(\alpha) \text{ do } \text{progr}([X \rightarrow \dots \alpha \dots]) \text{ od}$
 - $\text{progr}([X \rightarrow \dots (\alpha)^+ \dots]) =_{\text{def}} \text{repeat } \text{progr}([X \rightarrow \dots \alpha \dots]) \text{ until nextsym not in } Fl_1(\alpha)$
- Leeres Wort
 - $\text{progr}([X \rightarrow \dots \varepsilon \dots]) =_{\text{def}} ;$

Beispiel

- Geg.: $E \rightarrow T((+|-)T)^*$ aus G_e
- Es gilt:

`progr([E → .(T((+|-)T)*])`

= [Komposition]

`progr([E → .T((+|-)T)*]); progr([E → T.((+|-)T)*])`

= [Nichtterminal; Stern]

`T; while nextsym in Fi1((+|-)T) do progr([E → T.((+|-)T)]) od`

= [Def. Fi₁; Komposition]

`T; while nextsym in {„+“, „-“} do progr([E → T.(+|-)T]); progr([E → T((+|-).T)]) od`

= [Alternative; Nichtterminal]

`T; while nextsym in {„+“, „-“} do`

`case nextsym in FiFo([E → T((+|-)T)]) : progr([E → T((+|-)T)]);`

`otherwise progr([E → T((+|-).T)]) endcase; T od`

= [Def. FiFo; Terminal]

`T; while nextsym in {„+“, „-“} do`

`case nextsym in {„+“} : if nextsym = „+“ then scan else error(„+ erwartet“) fi;`

`otherwise if nextsym = „-“ then scan else error(„- erwartet“) fi endcase; T od`



Beispiel (Recursive descent Parser für G_e)

```
program parser;  
  var nextsym: symbol;  
  proc scan; (* liest nächstes Eingabesymbol in nextsym *)  
  proc error (meldung: string); (* gibt Fehlermeldung aus und stoppt Parserlauf *)  
  proc accept; (* meldet Ende der Analyse und stoppt Parserlauf *)  
  proc S; begin E end;  
  proc E;  
    begin T; while nextsym in {„+“, „-“} do  
      case nextsym in {„+“} : if nextsym = „+“ then scan else error(„+ erwartet“) fi;  
      otherwise if nextsym = „-“ then scan else error(„- erwartet“) fi endcase; T od end;  
  proc T;  
    begin F; while nextsym in {„*“, „/“} do  
      case nextsym in {„*“} : if nextsym = „*“ then scan else error(„* erwartet“) fi;  
      otherwise if nextsym = „/“ then scan else error(„/ erwartet“) fi endcase; F od end;  
  proc F;  
    begin case nextsym in {„<“}: E; if nextsym = „>“ then scan else error(„> erwartet“) fi;  
      otherwise if nextsym = „id“ then scan else error(„id erwartet“) fi endcase end;  
begin  
  scan;  
  S;  
  if nextsym = „#“ then accept else error(„kein korrekter Ausdruck“) fi  
end
```





Recursive-descent-Parser für Anweisungs-Grammatik

```
program parser;  
  var nextsym: symbol ;  
  
  proc scan;  
  proc error;  
  proc accept;  
  
  proc An_Folge;  
    begin  
      Anw;  
      while nextsym in {";" } do  
        if nextsym = ";" then scan else error fi;  
        Anw od  
      end;  
  
  proc Anw;  
    begin  
      case nextsym in  
        {"if"}: If_Anw;  
        {"while"}: While_Anw;  
        {"repeat"}: Repeat_Anw;  
        {"call"}: Proz_Aufruf;  
        otherwise Wertzuweisung  
      endcase  
    end;  
end;
```

```
proc If_Anw;  
  begin  
    if nextsym = "if" then scan else error fi;  
    if nextsym = "Bed" then scan else error fi;  
    if nextsym = "then" then scan else error fi;  
    An_Folge;  
    case nextsym in  
      {"fi"}: if nextsym = "fi" then scan else error fi;  
      otherwise if nextsym = "else" then scan else error fi;  
                An_Folge;  
                if nextsym = "fi" then scan else error fi  
    endcase  
  end;  
  
proc While_Anw;  
  begin  
    if nextsym = "while" then scan else error fi;  
    if nextsym = "Bed" then scan else error fi;  
    if nextsym = "do" then scan else error fi;  
    An_folge;  
    if nextsym = "od" then scan else error fi  
  end;
```



