Compilerbau

LL(K)-Parser Teilfolien

Prof. Dr. Franz-Karl Schmatzer
schmatzf@dhbw-loerrach.de

Literatur

- C.Wagenknecht, M.Hielscher; Formale Sprachen, abstrakte Automaten und Compiler; 3.Aufl. Springer Vieweg 2022;
- U.Meyer; Grundkurs Compilerbau; Rheinwerkverlag, 1. Aufl. 2021
- A.V.Aho, M.S.Lam,R.Savi,J.D.Ullman, *Compiler – Prinzipien,Techniken und Werkzeuge*. 2. Aufl., Pearson Studium, 2008.
- Güting, Erwin; *Übersetzerbau –Techniken, Werkzeuge, Anwendungen*, Springer Verlag 1999

Top-Down-Analyse

Agenda

- Prinzip der Top-Down-Analyse
- Beseitigen von Linksrekursionen

Top-Down-Analyse

Prinzip – Beispiel Grammatik

- Betrachte folgende Grammatik (Ausschnitt)

stmt	→	assignment | cond | loop
assignment	→	id :=expr
cond	→	if boolexpr then stmt fi |
		if boolexpr then stmt else stmt fi
Loop	→	while boolexpr do stmt od
expr	→	boolexpr | numexpr
boolexpr	→	numexpr cop numexpr
numexpr	→	numexpr + term | term
term	→	term*factor | factor
factor	→	id | const | (numexpr)

- Damit lassen sich Ausdrücke formulieren wie:
  - if b > 0 then a:=1 fi
  - while i<n then … do

5

Einleitung

Top-down-Analyse

if id[1] cop[>] const[2] then id[2] := const[1] end

- Bei Beginn zeigt der Zeiger des nächsten Tokens auf if
- Start mit der Start Regel: stmt: assignment | cond
- Dann errät man cond: if … | if …
- Nun erhält man eine match mit if
- Der Zeiger rutscht eins weiter und zeigt nun auf id[1]
- Problem:
  - wie weiß man, welche Regeln man anwenden soll?
  - Man möchte nicht Raten!

if id[1] cop[>] const[2] then id[2] := const[1] end

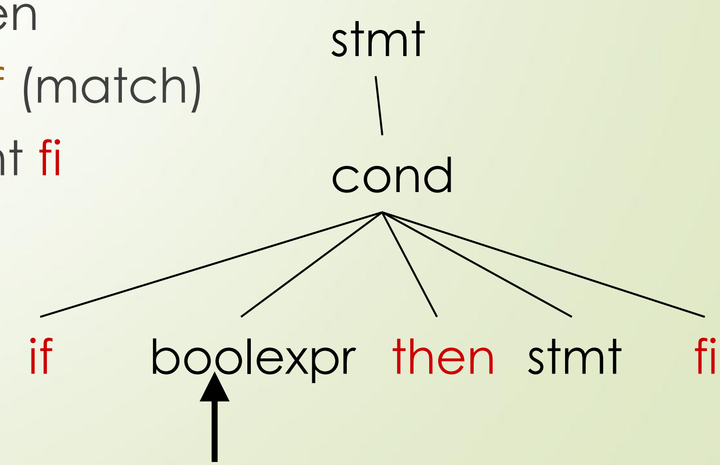
↑

Top-Down-Analyse

Prinzip – Start mit stmt

- Der Ausdruck if b>0 then a:=1 fi wird nach dem Lexer zu:

  if id cop const then id:= const fi
- Start mit stmt
  - Zeiger zeigt auf das erste Token
- stmt ≠ if ⇒
  - expandieren von stmt → assignment | cond | loop
  - Auswahl assignment: assignment ≠ if ⇒ expandieren
  - id :=expr ⇒ id ≠ if Terminal ≠ Token (**Sackgasse** ⇒ backtracking)
  - Auswahl cond: cond ≠ if ⇒ expandieren
  - Auswahl if boolexpr then stmt fi ⇒ if = if (match)
  - Weitermachen mit if boolexpr then stmt fi
  - Zeiger rückt auf das nächste Token



