# LR(1)-Itemmengenkonstruktion

- Grammatik: $S \to A\$ \mid xb$
  $A \to aAb \mid B$
  $B \to x$
- Sprache: $\{xb\} \cup \{a^n x b^n \mid n \geq 0\}$
- nicht LL(1)
  - $x \in FIRST(B) \Rightarrow x \in FIRST(A)$
  - also: FIRST/FIRST-Konflikt bei $S \to A\$ \mid xb$
- nicht SLR(1) (Abb. 2.96)
  - betrachte Item $B \to x\bullet$, mit $b \in FOLLOW(B) = \{b, \$\}$
  - also: SHIFT/REDUCE-Konflikt in Zustand $s_2$ bei Eingabe $b$
  - Folgemenge für alle Items einer Produktion gleich
- bei LR(1)
  - eine spezifische Folgemenge für jedes Item
  - LR(1)-Item: $N \to \alpha\bullet\beta\,\{\sigma\}$
  - Folge-Reduce-Item nur gültig, wenn Eingabetoken in $\sigma$

# Beispiel: Konflikt bei SLR(1)

```
S → A | xb
A → aAb | B
B → x
```

**Figure 2.95** Grammar for demonstrating the LR(1) technique.

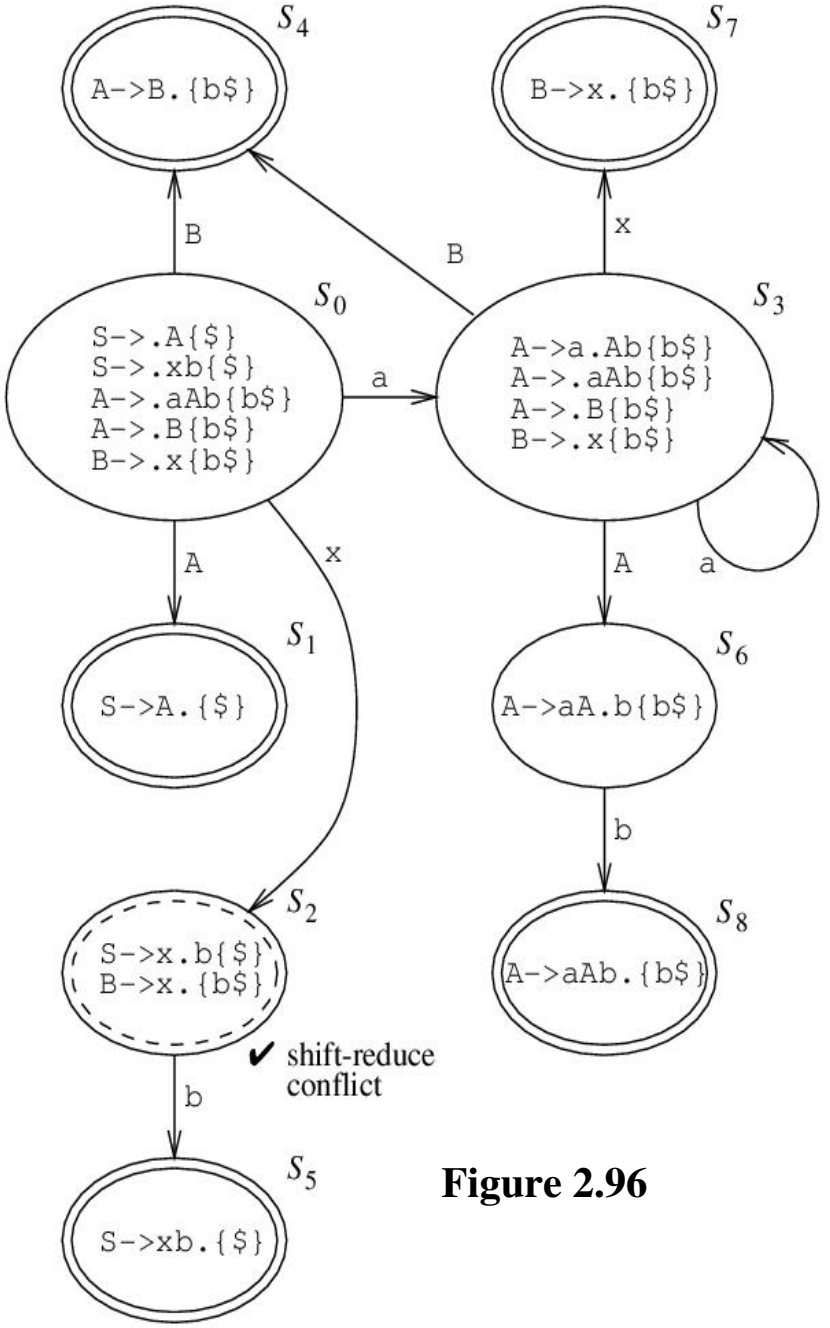
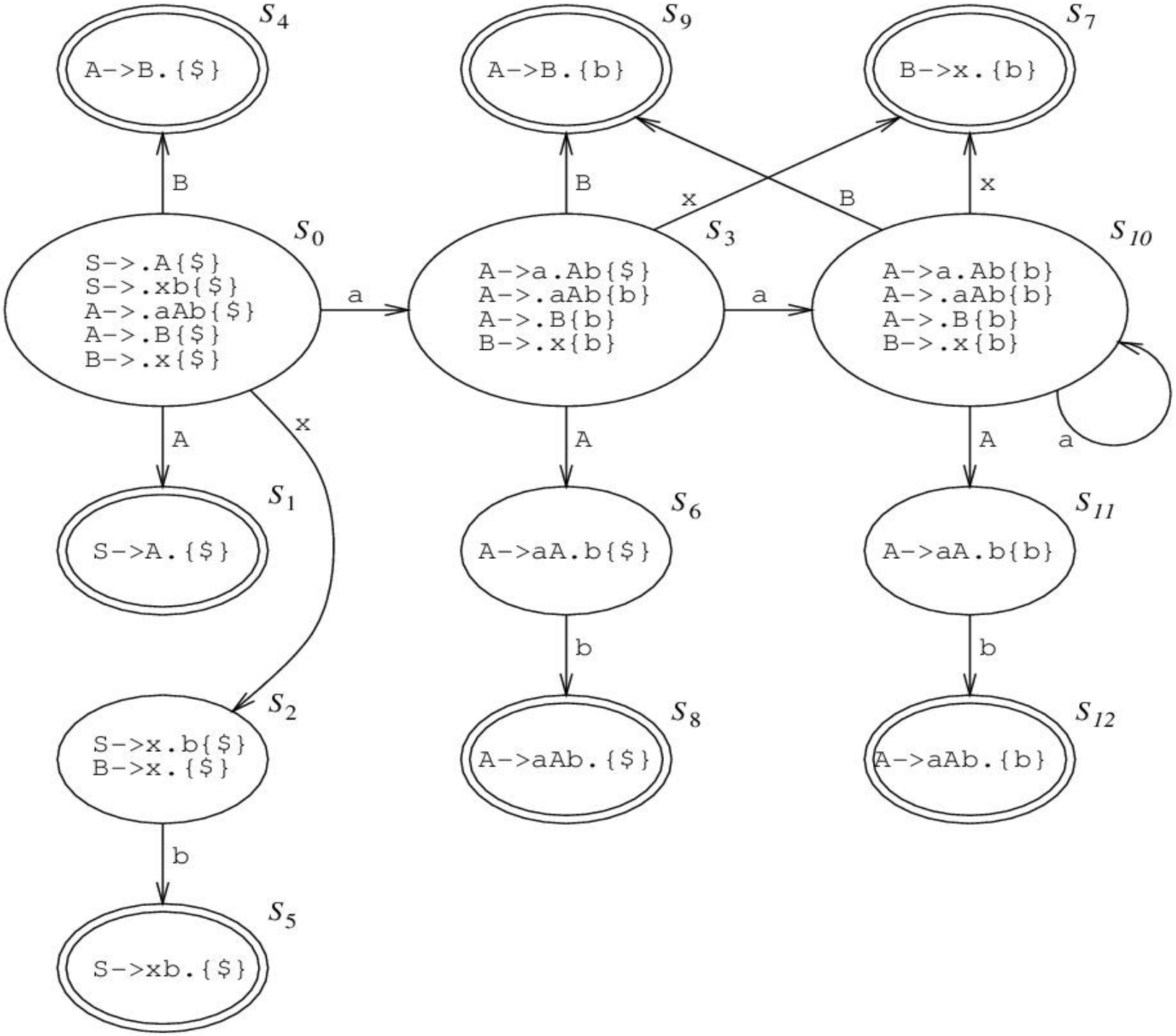



