Theoretische Informatik

Kontextfreie Sprachen und Parser

Inhalt

1. Grammatiken und Sprachen
   - Kontextfreie Grammatiken
   - Herleitungen, Linksherleitungen
   - Sprachen zu einer Grammatik
   - Äquivalenz
   - Chomsky-Normalform
   - Wortproblem, CYK
   - Pumping Lemma für CF-Sprachen

2. Stackautomaten
   - Definitionen und Beispiele
   - Konfigurationen, Läufe,
   - Sprache eines Stackautomaten
   - Parser

3. Parser
   - Mehrdeutigkeit
   - Bottom Up, Top Down
   - Recursive descent Parser
   - First, Follow
   - Semantische Aktionen

4. Shift-Reduce Parser
   - Konflikte
   - LR(0)-Zustände
   - Automat zur Grammatik
   - Shift-Reduce, Reduce-Reduce Konflikte
   - Präzedenz und Assoziativität
   - lex und yacc
   - Arbeitsweise eines Compilers

Der zweistufige Aufbau von Sprachen

Aus Alphabet der Token wird der syntaktische Anteil der Sprache aufgebaut,

Σ = Menge der Token

L = Menge aller syntaktisch korrekter Sätze.

Auf dieser Stufe sind Sprachen i.A. nicht regulär. (Geschachtelte Strukturen kann man nicht regulär beschreiben - Klammersprache).

Für die Definition muss was neues her:

Grammatiken

Syntax - Semantik

- Grammatik beschreibt die Syntax (Struktur) von Sätzen, nicht deren Semantik (Bedeutung).
- Semantik hängt von Syntax ab, aber nicht jeder syntaktisch korrekte Satz muss eine Semantik haben.

Syntaktisch falscher Satz:

gestohlen der hat Gans Fuchs die.

Syntaktisch richtiger, semantisch falscher Satz:

der Gans hat die Fuchs gestohlen.

Syntaktisch und semantisch richtiger Satz:

der Fuchs hat die Gans gestohlen.

Syntax - Semantik in PASCAL

Syntaktisch falsches Programm:

```
Test Program.
  i for 1 do n ; to n+1 := n
```

Syntaktisch richtiges, semantisch falsches Programm:

```
Program Test ;
Var i : Real;
Begin
    For i := 1 To n Do i := n-1
End.
```

Syntaktisch und semantisch korrektes Programm:

```
Program Test ;
Var i,n : Integer;
Begin
    For i := 1 To 50 Do n := n+i
End.
```

Sprachen und Grammatiken

Eine Grammatik ist eine Menge von Regeln mit deren Hilfe „syntaktisch korrekte“ Worte (Sätze) der Sprache konstruiert werden können.

Erzeugen

Mit Hilfe der Grammatik kann man Sätze einer Sprache erzeugen. Die Menge dieser Sätze bilden die zur Grammatik G gehörende Sprache, L(G).

Zerlegen

Ist ein Satz gegeben, so möchte man feststellen , ob er zur Sprache L(G) gehört oder nicht. Dabei wird auch die Struktur des Satzes erkannt.

Beispiel einer Grammatik

Java

```
Stmt        :: while ( Expr ) Stmt | if (Expr) Stmt else Stmt
             | id = Expr ; | { Stmts } ...

Expr        :: Expr op Expr | ( Expr ) | num | id ...

Stmts       :: Stmt Stmts | ε
    etc.
```

Bestandteile:

Token	: while, (,), if, else, id, =, ;, {, }, op, num, id
Variablen	: Stmt, Expr, Stmts,
Satzformen	: (Expr) , id = Expr ; , ε , ...

Die Zeichen

:: (definiert als)

| (Alternative)

sind Sonderzeichen für Grammatikdefinitionen

Kontextfreie Grammatik

Gegeben:

T	- ein Alphabet (Terminale, Token)
V	- eine Menge von Variablen (Nonterminale)
$S \in V$	- Startsymbol

mit $T \cap V = \emptyset$

Eine Kontextfreie Grammatik ist ein Quadrupel $G = (V, T, P, S)$
mit $T \cap V = \emptyset$,

$S \in V$ und $P \subseteq V \times (T \cup V)^*$

Begriffe

Satzform : Element α von $(V \cup T)^*$

Produktion : Jedes $(A,\alpha) \in P$. Notation: $A \rightarrow \alpha$

Startsymbol : S.

CF-Grammatik - Beispiele

Variable

Satzform

Binärzahlen

G = (V, T, P, S) , wobei

T = { 0, 1 }
V = { B }
S = B
P = { B → 0, B → 1, B → 0B, B → 1B }

Produktion
bzw.
Regel

CF-Grammatik - Beispiel

Ausdrücke

G = (V, T, P, S) , wobei

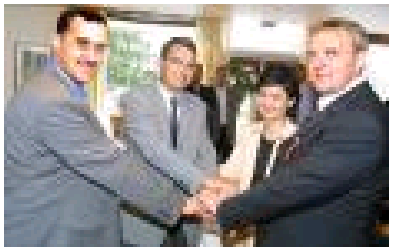
T = { id, num, +, *,), (}

V = { Term, Factor, Expr }

S = Expr

P = { Expr → Term , Expr → Expr + Term,
Term → Factor , Term → Term * Factor,
Factor → id, Factor → num,
Factor → (Expr)
}

Konventionen



- Variablen : Alle Symbole für die es mind. eine Produktion gibt
- Terminale : alle anderen Symbole
- Startsymbol :
  - Default: Variable der ersten Produktion.
- Produktionen mit gleicher linker Seite zusammenfassen.
  - Rechte Seiten durch | trennen.

Das vorige Beispiele kann man dann schreiben als :

G

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} + \text{Term} \\
&\mid \text{Term} \\
\text{Term} &\rightarrow \text{Term} * \text{Factor} \\
&\mid \text{Factor} \\
\text{Factor} &\rightarrow (\text{Expr}) \\
&\mid \text{id} \mid \text{num}
\end{aligned}
$$

Ersetzung - Herleitung

Eine Produktion $A \to \beta$ erlaubt, in einer Satzform $\alpha A \gamma$ die Variable A durch β ersetzen. Resultat ist die Satzform $\alpha\beta\gamma$.

Eine Produktion:

$$A \to \beta$$

Eine Ersetzung: $\alpha\, A\, \gamma \quad \Rightarrow \quad \alpha\, \beta\, \gamma$

$\sigma \Rightarrow^* \tau$ falls es $\alpha_1, \ldots \alpha_n$ gibt mit

$$\sigma = \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n = \tau$$

Herleitung

Beispiel einer Herleitung

Arithmetik

Expr → Expr + Term
| Term

Term → Term * Factor
| Factor

Factor → (Expr)
| id | num

Es gilt also:

Expr ⇒* id * (num+id)

Eine Herleitung:

Expr ⇒ Term
⇒ Term * Factor
⇒ Term * (Expr)
⇒ Term * (Expr + Term)
⇒ Term * (Term + Term)
⇒ Factor * (Term + Term)
⇒ id * (Term + Term)
⇒ id * (Term + Factor)
⇒ id * (Term + id)
⇒ id * (Factor + id)
⇒ id * (num + id).

Eine Links-Herleitung

Arithmetik

$$\begin{aligned} Expr &\rightarrow Expr + Term \\ &\mid Term \\ Term &\rightarrow Term * Factor \\ &\mid Factor \\ Factor &\rightarrow (Expr) \\ &\mid id \mid num \end{aligned}$$

Jedes Wort $w \in L(G)$ kann man auch durch eine Linksherleitung gewinnen

Bei einer Links-Herleitung wird immer das linkeste Nonterminal ersetzt

Eine Links-Herleitung:

$$\begin{aligned} Expr &\Rightarrow Term \\ &\Rightarrow Term * Factor \\ &\Rightarrow Factor * Factor \\ &\Rightarrow id * Factor \\ &\Rightarrow id * (Expr) \\ &\Rightarrow id * (Expr + Term) \\ &\Rightarrow id * (Term + Term) \\ &\Rightarrow id * (Factor + Term) \\ &\Rightarrow id * (num + Term) \\ &\Rightarrow id * (num + Factor) \\ &\Rightarrow id * (num + id). \end{aligned}$$

Links-Herleitung eines anderen Wortes

Arithmetik

Expr → Expr + Term
| Term

Term → Term * Factor
| Factor

Factor → (Expr)
| id | num

Auswahl anderer Regeln erzeugt i.A. andere Worte $w \in L(G)$

Bei einer Links-Herleitung wird immer das linkeste Nonterminal ersetzt

Eine Links-Herleitung unter Verwendung anderer Regeln:

Expr ⇒ Expr + Term
⇒ Term + Term
⇒ Factor + Term
⇒ num + Term
⇒ num + Factor
⇒ num + (Expr)
⇒ num + (Term)
⇒ num + (Term * Factor)
⇒ num + (Factor * Factor)
⇒ num + (id * Factor)
⇒ num + (id * num).

Herleitung als Baum

Herleitungen in Baumnotation:

- Wurzel: Startsymbol
- Anwendung einer Regel A→ α
  - Knotenwachstum.
- Symbole aus α werden Kindknoten

Arithmetik

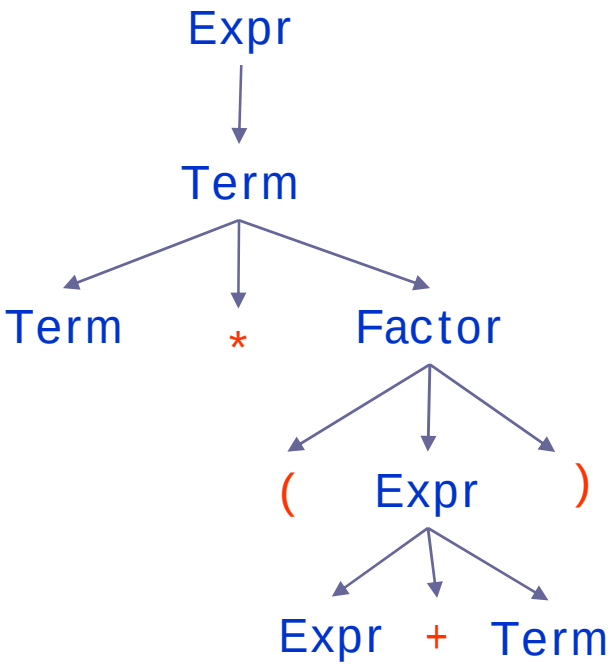
Expr → Expr + Term
| Term

Term → Term * Factor
| Factor

Factor → (Expr)
| id | num

Expr ⇒ Term
⇒ Term * Factor
⇒ Term * (Expr)
⇒ Term * (Expr + Term)

angefangene Herleitung

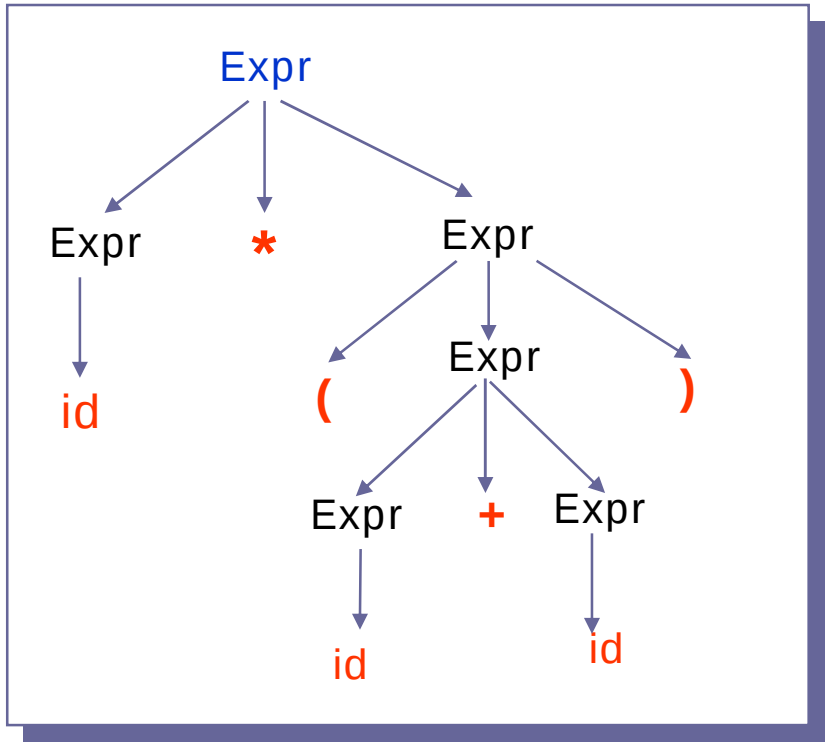


entsprechender Herleitungsbaum

Fertiger Herleitungsbaum

G

Expr → Expr + Expr
| Expr * Expr
| (Expr)
| id



Wurzel: Startsymbol

innere Knoten: Variablen

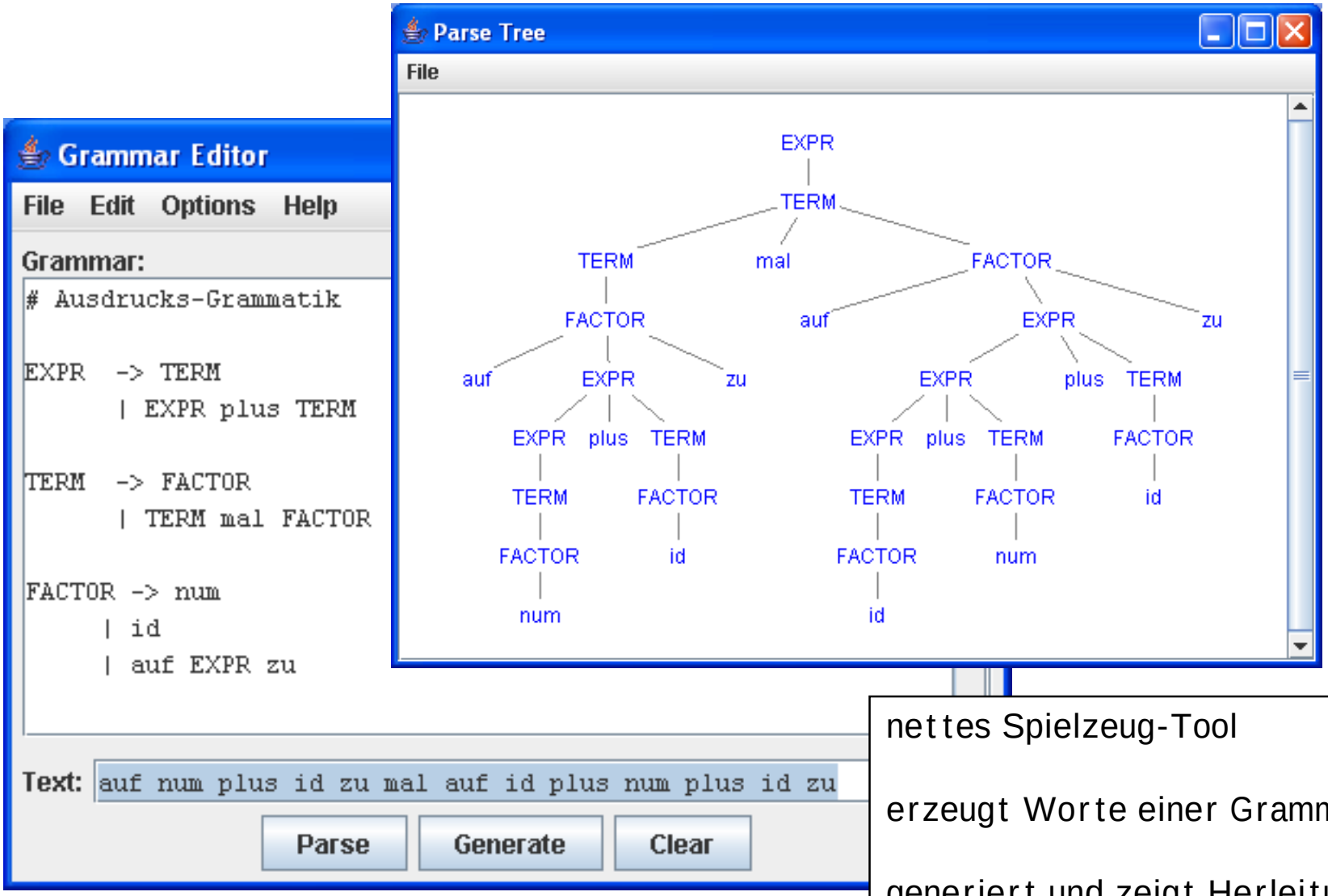
Söhne von A : Symbole von α, für $A \rightarrow \alpha \in P$

Blätter: Terminalsymbole

repräsentiert Herleitung

Expr $\Rightarrow^*$ id * (id + id)

Blätter des Baumes ergeben von links nach rechts das hergeleitete Wort.



Grammar-Editor (C. Burch)

nettes Spielzeug-Tool

erzeugt Worte einer Grammatik

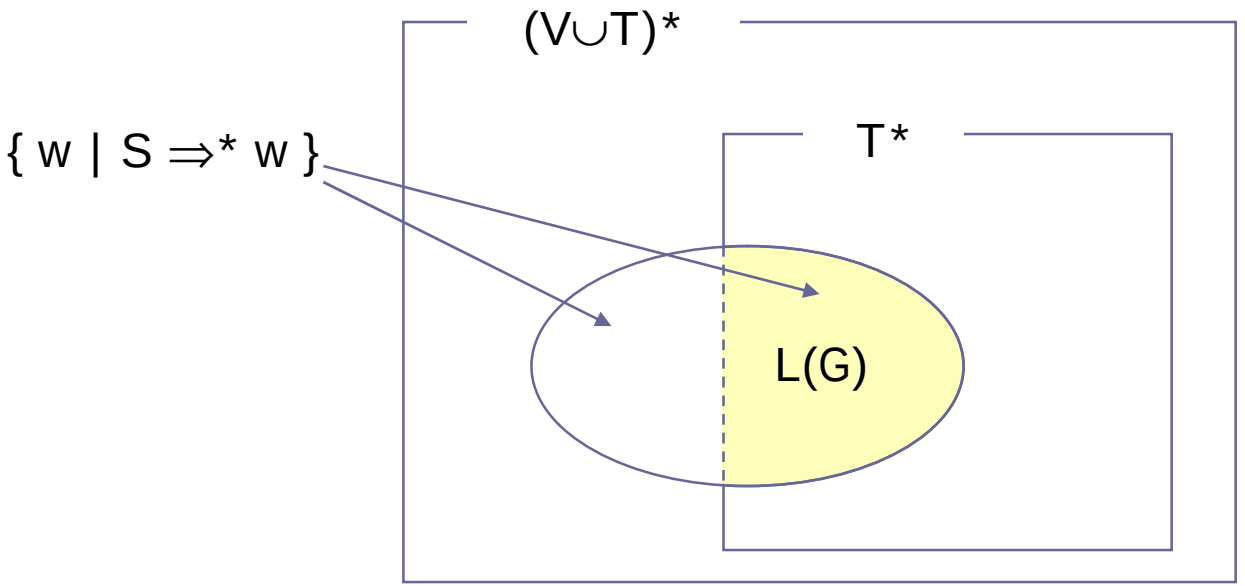
generiert und zeigt Herleitungsbaum

Die Sprache zu einer Grammatik

Ist G = (V, T, P, S) eine Grammatik, so ist

$$L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$$

die von G erzeugte Sprache.



Kontextfreie Sprachen

- Eine Sprache $L \subseteq T^*$ heißt kontextfrei, falls es eine kontextfreie Grammatik G gibt, mit $L = L(G)$.

Beispiel:

$L_{anbn} = \{ a^n b^n \mid n \geq 0 \}$ ist kontextfrei:

G

$S \rightarrow a\ S\ b$
$\mid \varepsilon$

Muss man das beweisen ? Kaum, denn es ist ziemlich offensichtlich.
Aber wir können daran üben, wie man einen Beweis macht. Also:

1. $L_{anbn} \subseteq L(G)$:

   Induktionsanfang n=0: klar, denn $a^0 b^0 = \varepsilon$ und $S \Rightarrow^* \varepsilon \in T^*$
   Ind. schritt:
   Sei $a^k b^k \in L(G)$ schon gezeigt, dann gibt es
   eine Herleitung $S \Rightarrow a^k b^k$.
   … und jetzt …. ??? Tja – doch nicht so einfach ??

Kontextfreie Sprachen

- Eine Sprache $L \subseteq T^*$ heißt kontextfrei, falls es eine kontextfreie Grammatik G gibt, mit $L = L(G)$.

Beispiel:

$L_{anbn} = \{ a^n b^n \mid n \geq 0 \}$ ist kontextfrei:

G

$S \rightarrow a\ S\ b$
$\mid \varepsilon$

$L_{anbn} = L(G)$.

Die Behauptung folgt aus der stärkeren Aussage:

Für jede Satzform $\alpha \in \{S,a,b\}^*$: $S \Rightarrow^* \alpha$ gdw. $\alpha = a^n S b^n \vee \alpha = a^n b^n$

Beweis: Sei $F = \{a^n S b^n \mid n \geq 0 \} \cup \{a^n b^n \mid n \geq 0 \}$. Dann gilt

$S \in F$ und

$\sigma \in F$, $\sigma \Rightarrow \tau$ impliziert $\tau \in F$

Daher: $S \Rightarrow^* \alpha$ impliziert $\alpha \in F$.

Umgekehrt: Für jedes n gilt $S \Rightarrow^* a^n S b^n$ (Induktion). Folglich auch $S \Rightarrow^* a^n S b^n \Rightarrow a^n b^n$.

Kontextfreie Sprachen

- Eine Sprache $L \subseteq T^*$ heißt *kontextfrei*, falls es eine kontextfreie Grammatik G gibt, mit L = L(G).

Die Klammersprache ist kontextfrei:

K

$$S \rightarrow (S) \mid S\ S \mid \varepsilon$$

Beispiel einer Herleitung:

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow (S\ S) \\ &\Rightarrow ((S)\ S) \\ &\Rightarrow (((S))\ S) \\ &\Rightarrow ((())(S)) \\ &\Rightarrow ((())()) \end{aligned}$$

Hier ist nichts zu beweisen, weil wir die Klammersprache nie definiert haben.
Im Gegenteil, wir nehmen die obige Grammatik als Definition für die Klammersprache.

Jede reguläre Sprache ist kontextfrei

Sei Σ gegeben. Die Basissprachen :

$G_\varnothing$

G_ε

$S \to \varepsilon$

G_a

$S \to a$

Seien L, L_1, L_2 durch Grammatiken G, G_1, G_2 mit Startsymbolen S_L, S_1, bzw. S_2 gegeben:

G_{L1+L2}

$S \to S_1$
$S \to S_2$

+ alle Prod. von G_1

+ alle Prod. von G_2

$G_{L1\ L2}$

$S \to S_1 S_2$

+ alle Prod. von G_1

+ alle Prod. von G_2

G_{L*}

$S \to S_L\ S$
$S \to \varepsilon$

+ alle Prod. von G

Sprachen zu einer Grammatik

Welche Sprache gehört zu folgender Grammatik ?

G

$S \rightarrow aB \mid bA \mid \varepsilon$
$A \rightarrow aS \mid bAA$
$B \rightarrow bS \mid aBB$

Vermutung: $L(G) = \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) \}$,
wobei

$\#(a,w)$ = Anzahl der a's in w.

Seien L_A, L_B, L_S die Sprachen die entstehen, wenn man in G das entsprechende Nonterminal als Startsymbol nimmt. Wir zeigen die stärkere Behauptung:

$L_S = \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) \}$
$L_A = \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) + 1 \}$
$L_B = \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) - 1 \}$

Simultaner Beweis-Teil 1

G

$S \rightarrow aB \mid bA \mid \varepsilon$
$A \rightarrow aS \mid bAA$
$B \rightarrow bS \mid aBB$

$L_S \subseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) \}$
$L_A \subseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) + 1 \}$
$L_B \subseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) - 1 \}$

Behauptung :

$S \Rightarrow^* w$ impliziert $\#(a,w) + \#(A,w) = \#(b,w) + \#(B,w)$.

Induktion über Länge der Herleitung $S \Rightarrow^* w$:

Gilt für w=S (Herleitung der Länge 0).
Jede Regelanwendung erhält die Eigenschaft (einzeln nachprüfen)
exemplarisch: $A \rightarrow bAA$

$\alpha A \beta \Rightarrow \alpha\, bAA \beta$ // links verschwindet ein A
// rechts kommen 2 A's und ein b hinzu.

Für $w \in L_S$ folgt also:

$S \Rightarrow^* w \in \{a,b\}^*$, somit $\#(a,w)=\#(b,w)$,

also $L_S \subseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) \}$. q.e.d.

Analog zeigt man die Behauptungen für L_A und L_B.

Simultaner Beweis-Teil 2

G

$S \to aB \mid bA \mid \varepsilon$
$A \to aS \mid bAA$
$B \to bS \mid aBB$

$L_S \supseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) \}$ $=: L_0$
$L_A \supseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) + 1 \}$ $=: L_{+a}$
$L_B \supseteq \{ w \in \{a,b\}^* \mid \#(a,w) = \#(b,w) - 1 \}$ $=: L_{+b}$

Behauptung:

w aus einer der rechten Seiten, dann auch aus der entsprechenden linken.

Induktion über die Länge von w:

$w = \varepsilon$: Wenn w aus einer der rechten Seiten ist, dann muss es aus der ersten sein.

$w \neq \varepsilon$: Beispiel: $w \in L_{+a}$: ($w \in L_0$, $w \in L_{+b}$ genauso)

1. Fall: w=a.v:

$v \in L_0$	// Def. L_{+a}, L_0
also $v \in L_S$	// Ind. hypothese (v kürzer als w)
also $S \Rightarrow^* v$	// Def L_B
$A \Rightarrow aS \Rightarrow^* a.v$	// Erste Regel für A
$A \Rightarrow^* w$	// $\Rightarrow$ transitiv, w=a.v
$w \in L_A$	// Def. L_A

2. Fall: w=b.v:

$\#(a,v) = \#(b,v)+2$	// klar
$v = st$ mit $s,t \in L_{+a}$	// v kann so zerlegt werden
also $s,t \in L_A$	// Ind. hypothese (s,t kürzer als w)
also $A \Rightarrow^* s$, $A \Rightarrow^* t$	// Def L_A
$A \Rightarrow bAA \Rightarrow^* bsA \Rightarrow^* bst$	// Zweite Regel für A
$A \Rightarrow^* w$	// $\Rightarrow$ transitiv, w=a.st
$w \in L_A$	// Def. L_A

Äquivalenz

Grammatiken G_1 und G_2 heißen äquivalent, falls $L(G_1) = L(G_2)$.

G_1

Sum → Sum + Sum
| id

G_2

Sum → Sum + id
| id

G_1 und G_2 sind äquivalent:

$$L(G_1) = \text{id} \; (\text{‘+’ id})^* = L(G_2)$$

Chomsky-Normalform

http://www.chomsky.info/

Zu jeder Grammatik G gibt es eine äquivalente Grammatik G‘, deren Regeln folgende einfache Bauart haben:

$A \to a$
$A \to BC$
$S \to \varepsilon$

a Terminal
A,B,C Nichtterminale.
S Startsymbol

Die Produktion $S \to \varepsilon$ ist genau dann vorhanden, wenn $\varepsilon \in L(G)$ ist.
In diesem Falle kommt S in keiner Regel rechts vor.

Der Satz von Chomsky ist ein theoretisches Ergebnis. In der Praxis ist es angenehmer, mit Regeln beliebiger Bauart spezifizieren zu dürfen.

Erreichbare Variablen

$A \in V$ heißt erreichbar, falls $S \Rightarrow^* \alpha A \beta$

Rekursive Charakterisierung

A erreichbar $\Leftrightarrow$ A = S
oder $(E \rightarrow \alpha A \beta) \in P$ und E erreichbar.

Fast schon ein rekursives Programm. Nur für Terminierung sorgen.
Iterative Implementierung *markiert* alle erreichbaren Variablen:

MarkierungsAlgorithmus

Markiere S.
DO
  FALLS $(E \rightarrow \alpha A \beta) \in P$ und E schon markiert, markiere A
BIS in einer Runde nichts neues markiert wurde

Nicht erreichbare Variablen kann man entfernen.

Produktive Variablen

$A \in V$ heißt **produktiv**, falls $A \Rightarrow^* w \in T^*$

Rekursive Charakterisierung

A produktiv $\Leftrightarrow$ $(A \to w) \in P$ für ein $w \in T^*$
oder $(A \to \alpha) \in P$ für ein $\alpha \in (V\text{-}\{A\} \cup T)^*$
und alle Variablen in α produktiv.

MarkierungsAlgorithmus

Markiere alle $A \in V$ mit $(A \to w) \in P$ und $w \in T^*$.

DO

FALLS $(A \to \alpha) \in P$ und alle Nonterminale in α schon markiert,
markiere A

BIS in einer Runde nichts neues markiert wurde

Nicht produktive Variablen kann man entfernen.

Kollabierende Variablen

$A \in V$ heißt kollabierend, falls $A \Rightarrow^* \varepsilon$

Rekursive Charakterisierung

A kollabierend $\Leftrightarrow$ $(A \to \varepsilon) \in P$
oder $(A \to B_1 B_2 \ldots B_n) \in P$, $B_i \neq A$,
und alle B_i kollabierend).

MarkierungsAlgorithmus

Markiere alle $A \in V$ mit $(A \to \varepsilon) \in P$.
DO
FALLS $(A \to B_1 \ldots B_n) \in P$ und alle B_1 schon markiert, markiere A
BIS in einer Runde nichts neues markiert wurde

Entfernung von ε-Regeln

- Jede Regel der Form $A \to \varepsilon$ heißt ε-Regel

Für jede kollabierende Variable C
  DO FÜR jede Regel der Form
    $A \to \alpha C \beta$
  füge eine neue Regel hinzu:
    $A \to \alpha\beta$
  BIS in einer Runde keine neue Regel hinzukam
Streiche alle ε-Regeln

Vorsicht: Falls $A \to \alpha C \beta C \gamma$ und C kollabierend, müssen sowohl $A \to \alpha\beta C\gamma$ als auch $A \to \alpha C\beta\gamma$ und $A \to \alpha\beta\gamma$ hinzukommen.

Aus Grammatik G entsteht eine neue Grammatik G^+ ohne ε-Regeln und es gilt

$$L(G^+) = L(G) - \{\varepsilon\}.$$

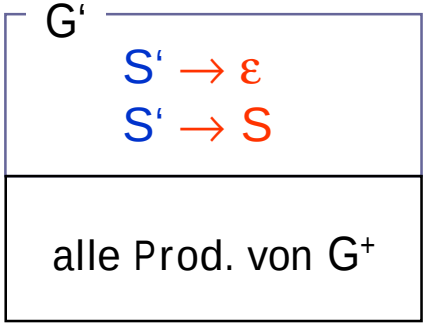
ε-Freiheit

G heißt ε-frei, falls
- keine Regel der Form $A \to \varepsilon$ vorkommt,
- oder ($S \to \varepsilon$ ist einzige ε-Regel,
  und S kommt in keiner Produktion rechts vor).

$L(G^+) = L(G) - \{\varepsilon\}$ und G^+ ist ε-frei. Daher:

1. Fall: $\varepsilon \notin L(G)$. Setze $G' = G^+$.

2. Fall: $\varepsilon \in L(G)$:



Zu jeder Grammatik G gibt es eine äquivalente ε-freie Grammatik.

Variablen-Regeln

Alle Variablen-Regeln, das sind Regeln der Form $A \to B$ mit $A,B \in V$ kann man eliminieren

Algorithmus

DO
  FÜR jede Variablenregel $A \to B$:
    FÜR jede Regel $B \to \alpha$
      füge neue Regel $A \to \alpha$ hinzu
  Streiche $A \to B$
BIS nichts mehr passiert.

Chomsky - Normalform

- Satz: Jede kontextfreie Grammatik G lässt sich äquivalent umformen in eine Grammatik G', deren Regel von der Form sind:

  $A \to a$

  $A \to BC$

  Falls $\varepsilon \in L(G)$ gibt es noch die Produktion $S \to \varepsilon$ und S kommt in keiner Produktion rechts vor.

1. Entferne nicht terminierende und nicht erreichbare Variablen

2. Mache G ε-frei und entferne alle Variablen-Regeln.

3. In jeder Regel $A \to \alpha$ mit $|\alpha| > 1$ ersetze Terminalsymbol a durch neue Variable X_a und füge die Regel $X_a \to a$ hinzu.

4. Ersetze jede Regel der Form

   $A \to B_1B_2 \ldots B_n$ mit Hilfe neuer Variablen $C_1 \ldots C_{n-1}$ durch

   $A \to B_1C_1$, $C_1 \to B_2C_2$ ... $C_{n-1} \to B_{n-1}B_n$

Beispiel: Gewinnung einer CNF – Anfang

- Wir wandeln die Klammersprache K in CNF um
- Jede Variable ist erreichbar und terminierend
  - Das ist normal für realistische Grammatiken !
- S ist kollabierend. Wir erhalten nacheinander
  - K^+ durch Entfernen der ε-Regeln
  - Kε als zu K äquivalente ε-freie Grammatik
  - KεV durch Elimination von Variablenregeln

K

$$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$$

K^+

$$S \rightarrow S\ S \mid S \mid (\ S\ ) \mid (\ ) \mid \varepsilon$$

Kε

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid S \mid (\ S\ ) \mid (\ )$$

KεV

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid (\ S\ ) \mid (\ )$$

$L(K^+) \neq L(K)$

$L(K\varepsilon) = L(K)$

$L(K\varepsilon V) = L(K)$

Beispiel: Gewinnung einer CNF – Schluss

- Wir wandeln die Klammersprache K in CNF um
  - Wir haben bereits KεV zu K äquivalent
    - alle Variablen terminierend und ε-frei
    - ohne Variablenregeln
  - Wir ersetzen nun Terminale durch Variablen
  - und verkürzen rechte Seiten
  - K_{CNF} ist äquivalent zu K und in Chomsky-Normalform

K

$$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$$

KεV

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid (\ S\ ) \mid (\ )$$

KεVT

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid A\ S\ Z \mid A\ Z$$

$$A \rightarrow ($$

$$Z \rightarrow)$$

K_{CNF1}

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid A\ C \mid A\ Z$$

$$C \rightarrow S\ Z$$

$$A \rightarrow ($$

$$Z \rightarrow)$$

Die Normalform

- Eine Variablenregel ist noch zu entfernen:

K

$$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$$

Ausgangs grammatik

K_{CNF1}

$$S' \rightarrow \varepsilon \mid S$$

$$S \rightarrow S\ S \mid A\ C \mid A\ Z$$

$$C \rightarrow S\ Z$$

$$A \rightarrow ($$

$$Z \rightarrow)$$

K_{CNF}

$$S' \rightarrow \varepsilon \mid S\ S \mid A\ C \mid A\ Z$$

$$S \rightarrow S\ S \mid A\ C \mid A\ Z$$

$$C \rightarrow S\ Z$$

$$A \rightarrow ($$

$$Z \rightarrow)$$

Wortproblem für kontextfreier Sprachen

Sei G eine Grammatik, $A \in V$ und $w = a_1 a_2 \ldots a_n \in T^*$.
Gilt

$$A \Rightarrow^* w \quad ?$$

Wir haben einen Algorithmus, um G in Chomsky-Normalform G_{CNF} zu überführen. Dann wenden wir den folgenden Algorithmus von Cocke, Younger und Kasami an:

CYK-Algorithmus - rekursiv

$A \Rightarrow^* w \quad \Leftrightarrow \quad w = \varepsilon$ und $S \to \varepsilon \in G_{CNF}$
oder $n=1$ und $S \to w \in G_{CNF}$
oder $n > 1$ und es gibt ein $1 \le k < n$ mit
$A \to BC \in G_{CNF}$ und $B \Rightarrow^* a_1 \ldots a_k$
und $C \Rightarrow^* a_{k+1} \ldots a_n$

Jede kontextfreie Sprache ist entscheidbar.

CYK-Algorithmus - imperativ

Imperativ implementiert man den CYK durch dynamisches Programmieren:

Gegeben $w = a_1 \dots a_n \in T^*$. Für jedes $1 \le i, k \le n$ sei

$$M_{ik} = \{ A \in V \mid A \Rightarrow^* a_i \dots a_{i+k-1} \}$$

Dann gilt:

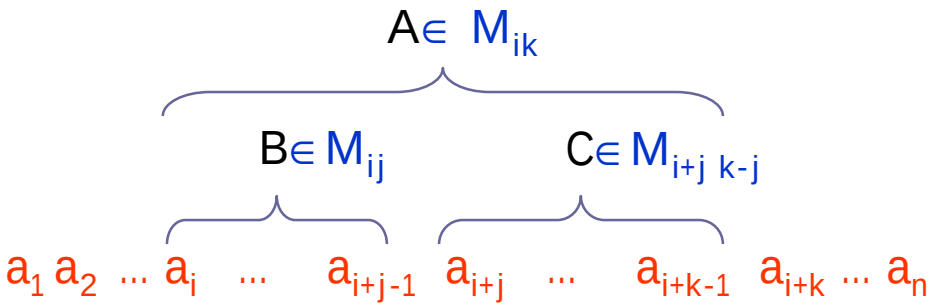
$$w \in L(G) \Leftrightarrow S \in M_{1n}.$$

Ist G in Chomsky-Normalform, so kann man die M_{ik} in eine Tabelle eintragen.

$$M_{i1} = \{ A \in V \mid A \to a_i \in G \}$$

$$M_{ik} = \{ A \in V \mid \exists_{1 \le j < k} : \exists A \to BC \in G : B \in M_{ij}, C \in M_{i+j\ k-j} \}$$

5	M_{15}				
4	M_{14}	M_{24}			
3	M_{13}	M_{23}	M_{33}		
2	M_{12}	M_{22}	M_{32}	M_{42}	
1	M_{11}	M_{21}	M_{31}	M_{41}	M_{51}
M_{ik}	1	2	3	4	5



CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8								
7								
6								
5								
4								
3								
2								
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8								
7								
6								
5								
4								
3								
2	-	-	S	-	-	S	-	
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8								
7								
6								
5								
4								
3	-	-	C	-	-	C		
2	-	-	S	-	-	S	-	
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8							
7							
6							
5							
4	-	S	-	-	-		
3	-	-	C	-	-	C	
2	-	-	S	-	-	S	-
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8								
7								
6								
5	-	-	-	-				
4	-	S	-	-	-			
3	-	-	C	-	-	C		
2	-	-	S	-	-	S	-	
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8								
7								
6	-	S	-					
5	-	-	-	-				
4	-	S	-	-	-			
3	-	-	C	-	-	C		
2	-	-	S	-	-	S	-	
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8							
7	-	C					
6	-	S	-				
5	-	-	-	-			
4	-	S	-	-	-		
3	-	-	C	-	-	C	
2	-	-	S	-	-	S	-
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

K^+_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

CYK- Algorithmus – ein Beispiel

- Gegeben sei die Sprache der nichtleeren Klammerausdrücke
- Gefragt: Ist (((())())) ∈ L(G) ?

8	S							
7	-	C						
6	-	S	-					
5	-	-	-	-				
4	-	S	-	-	-			
3	-	-	C	-	-	C		
2	-	-	S	-	-	S	-	
1	A	A	A	Z	Z	A	Z	Z
M_{ij}	(	(	(	)	)	(	)	)

$S \in M_{18} \Leftrightarrow w \in L(K^{+}_{CNF})$

K^{+}_{CNF}

$S \rightarrow S\ S \mid A\ C \mid A\ Z$

$C \rightarrow S\ Z$

$A \rightarrow ($

$Z \rightarrow)$

Grenzen von CF-Sprachen

- Gibt es Sprachen, die nicht kontextfrei sind ?
  - ja, z.B. $\{a^n b^n c^n \mid n \geq 0\}$
- Wie kann man das beweisen ?
  - Pumping Lemma für CF Sprachen – kommt gleich
- Sind CF-Sprachen unter $\cap$ abgeschlossen
  - Nein, denn

    $\{a^n b^n c^n \mid n \geq 0\} = \{a^n b^n c^k \mid n \geq 0, k \geq 0\} \cap \{a^k b^n c^n \mid n \geq 0, k \geq 0\}$

    und letztere beiden sind offensichtlich CF.
- Unter Komplement ?
  - Nein, denn sie sind unter $\cup$ abgeschlossen, und mit Komplement und Vereinigung kann man den Schnitt ausdrücken:

$$L \cap M = \Sigma^* - ((\Sigma^* - L) \cup (\Sigma^* - M))$$

Pumping Lemma für CF-Sprachen

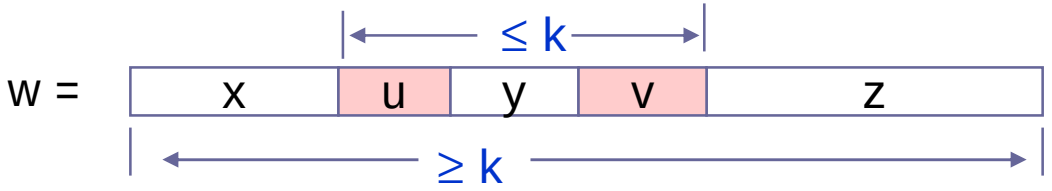
Für jede CF-Sprache L gibt es eine Zahl k, so daß
jedes Wort $w \in L$ mit $|w| \geq k$ sich zerlegen läßt als

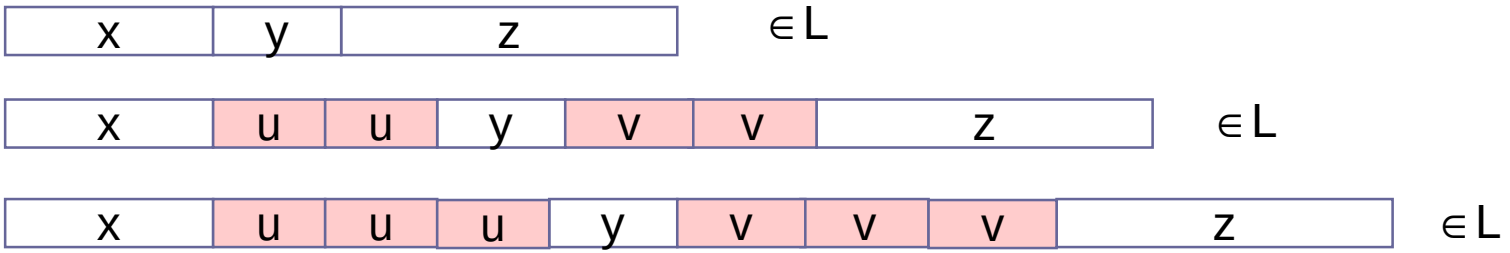
$$w = x\,u\,y\,v\,z$$

so dass

- $0 < |uv| \leq |uyv| \leq k$
- $\forall n \in Nat : xu^n yv^n z \in L$.

Also jedes große ($\geq k$) Wort hat ein kleines ($\leq k$) Teilwort uyv, das sich ringförmig aufpumpen lässt:



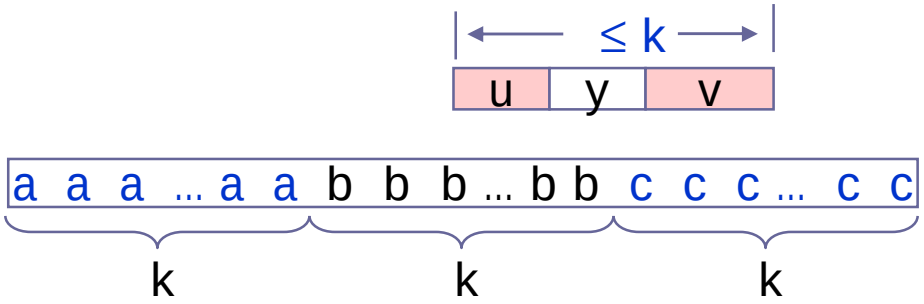


Anwendung des CF-Pumping Lemma

$L = \{a^n b^n c^n \mid n \geq 0\}$ ist nicht CF

- Angenommen L sei CF, dann gäbe es ein k wie im PL.
- Betrachte $w = a^k b^k c^k \in L$.

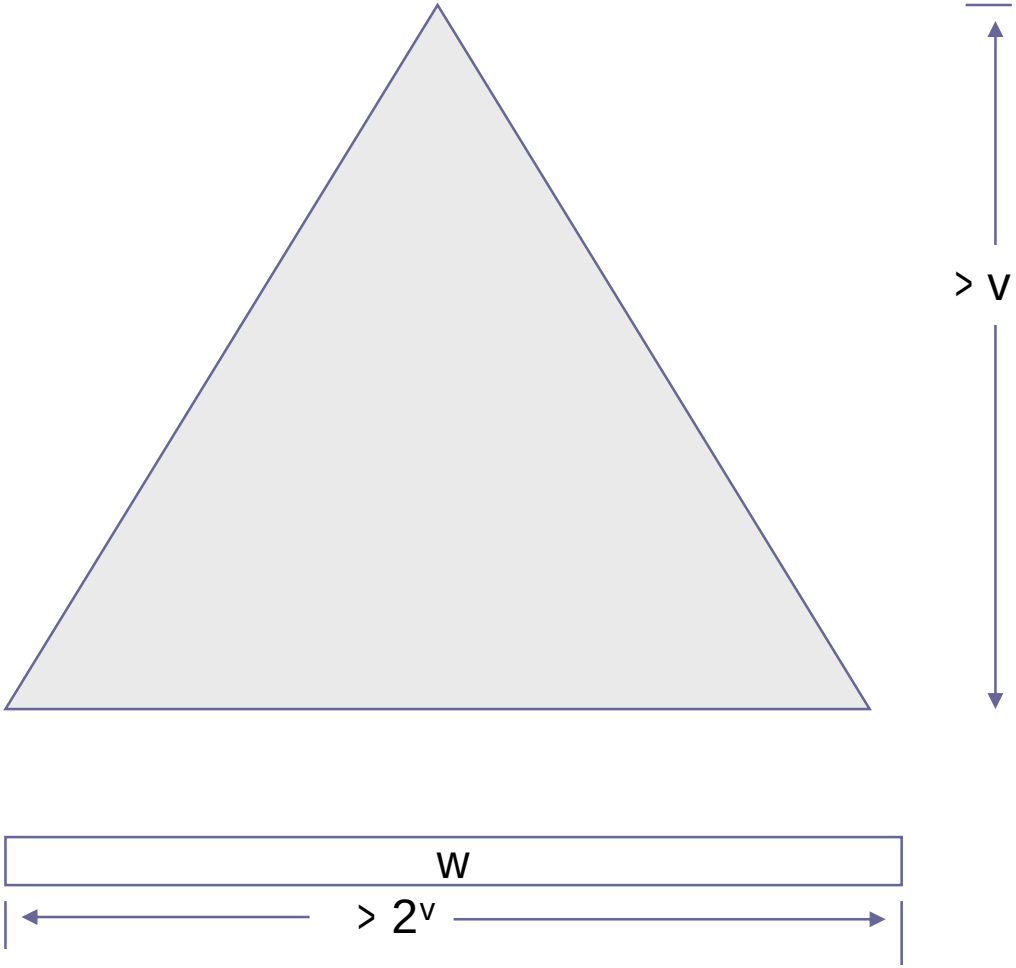
- Gemäß PL lässt sich dies zerlegen:
  $a^k b^k c^k = x\ u\ y\ v\ z$ mit $|u\ y\ v| \leq k$



In uyv kommt entweder kein a oder kein c vor. OBdA kein a.
Wegen |uv| > 0 hat dann xyz zu viele a's um in L zu sein.
Widerspruch zum PL.

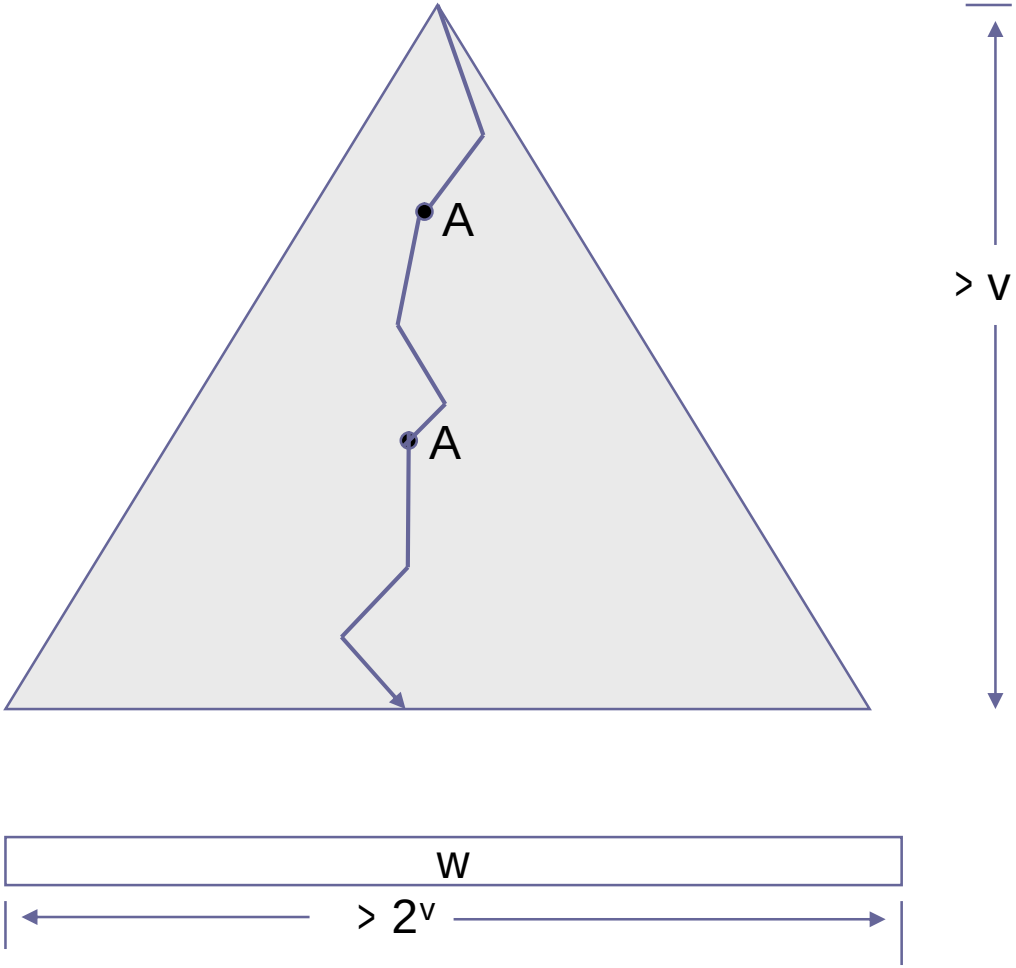
Beweis des Pumping Lemma

- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe $> v$.
-



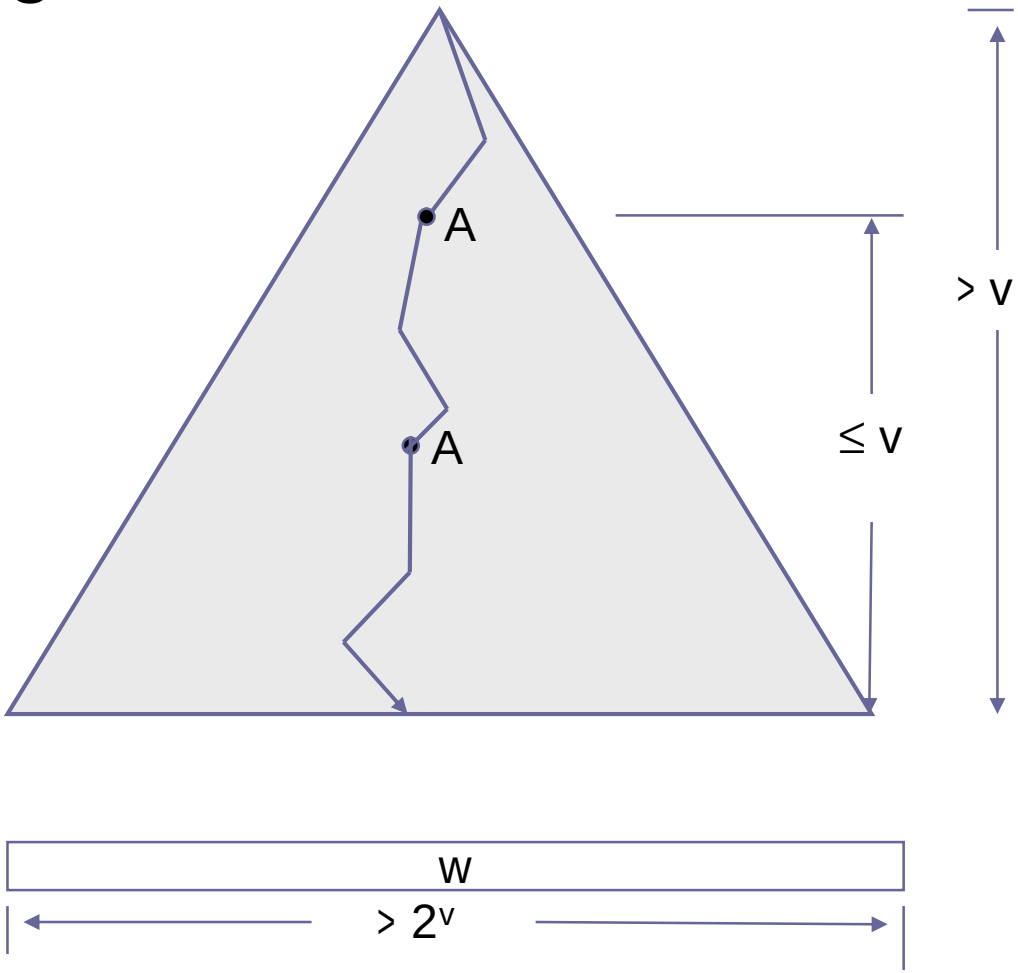
Beweis des Pumping Lemma

- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe $> v$.
- Es gibt also einen Pfad zur Wurzel, auf dem ein Nonterminal A zweimal vorkommt.



