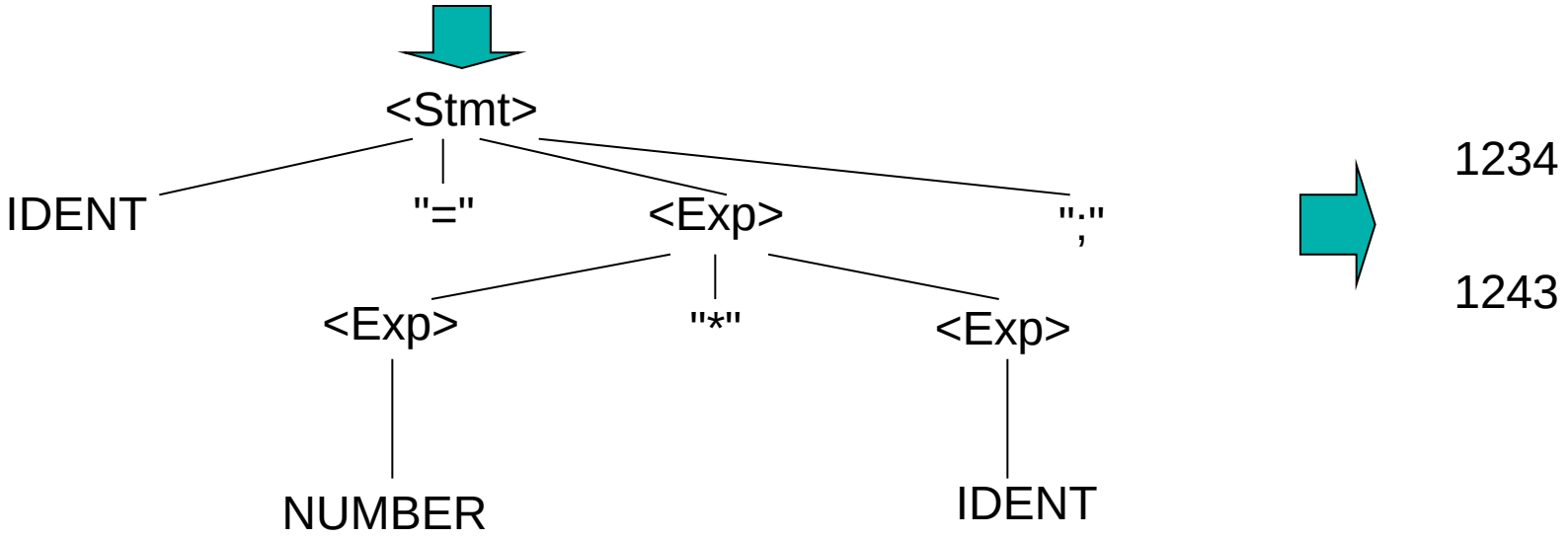
Syntax-Analyse

- **aus Einführung:** Parser
  - Programmteil, der Syntax-Analyse durchführt.
  - Eingabe: Tokenfolge von Scanner
  - Ausgabe: Syntaxbaum, falls Syntax o.k.
    meist in linearer Präfix-Darstellung
- Beispiel

`IDENT "=" NUMBER "*" IDENT ";"`



Syntaktische Einheiten

- **Deklarationen**
  - Konstantendeklarationen
    `final int eins = 1, zwei = 2;`
  - Variablendeklarationen
    `int bez1 = 17, zwei;`
- **Anweisungen**
  - Zuweisungen, print, Block, if-Verzweigung, while-Schleife
- **Ausdrücke**
  - z.B.: `7 * x;`
- **später:**
  - Funktions- & Prozedurdeklarationen `func f(int i){...}`
  - Funktions- & Prozeduraufrufe `f(5*x);`
  - ...

Unterschied: lex. / synt. Einheiten

- **lexikalische Einheiten**
  - linear angeordnet
  - Typ3-Strukturen
  - durch RAs beschreibbar
  - durch EAs erkennbar
- **syntaktische Einheiten**
  - verschachtelt -> Baumstrukturen
  - Typ2-Sprachen
  - durch CFG / Syntaxdiagramme / EBNF beschreibbar
  - durch PDA erkennbar

Problem bei PDAs?

deterministische Simulation

- **bei EAs :** **$DEA(\Sigma) = NEA(\Sigma)$**
- **bei PDAs:** **$DPDA(\Sigma) \subset PDA(\Sigma)$**
- **Wdh.: Deterministische Simulation bei PDAs**
  - Backtrackalgorithmus **exponentieller Zeitaufwand**
    2 Keller
  - Tabularmethoden **Zeit: $O(n^3)$**
    Cocke/Younger/Kasami **Platz: $O(n^2)$**

- **aber zum Glück**
  - Syntax von PSn durch spezielle CFGs beschreibbar
  - Analyse in linearer Komplexität möglich
  - hierzu: **DPDAs mit look-ahead auf der Eingabe**

