Syntaktische Analyse – Teil 2

Übersicht

- Top-Down-Syntaxanalyse
- LL(k)
  - Definition
  - Beispiele und Eigenschaften
- Grammatikumformungen und LL(k)
  - Grammatikumformungen
  - Alternative Definition für LL(k)
  - Starke LL(k)-Grammatiken
- LL(k)-Parser
- Linksrekursion
- Rechtsreguläre kontextfreie Grammatiken (rrkfG)
- $FIRST_1$ und $FOLLOW_1$ 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-Syntaxanalyse

Top down-Parser

- Konstruiert Syntaxbaum von der Wurzel her — *Ist Linksparser*
- Anfangssituation

Baum	Resteingabe
S	**id** + **id** * **id**

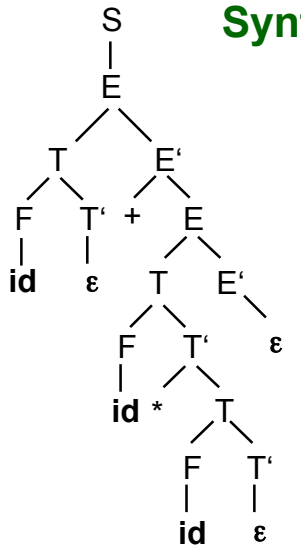
- Aktivitäten
  - Produktion an aktuelles Baumfragment anbauen (nichtterminales Blatt durch rechte Seite ersetzen)
  - Übereinstimmung zwischen angebauten Symbolen und Eingabesymbolen überprüfen

Beispiel

- Geg.: kfG G_2 = ({S,E,E‘,T,T‘,F}, {+, *,(,),**id**}, P, S) mit
  P = {S → E, E → TE‘, E‘ → ε, E‘ → +E, T → FT‘,
  T‘ → ε, T‘ → *T, F → (E), F → **id**}

Eingabewort **id** + **id** * **id**

Syntaxbaum



LL(k): Definition

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

LL(k): Beispiele, Eigenschaften (1/3)

Beispiel 1

- kfG G_3 mit den Produktionen

  STAT → **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

LL(k): Beispiele, Eigenschaften (2/3)

Es gilt

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

Beispiel 2

- kfG G_4 mit den Produktionen

  STAT → **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

LL(k): Beispiele, Eigenschaften (3/3)

Es gilt

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

Beispiel 3

- kfG G_5 mit den Produktionen

STAT →	**if id then** STAT **else** STAT **fi** |	
	while id do STAT **od** |	
	VAR := VAR |	
	id (IDLIST)	(* Prozeduraufruf *)
VAR →	**id** | **id** (IDLIST)	(* indizierte Variable *)
IDLIST →	**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

LL(k): Beispiele, Eigenschaften (3a/3)

Formale Beweise für die Beispiele

- **Beweis für Beispiel 1**:
  - Für STAT $\Rightarrow^*_{lm}$ u STATα $\Rightarrow_{lm}$ u$\beta\alpha$ $\Rightarrow^*_{lm}$ ux und
    STAT $\Rightarrow^*_{lm}$ u STATα $\Rightarrow_{lm}$ u$\gamma\alpha$ $\Rightarrow^*_{lm}$ uy mit 1: x = 1: y
  - gilt $\beta = \gamma$
  - Z.B. folgt aus 1: x = 1: y = „**id**“, dass $\beta = \gamma$ = „**id** := **id**“ (und analog für **if** und **while**)
- **Beweis für Beispiel 2**:
  - Für STAT $\Rightarrow^*_{lm}$ u STAT α $\Rightarrow_{lm}$ u **id** := **id** α [= u$\beta\alpha$] $\Rightarrow^*_{lm}$ ux,
    STAT $\Rightarrow^*_{lm}$ u STAT α $\Rightarrow_{lm}$ u **id**: STAT α [= u$\gamma\alpha$] $\Rightarrow^*_{lm}$ uy,
    STAT $\Rightarrow^*_{lm}$ u STAT α $\Rightarrow_{lm}$ u **id** (**id**) α [= u$\delta\alpha$] $\Rightarrow^*_{lm}$ uz
  - mit 1: x = 1: y = 1: z = **id**
  - gilt: β, γ und δ sind paarweise verschieden
- **Beweis für Beispiel 3**:
  - Für STAT $\Rightarrow_{lm}$ VAR := VAR $\Rightarrow_{lm}$ **id** (IDLIST) := VAR $\Rightarrow^*_{lm}$ **id** (**id**, **id**, ..., **id**) := **id** und
    STAT $\Rightarrow_{lm}$ **id** (IDLIST) $\Rightarrow^*_{lm}$ **id** (**id**, **id**, ..., **id**)
    mit |**id** (**id**, **id**, ..., **id**)| ≥ k
  - gilt: k: (**id** (**id**, **id**, ..., **id**) := **id**) = k: (**id** (**id**, **id**, ..., **id**))
  - aber: „VAR := VAR“ ≠ „**id** (IDLIST)“

Grammatikumformungen

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

    STAT → ... | VAR := VAR | **id** (IDLIST)

    ersetzt durch

    STAT → ... | ZUWPROZ | **id** := VAR
    ZUWPROZ → **id** (IDLIST) ZPREST
    ZPREST → := VAR | ε
- 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 $\{a^n0b^n \mid n \geq 0\} \cup \{a^n1b^{2n} \mid n \geq 0\}$

Alternative Definition für LL(k)

Satz (Alternative Definition für LL(k))

