Bottom-Up Parsing

Voorbeeld:

Program → begin Stmts end \$
Stmts → Stmt ; Stmts
             | λ
Stmt → simplestmt
             | begin Stmts end

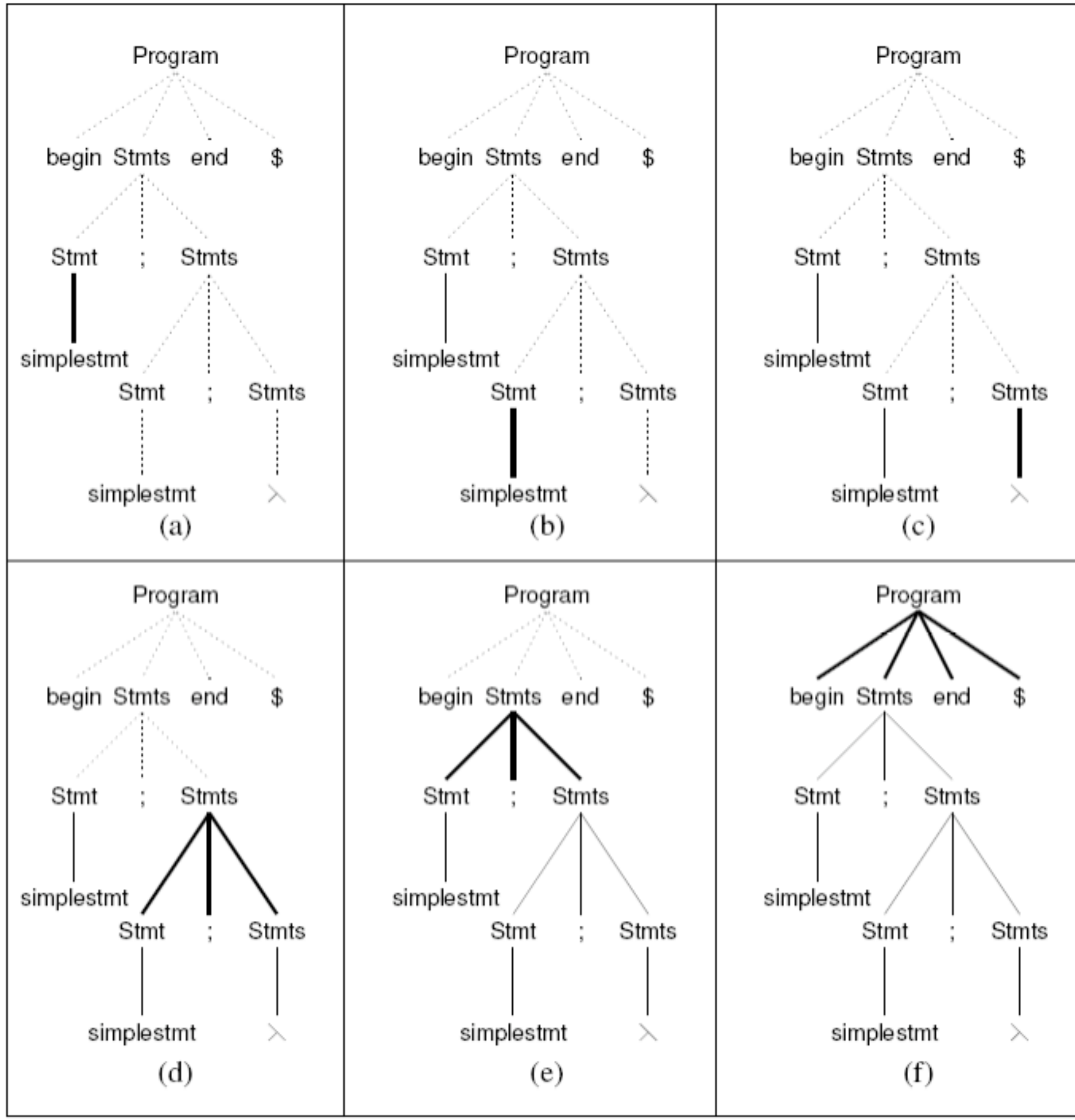

Figure 4.6: Parse of “begin simplestmt ; simplestmt ; end \$” using the bottom-up technique. Legend explained on page 126.