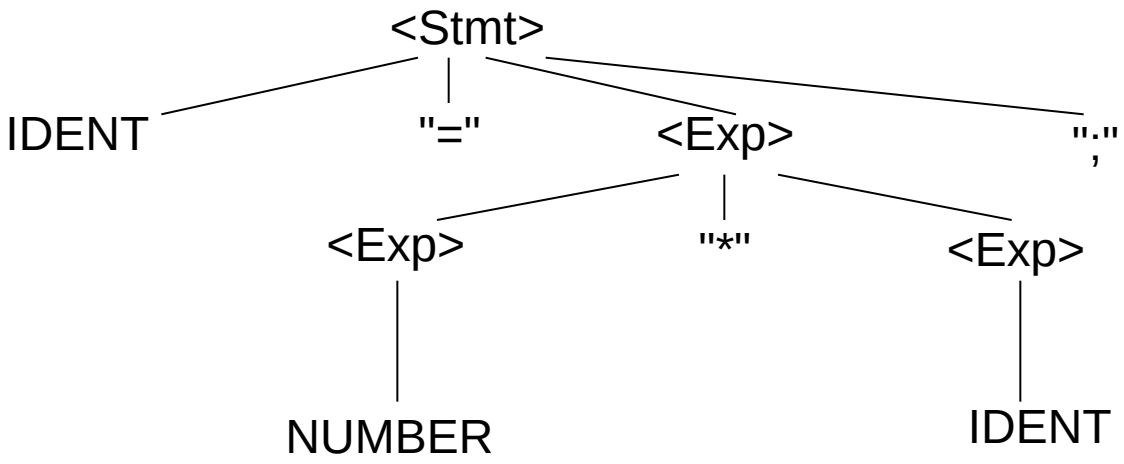
2 Analysemethoden mit PDAs

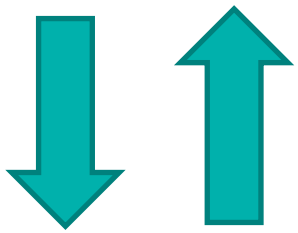
- **immer: PDA konstruiert Ableitungsbaum:**
- **Top-Down**
  - Wurzel -> Blätter
  - liefert Linksanalyse

1234

- **Bottom-Up**
  - Blätter -> Wurzel
  - liefert gespiegelte Rechtsanalyse (denn lese Eingabe von links)