Top-Down-Analyse

Prinzip – Problem Endlosschleife

Weitermachen mit if boolexpr then stmt fi

- Expandieren von boolexpr zu numexpr cop numexpr
- numexpr expandieren zu numexpr + term
- numexpr expandieren zu numexpr + term
- …
- **Endlosschleife!**

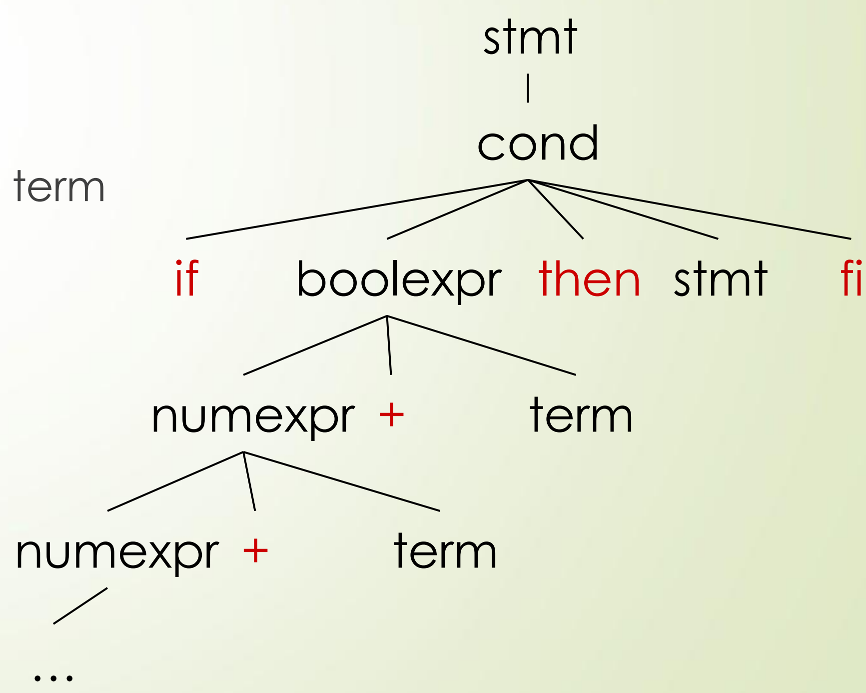
Die Regel: numexpr → numexpr + term | term

ist linksrekursiv

Top-Down-Analyse kann

linksrekursive Regeln einer Grammatik

nicht managen.



Top-Down-Analyse

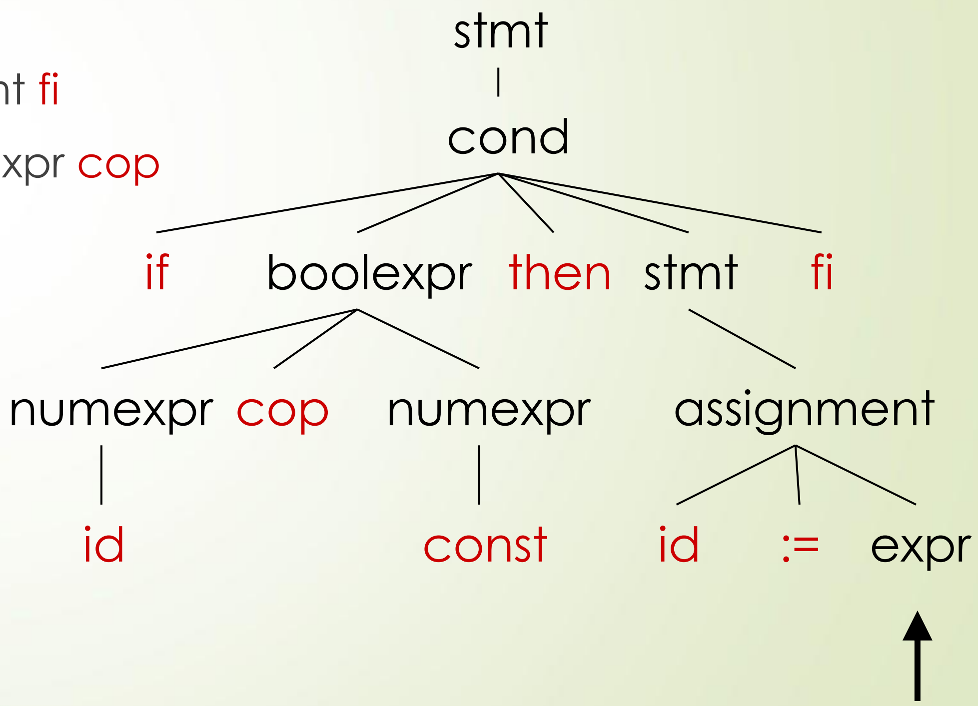
Prinzip - Mitte in der Analyse

if id cop const then id:= const fi (Token Strom)