**Figure 2.96**

# Konstruktion der Itemfolgemengen

- initiale Itemmenge:
  - alle initialen Items bekommen die Folgemenge {$}
- ε-Hülle: (Abb. 2.98)
  - für Item $P \to \alpha \bullet N \beta \{\sigma\}$ enthält die Hülle die Items $N \to \bullet \gamma \{FIRST(\beta\{\sigma\})\}$ für jede Produktion $N \to \gamma$
- Erweiterung der Definition von FIRST für LR(1):
  - $\varepsilon \notin FIRST(\beta) \Rightarrow FIRST(\beta\{\sigma\}) = FIRST(\beta)$
  - $\varepsilon \in FIRST(\beta) \Rightarrow FIRST(\beta\{\sigma\}) = (FIRST(\beta) \setminus \{\varepsilon\}) \cup \sigma$
- Auswirkungen auf $S_2$: (Abb. 2.97)
  - Shift für b
  - Reduce für $
- LR(1)-Algorithmus (Abb. 2.99)
- LR(*k*) mit *k*>1: ACTION-Tabelle indiziert mit Strings statt Zeichen

# Beispiel


$S_4$
A->B.{$}
$S_9$
A->B.{b}
$S_7$
B->x.{b}
B
B
x
B
x
$S_0$
S->.A{$}
S->.xb{$}
A->.aAb{$}
A->.B{$}
B->.x{$}
a
$S_3$
A->a.Ab{$}
A->.aAb{b}
A->.B{b}
B->.x{b}
a
$S_{10}$
A->a.Ab{b}
A->.aAb{b}
A->.B{b}
B->.x{b}
A
x
A
A
a
$S_1$
S->A.{$}
$S_6$
A->aA.b{$}
$S_{11}$
A->aA.b{b}
b
b
$S_2$
S->x.b{$}
B->x.{$}
$S_8$
A->aAb.{$}
$S_{12}$
A->aAb.{b}
b
$S_5$
S->xb.{$}

# ε-Hülle der LR(1)-Itemmengen

**Data definitions:**

A set $S$ of LR(1) items of the form $N \rightarrow \alpha \bullet \beta \{\sigma\}$.

**Initializations:**

$S$ is prefilled externally with one or more LR(1) items.