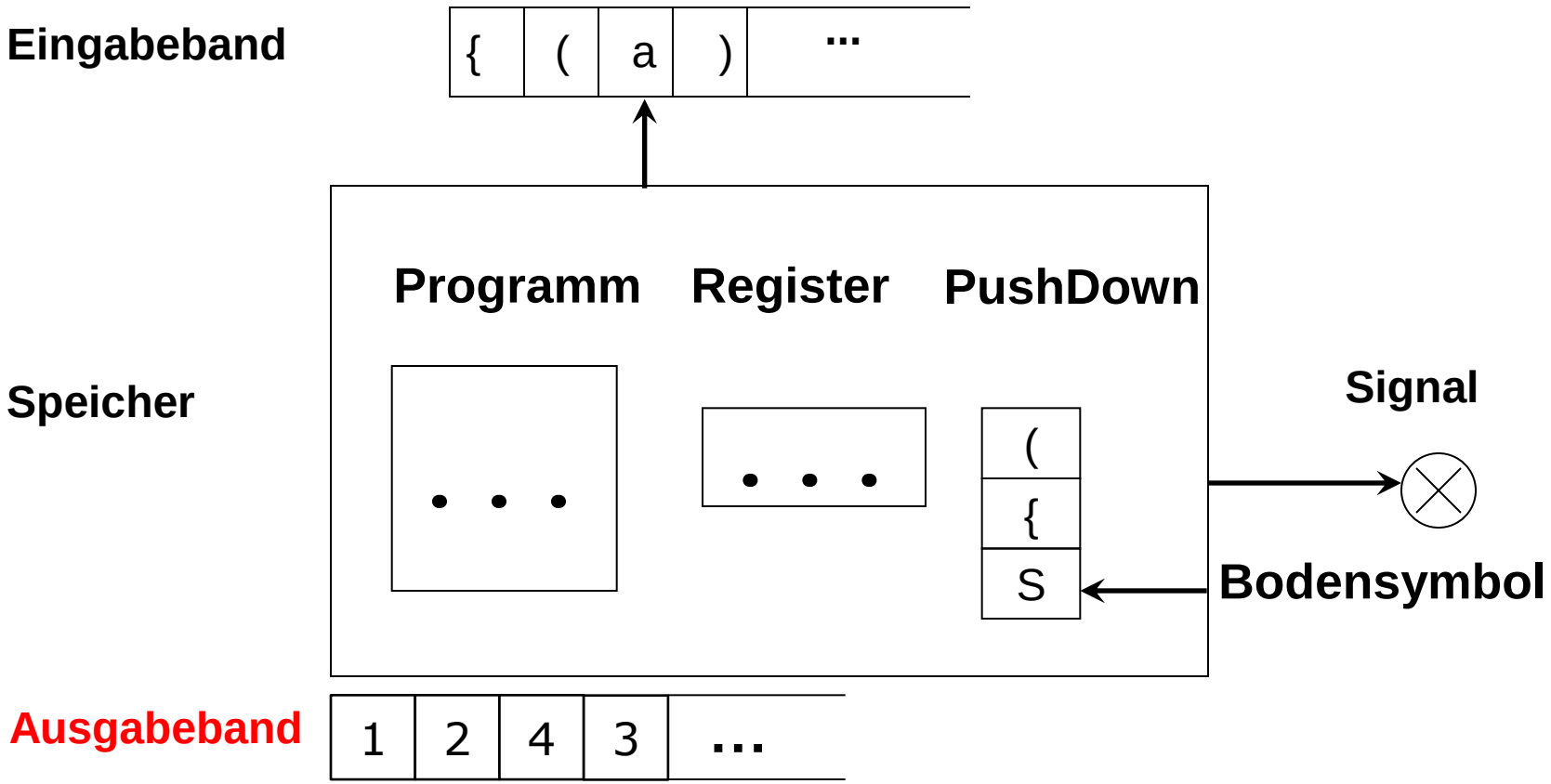
3421



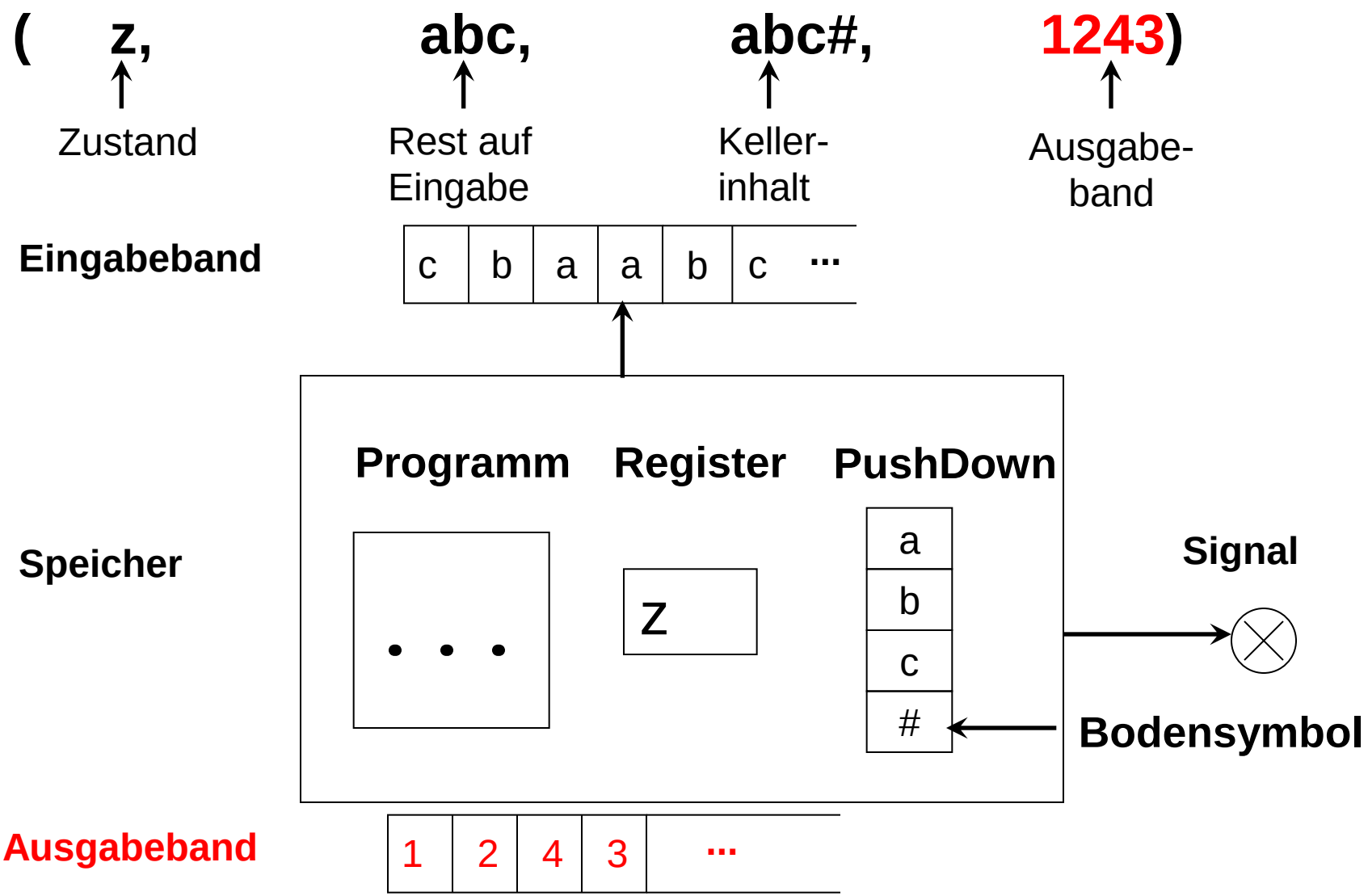


Wdh.: Syntax-Analyse PDA

- **Grundlage: Alphabet (endliche Menge) Alph**
- **Aufgabe: Überprüfung, ob ein Wort in einer Typ2-Sprache über Alph liegt & Syntaxbaumausgabe**



Konfiguration eines PDAs



Wdh.: Kontextfreie Grammatiken

- **Syntaxbeschreibungsmechanismus**
- **Ausgangspunkt für Analyse-PDA**
- **Bsp.: GSE = (V,Σ,S,R) mit**
  - **Variablenmenge** V = {<Stmt>,<Exp>} (Nichtterminalsymbole)
  - **Terminalmenge** Σ = {IDENT, NUMBER, "=", ";", "*"}
  - **Startsymbol** S = <Stmt>
  - **Regelmenge** R =

```
{
        <Stmt>  ->   IDENT "=" <Exp> ";        (1)
        <Exp>   ->    <Exp> "*" <Exp>          (2)
        <Exp>   ->    NUMBER                   (3)
        <Exp>   ->    IDENT                    (4)
}
```