Bottom-up parsing:

- Begin met bladeren van de parse tree (tokens)
- Genereer een rightmost derivatie, in omgekeerde volgorde
- Zoek telkens een RHS van een productie, vervang die door de LHS (= reductie)

shift-reduce techniek:

Herhaal:
Shift symbolen op een stack tot er een volledige RHS op de top staat; als dat het geval is, reduce (= vervang de RHS door de LHS, en zet die vooraan in de input – doe dus alsof die nonterminal het volgende inputsymbool is)

We werken opnieuw Table-driven
Deze methode is sterker dan Top-down

Voorbeeld: rightmost derivatie

1	Start	$\rightarrow$	E \$
2	E	$\rightarrow$	plus E E
3		|	num

Rule	Derivation
1	Start $\Rightarrow_{rm}$ E \$
2	$\Rightarrow_{rm}$ plus E E \$
3	$\Rightarrow_{rm}$ plus E num \$
3	$\Rightarrow_{rm}$ plus num num \$

Figure 6.2: Grammar and rightmost derivation of plus num num \$.

Deel van de overeenkomstige shift-reduce parse:

(a) na 2 shift operaties
(b) Reductie van num tot E
(c) Verdere shift, shift, reductie, shift
(d) Nog een reductie
(e) Shift
(f) Reductie, accept