**Inference rules:**

If $S$ holds an item of the form $P \rightarrow \alpha \bullet N \beta \{\sigma\}$, then for each production rule $N \rightarrow \gamma$ in $G$, $S$ must also contain the item $N \rightarrow \bullet \gamma \{\tau\}$, where $\tau = \text{FIRST}(\beta \{\sigma\})$.

**Figure 2.98** ε closure algorithm for LR(1) item sets for a grammar $G$.

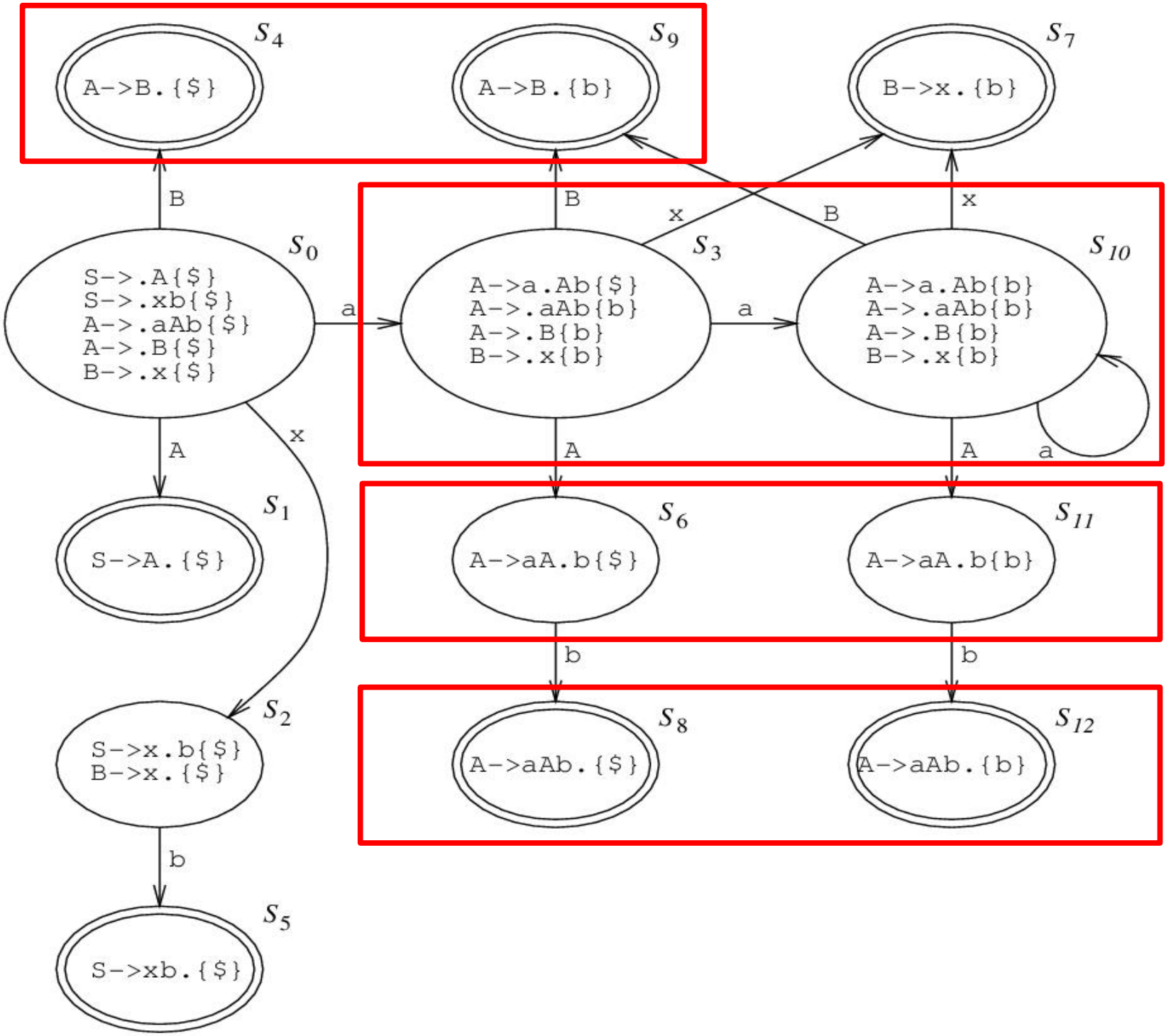
```
IMPORT Input token [1..];        // from the lexical analyzer

SET Input token index TO 1;
SET Reduction stack TO Empty stack;
PUSH Start state ON Reduction stack;

WHILE Reduction stack /= {Start state, Start symbol, End state}
    SET State TO Top of Reduction stack;
    SET Look ahead TO Input token [Input token index] .class;
    SET Action TO Action table [State, Look ahead];
    IF Action = "shift":
        // Do a shift move:
        SET Shifted token TO Input token [Input token index];
        SET Input token index TO Input token index + 1; // shifted
        PUSH Shifted token ON Reduction stack;
        SET New state TO Goto table [State, Shifted token .class];
        PUSH New state ON Reduction stack;      // cannot be Empty
    ELSE IF Action = ("reduce", N→α):
        // Do a reduction move:
        Pop the symbols of α from Reduction stack;
        SET State TO Top of Reduction stack;    // update State
        PUSH N ON Reduction stack;
        SET New state TO Goto table [State, N];
        PUSH New state ON Reduction stack;      // cannot be Empty
    ELSE Action = Empty:
        ERROR "Error at token ", Input token [Input token index];
```