Links- (Rechts-) Ableitungen

- **geg.: CFG G = (V,Σ,S,R)**
- **Def.: (Satzform)**
  Eine Satzform zu G ist ein Wort, das Variablen und Terminalsymbole von G enthalten kann, also ein Element aus $(V \cup \Sigma)^*$
- **Def.: (Links- (Rechts-) Ableitungen)**
  Bei einer Links- (Rechts-) Ableitung wird eine Satzform S1 in eine Satzform S2 überführt (Notation S1 =l> (=r>) S2), indem die am weitesten links (rechts) stehende Variable in S1 durch eine zugehörige rechte Regelseite ersetzt wird.
- **Bsp.: (Links- (Rechts-) Ableitungen)**
  - Grammatik GSE
  - `IDENT "=" NUMBER "*" IDENT ";"` soll erzeugt werden

Beispielableitungen

- **Startsatzform <stmt>**
- **Linksableitung**

<stmt>		Regel
=l>	IDENT "=" <Exp> ";"	(1)
=l>	IDENT "=" <Exp> "*" <Exp> ";"	(2)
=l>	IDENT "=" NUMBER "*" <Exp> ";"	(3)
=l>	IDENT "=" NUMBER "*" IDENT ";"	(4)

- **Rechtsableitung**

<stmt>		
=r>	IDENT "=" <Exp> ";"	(1)
=r>	IDENT "=" <Exp> "*" <Exp> ";"	(2)
=r>	IDENT "=" <Exp> "*" IDENT ";"	(4)
=r>	IDENT "=" NUMBER "*" IDENT ";"	(3)

Notationen

Ausgangspunkt: CFG G = (V,Σ,S,R)

a, b, c $\in \Sigma$

v, w, x, y $\in \Sigma^*$

A, B, C, S $\in$ **V**

$\alpha, \beta, \gamma \in (V \cup \Sigma)^*$

S ist Startsymbol

Top-Down Analyse PDA (nichtdeterministisch)

- **geg.: CFG G = (V,Σ,S,R) mit |R|=p**
- **Def.: Top-Down Analyse-PDA zu G**
  - Ein Zustand: **z**
  - Eingabealphabet von G: Σ
  - Kelleralphabet: **V** ∪ Σ
  - Ausgabealphabet: {1,...,p}
  - Einzelschrittrelation:
    - Ableitungsschritt: Sei A -> α die i-te Regel in R
      - (z, w, Aγ, $o_1...o_m$) => (z, w, αγ, $o_1...o_m$i)
    - Vergleichsschritt: Sei a ∈ Σ
      - (z, aw, aγ, $o_1...o_m$) => (z, w, γ, $o_1...o_m$)
  - Anfangskonfiguration für w∈Σ* : (z, w, S,)

Nichtdet.?

- **Satz: $o_1...o_m$ Linksableitung von w gdw. (z,w,S,) =l>* (z, , ,$o_1...o_m$)**

Beispielanalyse (nichtdeterministisch)

- **geg.:** - Beispielgrammatik GSE
  - Tokenfolge: `IDENT "=" NUMBER "*" IDENT ";"`

- **Top-Down Analyserechnung:**

=> (z, IDENT"="NUMBER"*"IDENT";",<Stmt>,)

=> (z, IDENT"="NUMBER"*"IDENT";", IDENT "=" <Exp> ";" , 1)

=> (z, "="NUMBER"*"IDENT";", "=" <Exp> ";" , 1)

=> (z, NUMBER"*"IDENT";", <Exp> ";" , 1)

=> (z, NUMBER"*"IDENT";",<Exp> "*" <Exp> ";" , 12)

=> (z, NUMBER"*"IDENT";",NUMBER "*" <Exp> ";" , 123)

=> (z, "*"IDENT";", "*" <Exp> ";" , 123)

=> (z, IDENT";", <Exp> ";" , 123)

=> (z, IDENT";", IDENT ";" , 1234)

=> (z, ";", ";" , 1234)

=> (z, , , 1234)

LL-Analyse

- **look-ahead auf Eingabe löst Nichtdeterminismus auf**
- **Einschränkung der Grammatiken auf**

  LL(k)-Grammatiken

  – durch look-ahead auf Eingabe um k Symbole wird Regelauswahl eindeutig

  – LL(k) steht für
    - lese von ***L***inks nach rechts auf Eingabe
    - erzeuge ***L***inksableitung
    - look-ahead auf Eingabe um ***k*** Symbole

Problemstellung

- **Wie kann man durch look-ahead auf Eingabe entscheiden, welche Regel angewendet wird?**
- **Bsp.: (Mini-Java Syntax)**
  - GSE nicht LL(k) für bel. k, da linksrekursiv, darum anders:
  - Regeln
    - <Term> -> NUMBER (1)
    - <Term> -> IDENT (2)
    - <Term> -> "(" <Exp> ")" (3)
  - Konfiguration (z, `IDENT ";"`, <Term> ";" , 123)
  - alle drei Regeln anwendbar
  - look-ahead auf Eingabe um 1: `IDENT` **Wann k=2?**
  - nur Regel (2) produziert als nächstes IDENT
  - Grammatik ist LL(1): look-ahead um 1 reicht

LL(2)-Beispiel

- **erstes Terminalsymbol von mehreren Regeln erzeugbar**
- **Bsp.: Doppelklammerung**
  - <Term> -> "(" NUMBER ")" (1)
  - <Term> -> "(" IDENT ")" (2)
  - <Term> -> "(" "(" <Exp> ")" ")" (3)
  - Konfiguration (z, "("IDENT")"";", <Term> ";" , 123)
  - alle drei Regeln erzeugen "("
  - mit look-ahead um 2 fällt wieder Entscheidung für Regel (2)

Wie LL(k)-Eigenschaft formalisieren?