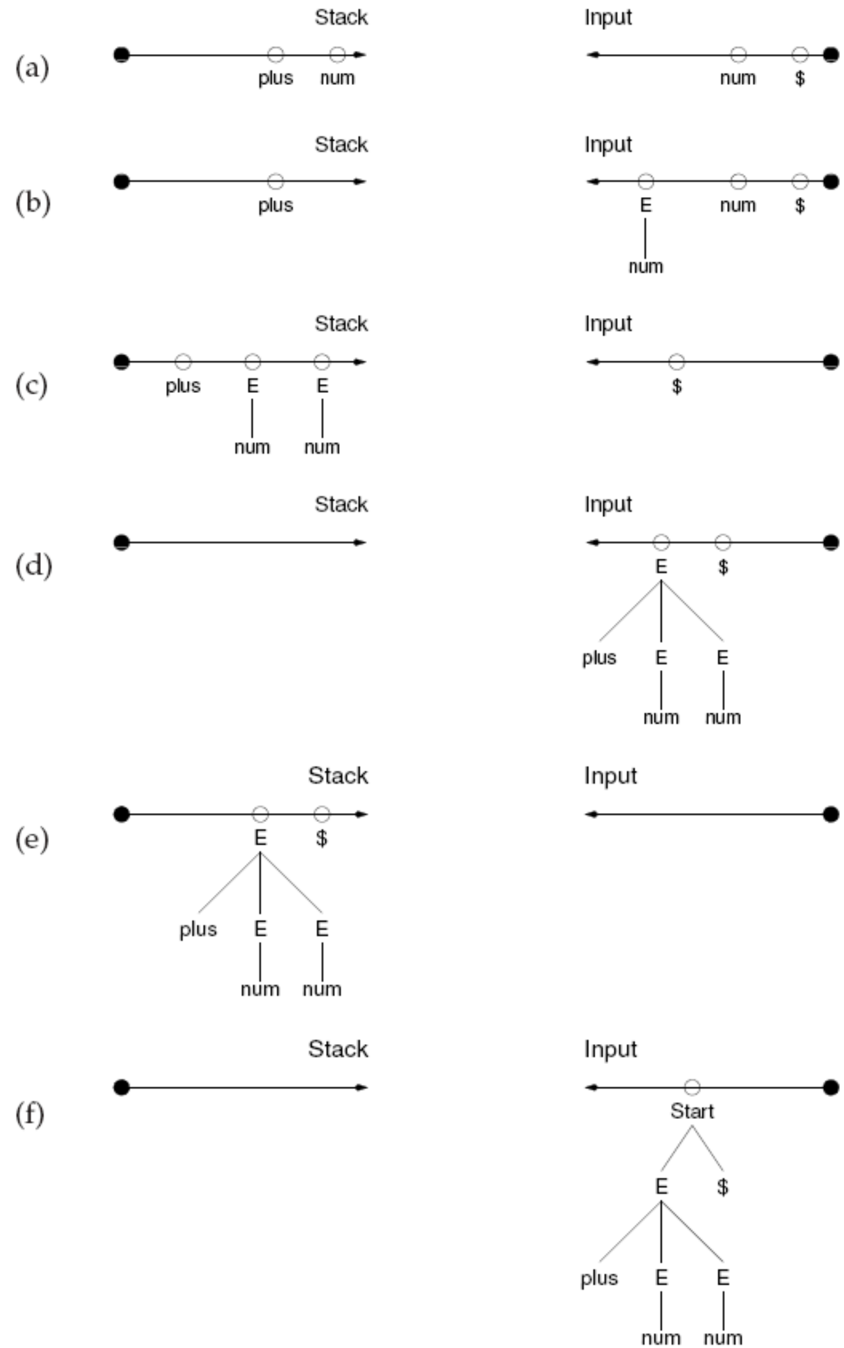

Figure 6.1 Bottom-up parsing resembles knitting

Wat moet de tabel bevatten om dit efficiënt te laten werken? We willen niet telkens alle mogelijke producties onderzoeken!

- Gegeven het symbool op de top van de stack en het volgende inputsymbol, moet de tabel ons vertellen welke actie nodig is: shift, reduce, accept of error.

- De stack zal niet enkel grammaticale symbolen bevatten, maar ook states; op de top van de stack zal een state staan die aangeeft welke producties op dat ogenblijk al gedeeltelijk herkend zijn (dus hun RHS is al gedeeltelijk herkend)

- We geven een constructie van de states: dat zullen sets van items zijn. Een item is een productie met een marker ● in de RHS.

```
call Stack.PUSH(StartState)
accepted ← false
while not accepted do
    action ← Table[Stack.TOS( )][InputStream.PEEK( )]          ①
    if action = shift s
    then
        call Stack.PUSH(s)                                      ②
        if s ∈ AcceptStates                                     ③
        then  accepted ← true
        else  call InputStream.ADVANCE( )
    else
        if action = reduce A → γ
        then
            call Stack.POP(|γ|)                                 ④
            call InputStream.PREPEND(A)                         ⑤
        else
            call ERROR( )                                       ⑥
```

Shift zet een state op de stack

Reduce popt states van de stack

Figure 6.3: Driver for a bottom-up parser.

Voorbeeld

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A C
3	C	$\rightarrow$	c
4		|	λ
5	A	$\rightarrow$	a B C d
6		|	B Q
7	B	$\rightarrow$	b B
8		|	λ
9	Q	$\rightarrow$	q
10		|	λ

Rule	Derivation
1	Start $\Rightarrow_{rm}$ S \$
2	$\Rightarrow_{rm}$ A C \$
3	$\Rightarrow_{rm}$ A c \$
5	$\Rightarrow_{rm}$ a B C d c \$
4	$\Rightarrow_{rm}$ a B d c \$
7	$\Rightarrow_{rm}$ a b B d c \$
7	$\Rightarrow_{rm}$ a b b B d c \$
8	$\Rightarrow_{rm}$ a b b d c \$

Figure 6.4: Grammar and rightmost derivation of a b b d c \$.

($\boxed{x}$ = shift state x)

State	a	b	c	d	q	$	Start	S	A	B	C	Q
0	$\boxed{3}$	$\boxed{2}$	8		8	8	accept	$\boxed{4}$	$\boxed{1}$	$\boxed{5}$		
1			$\boxed{11}$			4					$\boxed{14}$	
2		$\boxed{2}$	8	8	8	8				$\boxed{13}$		
3		$\boxed{2}$	8	8						$\boxed{9}$		
4						$\boxed{8}$						
5			10		$\boxed{7}$	10						$\boxed{6}$
6			6			6						
7			9			9						
8						1						
9			$\boxed{11}$	4							$\boxed{10}$	
10				$\boxed{12}$								
11				3		3						
12			5			5						
13			7	7	7	7						
14						2						

Figure 6.5: Parse table for the grammar shown in Figure 6.4.

0	Initial Configuration	a b b d c $
0, a 3	shift a	b b d c $
0, a 3, b 2	shift b	b d c $
0, a 3, b 2, b 2	shift b	d c $
0, a 3, b 2, b 2	Reduce λ to B	B d c $
0, a 3, b 2, b 2, B 13	shift B	d c $
0, a 3, b 2	Reduce b B to B	B d c $
0, a 3, b 2, B 13	shift B	d c $
0, a 3	Reduce b B to B	B d c $
0, a 3, B 9	shift B	d c $
0, a 3, B 9	Reduce λ to C	C d c $
0, a 3, B 9, C 10	shift C	d c $
0, a 3, B 9, C 10, d 12	shift d	c $
0	Reduce a B C d to A	A c $

(continue to Figure 6.7)

Figure 6.6: Bottom-up parse of a b b d c $.