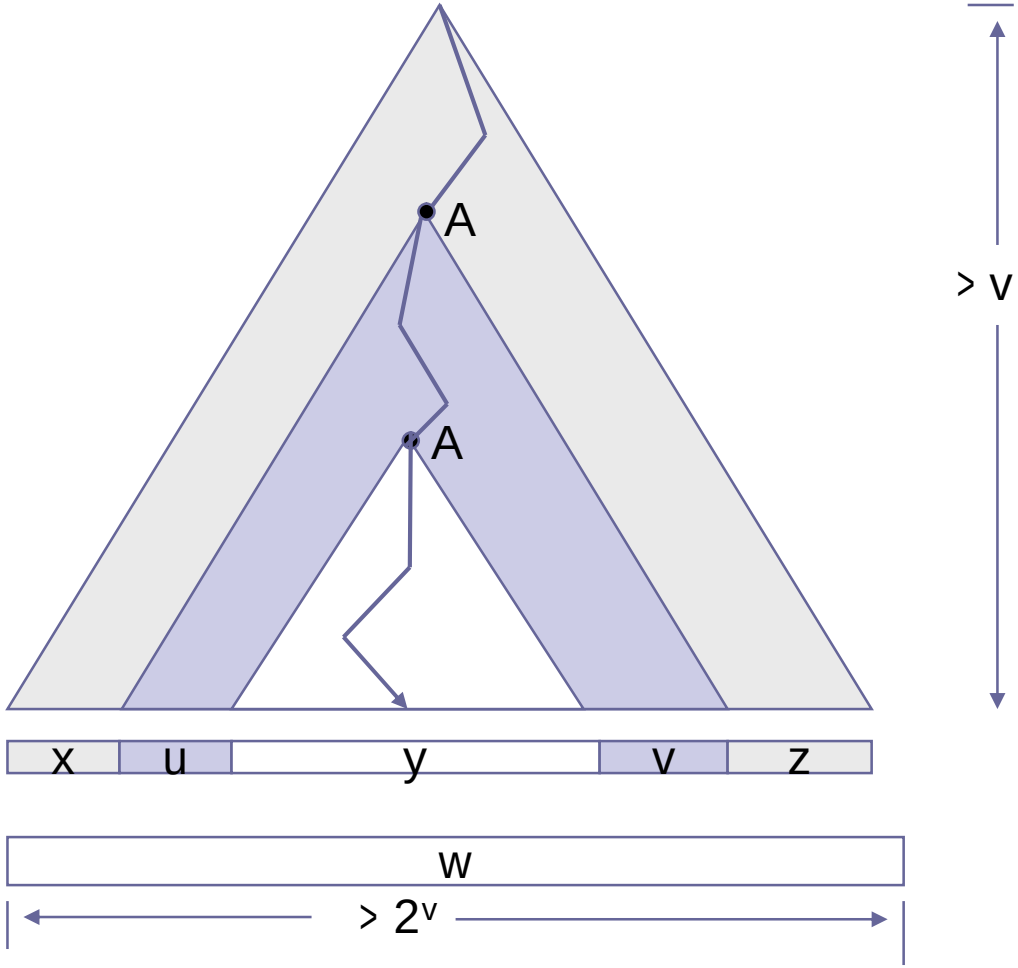
Beweis des Pumping Lemma

- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe $> v$.
- Es gibt also einen Pfad zur Wurzel, auf dem ein Nonterminal A zweimal vorkommt. Man kann die beiden A sogar von Tiefe $\leq v$ wählen.



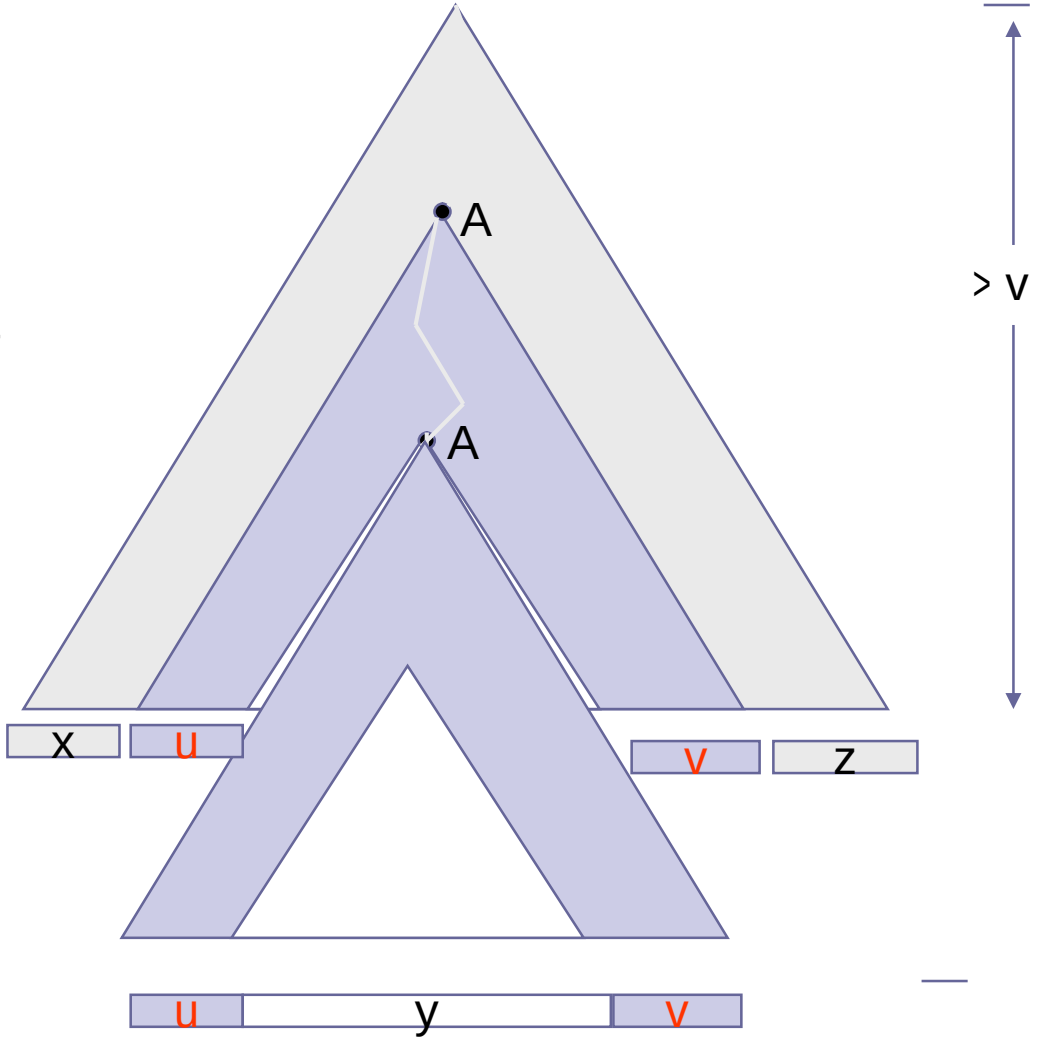
Beweis des Pumping Lemma

- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe $> v$.
- Es gibt also einen Pfad zur Wurzel, auf dem ein Nonterminal A zweimal vorkommt. Man kann die beiden A sogar von Tiefe $\leq v$ wählen.
- w zerlegt sich also als w = x u y v z wie in der Figur



Beweis des Pumping Lemma

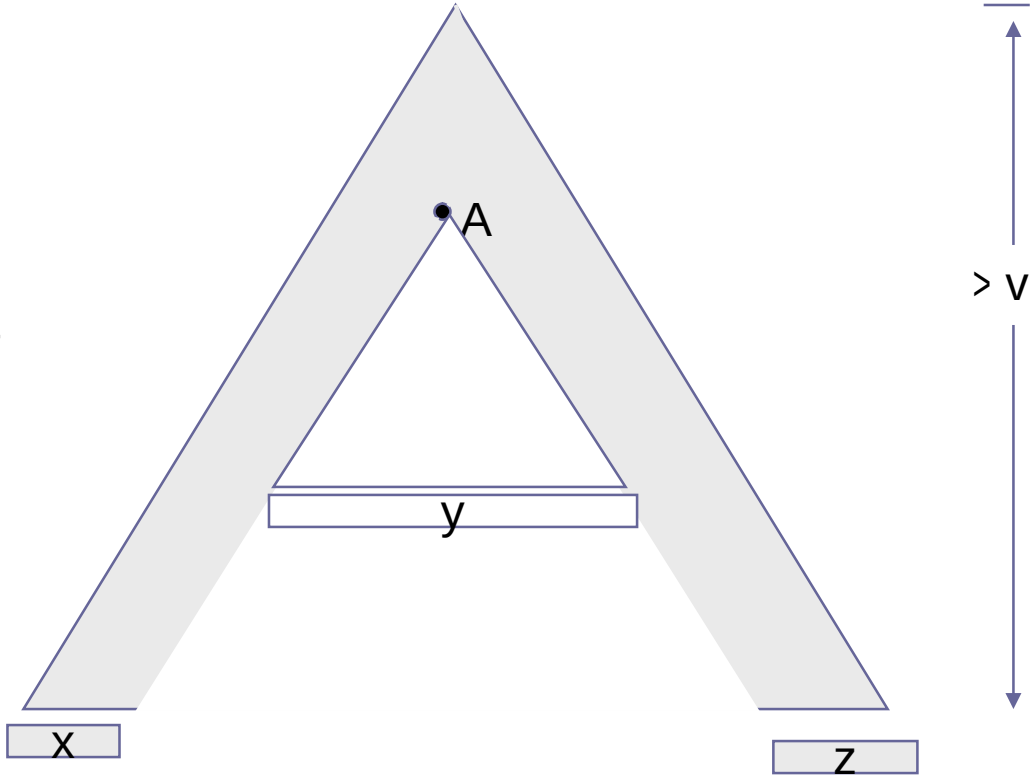
- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe > v.
- Es gibt also einen Pfad zur Wurzel, auf dem ein Nonterminal A zweimal vorkommt. Man kann die beiden A sogar von Tiefe ≤ v wählen.
- w zerlegt sich also als w = x u y v z wie in der Figur
- Wir könnten den Baum unter dem ersten Vorkommen von A auch nochmal verwenden
- und nochmal … xuuuyvvvz ∈ L



w = x u u y v v z ∈ L(G)

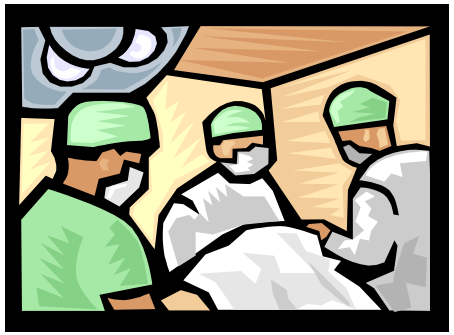
Beweis des Pumping Lemma

- Sei G eine Grammatik in Chomsky-Normalform für L mit v-1 Variablen.
- Setze $k=2^v$.
- Ist $w \in L(G)$ mit $|w| \geq k$, dann hat der Herleitungsbaum für w Tiefe > v.
- Es gibt also einen Pfad zur Wurzel, auf dem ein Nonterminal A zweimal vorkommt. Man kann die beiden A sogar von Tiefe ≤ v wählen.
- w zerlegt sich also als w = x u y v z wie in der Figur
- Wir könnten den Baum unter dem ersten Vorkommen von A auch nochmal verwenden
- und nochmal … xuuuyvvvz ∈ L
- oder auch gar nicht xyz ∈ L



w = x y z ∈ L(G)

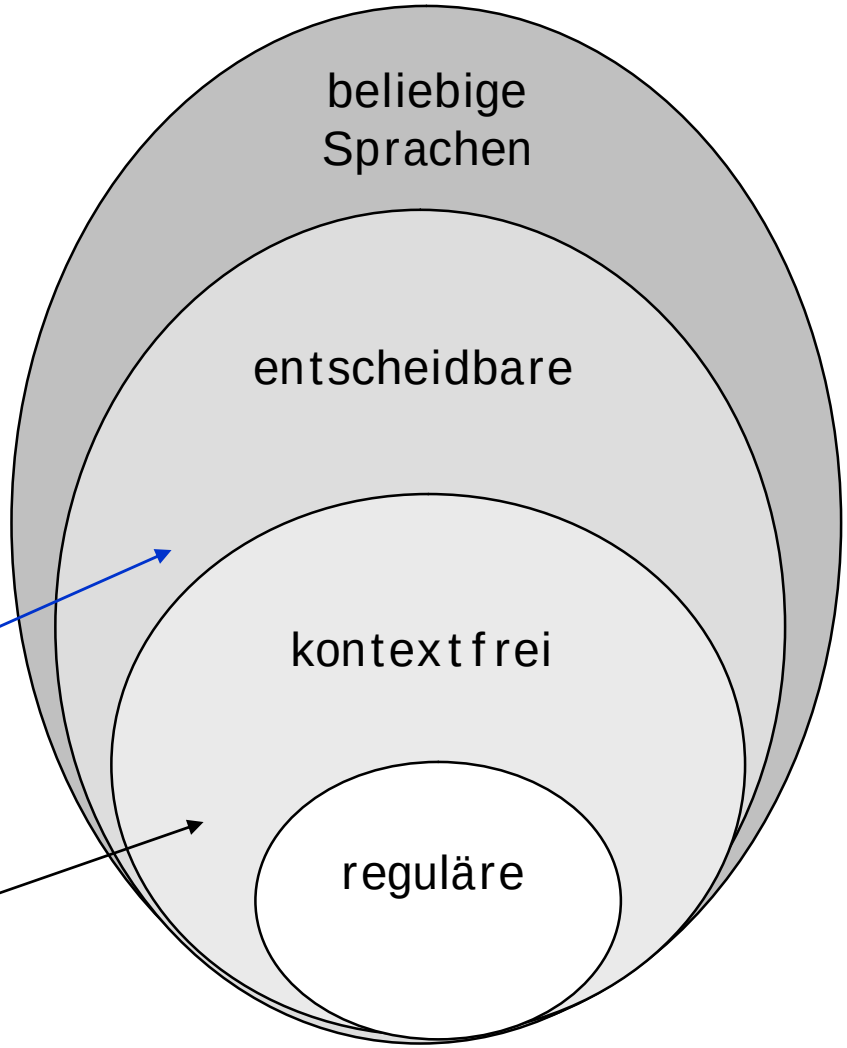
Abschluss unter Operationen



- Sind L_1, L_2, L kontextfrei, dann auch
  - $L_1 + L_2$ (Vereinigung)
  - $L_1 L_2$ (Verkettung)
  - L^* (Kleene-Stern)
- aber: kontextfreie Sprachen sind nicht abgeschlossen unter
  - $\Sigma^* - L$ (Komplement)
  - $L_1 \cap L_2$ (Schnitt)
- Gegenbeispiel :
  - $L_1 = \{ a^k b^m c^n \mid k=m \}$ Kontextfrei
  - $L_1 = \{ a^k b^m c^n \mid m=n \}$ Kontextfrei
  - $L_1 \cap L_2 = \{ a^k b^m c^n \mid k=m=n \}$ nicht Kontextfrei

Hierarchie von Sprachen

- Bisherige Hierarchie von Sprachen
  - Kann man noch verfeinern
  - Insbesondere werden wir noch Sprachen kennenlernen, die nicht entscheidbar sind



Inhalt

1. Grammatiken und Sprachen
   - Kontextfreie Grammatiken
   - Herleitungen, Linksherleitungen
   - Sprachen zu einer Grammatik
   - Äquivalenz
   - Chomsky-Normalform
   - Wortproblem, CYK
   - Pumping Lemma für CF-Sprachen

2. Stackautomaten
   - Definitionen und Beispiele
   - Konfigurationen, Läufe,
   - Sprache eines Stackautomaten
   - Parser

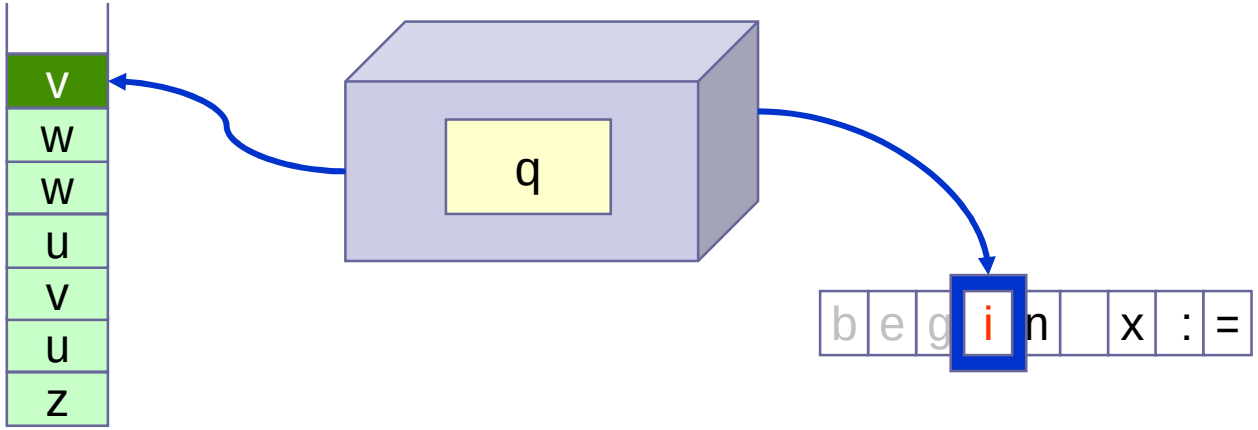
3. Parser
   - Mehrdeutigkeit
   - Bottom Up, Top Down
   - Recursive descent Parser
   - First, Follow
   - Semantische Aktionen

4. Shift-Reduce Parser
   - Konflikte
   - LR(0)-Zustände
   - Automat zur Grammatik
   - Shift-Reduce, Reduce-Reduce Konflikte
   - Präzedenz und Assoziativität
   - lex und yacc
   - Arbeitsweise eines Compilers

Stackautomat (Kellerautomat)

- DFA mit Stack. Dadurch kann sich der Automat beliebig viele Informationen “merken”.

In jedem Schritt kann er neben seinem Zustand auch noch das oberste Element des Stacks zu Rate ziehen und endlich viele neue Elemente auf dem Stack ablegen.



Im deutschen Sprachbereich findet man häufig das Wort “Keller” statt des Englischen “Stack” .

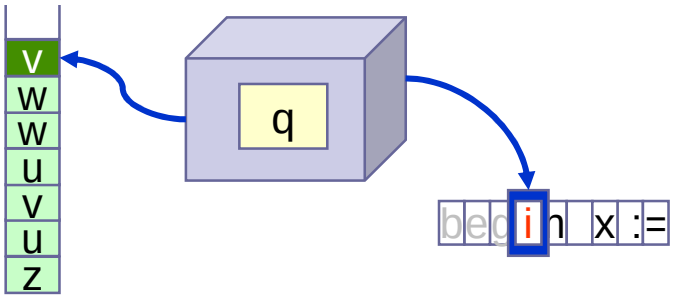
Nichtdeterministischer Stackautomat

Ein nichtdeterministischer Stackautomat (PDA) ist ein 6-Tupel

$$M = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$$

bestehend aus

- Q - endliche Menge (Zustände)
- Σ - endliches Alphabet (Inputalphabet)
- Γ - endliches Alphabet (Stackalphabet)
- $\delta : Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \rightarrow \wp_{finite}(Q \times \Gamma^*)$ (Transitionsfunktion)
- $q_0 \in Q$ (Startzustand)
- $s_0 \in \Gamma$ (Stack-Startsymbol)



Nichtdeterministischer Stackautomat

Ein nichtdeterministischer Stackautomat (PDA) ist ein 6-Tupel

$$M = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$$

bestehend aus

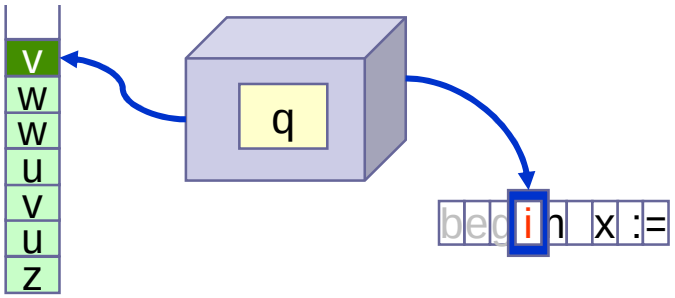
- Q - endliche Menge (Zustände)
- Σ - endliches Alphabet (Inputalphabet)
- Γ - endliches Alphabet (Stackalphabet)
- $\delta : Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \rightarrow \wp_{finite}(Q \times \Gamma^*)$ (Transitionsfunktion)
- $q_0 \in Q$ (Startzustand)
- $s_0 \in \Gamma$ (Stack-Startsymbol)

Interpretation von $(q', g_1 \ldots g_k) \in \delta\,(q,a,g)$:

Falls

Zustand = q
Input = a
top(Stack) = g,

dann



Nichtdeterministischer Stackautomat

Ein nichtdeterministischer Stackautomat (PDA) ist ein 6-Tupel

$$M = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$$

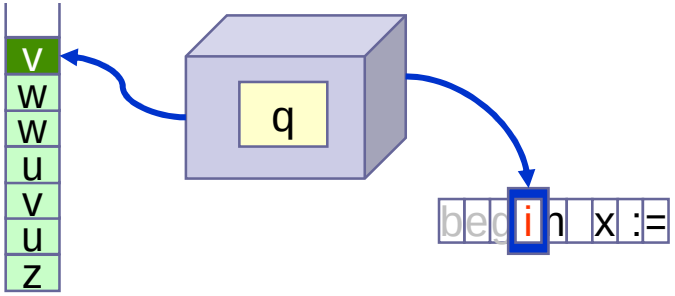
bestehend aus

- Q - endliche Menge (Zustände)
- Σ - endliches Alphabet (Inputalphabet)
- Γ - endliches Alphabet (Stackalphabet)
- $\delta : Q \times (\Sigma \cup \{ \varepsilon \}) \times \Gamma \rightarrow \wp_{finite}(Q \times \Gamma^*)$ (Transitionsfunktion)
- $q_0 \in Q$ (Startzustand)
- $s_0 \in \Gamma$ (Stack-Startsymbol)

Interpretation von $(q', g_1 \ldots g_k) \in \delta(q,a,g)$:

Falls
- Zustand = q
- Input = a
- top(Stack) = g,

dann
- neuer Zustand = q'
- pop, push g_k, ..., push g_1



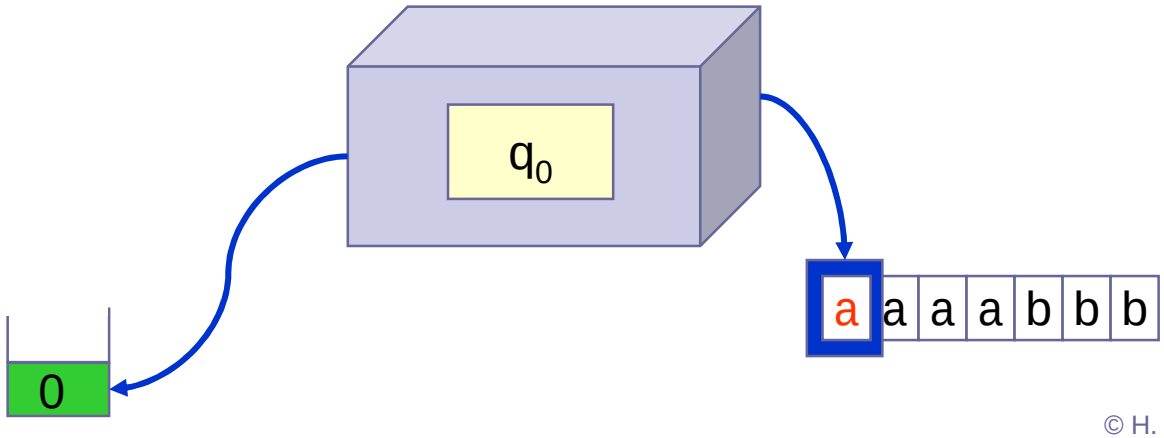
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0)$	$= \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0)$	$= \{ (q_1, 10) \}$
$\delta(q_1, a, 1)$	$= \{ (q_1, 11) \}$
$\delta(q_1, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0)$	$= \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k)$	$= \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$

$\Sigma = \{a, b\}$

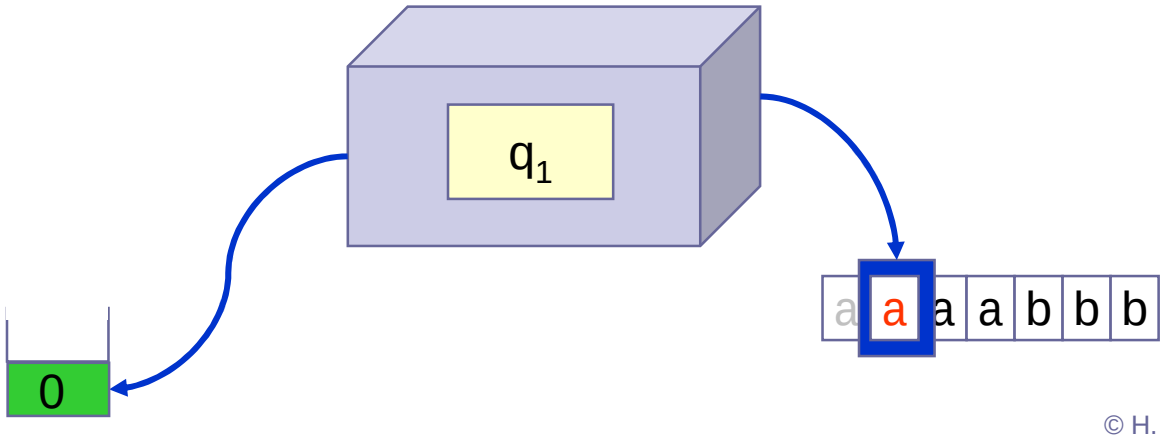
$\Gamma = \{ 0,1 \}$

$s_0 = 0$

M

$\delta(q_0, a, 0)$	$= \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0)$	$= \{ (q_1, 10) \}$
$\delta(q_1, a, 1)$	$= \{ (q_1, 11) \}$
$\delta(q_1, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0)$	$= \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k)$	$= \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$

$\Sigma = \{a, b\}$

$\Gamma = \{ 0,1 \}$

$s_0 = 0$

M

$\delta(q_0, a, 0) = \{ (q_1 , 0) , (q_1 , 10) \}$

$\delta(q_1, a, 0) = \{ (q_1, 10) \}$

$\delta(q_1, a, 1) = \{ (q_1, 11) \}$

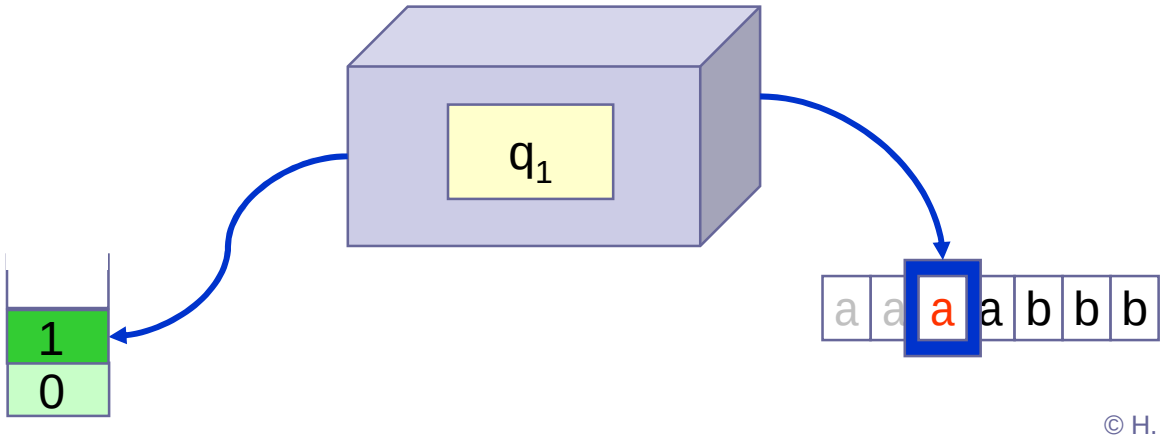
$\delta(q_1, b, 1) = \{ (q_2, \varepsilon) \}$

$\delta(q_2, b, 1) = \{ (q_2, \varepsilon) \}$

$\delta(q_2, \varepsilon, 0) = \{ (q_2 , \varepsilon) \}$

$\delta(q_i,u,k) = \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



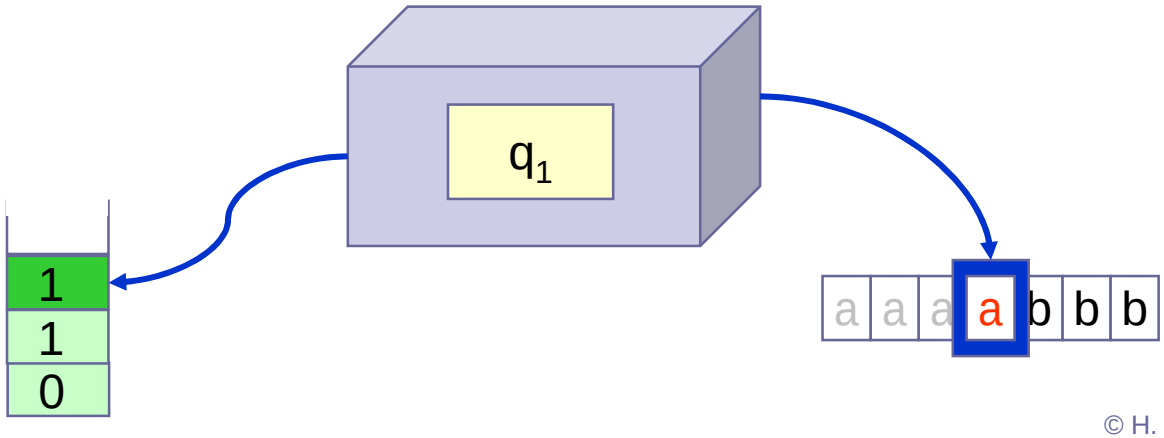
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0) = \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0) = \{ (q_1, 10) \}$
$\delta(q_1, a, 1) = \{ (q_1, 11) \}$
$\delta(q_1, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0) = \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



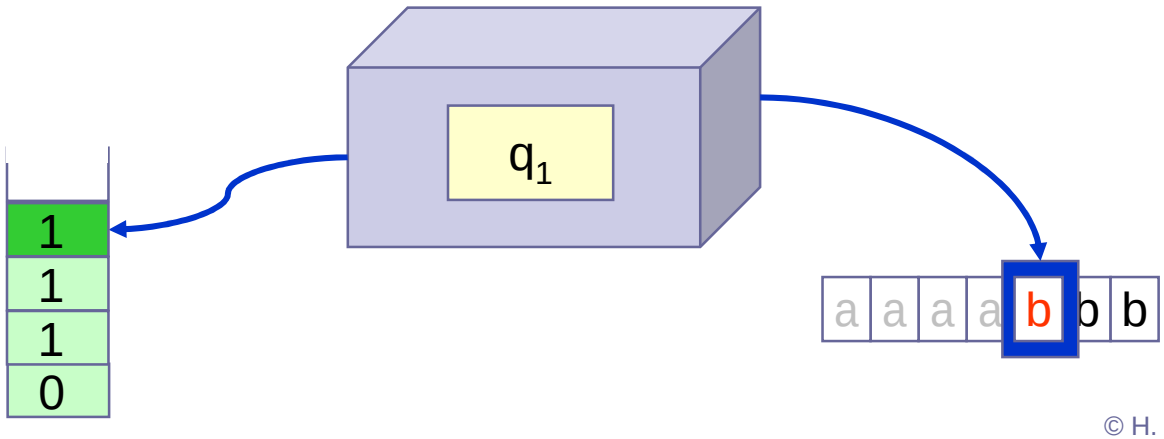
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0) = \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0) = \{ (q_1, 10) \}$
$\delta(q_1, a, 1) = \{ (q_1, 11) \}$
$\delta(q_1, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0) = \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



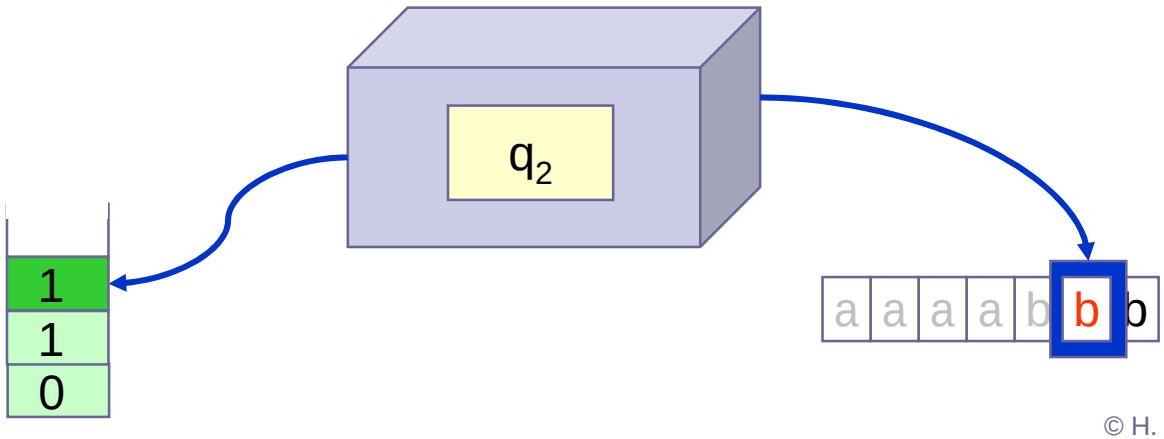
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0)$	$= \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0)$	$= \{ (q_1, 10) \}$
$\delta(q_1, a, 1)$	$= \{ (q_1, 11) \}$
$\delta(q_1, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0)$	$= \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k)$	$= \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



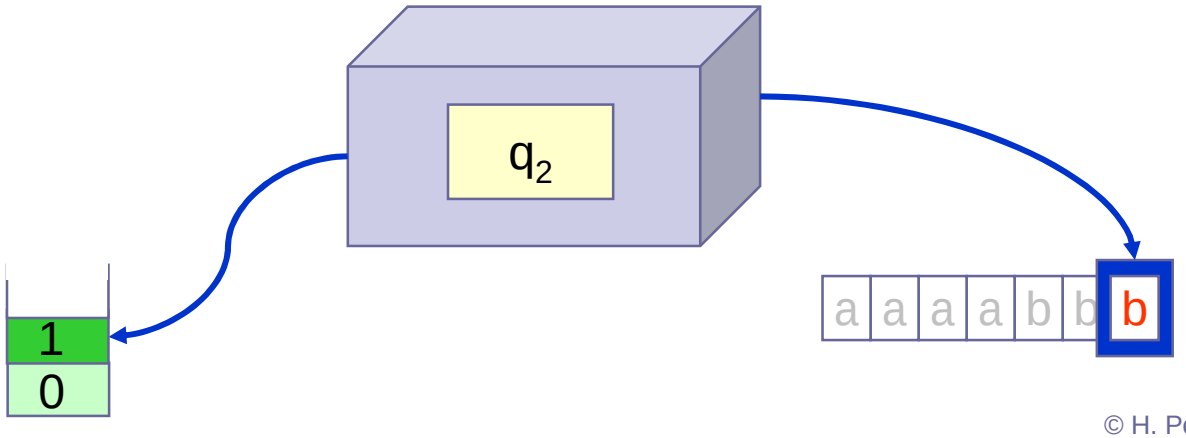
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0)$	$= \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0)$	$= \{ (q_1, 10) \}$
$\delta(q_1, a, 1)$	$= \{ (q_1, 11) \}$
$\delta(q_1, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1)$	$= \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0)$	$= \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k)$	$= \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



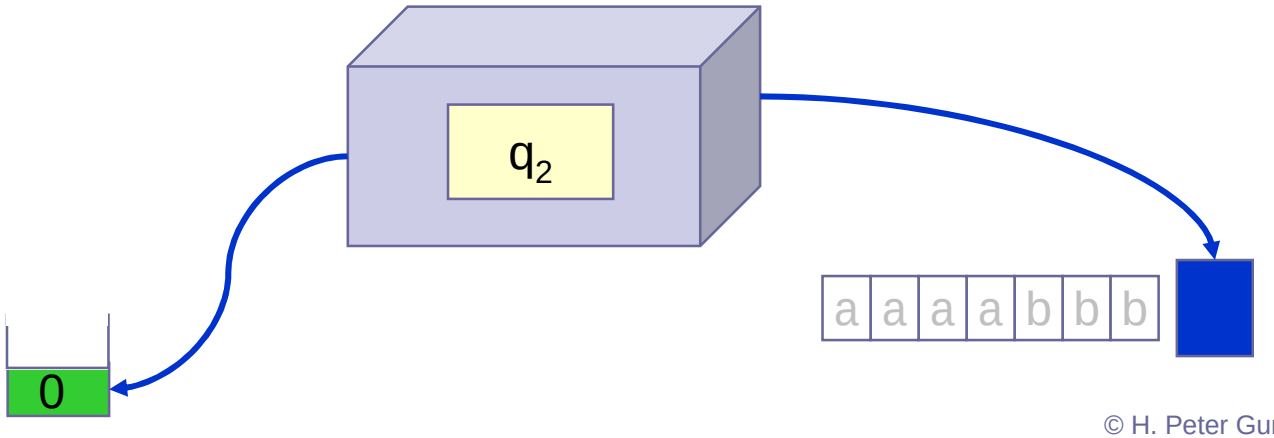
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0) = \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0) = \{ (q_1, 10) \}$
$\delta(q_1, a, 1) = \{ (q_1, 11) \}$
$\delta(q_1, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0) = \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



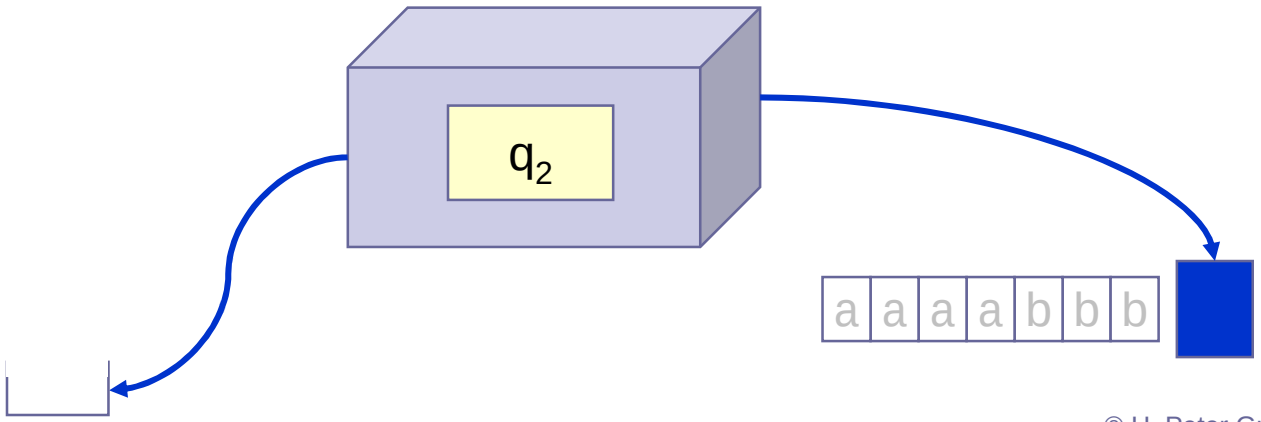
Beispiel einer Stackmaschine

$Q = \{ q_0, q_1, q_2 \}$
$\Sigma = \{a, b\}$
$\Gamma = \{ 0,1 \}$
$s_0 = 0$

M

$\delta(q_0, a, 0) = \{ (q_1 , 0) , (q_1 , 10) \}$
$\delta(q_1, a, 0) = \{ (q_1, 10) \}$
$\delta(q_1, a, 1) = \{ (q_1, 11) \}$
$\delta(q_1, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, b, 1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2, \varepsilon, 0) = \{ (q_2 , \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

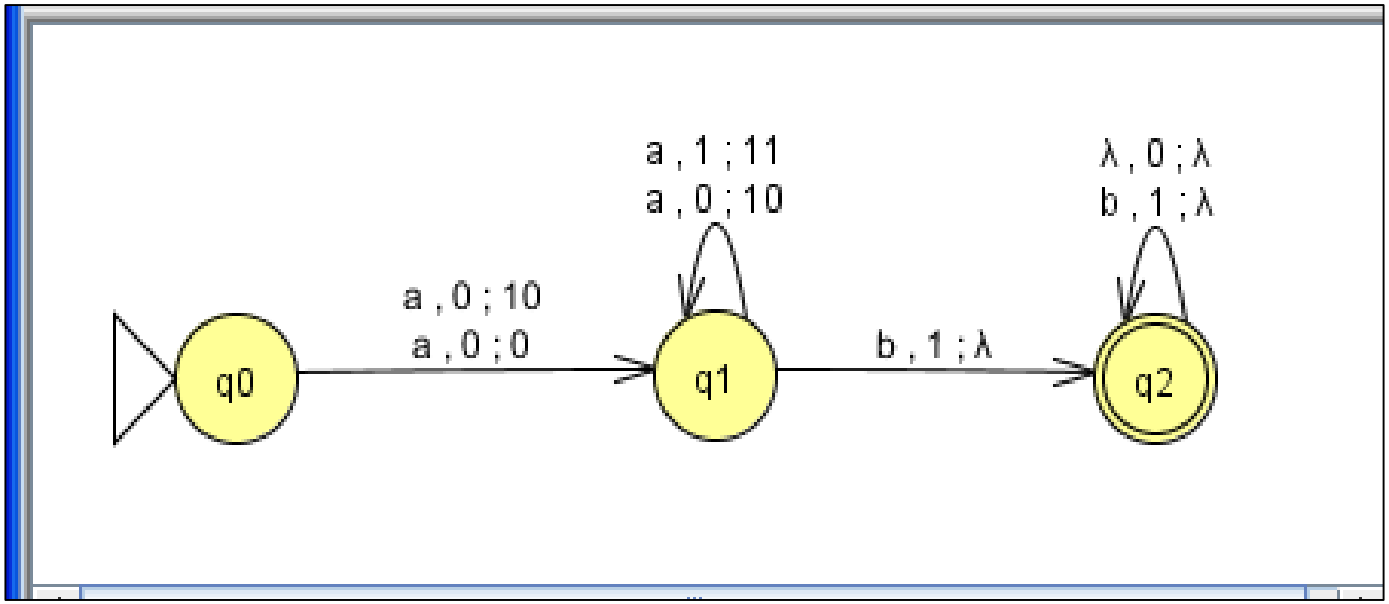
Stackmaschine beginnt immer mit s auf dem ansonsten leeren Stack.
Im Beispiel kann das Wort aaaabbb den Stack entleeren :



Die Stackmaschine in JFLAP

Es gibt Varianten in der Definition von Stackautomaten. Beispielsweise kann man Endzustände einführen und andere Terminierungsbedingungen.

Wir werden sehen, dass für die Spracherkennung Zustände unwichtig sind: man kommt mit einem Zustand aus, könnte daher auf Zustände gänzlich verzichten.



Die Stackmaschine von vorhin in JFLAP

Konfiguration und Übergang

Konfiguration $\kappa = (q, w, s)$ besteht aus :

- q - gegenwärtigen Zustand,
- w - Rest des noch zu lesenden Inputwortes,
- s - gegenwärtigen Stack.

$\sigma \vdash \tau \quad :\Leftrightarrow$ Automat kann von σ in τ übergehen

Inputzeichen wird verbraucht:

$$(q, a.w, g.u) \vdash (p, w, vu) \Leftrightarrow (p,v) \in \delta(q,a,g)$$

Kein Inputzeichen verbraucht

$$(q, w, g.u) \vdash (p, w, vu) \Leftrightarrow (p,v) \in \delta(q,\varepsilon,g)$$

Lauf

Ein Lauf ist eine Folge von Übergängen

M

$\delta(q_0,a,0) = \{ (q_1,0), (q_1,10) \}$

$\delta(q_1,a,0) = \{ (q_1, 10) \}$
$\delta(q_1,a,1) = \{ (q_1, 11) \}$
$\delta(q_1,b,1) = \{ (q_2, \varepsilon) \}$

$\delta(q_2,b,1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2,\varepsilon,0) = \{ (q_2, \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

$(q_0,aabb,0) \vdash (q_1,abb,10)$
$\vdash (q_1,bb,110)$

$\vdash (q_2,b,10)$
$\vdash (q_2,\varepsilon,0)$
$\vdash (q_2,\varepsilon,\varepsilon)$

d.h. $(q_0,aabb,0) \vdash^* (q_2,\varepsilon,\varepsilon)$

Zwei Läufe

$(q_0,aaabb,0) \vdash (q_1,aabb,0)$
$\vdash (q_1,abb,10)$
$\vdash (q_1,bb,110)$
$\vdash (q_2,b,10)$
$\vdash (q_2,\varepsilon,0)$
$\vdash (q_2,\varepsilon,\varepsilon)$

also : $(q_0,aaabb,0) \vdash^* (q_2,\varepsilon,\varepsilon)$

Sprache einer Stackmaschine

- *Sprache einer Stackmaschine*: Menge aller Worte, die von Anfangs- in eine Endkonfiguration überführen können.

$$L(M) = \{ w \in \Sigma^* \mid (q_0, w, s_0) \vdash^* (q_t, \varepsilon, \varepsilon) \}$$

M

$\delta(q_0,a,0) = \{ (q_1,0), (q_1,10) \}$

$\delta(q_1,a,0) = \{ (q_1, 10) \}$
$\delta(q_1,a,1) = \{ (q_1, 11) \}$
$\delta(q_1,b,1) = \{ (q_2, \varepsilon) \}$

$\delta(q_2,b,1) = \{ (q_2, \varepsilon) \}$
$\delta(q_2,\varepsilon,0) = \{ (q_2, \varepsilon) \}$
$\delta(q_i,u,k) = \{ \}$ sonst

Sprache von M im Beispiel:

$L(M) = \{ a^n b^m \mid n = m \vee n=m+1 \}$

Stackmaschinen für kontextfreie Sprachen

Satz : Für jede kontextfreie Sprache L gibt es eine non-deterministische Stackmaschine M, so daß L = L(M) .

Aus $G = (V, T, P, S)$ konstruiere $M(G) = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$ mit

- $Q = \{S\}$ - nur ein Zustand, also irrelevant
- $\Sigma = T$ - Input besteht aus Token
- $\Gamma = V \cup T$ - Stack für Terminale und Nonterminale
- $q_0 = s_0 = S$ - das Startsymbol

M(G)

$\delta(S, a, a) = \{(S, \varepsilon)\}$ für alle $a \in T$

Zustand irrelevant

$\delta(S, \varepsilon, B) = \{ (S, \beta) \mid B \rightarrow \beta \in P \}$

Stackmaschinen für kontextfreie Sprachen

Satz : Für jede kontextfreie Sprache L gibt es eine non-deterministische Stackmaschine M, so daß L = L(M) .