RLL(1)-Parser für Anweisungs-Grammatik (Fortsetzung)

```
proc Repeat_Anw;  
  begin  
    if nextsym = "repeat" then scan else error fi;  
    An_Folge;  
    if nextsym = "until" then scan else error fi;  
    if nextsym = "Bed" then scan else error fi  
  end;  
proc Proz_Aufruf;  
  begin  
    if nextsym = "call" then scan else error fi;  
    if nextsym = "name" then scan else error fi;  
    if nextsym = "(" then scan else error fi;  
    Ausdr_Folge;  
    if nextsym = ")" then scan else error fi  
  end;  
proc Wertzuweisung;  
  begin  
    if nextsym = "name" then scan else error fi;  
    if nextsym = "[:=" then scan else error fi;  
    if nextsym = "Ausdr" then scan else error fi  
  end;
```

```
proc Ausdr_Folge;  
  begin  
    if nextsym = "Ausdr" then scan else error fi;  
    while nextsym in {"", ","} do  
      if nextsym = "," then scan else error fi;  
      if nextsym = "Ausdr" then scan else error fi  
    od  
  end  
  
  begin  
    scan;  
    An_Folge;  
    if nextsym = "#" then accept else error fi  
  end
```

Arten von Fehlern (allgemein)

- Lexikalische Fehler (z.B. Tippfehler, Verwendung unzulässiger Zeichen, ...)
- Syntaktische Fehler (z.B. falsche Klammerung, fehlendes Schlüsselwort, ...)
- Fehler in der statischen Semantik (z.B. Typfehler, Deklariertheitsfehler, ...)

*Im Weiteren
betrachtet*

Erwünschte Reaktionen eines Parsers

- Fehler melden und lokalisieren (zum frühest möglichen Zeitpunkt)
 - Probleme
 - „Folgefehler“
 - meist nur Symptom erkennbar, nicht Fehler selbst
- Fehler diagnostizieren
 - Diagnose des Symptoms (Stelle im Programm, Parserkonfiguration)
- Fehler korrigieren
 - Meist nur lokal (Einsetzungen, Ersetzungen);
Globale Korrektur zu teuer
- Wieder „Tritt fassen“

Aho, Sethi, Ullman: „Abgesehen vielleicht von Fällen, in denen Studienanfänger kleine Programme schreiben, sind die Kosten bei extensiver Fehlerreparatur höher als ihr Gewinn“
Alternative: interaktive Programmierumgebungen

Grundlage

- LL(k)-Parser haben die Eigenschaft des fortsetzungsfähigen Präfix
(d.h. bestätigter Anfang eines Eingabewortes hat mindestens eine Fortsetzung zu einem Satz)

Idee der Fehlerbehandlung

- Keine Korrektur im bereits gelesenen Teil
- Stattdessen: Veränderung oder Überlesen in der Resteingabe