**Figure 2.99** LR(1) parsing with a push-down automaton.

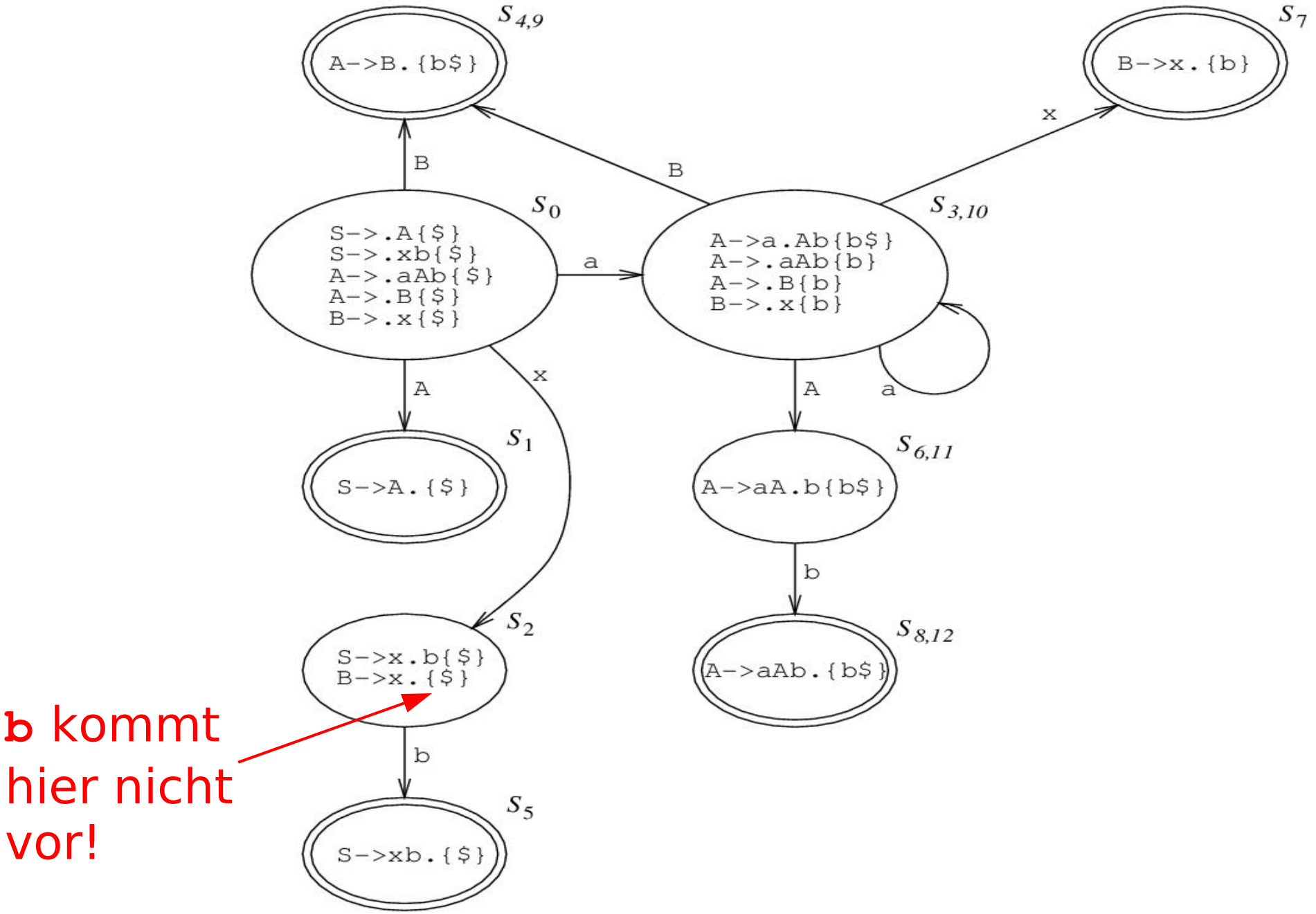
# LALR(1)-Itemkonstruktion

- Ziel:
  - Item-spezifische Folgeinformation (fast) wie in LR(1)
  - Tabellengröße von SLR(1), LR(0)
    (um Faktoren 10-100 kleiner als LR(1))
- Beobachtung:
  - viele LR(1)-Zustände nur in den Folgemengen unterschiedlich
  - *Kern* eines Zustands = Items ohne Folgemengen
    (LR(0)-Itemmengen)
  - die Kerne bestimmen die GOTO-Tabelle
    (LR(0)-Konstruktion)
- Idee: "Zurückrudern" von LR(1)...
  - ... unter Erhaltung der Itemspezifität von Folgemengen
  - fasse Zustände mit gleichem Kern zusammen
    (Tabellengröße von SLR(1), LR(0))
  - vereinige die entsprechenden Folgemengen Item-weise
    (potenziell zusätzliche Reduce/Reduce-Konflikte)

# Beispiel


$S_4$
A->B.{$}
$S_9$
A->B.{b}
$S_7$
B->x.{b}
B
B
x
B
x
$S_0$
S->.A{$}
S->.xb{$}
A->.aAb{$}
A->.B{$}
B->.x{$}
a
$S_3$
A->a.Ab{$}
A->.aAb{b}
A->.B{b}
B->.x{b}
a
$S_{10}$
A->a.Ab{b}
A->.aAb{b}
A->.B{b}
B->.x{b}
A
x
A
A
a
$S_1$
S->A.{$}
$S_6$
A->aA.b{$}
$S_{11}$
A->aA.b{b}
b
b
$S_2$
S->x.b{$}
B->x.{$}
$S_8$
A->aAb.{$}
$S_{12}$
A->aAb.{b}
b
$S_5$
S->xb.{$}

# Nach der Vereinigung




b kommt hier nicht vor!

# Verletzungen von LR(1)

- Hauptursache: Mehrdeutigkeiten
- Reduce/Reduce-Konflikte:
  - Analogon im Scanning:
    ein längstes Token passt in mehrere Tokenklassen
  - Vorgehensweise:
    Auswahl nach Reihenfolge der Produktionen in der Spezifikation
- Shift/Reduce-Konflikte:
  - Analogon im Scanning:
    ein Token ist Präfix eines anderen Tokens
  - Vorgehensweisen:
    - Shift-Präzedenz:
      - Bsp. *dangling else*: shifte in jedem Fall das `else`
        (und binde es so an das vorher gesehene `then`)
      - leichte Modifikation des Tabellengenerierungsalgorithmus
    - Benutzer-definierte Token-Präzedenz