Aus $G = (V, T, P, S)$ konstruiere $M(G) = (Q, \Sigma, \Gamma, \delta, q_0, s_0)$ mit

- $Q = \{S\}$ - nur ein Zustand, also irrelevant
- $\Sigma = T$ - Input besteht aus Token
- $\Gamma = V \cup T$ - Stack für Terminale und Nonterminale
- $q_0 = s_0 = S$ - das Startsymbol

M(G)

$\delta(a, a) = \{ \varepsilon \}$ für alle $a \in T$

$\delta(\varepsilon, B) = \{ \beta \mid B \to \beta \in P \}$

Zustand irrelevant

$\delta : (\Sigma \cup \{ \varepsilon \}) \times \Gamma \to \wp_{\text{finite}}(\Gamma^*)$

Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

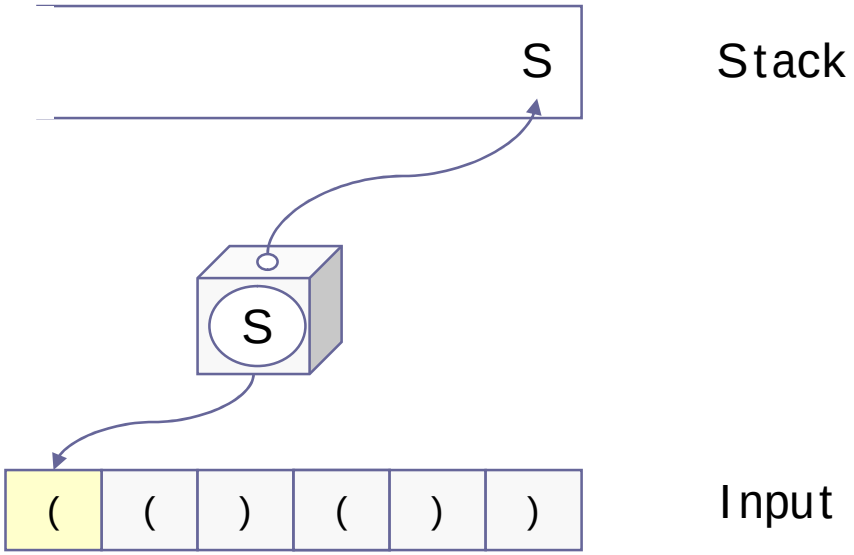
$\delta(a, a) = \varepsilon$ für alle $a \in T$

$\delta(\varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

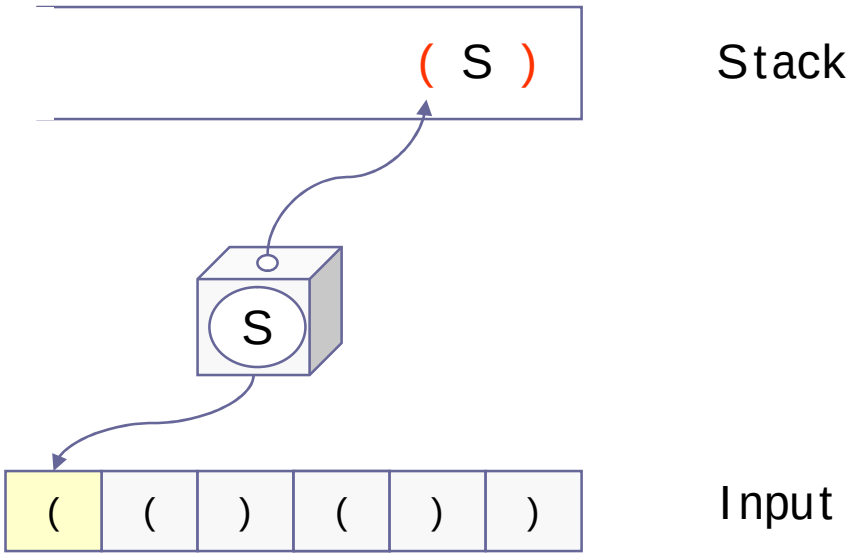
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

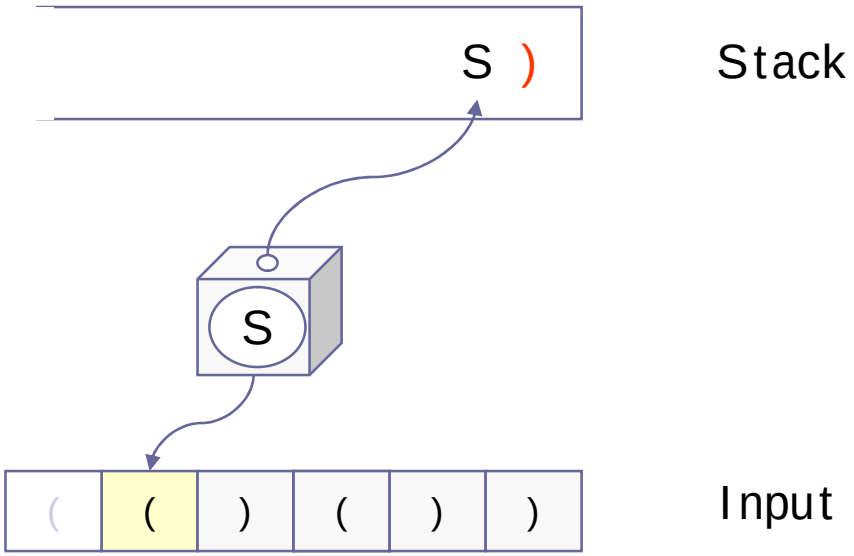
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

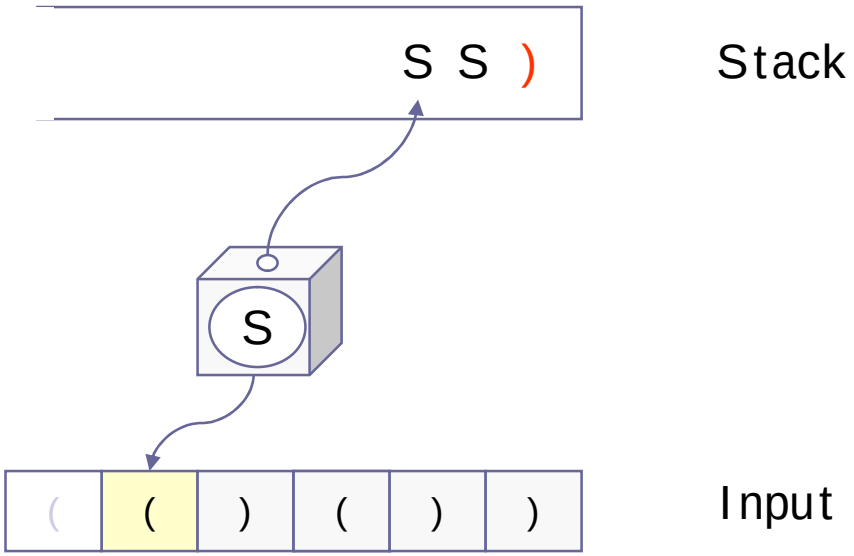
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

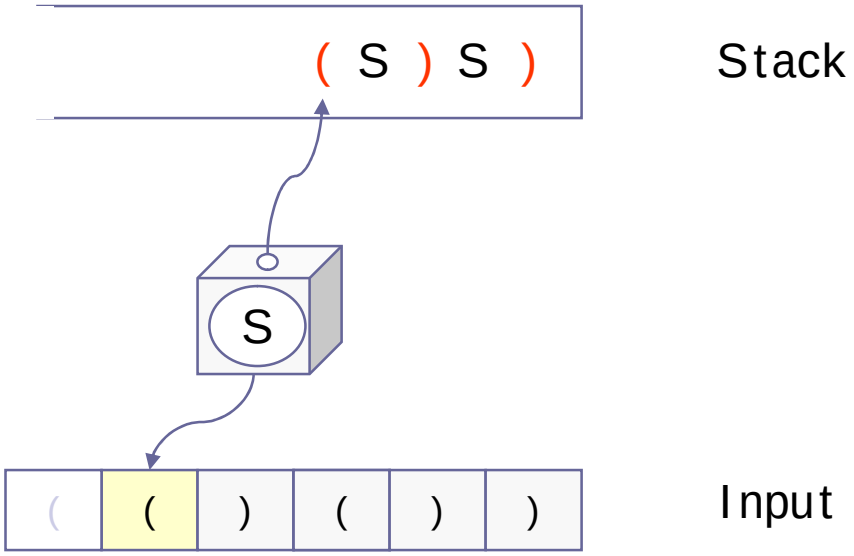
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

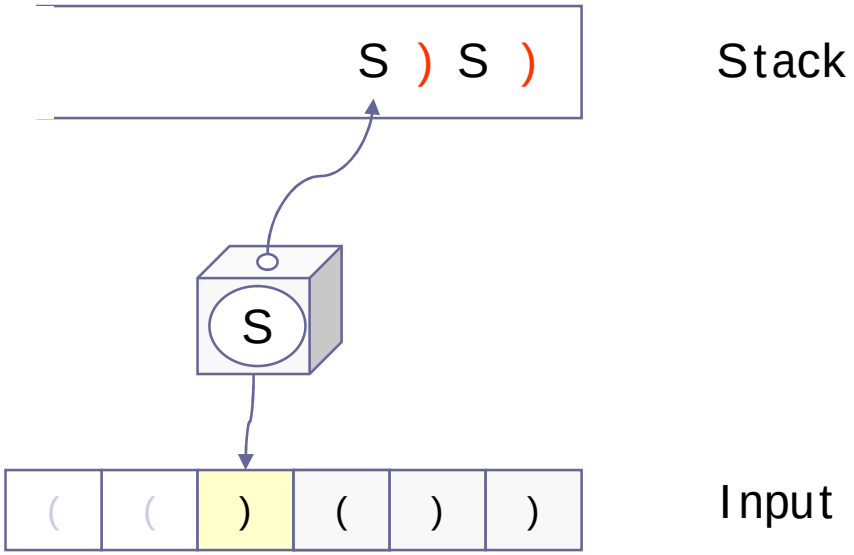
M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

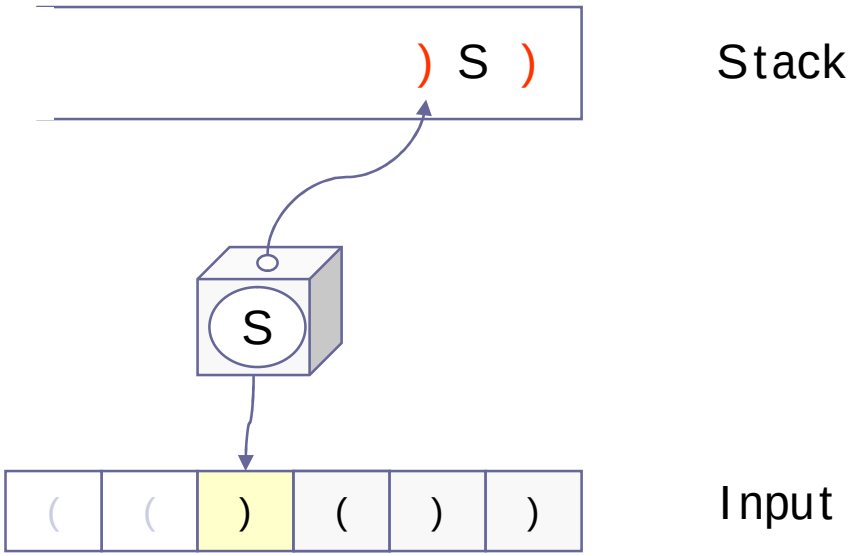
δ(S, a, a) = ε für alle a ∈ T

δ(S, ε, B) = { β | B → β ∈ P }

K

S → S S
| (S)
| ε

Invariante:
Satzform auf Stack
≈ Rest des Inputs



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

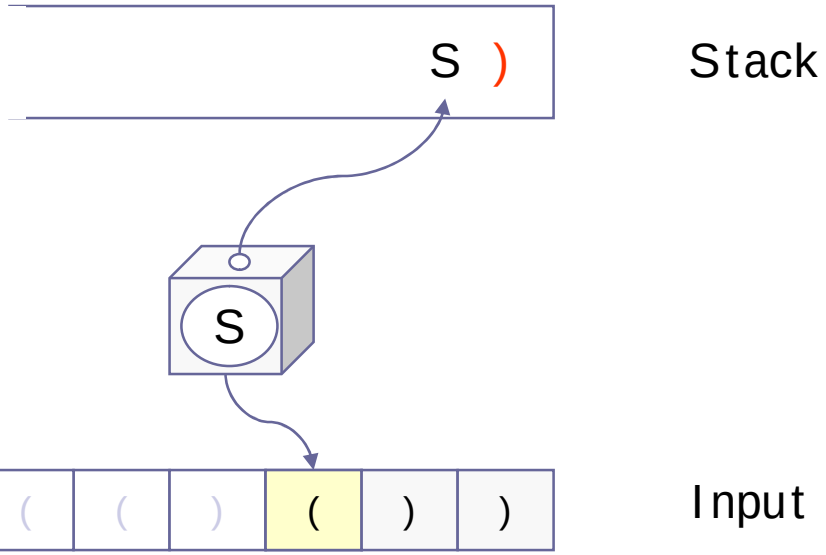
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

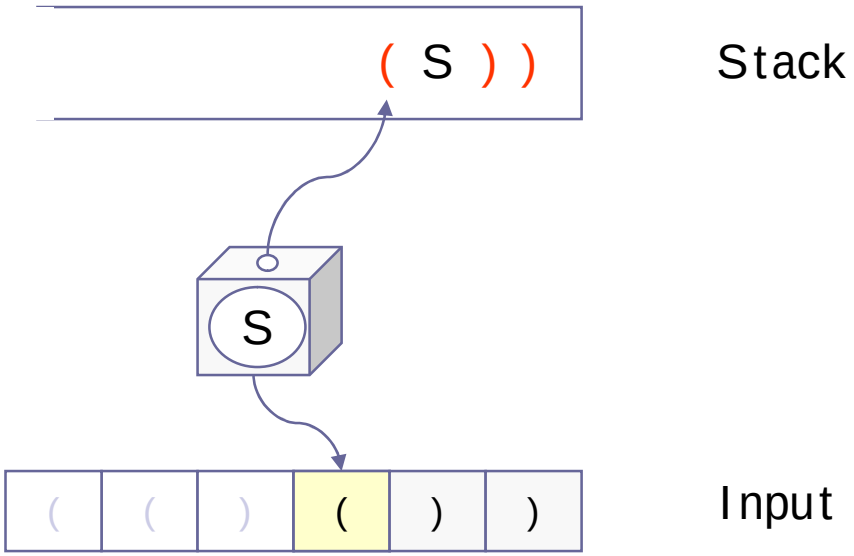
δ(S, a, a) = ε für alle a ∈ T

δ(S, ε, B) = { β | B → β ∈ P }

K

S → S S
| (S)
| ε

Invariante:
Satzform auf Stack
≈ Rest des Inputs



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

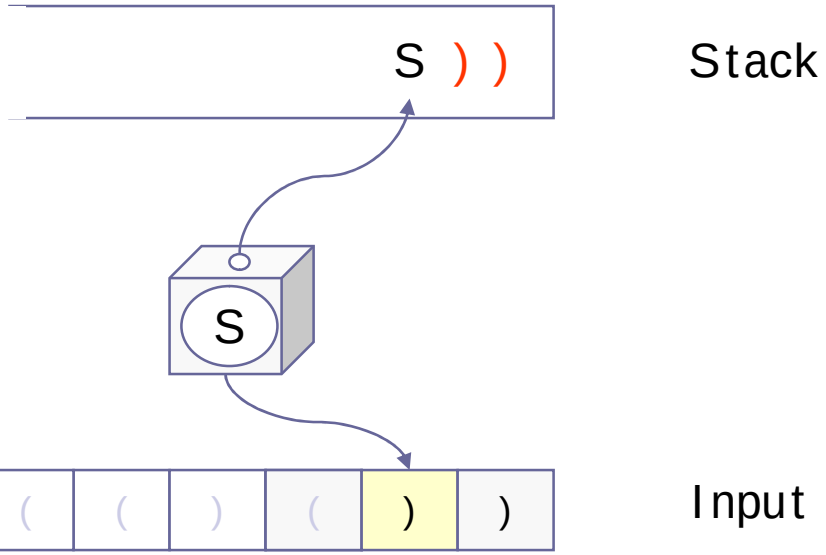
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

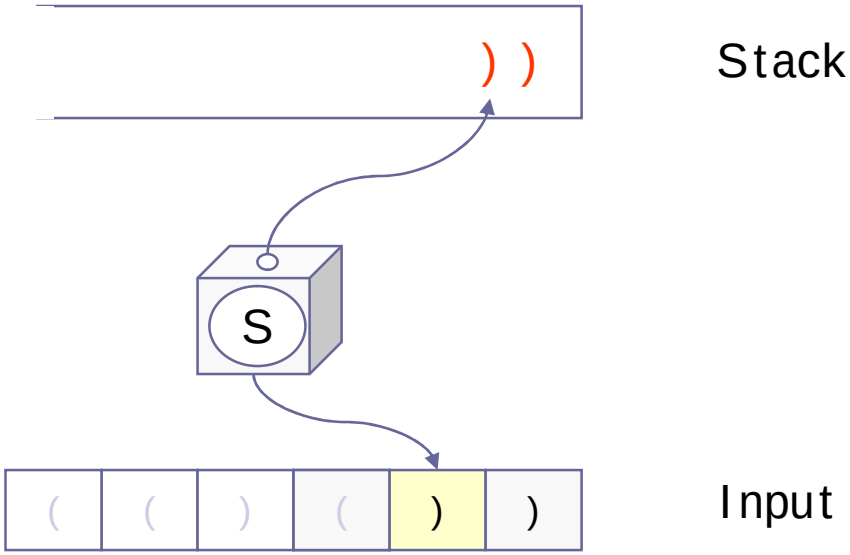
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

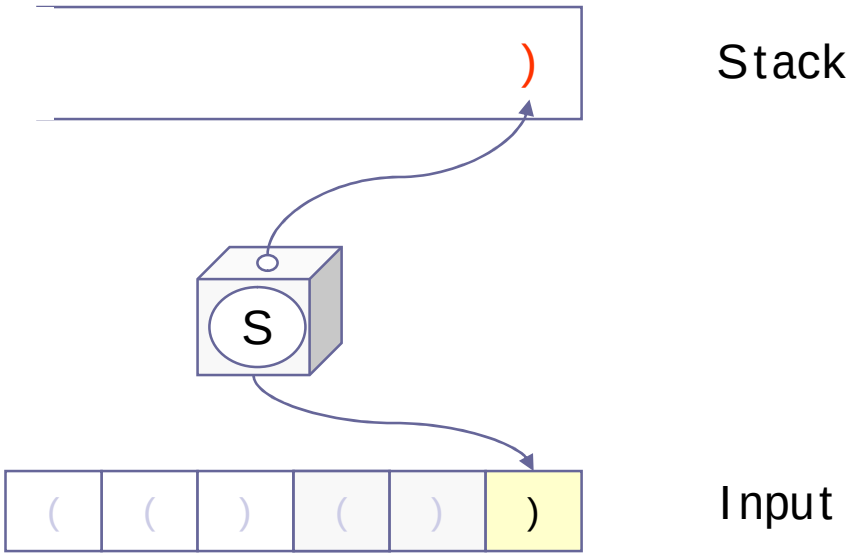
$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Invariante:
Satzform auf Stack
≈ Rest des Inputs

K

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



Beispiel: Maschine für Klammersprache

M(K) = ({S}, {),(}, {), (, S } ,δ,S, S)

M(G)

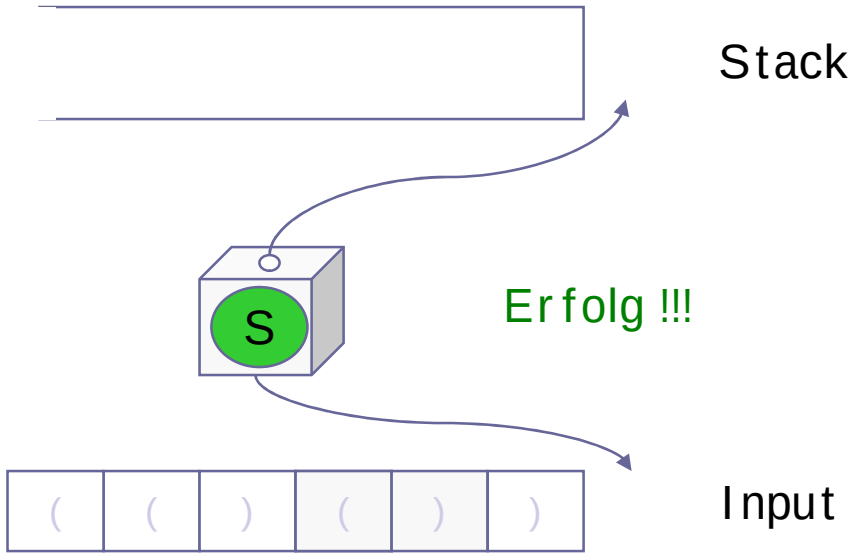
δ(S, a, a) = ε für alle a ∈ T

δ(S, ε, B) = { β | B → β ∈ P }

K

S → S S
| (S)
| ε

Invariante:
Satzform auf Stack
≈ Rest des Inputs



Stackmaschinen für kontextfreie Sprachen

Satz : Für jede kontextfreie Sprache L gibt es eine nicht-deterministische Stackmaschine M, so daß L = L(M) .

M(G)

$\delta(S,a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Beweis: Die Stackmaschine mache n Schritte und akzeptiere w. Im i-ten Schritt sei u_i der bereits verarbeitete Input und v_i der Rest (also immer $u_i v_i = w$) und α_i der Inhalt des Stacks.

Behauptung:

$S = u_0\alpha_0 \Rightarrow u_1\alpha_1 \Rightarrow \ldots \Rightarrow u_n\alpha_n = u_n$ ist eine Linksableitung von u_n.

Klar, denn anfangs ist $u_0=\varepsilon$ und $\alpha_0=S$ und am Schluss gilt $u_n=w$.

Für den k-ten Schritt gilt

entweder

$\alpha_k = a.\alpha_{k+1}$ und $v_k = a.v_{k+1}$ und $u_{k+1} = u_k a$ // Regel 1

also $u_k\alpha_k = u_{k+1}\alpha_{k+1}$

oder $\alpha_k = B.\gamma$, $\alpha_{k+1} = \beta.\gamma$ für ein $B \rightarrow \beta \in P$ // Regel 2

also $u_k\alpha_k = u_k B.\gamma \Rightarrow u_k \beta.\gamma = u_{k+1}\alpha_{k+1}$

in jedem Fall ein (evtl. trivialer) Schritt einer Linksherleitung

Umgekehrt, kann man jede Linksherleitung simulieren.

Anfangssituation

M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Linksherleitung:

$S = u_0 \alpha_0$

$\Rightarrow \ldots$

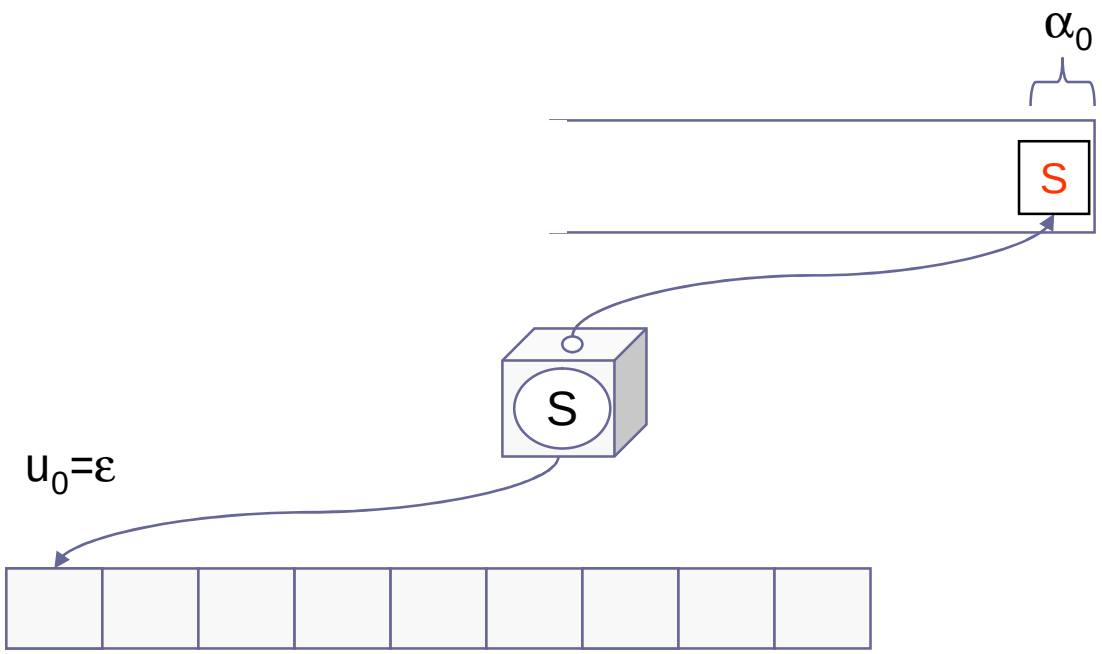
$\Rightarrow u_k \alpha_k$

$\Rightarrow$

$u_{k+1} \alpha_{k+1}$

$\Rightarrow \ldots$

$\Rightarrow w$



Stack

Input

Anwendung der ersten Regel

M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Linksherleitung:

$S = u_0 \alpha_0$

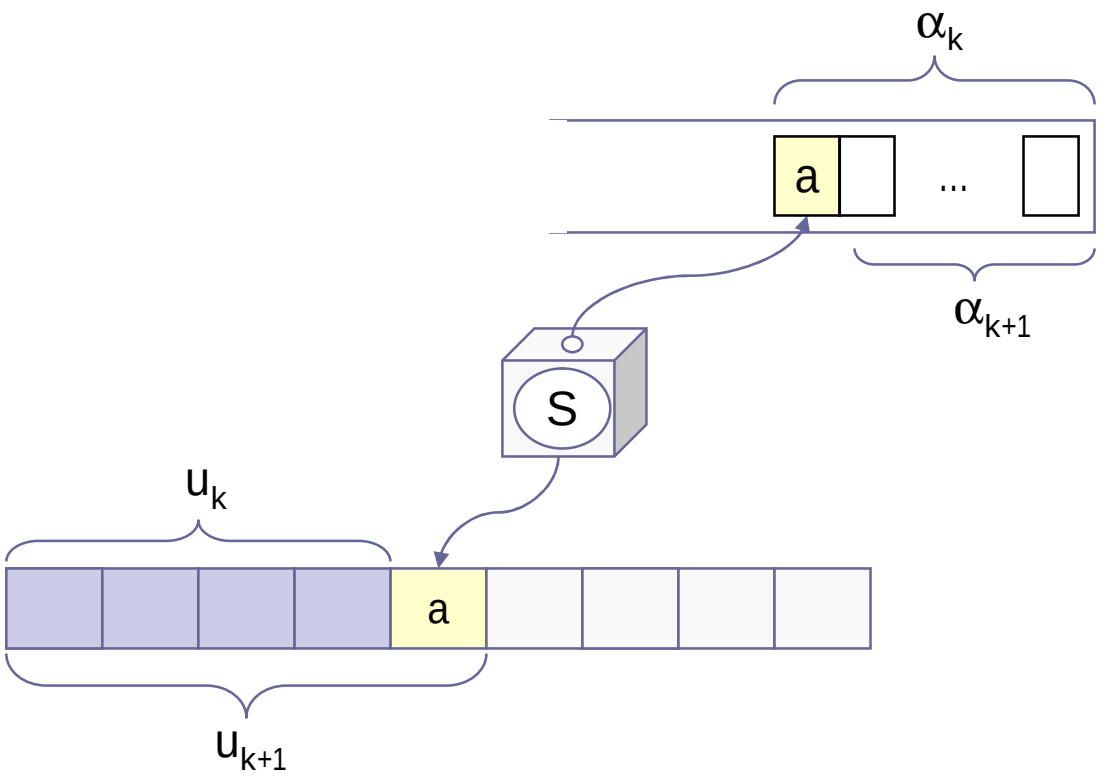
$\Rightarrow \ldots$

$\Rightarrow u_k \alpha_k$

$= u_{k+1} \alpha_{k+1}$

$\Rightarrow \ldots$

$\Rightarrow w$



Stack

Input

Anwendung der zweiten Regel

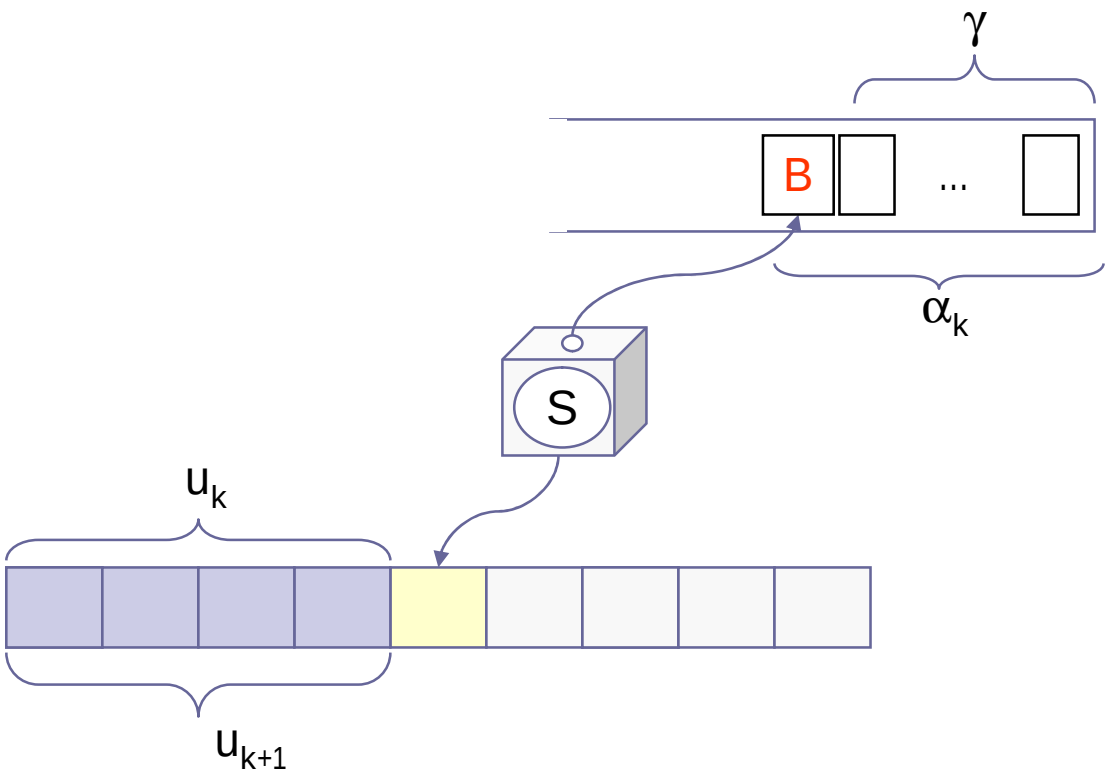
M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Linksherleitung:

$S = u_0 \alpha_0$
$\Rightarrow \ldots$
$\Rightarrow u_k \alpha_k$
$= u_k B \gamma$
$\Rightarrow u_k \beta \gamma$
$= u_{k+1} \alpha_{k+1}$
$\Rightarrow \ldots$
$\Rightarrow w$



Anwendung der zweiten Regel

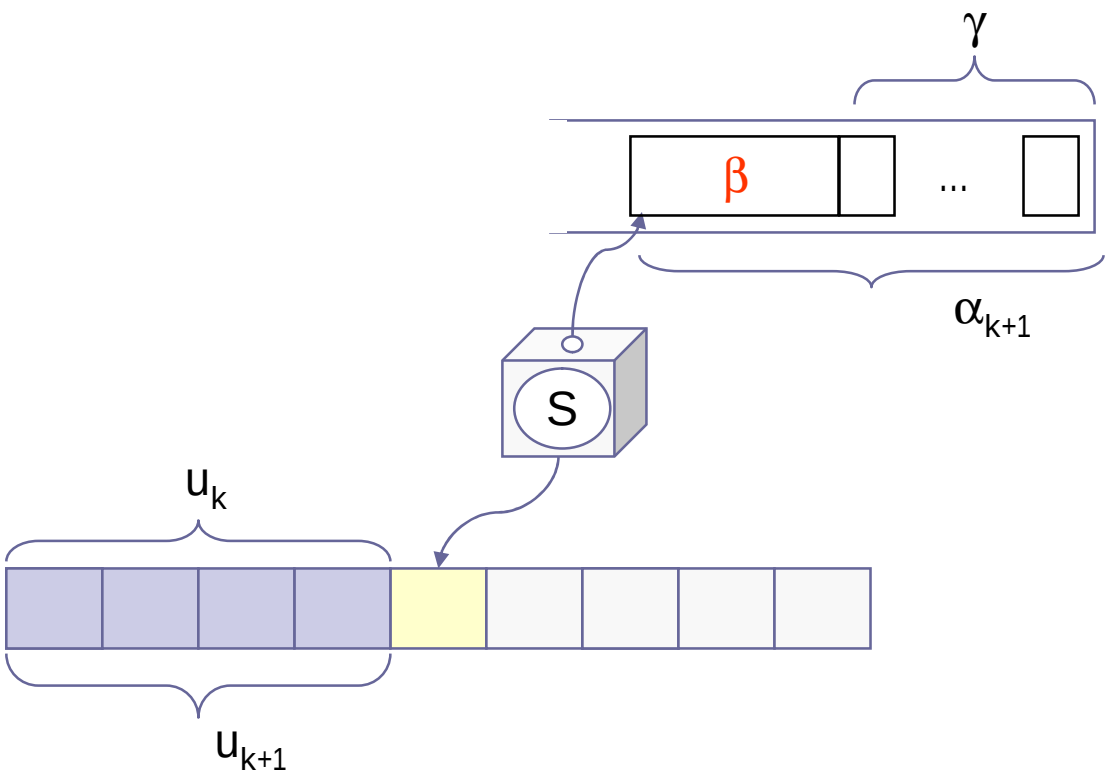
M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Linksherleitung:

$S = u_0 \alpha_0$
$\Rightarrow \ldots$
$\Rightarrow u_k \alpha_k$
$= u_k B \gamma$
$\Rightarrow u_k \beta \gamma$
$= u_{k+1} \alpha_{k+1}$
$\Rightarrow \ldots$
$\Rightarrow w$



Endsituation

M(G)

$\delta(S, a, a) = \varepsilon$ für alle $a \in T$

$\delta(S, \varepsilon, B) = \{ \beta \mid B \rightarrow \beta \in P \}$

Linksherleitung:

$S = u_0 \alpha_0$

$\Rightarrow \ldots$

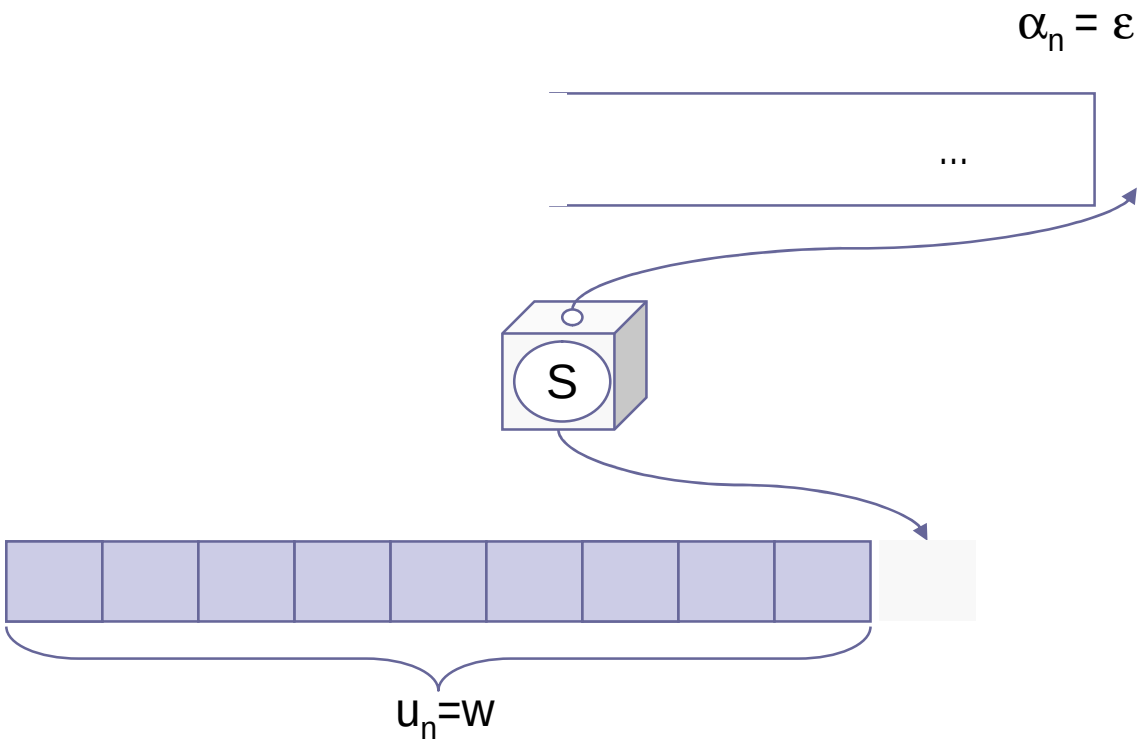
$\Rightarrow u_k \alpha_k$

$\Rightarrow$

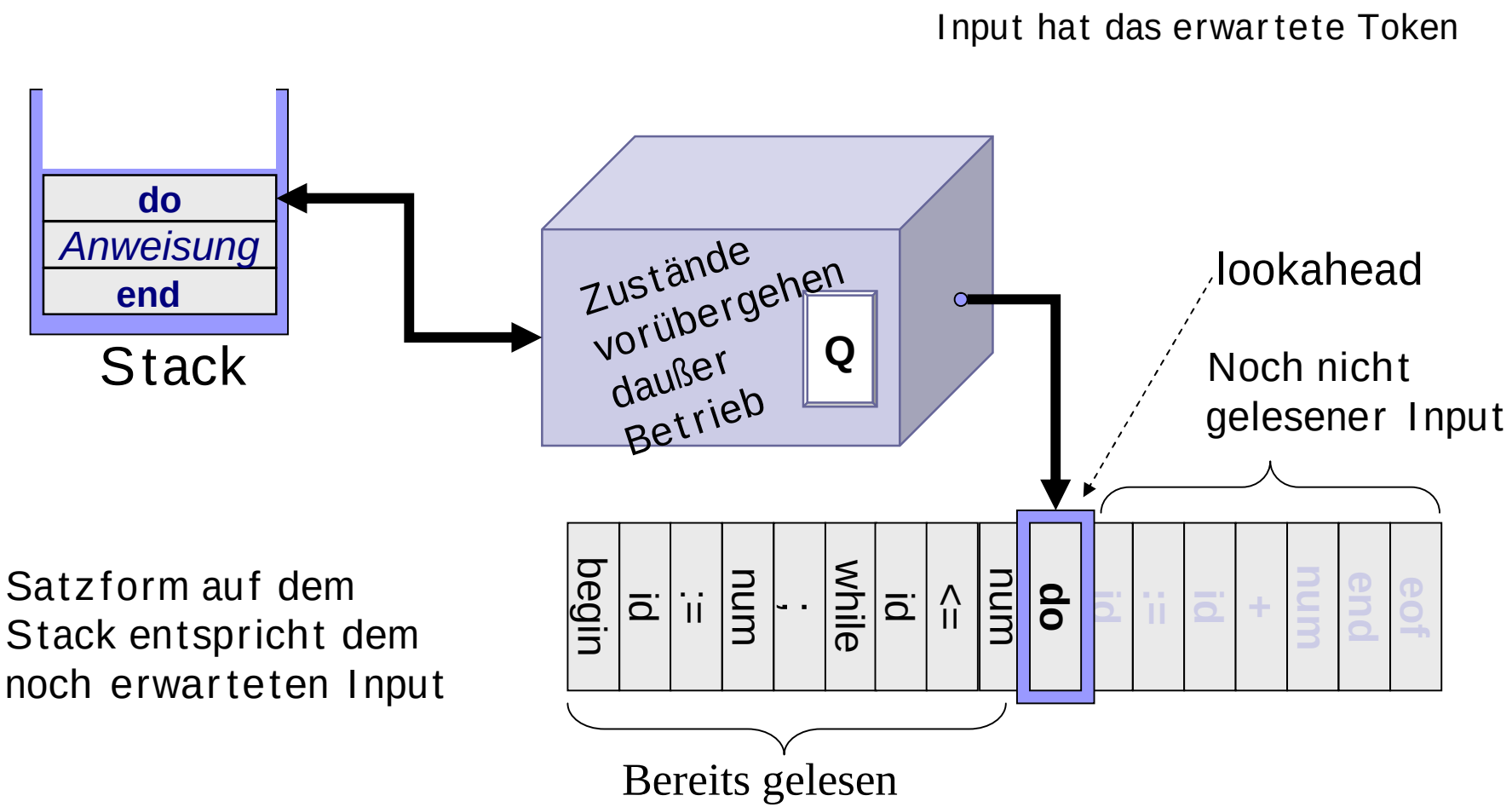
$u_{k+1} \alpha_{k+1}$

$\Rightarrow \ldots$

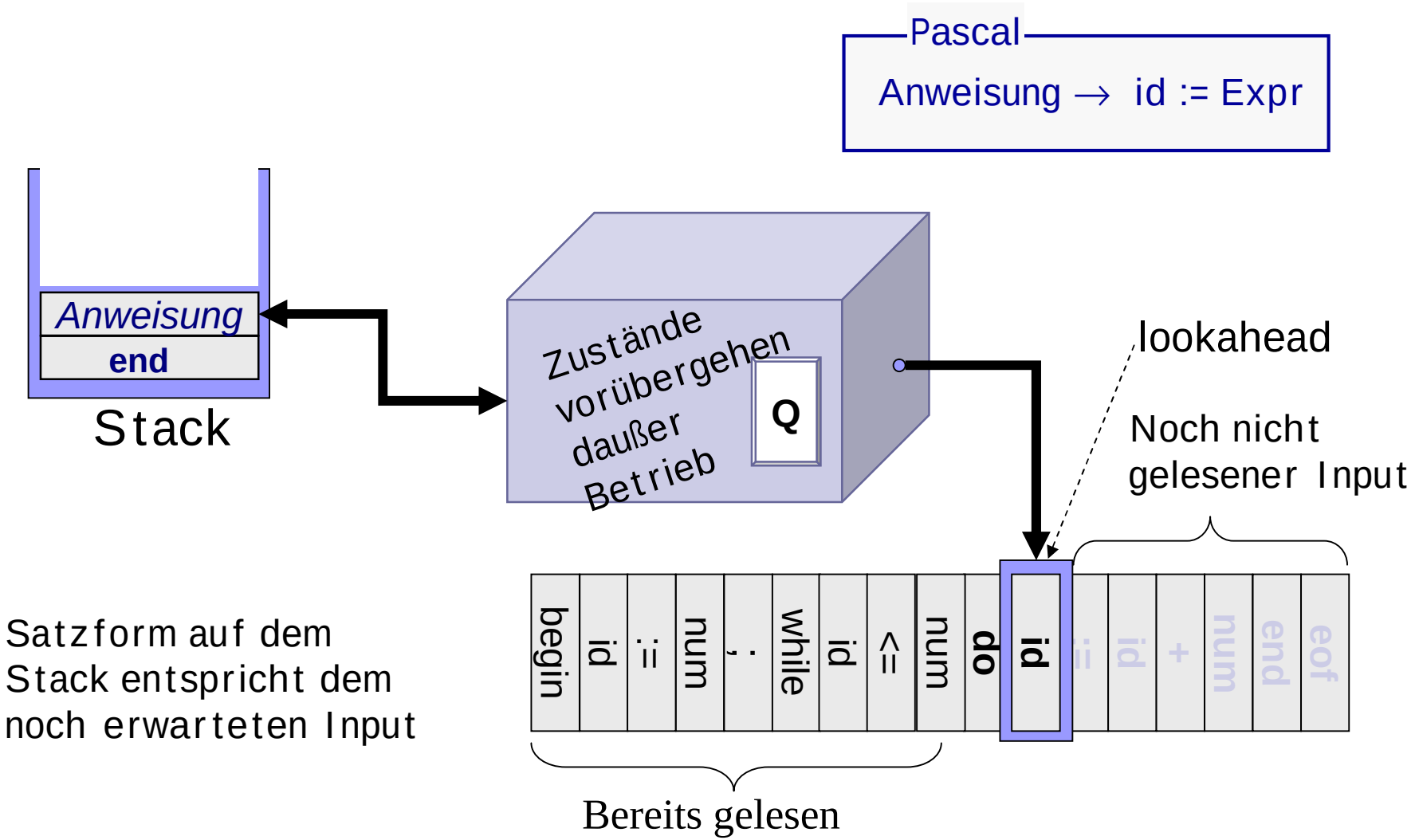
$\Rightarrow u_n = w$



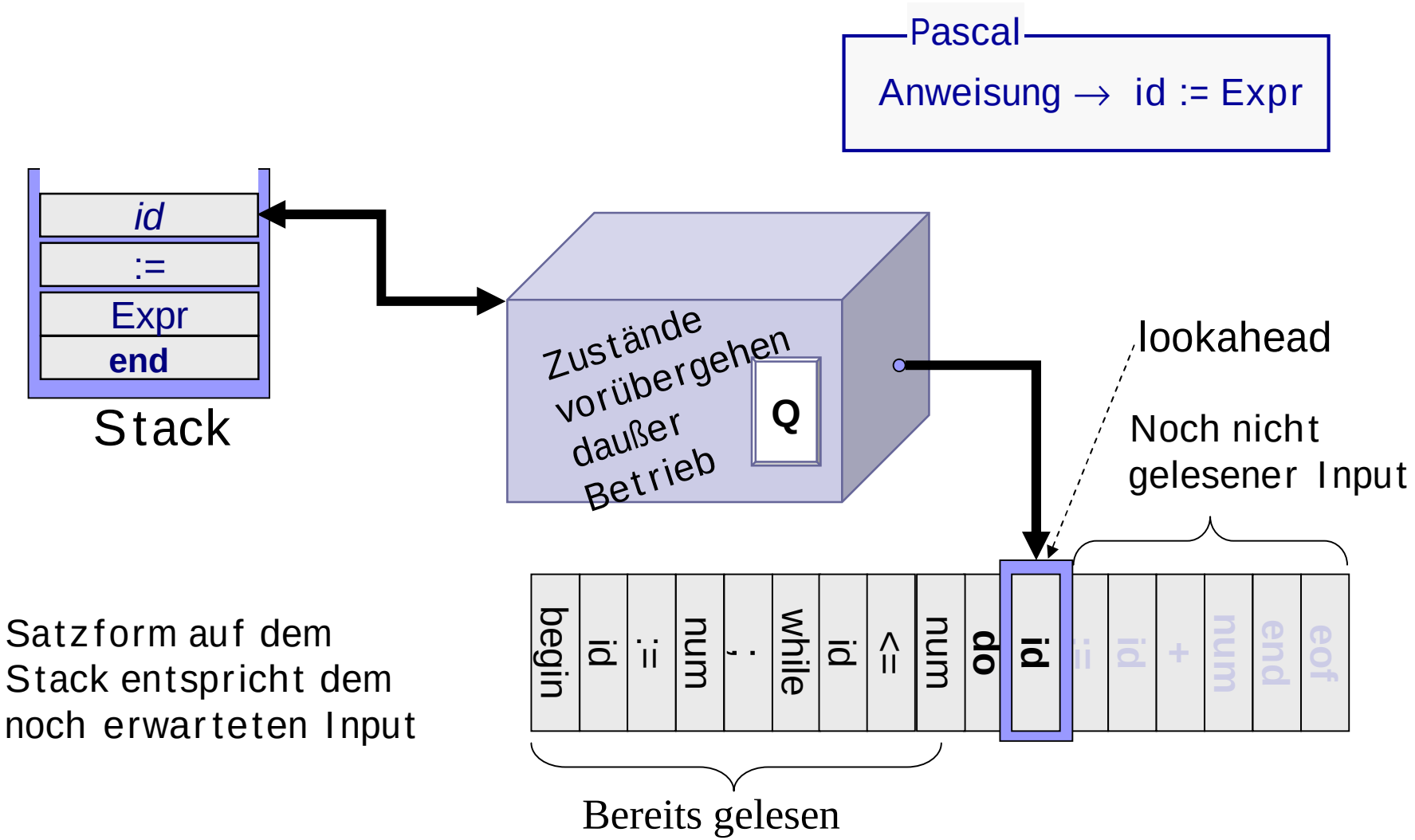
Nichtdeterministischer Stackautomat



Nichtdeterministischer Stackautomat



Nichtdeterministischer Stackautomat



Satzform auf dem Stack entspricht dem noch erwarteten Input

Stackmaschinen und kontextfreie Sprachen

Satz : Für jede kontextfreie Sprache L gibt es eine nicht-deterministische Stackmaschine M, so daß L = L(M) .

Ein Algorithmus, der überprüft, ob ein Wort w in der Sprache ist, insbesondere auch wie es hergeleitet werden kann, nennt man einen

Parser

Der obige Satz hat auch eine Umkehrung:

Ztas : Für Stackmaschine M gibt es eine CF-Grammatik G so dass L(M)=L(G) ..

Da dieser aber für die Praxis (bisher) ohne Relevanz ist, nehmen wir ihn ohne Beweis zur Kenntnis.

Inhalt

1. Grammatiken und Sprachen
   - Kontextfreie Grammatiken
   - Herleitungen, Linksherleitungen
   - Sprachen zu einer Grammatik
   - Äquivalenz
   - Chomsky-Normalform
   - Wortproblem, CYK
   - Pumping Lemma für CF-Sprachen

2. Stackautomaten
   - Definitionen und Beispiele
   - Konfigurationen, Läufe,
   - Sprache eines Stackautomaten
   - Parser

3. Parser
   - Mehrdeutigkeit
   - Bottom Up, Top Down
   - Recursive descent Parser
   - First, Follow
   - Semantische Aktionen

4. Shift-Reduce Parser
   - Konflikte
   - LR(0)-Zustände
   - Automat zur Grammatik
   - Shift-Reduce, Reduce-Reduce Konflikte
   - Präzedenz und Assoziativität
   - lex und yacc
   - Arbeitsweise eines Compilers

Mehrdeutigkeit

Besitzt L(G) ein Wort w, zu dem es mehrere Herleitungsbäume gibt, so heißt G mehrdeutig.

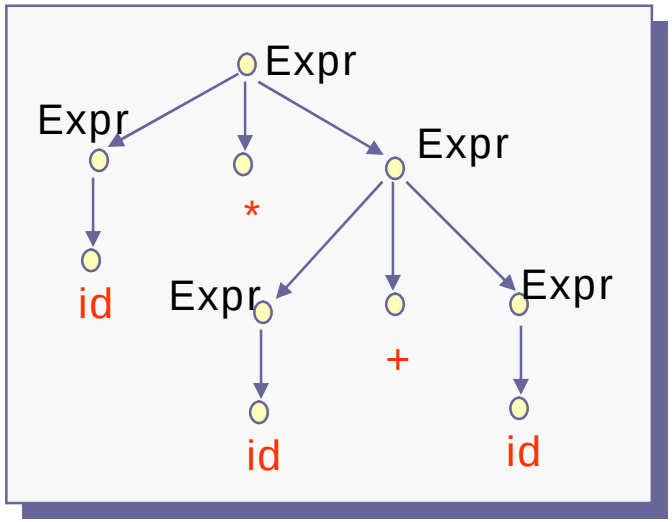
E

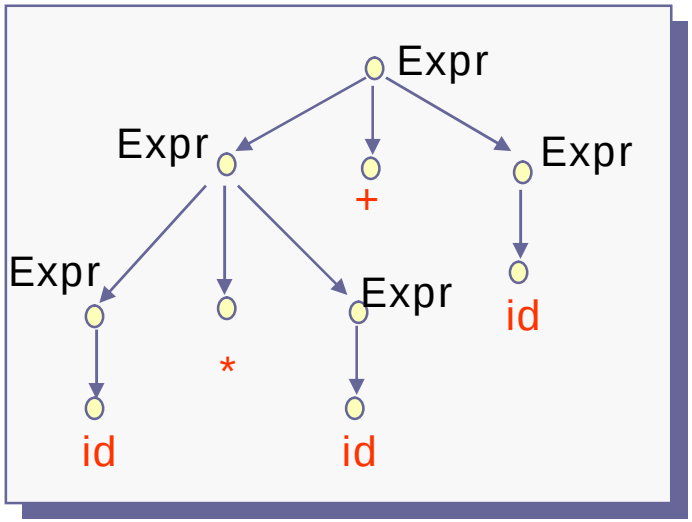
Expr → Expr + Expr
| Expr * Expr
| (Expr)
| id

Diese Grammatik ist mehrdeutig, denn das Wort

id * id + id

hat zwei Herleitungsbäume
(äqu.: zwei Linksherleitungen)





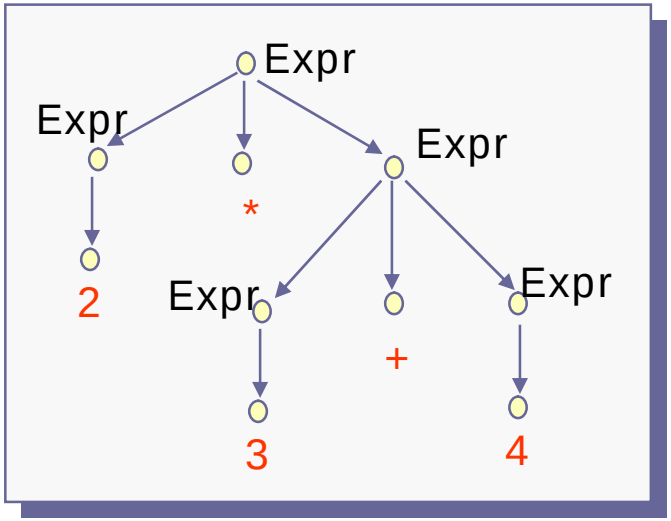
Bedeutung - Semantik

G. Frege

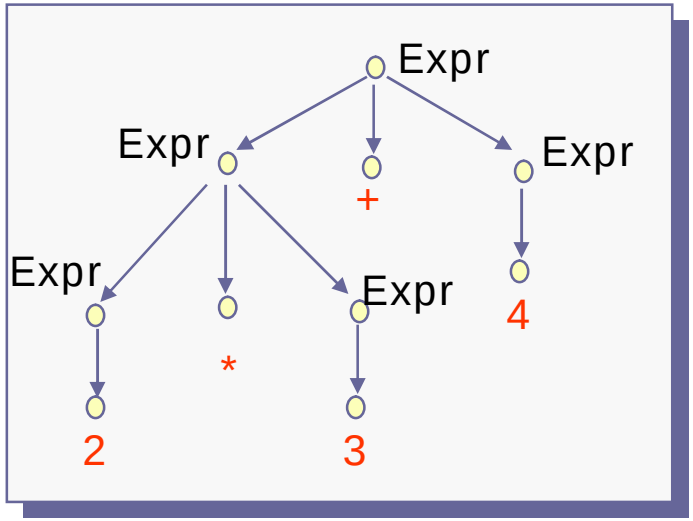
Denotationelles Prinzip (Frege): Die Bedeutung des Ganzen ist eine Funktion der Bedeutung der Teile.

Herleitungsbaum gibt Aufbau und damit die intendierte Bedeutung des Satzes eindeutig wieder. Lineare Schreibweise als String ist oft nicht eindeutig:

2 * 3 + 4 = ?



14



10

Strategien zur Mehrdeutigkeit

Im Sinne einer einfachen Sprachbeschreibung nimmt man Mehrdeutigkeiten einer Grammatik oft in Kauf. Beim Parsen gibt es verschiedene Strategien mit der Mehrdeutigkeit umzugehen :

G_1

Expr → Expr + Expr
| num

Mehrdeutigkeit unerheblich,
da Addition assoziativ.

G_2

Expr → Expr - Expr
| num

Mehrdeutigkeit wesentlich,
da Subtraktion nicht assoziativ.

G'_2

Expr → Expr - num
| num

Umgewandelt in
eindeutige Grammatik.

Strategien zur Mehrdeutigkeit

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} + \text{Expr} \\
&\mid \text{Expr} * \text{Expr} \\
&\mid (\text{Expr}) \\
&\mid \text{num}
\end{aligned}
$$

Mehrdeutigkeit erheblich,
z.B. 7 + 3 * 4 hat zwei Bäume.

Neue syntaktische Kategorien (Nonterminale) für jede Präzedenzstufe

G'_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} + \text{Term} \\
&\mid \text{Term} \\
\text{Term} &\rightarrow \text{Term} * \text{Factor} \\
&\mid \text{Factor} \\
\text{Factor} &\rightarrow (\text{Expr}) \\
&\mid \text{num}
\end{aligned}
$$

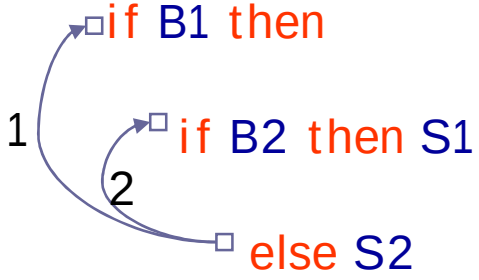
Äquivalente eindeutige Grammatik .

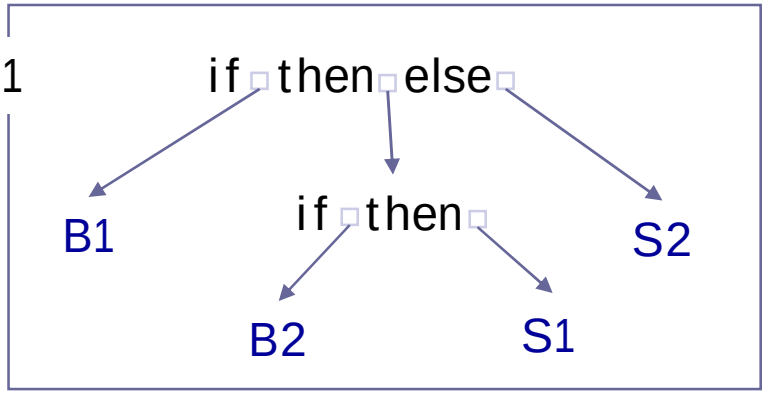
Strategien zur Mehrdeutigkeit

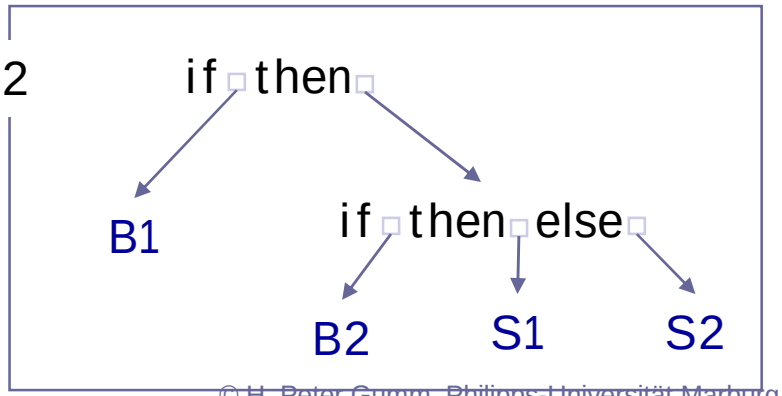
G_4

Stmt → if Bexpr then Stmt else Stmt
| if Bexpr then Stmt
| ...

Mehrdeutig:







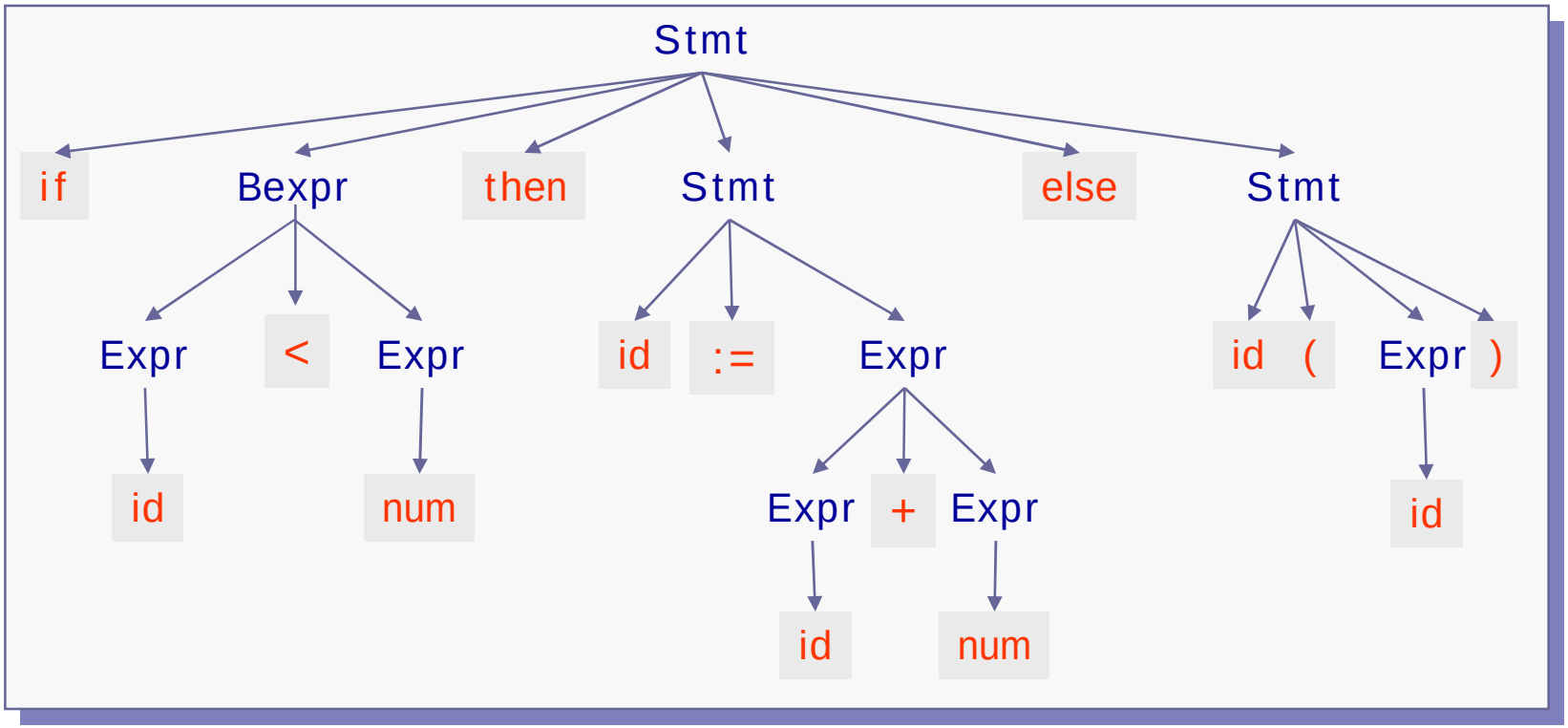
Verabredung :

Jedes else ergänzt das
letzte unergänzte then

Parsen

Parsen bedeutet, zu entscheiden, ob $w \in L(G)$ und ggf. einen Herleitungsbaum zu finden.