Stack	Action	Input
[0]	*(continued from Figure 6.6)*	A c $
[0] [A 1]	shift A	c $
[0] [A 1] [c 11]	shift c	$
[0] [A 1]	Reduce c to C	C $
[0] [A 1] [C 14]	shift C	$
[0]	Reduce A C to S	S $
[0] [S 4]	shift S	$
[0] [S 4] [$ 8]	shift $	$
[0]	Reduce S $ to Start	Start $
[0] [Start 0]	shift Start	$
	Accept	

Figure 6.7: Continued bottom-up parse of a b b d c $.

Constructie van de states (en de tabel):

- De states zijn die van een specifieke eindige automaat: de karakteristieke automaat (CFSM) van de grammatica. De shift acties komen overeen met de transities vande automaat, de reduce-acties met de eindtoestanden

- De taal herkend door de CFSM bestaat uit de right sentential forms die eindigen met een prefix van een RHS, en die ten hoogste één RHS bevatten. Dit noemt men de viable prefixes van de grammatica.

- viable prefix = prefix van een right sentential form die zich niet uitstrekt voorbij de handle.

- De handle van een right sentential form = occurrence van de RHS van de laatste stap in zijn rightmost derivatie.

- De inhoud van de stack zal overeenkomen met een viable prefix. Reductie is nodig wanneer een volledige handle op de top van de stack staat.

Door de states op de stack te zetten kunnen we vermijden om het al verwerkte deel van de invoer steeds opnieuw te moeten analyseren na een reduce actie; we moeten enkel terugkeren naar de state onder de handle.

De klasse van grammatica's die op deze manier kunnen verwerkt worden voldoet aan de zgn LR(k) conditie:

als

$$S \overset{*}{\Rightarrow}_{rm} \alpha A w \overset{*}{\Rightarrow}_{rm} \alpha \beta w$$

$$S \overset{*}{\Rightarrow}_{rm} \gamma B x \overset{*}{\Rightarrow}_{rm} \alpha \beta y$$

$$\mathrm{First}_k(w) = \mathrm{First}_k(y)$$

dan $\alpha A y = \gamma B x$

Constructie van de states: LR(0) items

De items in een state geven aan hoever de parser al gevorderd is in de RHS van de verschillende regels. Als het eind van de RHS van een regel bereikt is, dan is het tijd voor een reductie.

LR(0) item	Progress of rule in this state
$E \rightarrow \bullet$ plus E E	Beginning of rule
$E \rightarrow$ plus $\bullet$ E E	Processed a plus, expect an E
$E \rightarrow$ plus E $\bullet$ E	Expect another E
$E \rightarrow$ plus E E $\bullet$	Handle on top-of-stack, ready to reduce

Figure 6.8: LR(0) items for production $E \rightarrow$ plus E E.

Reduce item: item met markerop het einde. De andere items zijn shift items

```
function ComputeLR0(Grammar) returns (Set, State)
    States ← ∅
    StartItems ← { Start → • RHS(p) | p ∈ ProductionsFor(Start) }   ⑦
    StartState ← AddState(States, StartItems)
    while (s ← WorkList.ExtractElement()) ≠ ⊥ do                      ⑧
        call ComputeGoto(States, s)
    return ((States, StartState))
end
function AddState(States, items) returns State
    if items ∉ States                                                 ⑨
    then
        s ← newState(items)                                           ⑩
        States ← States ∪ { s }
        WorkList ← WorkList ∪ { s }                                   ⑪
        Table[s][★] ← error                                           ⑫
    else  s ← FindState(items)
    return (s)
end
function AdvanceDot(state, X) returns Set
    return ({ A → αX • β | A → α • Xβ ∈ state })                      ⑬
end
```

Figure 6.9: LR(0) construction.

Als er een nonterminal B op de marker volgt, voeg dan de items toe waarin aan producties voor B begonnen wordt

```
function CLOSURE(state) returns Set
    ans ← state
    repeat
        prev ← ans
        foreach A → α • Bγ ∈ ans do
            foreach p ∈ PRODUCTIONSFOR(B) do
                ans ← ans ∪ { B → • RHS(p) }
    until ans = prev
    return (ans)
end
procedure COMPUTEGOTO(States, s)
    closed ← CLOSURE(s)
    foreach X ∈ (N ∪ Σ) do
        RelevantItems ← ADVANCEDOT(closed, X)
        if RelevantItems ≠ ∅
        then
            Table[s][X] ← shift ADDSTATE(States, RelevantItems)
end
```

Figure 6.10: LR(0) closure and transitions.