Möglichkeit: „Panik-Modus“

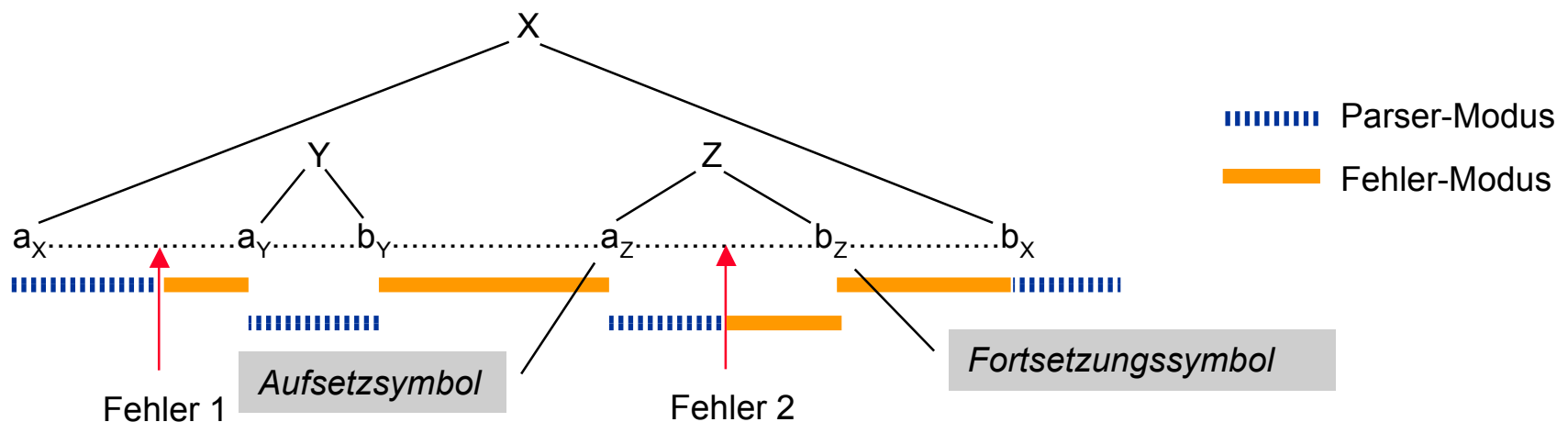
- Ende- oder Trennsymbol für aktuelles Nichtterminal suchen
- Dazwischen: überlesen

Nachteile

- Es wird evtl. zu viel überlesen
- Parser kommt unter Umständen außer Tritt
(z.B. durch falsche Klammerzuordnung)

Zwei Modi

- Parser-Modus
 - „Normalmodus“: Analyse eines Worts für Nichtterminal X
 - In Anfangskonfiguration gestartet: Ende der Analyse, falls $X = S$
 - Rekursiv aus Fehler-Modus gestartet: Rückkehr in Fehler-Modus
- Fehler-Modus
 - Modus zur Fehlerbehandlung: Eintritt bei Auftreten eines Fehlers
 - Überlesen bis zu geeignetem Endesymbol („Fortsetzen bei einem *Fortsetzungssymbol*“); Rückkehr in Parser-Modus
 - Rekursiver Eintritt in Parser-Modus (für X), falls charakteristisches Anfangssymbol für X erkannt („Aufsetzen bei einem *Aufsetzsymbol*“)



Integration der Fehlerbehandlung

charakterisieren eindeutig Ende bzw. Anfang von Teilbäumen

- Klassen von Fortsetzungs- und Aufsetzsymbolen (LAST/FOLLOW bzw. BEG/PREC)
- Schrittweise Modifikation des Generierungsschemas für recursive descent RLL(1)-Parser (unter Erhaltung der Korrektheit)

Ziel der Modifikation

- Keine Folgefehlermeldungen für den gleichen Fehler aus verschiedenen Parserprozeduren (d.h. Fehler wird in genau einer Prozedur behandelt)

Prinzipielle Vorgehensweise

- Zusätzliche Parameter (Mengen jeweils relevanter Fortsetzungs- und Aufsetzsymbole)
- (boolesche) Parserfunktionen statt -prozeduren; Ergebnis
 - **true**: Aufrufer soll im Parser-Modus fortfahren (Fehler erfolgreich behandelt)
 - **false**: Aufrufer muss im Fehler-Modus fortfahren (Fehlerbehandlung nicht abgeschlossen)
- Fehlermeldung in der Funktion, die den Fehler erkennt
- Ende des Rumpfes jeder Parserfunktion
 - fehler: (* Ausgabe einer für die Situation bezeichnenden Fehlermeldung *)
 - behandl: (* Fortsetzen oder Aufsetzen *)



LAST-Symbol

- Letztes Symbol in einem Wort für X
- Definition analog zu FIRST: Sei $w = w_1..w_n \in V_T^*$ und $k \in \mathbb{N}$:

- k-Suffix $w : k =_{\text{def}} w_{\max(1, n-k+1)} \dots w_n$
- $\text{LAST}_k(X) =_{\text{def}} \{w : k \mid X \xRightarrow{R} \text{Im}^* w\}$ für $X \in V_N$

Fehlerbehandlungsteil in der Funktion X

- fehler: (* melde Fehler *)
- behandl: **while** nextsym **not in** $\text{LAST}_1(X)$ **do** scan **od**; scan; **return(true)**

Lesen bis einschließlich LAST-Symbol

Lesen des nächsten Symbols

Fehler erfolgreich behandelt

LAST(G)

- Menge von Nichtterminalen, bei denen Fortsetzung über LAST-Symbole gewünscht ist
 - Vom Benutzer anzugeben
 - Kandidaten: „rechte Klammern“ (Vorsicht bei rekursiven Produktionen: evtl. falsche Klammerzuordnung)
- Probleme
 - Fehler = fehlendes LAST-Symbol
 - Es gibt kein LAST-Symbol für aktuelles Nichtterminal
- Abhilfe
 - Suche nach LAST-Symbolen „umfassender“ Nichtterminale (die den Funktionen als zusätzliche Argumente mitgegeben werden)



Für Nichtterminal $X \in \text{LAST}(G)$

```

func X (lasts: set of symbol) bool;
  var s: set of symbol;
  begin
    s := lasts  $\cup$  LAST1(X);
    progr([X  $\rightarrow$  . $\alpha$ ]);
    return(true);
  fehler: error(„...“);
  behandl: while nextsym not in s do scan od;
    if nextsym in LAST1(X)
      then scan; return(true)
      else return(false) fi
  end

```

```

progr([Y  $\rightarrow$  ...X...]) =def
  if nextsym in FiFo([Y  $\rightarrow$  ...X...])
  then if not X(s) then goto behandl fi
  else goto fehler fi

```

Für Nichtterminal $X \notin \text{LAST}(G)$

```

func X (lasts: set of symbol) bool;
  var s: set of symbol;
  begin
    s := lasts;
    progr([X  $\rightarrow$  . $\alpha$ ]);
    return(true);
  fehler: error(„...“);
  behandl: return(false)
  end

progr([Y  $\rightarrow$  ...X...]) =def
  if not X(s) then goto behandl fi

```

Beachte

Für alle anderen regulären Items ist progr wie oben in „recursive descent RLL(1)-Parser“ (ohne Fehlerbehandlung) definiert



Modifiziertes Hauptprogramm des RLL(1)-Parsers

```
program parser;  
  var nextsymbol: symbol;  
  proc scan; (* liest nächstes Eingabesymbol in nextsym *)  
  proc error (meldung: string); (* gibt Fehlermeldung aus – stoppt den Parserlauf nicht mehr*)  
  proc accept; (* meldet Ende der Analyse und stoppt Parserlauf *)  
  func  $X_0$  (lasts: set of symbol) bool; begin ... end;  
  func  $X_1$  (lasts: set of symbol) bool; begin ... end; ....  
  func  $X_n$  (lasts: set of symbol) bool; begin ... end;  
begin  
  scan;  
  if nextsym in  $Fl_1(X_0)$  (* nur generiert, wenn  $X_0 \in \text{LAST}(G)$  *)  
  then if not  $X_0(\{„\#“\})$  then goto behandl fi  
  else goto fehler fi;  
  if nextsym = „#“ then accept else goto fehler fi  
  fehler: error(...)  
  behandl: while nextsym not in {„#“} do scan od  
end
```

Rümpfe wie oben, abhängig von
der Definition von $\text{LAST}(G)$



FOLLOW-Symbole

- Charakteristische Symbole, die auf ein Vorkommen eines Nichtterminals folgen können
- Wieder vom Benutzer anzugeben

z.B. „;“ bei STAT

FOLLOW(G)

- Menge von Nichtterminalen bei denen Fortsetzung über FOLLOW-Symbole gewünscht wird