$first_k$-Mengen

- **interessant:**
  die ersten k Symbole aus Σ, die aus Satzform α erzeugt werden können -> $first_k(\alpha)$ können mehrere sein
- **Bsp.:**
  - geg.: Regeln der vorigen Folie
  - Satzform α = <Term> "+" NUMBER
  - **$first_1(\alpha)$ = {"("}**
  - **$first_2(\alpha)$ = {"(" NUMBER, "(" IDENT, "(" "(" }**
- **Def. $first_k(\alpha)$:** Sei G = (V,Σ,S,R) CFG

$$first_k : (\Sigma \cup V)^* \to P(\Sigma^*)$$

$$- w \in \Sigma^* : first_k(w) := \begin{cases} \{w\}, \text{falls} \mid w \mid \leq k \\ \{v\}, \text{falls } w = vv', \mid v \mid = k \end{cases}$$

$$- \alpha \in (\Sigma \cup V)^* : first_k(\alpha) := \{v \in \Sigma^* \mid \alpha \ =l>^* \ w \in \Sigma^*, \{v\} = first_k(w)\}$$

Bsp.: $first_k$-Mengen

- **Beispielgrammatik GSE**
- **Satzform:** α = <Exp> ";"

  =|>* NUMBER ";" | IDENT ";" |
  NUMBER "*" NUMBER ";" |
  NUMBER "*" IDENT ";" |
  IDENT "*" NUMBER ";" |
  IDENT "*" IDENT ";" | .....

- $first_0(\alpha)$ = $\{\varepsilon\}$
- $first_1(\alpha)$ = {NUMBER, IDENT}
- $first_2(\alpha)$ = {NUMBER ";", IDENT ";", NUMBER "*", IDENT "*"}
- $first_3(\alpha)$ = {NUMBER ";", IDENT ";", NUMBER "*" NUMBER , IDENT "*" NUMBER, NUMBER "*" IDENT , IDENT "*" IDENT}

LL(k)-Grammatik

- durch look-ahead von k Symbolen wird Ableitungsschritt eindeutig
- d.h.: Schnitt der $first_k$-Mengen zweier Alternativen muss leer sein.
- **Def.:** Seien G = (V,Σ,S,R) CFG, k>0,
  w, x, y $\in \Sigma^*$; A $\in$ V; $\alpha, \beta, \gamma \in (V \cup \Sigma)^*$

  G $\in$ LL(k), genau dann wenn für alle Linksableitungen der Form

  S =l>* wAα =l> wβα und

  S =l>* wAα =l> wγα mit $\beta \neq \gamma$

  gilt: $first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \varnothing$

- **Damit:** Wähle die Regel, bei der die ersten k Symbole auf Eingabe in $first_k$-Menge der nachfolgenden Satzform liegen

Bsp.: für nicht LL(k)

- Beispielgrammatik GSE & Wort= IDE "=" IDE "*" IDE ";"

  <Stmt> =I>* $\underbrace{\text{IDE} \quad \text{"="}}_{w} \quad \underbrace{\text{<Exp>}}_{A} \quad \underbrace{\text{"*"} \quad \text{<Exp>} \quad \text{";"}}_{\alpha}$

- **2 interessante Folgekonfigurationen:**

  – $\underbrace{\text{IDE} \quad \text{"="}}_{w} \quad \underbrace{\text{<Exp>} \quad \text{"*"} \quad \text{<Exp>}}_{\beta} \quad \underbrace{\text{"*"} \quad \text{<Exp>} \quad \text{";}}_{\alpha}$ (2)

  – $\underbrace{\text{IDE} \quad \text{"="}}_{w} \quad \underbrace{\text{IDE}}_{\gamma} \quad \underbrace{\text{"*"} \quad \text{<Exp>} \quad \text{";"}}_{\alpha}$ (4)

  $\beta \mathrel{!}= \gamma$

Bsp.: für nicht LL(k) 2

- **2 der interessanten Endkonfigurationen:**
  - $\underbrace{\text{IDE "="}}_{w}\ \underbrace{\text{IDE "*" IDE "*" IDE ";"}}_{x}$
  - $\underbrace{\text{IDE "="}}_{w}\ \underbrace{\text{IDE "*" IDE ";"}}_{y}$

- **einige $first_k(x)$ und $first_k(y)$-Mengen**
  - $first_1(x) = first_1(y) = \{ \text{IDE} \}$ -> GSE nicht LL(1)
  - $first_2(x) = first_2(y) = \{ \text{IDE "*"} \}$ -> GSE nicht LL(2)
  - $first_3(x) = first_3(y) = \{ \text{IDE "*" IDE} \}$ -> GSE nicht LL(3)
  - $first_4(x) = \{ \text{IDE "*" IDE "*"} \}$ $first_4(y) = \{ \text{IDE "*" IDE ";"} \}$

braucht mindestens look-ahead von 4

Look-Ahead-Eindeutigkeit

- **Wdh.:**

G $\in$ LL(k), genau dann wenn für alle Linksableitungen der Form

S =l>* wAα =l> w$\beta\alpha$ und

S =l>* wAα =l> w$\gamma\alpha$ mit $\beta \neq \gamma$

gilt: $first_k(\beta\alpha) \cap first_k(\gamma\alpha) = \emptyset$

- Kenntnis der la-Mengen $first_k(\beta\alpha)$ erlaubt Entscheidung des Linksableitungsschrittes: Wenn ersten k Symbole auf Eingabe in
  - $first_k(\beta\alpha)$, dann Regel A -> β anwenden
  - $first_k(\gamma\alpha)$, dann Regel A -> γ anwenden