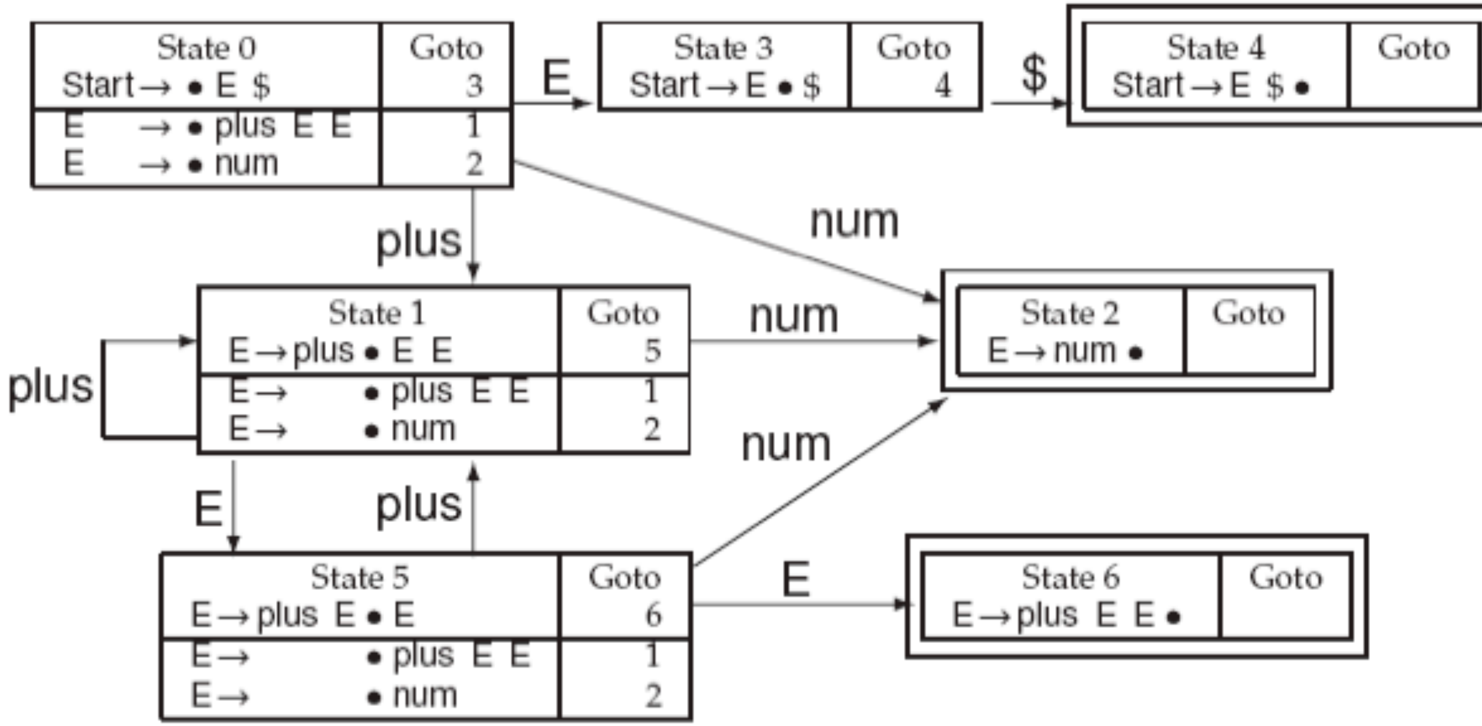

Figure 6.11: LR(0) computation for Figure 6.2, shown as a characteristic finite-state machine. State 0 is the initial state, and the double-boxed states are accept states.

In een state: boven de lijn – kernel items
eronder – items toegevoegd door de closure operatie

Van de CFSM

Sentential Prefix	Transitions	Resulting Sentential Form
		plus plus num num num $
plus plus num	States 1, 1, and 2	plus plus E num num $
plus plus E num	States 1, 1, 5, and 2	plus plus E E num $
plus plus E E	States 1, 1, 5, and 6	plus E num $
plus E num	States 1, 5, and 2	plus E E $
plus E E	States 1, 5, and 6	E $
E $	States 1, 3, and 4	Start

Figure 6.12: Processing of plus plus num num num $ by the LR(0) machine in Figure 6.11.