End-Ergebnis:



Input:

```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (  z  )
```

Parserstrategien

- Bottom Up
  - Baum wächst von den Blättern zur Wurzel
- Top Down
  - Baum wächst von der Wurzel zu den Blättern

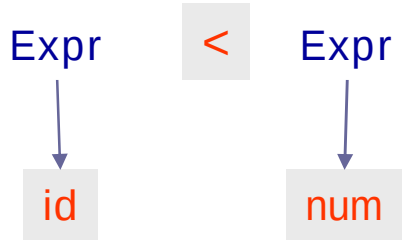
Beispiel anhand nebenstehender Grammatik :

G

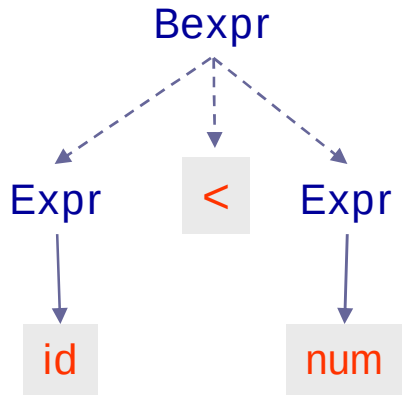
```
Stmt  → if Bexpr then Stmt else Stmt
      | id := Expr
      | id (Expr)

Expr  → Expr + Expr
      | id
      | num

Bexpr → Expr < Expr
      | true
```

Bottom Up Strategie

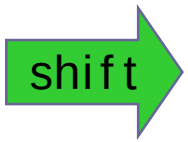
- Bottom Up
  - Baum wächst von den Blättern zur Wurzel
  - jederzeit ein Wald (mehrere Bäume) vorhanden
- Zwei Aktionen
  - Shift
    - Lese nächstes Token im Input
    - mache es zur Wurzel eines neuen Baumes
  - Reduce
    - Die Wurzeln der letzten Bäume bilden die rechte Seite einer Regel A → w_1 w_2 ….w_n
    - Füge sie unter dem neuen Knoten A zusammen



Bexpr → Expr < Expr
| true

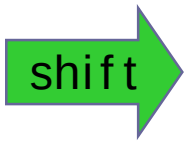


Bottom Up



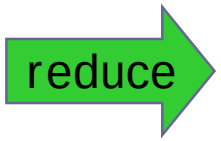
Bottom Up

```
if
```




```
if   x
```

Bottom Up



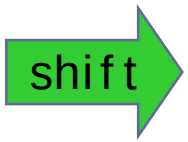
Expr → Expr + Expr
Expr → id
Expr → num

if

id

if	x	<	

Bottom Up



if	x	<	

Bottom Up

Bottom Up

reduce

Expr → Expr + Expr
Expr → id
Expr → num

if

Expr <

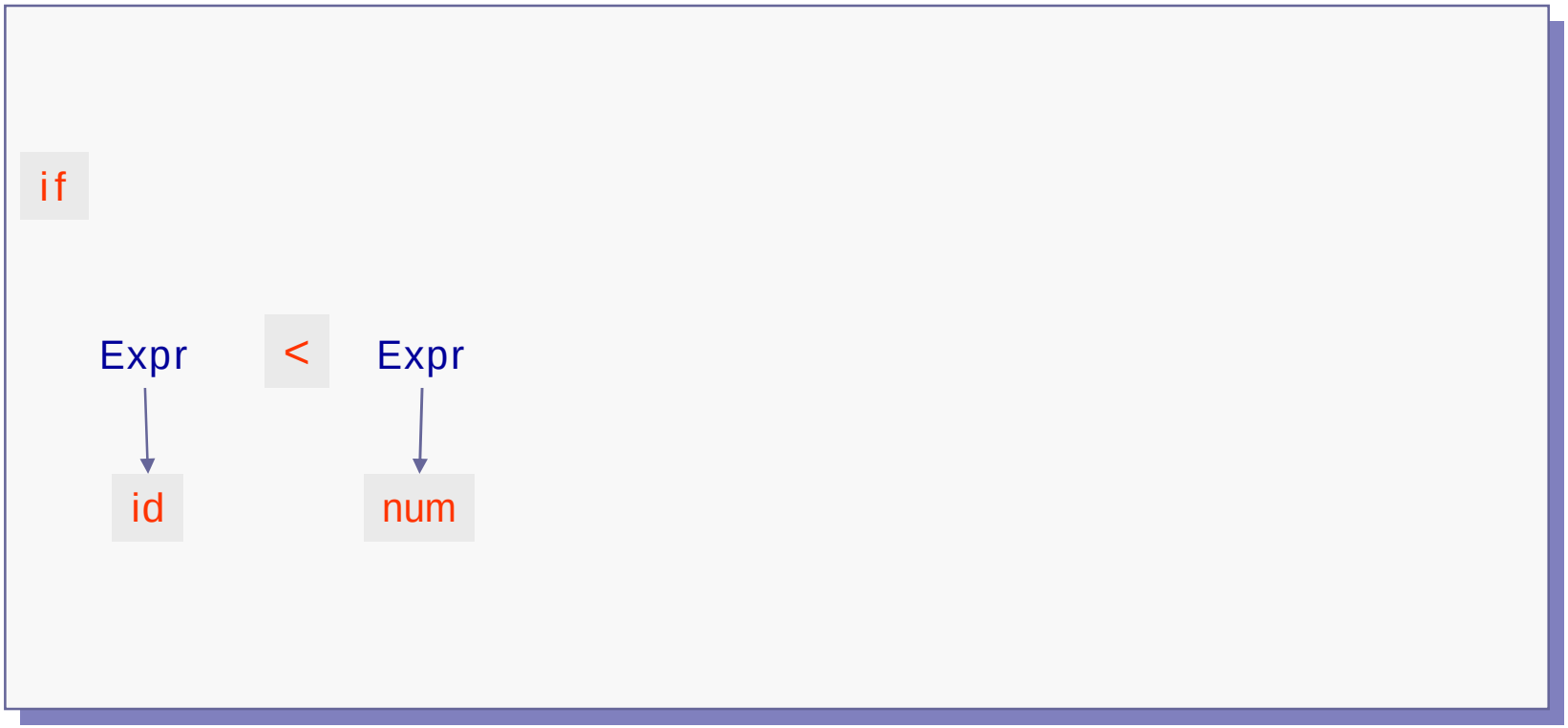
id num

if	x	<	0	then	

Bottom Up

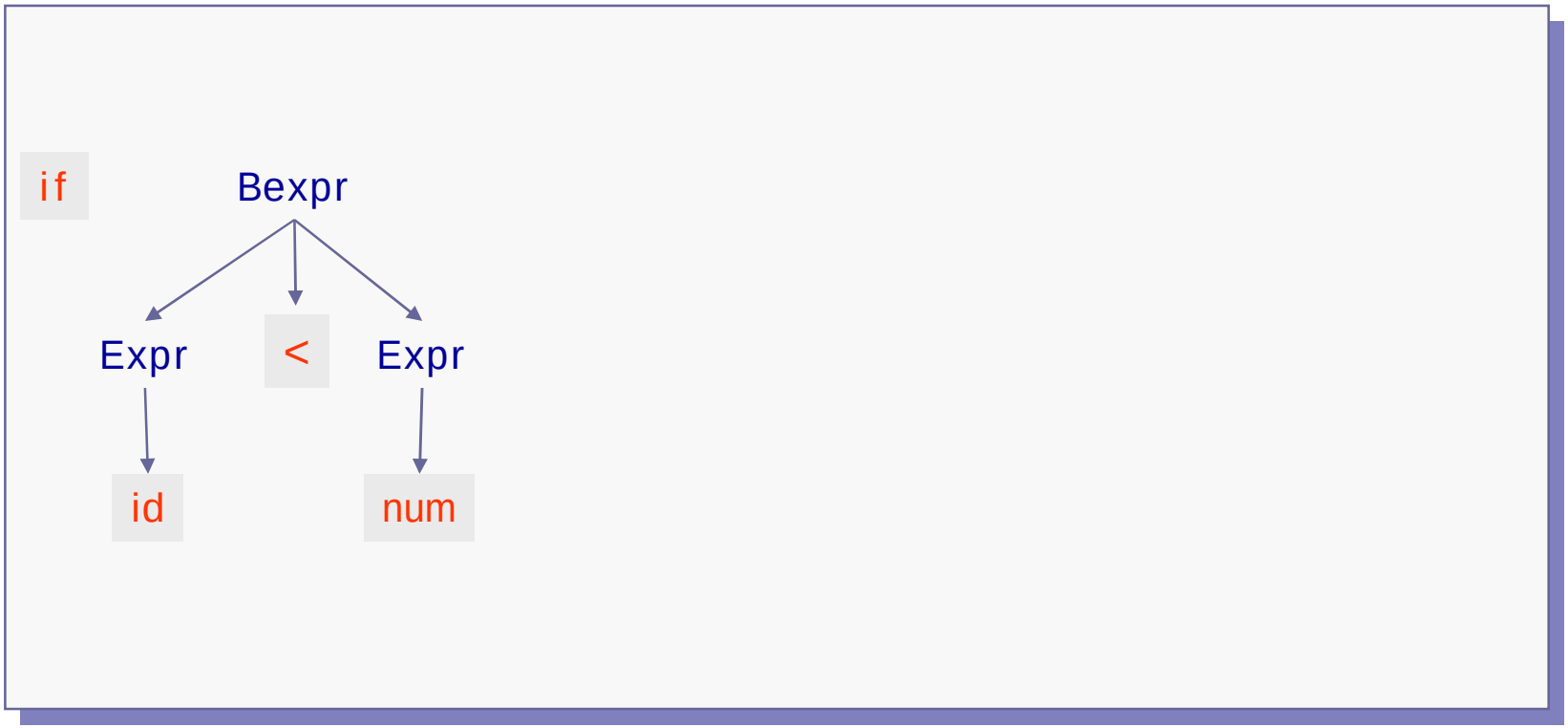
reduce

Bexpr → Expr < Expr
Bexpr → true



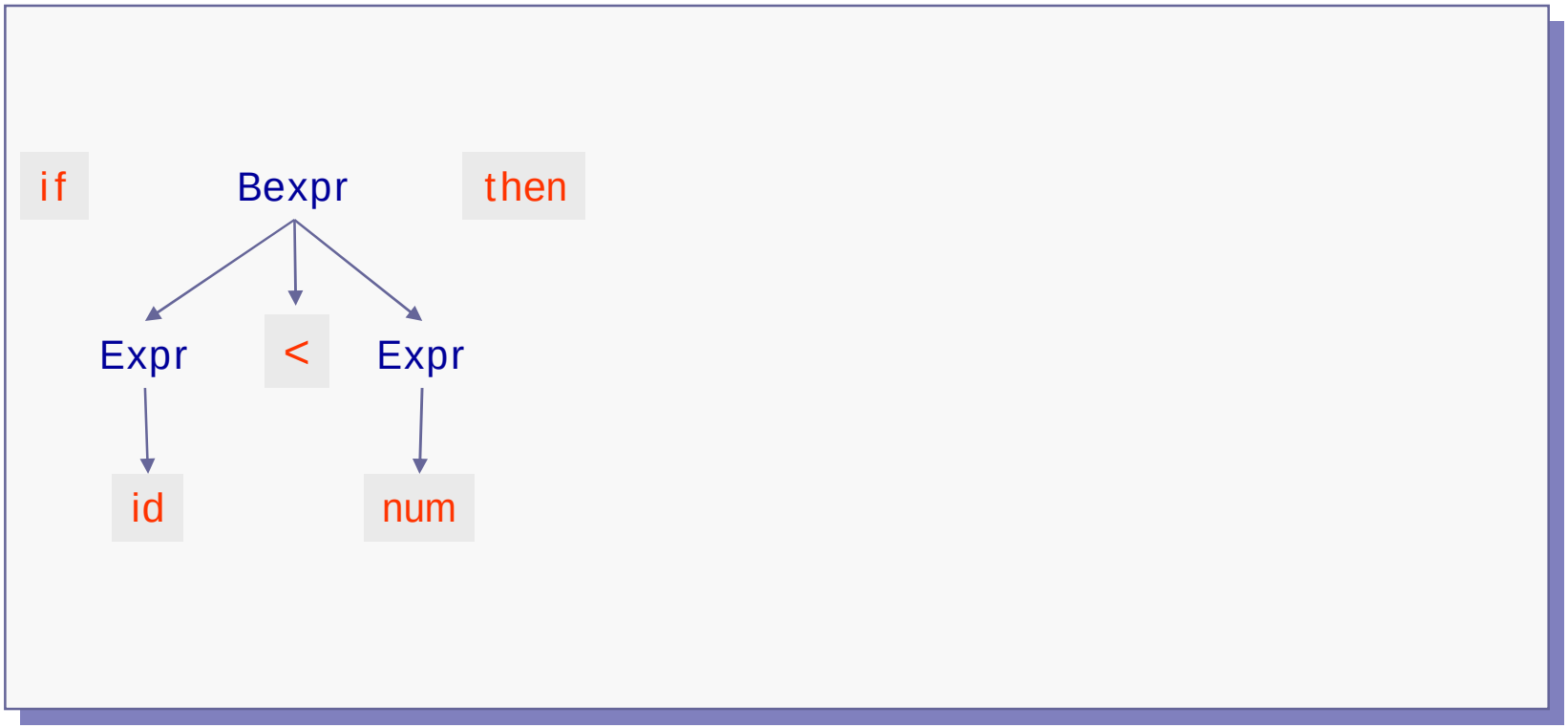
if x < 0 then	

Bottom Up



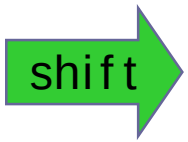

```
if   x    <   0  then
```

Bottom Up



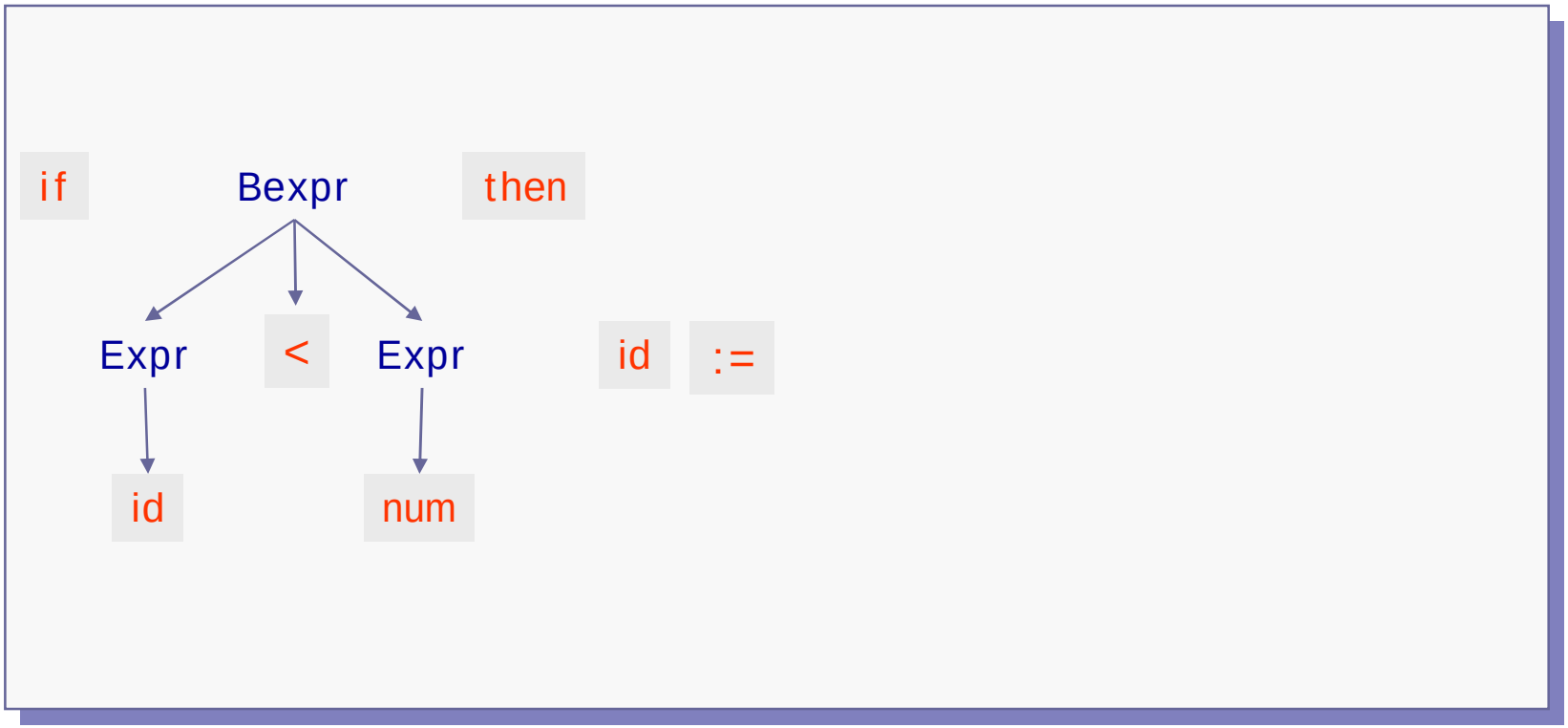

```
if   x    <   0  then  y
```

Bottom Up




```
if   x    <   0  then  y  :=
```

Bottom Up



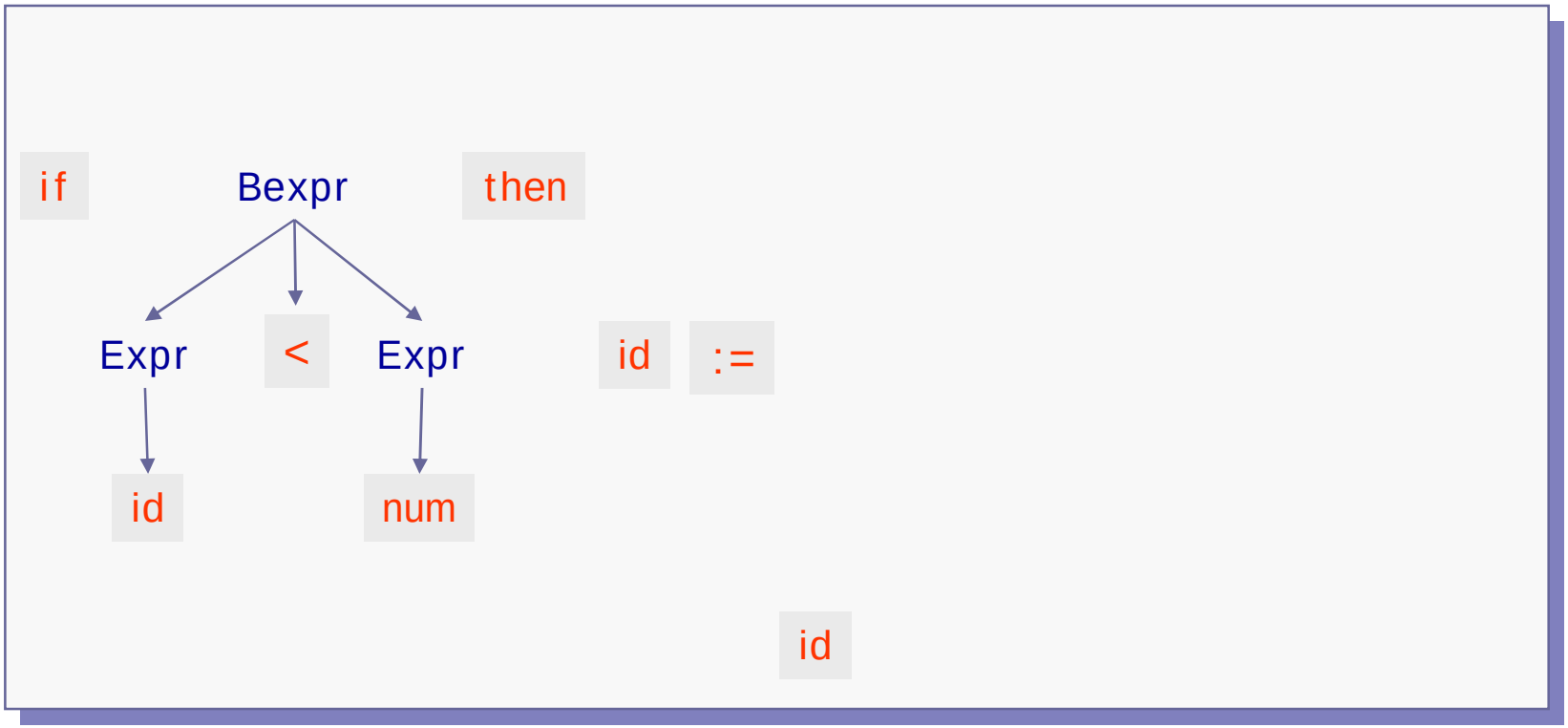
shift

if	x	<	0	then	y	:=	y	

Bottom Up

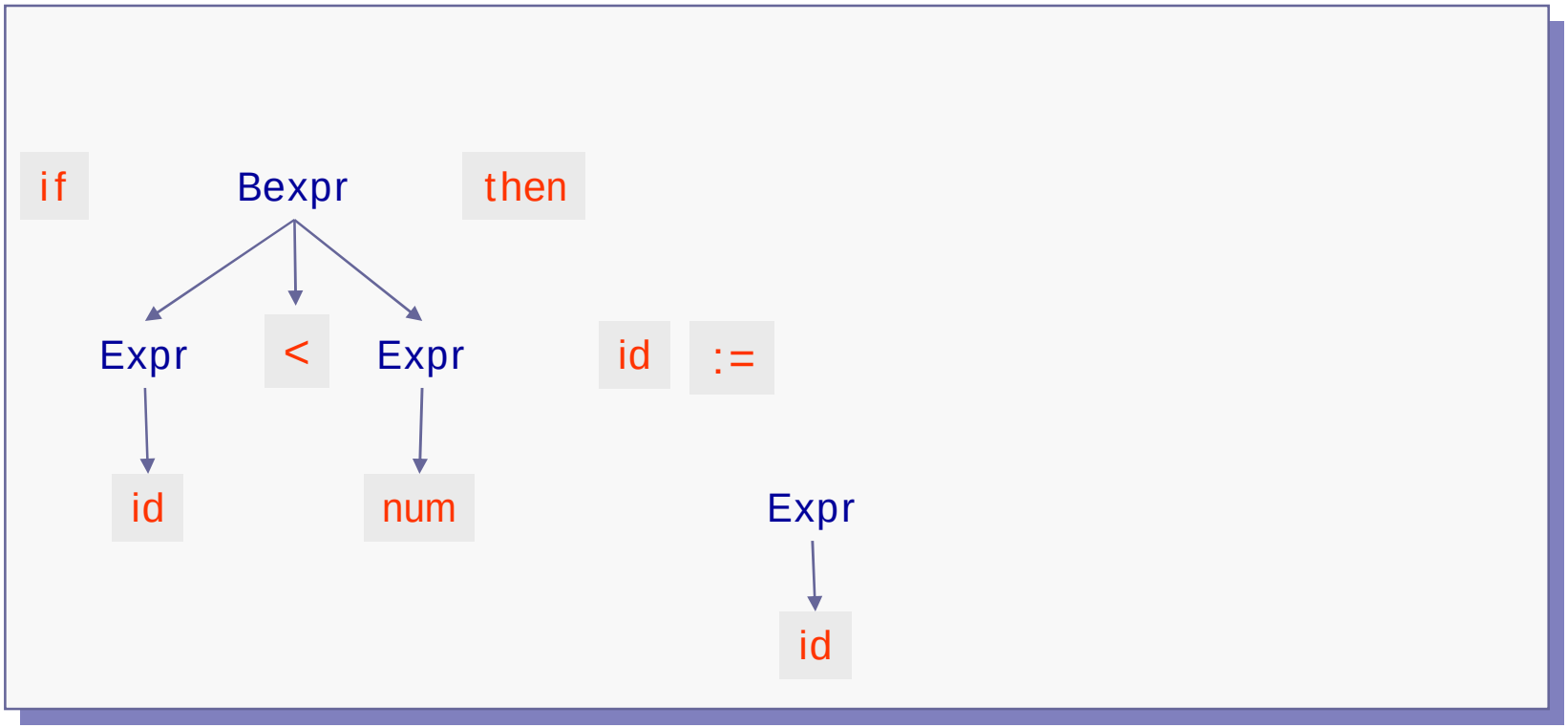
reduce

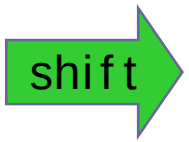
Expr → Expr + Expr
Expr → id
Expr → num



if	x	<	0	then	y	:=	y	+	

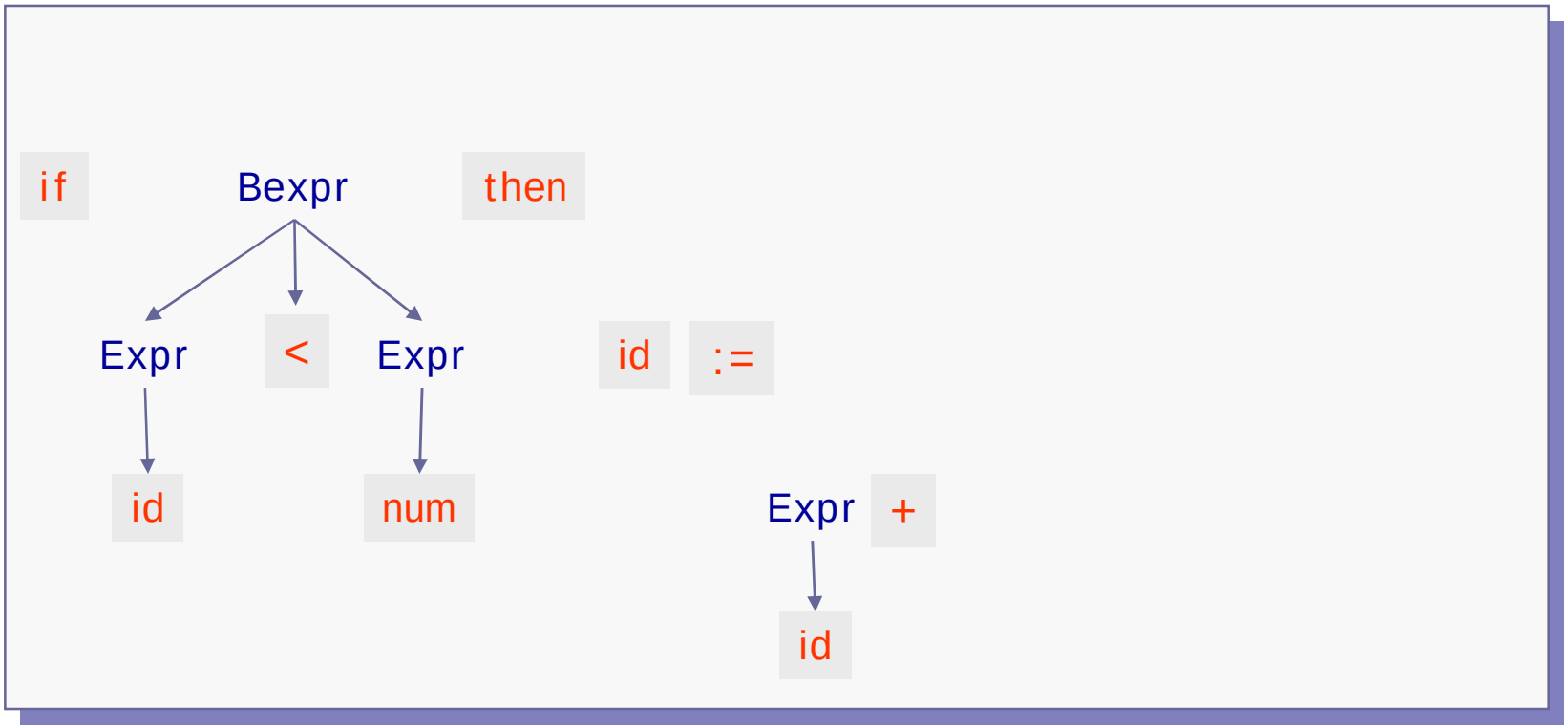
Bottom Up

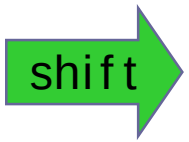





```
if   x    <   0  then  y  :=  y  +
```

Bottom Up



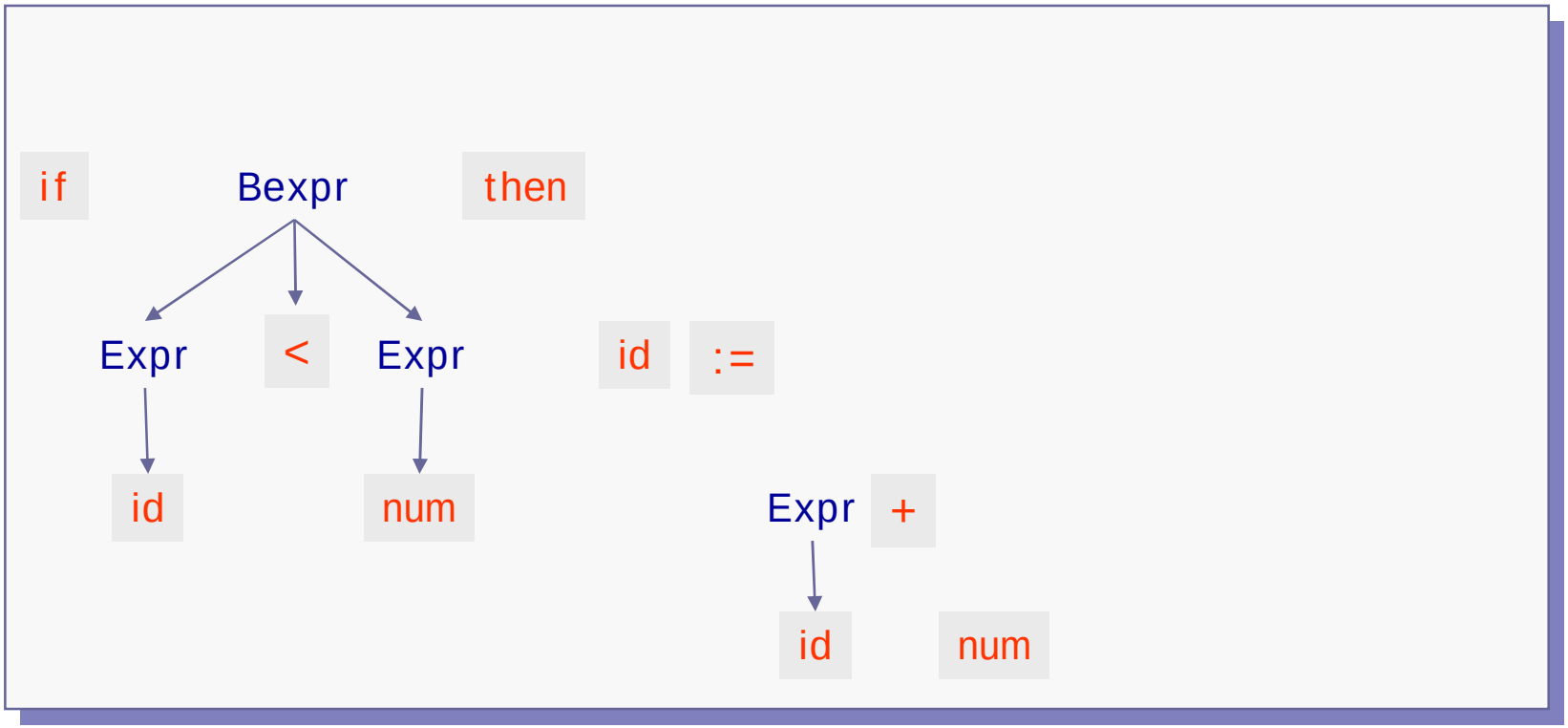



```
if  x  <  0  then  y  :=  y  +  1
```

Bottom Up

reduce

Expr → Expr + Expr
Expr → id
Expr → num

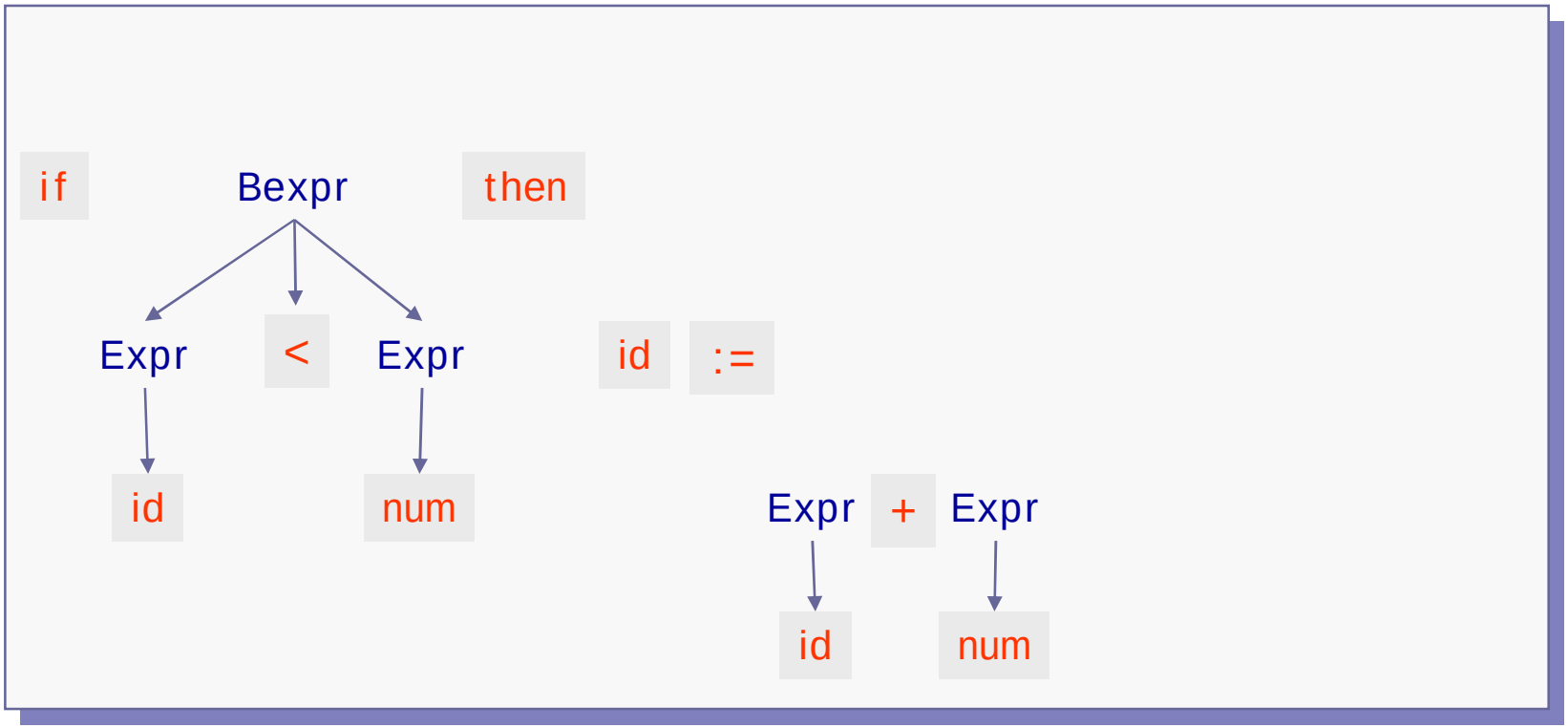


if	x	<	0	then	y	:=	y	+	1	else	

Bottom Up

reduce

Expr → Expr + Expr
Expr → id
Expr → num

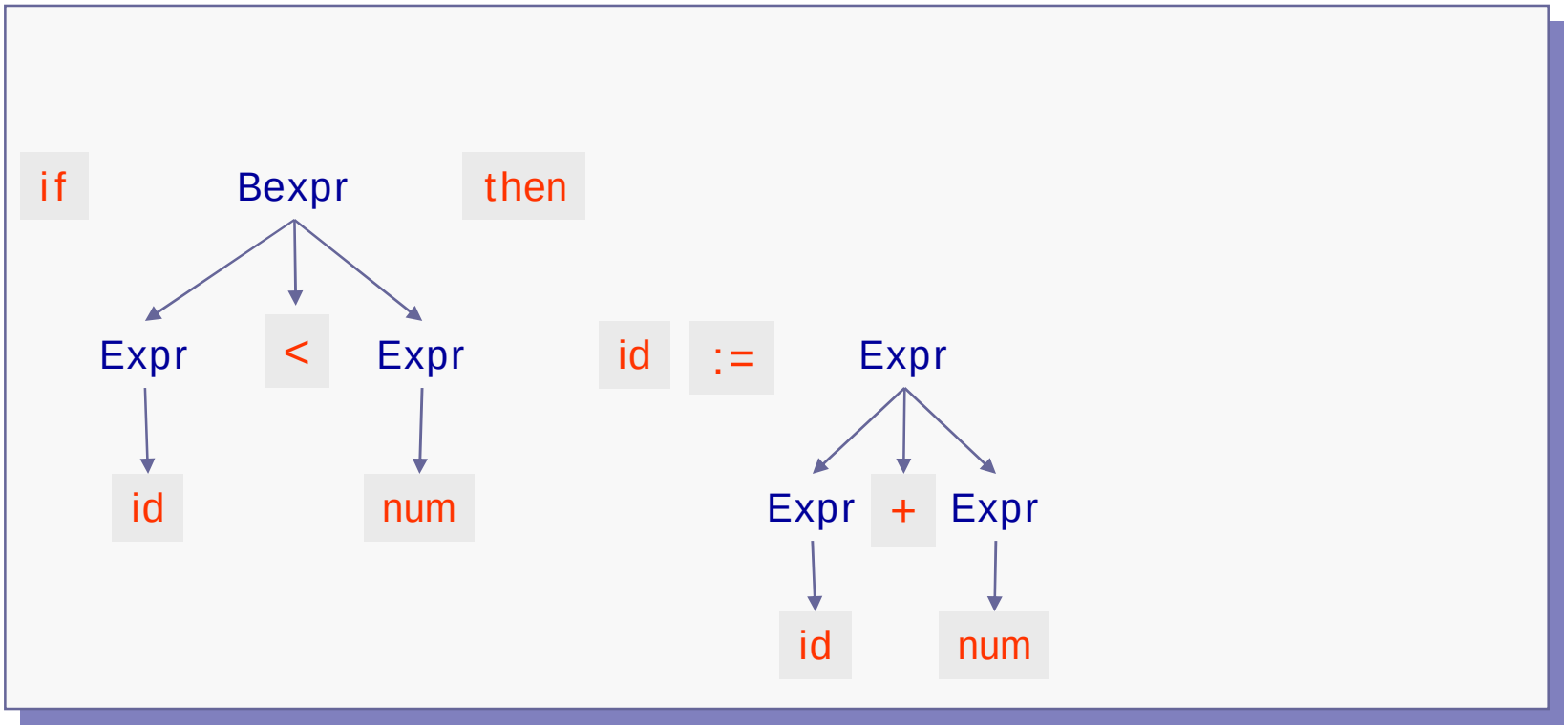


if	x	<	0	then	y	:=	y	+	1	else	

Bottom Up

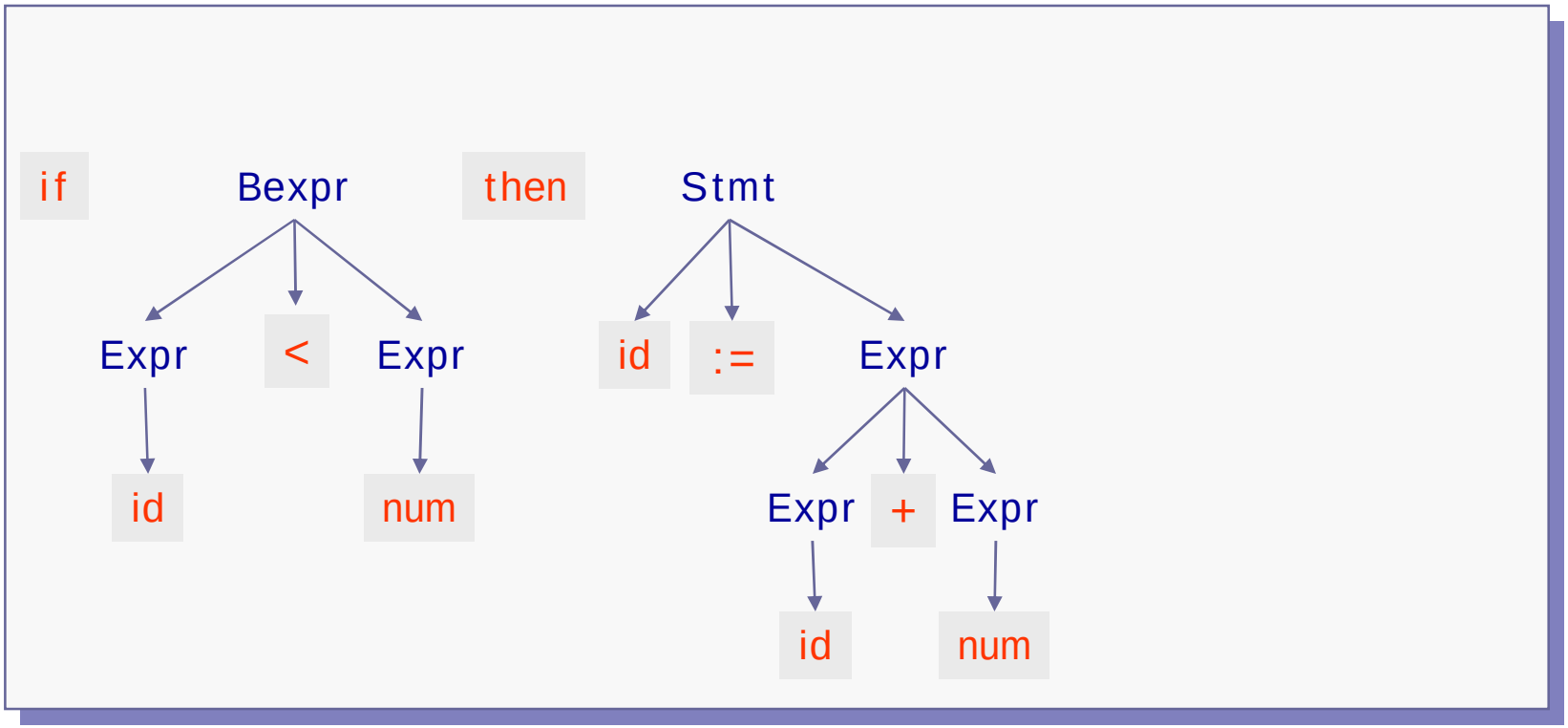
reduce

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)