- **deterministischer Top-Down-Analyseautomat möglich**

Problem & follow$_k$-Mengen

- betrachte la-Mengen first$_k(\beta\alpha)$
- können unendlich viele Rechtskontexte α auftreten
- Bestimmung der la-Mengen muss endlicher Prozess werden!
- hierzu: Bestimmung der la-Mengen nur aus Regeln A -> β
- Beachte: Insbesondere bei ε-Regeln wird look-ahead-String aus dem Rechtskontext abgeleitet
- Abhängigkeit vom Rechtskontext auf follow-Mengen zurückführen (ersten k Symbole, die rechts einer Satzform auftreten können)
- Def.: Seien G = (V,Σ,S,R) CFG und k>0

$$follow_k : (\Sigma \cup V)^* \rightarrow P(\Sigma^*)$$

$$follow_k(\beta) := \{w \in \Sigma^* \mid S \Rightarrow^* \alpha\beta\gamma, w \in first_k(\gamma)\}$$

Der Fall k=1

- Für Praxis genügt k=1 (k>1 zu aufwändig)
- Schreibe dann: fi = $first_1$ und fo = $follow_1$
- **Satz: (LL(1)-Charakterisierung)**

  G ∈ LL(1) genau dann, wenn für alle Regelpaare

  A -> β und A -> γ mit $\beta \neq \gamma$

  gilt: $fi(\beta fo(A)) \cap fi(\gamma fo(A)) = \varnothing$

- **Definition der look-ahead-Mengen für Regeln**

  Ist G ∈ LL(1) und A -> β eine Regel von G. Dann heißt

  la(A, β) = fi(β fo(A)) $\subseteq \Sigma \cup \{\varepsilon\}$

  die look-ahead-Menge der Regel A -> β .

- **Wähle Regel A-> β, falls look-ahead auf Eingabe in la(A, β)**

Berechnung der fi- und fo-Mengen

$fi(\alpha)$ mit $\alpha \in (V \cup \Sigma)^*$:

- $fi(\varepsilon) = \{\varepsilon\}$
- $fi(a) = \{a\}$ mit $a \in \Sigma$
- $fi(A) = \bigcup_{A->\beta} fi(\beta)$ mit $A \in V$
- $fi(\alpha\beta) = \begin{cases} fi(\alpha), & \text{falls } \varepsilon \notin fi(\alpha) \\ fi(\alpha) \setminus \{\varepsilon\} \cup fi(\beta), & \text{falls } \varepsilon \in fi(\alpha) \end{cases}$ sonst

$fo(Var)$ mit $Var \in V$, $\beta \neq \varepsilon$:

- $\varepsilon \in fo(S)$
- $A->\alpha B\beta$, $a \in fi(\beta) : a \in fo(B)$
- $A->\alpha B$, $x \in fo(A) : x \in fo(B)$ mit $x \in \Sigma \cup \{\varepsilon\}$
- $A->\alpha B\beta$, $\varepsilon \in fi(\beta)$, $x \in fo(A) : x \in fo(B)$

Problem Linksrekursion

- Grammatik **linksrekursiv**, wenn es eine Variable A gibt und eine Ableitung $A =l>^{+} A\alpha$
- **Praktikum 1: direkte Linksrekursion G_{AE}:**
  - E -> E + T
  - E -> T
  - T -> T * F
  - T -> F
  - F -> (E)
  - F -> id

	S
fi	a
fo	ε a

Reg. i	la(i)
1	{a}
2	{a}

- **linksrekursive Grammatiken sind nicht LL(1)**
- **Bsp.:** S -> Sa (1)
  S -> a (2)

$$la(S, Sa) \cap la(S, a) = \{a\} \neq \emptyset$$

Elimination direkter Linksrekursion

- geg.: linksrekursive Grammatik G
- Ordne Regeln für jede Variable A mit direkter Linksrekursion wie folgt an:
  - A -> $A\alpha_1$ | ... | $A\alpha_m$ alle $\alpha_i \neq \varepsilon$
  - A -> β_1 | ... | β_n

  Kein β_i beginnt mit A.
- Ersetze diese Regeln durch folgende neue Regeln:
  - A -> $\beta_1 A'$ | ... | $\beta_n A'$
  - A' -> $\alpha_1 A'$ | ... | $\alpha_m A'$ | ε
- Elimination indirekter Linksrekursion siehe ALSU

Elimination Linksrekursion & la-Mengen

- G'_{AE}=({E,E',T,T',F},{id,*,+,(,)},E,R) mit
  R={
  - E -> TE' (1)
  - E' -> +TE' | ε (2,3)
  - T -> FT' (4)
  - T' -> *FT' | ε (5,6)
  - F -> (E) | id (7,8)

  }

	E	E'	T	T'	F
fi	(id	+ ε	(id	* ε	(id
fo	ε)	ε)	ε) +	ε) +	ε) + *

Reg. i	la(i)
1	{(,id}
2	{+}
3	{ε,)}
4	{(,id}
5	{*}
6	{ε,),+}
7	{(}
8	{id}

$$la(2) \cap la(3) =$$
$$la(5) \cap la(6) =$$
$$la(7) \cap la(8) = \varnothing$$

Analysetabelle

- Formalisierung des Top-Down-Analyseautomaten als Funktion act (Analysetabelle)
- **Argumente:**
  - Kellerspitze
  - la-Symbol
- **Ausgabe:**
  - bei Ableitungsschritt: rechte Regelseite & Regelnummer
  - bei Vergleichsschritt: `POP`
  - Fehlerfall: `ERROR`
  - Erfolg: `ACCEPT`

act-Funktion

$$act : (V \cup \Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\}) \rightarrow \{\alpha \mid A -> \alpha \in R\} \times \{1,..., p\} \cup \{POP, ERROR, ACCEPT\}$$