Vervolledigen van de tabel: reduce entries toevoegen (algemeen schema)

```
procedure CompleteTable(Table, grammar)
    call ComputeLookahead( )
    foreach state ∈ Table do
        foreach rule ∈ Productions(grammar) do
            call TryRuleInState(state, rule)
    call AssertEntry(StartState, GoalSymbol, accept)
end
procedure AssertEntry(state, symbol, action)
    if Table[state][symbol] = error
    then   Table[state][symbol] ← action
    else
        call ReportConflict(Table[state][symbol], action)
end
```

(zie later)

(kijk na of *rule* in *state* aanleiding geeft tot reductie zie Fig 6.14)

Figure 6.13: Completing an LR(0) parse table.

procedure ComputeLookahead( )
    /$\star$ Reserved for the LALR(k) computation given in Section 6.5.2 $\star$/
end
procedure TryRuleInState(s, r)
    if $\text{LHS}(r) \rightarrow \text{RHS}(r) \bullet \in s$
    then
        foreach $\mathcal{X} \in (\Sigma \cup N)$ **do** **call** AssertEntry($s, \mathcal{X}$, reduce r)
end

Figure 6.14: LR(0) version of TryRuleInState.

<table>
<tr><th>State</th><th>num</th><th>plus</th><th>$</th><th>Start</th><th>E</th></tr>
<tr><td>0</td><td>$\boxed{2}$</td><td>$\boxed{1}$</td><td></td><td>accept</td><td>$\boxed{3}$</td></tr>
<tr><td>1</td><td>$\boxed{2}$</td><td>$\boxed{1}$</td><td></td><td></td><td>$\boxed{5}$</td></tr>
<tr><td>2</td><td colspan="5">reduce 3</td></tr>
<tr><td>3</td><td></td><td></td><td>$\boxed{4}$</td><td></td><td></td></tr>
<tr><td>4</td><td colspan="5">reduce 1</td></tr>
<tr><td>5</td><td>$\boxed{2}$</td><td>$\boxed{1}$</td><td></td><td></td><td>$\boxed{6}$</td></tr>
<tr><td>6</td><td colspan="5">reduce 2</td></tr>
</table>

Een hele rij

Figure 6.15: LR(0) parse table for the grammar in Figure 6.2.

Conflict Diagnose

De constructie kan mislukken om twee redenen:

- Zowel een shift- als een reduce- item in eenzelfde state = shift - reduce conflict
- Meer dan één reduce- item in een state = reduce - reduce conflict

Dergelijke “foute” states heten inadequate states

Gebruik van inadequate states om sentential forms te vinden die verschillende parse trees hebben

1	Start	→	E $
2	E	→	E plus E
3		|	num

State 0	Goto
Start → • E $	2
E → • E plus E	2
E → • num	1

State 1	Goto
E → num •	

State 2	Goto
E → E • plus E	3
Start → E • $	4

State 3	Goto
E → E plus • E	5
E → • E plus E	5
E → • num	1

State 4	Goto
Start → E $ •	

State 5	Goto
E → E plus E •	
E → E • plus E	3

Figure 6.16: An ambiguous expression grammar.