if	x	<	0	then	y	:=	y	+	1	else	

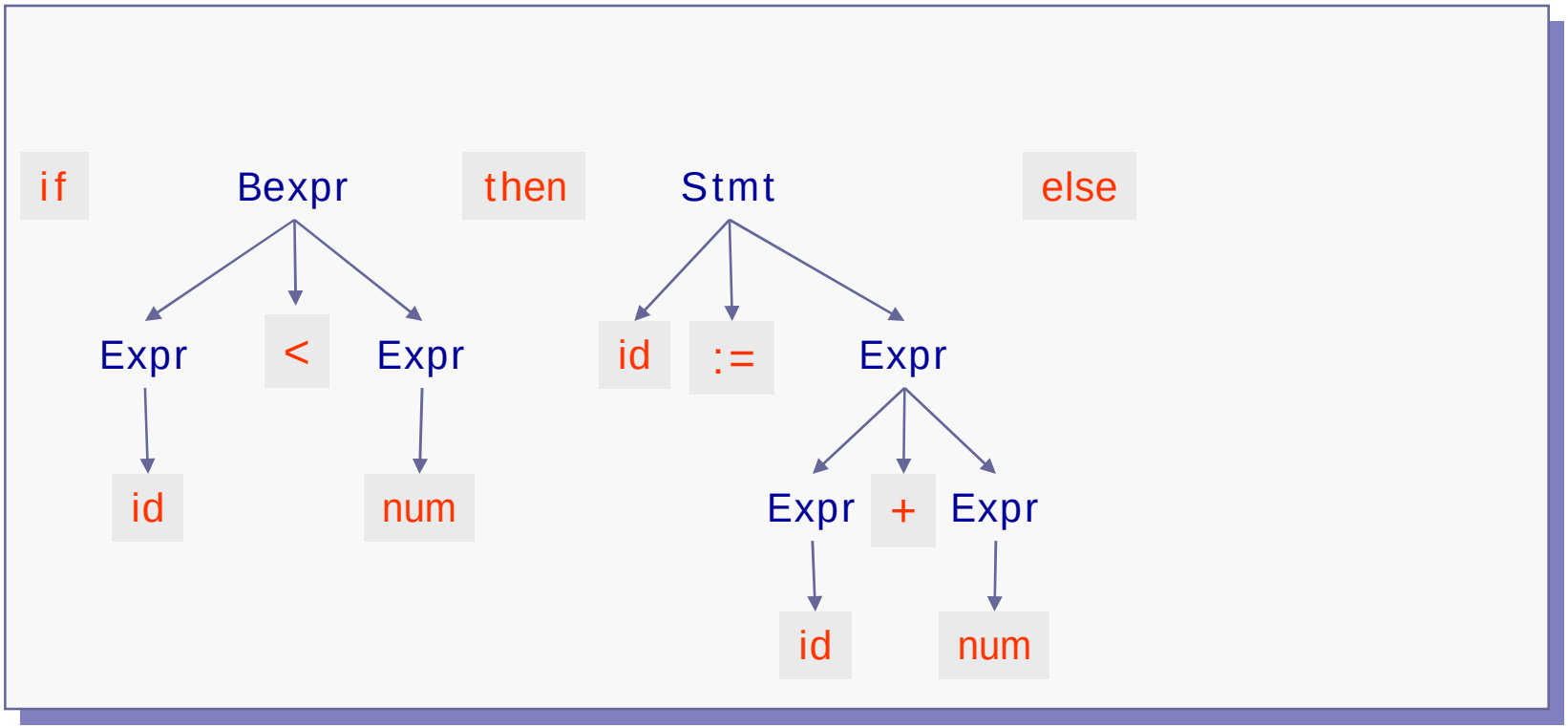
Bottom Up



shift

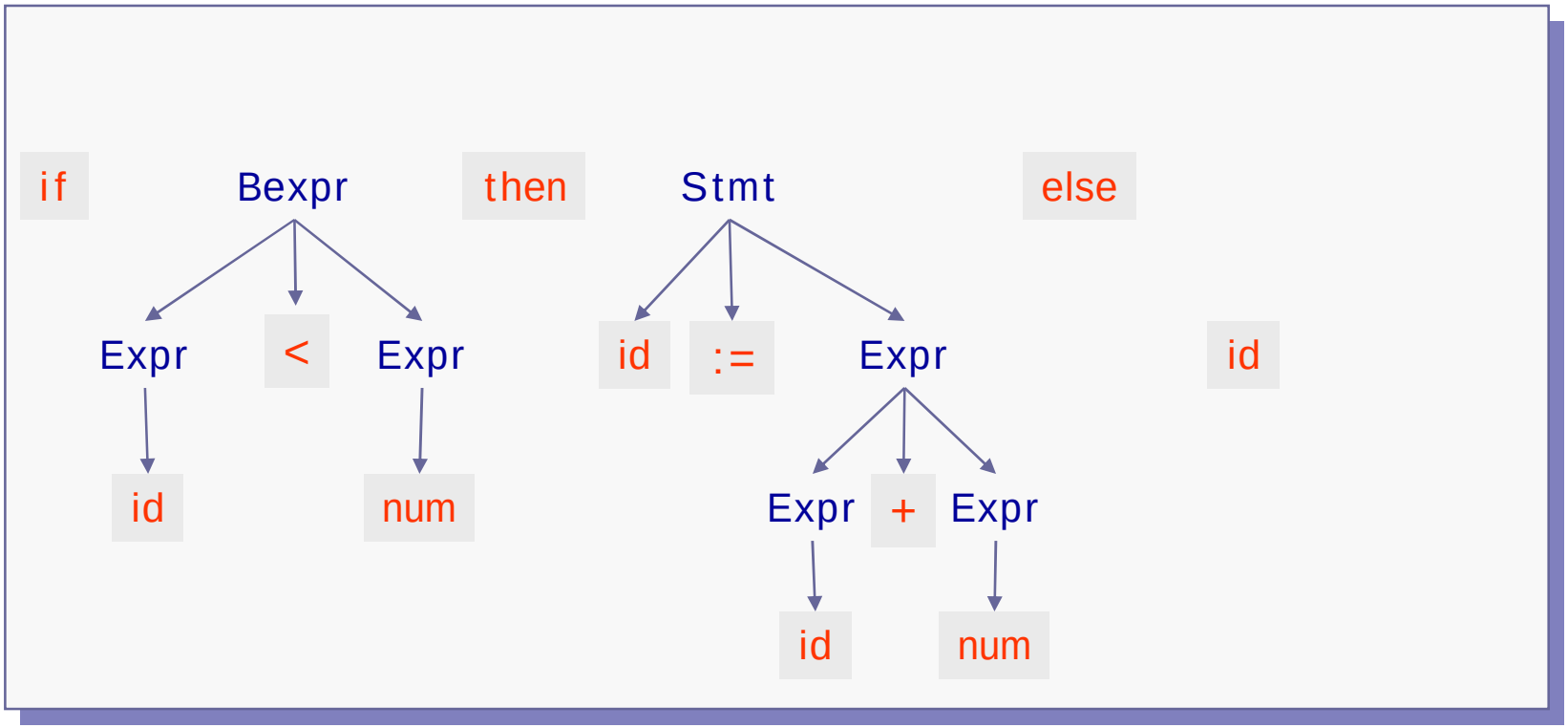
if	x	<	0	then	y	:=	y	+	1	else	

Bottom Up



if	x	<	0	then	y	:=	y	+	1	else	Inc	

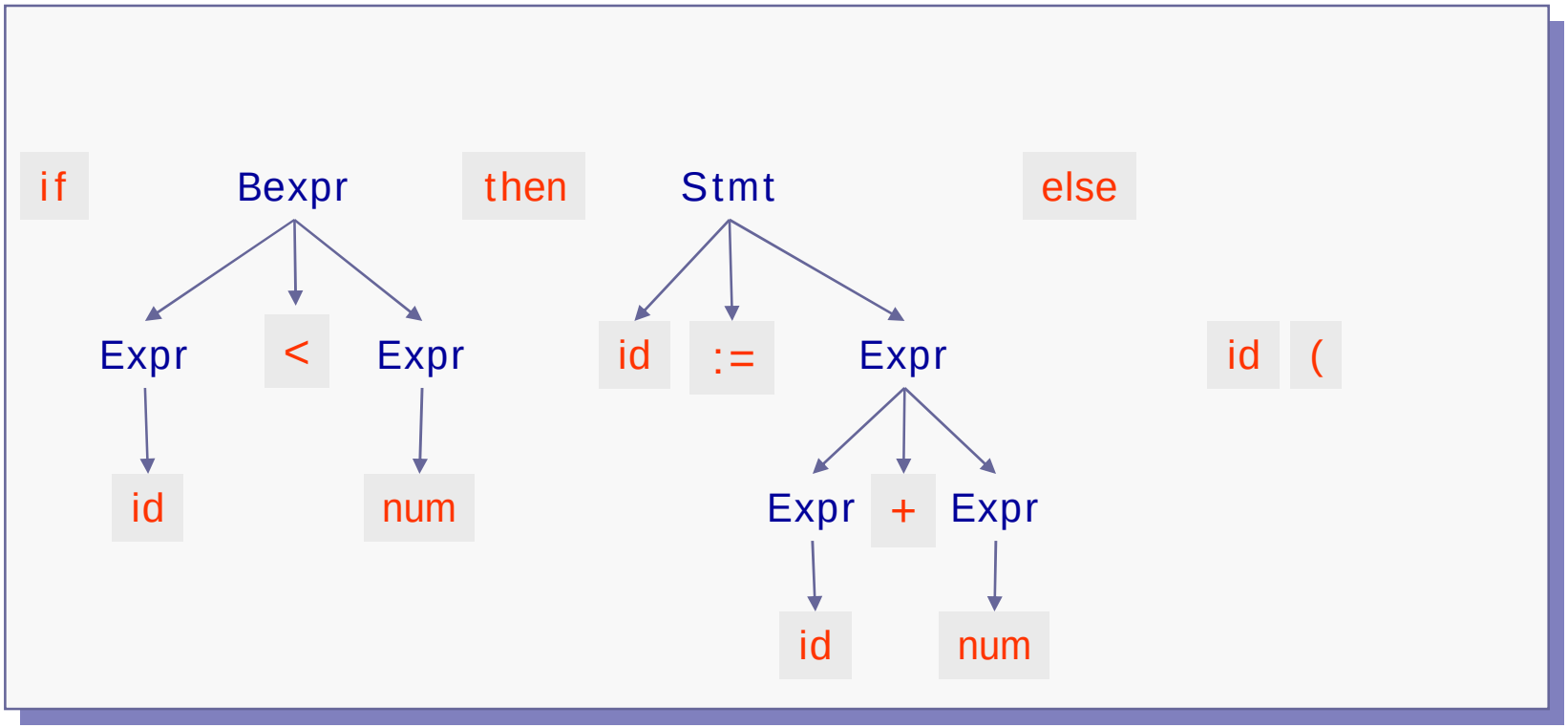
Bottom Up



shift

if	x	<	0	then	y	:=	y	+	1	else	Inc	(	

Bottom Up



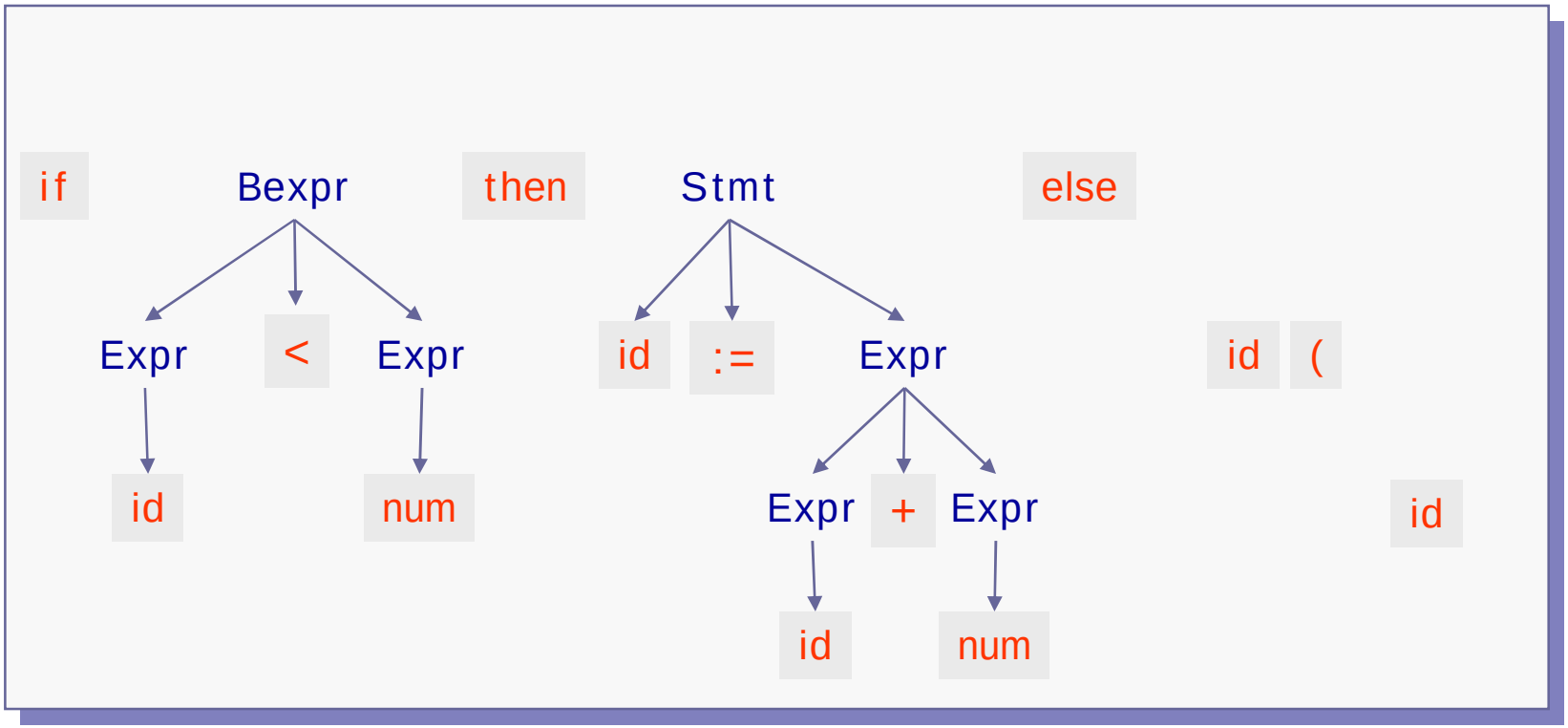
shift

if	x	<	0	then	y	:=	y	+	1	else	Inc	(	z	

Bottom Up

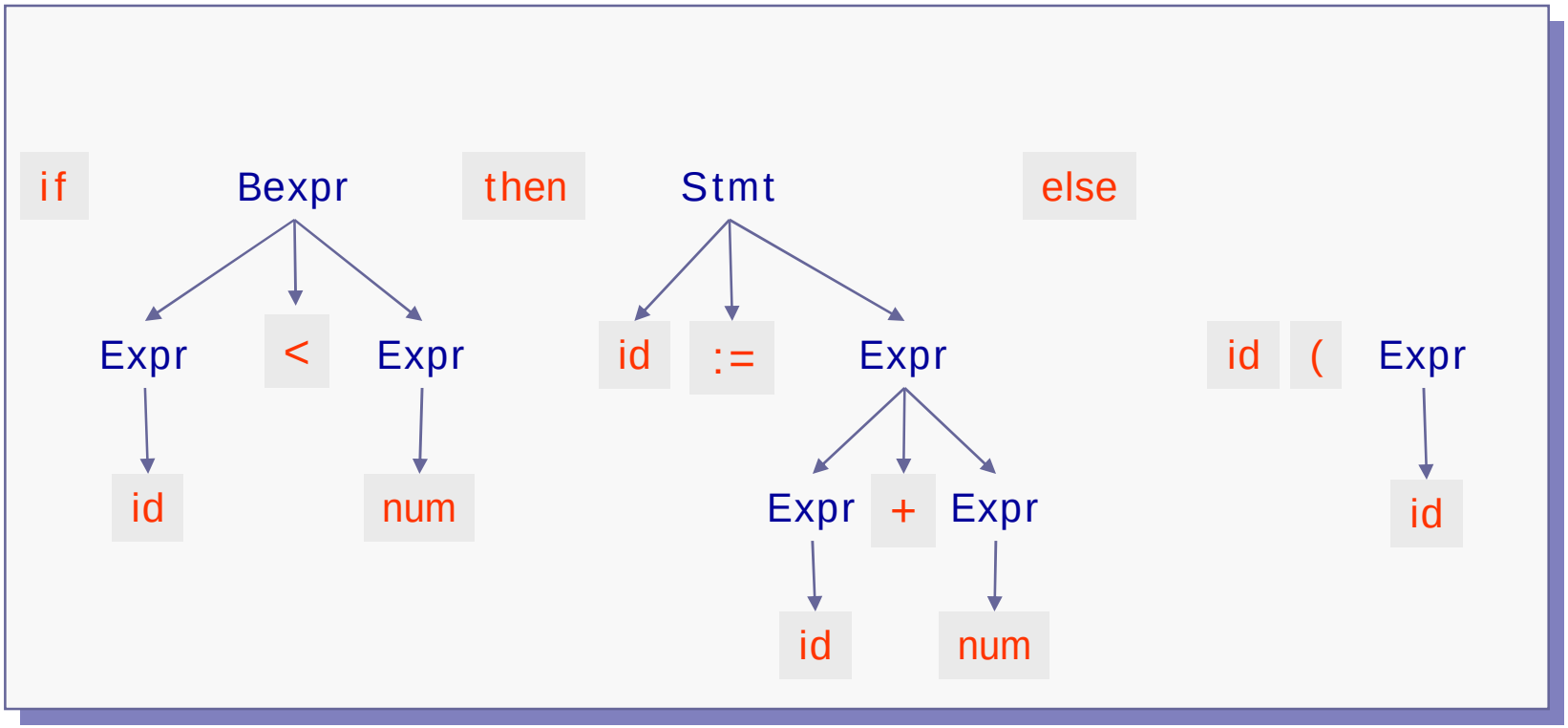
reduce

Expr → Expr + Expr
Expr → id
Expr → num




```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Bottom Up



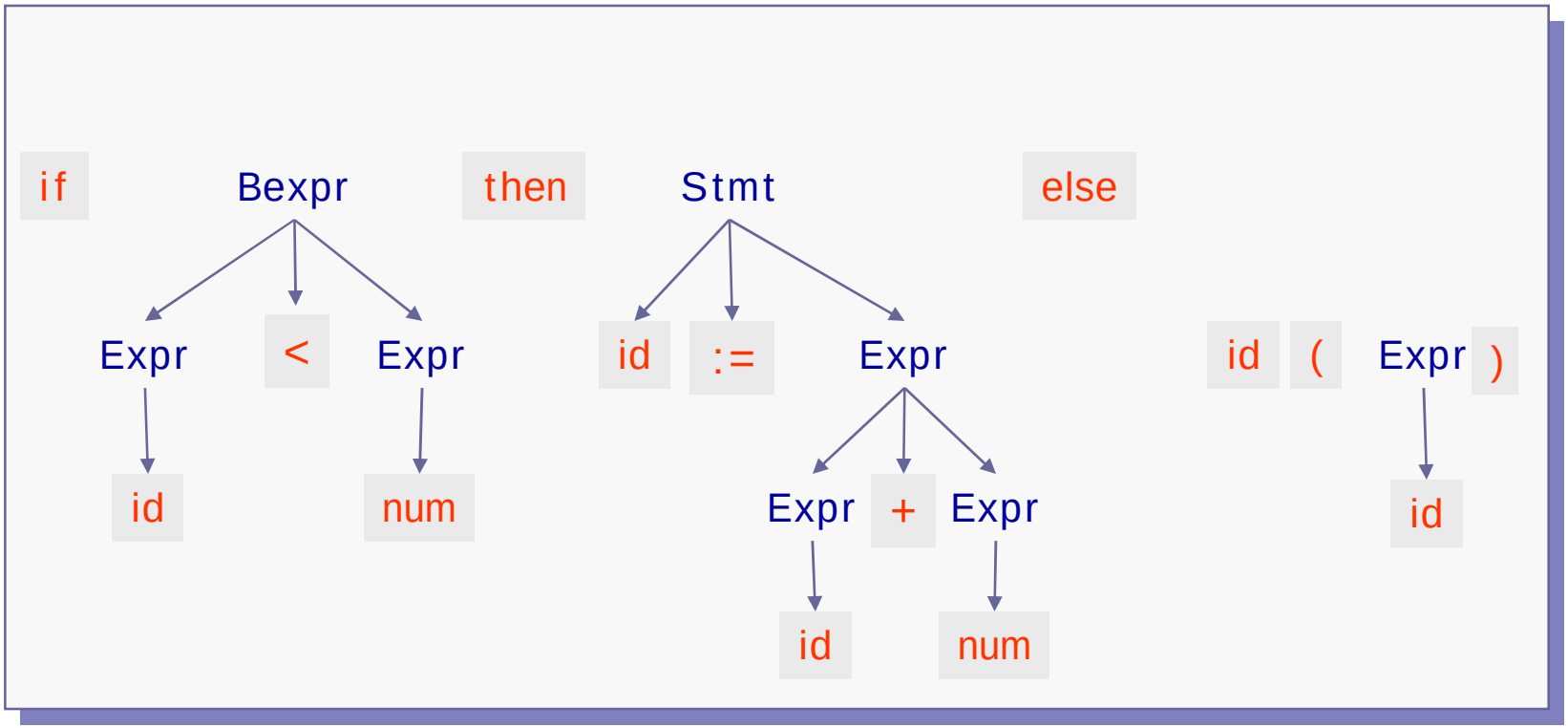
shift

```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Bottom Up

reduce

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)

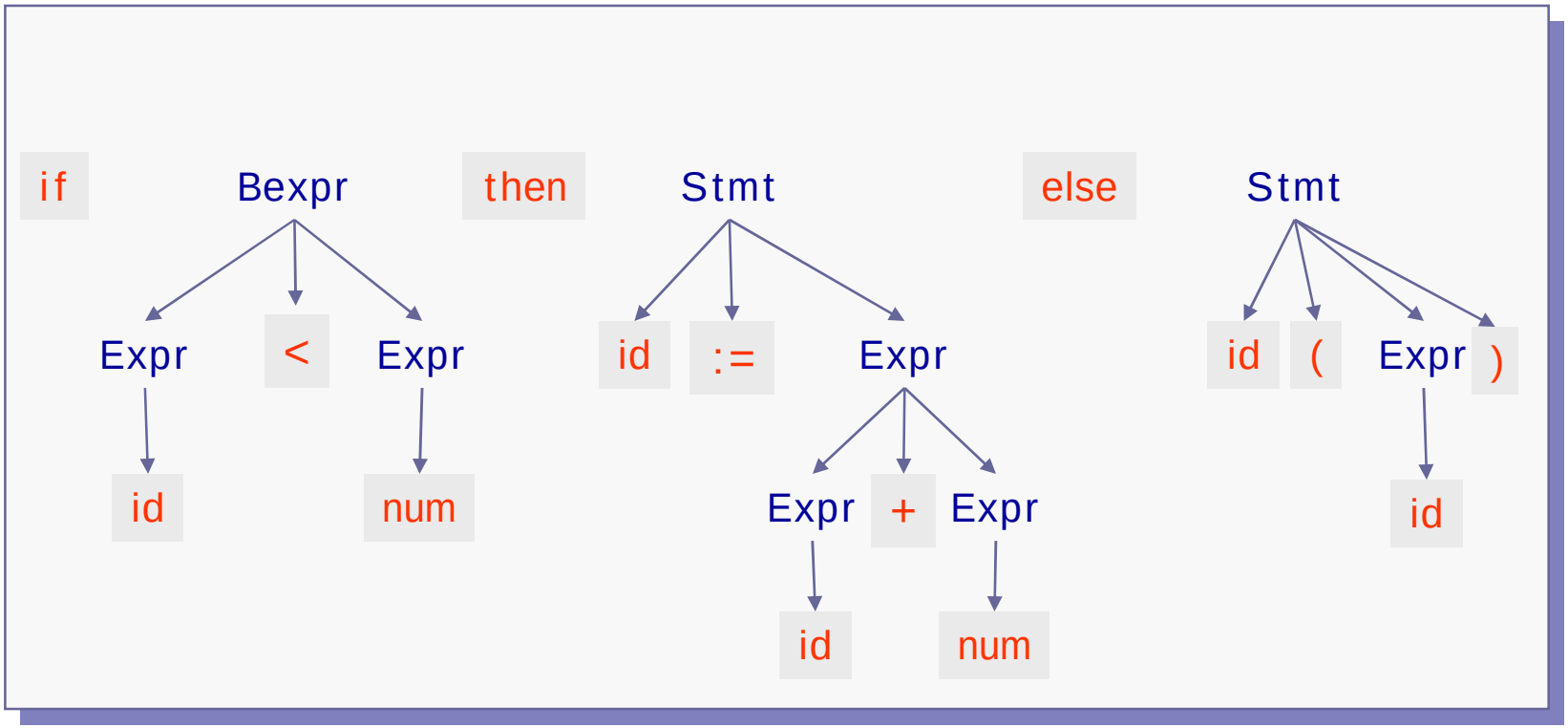



```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Bottom Up

reduce

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)



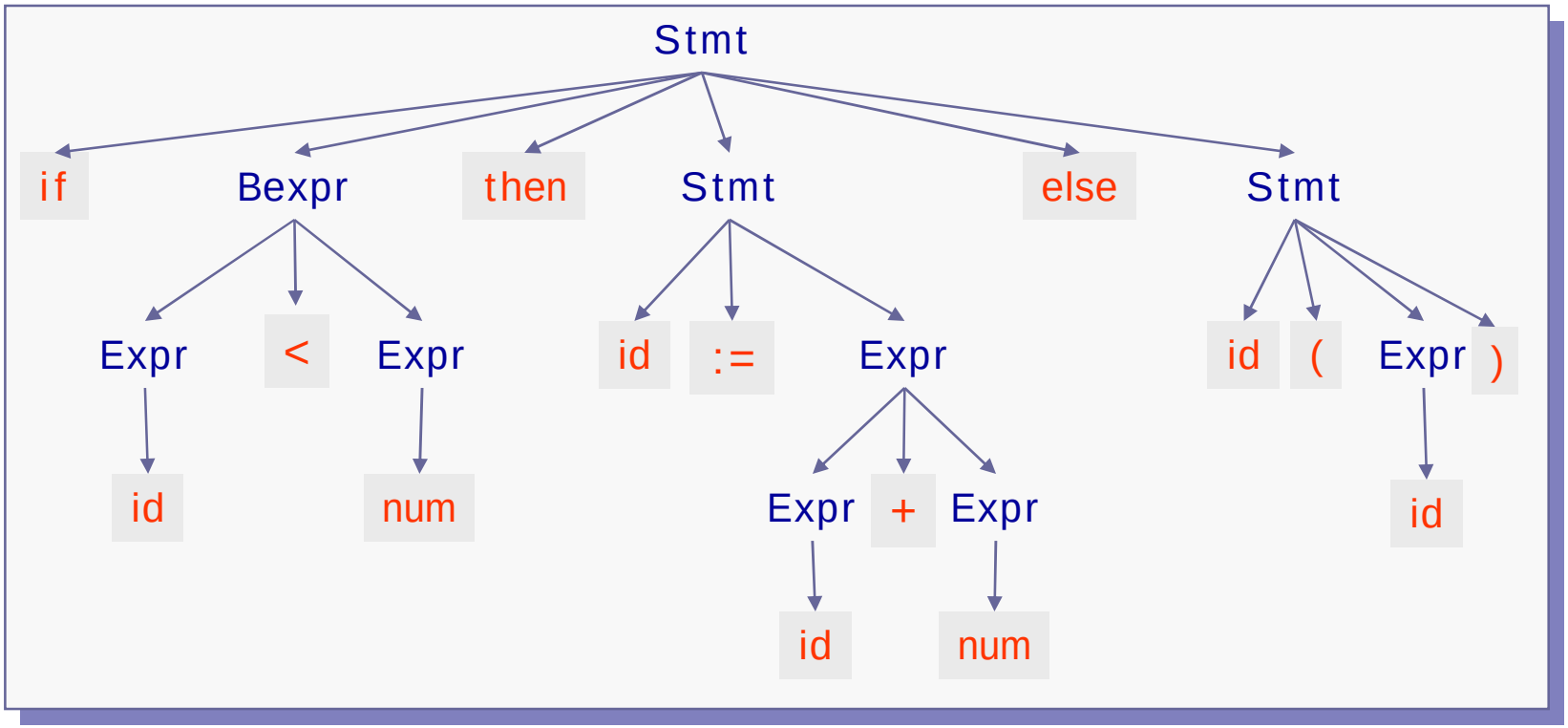

```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Bottom Up

reduce

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)

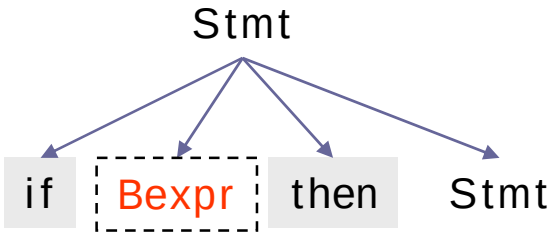
Fertig




```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Top Down Strategie

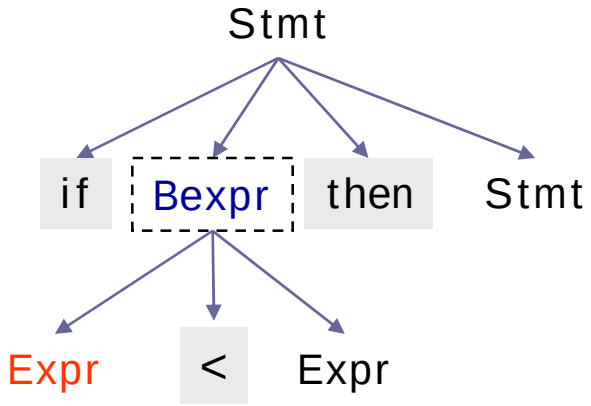
- Top Down
  - Baum wächst von der Wurzel zu den Blättern
  - jederzeit ein Baum
  - Blätter: Terminale und Nonterminale



- Aktuelles Blatt
  - gibt an, was im Input erwartet wird
  - alle Blätter links davon sind schon terminal

$$\text{Bexpr} \rightarrow \text{Bexpr} < \text{Bexpr} \mid \text{true}$$

- Falls aktuelles Blatt B
  - terminal
    - Lese nächstes Token im Input
    - prüfe, dass es mit B übereinstimmt
    - wenn nicht: „Error: B expected“
    - rechter Nachbar wird aktuelles Blatt
  - nonterminal
    - Wähle Grammatikregel $B \rightarrow w_1 \ldots w_n$
    - Hänge neue Söhne $w_1 \ldots w_n$ an
    - w1 wird neues aktuelles Blatt



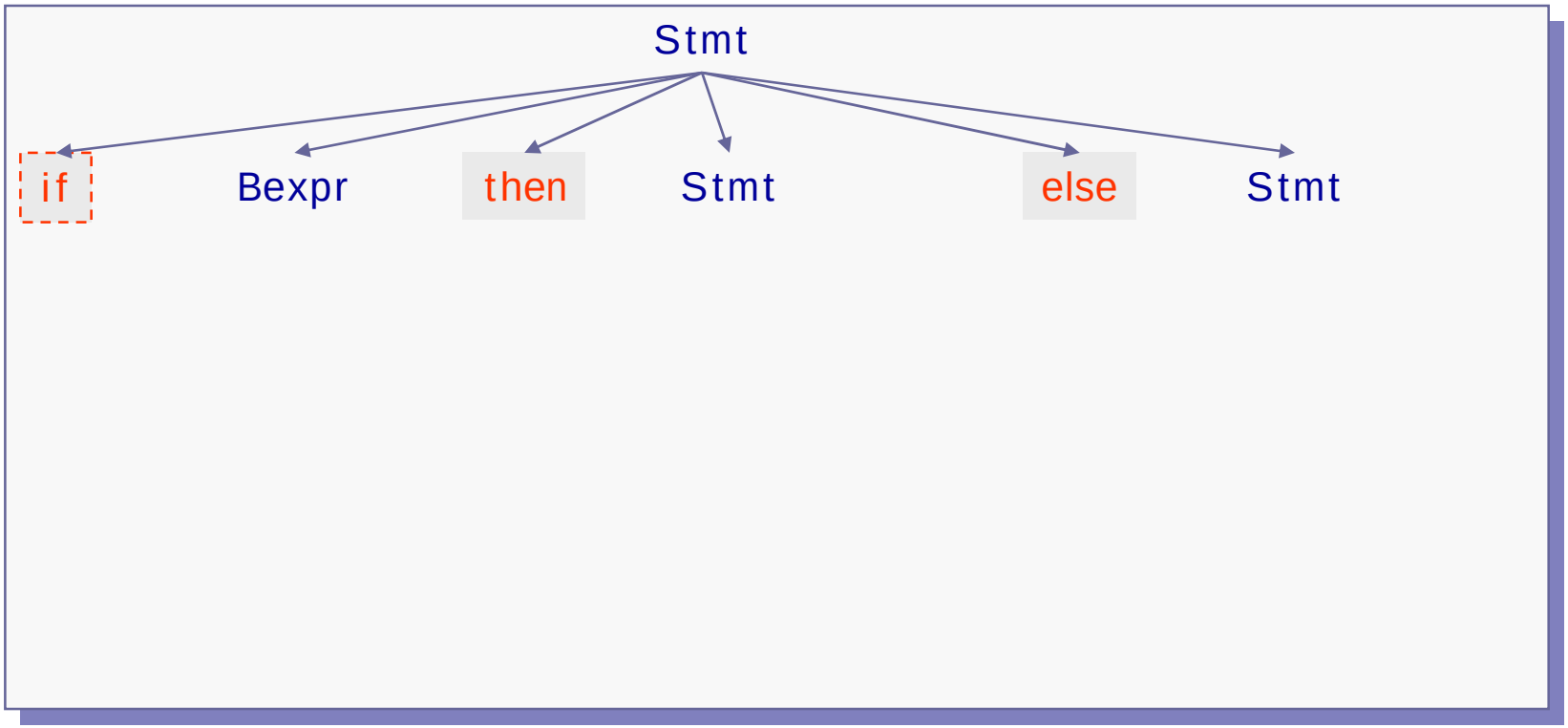
Top Down

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)

Stmt

if	

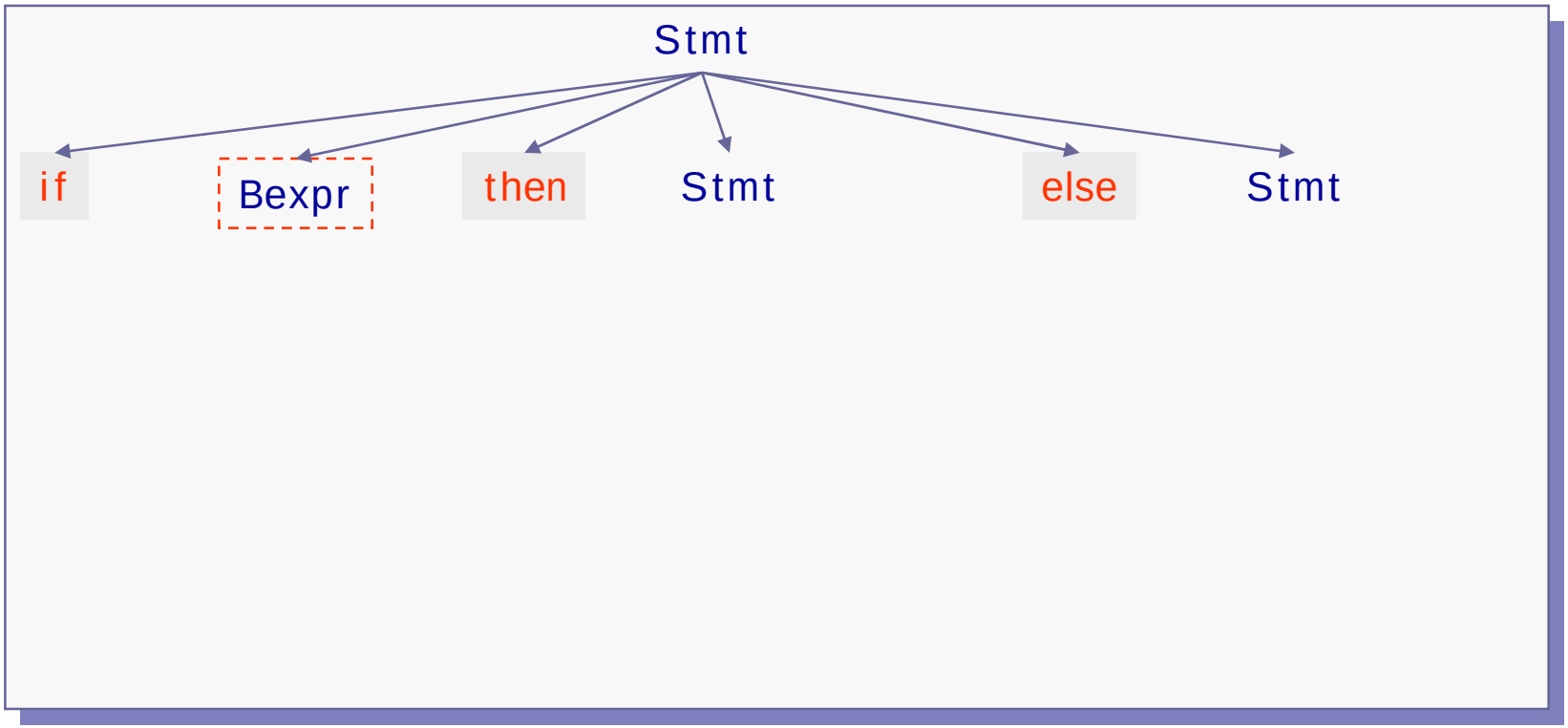
Top Down



if	

Top Down

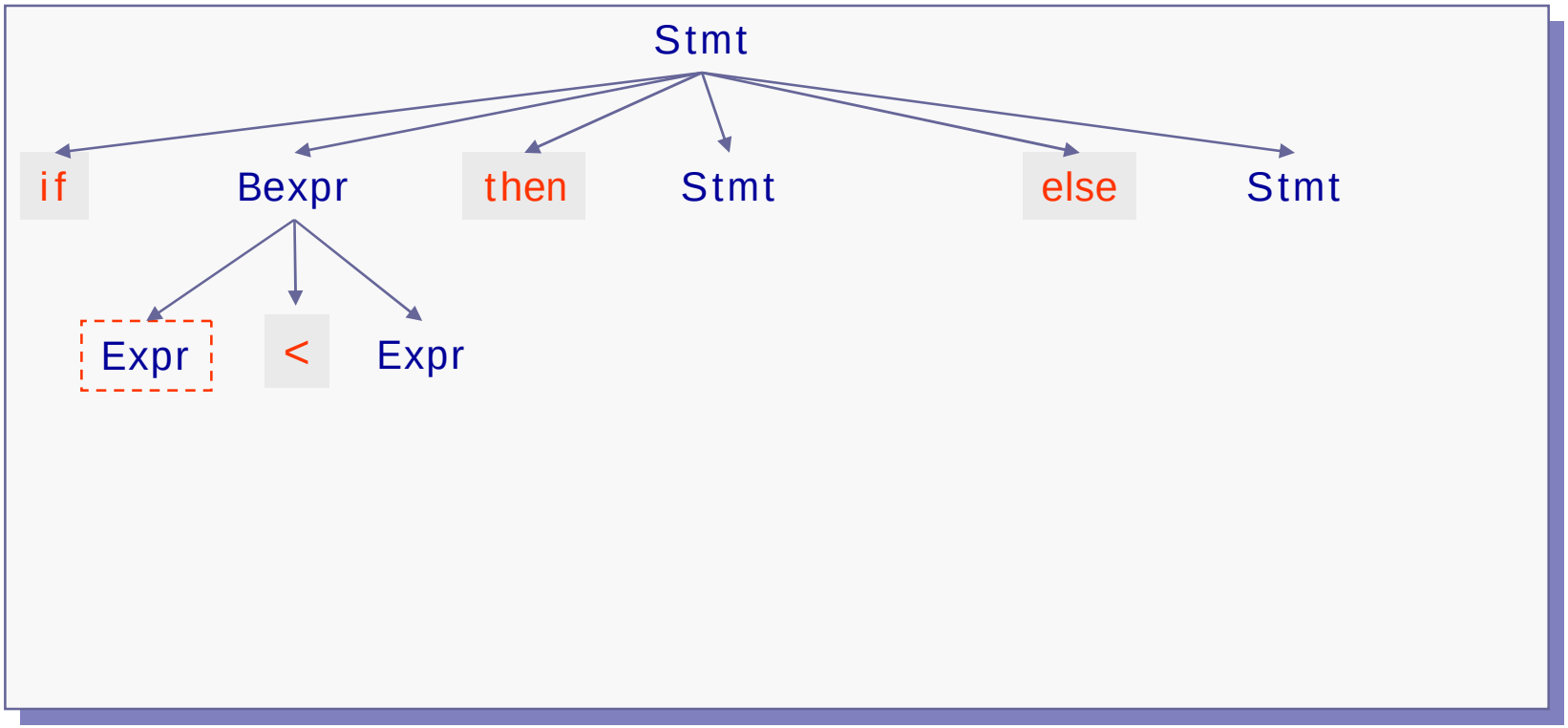
Bexpr → Expr < Expr
Bexpr → true



if x	

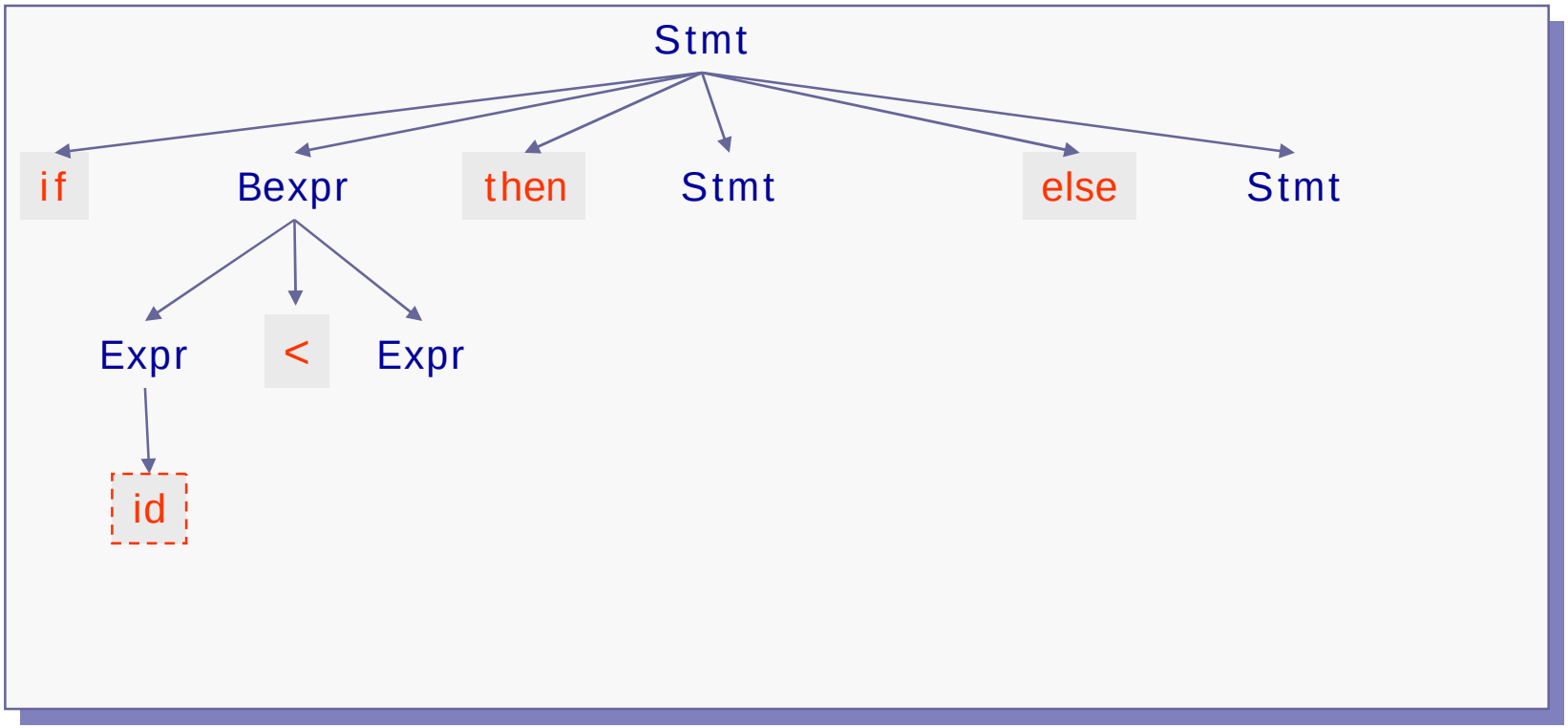
Top Down

Expr → Expr + Expr
Expr → id
Expr → num



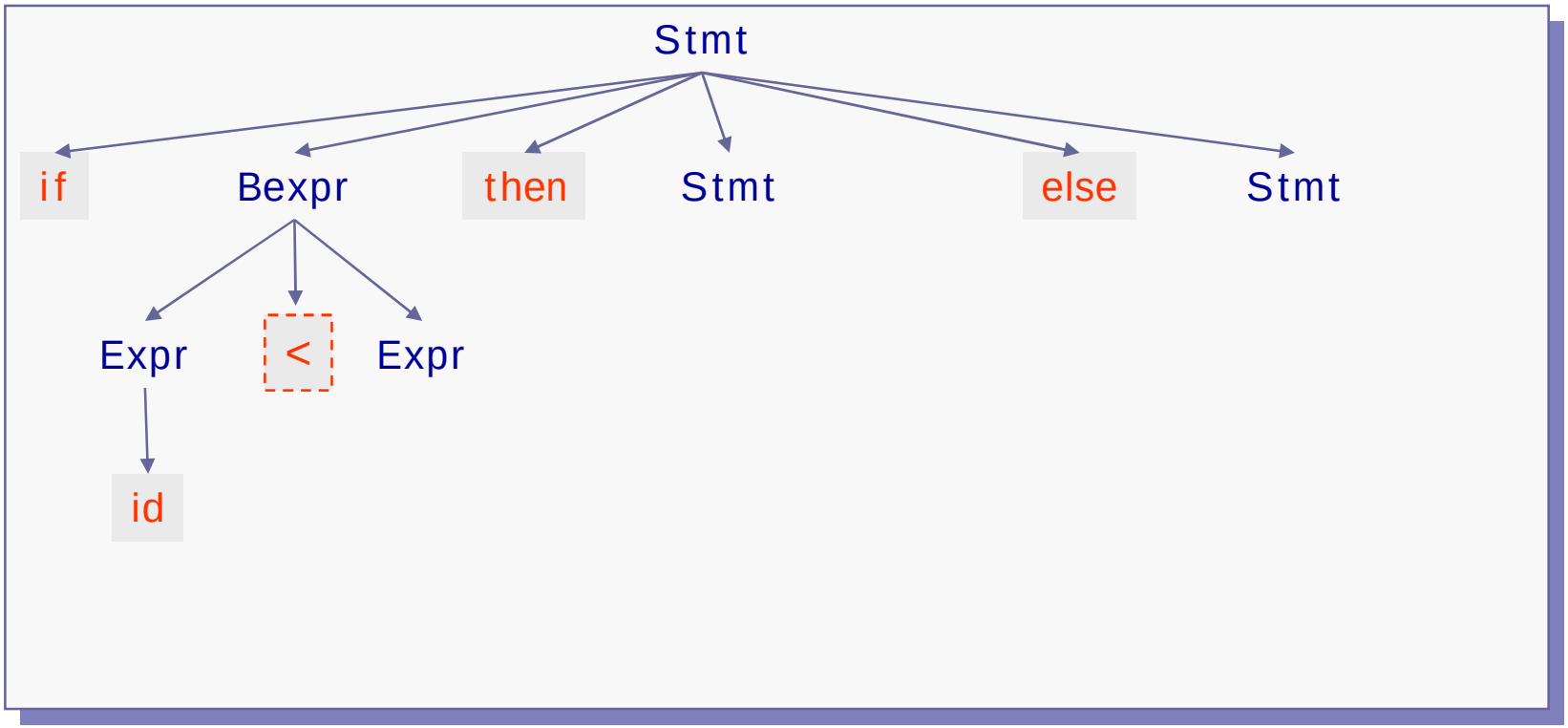
if x	

Top Down



`if x`	

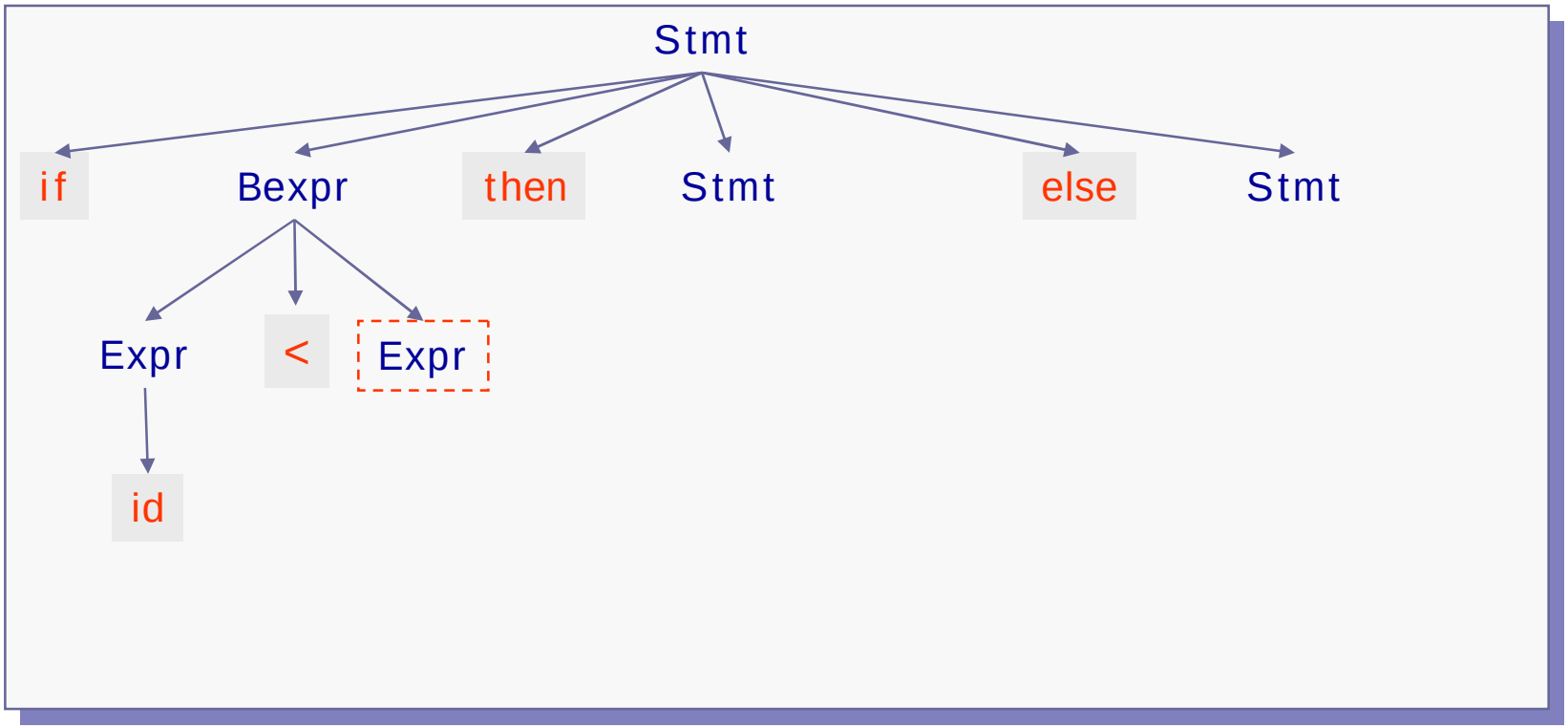
Top Down



`if x <`	

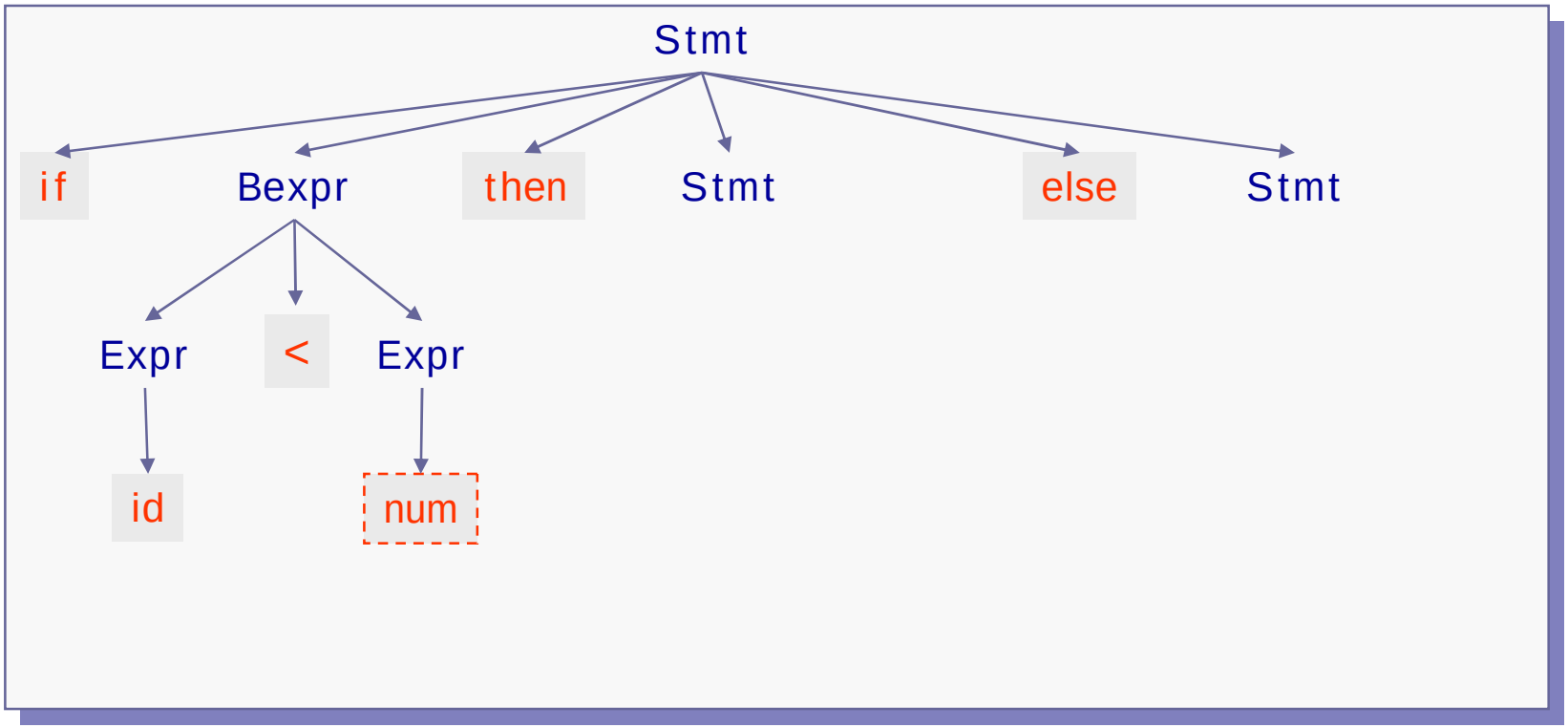
Top Down

Expr → Expr + Expr
Expr → id
Expr → num

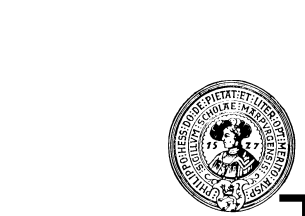


if	x	<	0

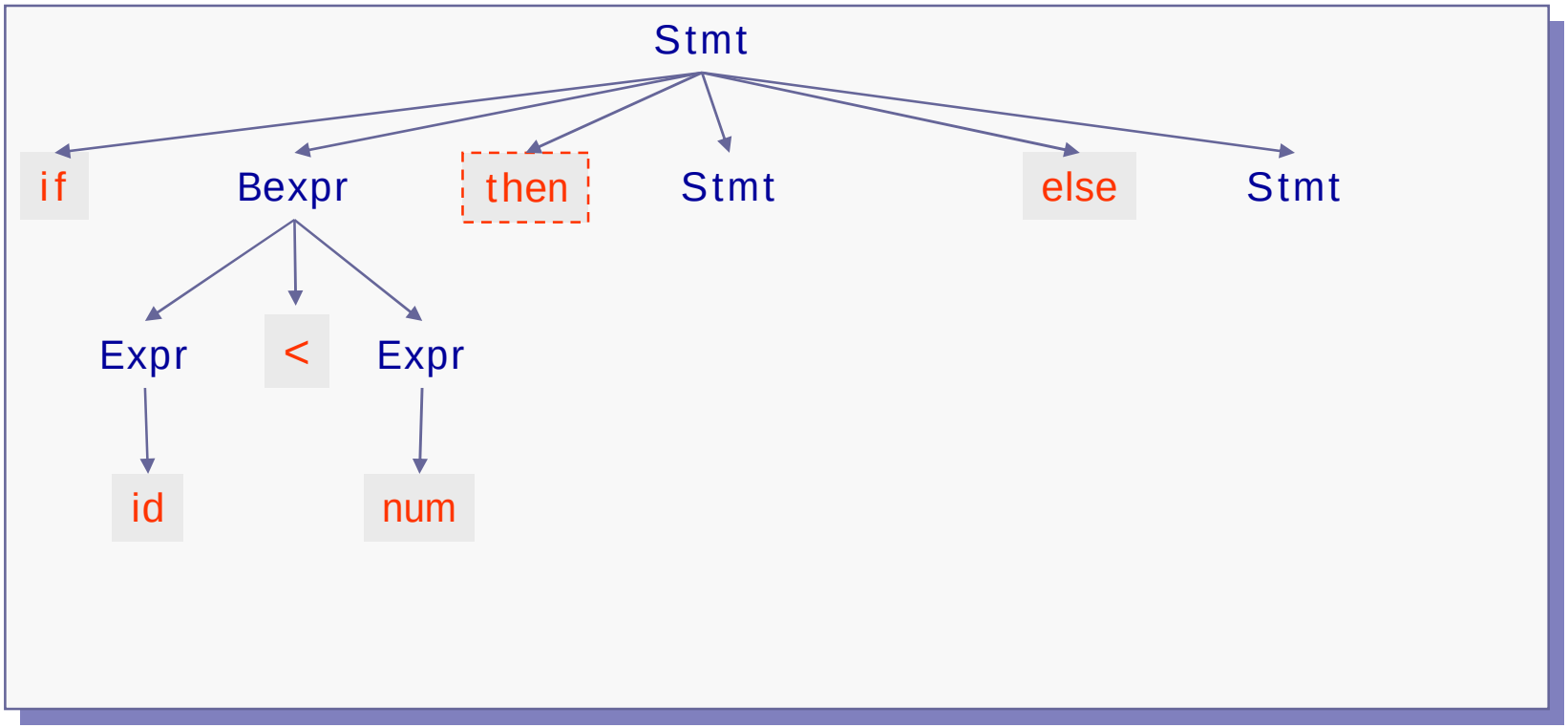
Top Down



`if x < 0`	



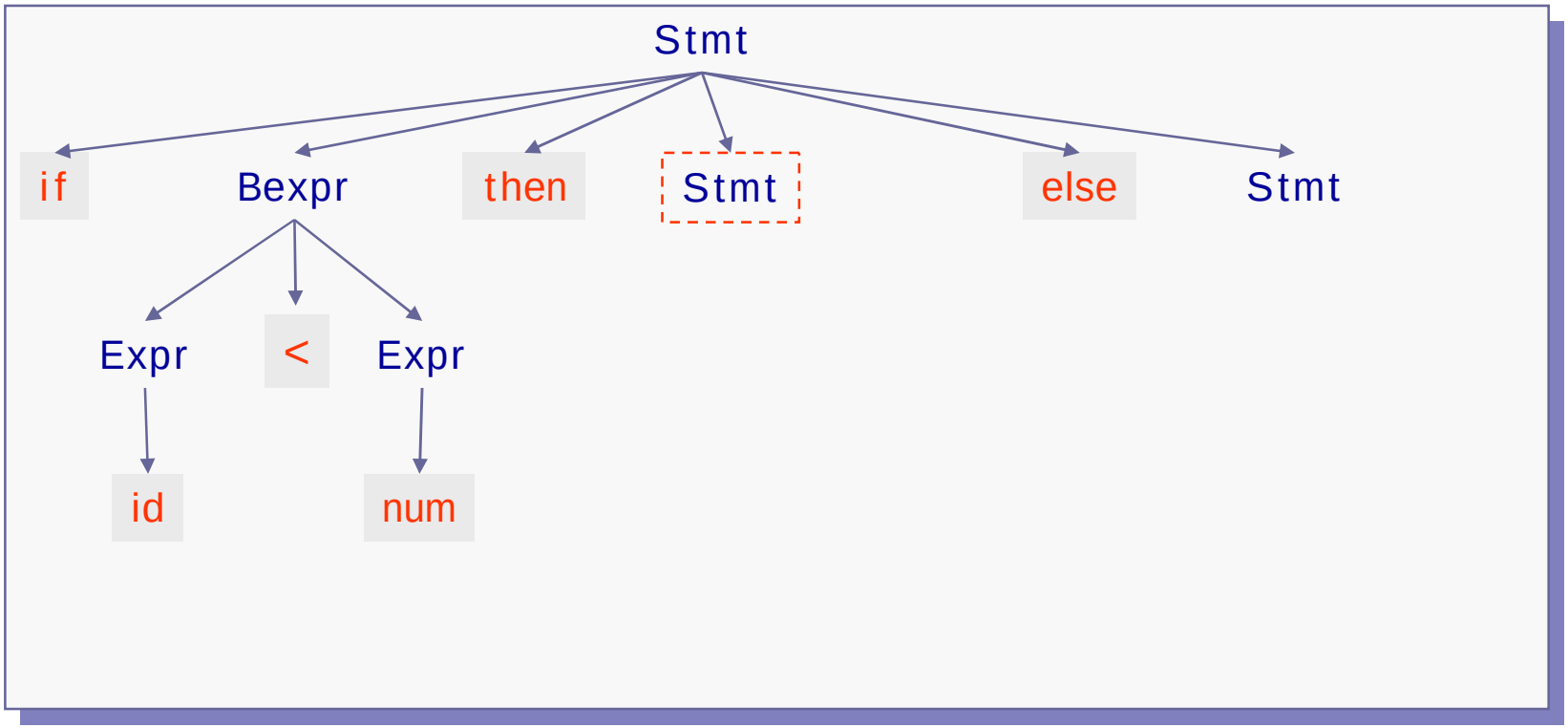
Top Down