Modifikation des Generierungsschemas

- Parserfunktion X erhält weiteren Parameter für Mengen von Folgesymbolen für das jeweilige angewandte Vorkommen von X



Für Nichtterminal $X \in \text{LAST}(G)$

```
func X (lasts, follows : set of symbol) bool;
  var s: set of symbol;
  begin
    s := lasts  $\cup$   $\text{LAST}_1(X)$   $\cup$  follows;
    progr([X  $\rightarrow$  . $\alpha$ ]);
    return(true);
  fehler: error(„...“);
  behandl: while nextsym not in s do scan od;
    if nextsym in  $\text{LAST}_1(X)$ 
      then scan; return(true) fi;
    if nextsym in follows
      then return(true) fi;
    return(false)
  end
```

Für Vorkommen von $X \in \text{FOLLOW}(G)$

```
progr([Y  $\rightarrow$  ...X...]) =def
  if nextsym in FiFo([Y  $\rightarrow$  ...X...])
    then if not X(s,  $\text{FO}_1([Y \rightarrow \dots X \dots])$ )
      then goto behandl fi
    else goto fehler fi
```

Für Nichtterminal $X \notin \text{LAST}(G)$

```
func X (lasts, follows : set of symbol) bool;
  var s: set of symbol;
  begin
    s := lasts  $\cup$  follows;
    progr([X  $\rightarrow$  . $\alpha$ ]);
    return(true);
  fehler: error(„...“);
  behandl: if follows  $\neq \emptyset$ 
    then while nextsym not in s do scan od;
      if nextsym in follows
        then return (true) fi fi;
    return(false)
  end
```

Für Vorkommen von $X \notin \text{FOLLOW}(G)$

```
progr([Y  $\rightarrow$  ...X...]) =def
  if nextsym in FiFo([Y  $\rightarrow$  ...X...])
    then if not X(s,  $\emptyset$ )
      then goto behandl fi
    else goto fehler fi
```



Beispiel (Anweisungsgrammatik: Fortsetzung bei LAST und FOLLOW-Symbolen)

$LAST(G) = \{If_Anw, While_Anw, Repeat_Anw, Proz_Aufruf, Wertzuweisung\}$

$LAST_1$ -Mengen

$LAST_1(If_Anw) = \{ "fi" \}$

$LAST_1(While_Anw) = \{ "od" \}$

$LAST_1(Repeat_Anw) = \{ "Bed" \}$

$LAST_1(Proz_Aufruf) = \{ "()" \}$

$LAST_1(Wertzuweisung) = \{ "Ausdr" \}$

$LAST_1(Anw) = \{ "fi", "od", "Bed", ")", "Ausdr" \}$

$FOLLOW(G) = \{ An_Folge, Anw, Ausdr_Folge \}$

$FOLLOW_1$ -Mengen

$FO_1([If_Anw \rightarrow \text{if Bed then } .An_folge \{fi \mid \text{else } An_folge \}]) = \{ "fi", "else" \}$

$FO_1([If_Anw \rightarrow \text{if Bed then } An_folge \{fi \mid \text{else } .An_folge \}]) = \{ "fi" \}$

$FO_1([While_Anw \rightarrow \text{while Bed do } .An_folge \text{ od}]) = \{ "od" \}$

$FO_1([Repeat_Anw \rightarrow \text{repeat } .An_folge \text{ until Bed}]) = \{ "until" \}$

$FO_1([Proz \rightarrow .An_folge]) = \{ "\#" \}$

$FO_1([An_Folge \rightarrow .Anw \{ ; Anw \}^*]) = FO_1([An_Folge \rightarrow Anw \{ ; .Anw \}^*]) = \{ ";", "od", "until", "fi", "else", "\#" \}$

$FO_1([Proz_Aufruf \rightarrow \text{call name } (.Ausdr_folge)]) = \{ "()" \}$

$FO_1([Anw \rightarrow .If_Anw \mid While_Anw \mid Repeat_Anw \mid Proz_Aufruf \mid Wertzuweisung]) =$