# Dangling `else`, Wahl: shift

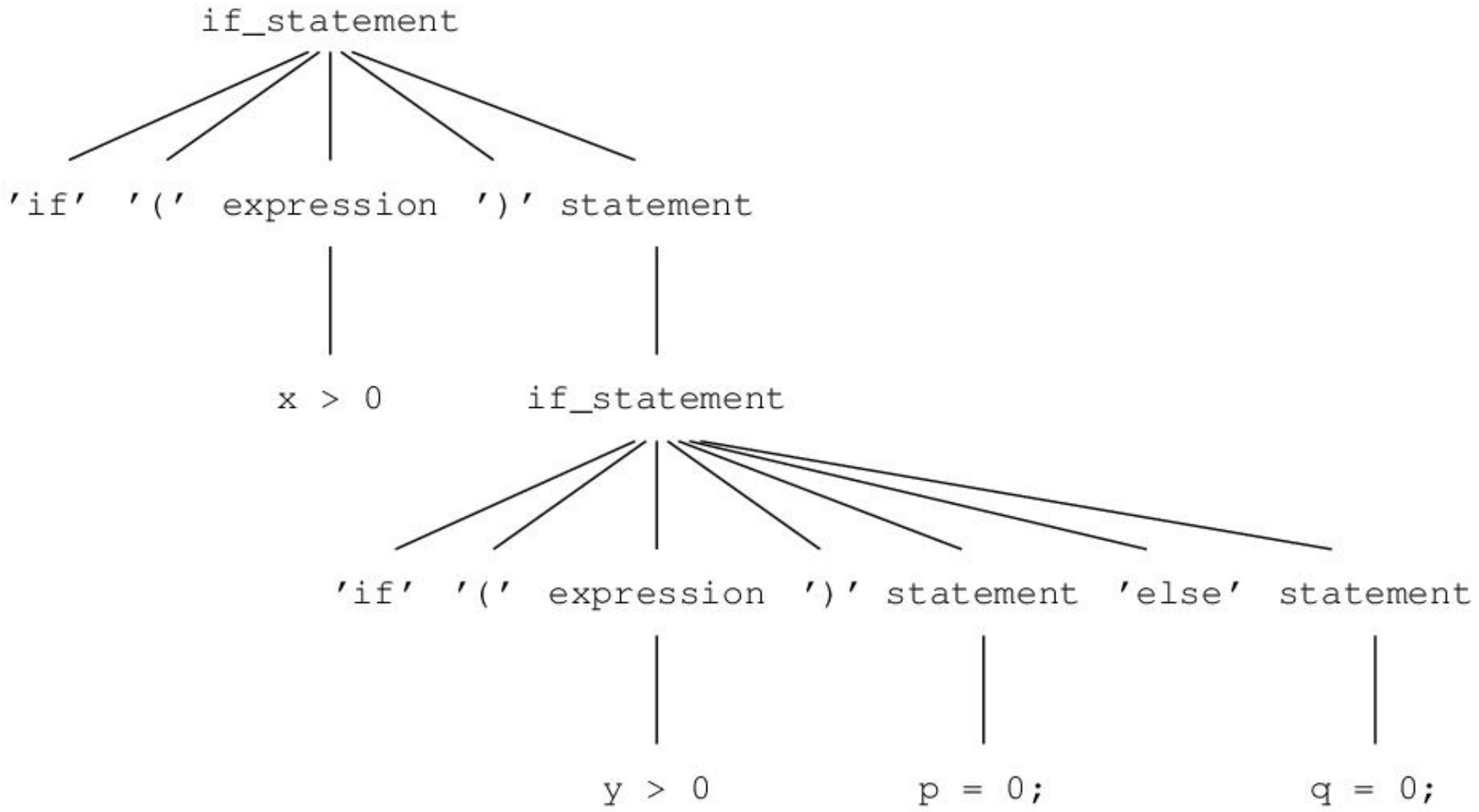



**Figure 2.101** A possible syntax tree for a nested if-statement.

# Dangling `else`, Wahl: reduce

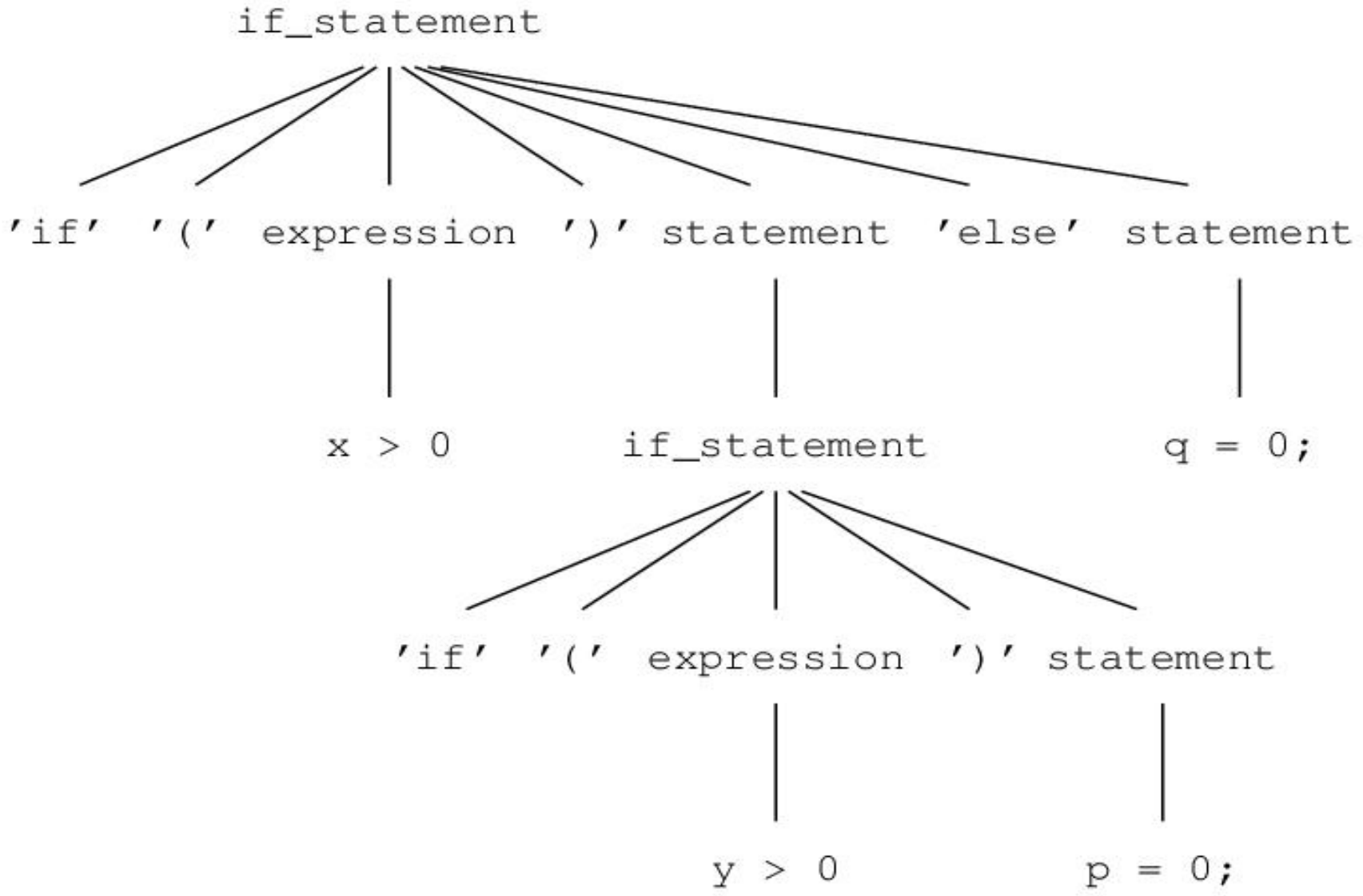



**Figure 2.102** An alternative syntax tree for the same nested if-statement.

# Benutzer-definierte Tokenprioritäten

- Voraussetzung:
  - Eingabetoken t
    Konfliktzustand enthält die Items:
    (a) $P \to \alpha \cdot t \beta \{ \ldots \}$ (Shift-Item)
    (b) $Q \to \gamma u R \cdot \{ \ldots, t, \ldots \}$ (Reduce-Item)
  - R ist entweder leer oder ein einzelnes Nichtterminal
- Aktion:
  - $u <\cdot t$ (u hat kleinere Priorität als t)⇒ shifte
  - $u \doteq t$ (u hat gleiche Priorität wie t)⇒ shifte
  - $u \cdot > t$ (u hat größere Priorität als t)⇒ reduziere (gehe zurück bis zum letzten $<\cdot$, dann Handle erkannt)
- häufige Anwendung:
  - Ausdruckssprache mit priorisierten Infixoperatoren (Operator-Prioritäts-Grammatiken)
  - Prioritäten können leicht verändert werden
  - man spart Nichtterminale und Reduktionen auf diese

# Operator-Prioritäts-Grammatiken

- Produktionen: `E → E+E | E−E | E*E | E/E | E↑E | (E) | i`
- Prioritäten:

| ↱ | + | − | * | / | ↑ | i | ( | ) | $ |
|---|---|---|---|---|---|---|---|---|---|
| + | ·> | ·> | <· | <· | <· | <· | <· | ·> | ·> |
| − | ·> | ·> | <· | <· | <· | <· | <· | ·> | ·> |
| * | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| / | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| ↑ | ·> | ·> | ·> | ·> | <· | <· | <· | ·> | ·> |
| i | ·> | ·> | ·> | ·> | ·> | | | ·> | ·> |
| ( | <· | <· | <· | <· | <· | <· | <· | ≐ | |