- Wir setzen im Folgenden die Regel numexpr durch: numexpr $\rightarrow$ id | const

Weitermachen mit if boolexpr then stmt fi

- Expandieren von boolexpr zu numexpr cop numexpr
- numexpr zu id, $\Rightarrow$ match
- Zeiger rückt eins weiter
- …
- Der Zeiger befindet sich nun
- über expr



Top-Down-Analyse

Prinzip - Problem Sackgasse

if id cop const then id:= const fi
(Token Strom)

const muss gefunden werden !

Expandieren von expr zu boolexpr

boolexpr zu numexpr cop numexpr

numexpr zu id ⇒ kein match

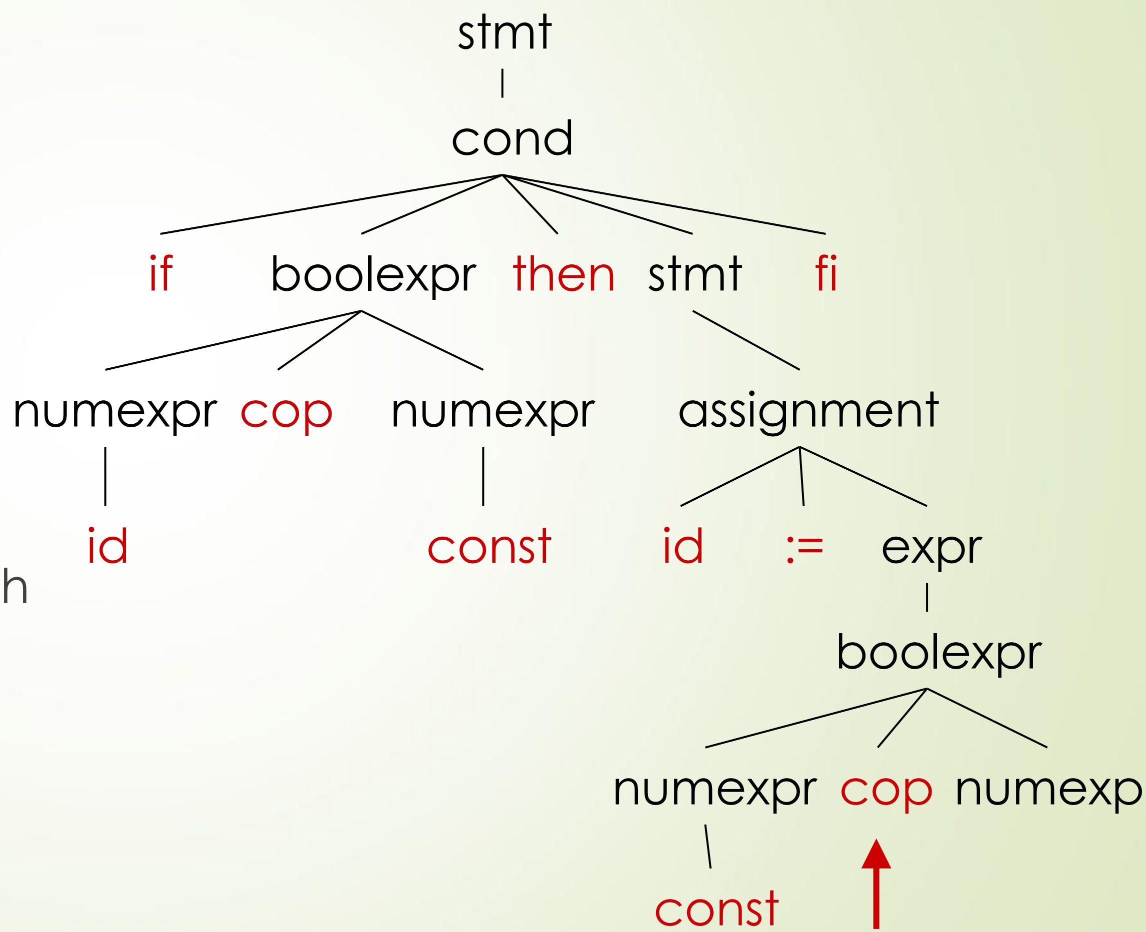
numexpr zu const ⇒ match

Zeiger rückt eins weiter.