$FO_1([Anw \rightarrow If_Anw \mid .While_Anw \mid Repeat_Anw \mid Proz_Aufruf \mid Wertzuweisung]) =$

$FO_1([Anw \rightarrow If_Anw \mid While_Anw \mid .Repeat_Anw \mid Proz_Aufruf \mid Wertzuweisung]) =$

$FO_1([Anw \rightarrow If_Anw \mid While_Anw \mid Repeat_Anw \mid .Proz_Aufruf \mid Wertzuweisung]) =$

$FO_1([Anw \rightarrow If_Anw \mid While_Anw \mid Repeat_Anw \mid Proz_Aufruf \mid .Wertzuweisung]) =$

$FO_1([Anw \rightarrow .(If_Anw \mid While_Anw \mid Repeat_Anw \mid Proz_Aufruf \mid Wertzuweisung)]) = \{ ";", "od", "until", "fi", "else", "\#" \}$

```
Anw  → If_Anw | While_Anw | Repeat_Anw |
      Proz_Aufruf | Wertzuweisung
If_Anw  →  if Bed then An_Folge
           {fi | else An_Folge fi}
While_Anw  →  while Bed do An_Folge od
Repeat_Anw  →  repeat An_Folge until Bed
Proz_Aufruf  →  call name ( Ausdr_Folge )
Wertzuweisung  →  name := Ausdr
Ausdr_Folge  →  Ausdr {, Ausdr}*
An_Folge  →  Anw {; Anw}*
Progr  →  An_Folge
```





Probleme der bisherigen Fehlerbehandlung

- Evtl. zu viel überlesen
- Rekursive Nichtterminale

Beispiel

... while ... do a := a+1 while ... do ... od; ... od ...



Fehlerstelle

Abhilfe

- (Rekursiver) Übergang in den Parser-Modus, wenn ein charakteristisches Anfangssymbol eines Nichtterminals X gefunden wird
- Beschränkung auf solche X, die vom aktuellen Nichtterminal erreichbar sind

Forderungen an für das Aufsetzen geeignete Anfangssymbole

- Sollten ausschließlich als erste Symbole von Worten für Nichtterminale auftreten
- Anfangssymbole eines Nichtterminals müssen eindeutig zu einem der aus diesem Nichtterminal ableitbaren Nichtterminale gehören

BEG(G)

- Vom Benutzer festzulegende Menge von Nichtterminalen bei denen Aufsetzen bei Anfangssymbolen gewünscht ist



Anfangssymbol

- Sei $G = (V_N, V_T, p, S)$ rrkfG und $X \in V_N$. Weiter sei
 - $\text{prod}(X) =_{\text{def}} \{Y \mid X \xRightarrow{R}_{\text{lm}}^+ \dots Y \dots\}$:
Menge der aus X produzierbaren Nichtterminale
 - $\text{prod}_b(X) =_{\text{def}} \text{prod}(X) \cap \text{BEG}(G)$:
Teilmenge der (aus X produzierbaren) „Aufsetzsymbole“

- Menge der **Anfangssymbole**

- Für Y im Kontext X : $\text{BEGSYMBS}(X)_Y =_{\text{def}} \text{FI}_1(Y) \setminus \bigcup_{Z \in \text{prod}(X), Z \neq Y} \text{FI}_1(Z)$
- Im Kontext X : $\text{BEGSYMBS}(X) =_{\text{def}} \bigcup_{Y \in \text{prod}_b(X)} \text{BEGSYMBS}(X)_Y$

Die für Y im Kontext X
„eindeutigen“ Anfänge

Es gilt

- $\text{BEGSYMBS}(X)$ bestimmt eindeutige Fortsetzung mit einem Nichtterminal $Y_i \in \text{prod}(X)$



Modifiziertes Hauptprogramm für Fortsetzung bei Anfangssymbolen

```
begin
  scan;
  if nextsym in FiFo([S' → .X0]))
    then if not X0({„#“}, ∅, BEGSYMBS(X0)) then goto behandl fi
    else goto fehler fi;
  if nextsym = „#“ then accept else goto fehler fi
fehler: error(...)
behandl: while nextsym ≠ „#“ do
  if nextsym in BEGSYMBS(X0) then recover(X0) else scan fi od
end
```