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

Intuitiv: Nächste für k Eingabezeichen beseitigen den Nicht-Determinismus

Daraus

- Gute Kriterien für die Zugehörigkeit zu gewissen Teilklassen der LL(1)-Grammatiken
  (da $FIRST_1(\beta\alpha) = FIRST_1(\beta) \oplus_1 FOLLOW_1(A)$ für alle α mit $S \Rightarrow^*_{lm} uA\alpha$)

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 $A \in V_N$)
  - $A \to \beta, A \to \gamma \in P \wedge \beta \neq \gamma \Rightarrow (FIRST_1(\beta) \oplus_1 FOLLOW_1(A)) \cap (FIRST_1(\gamma) \oplus_1 FOLLOW_1(A)) = \varnothing$
- Sei G ε-freie kfG (d.h. ohne Produktionen der Form $A \to \varepsilon$). G ist genau dann LL(1), wenn
  - Für jedes $A \in V_N$ mit $A \to \alpha_1 \mid ... \mid \alpha_n$ gilt: $FIRST_1(\alpha_1), ..., FIRST_1(\alpha_n)$ sind paarweise disjunkt
- Eine kfG G ist genau dann LL(1), wenn für alle Alternativen $A \to \alpha_1 \mid ... \mid \alpha_n$ ($A \in V_N$) gilt
  - $FIRST_1(\alpha_1), ..., FIRST_1(\alpha_n)$ sind paarweise disjunkt; höchstens ein $FIRST_1(\alpha_i)$ enthält ε
  - Aus $\alpha_i \Rightarrow^*_{lm} \varepsilon$ folgt: $FIRST_1(\alpha_j) \cap FOLLOW_1(A) = \varnothing$ für $1 \leq j \leq n, j \neq i$

Starke LL(k)-Grammatiken

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
  - $(FIRST_k(\beta) \oplus_k FOLLOW_k(A)) \cap (FIRST_k(\gamma) \oplus_k FOLLOW_k(A)) = \varnothing$

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 → aAaa | bAba, A → b | ε}, 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)

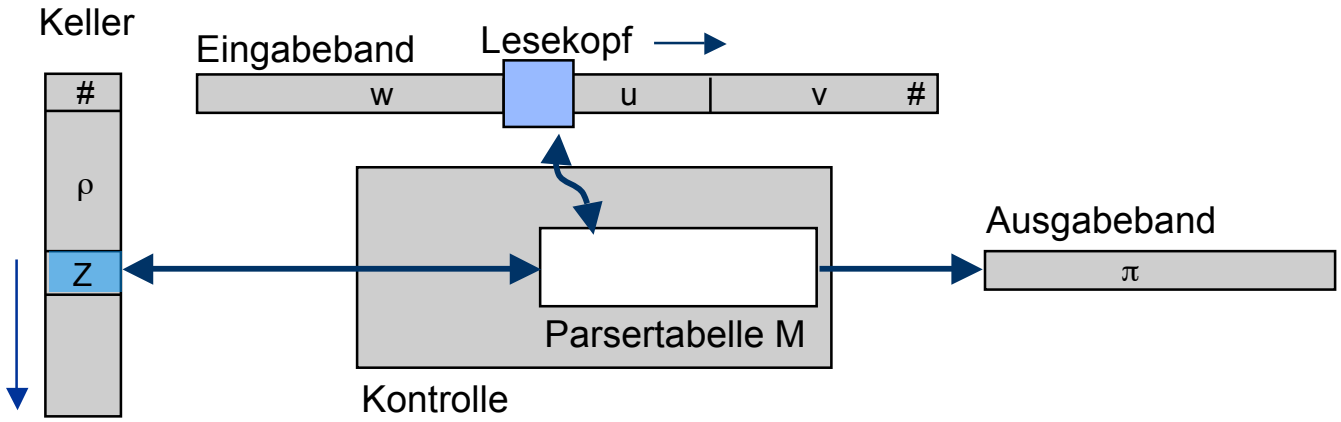
Bei gegebenem Linkskontext von A reichen 2 Zeichen Vorausschau

Für $A \rightarrow b$ *und* $A \rightarrow \varepsilon$ *gilt:*
$(FIRST_2(b) \oplus_2 FOLLOW_2(A)) \cap (FIRST_2(\varepsilon) \oplus_2 FOLLOW_2(A)) =$
$(b \oplus_2 \{aa, ba\}) \cap \{aa, ba\} =$
$\{ba, bb\} \cap \{aa, ba\} = \{ba\} \neq \varnothing$

(starke) LL(k)-Parser

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

LL(1)-Parser (1/2)

Erzeugung der LL(1)-Parser-Tabelle

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)*

- Eingabe: LL(1)-Grammatik G, $FIRST_1$ und $FOLLOW_1$ für G
- Ausgabe: Parsertabelle M, die folgendermaßen aufgebaut wird
  - Für alle $X \rightarrow \alpha \in P$ und alle Terminalsymbole $a \in FIRST_1(\alpha)$: M[X, a] auf $(X \rightarrow \alpha)$ setzen
  - Falls $eps(\alpha)$ = **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 → E, E → TE‘, E‘ → +E, E‘ → ε, T → FT‘, T‘ → *T, T‘ → ε, F → (E), F → **id**}
  - $FIRST_1$-Mengen: $FIRST_1(S) = FIRST_1(E) = FIRST_1(T) = FIRST_1(F) = \{(, \textbf{id}\}$, $FIRST_1(E') = \{+, \varepsilon\}$, $FIRST_1(T') = \{*, \varepsilon\}$
  - $FOLLOW_1$-Mengen: $FOLLOW_1(S) = \{\#\}$, $FOLLOW_1(E') = FOLLOW_1(E) = \{), \#\}$, $FOLLOW_1(T') = FOLLOW_1(T) = \{+,), \#\}$, $FOLLOW_1(F) = \{*, +,), \#\}$
  - Parsertabelle M