Vergleich von cop mit fi

⇒ kein match

Backtracking bis zu expr

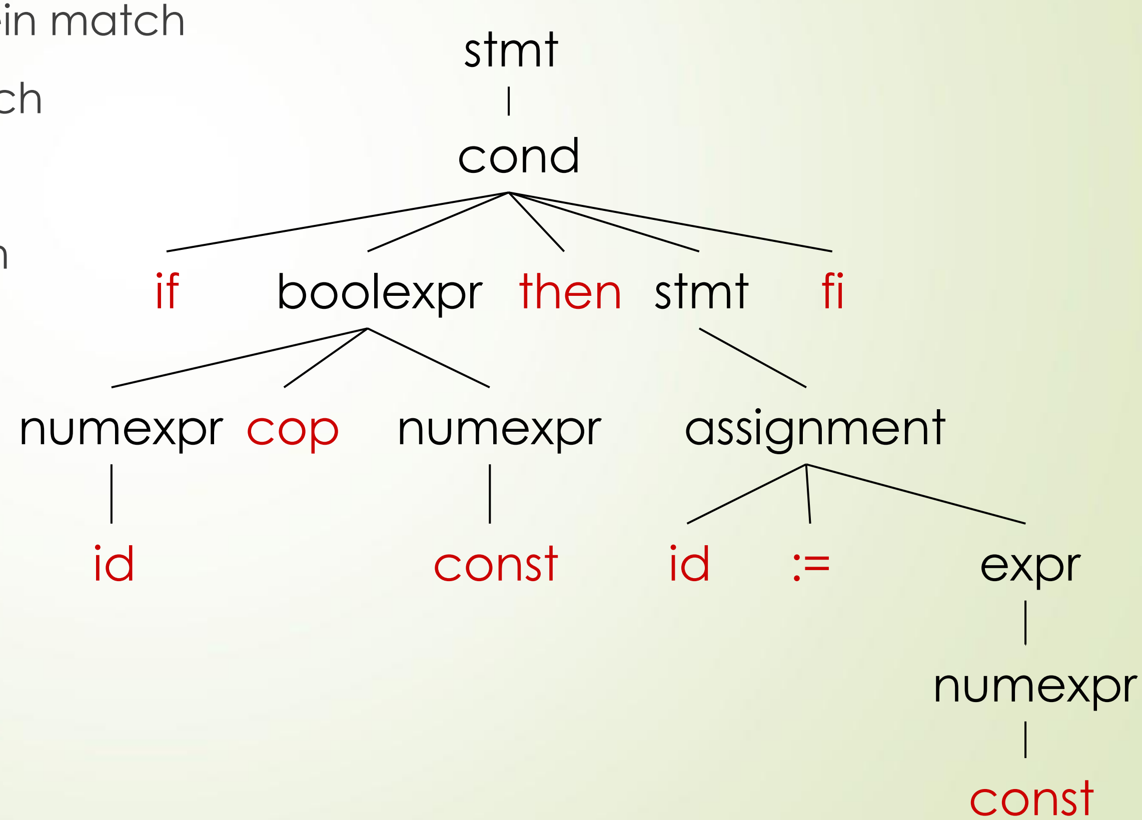


Top-Down-Analyse

Prinzip – Ende der Analyse

if id cop const then id:= const fi (Token Strom)

- Expandieren expr zu numexpr
- numexpr dann zu id ⇒ kein match
- numexpr zu const ⇒ match
- Zeiger rückt eins weiter.
- Vergleich fi mit fi ⇒ match
- Ende der Analyse