Wobei für alle $X \in V_N$ das Macro $\text{recover}(X)$ definiert ist durch

```
case nextsym in
  BEGSYMBS(X)Y1: Y1(s, ∅, b)
  BEGSYMBS(X)Y2: Y2(s, ∅, b) ...
  BEGSYMBS(X)Yn: Yn(s, ∅, b)
endcase
```

mit

- s: Menge der akkumulierten Endesymbole
- b: Menge der akkumulierten Anfangssymbole



Für Nichtterminal $X \in \text{LAST}(G)$

```

func X (lasts, follows, begins : set of symbol) bool;
var s, b: set of symbol;
begin
  s := lasts  $\cup$   $\text{LAST}_1(X)$   $\cup$  follows;
  b := begins  $\cup$  BEGSYMBS(X);
  progr([X  $\rightarrow$  . $\alpha$ ]);
  return(true);
fehler: error(„...“);
behandl: while nextsym not in s do
  if nextsym in b
    then if nextsym in BEGSYMBS(X)
      then recover(X)
      else return(false) fi
    else scan fi od;
  if nextsym in  $\text{LAST}_1(X)$ 
    then scan; return(true) fi;
  if nextsym in follows
    then return(true) fi;
  return(false)
end

```

Für Vorkommen von $X \in \text{FOLLOW}(G)$

$X(s, \text{FO}_1([Y \rightarrow \dots X \dots]), b)$ für $X(s, \text{FO}_1([Y \rightarrow \dots X \dots]))$

Für Nichtterminal $X \notin \text{LAST}(G)$

```

func X (lasts, follows, begins : set of symbol) bool;
var s, b: set of symbol;
begin
  s := lasts  $\cup$  follows;
  b := begins;
  if follows  $\neq \emptyset$  then b := b  $\cup$  BEGSYMBS(X) fi;
  progr([X  $\rightarrow$  . $\alpha$ ]);
  return(true);
fehler: error(„...“);
behandl: if follows  $\neq \emptyset$ 
  then while nextsym not in s do
    if nextsym in b
      then if nextsym in BEGSYMBS(X)
        then recover(X)
        else return(false) fi
      else scan fi od;
    if nextsym in follows then return (true) fi fi;
  return(false)
end

```

Für Vorkommen von $X \notin \text{FOLLOW}(G)$

$X(s, \emptyset, b)$ für $X(s, \emptyset)$ —

Ansonsten wie bei Fortsetzen
mit FOLLOW-Symbolen



Vorgängersymbole

- Charakteristische Symbole, die einem Vorkommen eines Nichtterminals immer vorangehen
- Rekursive Definition analog zu FOLLOW₁
 - Sei $[X \rightarrow \alpha . \beta \gamma]$ erweitertes kontextfreies Item
 - $\text{PRECEDE}([X \rightarrow \alpha . \beta \gamma]) =_{\text{def}} \{a \in V_T \mid S \xRightarrow{R}_{lm}^* wX\delta \xRightarrow{R}_{lm} w\alpha\beta\gamma\delta \xRightarrow{R}_{lm}^* wua\beta\gamma\delta\}$
Terminale, die in einer regulären Linkssatzform diesem Vorkommen von β vorangehen können

PREC(G)

- Menge vom Benutzer festzulegender Vorkommen von regulären Unterausdrücken zum Aufsetzen bei Vorgängersymbolen

Bedingungen für die Berechnung von Aufsetzsymbolen

- Keine Symbole aus $\text{BEG}(G) \cap \text{PREC}(G)$ als Aufsetzsymbole (Eindeutigkeit)
- Jedes Aufsetzsymbol sollte eindeutig ein Nichtterminal bestimmen, für das rekursiv in den Parser-Modus gegangen wird