| ) | ·> | ·> | ·> | ·> | ·> | | | ·> | ·> |
| $ | <· | <· | <· | <· | <· | <· | <· | | |

  - `+,−,*,/` linksassoziativ
  - `↑` rechtsassoziativ
  - `i` hat höchste, `$` niedrigste Priorität
  - linke Klammer entspricht `<·`, rechte Klammer `·>`
  - leere Einträge: keine Nachbarschaft möglich
- erweiterte Eingabe: `$<·i·>+<·i·>*<·i·>$`

# Operator-Prioritäts-Algorithmus

- kann von Hand implementiert werden
- arbeitet mit Stack, Input und Präzedenztabelle (Drachenbuch, Abb. 4.24)
- kann auch mit unären Operatoren umgehen
- Prioritätsinformation kann oft mit Hilfe zweier Integer-Funktionen kodiert werden:
  - `f(u)<g(t)` ⇔ `u<·t`
  - `f(u)=g(t)` ⇔ `u=t`
  - `f(u)>g(t)` ⇔ `u·>t`

| | + | - | * | / | ↑ | ( | ) | i | $ |
|---|---|---|---|---|---|---|---|---|---|
| f | 2 | 2 | 4 | 4 | 4 | 0 | 6 | 6 | 0 |
| g | 1 | 1 | 3 | 3 | 5 | 5 | 0 | 5 | 0 |

- Achtung: die Prioritätsfunktionen sind total (keine Fehlerinfo!)

# LR-Fehlererkennung

- LL-Idee: greift bei LR nicht
  - Panic Recovery (Acceptable Set = {korrekte Tokens})
    genauso destruktiv wie bei LL
  - Erkennung von Continuations viel aufwändiger
    (da inverse Rechtsableitung)
- LR-Ansatz: Modifikation des Stacks (yacc)
  - bestimme Fehler-behandelnde Nichtterminale
    (error-recovering nonterminals)
  - definieren typischerweise große Sprachsegmente
    (**`declaration`**, **`expression`**, **`statement`**, etc.)
- Vorgehen

  wenn Fehler bei Erkennung eines solchen Nichtterminals *R* auftritt
  - brich Versuch der normalen Herleitung von *R* ab
  - füge einen Dummyknoten *R* ein
  - suche in der Eingabe nach einem Token, das *R* folgen kann
    (schwierig!)