	(	)	+	*	**id**	**#**
S	(S → 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*

LL(1)-Parser (2/2)

Kellerinhalt	Eingabe
# [S → .E]	id * id #
# [S → .E] [E → .TE‘]	id * id #
# [S → .E] [E → .TE‘] [T → .FT‘]	id * id #
# [S → .E] [E → .TE‘] [T → .FT‘] [F → **.id**]	id * id #
# [S → .E] [E → .TE‘] [T → .FT‘] [F → **id.**]	* id #
# [S → .E] [E → .TE‘] [T → F.T‘]	* id #
# [S → .E] [E → .TE‘] [T → F.T‘] [T‘ → .*T]	* id #
# [S → .E] [E → .TE‘] [T → F.T‘] [T‘ → *.T]	id #
# [S → .E] [E → .TE‘] [T → F.T‘] [T‘ → *.T] [T → .FT‘]	id #
# [S → .E] [E → .TE‘] [T → F.T‘] [T‘ → *.T] [T → .FT‘] [F → **.id**]	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.]	#
#	#

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 (1/2)

Linksrekursion

- Sei G = (V_N, V_T, P, S) kfG, p ∈ P, A ∈ V_N
  - p ist **direkt rekursiv** ⇔ p hat die Form A → αAβ
  - p ist **direkt linksrekursiv** (bzw. **rechtsrekursiv**) ⇔ p hat die Form A → Aβ (bzw. A → αA)
  - A ist rekursiv ⇔ es gibt Ableitung A ⇒$^+$ αAβ
  - A ist **linksrekursiv** (bzw. **rechtsrekursiv**) ⇔ es gibt Ableitung A ⇒$^+$ Aβ (bzw. A ⇒$^+$ αA)
  - G ist **linksrekursiv** ⇔ G enthält mindestens 1 linksrekursives Nichtterminal

Satz

- Sei G kfG
  - G ist linksrekursiv ⇒ G ist nicht LL(k) für jedes k
  - G ist LL(k)-Grammatik ⇒ 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

Linksrekursion (2/2)

Beispiel

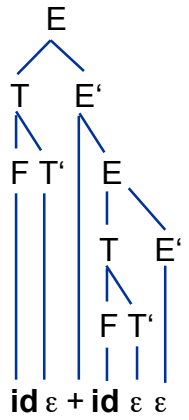
- Geg.: Varianten der Ausdrucksgrammatik
  - G_0 mit P = {E → E+T | T, T → T*F | F, F → (E) | **id**}
  - G_2‘ mit P = {E → TE‘, E‘ → +E | ε, T → FT‘, T‘ → *T | ε, F → (E) | **id**}
- Syntaxbäume für **id** + **id**

3 Nichtterminale
6 Produktionen

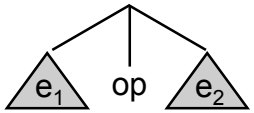
5 Nichtterminale
8 Produktionen

(für G_0)

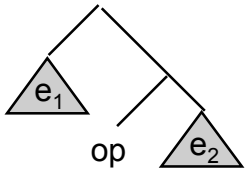
(für G_2‘)



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 Grammatiken (1/3)

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)

- $G_e = (\{S,E,T,F\}, \{+,–,*, /,‹,›,\mathbf{id}\}, \{S \rightarrow E, E \rightarrow T((+|–)T)^*, T \rightarrow F((*|/)\ F)^*, F \rightarrow (‹E›|\mathbf{id})\}, S)$

Als Terminalsymbole werden hier zur besseren Lesbarkeit die Klammern ‹ und › verwendet

Begriffe

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

Rechtsreguläre kontextfreie Grammatiken (2/3)

Beispiel

- Geg.: G_e = ({S,E,T,F}, {+,–,∗, /,‹,›,**id**}, {S → E, E → T((+|–)T)*, T → F((∗|/) F)*, F → (‹E›|**id**)}, S)
- Reguläre Linksableitung zum Wort **id** + **id** ∗ **id**

$$S \underset{R}{\Rightarrow}_{lm} E \underset{R}{\Rightarrow}_{lm} T((+|–)T)^* \underset{R}{\Rightarrow}_{lm} F((*|/)\ F)^*((+|–)T)^* \underset{R}{\Rightarrow}_{lm} (‹E›|\mathbf{id})\ ((*|/)\ F)^*((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id}\ ((*|/)\ F)^*((+|–)T)^*$$
$$\underset{R}{\Rightarrow}_{lm} \mathbf{id}\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id}\ (+|–)T((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + T((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + F\ ((*|/)\ F)^*\ ((+|–)T)^*$$
$$\underset{R}{\Rightarrow}_{lm} \mathbf{id} + (‹E›|\mathbf{id})\ ((*|/)\ F)^*\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id}\ ((*|/)\ F)^*\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id}\ (*|/)\ F\ ((*|/)\ F)^*\ ((+|–)T)^*$$
$$\underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id} * F\ ((*|/)\ F)^*\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id} * (‹E›|\mathbf{id})\ ((*|/)\ F)^*\ ((+|–)T)^*$$
$$\underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id} * \mathbf{id}\ ((*|/)\ F)^*\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id} * \mathbf{id}\ ((+|–)T)^* \underset{R}{\Rightarrow}_{lm} \mathbf{id} + \mathbf{id} * \mathbf{id}$$