```
if   x    <    0  then
```

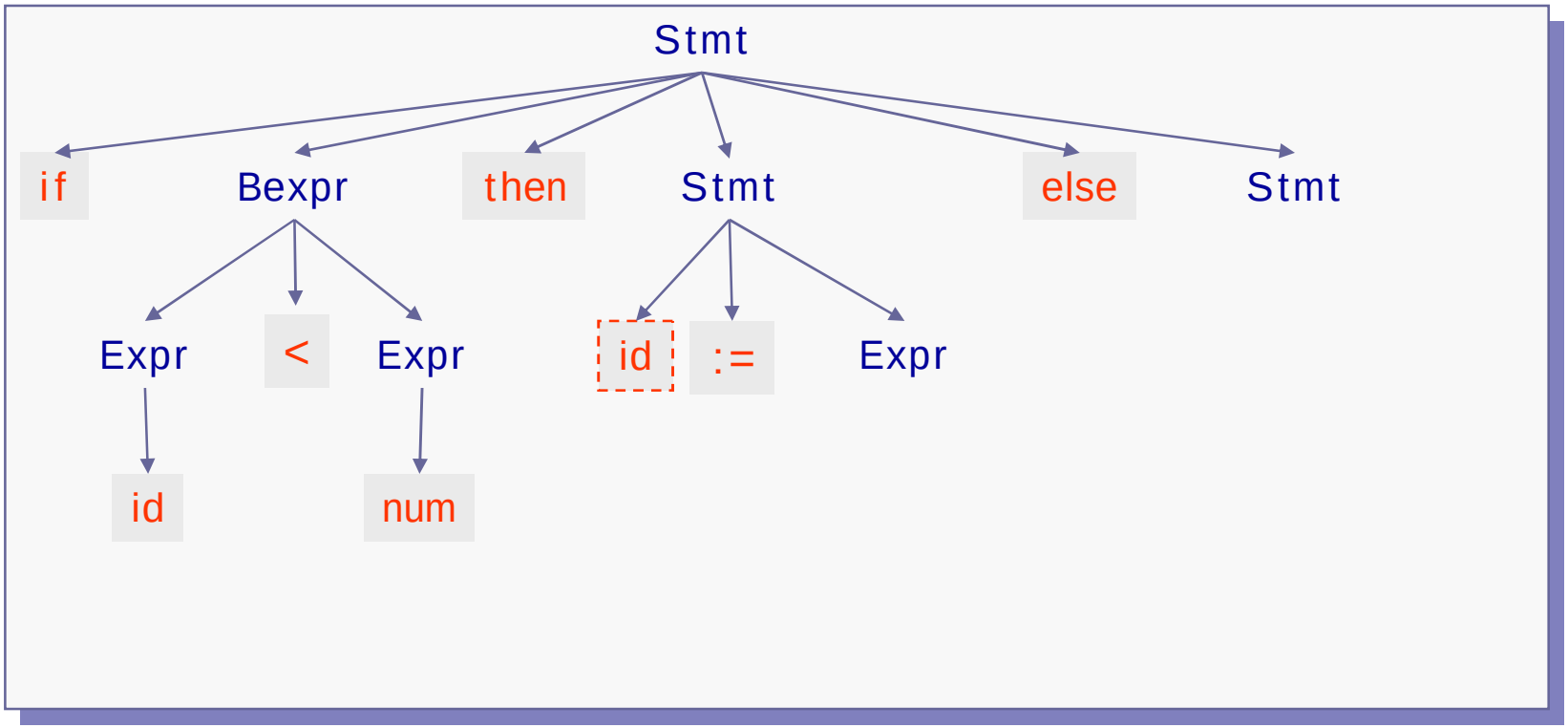
Top Down

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)

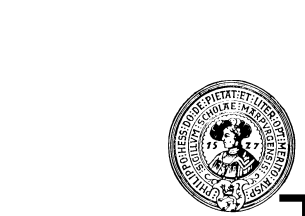


if	x	<	0	then	y	

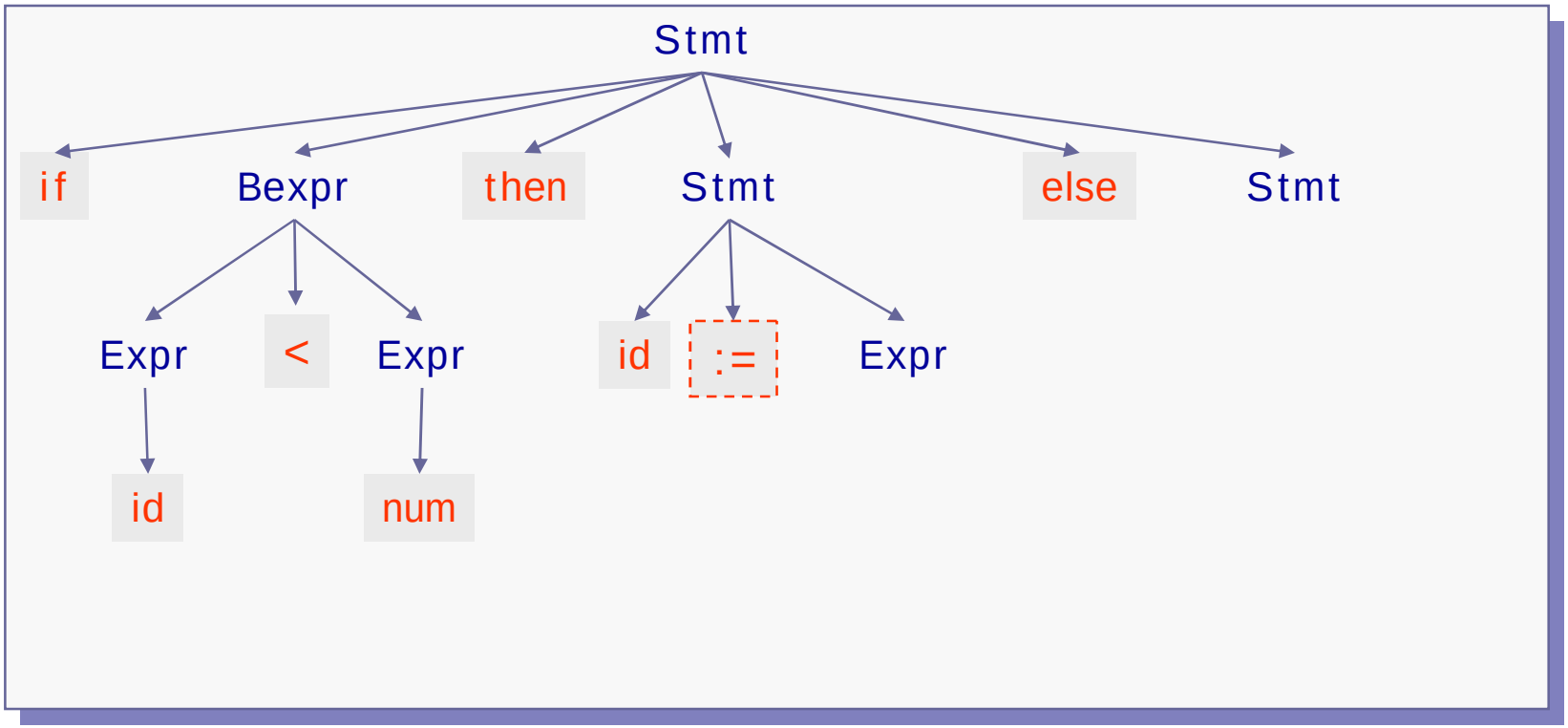
Top Down



if x < 0 then y	



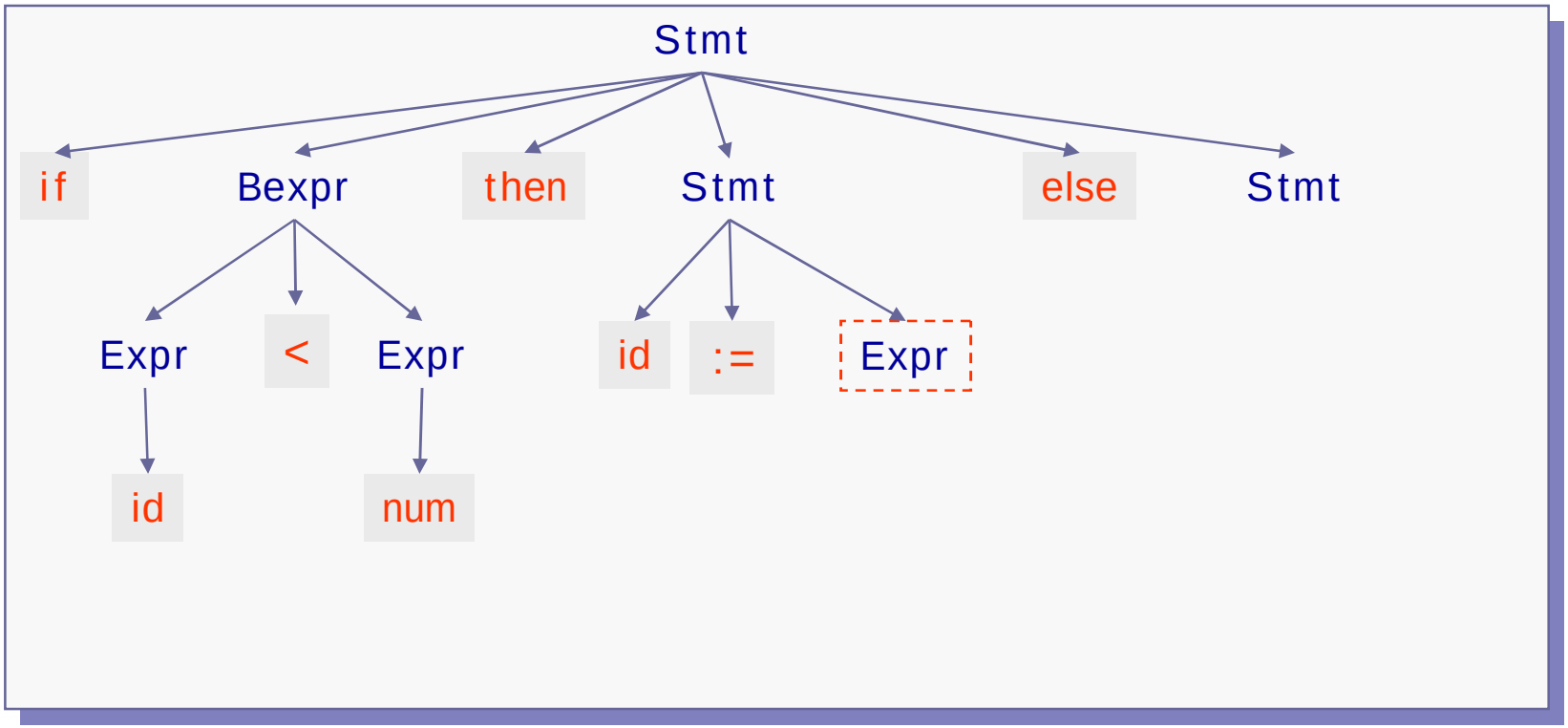
Top Down



if	x	<	0	then	y	:=	

Top Down

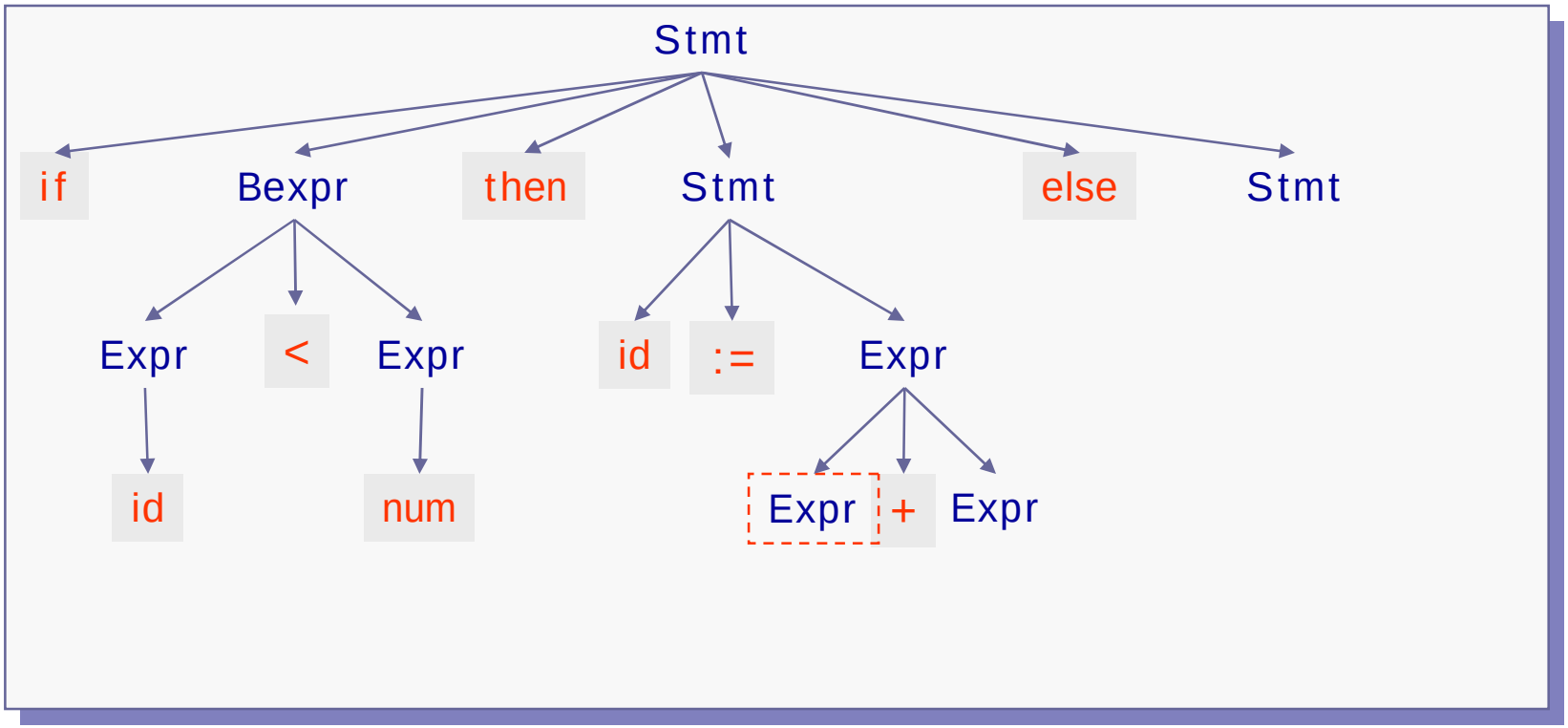
Expr → Expr + Expr
Expr → id
Expr → num



if	x	<	0	then	y	:=	y	

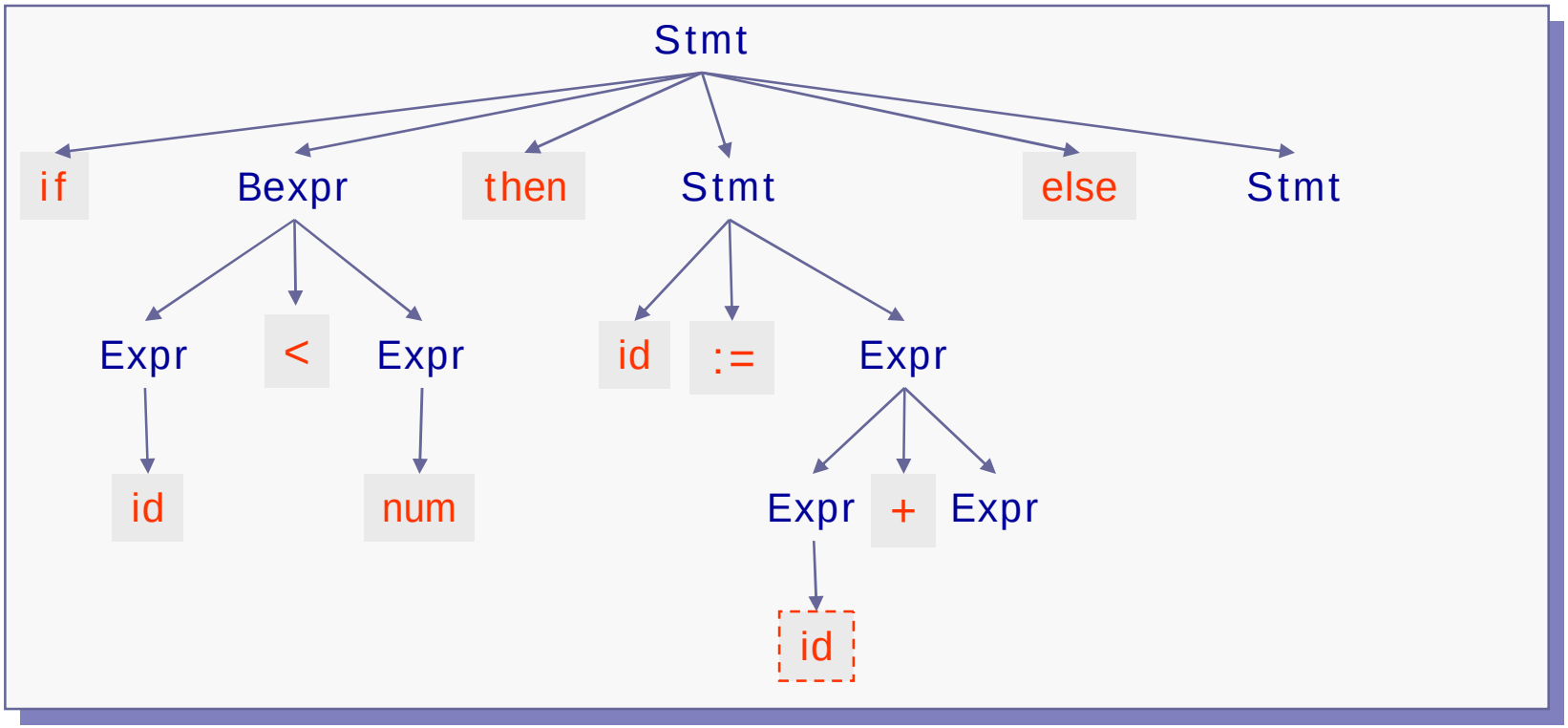
Top Down

Expr → Expr + Expr
Expr → id
Expr → num



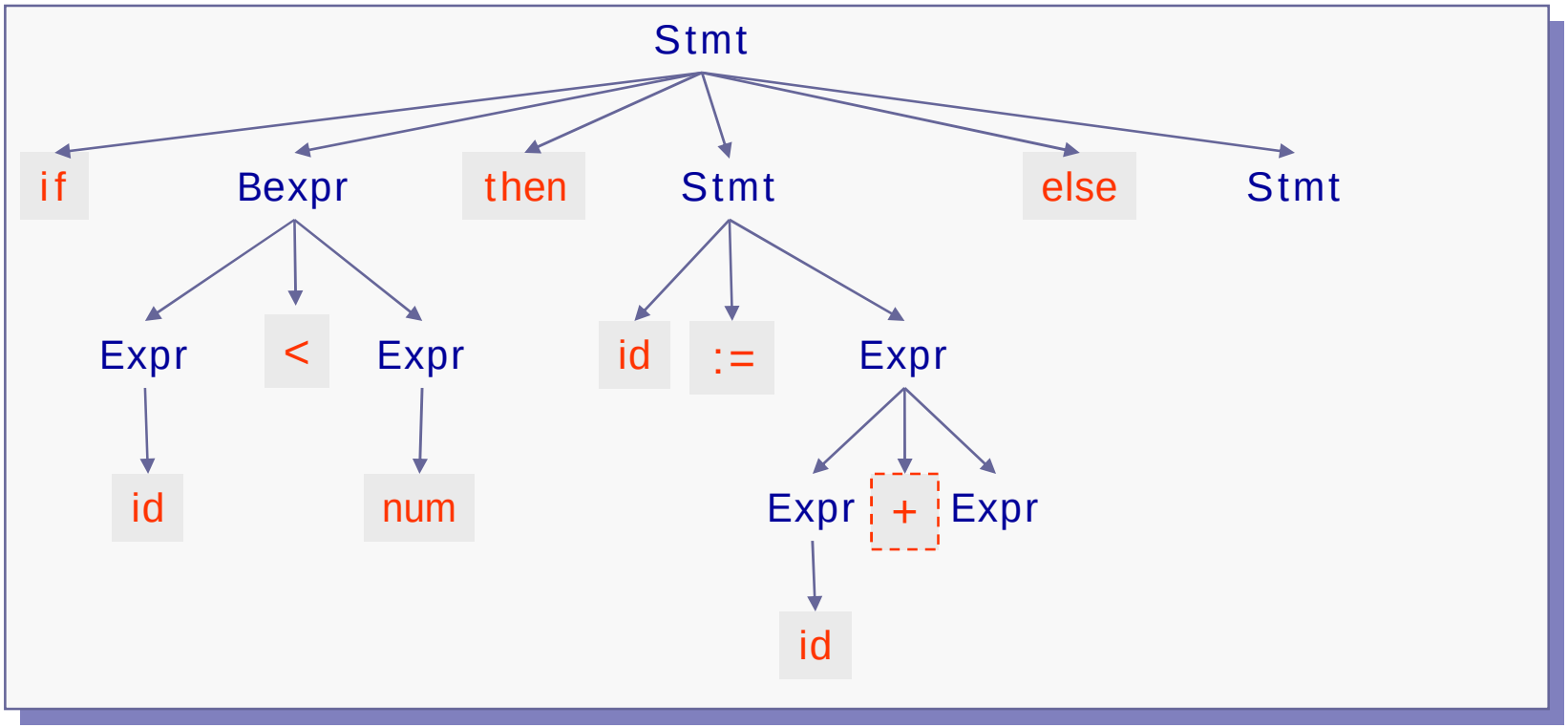

```
if   x    <    0  then  y  :=  y
```

Top Down



if	x	<	0	then	y	:=	y	

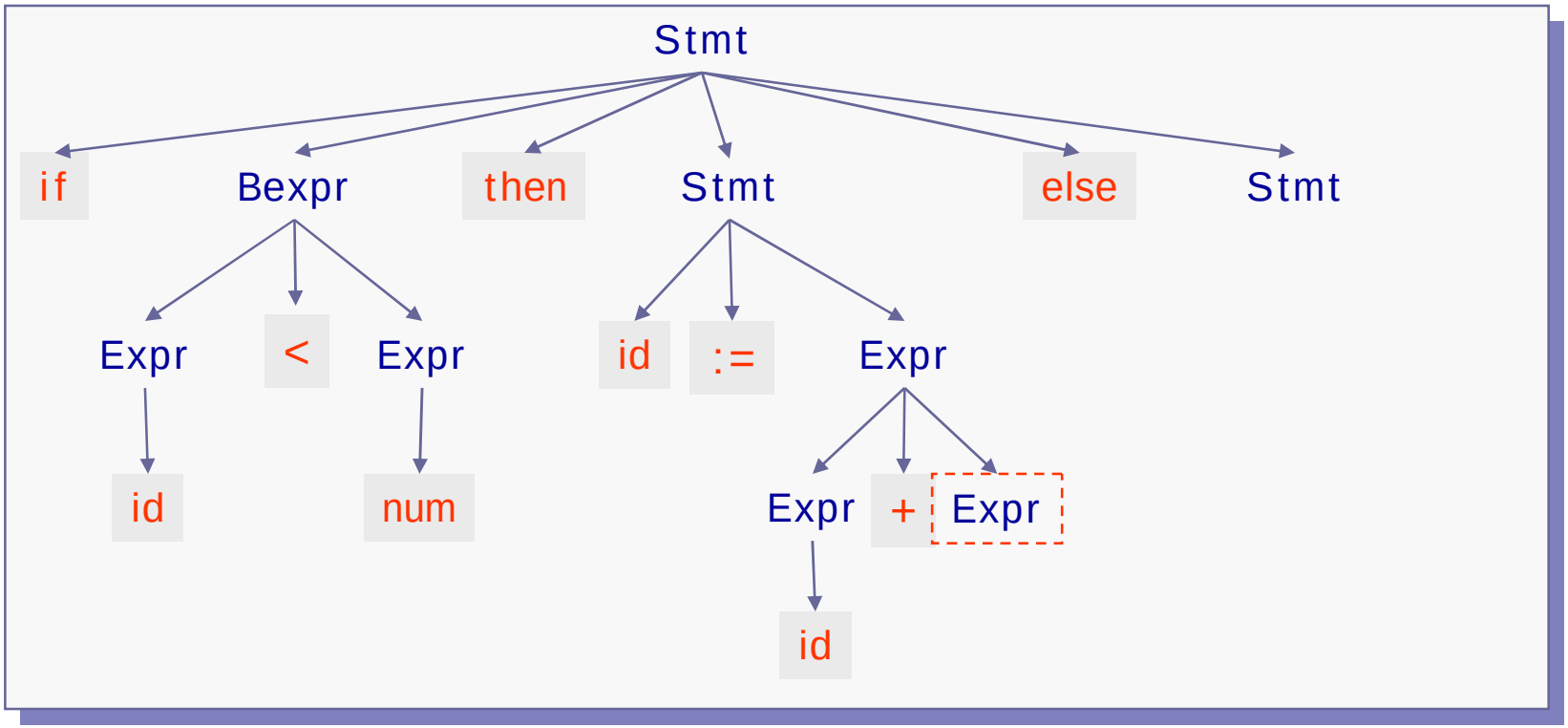
Top Down



if	x	<	0	then	y	:=	y	+	

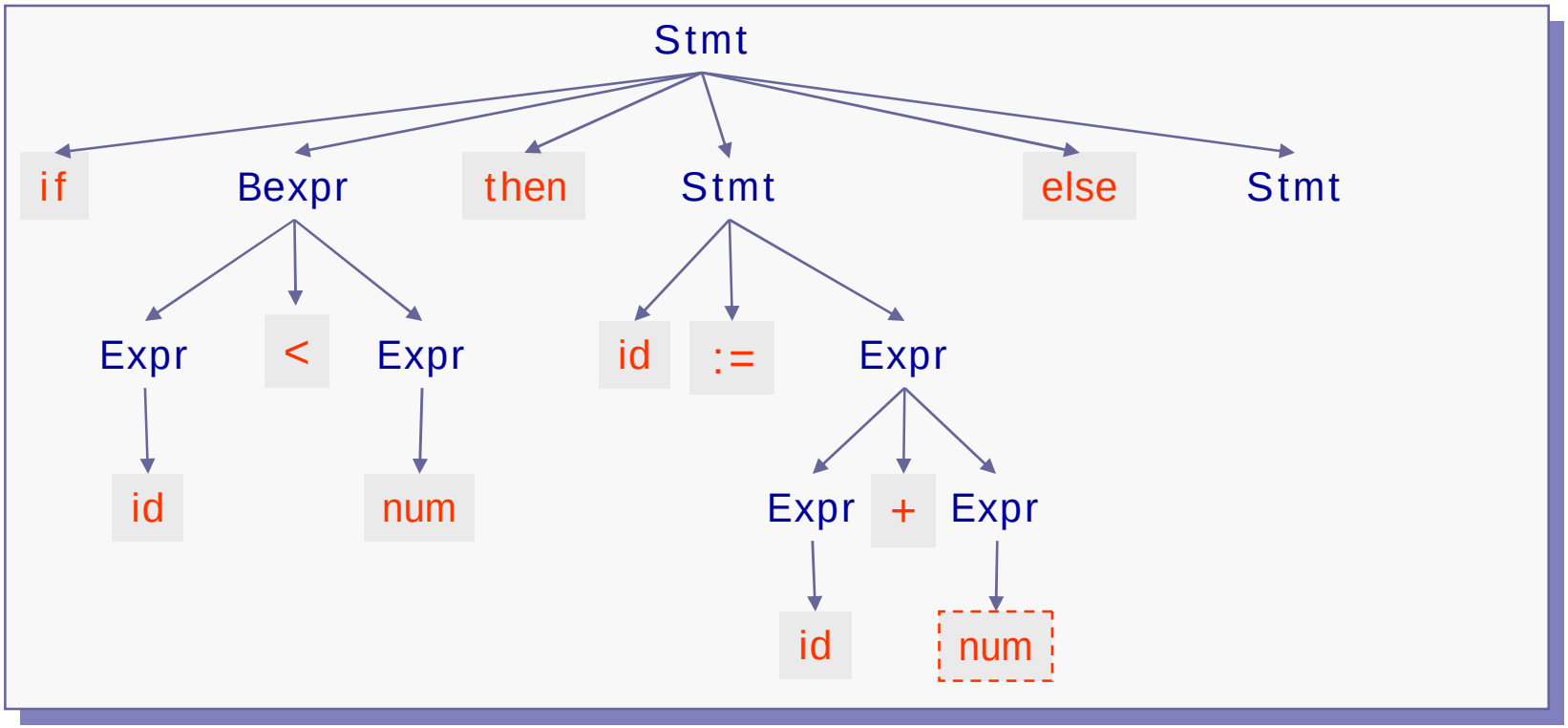
Top Down

Expr → Expr + Expr
Expr → id
Expr → num

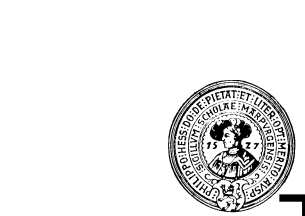



```
if   x    <    0  then  y  :=  y  +  1
```

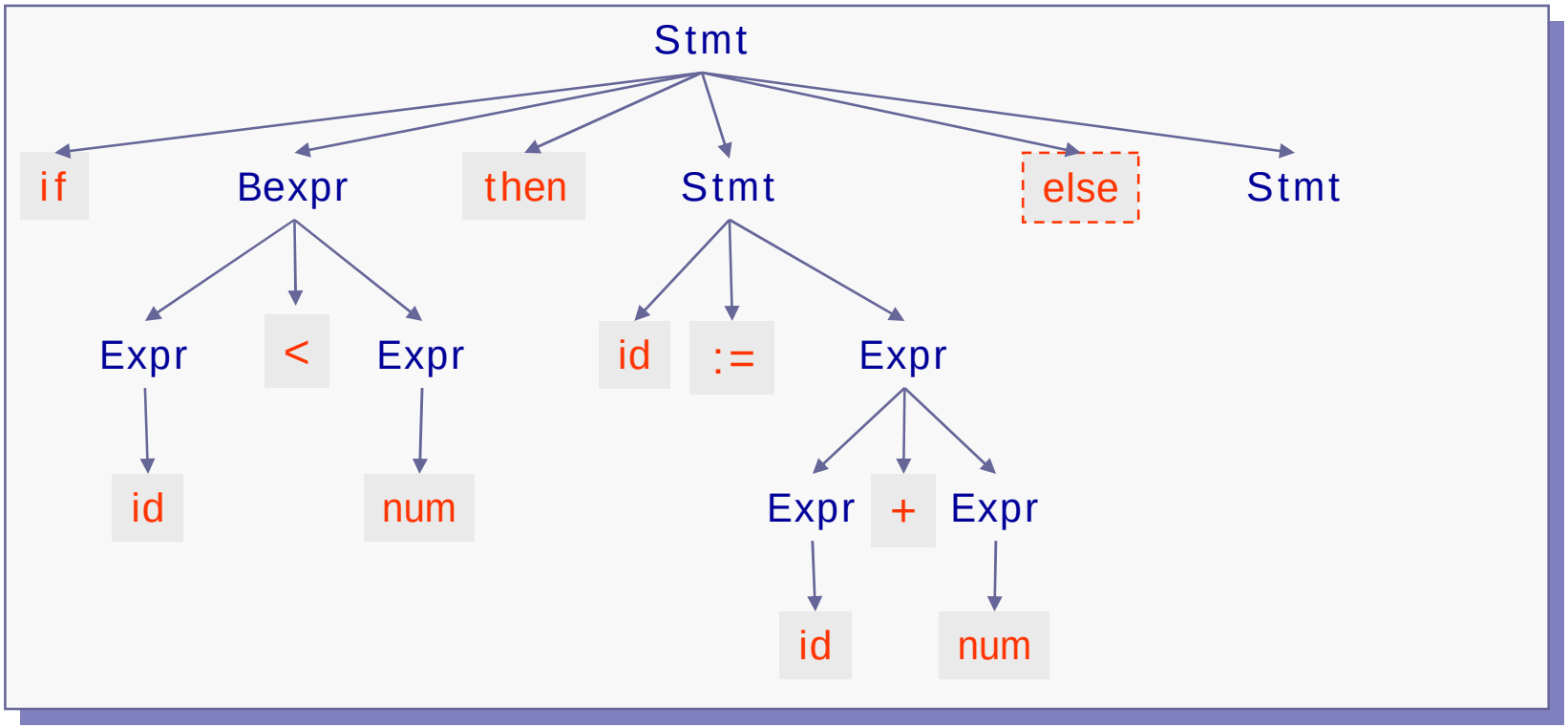
Top Down



if	x	<	0	then	y	:=	y	+	1	



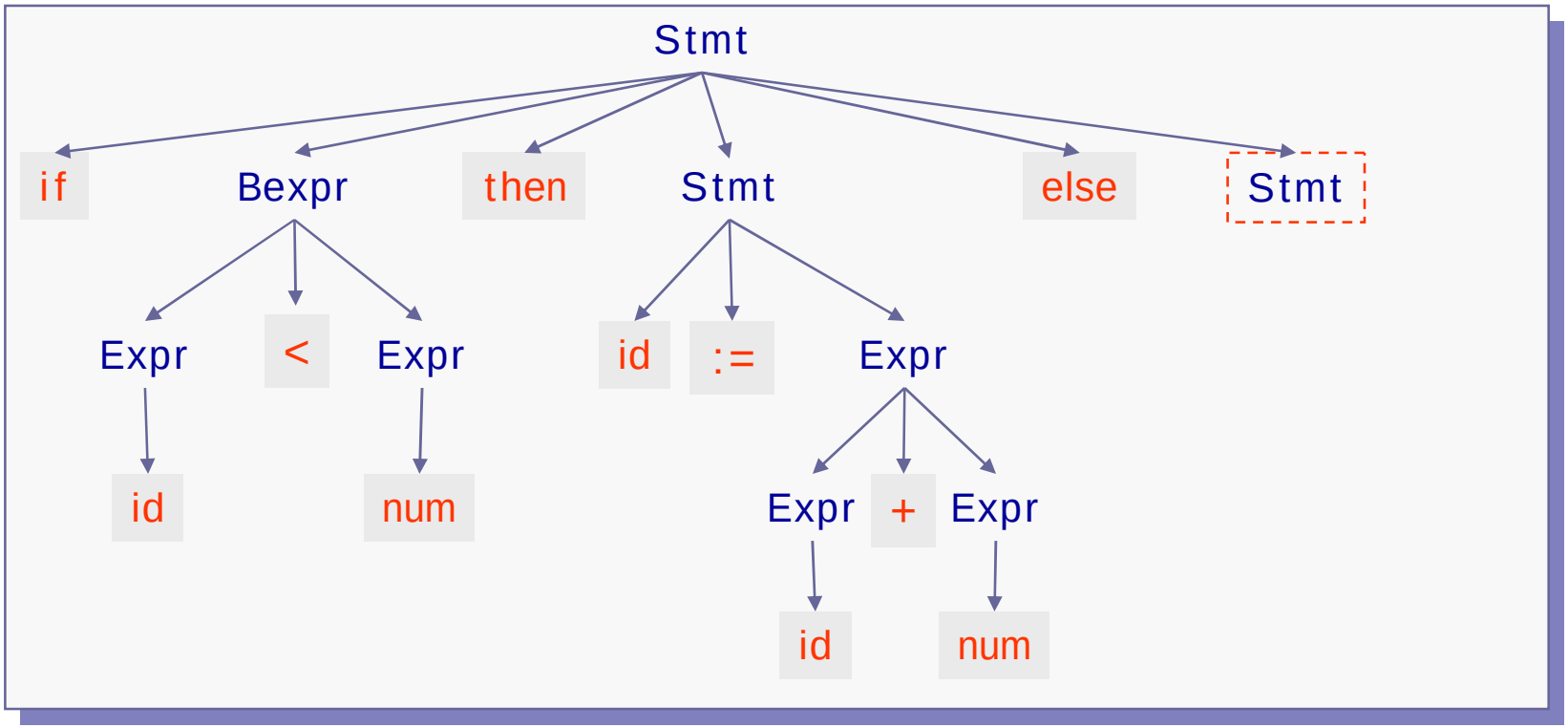
Top Down



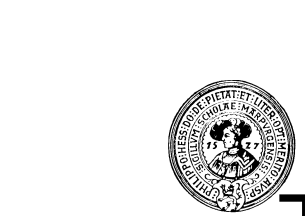
if	x	<	0	then	y	:=	y	+	1	else	

Top Down

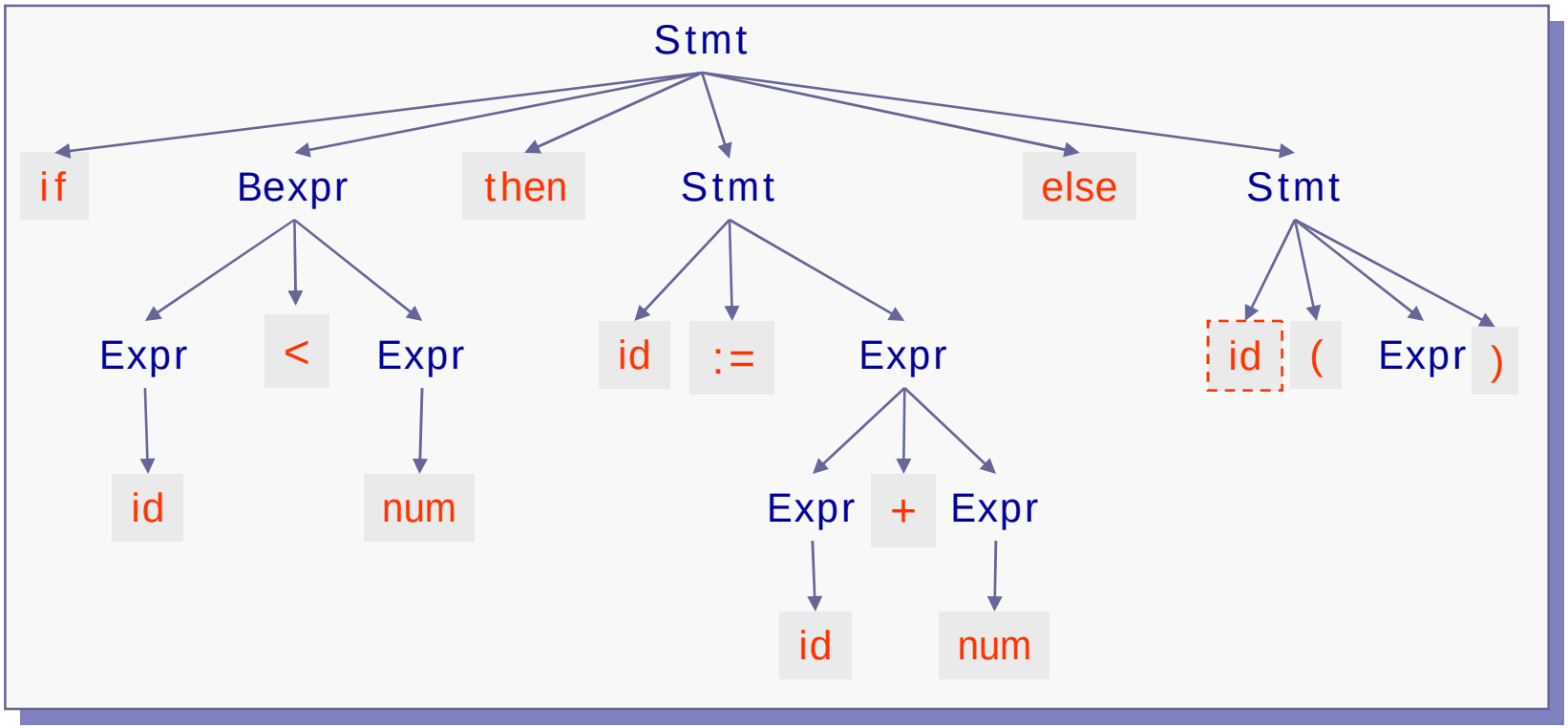
Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)



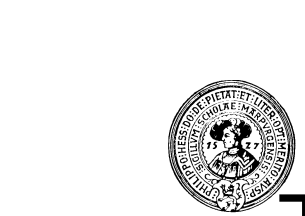
if	x	<	0	then	y	:=	y	+	1	else	Inc	



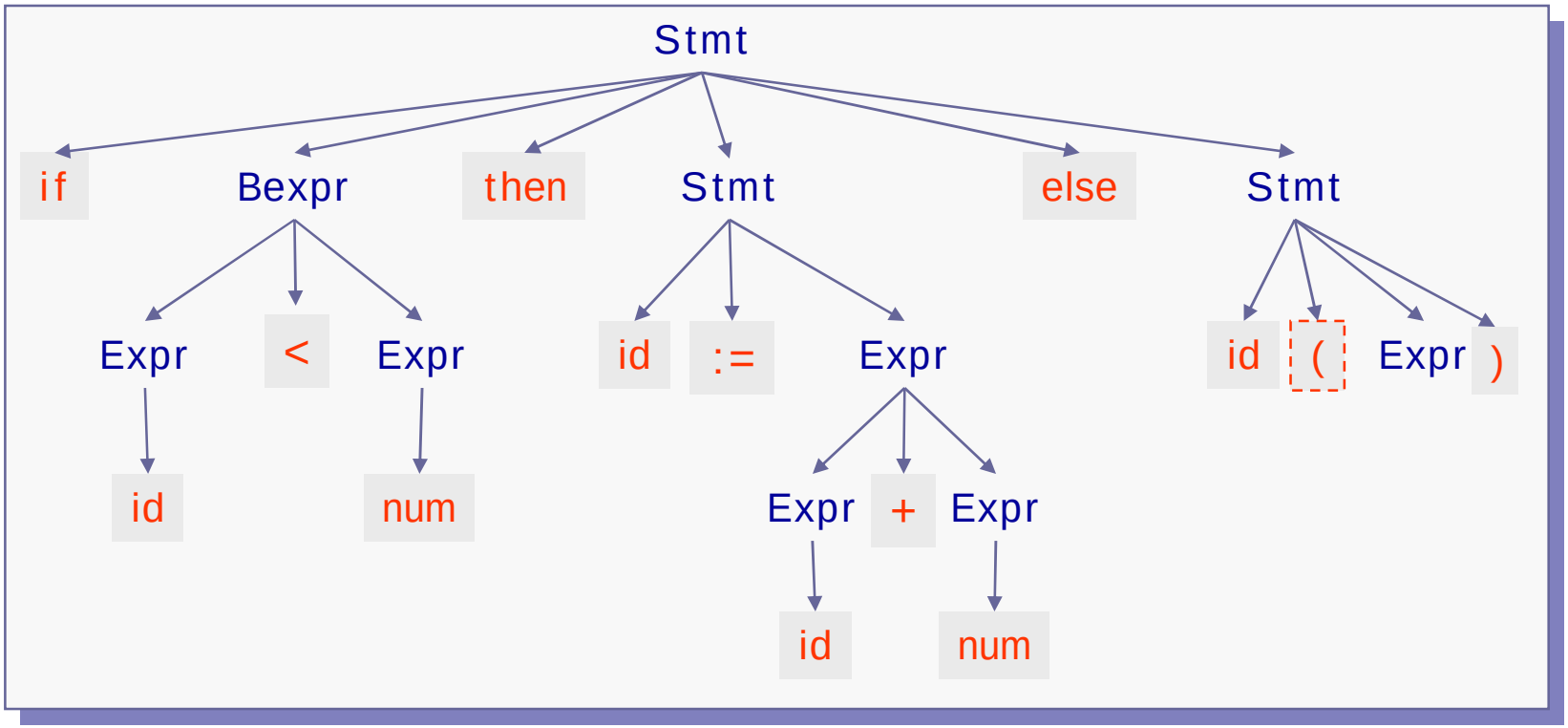
Top Down



if	x	<	0	then	y	:=	y	+	1	else	Inc	



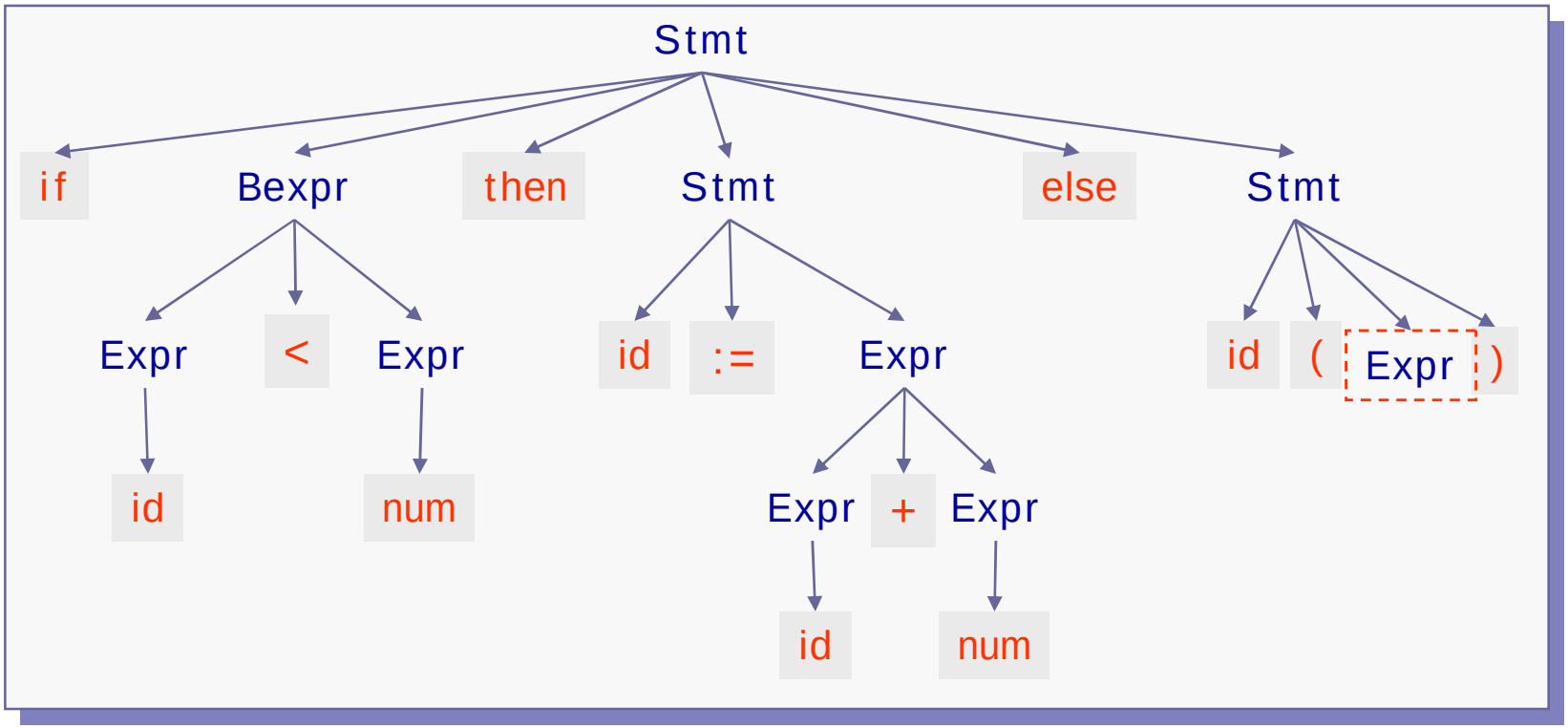
Top Down



if	x	<	0	then	y	:=	y	+	1	else	Inc	(	

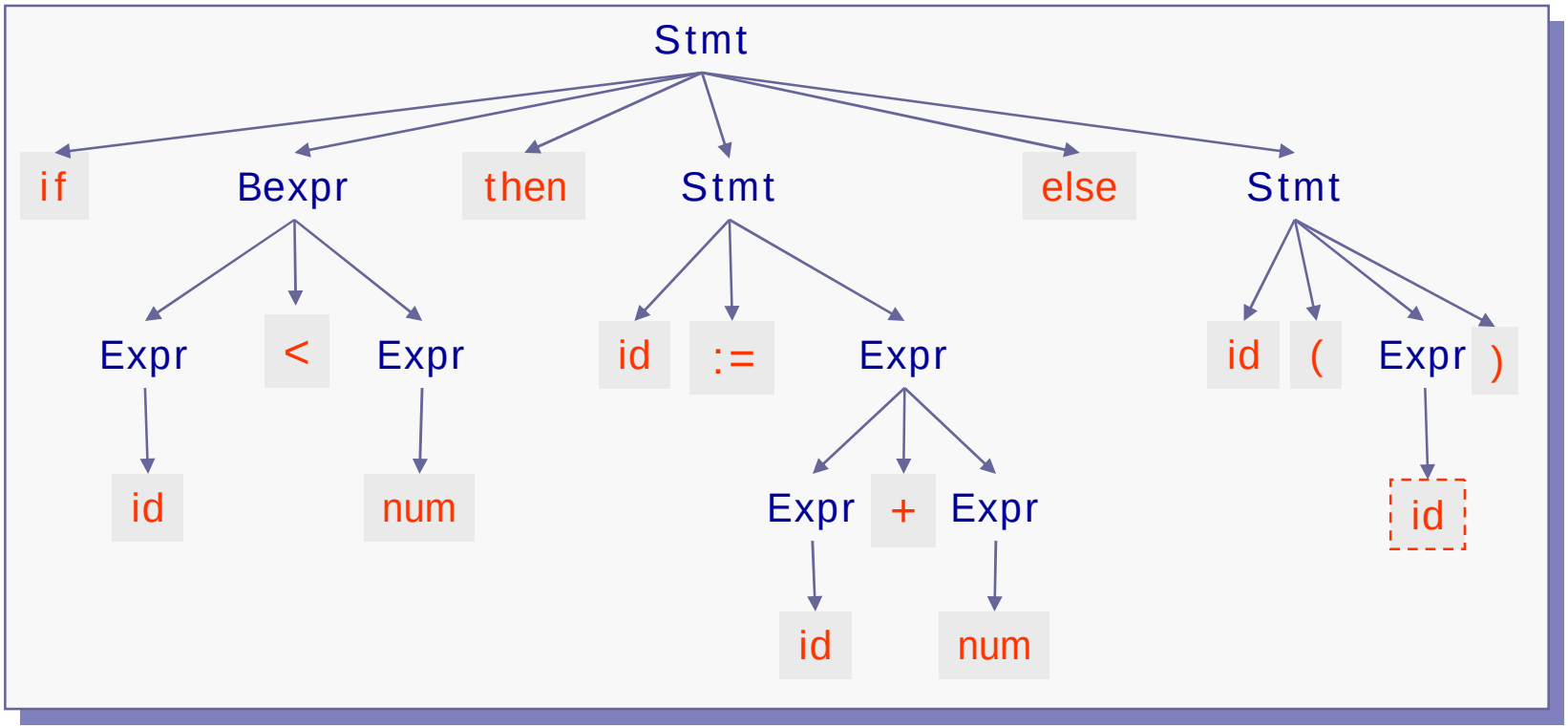
Top Down

Expr → Expr + Expr
Expr → id
Expr → num



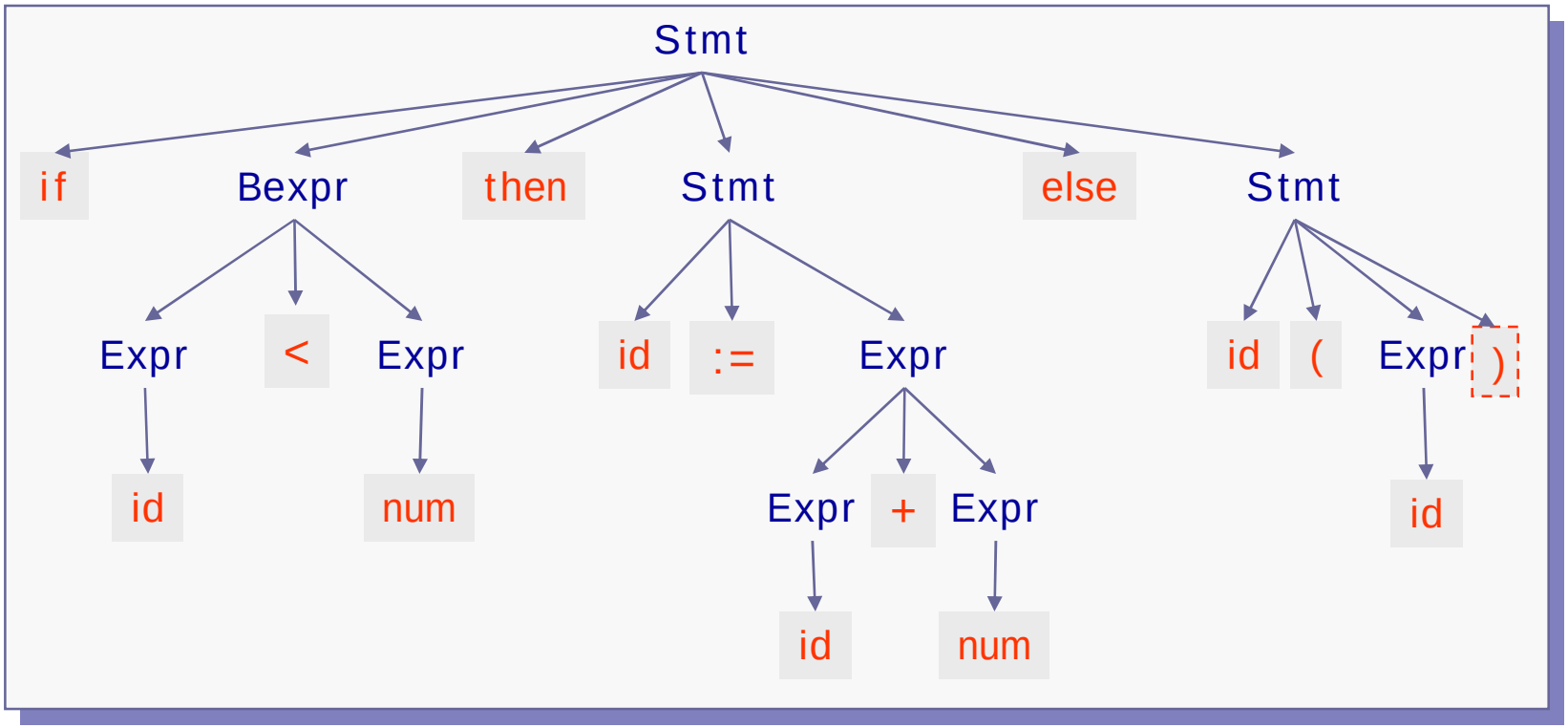
if	x	<	0	then	y	:=	y	+	1	else	Inc	(	z	

Top Down




```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z
```

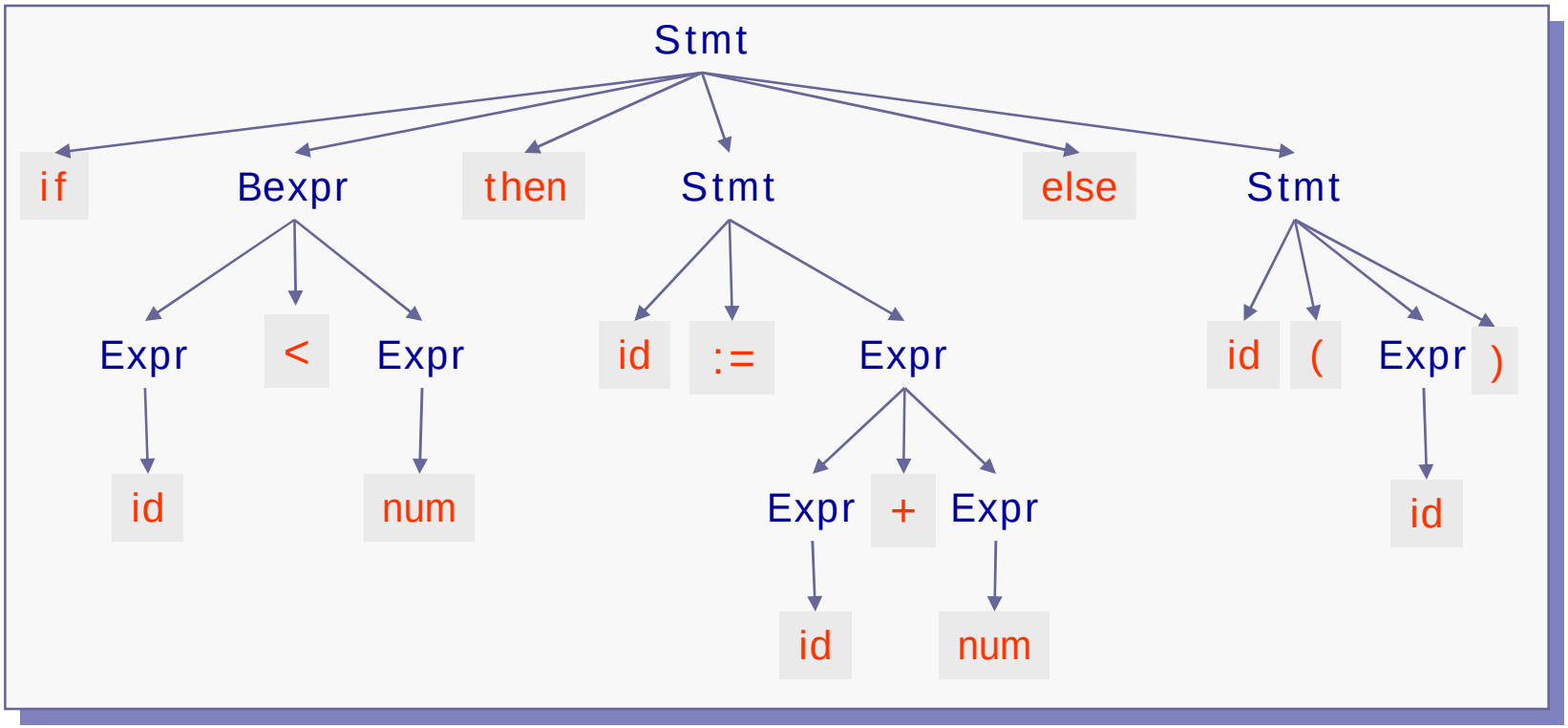
Top Down




```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Top Down

Fertig




```
if   x    <    0  then  y  :=  y  +  1  else  Inc  (   z   )
```

Recursive Descent Parsing

Eine Produktion $A \rightarrow \beta$ kann man als Bedingung verstehen, unter welchen Umständen ein A akzeptiert werden soll.