Top-Down-Analyse

Prinzip – Probleme

- Bei der Top-Down-Analyse sind 2 Probleme aufgetreten
  - Endlosschleifen durch linksrekursive Grammatiken und
  - Verlauf in Sackgasse
- Beide Probleme lassen sich beheben
  - Linksrekursive Grammatiken kann man so umschreiben, dass keine Linksrekursionen mehr vorkommen, ohne die Sprache zu ändern.
  - Den Verlauf in Sackgassen, kann man beheben indem man eine vorausschauende Syntaxanalyse betreibt. Dies führt zu den LL(K) Grammatiken

Beseitigen von Linksrekursionen

- Es gibt zwei Arten von Linksrekursionen

  $A \rightarrow A\alpha$ (direkte Linksrekursion)

  $A \Rightarrow^* A\alpha$ (indirekte Linksrekursion)

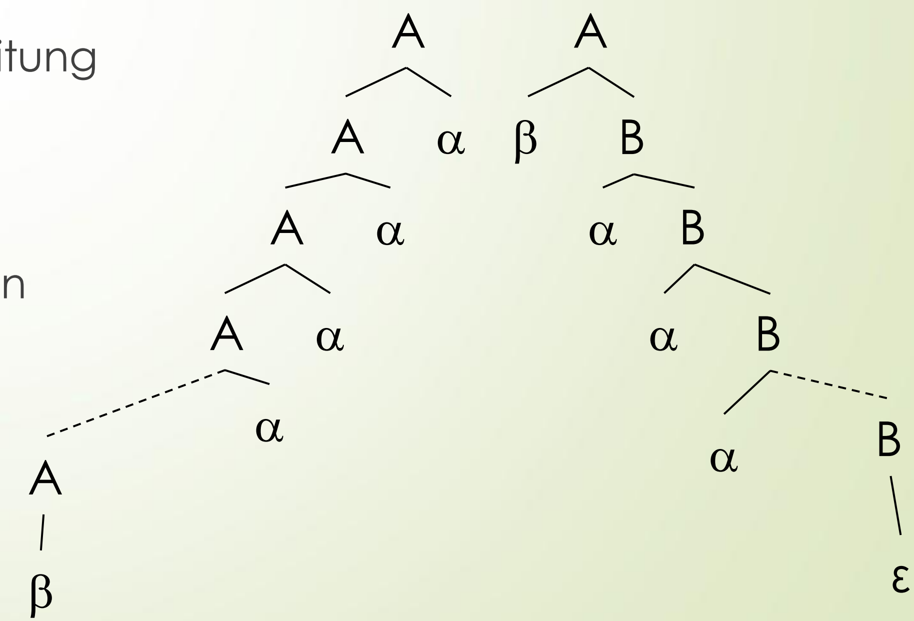
- Die direkte Linksrekursion ist für die Praxis häufig relevant
- Was passiert bei der Ableitung
  - $A \rightarrow A\alpha \mid \beta$
  - Das Wort $w = \beta\alpha^*$ hat eine Ableitung

    Z.B. $w = \beta\alpha\alpha\alpha\alpha$

  - Dies kann man durch eine rechtsrekursive Variante ersetzen

    $A \rightarrow \beta B$

    $B \rightarrow \alpha B \mid \varepsilon$



Beseitigen von direkten Linksrekursionen

- Anwenden dieser Technik auf die Regel:

  numexpr → numexpr + term | term

  ergibt:

  numexpr → term nexpr | term

  nexpr → + term nexpr | ε

- Anwenden dieser Technik auf die Regel:

  term → term * factor | factor

  ergibt:

  term → factor nterm | factor

  nterm → * factor nterm | ε

$A \to A\alpha \mid \beta$

$A \to \beta B$

$B \to \alpha B \mid \varepsilon$

Beseitigen von direkten Linksrekursionen

- Diese Regel lässt sich verallgemeinern zu:
  - Sei:

    $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \beta_3 .. \mid \beta_k$

  Dies wird ersetzt zu:

    $A \rightarrow \beta_1 B \mid \beta_2 B \mid \beta_3 B \mid .. \mid \beta_k B$

    $B \rightarrow \alpha_1 B \mid \alpha_2 B \mid \alpha_3 B \mid .. \mid \alpha_n B \mid \varepsilon$

Beseitigen aller Linksrekursionen

- Algorithmus nach Aho et al.