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
  - $w(r_1 \mid ... \mid r_n)\beta$ Entscheidung für richtige Alternative, und bei
  - $w(r)^*\beta$ Entscheidung für Fortsetzung bzw. Beendigung der Iteration

  mithilfe des nächsten Symbols der Resteingabe treffen kann

Rechtsreguläre kontextfreie Grammatiken (3/3)

Übertragung einiger Begriffe auf rrkfG

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

- Seien r, r_i ($1 \le i \le n$) reguläre Ausdrücke
  - **Direkter regulärer Unterausdruck**
    - r_i: von $(r_1 \mid ... \mid r_n)$ und $(r_1 ... r_n)$
    - 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 $p(X) = \beta\alpha\gamma$
    - $\alpha, \beta, \gamma \in (V_N \cup V_T \cup \{(,), *, |, \varepsilon\})^*$
    - α ist regulärer Unterausdruck von $\beta\alpha\gamma$

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

Beispiel

- Geg.: Regulärer Ausdruck **id** + **id** * F ((*|/) F)* ((+|-)T)*
  - Direkt reguläre Unterausdrücke: **id**, +, *, F, ((*|/) F)*, ((+|-)T)*
  - Reguläre Unterausdrücke: (*|/) F, (*|/), F, *, /, (+|-)T, (+|-), T, +, -, ...
  - Erweiterte kontextfreie Items von E → T((+|-)T)*:
    E → **.**T((+|-)T)*, E → T**.**((+|-)T)*, E → T(**.**(+|-)T)*, E → T((**.**+|-)T)*, E → T((+|**.**-)T)*, E → T((+|-)**.**T)*,
    E → T((+|-)T)***.**

Keine direkt regulären Unterausdrücke sind z.B. * F, + **id** * F, (+|-)

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

$FIRST_1$ und $FOLLOW_1$ für rrkfG (1/2)

$FIRST_1$- und $FOLLOW_1$-Berechnung für rrkfG

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

Beispiel

- Geg.: $G_e = (\{S,E,T,F\}, \{+,-,*, /,\langle,\rangle,$ **id**$\},p, S)$
  mit p = $\{S \to E,\ E \to T((+|-)T)^*,$
  $T \to F((*|/)\ F)^*,\ F \to (\langle E \rangle | \textbf{id})\}$
- Es gilt:
  - $eps(X) =_{def}$ **false** für alle $X \in V_N$
  - ε-ffi-Mengen = $FIRST_1$-Mengen
    $FI_1(S) = FI_1(E) = FI_1(T) = FI_1(F) =$
    $\{\langle$, **id**$\}$

$FIRST_1$ und $FOLLOW_1$ für rrkfG (2/2)

Unterschied $FIRST_1$- und $FOLLOW_1$-Mengen regulärer (Unter-)Ausdrücke

- $FIRST_1$-Mengen unabhängig vom Kontext
- $FOLLOW_1$-Mengen i.a. verschieden für verschiedene Vorkommen (in erweiterten regulären Items)
  - (1) $FO_1([S' \rightarrow .S]) =_{def} \{\#\}$
  - (2) $FO_1([X \rightarrow \beta\ (r_1 |...| .r_i |... | r_n)\ \gamma]) =_{def} FO_1([X \rightarrow \beta\ .(r_1 |...| r_i |... | r_n)\ \gamma])$ für $1 \le i \le n$
  - (3) $FO_1([X \rightarrow \beta\ (r_1... .r_i\ r_{i+1}...r_n)\ \gamma]) =_{def} \varepsilon\text{-ffi}(r_{i+1})$, falls $eps(r_{i+1})$ = **false**
  - (4) $FO_1([X \rightarrow \beta\ (r_1... .r_i\ r_{i+1}...r_n)\ \gamma]) =_{def} \varepsilon\text{-ffi}(r_{i+1}) \cup FO_1([X \rightarrow \beta\ (r_1... r_i.\ r_{i+1}...r_n)\ \gamma])$, sonst
  - (5) $FO_1([X \rightarrow \beta\ (r_1... r_{n-1}.r_n)\ \gamma]) =_{def} FO_1([X \rightarrow \beta\ .(r_1... r_{n-1}r_n)\ \gamma])$
  - (6) $FO_1([X \rightarrow \beta\ (.r)^*\ \gamma]) =_{def} \varepsilon\text{-ffi}(r) \cup FO_1([X \rightarrow \beta\ .(r)^*\ \gamma])$
  - (7) $FO_1([X \rightarrow .r]) =_{def} \cup_{Y \in V_N} FO_1([Y \rightarrow \beta\ .X\ \gamma])$

Beispiel

Nur zur Verdeutlichung

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

RLL(1)-Grammatik (1/2)