G

```
Factor → id
       | ( Expr )
```

```
TO parse( Factor )
        consume( id )
     OR
        consume( ( ) ;  parse( Expr ) ; consume( ) ).
```

Die Grammatik ist (fast) schon ein Programm

Recursive Descent Parsing

Ein „Recursive Descent Parser" (RD-Parser) entsteht aus einer Grammatik auf folgende Weise:

1. Eine Methode „consume" für die Terminale:

```
void consume(Token t){
  Token in = nextToken(); // ScannerAufruf
  if (in==t) return;
  else fehler(t + "expected");
}
```

2. Für jedes Nonterminal A eine Methode parseA:

```
void parseA(){


}
```

Grammatik zum RD-Parser

Aus der Grammatik ...

Expr → Term Rest

Rest → + Term Rest
| - Term Rest
| ε

Nondeterminismus muss noch entfernt werden

.... werden (fast) Methoden

```
void parseExpr(){
   parseTerm();
   parseRest();
}
```

```
void parseRest(){
 ENTWEDER
   consume(+);
   parseTerm();
   parseRest()
 ODER
   consume(-);
   parseTerm();
   parseRest();
 ODER
   return;
}
```

Auflösung des Nondeterminismus.

- Entscheidung zwischen den Alternativen durch Vorausschau auf das nächste Token (lookahead).

- G heißt LL(1), falls dieses die richtige Regel eindeutig bestimmt.

G

Rest → + Term Rest
| - Term Rest
| ε

```
void parseRest(){

  switch( lookahead )

   case + : consume(+);
            parseTerm();
            parseRest();
          return;

   case - : consume(-);
            parseTerm();
            parseRest();
          return;

   default : return;
}
```

First

G enthalte

$A \to \sigma$

$A \to \tau$.

In parseA ist Entscheidung anhand eines lookahead nur möglich, falls aus σ bzw. τ hergeleitete Worte nicht mit dem gleichen Terminal beginnen können,

Def.

$First(\alpha) := \{ t \in T \mid \alpha \Rightarrow^* tw \}$

zusätzlich

$\varepsilon \in First(\alpha) \Leftrightarrow \alpha \Rightarrow^* \varepsilon$

Notwendige Bedingung für LL(1): Für je zwei $A \to \sigma$, $A \to \tau \in P$ mit $\sigma, \tau \neq \varepsilon$ ist

$$First(\sigma) \cap First(\tau) = \emptyset$$

Berechnung von First(α)

$$
\begin{aligned}
First(\varepsilon) &= \{\varepsilon\} \\
First(\ t.\tau\ ) &= \{\ t\ \} \qquad\qquad // \ \ t \text{ terminal} \\
First(\ A.\tau\ ) &= \cup\ \{\ First(\alpha\ ) \mid A \rightarrow \alpha \in G\ \} \\
&\quad \cup\ \{\ First(\tau) \mid \text{falls A kollabierend}\ \}
\end{aligned}
$$

Fast wieder ein rekursives Programm. Nur für Terminierung sorgen !!

ε-Produktionen

Zwei kollabierende Variablen reklamieren das leere Wort:

$$\begin{aligned} String &\rightarrow Expr \\ &\mid String + String \\ &\mid \varepsilon \end{aligned}$$

$$\begin{aligned} Expr &\rightarrow num\ Rest_E \\ Rest_E &\rightarrow + num\ Rest_E \\ &\mid \varepsilon \end{aligned}$$

Angenommen, wir parsen einen String (parseString)

3 + ‘=‘ + 1 + 2

An dieser Stelle kann sich $parseRest_E$ nicht allein anhand des nächsten Tokens + entscheiden ! Hängt davon ab, was folgen kann.

3 + 4 ‘=‘ 7

Grammatik nicht LL(1) !!

Follow

$String \rightarrow Expr$
$\quad | \; String + String$
$\quad | \; \varepsilon$

$Expr \rightarrow num \; Rest_E$

$Rest_E \rightarrow + \; num \; Rest_E$
$\quad | \; \varepsilon$

- RestE kann mit einem + beginnen,
  - spricht für $RestE \rightarrow + \; num \; RestE$
- auf ein RestE kann aber auch ein + folgen
  - spricht für $RestE \rightarrow \varepsilon$

$$Follow(A) = \{ \; t \mid t \in T, \quad S \Rightarrow^* \alpha \; A \; t \; \beta \; \}$$

Problem im Beispiel: $First(+ \; num \; Rest_E) \cap Follow(Rest_E) = \{ + \} \neq \varnothing$

LL(1) Charakterisierung

- Eine Grammatik G ist LL(1), falls sie für jedes Nonterminal A
- die First-Bedingung erfüllt
  - $A \to \sigma$,
  - $A \to \tau \in P$

  mit $\sigma, \tau \neq \varepsilon$ impliziert
  - $First(\sigma) \cap First(\tau) = \varnothing$
- die Follow-Bedingung erfüllt
  - $A \to \sigma$,
  - $A \to \varepsilon$

  mit $\sigma \neq \varepsilon$ impliziert
  - $First(\sigma) \cap Follow(A) = \varnothing$
- keine Linksrekursionen enthält
  - keine Regel der Form $A \to A\sigma$

Die Berechnung der Follow-Mengen

Algorithmus für Follow

Initialisierung:

Follow(S) = { eof }

Follow(A) = { } für alle A≠S.

DO

Loop

Für jede Produktion $A \to \alpha B \beta$

Follow(B) $\cup=$ First(β)-{ε}

Für jede Produktion $A \to \alpha B \beta$ mit $\varepsilon \in$ First(β)

Follow(B) $\cup=$ Follow(A)

BIS sich nichts mehr ändert

Beachte

First(α) ist für jede Satzform α definiert,
Follow(A) nur für Variablen.

Beispiel

$$\begin{aligned} String &\rightarrow Expr \\ &\mid String + String \\ &\mid \varepsilon \end{aligned}$$

$$\begin{aligned} Expr &\rightarrow num\ Rest_E \\ Rest_E &\rightarrow + num\ Rest_E \\ &\mid \varepsilon \end{aligned}$$

- First-Mengen der rechten Seiten
  - First(Expr)={num}
  - First(String + String } = { num, + }
  - First(num $Rest_E$) = { num }
  - First(ε) = {ε}
- Follow-Mengen der Nonterminale
  - Follow(String) ={ eof, +, }
  - Follow(Expr) ={ eof, + }
  - Follow($Rest_E$) ={ eof, + }
- First(Expr) ∩ First(String+String) ≠ ∅
- First(String + String) ∩ Follow(String) ≠ ∅

Linksrekursion

Direkte Übersetzung einer Grammatik in (rekursiven) RD-Parser kann zu Endlosschleifen führen:

```
Expr → Expr + Expr
Expr → id
Expr → num
```

Hier würde `parseExpr` sofort wieder `parseExpr` aufrufen ohne einen Fortschritt zu machen.

Problem lösbar durch äquivalente Umformungen der Grammatik :

```
Expr → T Rest

Rest → + T Rest
     | ε

T    → id | num
```

Elimination von Linksrekursionen

Angenommen die Produktionen für Nonterminal A sind

$$
\begin{aligned}
A &\rightarrow A\ \alpha_1 \\
&\ldots \\
A &\rightarrow A\ \alpha_n \\
\\
A &\rightarrow \beta_1 \\
&\ldots \\
A &\rightarrow \beta_n
\end{aligned}
$$

wobei die α_i, β_i nicht mit A beginnen. Man kann sie ersetzen durch:

$$
\begin{aligned}
A &\rightarrow \beta_1\ Rest_A \\
&\ldots \\
A &\rightarrow \beta_n\ Rest_A
\end{aligned}
$$

$$
\begin{aligned}
Rest_A &\rightarrow \alpha_1\ Rest_A \\
&\ldots \\
Rest_A &\rightarrow \alpha_n\ Rest_A \\
\\
Rest_A &\rightarrow \varepsilon
\end{aligned}
$$

LL(k) Sprachen

- Verallgemeinerung: k token lookahead.
- Sprache L ist LL(k), falls LL(k)-Grammatik für L existiert.
- Falls Grammatik nicht LL(k), versuche in äquiv. LL(k)-Grammatik zu transformieren.

Grammatik ist nicht LL(1),

Sprache ist LL(1) !!

G_1

Expr → Expr + Expr
| Expr * Expr
| (Expr)
| id
| num

In einer Reihe von Schritten werden wir diese Grammatik in eine äquivalente LL(1) Grammatik umformen.

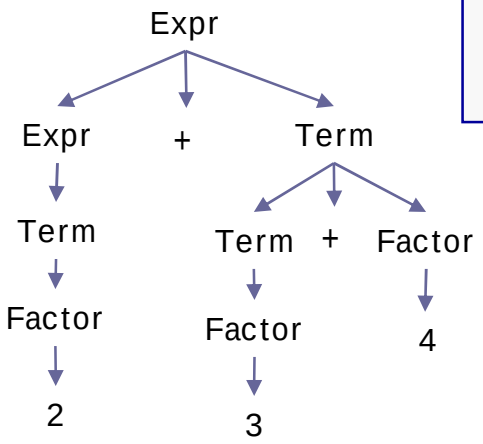
Von der Grammatik zum RD-Parser

Entferne Mehrdeutigkeit durch Einführung von getrennten Nonterminalen für jede Präzedenzstufe

G_1

Expr → Expr + Expr
| Expr * Expr
| (Expr)
| id
| num

2 + 3 * 4 hat zwei Herleitungsbäume



G_2

Expr → Expr + Term
| Term

Term → Term * Factor
| Factor

Factor → (Expr)
| id
| num

2 + 3 * 4 hat eindeutigen Herleitungsbaum

Von der Grammatik zum RD-Parser

- Linksrekursionen entfernen

G_2

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} + \text{Term} \\
&\mid \text{Term} \\
\text{Term} &\rightarrow \text{Term} * \text{Factor} \\
&\mid \text{Factor} \\
\text{Factor} &\rightarrow (\text{Expr}) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term}\ \text{Rest}_E \\
\text{Rest}_E &\rightarrow + \text{Term}\ \text{Rest}_E \\
&\mid \varepsilon \\
\text{Term} &\rightarrow \text{Factor}\ \text{Rest}_T \\
\text{Rest}_T &\rightarrow * \text{Factor}\ \text{Rest}_T \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow (\text{Expr}) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

Von der Grammatik zum RD-Parser

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term Rest}_E \\
\text{Rest}_E &\rightarrow + \text{ Term Rest}_E \\
&\mid \varepsilon \\
\text{Term} &\rightarrow \text{Factor Rest}_T \\
\text{Rest}_T &\rightarrow * \text{ Factor Rest}_T \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow (\text{ Expr }) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

First-Mengen berechnen:

$$
\begin{aligned}
&\text{First}(\text{ Term Rest}_E) = \{ (, \text{id}, \text{num} \} \\
&\text{First}(+ \text{ Term Rest}_E) = \{ + \} \\
&\text{First}(\text{ Rest}_E) = \{ +, \varepsilon \} \\
&\text{First}(\text{ Factor Rest}_T) = \{ (, \text{id}, \text{num} \} \\
&\text{First}(* \text{ Factor Rest}_T) = \{ * \} \\
&\text{First}(\text{ Rest}_T) = \{ *, \varepsilon \} \\
&\text{First}((\text{ Expr })) = \{ (\} \\
&\text{First}(\text{ id }) = \{ \text{id} \} \\
&\text{First}(\text{ num }) = \{ \text{num} \}
\end{aligned}
$$

Von der Grammatik zum RD-Parser

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term Rest}_E \\
\text{Rest}_E &\rightarrow + \text{ Term Rest}_E \\
&\mid \varepsilon \\
\text{Term} &\rightarrow \text{Factor Rest}_T \\
\text{Rest}_T &\rightarrow * \text{ Factor Rest}_T \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow (\text{ Expr }) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

Follow-Mengen berechnen:

$$
\begin{aligned}
\text{Follow}(\text{Expr}) &= \{), \text{eof} \} \\
\text{Follow}(\text{Rest}_E) &= \{), \text{eof} \} \\
\text{Follow}(\text{Term}) &= \{ +,), \text{eof} \} \\
\text{Follow}(\text{Rest}_T) &= \{ +,), \text{eof} \} \\
\text{Follow}(\text{Factor}) &= \{ *, +,), \text{eof} \}
\end{aligned}
$$

Von der Grammatik zum RD-Parser

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term Rest}_E \\
\text{Rest}_E &\rightarrow + \text{Term Rest}_E \\
&\mid \varepsilon \\
\text{Term} &\rightarrow \text{Factor Rest}_T \\
\text{Rest}_T &\rightarrow * \text{Factor Rest}_T \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow (\text{Expr}) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

Überprüfen der First Bedingungen:

$$
\begin{aligned}
\text{First}(+ \text{Term Rest}_E) \cap \text{First}(\varepsilon) &= \varnothing \\
\text{First}(* \text{Factor Rest}_T) \cap \text{First}(\varepsilon) &= \varnothing \\
\text{First}((\text{Expr})) \cap \text{First}(\text{id}) &= \varnothing \\
\text{First}((\text{Expr})) \cap \text{First}(\text{num}) &= \varnothing \\
\text{First}(\text{id}) \cap \text{First}(\text{num}) &= \varnothing
\end{aligned}
$$

Überprüfen der Follow Bedingungen:

$$
\begin{aligned}
\text{First}(+ \text{Term Rest}_E) \cap \text{Follow}(\text{Rest}_E) &= \varnothing \\
\text{First}(* \text{Factor Rest}_T) \cap \text{Follow}(\text{Rest}_T) &= \varnothing
\end{aligned}
$$

Von der Grammatik zum RD-Parser

- Da die LL(1) Bedingungen erfüllt sind, ist die Transformation in einen RD-Parser praktisch automatisch:

G_3

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term Rest}_E \\
\text{Rest}_E &\rightarrow + \text{ Term Rest}_E \\
&\mid \varepsilon \\
\text{Term} &\rightarrow \text{Factor Rest}_T \\
\text{Rest}_T &\rightarrow * \text{ Factor Rest}_T \\
&\mid \varepsilon \\
\text{Factor} &\rightarrow (\text{ Expr }) \\
&\mid \text{id} \\
&\mid \text{num}
\end{aligned}
$$

/* parser */

```
void parseExpr(){
  parseTerm();
  parseRestE();
}

void parseRestE(){
switch(lookahead)
 case PLUS :
        consume(PLUS);
        parseTerm();
        parseRestE();
        return;
 default: return;
}
```

... etc.

Semantische Aktionen

Parser soll Sprache nicht nur entscheiden,
sondern z.B. auch

- Wert des Ausdrucks berechnen
- Ausdruck “pretty-print”en
- Ausdruck compilieren.

Wichtige Voraussetzung

- Resultat für Gesamtausdruck ist Kombination von Resultaten für Teilausdrücke
- denotationelles Prinzip

Vorgehensweise

- Führe an passender Stelle Aktionen ein
- z.B. Programmcode
- in der Ausgangsgrammatik
- Transformiere Grammatik
- Behandle Aktionen in der Transformation wie Terminale
- ignoriere sie bei First- und Follow-Bestimmung

Beispiel: Infix-Postfix Übersetzer

- Aufgabe:
  - Infix Ausdrücke lesen,
  - in Postfix übersetzen.

G_1

```
Expr  → Expr + Expr
      | Expr * Expr
      | ( Expr )
      | id
      | num
```

Gehe davon aus, dass jedes Nonterminal auf der rechten Seite bereits seine Aktion ausführt

$G_{postfix}$

```
Expr  → Expr + Expr    write("+");
      | Expr * Expr    write("*");
      | ( Expr )       // Nix
      | id             write("id");
      | num            write("num");
```

Für (Expr) keine Aktion notwendig:
- Expr führt seine Aktion selber aus
- Postfix braucht keine Klammern

Ein Infix-Postfix Übersetzer

Mehrdeutigkeit entfernen durch Präzedenzstufen

$G_{postfix}$

```
Expr  → Expr + Expr    write("+");
      | Expr * Expr    write("*");
      | ( Expr )
      | id             write("id");
      | num            write("num");
```

Man hätte auch gleich hiermit beginnen können:

G_2

```
Expr   → Expr + Term     write("+");
       | Term

Term   → Term * Factor   write("*");
       | Factor

Factor → ( Expr )
       | id              write("id");
       | num             write("num");
```

Ein Infix-Postfix Übersetzer

Entferne Linksrekursion. Behandle dabei Aktionen wie Terminale.

G_2

```
Expr   → Expr + Term  write("+");
       | Term

Term   → Term * Factor  write("*");
       | Factor

Factor → ( Expr )
       | id      write("id");
       | num     write("num");
```

G_3

```
Expr    → Term Rest_E

Rest_E  → + Term  { write("+"); }  Rest_E
        | ε

Term    → Factor Rest_T

Rest_T  → * Factor { write("*"); }  Rest_T
        | ε

Factor  → ( Expr )
        | id      { write("id"); }
        | num     { write("num"); }
```

Ein Infix-Postfix Übersetzer

Prüfe LL(1) Bedingungen. Ignoriere dabei die Aktionen

Transformiere in Programm:

G_3

$Expr \rightarrow Term\ Rest_E$

$Rest_E \rightarrow +\ Term\ \{ write(\text{“+“}); \}\ Rest_E \mid \varepsilon$

$Term \rightarrow Factor\ Rest_T$

$Rest_T \rightarrow *\ Factor\ \{ write(\text{“*“}); \}\ Rest_T \mid \varepsilon$

$Factor \rightarrow (\ Expr\ ) \mid id\ \{ write(\text{“id“}); \} \mid num\ \{ write(\text{“num“}); \}$

/* Postfix--Übersetzer */

```
void parseExpr(){
  parseTerm();
  parseRestE();
}

void parseRestE(){
switch(lookahead)
 case PLUS :
       consume(PLUS);
       parseTerm();
       write(“+“);
       parseRestE();
       return;
 default: return;
}
```

... etc.

Inhalt

1. Grammatiken und Sprachen
   - Kontextfreie Grammatiken
   - Herleitungen, Linksherleitungen
   - Sprachen zu einer Grammatik
   - Äquivalenz
   - Chomsky-Normalform
   - Wortproblem, CYK
   - Pumping Lemma für CF-Sprachen

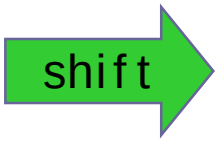
2. Stackautomaten
   - Definitionen und Beispiele
   - Konfigurationen, Läufe,
   - Sprache eines Stackautomaten
   - Parser

3. Parser
   - Mehrdeutigkeit
   - Bottom Up, Top Down
   - Recursive descent Parser
   - First, Follow
   - Semantische Aktionen

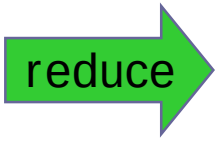
4. Shift-Reduce Parser
   - Konflikte
   - LR(0)-Zustände
   - Automat zur Grammatik
   - Shift-Reduce, Reduce-Reduce Konflikte
   - Präzedenz und Assoziativität
   - lex und yacc
   - Arbeitsweise eines Compilers

Shift-Reduce Parser

Shift-Reduce Parser sind "bottom-up parser", d.h. sie bauen den Syntaxbaum von den Blättern her auf. Aktionen sind :



Lese ein weiteres Zeichen aus dem Input



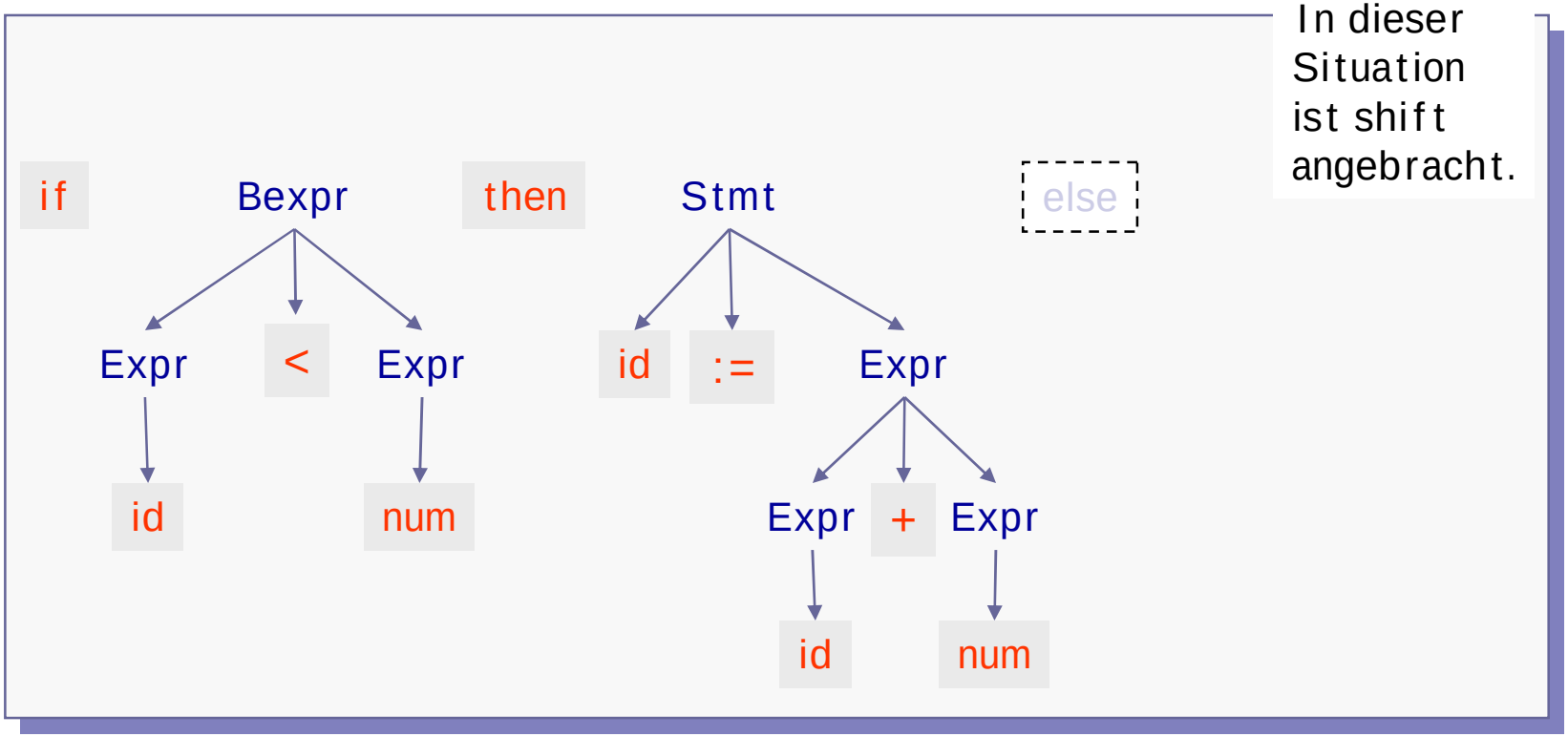
Die rechte Seite einer Produktion wurde im Input erkannt. Ersetze sie durch die linke Seite.

Input kann dabei eine beliebige Satzform sein.

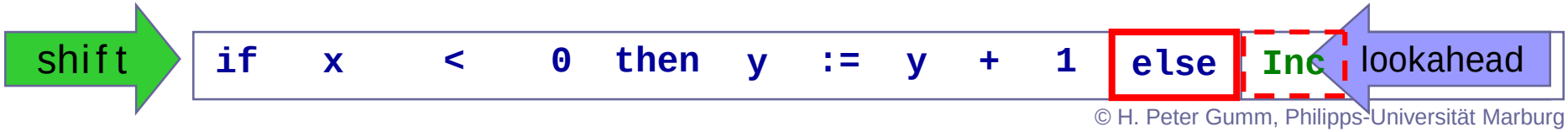
Shift-Schritt

lookahead einlesen
Scanner legt nächstes Token in lookahead

Stmt → if Bexpr then Stmt else Stmt
Stmt → if Bexpr then Stmt
Stmt → id (Expr)



In dieser Situation ist shift angebracht.

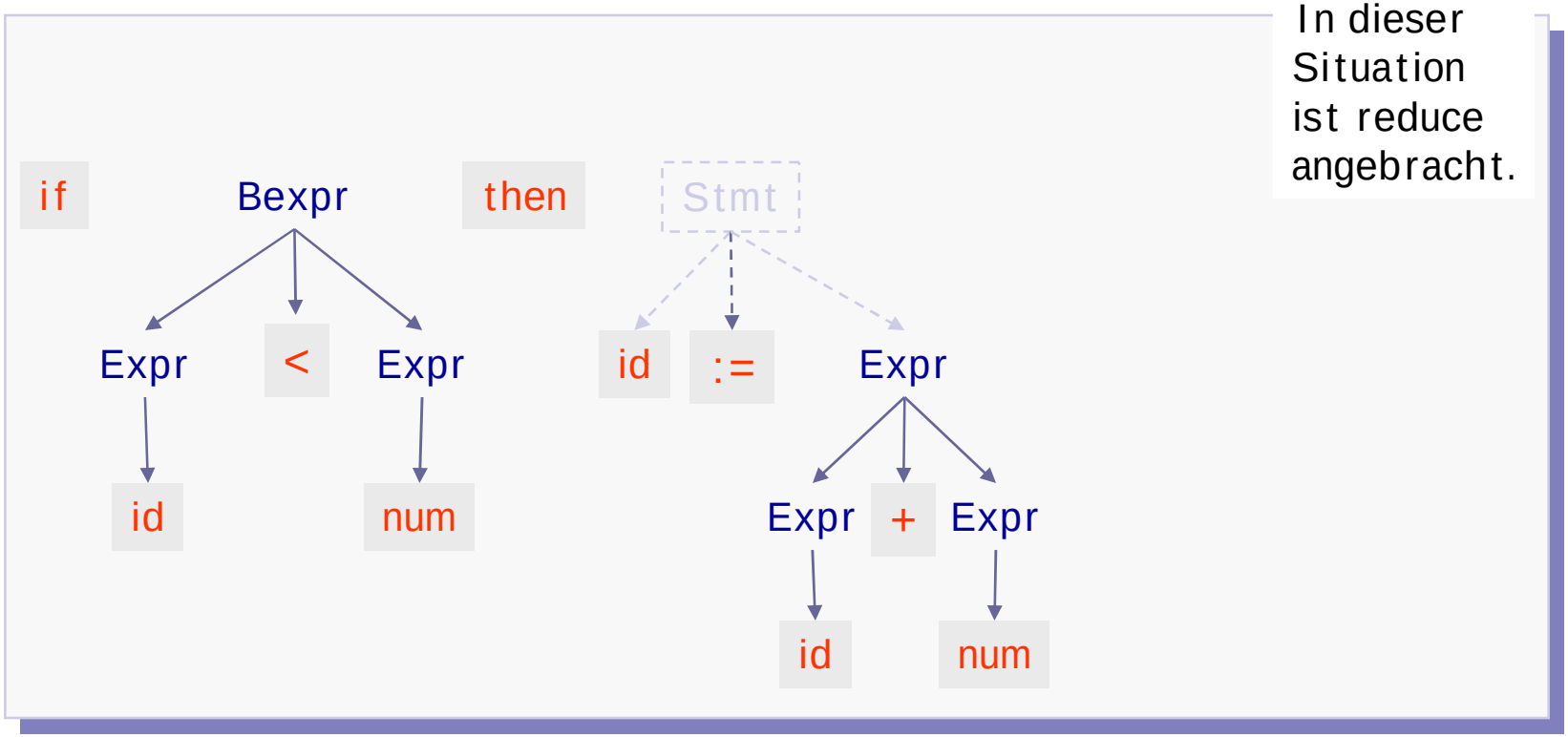


Reduce-Schritt

Regel auswählen, deren rechte Seite auf die letzten Bäume im Wald passt

reduce

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)



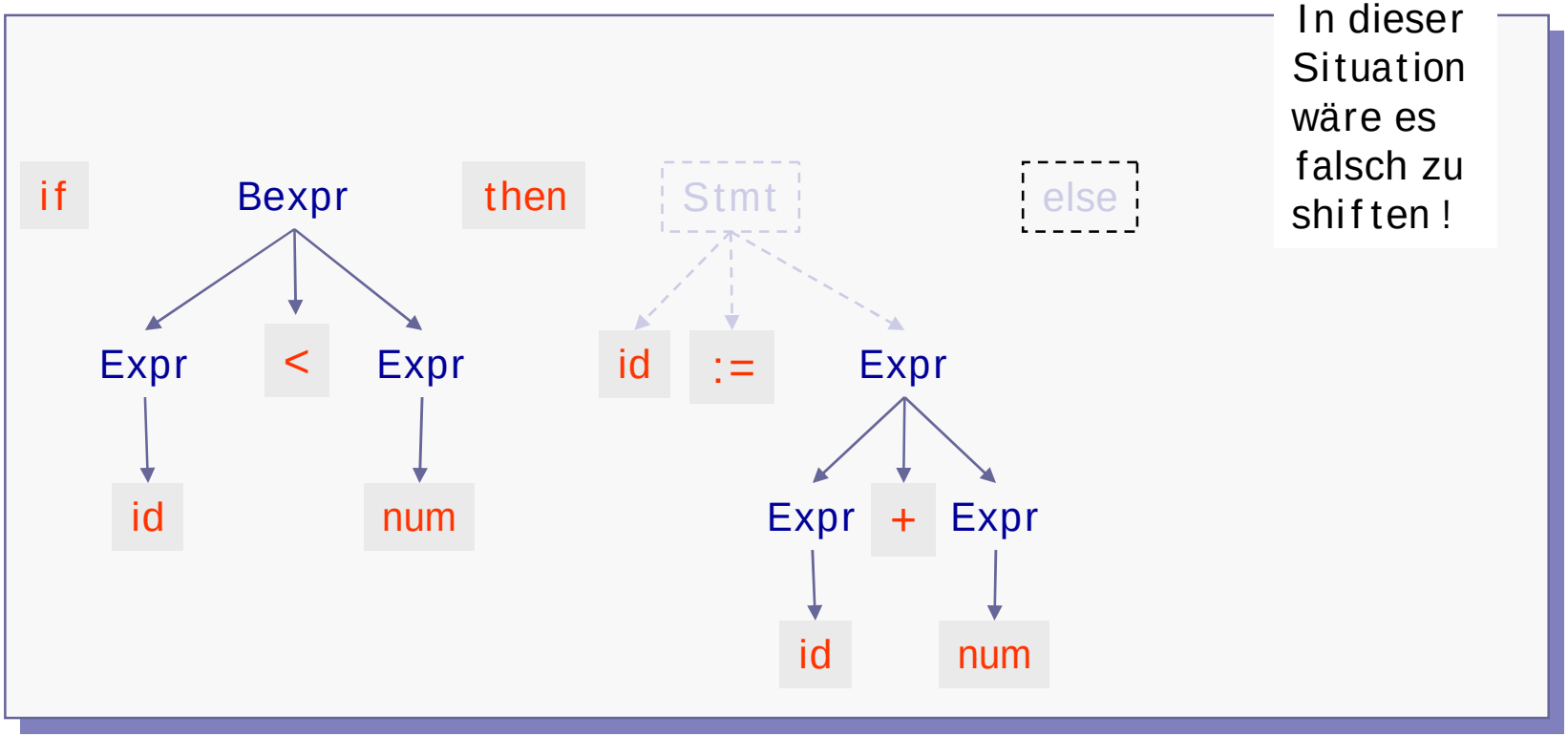
In dieser Situation ist reduce angebracht.

if	x	<	0	then	y	:=	y	+	1	else

lookahead

Shift oder reduce ?

Stmt → if Bexpr then Stmt else Stmt
Stmt → id := Expr
Stmt → id (Expr)



In dieser Situation wäre es falsch zu shiften !

shift

```
if  x  <  0  then  y  :=  y  +  1  else
```

lookahead

Reduce-reduce-Konflikt

reduce

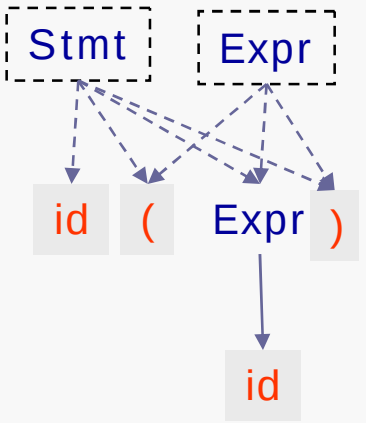
Stmt → id (Expr)

Expr → (Expr)

- Es gibt zwei Regeln, mit denen reduziert werden könnte

- Das lookahead bringt keine Entscheidung

Reduce-Reduce-Konflikte sind unangenehm. Sie werden oft durch Einführung von Präzedenzen entschieden




```
if  x  <  0  then  y  :=  y  +  1  else  Inc  (  z  )
```

Shift-Reduce-Konflikt

Stmt → if Bexpr then Stmt else Stmt

reduce

Stmt → if Bexpr then Stmt

Es könnte mit der zweiten Regel reduziert werden

Die Existenz der ersten Regel lässt auch ein shift zu, da noch ein Stmt folgen kann.

Das lookahead bringt keine Entscheidung

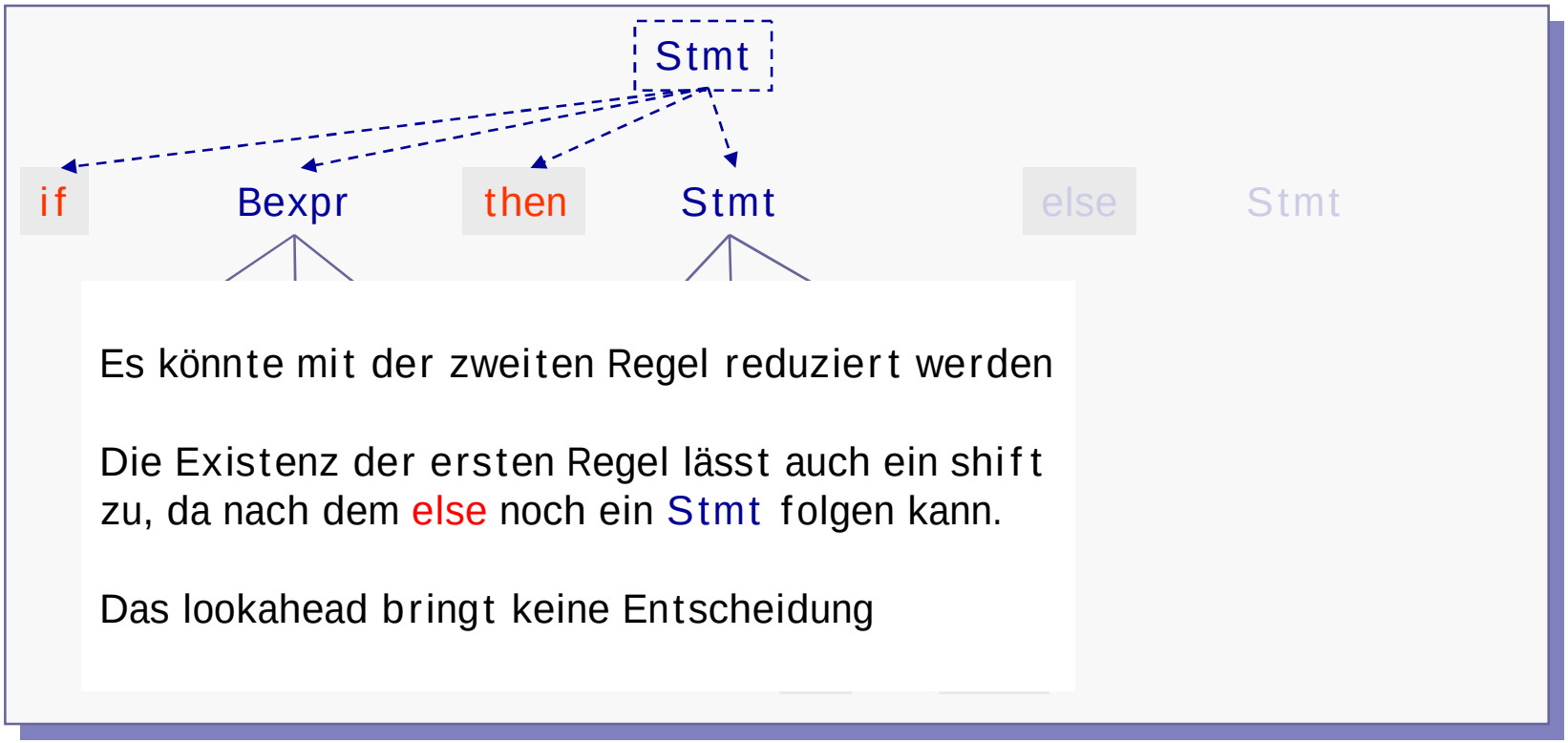
```
if  x  <  0  then  y  :=  y  +  1  else
```

Shift-Reduce-Konflikt

Stmt → if Bexpr then Stmt else Stmt

reduce

Stmt → if Bexpr then Stmt



Es könnte mit der zweiten Regel reduziert werden

Die Existenz der ersten Regel lässt auch ein shift zu, da nach dem else noch ein Stmt folgen kann.

Das lookahead bringt keine Entscheidung

```
if  x  <  0  then  y  :=  y  +  1  else
```

Zustände

- Konflikte können nicht allein aufgrund des lookahead entschieden werden.
- Parser merkt sich bisher nicht, was er gesehen hat
- Lösung: Zustände einführen, die besagen, was der Parser gerade zu erkennen versucht.

G

Expr	→ Expr + Term	// 1
	| Term	// 2

Bisher erkannt:	lookahead	Aktion:
... THEN id := Term	+	red(2)
... THEN id := Expr + Term	+	red(1)
... THEN id := Expr	+	shift

Merke die lokale Situation. Notation :

Expr → Term • : wollte Expr sehen, habe Term erkannt

Expr → Expr • + Term : wollte Expr sehen, habe Expr erkannt
jetzt muss + kommen, dann ein Term

Zustand:	lookahead	Aktion:
Expr → Term •	+	red (2)
Expr → Expr + Term •	+	red (1)
Expr → Expr • + Term	+	shift

LR(0) - items

Ein item ist eine Produktion $A \rightarrow \alpha$ mit einer markierten Position in α.

Zustand beinhaltet also

- Nonterminal, das der Parser gerade erkennen will
- mit welcher Regel er dies versucht und
- wieviel von der rechten Seite der Regel er bereits erkannt hat.

Beispiel: Die Produktion $Expr \rightarrow Expr + Term$ liefert genau 4 items mit folgenden Bedeutungen:

	Bei dem Versuch ein Expr zu finden ...
$Expr \rightarrow \bullet\ Expr + Term$	... erwarte $Expr + Term$
$Expr \rightarrow Expr \bullet + Term$	... Expr gesehen, erwarte $+ Term$
$Expr \rightarrow Expr + \bullet\ Term$	... $Expr +$ gesehen, erwarte $Term$
$Expr \rightarrow Expr + Term \bullet$	... $Expr + Term$ gesehen

LR - steht für Left-Right, gemeint ist die Abarbeitung des Inputs von Links nach rechts

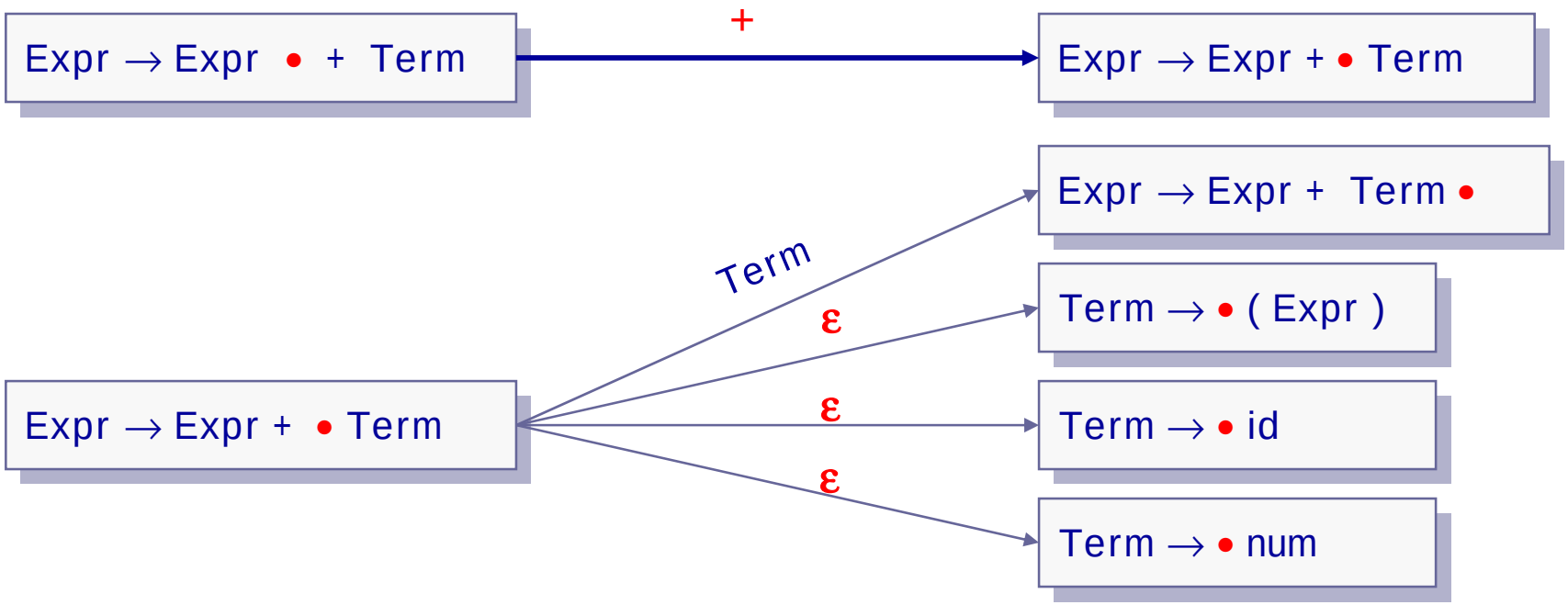
Automat einer Grammatik

- Ein item drückt den jeweiligen Zustand des Parsers beim Erkennen eines Nonterminals aus.
- Mit dem Lesen des nächsten Tokens geht der Parser in den nächsten Zustand über. Es gibt zwei mögliche Übergänge :

G_{simpl}

Expr → Expr + Term
| Term

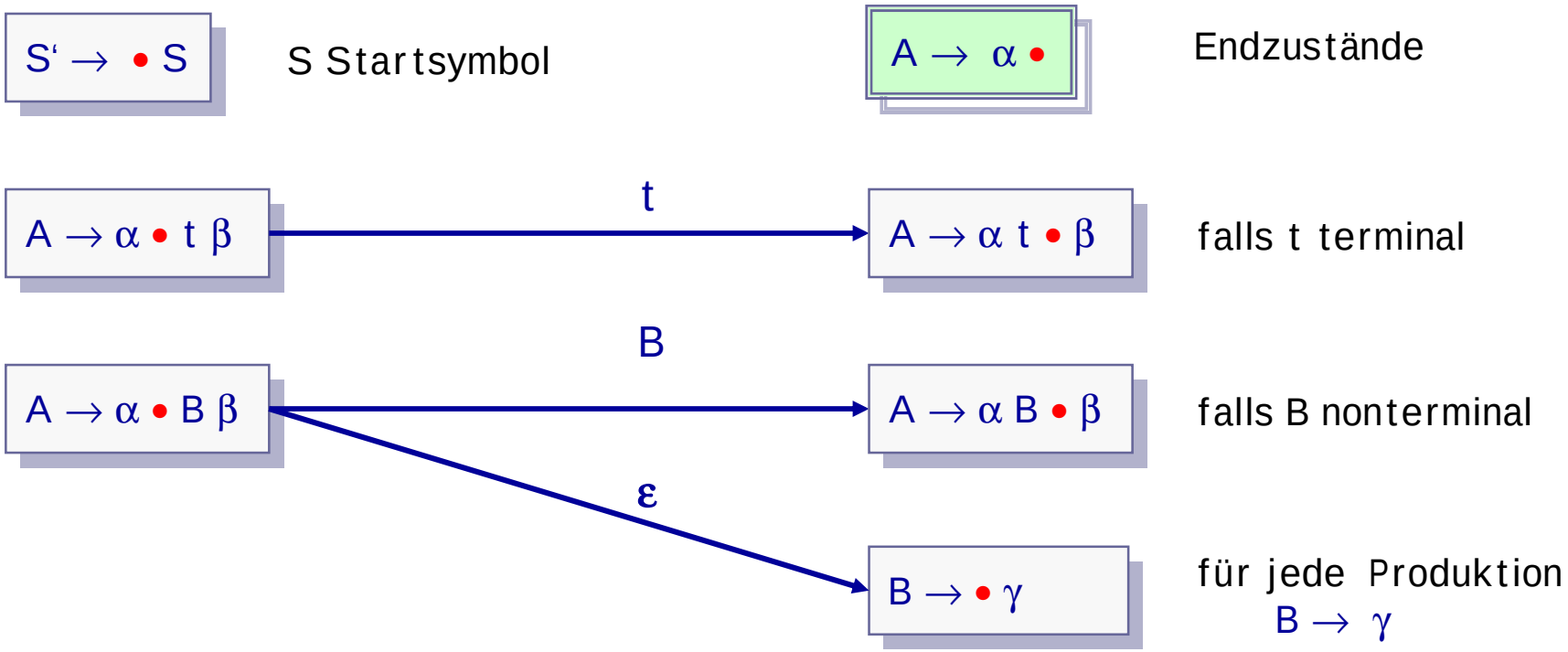
Term → (Expr)
| id
| num



Dies definiert einen nichtdeterministischen Automaten: N(G).