# LR-Fehlererkennung: Beispiel

- Fehlersituation: $\mathtt{N} \rightarrow \alpha \bullet \mathtt{R} \beta$
- **R** Fehler-behandelndes Nichtterminal

  `R → G H I`

  `R → erroneous_R` (hinzugefügtes Fehlerterminal)
- Vorgehensweise:

  1. wirf bisher konstruierte Knoten des Teilbaums für `R` weg (Abb. 2.103-104)
  2. füge Fehlerterminal für `R` ein und reduziere normal (Abb. 2.105)
  3. übergehe Eingabe bis zu einem Token, das im gegenwärtigen Zustand erwartet wird (Abb. 2.106)
  4. fahre wie gewöhnlich fort (Abb. 2.107)

# LR-Fehlererkennung: Beispiel

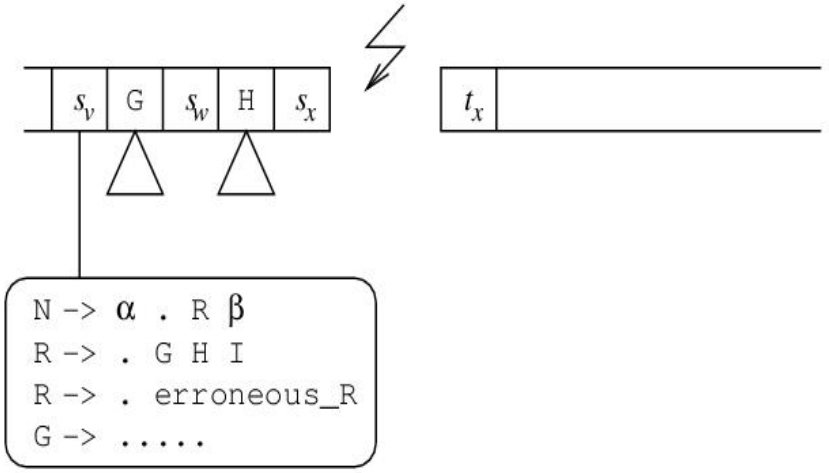



**Figure 2.103** LR error recovery – detecting the error.

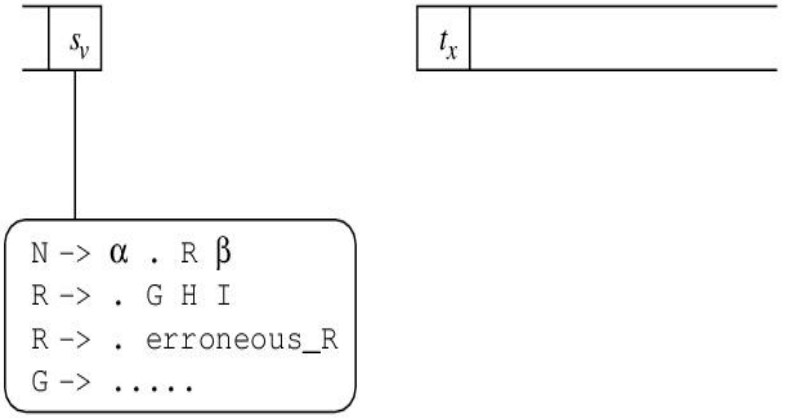
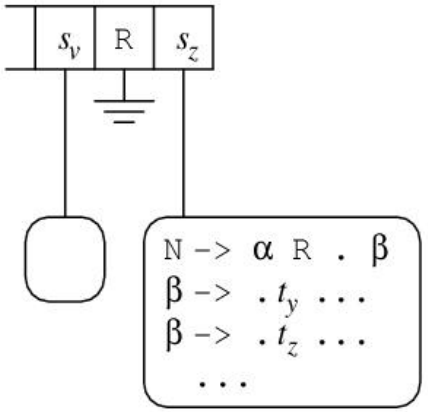



**Figure 2.104** LR error recovery – finding an error recovery state.



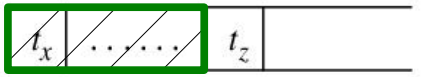




Fig. 2.106: repairing the input

**Figure 2.105** LR error recovery – repairing the stack.

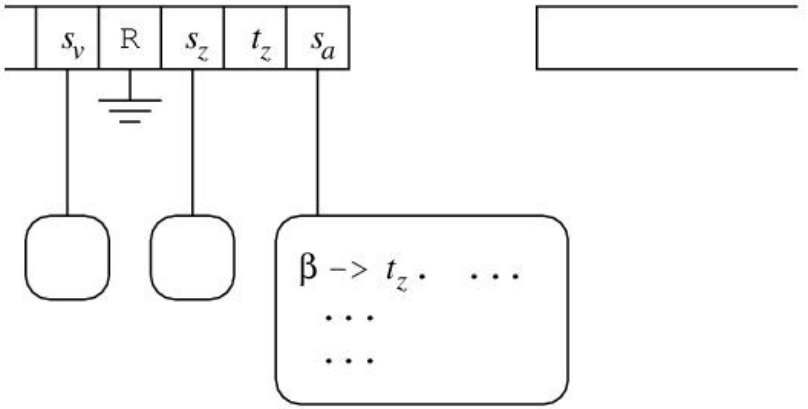



**Figure 2.107** LR error recovery – restarting the parser.