RLL(1)-Grammatik

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

Eindeutige Alternativen-auswahl

Abbruch oder Fortsetzung der Iteration eindeutig

Beispiel

- Geg.: G_e = ({S,E,T,F}, {+,–,*, /,‹,›,**id**}, {S → E, E → T((+|–)T)*, T → F((*|/) F)*, F → (‹E›|**id**)}, S)
- G_e ist RLL(1):
  - $(FI_1(+) \oplus_1 FO_1([E \to T(.(+|–)T)^*])) \cap (FI_1(-) \oplus_1 FO_1([E \to T(.(+|–)T)^*])) = \{+\} \cap \{-\} = \varnothing$
  - $(FI_1(*) \oplus_1 FO_1([T \to F(.(*|/) F)^*])) \cap (FI_1(/) \oplus_1 FO_1([T \to F(.(*|/) F)^*])) = \{*\} \cap \{/\} = \varnothing$
  - $(FI_1(‹) \oplus_1 FO_1([F \to .(‹E›|\mathbf{id})])) \cap (FI_1(\mathbf{id}) \oplus_1 FO_1([F \to .(‹E›|\mathbf{id})])) = \{‹\} \cap \{\mathbf{id}\} = \varnothing$
  - $FI_1((+|–)T) \cap FO_1([E \to T.((+|–)T)^*]) = \{+,-\} \cap \{\#, ›\} = \varnothing$ und eps((+|–)T) = **false**
  - $FI_1((*|/) F) \cap FO_1([T \to F.((*|/) F)^*]) = \{*,/\} \cap \{\#, ›,+,-\} = \varnothing$ und eps((*|/) F) = **false**

RLL(1)-Grammatik (2/2)

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 $FIRST_1$- und $FOLLOW_1$-Mengen

  FI_1(If_Anw) = {**if**}, FI_1(While_Anw) = {**while**}, FI_1(Repeat_Anw) = {**repeat**}, FI_1(Proz_Aufruf) = {**call**}

  FI_1(Wertzuweisung) = {**name**}, FI_1(Ausdr_Folge) = {**Ausdr**}

  FI_1(Progr) = FI_1(An_Folge) = FI_1(Anw) = {**if**, **while**, **repeat**, **call**, **name**}

  FO_1([Ausdr_Folge → **Ausdr.**(**,** **Ausdr**)*]) = {**)**}, FO_1([An_Folge → Anw.(**;** Anw)*]) = {**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(**,** **Ausdr**) ∩ FO_1([Ausdr_Folge → **Ausdr.**(**,** **Ausdr**)*]) = {**,**} ∩ {**)**} = ∅ und eps(**,** **Ausdr**) = **false**

     b) FI_1(**;** Anw) ∩ FO_1([An_Folge → Anw.(**;** Anw)*]) = {**;**} ∩ {**fi**, **else**, **od**, **until**, **#**} = ∅

     und eps(**,** Anw) = **false**

RLL(1)-Parser (Tabellenversion) (1/2)

RLL(1)-Parser für rrkfG (Tabellenversion)

- Parsertabelle M
  - Repräsentiert partielle Abbildung m: $It_G \times V_T \rightarrow It_G \cup \{error\}$
  - Enthält nur Zeilen für Items, in denen
    - Eine Alternative ausgewählt werden muss, d.h. Form $[X \rightarrow \cdots.(r_1 \mid \cdots \mid r_n)\cdots]$
    - Eine Iteration bearbeitet werden muss, d.h. Form $[X \rightarrow \cdots.(r)^*\cdots]$
- Arbeitsweise
  - Anfangskonfiguration: ([S‘ →.S], w#)
  - Oberstes Item ρ legt fest, ob Tabelle konsultiert werden muss; falls ja
    - M[ρ, a] ≠ error: Folge-Item (gemäß Tabelle)
    - M[ρ, a] = error: Syntaxfehler
  - Endkonfiguration: ([S‘ →S.], #)
  - Sonstige Übergänge
    - $\delta([X \rightarrow \cdots.a\cdots], a) = [X \rightarrow \cdots a.\cdots]$ [Lesen]
    - $\delta([X \rightarrow \cdots.Y\cdots], \varepsilon) = [X \rightarrow \cdots.Y\cdots]\ [Y \rightarrow .p(Y)]$ [Expansion]
    - $\delta([X \rightarrow \cdots.Y\cdots]\ [Y \rightarrow p(Y).], \varepsilon) = [X \rightarrow \cdots Y.\cdots]$ [Reduktion]
  - „Korrekturen“ und Optimierung durch Modifikation der Übergangstabelle, z.B.
    - $[X \rightarrow \cdots(\cdots \mid r_i. \mid \cdots)\cdots] \Rightarrow [X \rightarrow \cdots(\cdots \mid r_i \mid \cdots).\cdots]$
    - $[X \rightarrow \cdots (r.)^*\cdots] \Rightarrow [X \rightarrow \cdots.(r)^*\cdots]$

    } *Falls bei Lesen/Reduktion „illegale Items“ entstehen*
    - $[X \rightarrow \cdots.(r_1 \cdots r_n)\cdots] \Rightarrow [X \rightarrow \cdots (.r_1 \cdots r_n)\cdots]$ — *Optimierung*

RLL(1)-Parser (Tabellenversion) (2/2)

Erzeugung der RLL(1)-Parsertabelle

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

Beispiel

- Geg.: G_e = ({S,E,T,F}, {+,–,*, /,‹,›,**id**}, {S → E, E → T((+|–)T)*, T → F((*|/) F)*, F → (‹E›|**id**)}, S)
- Es war: $FI_1(S) = FI_1(E) = FI_1(T) = FI_1(F) = \{‹, \textbf{id}\}$, $FO_1([E \to T.((+|–)T)^*]) = \{\#,›\}$, $FO_1([T \to F.((*|/)F)^*]) = \{\#,›,+,-\}$
- Parsertabelle

Beachte: Komprimierung, d.h. direkter Übergang von [E → T.((+|–)T)*] *in* [E → T((.+|–)T)*] *unter* +; *analog für* –, * *und* /