Automat einer Grammatik

- Allgemeine Definition



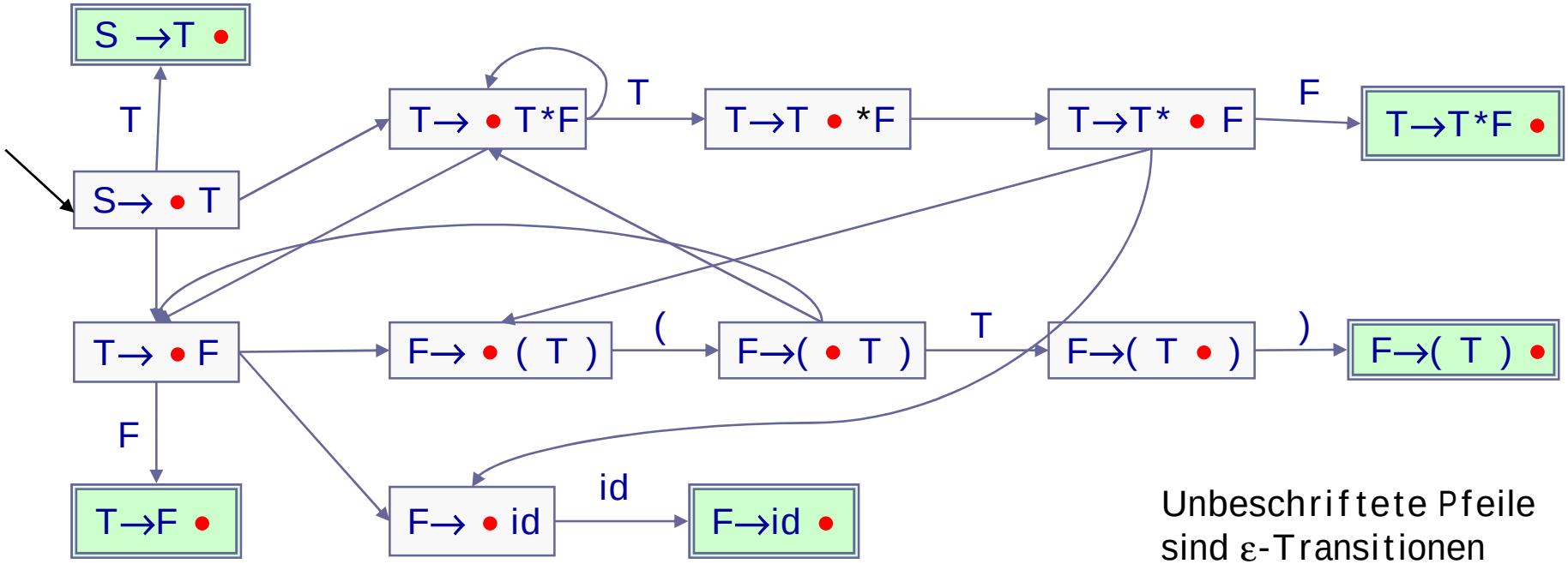
N(G) : Automat zu einer Grammatik G

Definiere NFA mit den items als Zuständen, den Terminalen und Nonterminalen der Grammatik als Alphabet :

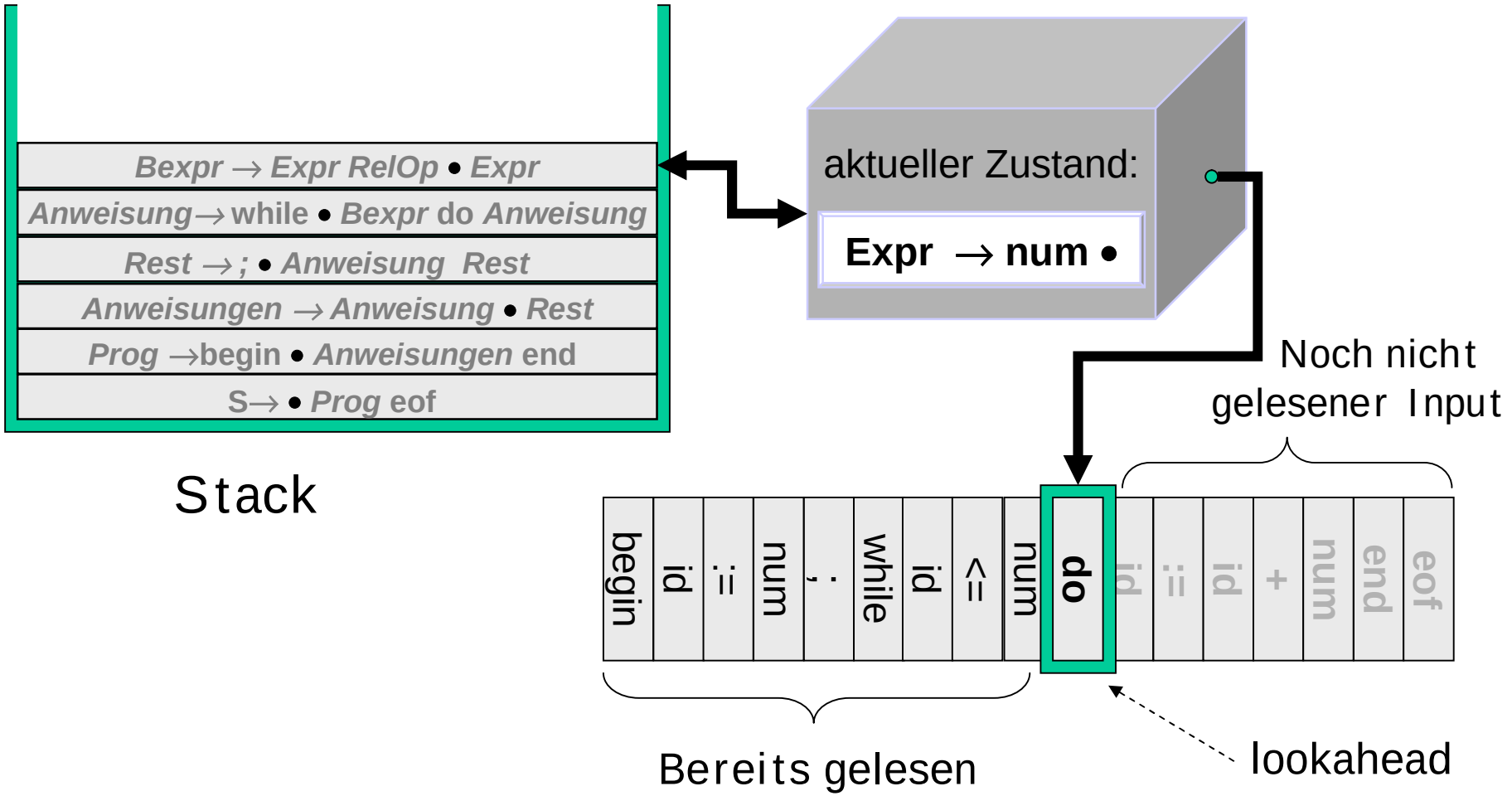
- S = Menge aller items
- $\Sigma = V \cup T$
- $s_0 = \bullet S$
- $T = \{ X \to \beta \bullet \mid X \to \beta \in P \}$

G

$S \to T$
$T \to T * F \mid F$
$F \to (T) \mid id$

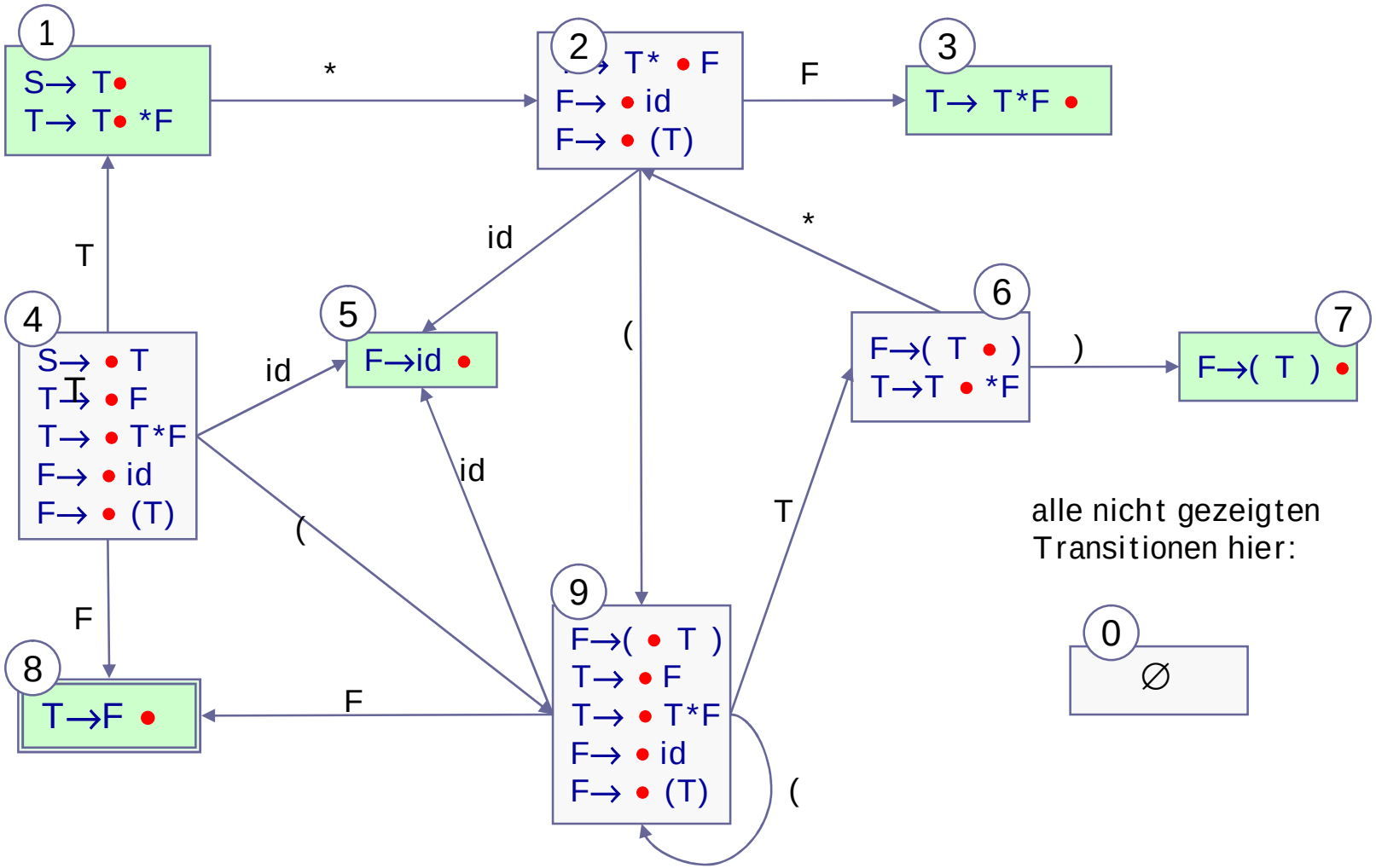


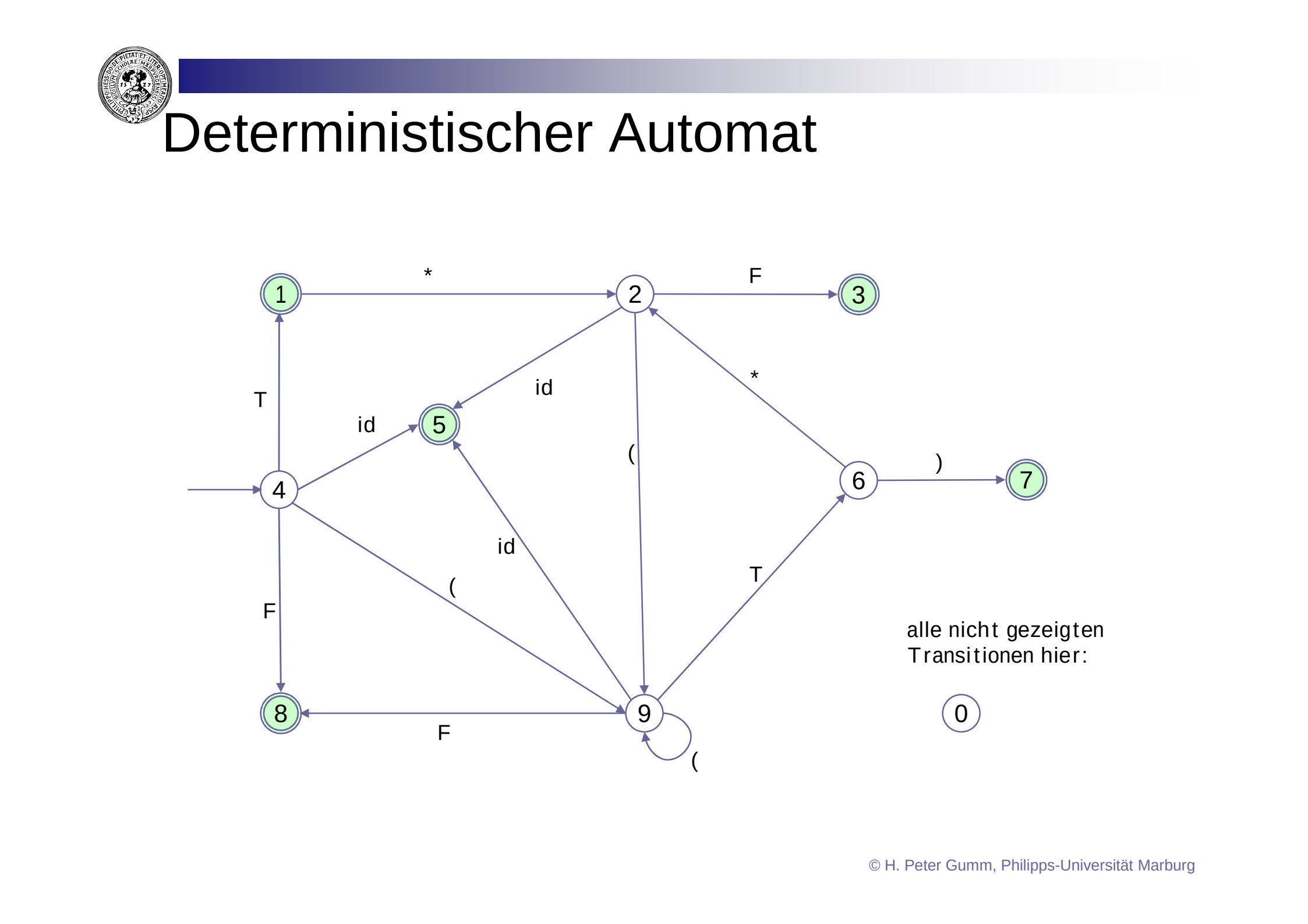
Unbeschriftete Pfeile sind ε-Transitionen



Nichtdeterministischer Stackautomat

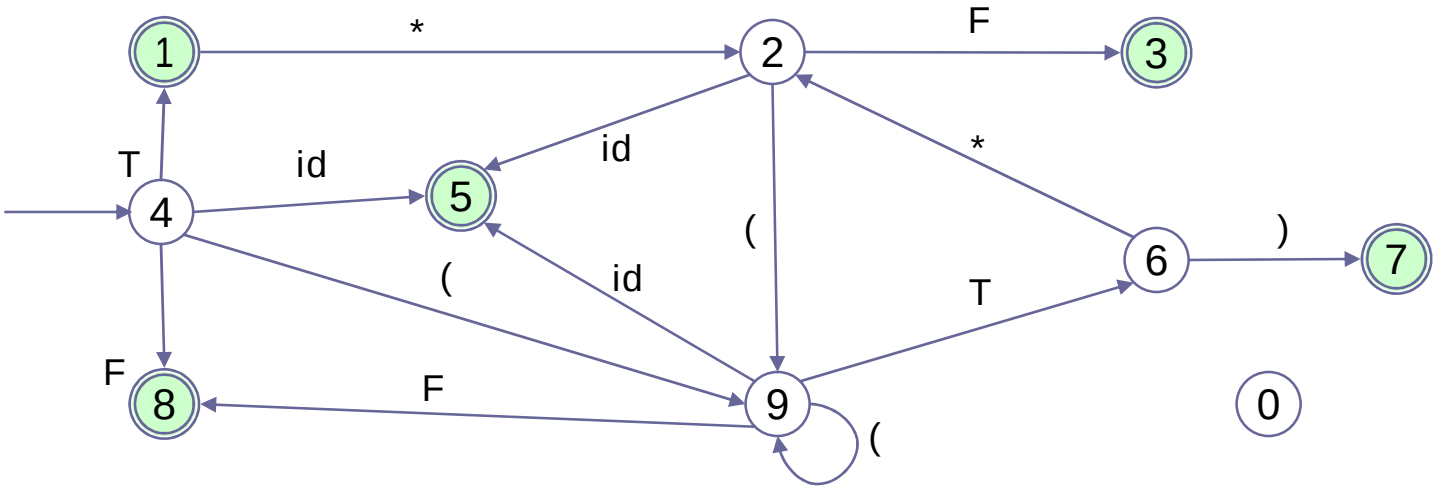
Deterministischer Automat





Deterministischer Automat

Deterministischer Automat

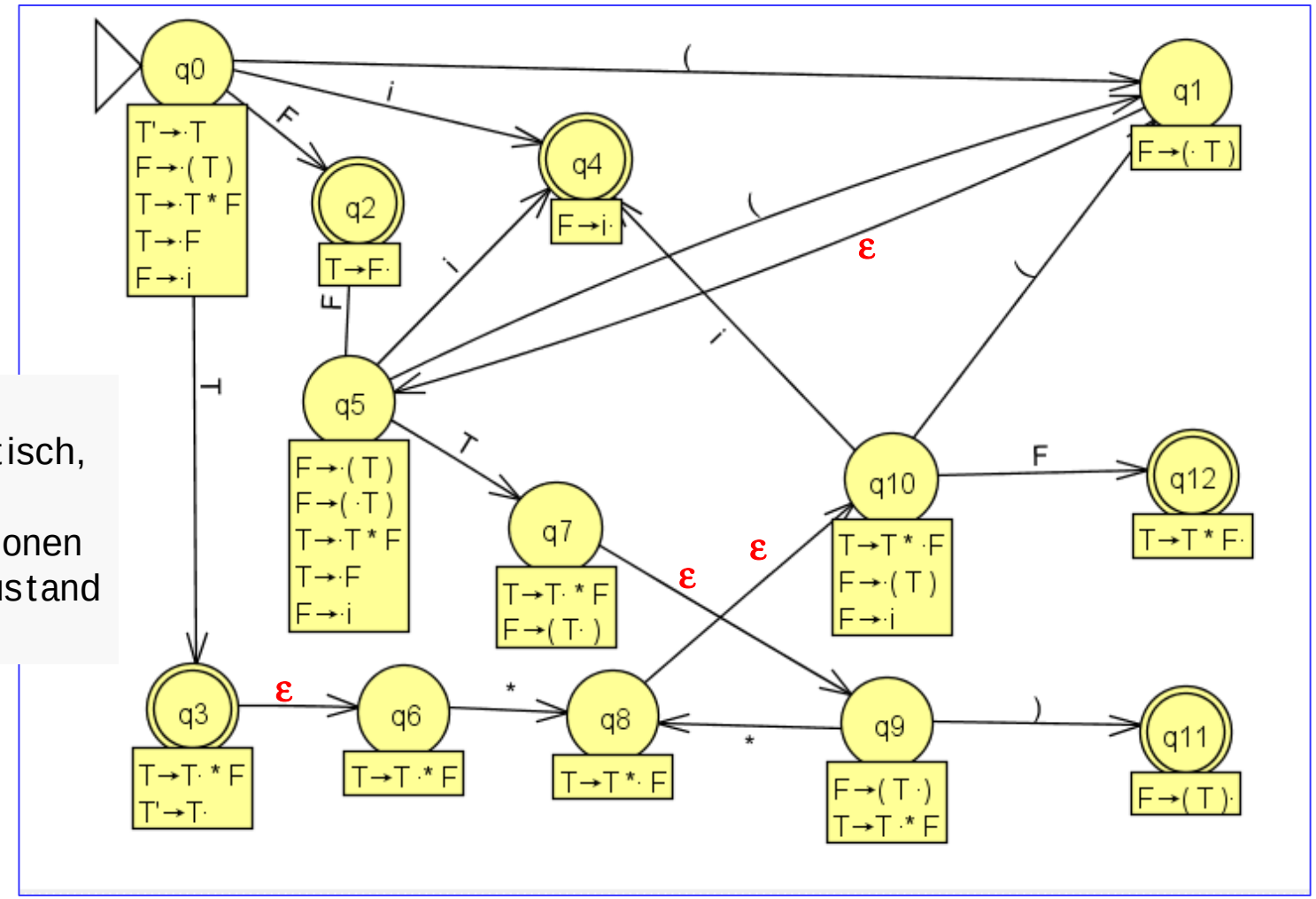


GoTo	T	F	id	*	(	)
1				2		
2		3	5			
4	1	8	5		9	
6				2		7
8						
9	6	8	5		9	

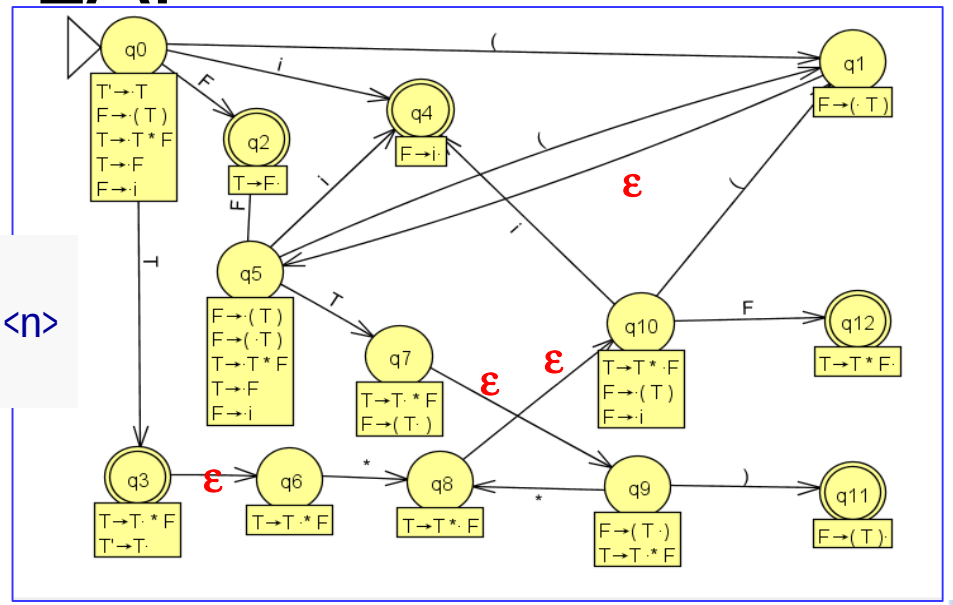
SLR-Automat in JFLAP erzeugt

SLR steht für Simple-LR

Der Automat ist nichtdeterministisch, weil er
-noch ε -Transitionen
-keinen Fehlerzustand enthält:



Parsertabelle in JFLAP



Einträge bedeuten:

s<n> : Shift und gehe in Zustand <n>

r<k> : Reduce mit Regel Nr. <k>

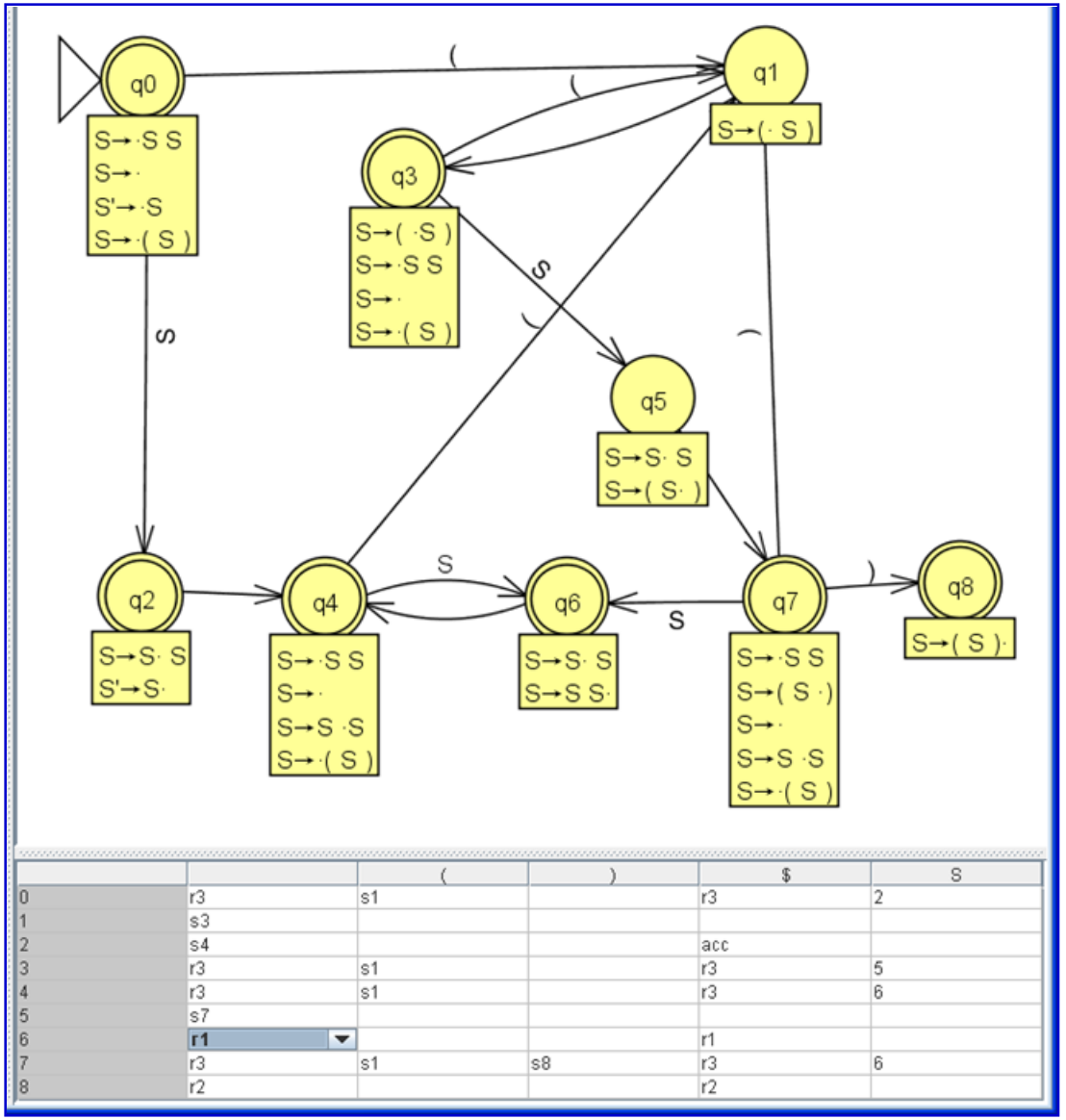
		(	)	*	i	$	F	T
0		s1			s4		2	3
1	s5							
2	r2					r2		
3	s6					acc		
4	r3					r3		
5		s1			s4		2	7
6				s8				
7	s9							
8	s10							
9			s11	s8				
10		s1			s4		12	
11	r4					r4		
12	r1					r1		

SLR-Automat für Klammergrammatik

G_{klamm}

$S' \rightarrow S$

$S \rightarrow S\ S \mid (\ S\ ) \mid \varepsilon$



		(	)	$	S
0	r3	s1		r3	2
1	s3				
2	s4			acc	
3	r3	s1		r3	5
4	r3	s1		r3	6
5	s7				
6	r1			r1	
7	r3	s1	s8	r3	6
8	r2			r2	

Die Notwendigkeit eines Stacks

Angenommen der Parser ist in einem Zustand (mit dem item)

Term → (• Expr)

er muß nun also in einen Zustand springen, der das item

Expr → • Expr + Term

enthält. Nach dessen Abarbeitung kommt er in den Zustand :

Expr → Expr + Term •

Nun muß er aber zurückspringen in den Zustand

Term → (Expr •)

Offensichtlich muß bei jedem Sprung der alte Zustand auf einem Stack aufbewahrt werden und bei der Beendigung des items zurückgesprungen werden, verbunden mit einem POP des Stacks.

shift items / reduce items

Ein shift item hat die Form
$A \to \alpha \bullet u\ \gamma$, mit $u \in T$.

shift ≈ push

Drückt aus, daß ein u erwartet wird.

Im Parserzustand Y = { ... , $A \to \alpha \bullet u\ \gamma$ ... } mit lookahead u wird dieses konsumiert und der Folgezustand Z = Goto(Y,u) = { ... $A \to \alpha u \bullet \gamma$, } auf den Stack *gepusht*.

Ein reduce item hat die Form $A \to \alpha \bullet$

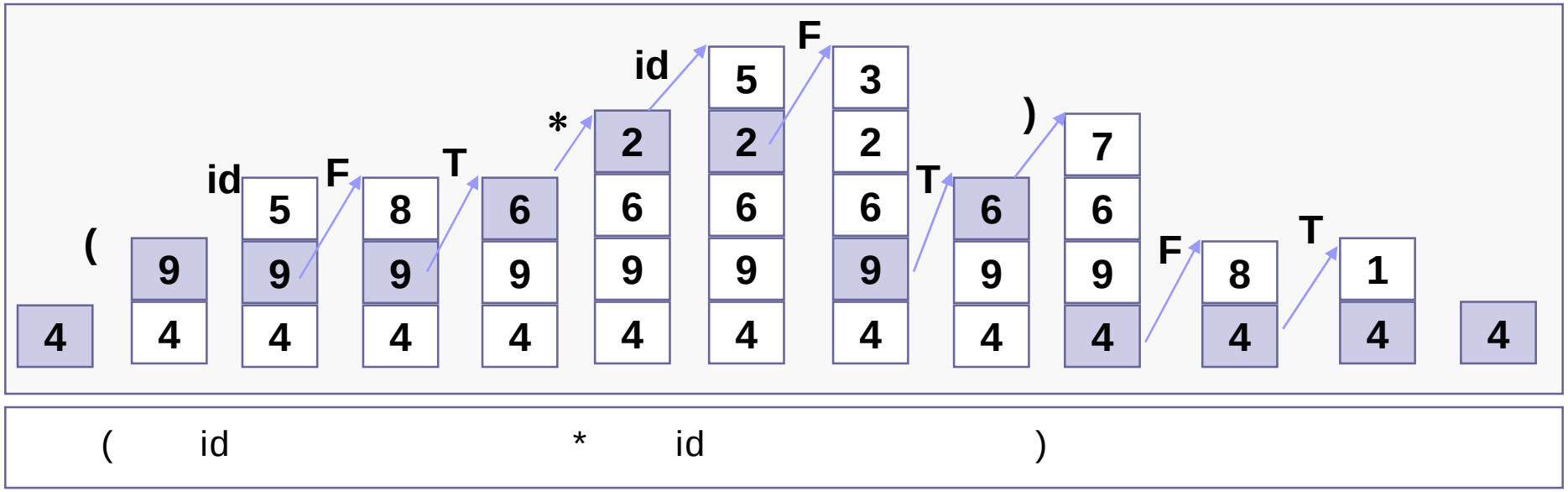
reduce ≈ pop

Drückt aus, daß erfolgreich ein A erkannt wurde.

Auf dem Stack sollte ein Zustand Y = { ... , $X \to \beta \bullet A\ \gamma$, ... } liegen ,

darüber | α | weitere Zustände. Wir *poppen* also |α | viele Zustände und ersetzen sie durch Goto(Y,A) = { ..., $X \to \beta \bullet A\ \gamma$, ... }.

Der Stack beim Parsen von (id * id)



GoTo	T	F	id	*	(	)
1				2		
2		3	5			
4	1	8	5		9	
6				2		7
8						
9	6	8	5		9	

1
S→ T•
T→ T• *F

2
T→ T* • F
F→ • id
F→ • (T)

3
T→ T*F •

4
S→ • T
T→ • F
T→ • T*F
F→ • id
F→ • (T)

5
F→id •

6
F→(T •)
T→T • *F

7
F→(T) •

8
T→F •

9
F→(• T)
T→ • F
T→ • T*F
F→ • id
F→ • (T)

Shift-Zustände - Reduce Zustände

Ein reduce-Zustand ist ein Zustand, der ein reduce item enthält.

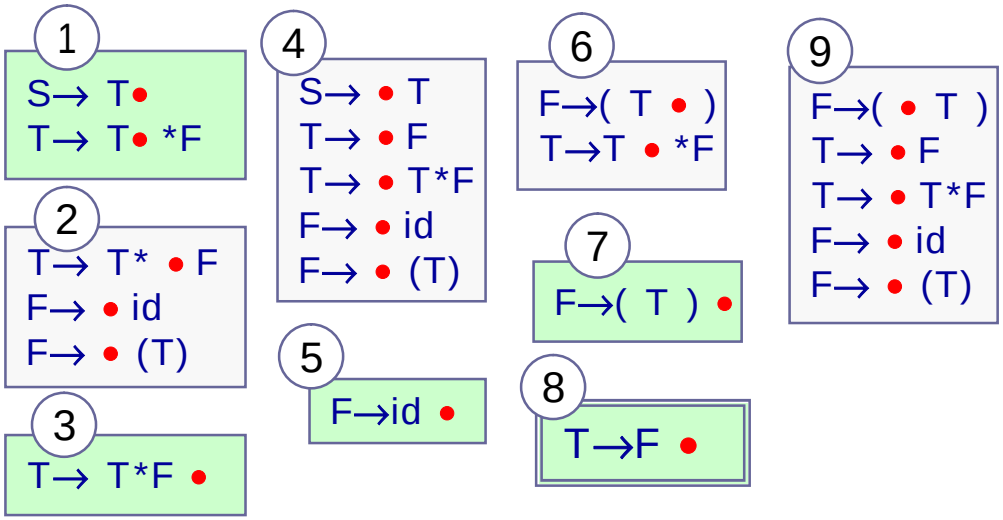
Ein shift-Zustand ist ein Zustand, der ein shift item enthält.

Ein accept-Zustand ist ein reduce-Zustand der Form $Start \rightarrow \gamma \bullet$

Shift Zustände :
1, 2, 4, 6, 9

Reduce Zustände :
1, 3, 5, 7, 8

Accept Zustand:
8



Falls $L(X) \neq \emptyset$ für alle $X \in V$ dann ist jeder nichtleere Zustand ein Shift-Zustand oder ein Reduce-Zustand.

Der Zustand, der der leeren Menge von items entspricht kann immer als Fehlerzustand interpretiert werden.

Shift-Reduce Konflikte

Enthält ein Zustand Z sowohl ein shift item als auch ein reduce item, so muß das lookahead entscheiden, was zu tun ist

$$Z = \{ \ldots, A \to \alpha \bullet u\, \gamma, \ \ldots \ , B \to \beta \bullet, \ldots \}$$

lookahead = u	$\Rightarrow$	shift,
Lookahead $\in$ Follow(B)	$\Rightarrow$	reduce

Falls $u \in$ Follow(B) gibt es einen *Shift-Reduce-Konflikt für u.*
Pragmatisch entscheidet man sich meist für Shift

Reduce-Reduce Konflikte

Enthält ein Zustand Z mehrere reduce items so müssen die Follow-Mengen entscheiden, was zu tun ist

$$Z = \{ \ldots , A \to \alpha \bullet , \ldots , B \to \beta \bullet, \ldots \}$$

lookahead $\in$ Follow(A)	$\Rightarrow$	reduce mit $A \to \alpha$
Lookahead $\in$ Follow(B)	$\Rightarrow$	reduce mit $B \to \beta$

Falls u$\in$ Follow(B) $\cap$ Follow(B) gibt es einen *Reduce-Reduce-Konflikt* für u.

Solche Konflikte sollte man versuchen zu vermeiden.
Notfalls die Grammatik ändern !

Syntaxfehler

Ein Syntaxfehler im Input ist beim LR-Parsen immer zum frühest-möglichen Zeitpunkt erkennbar :

Ein Syntaxfehler liegt vor, wenn in Zustand Z mit lookahead u

Z kein shift-item $A \to \alpha \bullet u\, \gamma$ enthält und
für alle Reduce items $B \to \beta \bullet$ in Z gilt : $u \notin$ Follow(B)

4

$S \to \bullet\, T$
$T \to \bullet\, F$
$T \to \bullet\, T * F$
$F \to \bullet\, id$
$F \to \bullet\, (\ T\ )$

id or (expected

1

$S \to T \bullet$
$T \to T \bullet * F$

* or end-of-file expected

Gute Fehlermeldungen in Recursive Descent Parser einzubauen ist viel schwieriger!

Präzedenz und Assoziativität

Sowohl Präzedenz, als auch durch Assoziativität bedingte Mehrdeutigkeit der Grammatik führt zu Shift-Reduce-Konflikten.

G

$S \rightarrow E$
$E \rightarrow E + E \mid E * E \mid (E) \mid id$

erzeugt u.a. die Zustände

Z_1

$E \rightarrow E + E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

shift-reduce Konflikt für + und für *.

Z_2

$E \rightarrow E * E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

shift-reduce Konflikt für + und für *.

Wie sind diese Konflikte zu lösen ?

Z_1

$E \rightarrow E + E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

Für * $\Rightarrow$ shift (bewirkt Präzedenz von * über +)
für + $\Rightarrow$ reduce (bewirkt Links-Klammerung von +)

Z_2

$E \rightarrow E * E \bullet$
$E \rightarrow E \bullet + E$
$E \rightarrow E \bullet * E$

Für * $\Rightarrow$ reduce (bewirkt Links-Klammerung von *)
für + $\Rightarrow$ reduce (garantiert Präzedenz von * über +)

LR(1) - items

LR Parsen ist mächtiger als Recursive Descent-Parsing.

Eine Grammatik heißt LR(0), falls es einen konfliktfreien SLR-Parser gibt.

Eine Sprache heißt LR(0), falls es eine LR(0) Grammatik für sie gibt.

Mächtigere Parsing-Methoden berücksichtigen noch lookaheads bei der Berechnung der Items:

Ein LR(1) item ist ein Paar $[A \to \alpha \bullet X\beta, a]$ aus einem LR(0) item und einem Terminalsymbol. Es gilt immer $a \in$ Follow(A)

Angefangen mit

$[\ Start \to \bullet\ S,\ eof\ ]$

konstruieren wir den NFA zur Grammatik, mit den Transitionen

$[\ A \to \alpha \bullet u\beta,\ a]$ —u→ $[\ A \to \alpha u \bullet \beta,\ a]$ falls u ein Terminal

$[\ A \to \alpha \bullet B\beta,\ a]$ —ε→ $[\ B \to \bullet\ \gamma,\ \mathbf{b}]]$ für jede Produktion $B \to \bullet\ \gamma$ und jedes $b \in$ First(βa).

LALR(1)-Parsing

- Tatsachen
  - LR(1) Parsing ist mächtiger als SLR-Parsing
  - LR(1) Parsing hat weit mehr items, also umfangreichere Parsertabelle
- Kompromiss besteht aus
  - Berechnung der LR(1) Zustände, danach
  - Verschmelzung zweier Zustände, wenn sie
    - die gleichen LR(0)-items, aber
    - evtl. verschiedene lookahead Mengen
    - besitzen
- Methode ist für praktische Zwecke mächtig genug
  - implementiert in yacc
    - yet another compiler-compiler

yacc

- Yacc : Yet another Compiler Compiler (Parser Generator). Aus kontextfreier Grammatik erzeuge LALR(1) Parser.
- C-Funktion `yyparse()`
- yacc und lex arbeiten als Team. lex erkennt die Token, während yacc für die darauf aufbauende Grammatik verantwortlich ist.

```
# yacc Input File für Baby Deutsch

%start Satz

%Token  ARTIKEL NAME HAUPTWORT AUX VERB PUNKT
%%
Satz    :       Subjekt Praedikat Objekt PUNKT
        |       Subjekt Praedikat PUNKT
        ;

Subjekt :       ARTIKEL HAUPTWORT
        |       NAME
        ;

Objekt  :       Subjekt  ;

Praedikat       :       AUX   VERB ;
                |       VERB
                |       error{“Verb erwartet“};
                ;
```

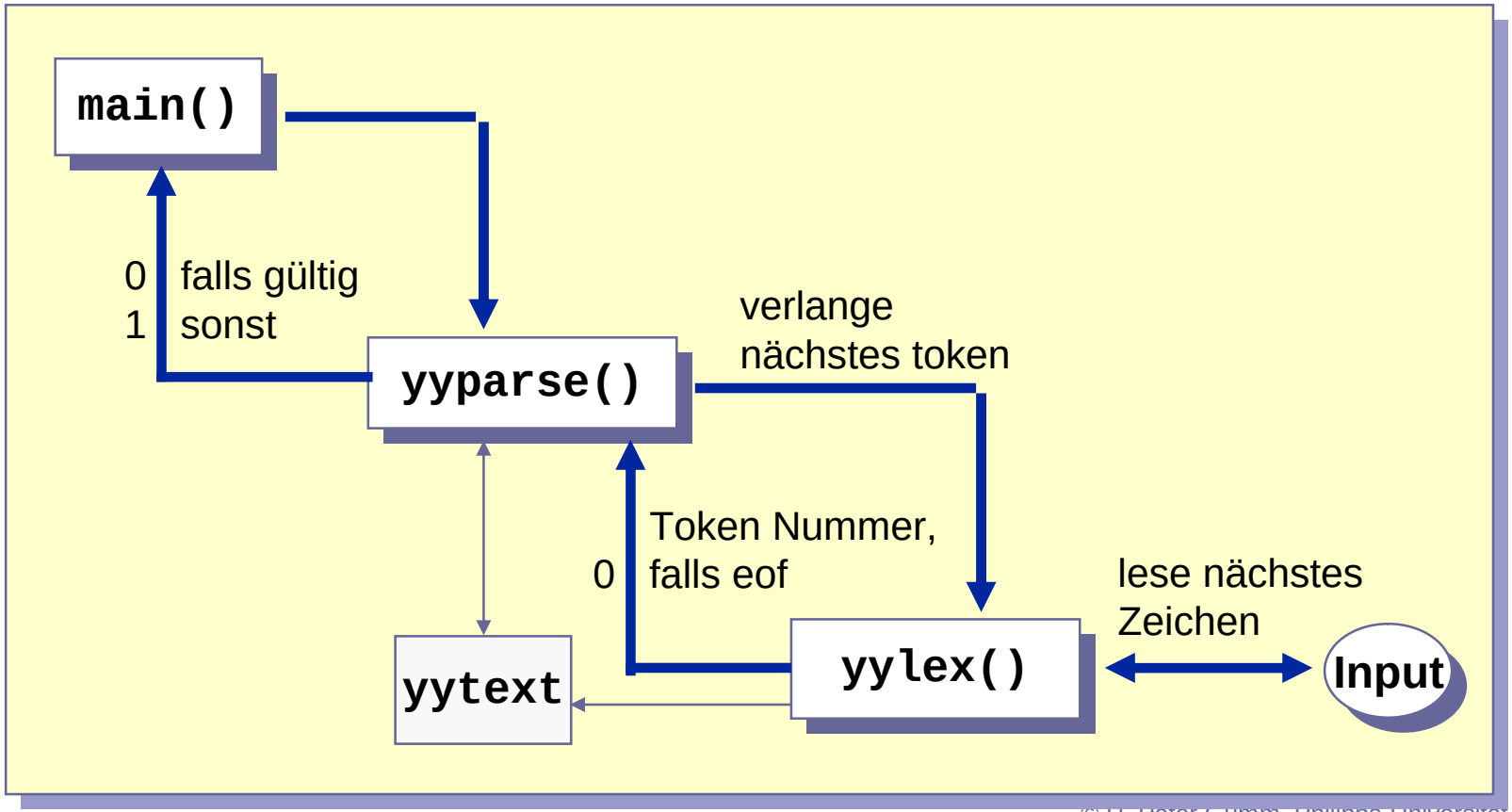
Startsymbol

Für diese Token ist lex verantwortlich

Aktionen wieTerminal-symbole behandeln

Das Duo : lex und yacc

- yacc erzeugt die Funktion yyparse(), lex die Funktion yylex().
- yylex() liefert die Nummer des Tokens. Der String aus dem dieses besteht ist immer in der (globalen) Variablen yytext vorhanden.



Beispiel: Ein Compiler für Expressions

Aufgabe: Compiler soll algebraische Ausdrücke in Postfix Notation verwandeln. Starte mit der Grammatik

G

Expr	→	Expr + Expr
	|	Expr - Expr
	|	Expr * Expr
	|	Expr / Expr
	|	(Expr)
	|	id
	|	num

Beseitigung der Linksrekursion :	nicht nötig
Präzedenzstufen (Term, Faktor) :	nicht nötig

Das lex file

- C-Funktion sscanf wandelt eine Ziffernfolge (ASCII-String) in Zahlenwert (int).
- Die int-Konstanten PLUS, MINUS, TIMES, QUOT werden später in dem zugehörigen yacc-file spezifiziert.

```
letter  [a-zA-Z]
digit   [0-9]
%%
[ \t]+                          ;
"+"                             { return(PLUS)  }
"-"                             { return(MINUS) }
"*"                             { return(TIMES) }
"/"                             { return(QUOT)  }
{digit}+                        { sscanf(yytext,"%d", &yylval);
                                  return(NUM)}
{letter}({letter}|{digit})*     {return(ID)}
%%
```

Das yacc file

- Deklariere token ID, NUM, PLUS, MINUS, TIMES, QUOT als Operatoren
  - Wahlweise links-/rechts- oder nonassoziativ.
  - Bewirkt richtige Auflösung von shift-reduce Konflikten.
- Reihenfolge wichtig (PLUS, MINUS, TIMES, QUOT)
  - bewirkt gewünschte Präzedenz.
- yylval und yytext speichern Attribute der Token (hier NUM und ID)

```
%token ID, NUM
%left  PLUS, MINUS
%left  TIMES, QUOT
%start expr
%%
expr    : expr PLUS expr   { printf(“add “); }
        | expr MINUS expr  { printf(“sub “); }
        | expr TIMES expr  { printf(“mult “); }
        | expr QUOT expr   { printf(“div “); }
        | NUM              { printf("%d ",yylval);}
        | ID               { printf("%s ",yytext);}
        ;
%%
#include "lex.yy.c"
int main(){
    printf(“Bitte geben Sie einen Ausdruck ein :\n”);
    yyparse(); }
```

Das von lex erzeugte C-Programm

Das fertige C-Programm

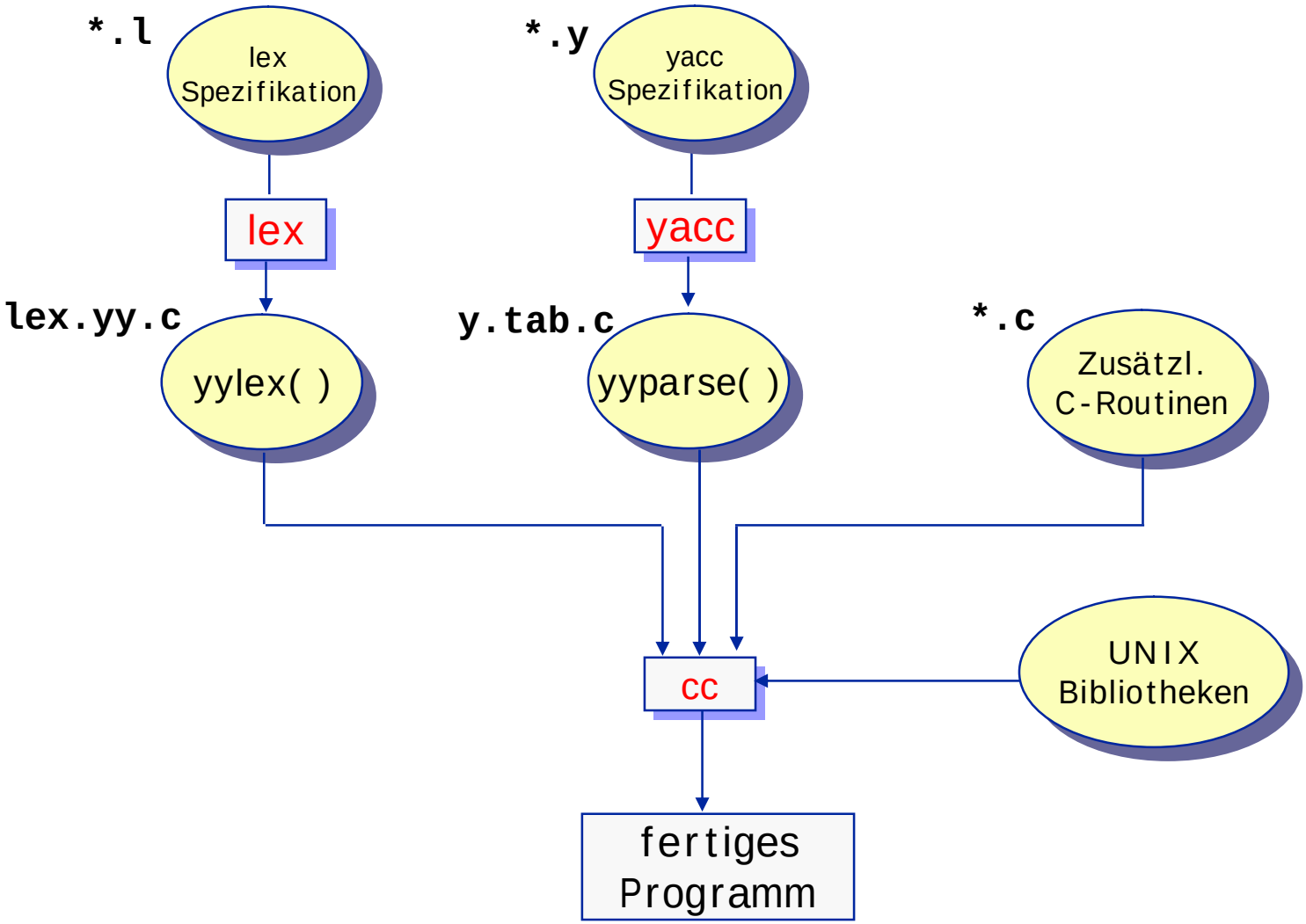
> lex in2post.l
  - Erzeugt aus in2post.l ein C-File lex.yy.c.

> yacc in2post.y
  - Erzeugt aus in2post.y ein C-File y.tab.c.

> cc –o myProg y.tab.c –ll -ly
  - Compiliere mit Bibliotheksroutinen
    - für lex -ll und
    - für yacc -ly

Endungen .l für lex/flex-Datei und .y für yacc/bison-Datei üblich

Das Zusammenspiel von lex & yacc



Der Aufbau eines Syntaxprüfers

Die Arbeitsweise eines Syntaxprüfers geschieht in 2 Phasen.
Diese können zeitlich verschachtelt ablaufen.

Phase 1: Scannen

Scanner zerlegt inputfile anhand regulärer Definitionen in Liste von Token.

File of char:

b	e	t	r	a	g			:	=	b	e	t	r	a	g		*	(	1	+	z	i	n	s	)	

File of token:

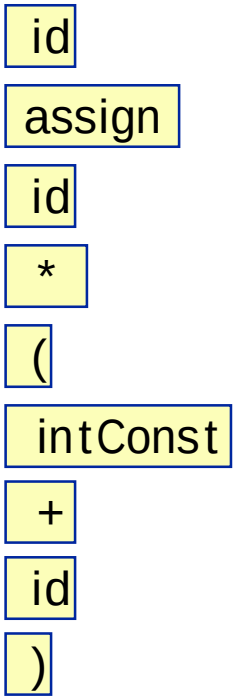
id	assign	id	*	(	intConst	+	id	)

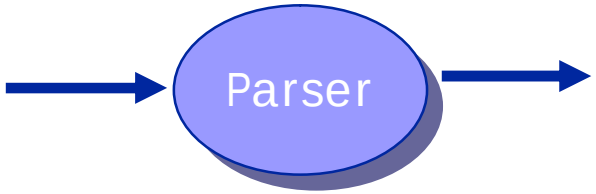
Der Aufbau eines Syntaxprüfers

Phase 2: Parsen

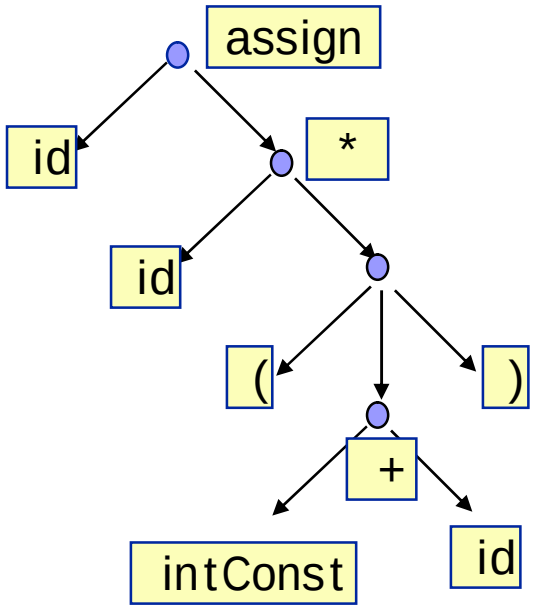
Parser versucht Tokenliste zu einem Herleitungsbaum zu gruppieren

File of token:



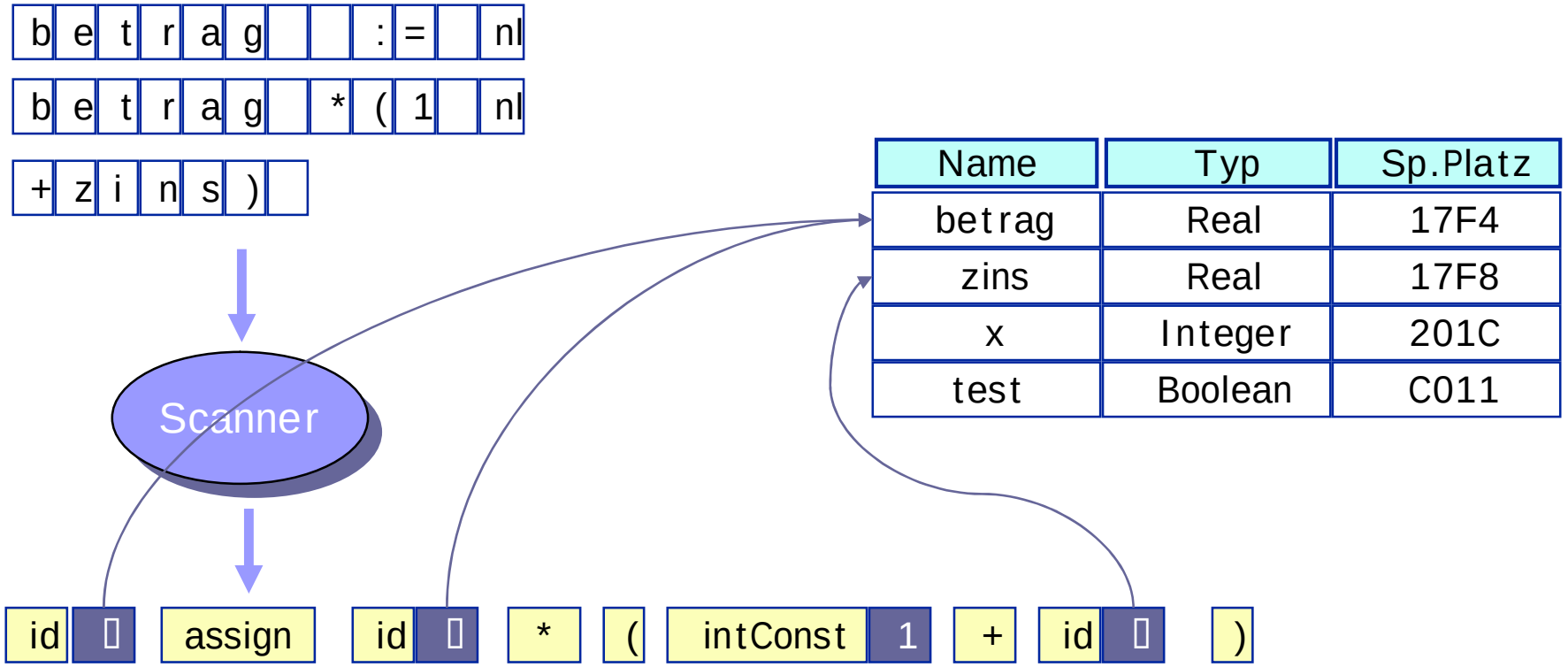


Parse Tree :



Symboltabelle

- Um aus einem Syntaxbaum Code zu erzeugen muß die Identität einiger Token bekannt sein.
  - welche Bezeichner sind identisch
  - welchen Wert eine intConst , etc.
- Scanner legt diese Information in Symboltabelle ab und liefert dem Parser die Token samt Zeiger in diese Tabelle.

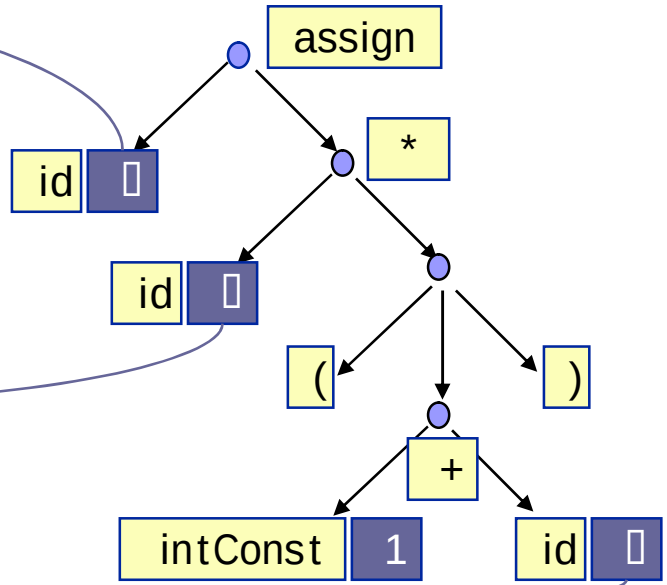


Name	Typ	Sp.Platz
betrag	Real	17F4
zins	Real	17F8
x	Integer	201C
test	Boolean	C011

Symboltabelle

Name	Typ	Sp.Platz
betrag	Real	17F4
zins	Real	17F8
x	Integer	201C
test	Boolean	C011

Parse Tree :



Gewisse Token im Syntaxbaum haben einen Link in die Symboltabelle

Stackprozessor

- Aus Programmtext soll Code für eine Maschine erzeugt werden.
  - Einfaches Maschinenmodell: Stackprozessor.

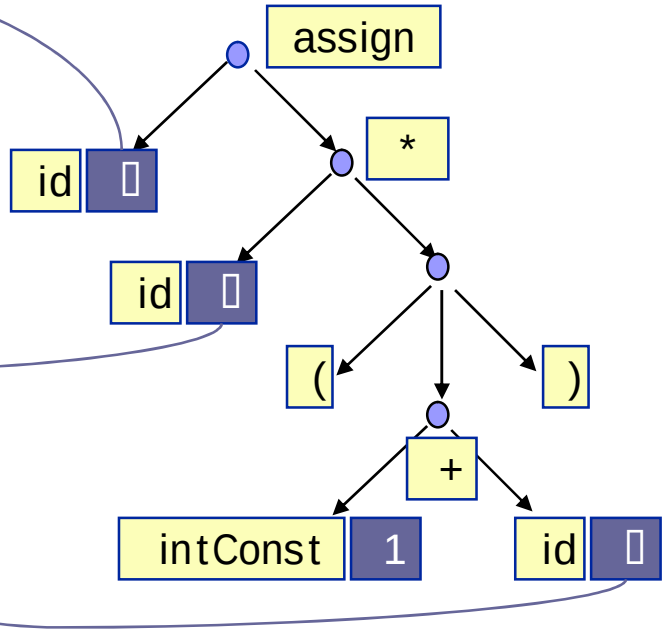
Typische Befehle eines Stackprozessors :

PUSH <num>	Lege den Zahlenwert <num> auf dem Stack ab
LOAD <hex>	Lege den Inhalt von Adresse <hex> auf dem Stack ab
STORE <hex>	Speichere den Top des Stacks an Adresse <hex>. Der Stack wird dabei gepopped.
ADD	Ersetze das oberste Element des Stacks durch die Summe der beiden obersten
MULT	Ersetze das oberste Element des Stacks durch das Produkt der beiden obersten

Code-Erzeugung

Name	Typ	Sp.Platz
betrag	Real	17F4
zins	Real	17F8
x	Integer	201C
test	Boolean	C011

Parse Tree :



Code für einen Stackprozessor

```
LOAD 17F4
PUSH 1
LOAD 17f8
ADD
MULT
STORE 17F4
```

Code Erzeugung aus dem Parser

- Code-Erzeugung durch semantische Aktionen:

```
%{ int* loc }%

expr    : expr + expr       { printf("ADD \n"); }
        | expr * expr       { printf("MULT\n"); }
        | intConst          { printf("PUSH ");
                              printf("%d\n",yylval);}
        | ID                {  printf("LOAD ");
                               printf(lookup(yytext));}
        ;

stmt  : ID  { loc=lookup(yytext);}
        ASSIGN expr  { printf("STORE ");
                       printf(&loc);     }

     | ... etc. ...

%%
#include lex.yy.c
...
```