inadequate

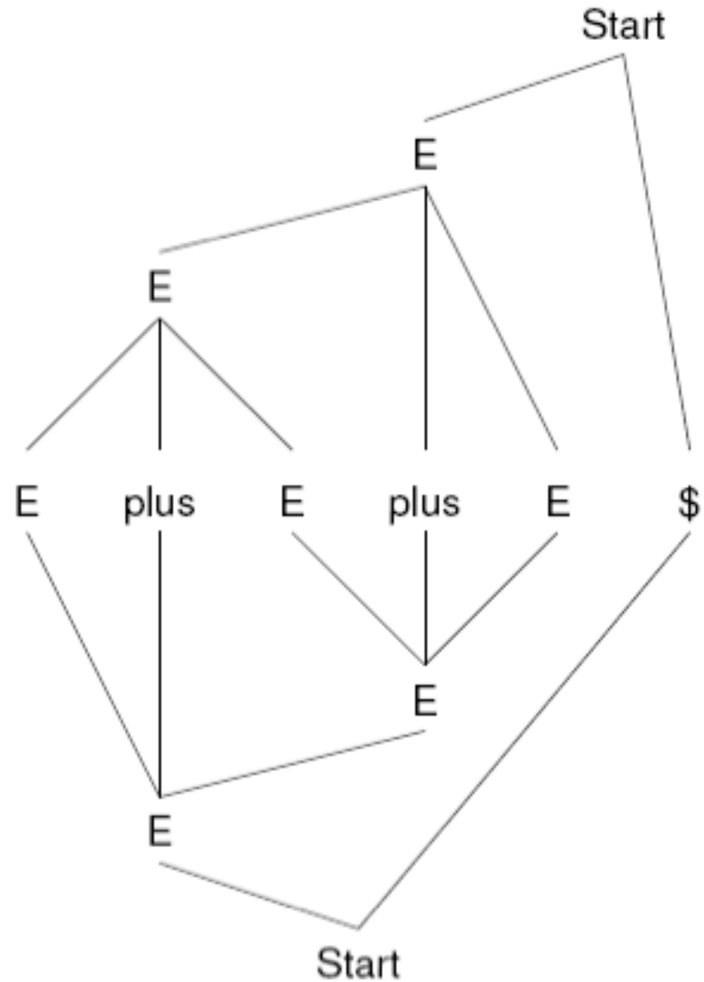

Figure 6.17: Two derivations for E plus E plus E \$. The parse tree on top favors reduction in State 5; the parse tree on bottom favors a shift.

1	Start	$\rightarrow$	E	\$
2	E	$\rightarrow$	E plus num	
3		|	num	

State 0	Goto
Start $\rightarrow$ $\bullet$ E \$	2
E $\rightarrow$ $\bullet$ E plus num	2
E $\rightarrow$ $\bullet$ num	1

State 1	Goto
E $\rightarrow$ num $\bullet$	

State 2	Goto
E $\rightarrow$ E $\bullet$ plus num	3
Start $\rightarrow$ E $\bullet$ \$	4

State 3	Goto
E $\rightarrow$ E plus $\bullet$ num	5

State 4	Goto
Start $\rightarrow$ E \$ $\bullet$	

State 5	Goto
E $\rightarrow$ E plus num $\bullet$	

Figure 6.18: Unambiguous grammar for infix sums and its LR(0) construction.

1	Start	$\rightarrow$	Exprs	\$
2	Exprs	$\rightarrow$	E	a
3		|	F	b
4	E	$\rightarrow$	E plus num	
5		|	num	
6	F	$\rightarrow$	F plus num	
7		|	num	

State 0	Goto
Start $\rightarrow \bullet$ Exprs \$	1
Exprs $\rightarrow \bullet$ E a	4
Exprs $\rightarrow \bullet$ F b	3
E $\rightarrow \bullet$ E plus num	4
E $\rightarrow \bullet$ num	2
F $\rightarrow \bullet$ F plus num	3
F $\rightarrow \bullet$ num	2

State 2	Goto
E $\rightarrow$ num $\bullet$	
F $\rightarrow$ num $\bullet$	

Figure 6.19: A grammar that is not LR(k).

Conflict resolutie – sterkere methoden

- SimpleLR(1), SLR(1)
- LALR(1)

Zelfde states als LR(0), maar met extra mechanisme om inadequate states af te handelen

- LR(k)

Voorbeeld: expressies met plus en times

$$
\begin{array}{llcl}
1 & \text{Start} & \rightarrow & \text{E \$} \\
2 & \text{E} & \rightarrow & \text{E plus num} \\
3 & & | & \text{E times num} \\
4 & & | & \text{num}
\end{array}
$$