	+	–	#	›	*	/	‹	**id**
[E → T.((+|–)T)*] [T → F.((*|/)F)*] [F → .(‹E›|**id**)]	[E → T((.+|–)T)*] [T → F((*|/)F)*.] *error*	[E → T((+|.–)T)*] [T → F((*|/)F)*.] *error*	[E → T((+|–)T)*.] [T → F((*|/)F)*.] *error*	[E → T((+|–)T)*.] [T → F((*|/)F)*.] *error*	*error* [T → F((.*|/)F)*] *error*	*error* [T → F((*|./)F)*] *error*	*error* *error* [F → (.‹E›|**id**)]	*error* *error* [F → (‹E›|.**id**)]

RLL(1)-Parser (Recursive descent) (1/4)

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 $V_N = \{X_0, ..., X_n\}$, $S = X_0$, $p = \{X_0 \rightarrow \alpha_0, ..., X_n \rightarrow \alpha_n\}$
- Außerdem sei $FiFo([X \rightarrow \cdots.\beta\cdots]) =_{def} FI_1(\beta) \oplus_1 FO_1([X \rightarrow \cdots.\beta\cdots])$

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

RLL(1)-Parser (Recursive descent) (2/4)

Behandlung der regulären Ausdrücke

- Nichtterminale $Y \in V_N$
  - progr([X → ···.Y···]) $=_{def}$ Y
- Terminale $a \in V_T$
  - progr([X → ···.a···]) $=_{def}$ **if** nextsym = a **then** scan **else** error(„a erwartet“) **fi**
- Komposition
  - progr([X → ···.(α_1 α_2… α_k)···]) $=_{def}$
    progr([X → ···(.$\alpha_1\alpha_2$…α_k)···]); progr([X → ···(α_1.α_2…α_k)···]); … progr([X → ···($\alpha_1\alpha_2$… .α_k)···]);
- Alternative
  - progr([X → ···.(α_1 | α_2 |… |α_{k-1} | α_k)···]) $=_{def}$
    case nextsym **in**
    FiFo([X → ··· (.α_1 | α_2 |… |α_{k-1} | α_k)···]): progr([X → ··· (.α_1 | α_2 |… |α_{k-1} | α_k)···]);
    FiFo([X → ··· (α_1 | .α_2 |… |α_{k-1} | α_k)···]): progr([X → ··· (α_1 | .α_2 |… |α_{k-1} | α_k)···]); …
    FiFo([X → ··· (α_1 | α_2 |… |.α_{k-1} | α_k)···]): progr([X → ··· (α_1 | α_2 |… |.α_{k-1} | α_k)···])
    otherwise progr([X → ··· (α_1 | α_2 |… |α_{k-1} | .α_k)···]) **endcase**
- Stern und Plus
  - progr([X → ···.(α)*···]) $=_{def}$ **while** nextsym **in** $FI_1(\alpha)$ **do** progr([X → ···.α···]) **od**
  - progr([X → ···.(α)+···]) $=_{def}$ **repeat** progr([X → ···.α···]) **until** nextsym **not in** $FI_1(\alpha)$
- Leeres Wort
  - progr([X → ···.ε···]) $=_{def}$;

RLL(1)-Parser (Recursive descent) (3/4)

Beispiel

- Geg.: E → 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** FI_1((+|–)T) **do** progr([E → T.((+|–)T)]) **od**
= [Def. FI_1; 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**

RLL(1)-Parser (Recursive descent) (4/4)

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 (1/2)

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;
```

```
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;
```

Recursive-descent-Parser (2/2)

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
```

Behandlung von Syntaxfehlern

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*

Fehlerbehandlung in LL(k)-Parsern

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

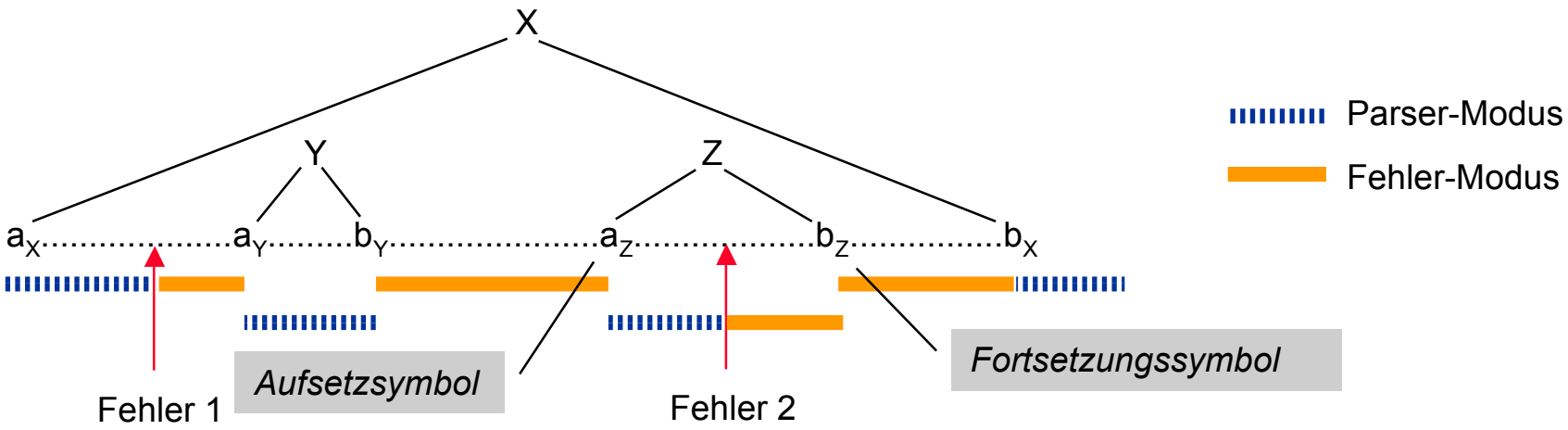
Nachteile

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

Differenziertere Fehlerbehandlung (1/2)