- $act(A, x) = (\alpha, i)$, falls Regel i $= A -> \alpha$ und $x \in la(A, \alpha)$
- $act(a, a) = POP$
- $act(\varepsilon, \varepsilon) = ACCEPT$
- $act(X, x) = ERROR$, sonst

Analysetabelle für G'_{AE}

	id	(	)	+	*	ε
E	TE',1	TE',1				
E'			ε,3	+TE',2		ε,3
T	FT ',4	FT ',4				
T '			ε,6	ε,6	*FT ',5	ε,6
F	id,8	(E),7				
id	POP					
(		POP				
)			POP			
+				POP		
*					POP	
ε						ACCEPT

kein Eintrag: ERROR, frühere Fehlererkennung

Top-Down Analyse DPDA (deterministisch)

- geg.: CFG G = (V,Σ,S,R) mit |R|=p
- **Def.: Top-Down Analyse DPDA zu G**
  - Zustand, Eingabealphabet, Kelleralphabet, Ausgabealphabet, Anfangskonfiguration: **analog zu nichtdeterministischem Fall**
  - Einzelschrittrelation:
    - Ableitungsschritte:
      - Falls act(A,a) = (α,i)
        (z,aw,Aγ,o_1...o_m) => (z,aw,αγ,o_1...o_mi)
      - Falls act(A,ε) = (α,i)
        (z, ε,Aγ,o_1...o_m) => (z, ε,αγ,o_1...o_mi)
    - Vergleichsschritt: Sei a ∈ Σ
      - (z,aw,aγ,o_1...o_m) => (z,w,γ,o_1...o_m)

Beispielanalyse (deterministisch)

- Satzformfolge des det. Top Down-Analyseautomaten für G'_{AE}
  - (z, "(" id "*" id ")", E,)
  - (z, "(" id "*" id ")", TE', 1)
  - (z, "(" id "*" id ")", FT'E', 14)
  - (z, "(" id "*" id ")", (E)T'E', 147)
  - (z, id "*" id ")", E)T'E', 147)
  - (z, id "*" id ")", TE')T'E', 1471)
  - (z, id "*" id ")", FT'E')T'E', 14714)
  - (z, id "*" id ")", idT'E')T'E', 147148)
  - (z, "*" id ")", T'E')T'E', 147148)
  - (z, "*" id ")", *FT'E')T'E', 1471485)
  - (z, id ")", FT'E')T'E', 1471485)
  - ...

Bem. zu DPDA

- DPDA arbeitet deterministisch, weil Kellerspitze und la-Symbol Aktion eindeutig bestimmen
- 2. Form des Ableitungsschrittes nur bei leerer Eingabe möglich (statt leerer Eingabe auch eof möglich)
- Eingabesymbol nur bei Vergleichsschritten gelöscht (konsumiert)

  -> kein echter PDA

  könnte man mit neuen Zuständen simulieren

Transformation nach LL(1)

- **Beseitigung von Linksrekursion**
  - bereits gesehen
  - linksrekursive Grammatik nicht LL(1)
  - hier nur gezeigt: Beseitigung direkter Linksrekursion
  - folgt nicht: G nicht linksrekursiv, dann G in LL!!! **Übung**
- **Links-Faktorisieren**
  - Bsp.: <stmt> -> if <cond> then <stmt> else <stmt>
    <stmt> -> if <cond> then <stmt>
  - keine Entscheidung bei Lesen von if
  - Idee: Verschiebe Entscheidung bis "Alternativstelle" erreicht
  - Ersetze A -> $\alpha\beta$ & A -> $\alpha\gamma$
    durch A -> αA' & A' -> β | γ

Linksfak. für Bsp.?

  - Bsp.: <stmt> -> if <cond> then <stmt> S'
    S' -> else <stmt> | ε

TD-Analyse mit rekursiven Prozeduren

- Bisher: Top-Down-Analyse durch explizite Implementierung durch PDA
- Jetzt: Ausnutzen der impliziten Verwendung eines Laufzeitkellers zur Implementierung von Rekursion
- Vorteil: Leichtere Programmierung
- Analogie: Syntaxdiagramme als System sich rekursiv aufrufender endlicher Automaten
- Idee: Für jede Variable A
  -> eine parameterlose Prozedur A()
- **2 Verfahren:**
  - rekursiver Abstieg ohne la-Mengen (schlechtere Fehleranalyse)
  - rekursiver Abstieg mit la-Mengen
- **Bsp.: jeweils für Grammatik G'_{AE}**

Rekursiver Abstieg ohne la-Mengen

```
void E() {
   print("1");
   T();
   E'();
   print("SUCCESS");
}

void E'() {
   if sym == "+" {
      print("2");
      nextsym();
      T();
      E'();
   }
   else print("3");
}
// T(), T'() analog
```

Rekursiver Abstieg ohne la-Mengen

```
void F() {
   if sym == "(" {
      print("7");
      nextsym();
      E();
      if sym == ")"
         nextsym();
      else
         stop("ERROR");
   }
   if sym == "id"
   {
      print("8");
      nextsym();
   }
   else stop("ERROR");
}
```