  Bringe alle Nichterminale in eine Reihenfolge $A_1,\ldots A_n$

  For(für jedes i von 1 bis n){

  For(für jedes k von 1 bis i-1) {

  Ersetzte jede Produktion der Form $A_i \rightarrow A_k\gamma$ durch

  $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$, wobei $A_k \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ alle aktuellen A_k Produktionen sind.

  }

  Eliminiere die unmittelbaren Linksrekursionen.

  }

Beseitigen aller Linksrekursionen

Beispiel

- Gegeben: $S \to Aa \mid b$, $A \to Ac \mid Sd \mid \varepsilon$
  - Diese Produktion hat keine direkte Linksrekursion in S, jedoch eine unmittelbare Rekursion $S \to Aa \to Sda$
- Nach dem Algorithmus der vorherigen Seite:
  - Ordnen der Produktionen: S, A
  - (i=1, d.h. Produktion S) S hat keine unmittelbare Linksrekursion, d.h. für i=1 der äußeren Schleife passiert nichts.
  - (i=2, d.h. Produktion A) Hier muss man nun S in der Regel $A \to Sd$ ersetzen durch $S \to Aa \mid b$ und man erhält nun $A \to Ac \mid Aad \mid bd \mid \varepsilon$
  - Eliminieren der unmittelbaren Linksreduktionen in A ergibt nun folgende Produktionsregeln

    $S \to Aa \mid b$,

    $A \to bdB \mid B$,

    $B \to cB \mid adB \mid \varepsilon$

Aufgabe Linksrekursion

- Entfernen Sie die Linksrekursionen aus folgenden Grammatiken
- G1 = ({S,A},{a,b,c,d},{S $\rightarrow$ Aa | b, A $\rightarrow$ Ac | Ad | b}, S)
- G2 = ({E,T,F},{a,+,*,(,)},{E$\rightarrow$E+T | T, T $\rightarrow$ T*F | F, F $\rightarrow$ (E) | a}, E)

- Erstellen Sie die Grammatiken G1 mit und ohne Linksrekursion in FLACI und nehmen Sie das Wort w=bcddcdca für G1. Wie sehen der Ableitungsbäume aus.
- Erstellen Sie die Grammatiken G2 mit und ohne Linksrekursion in FLACI und nehmen Sie das Wort w=a*(a+a*a+a+a)*a*a für G2. Wie sehen die Ableitungsbäume aus.

Lösung Linksrekursion

$S \to Aa \mid b$

$A \to Ac \mid Ad \mid b$

wird zu:

$S \to Aa \mid b$

$A \to bB$

$B \to cB \mid dB \mid \varepsilon$

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid a$

wird zu:

$E \to T\,E'$

$E' \to +\,T\,E' \mid \varepsilon$

$T \to F\,T'$

$T' \to *\,F\,T' \mid \varepsilon$

$F \to (E) \mid a$

$A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid .. \mid \beta_k$

Dies wird ersetzt zu:

$A \to \beta_1 B \mid \beta_2 B \mid .. \mid \beta_k B$

$B \to \alpha_1 B \mid \alpha_2 B \mid .. \mid \alpha_n B \mid \varepsilon$

LL(K) Grammatiken

- Definition

  Sei $L \subseteq \Sigma^*$ eine beliebige Sprache und sei $k > 0$. Dann ist

  $Start_k(L) := \{w \mid w \in L \text{ und } |w| < k \text{ oder } \exists\, wu \in L \text{ und } |w| = k\}$

  Für ein Wort $v \in \Sigma^*$ sei

  $start_k(v) :=$ v, falls $|v| < k$ sonst u, falls $v=ut$ mit $|u| = k$

- Eine kontextfreie Grammatik $G = (N,\Sigma,P,S)$ heißt LL(K) Grammatik, wenn für die Produktionen $A \rightarrow \alpha \mid \beta$ gilt aus:

  $S \Rightarrow^* wA\sigma \Rightarrow w\alpha\sigma \Rightarrow^* wx$ und

  $S \Rightarrow^* wA\sigma \Rightarrow w\beta\sigma \Rightarrow^* wy$ und

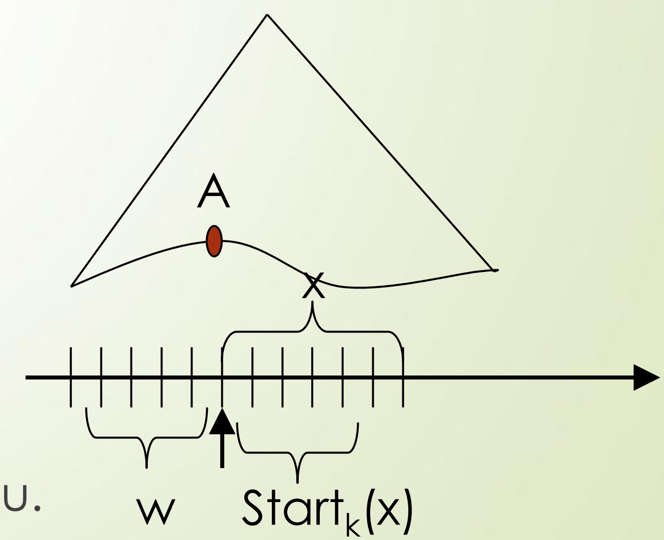
  $start_k(x) = start_k(y)$

  folgt $\alpha = \beta$

- Bezeichnung LL(k):

  Lesen von links nach rechts mit

  Linksableitung und k Token Vorausschau.



LL(K) Grammatiken

- Eine kontextfreie Grammatik $G = (N,\Sigma,P,S)$ heißt **starke** LL(K) Grammatik, wenn für die Produktionen $A\rightarrow \alpha \mid \beta$ gilt aus:

  $S \Rightarrow^* wA\sigma \Rightarrow w\alpha\sigma \Rightarrow^* wx$ und

  $S \Rightarrow^* uA\tau \Rightarrow u\beta\tau \Rightarrow^* uy$ und

  $start_k(x) = start_k(y)$

  folgt $\alpha = \beta$
- Bem:
  - Bei der starken LL(K) Grammatik ist die Umgebung von A unerheblich
  - D.h. das gelesene Wort (w bzw. u) wie auch die Nachfolgezeichen (σ, τ) sind für die Auswahl der Produktion nicht erheblich.
  - **Sind für den Compilerbau interessant**
- Als nächsten werden die LL(K) Grammatiken näher charakterisiert

Aufgabe LL(K)-Grammatik

- Zeigen Sie, dass

  $G = (\{S\}, \{a,b\}, P, S)$ mit $P = \{S \rightarrow aSab \mid aSabb \mid b\}$

keine LL(k) Grammatik ist.

Aufgabe LL(K)-Grammatik

Lösung

- Zeigen Sie, dass folgende Grammatik keine LL(k) Grammatik ist.

  G = ({S}, {a,b},P,S) mit P = {S→aSab | aSabb | b}

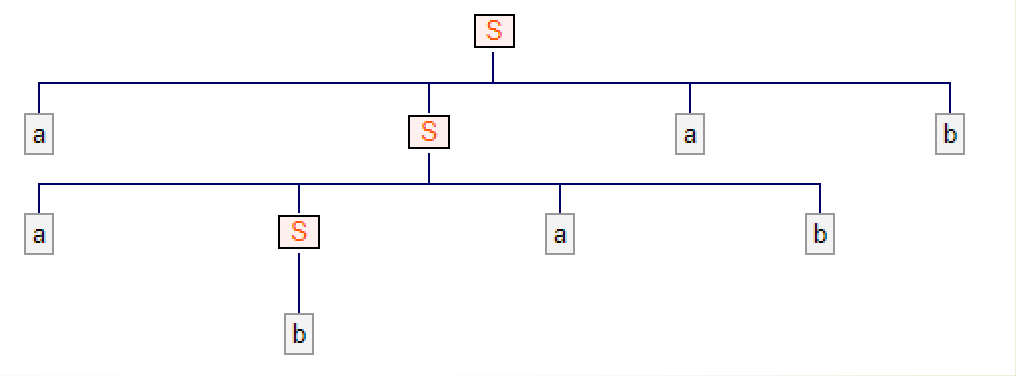
Beispiel: w1 = ax

w1 = aababab

w1 = ax

start(x) = a

S -> aSab



w2 = aababbab

w2 = a y

start(y) = a

S -> aSabb