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*“)



Differenziertere Fehlerbehandlung (2/2)

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 *)

Fortsetzen bei LAST-Symbolen (1/3)

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 $w:k =_{def} w_{max(1,n-k+1)}... w_n$
  - $LAST_k(X) =_{def} \{w:k \mid X \underset{R}{\Rightarrow}_{lm}^* w\}$ für $X \in V_N$

Fehlerbehandlungsteil in der Funktion X

- fehler: (* melde Fehler *)
- behandl: **while** nextsym **not in** $LAST_1(X)$ **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)

Fortsetzen bei LAST-Symbolen (2/3)

Für Nichtterminal X ∈ LAST(G)

```
func X (lasts: set of symbol) bool;
    var s: set of symbol;
    begin
      s := lasts ∪ LAST1(X);
      progr([X → .α]);
      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 → ···.X···]) =$_{def}$

```
  if nextsym in FiFo([Y → ···.X···])
    then if not X(s) then goto behandl fi
    else goto fehler fi
```

Für Nichtterminal X ∉ LAST(G)

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

progr([Y → ···.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

Fortsetzen bei LAST-Symbolen (3/3)

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 X0 (lasts: set of symbol) bool; begin ... end;
  func X1 (lasts: set of symbol) bool; begin ... end; ....
  func Xn (lasts: set of symbol) bool; begin ... end;
  begin
      scan;
      if nextsym in FI1(X0)          (* nur generiert, wenn X0 ∈ LAST(G) *)
          then if not X0({„#“}) 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 LAST(G)

Fortsetzen bei FOLLOW-Symbolen (1/2)

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

Fortsetzen bei FOLLOW-Symbolen (2/2)

Für Nichtterminal X ∈ LAST(G)

```
func X (lasts, follows : set of symbol) bool;
    var s: set of symbol;
    begin
       s := lasts ∪ LAST1(X) ∪ follows;
       progr([X → .α]);
       return(true);
    fehler: error(„...“);
    behandl:  while nextsym not in s do scan od;
              if nextsym in LAST1(X)
                 then scan; return(true) fi;
              if nextsym in follows
                 then return(true) fi;
              return(false)
    end
```

Für Nichtterminal X ∉ LAST(G)

```
func X (lasts, follows : set of symbol) bool;
    var s: set of symbol;
    begin
       s := lasts ∪ follows;
       progr([X → .α]);
       return(true);
    fehler: error(„...“);
    behandl: if follows ≠ ∅
             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 ∈ FOLLOW(G)

```
progr([Y → ···.X···]) =def
   if nextsym in FiFo([Y → ···.X···])
     then if not X(s, FO1([Y → ···.X···]))
             then goto behandl fi
     else goto fehler fi
```

Für Vorkommen von X ∉ FOLLOW(G)

```
progr([Y → ···.X···]) =def
   if nextsym in FiFo([Y → ···.X···])
   then if not X(s, ∅)
             then goto behandl fi
   else goto fehler fi
```

LAST und FOLLOW

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 → **if** Bed **then** .An_folge {**fi** | **else** An_folge **fi**}]) = {"**fi**", "**else**"}
FO_1([If_Anw → **if** Bed **then** An_folge {**fi** | **else** .An_folge **fi**}]) = {"**fi**"}
FO_1([While_Anw → **while** Bed **do** .An_folge **od**]) = {"**od**"}
FO_1([Repeat_Anw → **repeat** .An_folge **until** Bed]) = {"**until**"}
FO_1([Prog → .An_folge]) = {"#"}
FO_1([An_Folge → .Anw {; Anw}*]) = FO_1([An_Folge → Anw {; .Anw}*]) = {";", "**od**", "**until**", "**fi**", "**else**", "#"}
FO_1([Proz_Aufruf → **call** name (.Ausdr_folge)]) = {")"}
FO_1([Anw → .If_Anw | While_Anw | Repeat_Anw | Proz_Aufruf | Wertzuweisung]) =
FO_1([Anw → If_Anw | .While_Anw | Repeat_Anw | Proz_Aufruf | Wertzuweisung]) =
FO_1([Anw → If_Anw | While_Anw | .Repeat_Anw | Proz_Aufruf | Wertzuweisung]) =
FO_1([Anw → If_Anw | While_Anw | Repeat_Anw | .Proz_Aufruf | Wertzuweisung]) =
FO_1([Anw → If_Anw | While_Anw | Repeat_Anw | Proz_Aufruf | .Wertzuweisung]) =