Berechnung der Anfangs- bzw. Vorgängersymbole von Y im Kontext X

- Initiale Anfangs- und Vorgängersymbole
 - Berechne $B1(X)_Y =_{\text{def}} FI_1(Y)$, falls $Y \in \text{prod}_b(X)$, und $\emptyset$ sonst
 - Berechne $P1(X)_Y =_{\text{def}} \bigcup_{Z \in \text{prod}(X) \cup \{X\}} \{\text{PRECEDE}([Z \rightarrow \dots Y \dots]) \mid [Z \rightarrow \dots Y \dots] \in \text{PREC}(G)\}$
- Elimination von Symbolen, die gleichzeitig Anfangs- und Vorgängersymbole sind
 - $B2(X)_Y =_{\text{def}} B1(X)_Y \setminus \bigcup_Z P1(X)_Z$
 - $P2(X)_Y =_{\text{def}} P1(X)_Y \setminus \bigcup_Z B1(X)_Z$
- Elimination mehrfach vorkommender Anfangs- und Vorgängersymbole
 - $BEGS(X)_Y =_{\text{def}} B2(X)_Y \setminus \bigcup_{Z \neq Y} B2(X)_Z$
 - $\text{PRECS}(X)_Y =_{\text{def}} P2(X)_Y \setminus \bigcup_{Z \neq Y} P2(X)_Z$
- $\text{BEGSYMBS}(X) =_{\text{def}} \bigcup_Y (\text{BEGS}(X)_Y \cup \text{PRECS}(X)_Y)$

Modifikation des Generierungsschemas für ELL(1)-Parser

- Für alle $X \in V_N$ wird das Macro $\text{recover}(X)$ geändert in

```

case nextsym in
     $\text{BEGS}(X)_{Y_1} : Y_1(s, \emptyset, b) \dots$ 
     $\text{BEGS}(X)_{Y_n} : Y_n(s, \emptyset, b)$ 
     $\text{PRECS}(X)_{Z_1} : \text{begin scan; } Z_1(s, \emptyset, b) \text{ end } \dots$ 
     $\text{PRECS}(X)_{Z_m} : \text{begin scan; } Z_m(s, \emptyset, b) \text{ end}$ 
endcase
    
```



Beispiel (Anweisungsgrammatik)

- Aufsetzen bei Anfangs- /Vorgänger-Symbolen
 - $BEG(G) = \{Anw\}$
 - $PREC(G) = \{[An_Folge \rightarrow Anw \{; .Anw\}^*]\}$, damit
 - $PRECEDE([An_Folge \rightarrow Anw \{; .Anw\}^*]) = \{";"\}$,
- Berechnung von BEGS, PRECS, BEGSYMBS

- Sei $VORG(Anw)$
 $= \{X \in V_N \mid Anw \in prod(X)\}$
 $= \{Anw, If_Anw, While_Anw, Repeat_Anw, An_Folge, Progr\}.$
- $B1(X)_{Anw} = \{"if", "while", "repeat", "call", "name"\}$ für alle $X \in VORG(Anw)$; $B1(X)_Y = \emptyset$, sonst
- $P1(X)_{Anw} = \{";"\}$ für alle $X \in VORG(Anw)$; $P1(X)_Y = \emptyset$, sonst
- $B2 = B1$
- $P2 = P1$
- $BEGS = B1$
- $PRECS = P1$
- $BEGSYMBS(X) = \{"if", "while", "repeat", "call", "name", ";"\}$ für alle $X \in VORG(Anw)$;
 $BEGSYMBS(X) = \emptyset$, sonst

```

Anw  →   If_Anw | While_Anw | Repeat_Anw |
         Proz_Aufruf | Wertzuweisung
If_Anw → if Bed then An_Folge {fi | else An_Folge fi}
While_Anw → while Bed do An_Folge od
Repeat_Anw → repeat An_Folge until Bed
Proz_Aufruf → call name ( Ausdr_Folge )
Wertzuweisung → name := Ausdr
Ausdr_Folge → Ausdr {, Ausdr}*
An_Folge → Anw {; Anw}*
Progr → An_Folge
    
```