Rekursiver Abstieg mit la-Mengen

```
void E() {
   if (sym == "(" || sym == "id")
   {
      print("1"); T(); E'(); print("SUCCESS");
   }
   else break("ERROR");
}
//analog für die anderen Prozeduren
```

- **Vorteil:** Fehler werden früher erkannt!
- bei beiden Methoden werden ε–Regeln übersprungen!
- ähnlich bei Verwendung von javacc, dort jedoch:
  - Definition in EBNF-ähnlicher Form
  - kann zusätzlichen Code (z.B. für Codeerzeugung) einfügen

Komplexität der LL(1)-Analyse

- **G ist LL(1) -> G nicht linksrekursiv**

  -> Analyse in Linearzeit mit Faktor |V|
  jede Variable kommt höchstens einmal vor

  -> Kellerlänge linear beschränkt mit Faktor
  |V| * max { | α | | A -> α in R }
  jede Variable kommt höchstens einmal vor

- **Theorie**

  Sprachen, die durch LL(k)-Grammatiken beschreibbar

  ⊂ Sprachen, die durch LL(k+1)-Grammatiken beschreibbar

  ⊂ deterministische Typ2-Sprachen

Bottom-Up Analyse PDA (nichtdeterminist.)

- geg.: CFG G = (V,Σ,S,R) mit |R|=p
- **Def.: Bottom-Up Analyse PDA zu G**
  - Ein Zustand: z
  - Eingabealphabet von G: Σ
  - Kelleralphabet: **V ∪ Σ ∪ {$}** **$: Kellerbodensymb.**
  - Ausgabealphabet: {1,...,p}
  - Einzelschrittrelation:
    - Reduktionsschritt: Sei A -> α die i-te Regel in R
      - $(z,w, \alpha^{Rev}\gamma, o_1...o_m) \Rightarrow (z,w,A\gamma,o_1...o_m i)$
    - Shiftschritt: Sei a ∈ Σ
      - $(z,aw, \gamma, o_1...o_m) \Rightarrow (z,w, a\gamma,o_1...o_m)$
    - Stopschritt:
      - $(z, , S\$, o_1...o_m) \Rightarrow (z, , , o_1...o_m)$
  - Anfangskonfiguration für $w \in \Sigma^*$: $(z,w,\$,)$

Beispielanalyse (nichtdet.)

- **geg.: - Beispielgrammatik GSE**
  - Tokenfolge: `IDENT "=" NUMBER "*" IDENT ";"`
- **Bottom-Up Analyserechnung:**

(z, IDENT"="NUMBER"*"IDENT";",$,)
=> (z, "="NUMBER"*"IDENT";", IDENT$,) Shift oder Reduce (4)?
=> (z, NUMBER"*"IDENT";", "=" IDENT$,)
=> (z, "*"IDENT";", NUMBER "=" IDENT$,) Regelseitenende?
=> (z, "*"IDENT";", <Exp> "=" IDENT$, 3)
=> (z, IDENT";", "*" <Exp> "=" IDENT$, 3)
=> (z, ";", IDENT "*" <Exp> "=" IDENT$, 3)
=> (z, ";", <Exp> "*" <Exp> "=" IDENT$, 34)
=> (z, ";", <Exp> "=" IDENT$, 342)
=> (z, , ";" <Exp> "=" IDENT$, 342)
=> (z, , <Stmt>$, 3421) => (z, , , 3421)

Bem.: Bottom-Up-Analyse

- **Satz:** $o_1 \ldots o_m$ **Rechtsableitung von w**
  gdw. $(z,w,\$,) =r>^* (z, , , o_m \ldots o_1)$
- **Shift-Reduce-Verfahren:**
  - Shifte solange von Eingabe auf Keller, bis dort rechte Regelseite gespiegelt drauf liegt
  - Reduziere dann rechte Regelseite auf Keller zu einer entsprechenden linken Regelseite
- **Nichtdeterminismen:**
  - Shift- oder Reduktionsschritt?
  - Reduktion: welcher Teil des Stacks gehört zur rechten Seite?
  - Reduktion: welche linke Regelseite wird ausgewählt?
  - Stopschritt bei (z, , S$, o1...om) oder weiter rechnen?
- **Auflösung der Nichtdeterminismen durch look-ahead auf Eingabe hier nicht!!!**

LR-Analyse

- **Bottom-Up Analyse ist LR-Analyse**
- **LR(k)-Grammatiken**
  - durch look-ahead auf Eingabe um k Symbole wird Analyse deterministisch
  - LR(k) steht für
    - lese von ***L***inks nach rechts auf Eingabe
    - erzeuge ***R***echtsableitung
    - look-ahead auf Eingabe um ***k*** Symbole
- **Theorie**

  Sprachen, die durch LR(0)-Grammatiken beschreibbar

  $\subset$ Sprachen, die durch LR(1)-Grammatiken beschreibbar

  $=$ deterministische Typ2-Sprachen

Bottom-Up-Analyse in yacc

- **yacc:**
  - C-Pendant zu javacc
  - steht für yet another compiler compiler
  - implementiert Bottom-up-Parsing
  - verwendet nicht LR(1)-Grammatiken, weil zu aufwändig
  - aber LR(0)-Grammatiken erzeugen zu kleine Klasse
  - darum: Kompromiss **LALR(1)-Grammatiken**
- **Nachweis der LALR(1)-Eigenschaft ziemlich kompliziert!**