FO_1([Anw → .(If_Anw | While_Anw | Repeat_Anw | Proz_Aufruf | 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
```

Aufsetzen bei Anfangssymbolen (1/4)

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

Aufsetzen bei Anfangssymbolen (2/4)

Anfangssymbol

- Sei $G = (V_N, V_T, p, S)$ rrkfG und $X \in V_N$. Weiter sei
  - $prod(X) =_{def} \{Y \mid X \underset{R}{\Rightarrow}_{lm}^{+} \cdots Y \cdots\}$:
    Menge der aus X produzierbaren Nichtterminale
  - $prod\_b(X) =_{def} prod(X) \cap BEG(G)$:
    Teilmenge der (aus X produzierbaren) „Aufsetzsymbole“
- Menge der **Anfangssymbole**
  - Für Y im Kontext X: $BEGSYMBS(X)_Y =_{def} FI_1(Y) \setminus \bigcup_{Z \in prod(X),\, Z \neq Y} FI_1(Z)$
  - Im Kontext X: $BEGSYMBS(X) =_{def} \bigcup_{Y \in prod\_b(X)} BEGSYMBS(X)_Y$

Die für Y im Kontext X „eindeutigen“ Anfänge

Es gilt

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

Aufsetzen bei Anfangssymbolen (3/4)

Modifiziertes Hauptprogramm für Fortsetzung bei Anfangssymbolen

begin
  scan;
  if nextsym **in** FiFo([S‘ → .X_0]))
    then if not X_0({„#“}, ∅, BEGSYMBS(X_0)) **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(X_0) **then** recover(X_0) **else** scan **fi** **od**
end

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

case nextsym **in**
  BEGSYMBS(X)$_{Y_1}$: Y_1(s, ∅, b)
  BEGSYMBS(X)$_{Y_2}$: Y_2(s, ∅, b) ...
  BEGSYMBS(X)$_{Y_n}$: Y_n(s, ∅, b)
endcase

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

Aufsetzen bei Anfangssymbolen (4/4)

Für Nichtterminal X ∈ LAST(G)

```
func X (lasts, follows, begins : set of symbol) bool;
    var s, b: set of symbol;
    begin
       s := lasts ∪ LAST1(X) ∪ follows;
       b := begins ∪ BEGSYMBS(X);
       progr([X → .α]);
       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 LAST1(X)
                 then scan; return(true) fi;
              if nextsym in follows
                 then return(true) fi;
              return(false)
    end
```

Für Vorkommen von X ∈ FOLLOW(G)

X(s, FO$_1$([Y → ···.X···]), b) für X(s, FO$_1$([Y → ···.X···]))

Für Nichtterminal X ∉ LAST(G)

```
func X (lasts, follows, begins : set of symbol) bool;
    var s, b: set of symbol;
    begin
       s := lasts ∪ follows;
       b := begins;
       if follows ≠ ∅ then b := b ∪ BEGSYMBS(X) fi;
       progr([X → .α]);
       return(true);
    fehler: error(„...“);
    behandl: if follows ≠ ∅
                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 ∉ FOLLOW(G)

X(s, ∅, b) für X(s, ∅)

Ansonsten wie bei Fortsetzen mit FOLLOW-Symbolen

Aufsetzen bei Vorgängersymbolen

Vorgängersymbole

- Charakteristische Symbole, die einem Vorkommen eines Nichtterminals immer vorangehen
- Rekursive Definition analog zu $FOLLOW_1$
  - Sei $[X \rightarrow \alpha\ .\beta\gamma]$ erweitertes kontextfreies Item
  - $PRECEDE([X \rightarrow \alpha\ .\beta\gamma]) =_{def} \{a \in V_T \mid S \underset{R}{\Rightarrow}_{lm}^{*} wX\delta \underset{R}{\Rightarrow}_{lm} w\alpha\beta\gamma\delta \underset{R}{\Rightarrow}_{lm}^{*} wua\beta\gamma\delta\}$
    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 $BEG(G) \cap PREC(G)$ als Aufsetzsymbole (Eindeutigkeit)
- Jedes Aufsetzsymbol sollte eindeutig ein Nichtterminal bestimmen, für das rekursiv in den Parser-Modus gegangen wird

Anfangs- und Vorgängersymbole (1/2)

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

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

Modifikation des Generierungsschemas für ELL(1)-Parser

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

case nextsym **in**
  $BEGS(X)_{Y_1}$: $Y_1(s, \varnothing, b)$ **...**
  $BEGS(X)_{Y_n}$: $Y_n(s, \varnothing, b)$
  $PRECS(X)_{Z_1}$: **begin** scan; $Z_1(s, \varnothing, b)$ **end ...**
  $PRECS(X)_{Z_m}$: **begin** scan; $Z_m(s, \varnothing, b)$ **end**
endcase

Anfangs- und Vorgängersymbole (2/2)

Beispiel (Anweisungsgrammatik)

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

- Aufsetzen bei Anfangs- /Vorgänger-Symbolen
  - BEG(G) = {Anw}
  - PREC(G) = {[An_Folge → Anw {; **.**Anw}*]}, damit
  - PRECEDE([An_Folge → Anw {; **.**Anw}*]) = {";"},
- Berechnung von BEGS, PRECS, BEGSYMBS
  - Sei VORG(Anw)
    = {X ∈ V_N | Anw ∈ prod(X)}
    = {Anw, If_Anw, While_Anw, Repeat_Anw, An_Folge, Progr}.
  - $B1(X)_{Anw}$ = {"**if**", "**while**", "**repeat**", "**call**", "**name**"} für alle X ∈ VORG(Anw); $B1(X)_Y$ = Ø, sonst
  - $P1(X)_{Anw}$ = {";"} für alle X ∈ VORG(Anw); $P1(X)_Y$ = Ø, sonst
  - B2 = B1
  - P2 = P1
  - BEGS = B1
  - PRECS = P1
  - BEGSYMBS(X) = {"**if**", "**while**", "**repeat**", "**call**", "**name**", ";"} für alle X ∈ VORG(Anw);
    BEGSYMBS(X) = Ø, sonst