LR(0), maar genereert niet de juiste parse trees

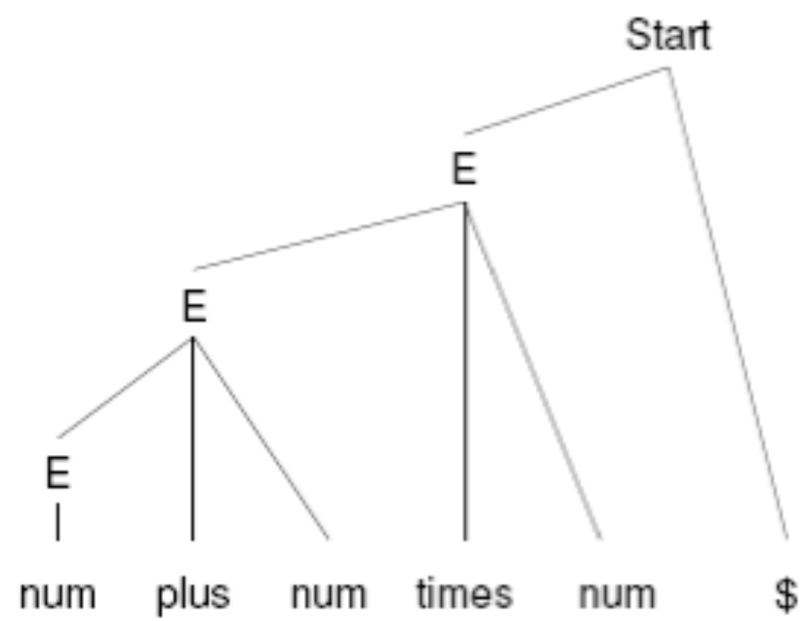

Figure 6.20: Expressions with sums and products.

Beter: expressie is een som van termen, een term is een product van factoren

1	Start	$\rightarrow$	E \$
2	E	$\rightarrow$	E plus T
3		|	T
4	T	$\rightarrow$	T times num
5		|	num

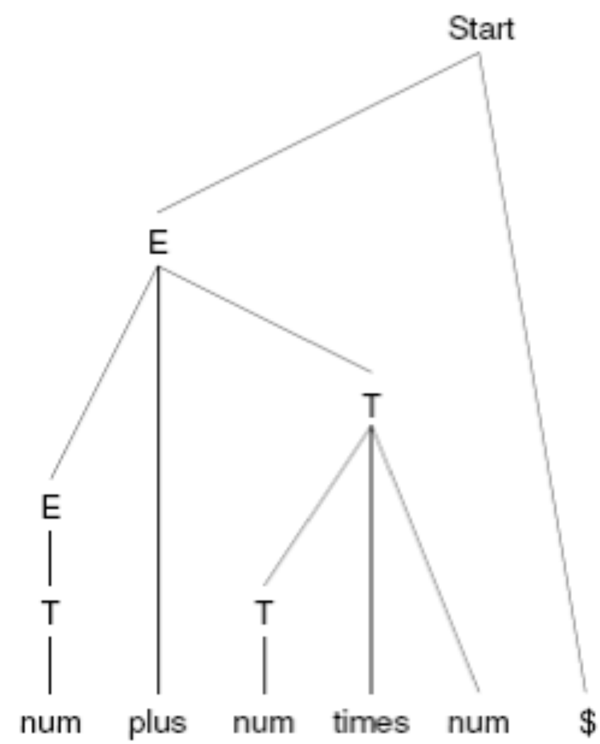

Figure 6.21: Grammar for sums of products.

State 0	Goto
Start → • E $	3
E → • E plus T	3
E → • T	1
T → • T times num	1
T → • num	2

State 1	Goto
E → T •	
T → T • times num	7

State 2	Goto
T → num •	

State 3	Goto
Start → E • $	5
E → E • plus T	4

inadequate

State 4	Goto
E → E plus • T	6
T → • T times num	6
T → • num	2

State 5	Goto
Start → E $ •	

State 6	Goto
E → E plus T •	
T → T • times num	7

State 7	Goto
T → T times • num	8

State 8	Goto
T → T times num •	

SLR(1): Voeg alleen een reduce actie toe als de Follow set van de reduce item past: bv. het item

$$E \rightarrow E \text{ plus } T\bullet$$

in state 6 leidt enkel tot een reduce actie als het in de input gevolgd wordt door een symbool van Follow(E), dus een plus of een $.

Op die manier voeren we een regel in die een eventueel conflict oplost.

In het algoritme voor het opstellen van de tabel:

procedure TRYRULEINSTATE(s, r)
    if LHS(r) $\rightarrow$ RHS(r) $\bullet \in s$
    then
        foreach $X \in$ Follow(LHS(r)) **do**
            call ASSERTENTRY(s, X, reduce r)
end

Figure 6.23: SLR(1) version of TRYRULEINSTATE.

Inadequate states 1 and 6 pose no problem

State	num	plus	times	$	Start	E	T
0	$\boxed{2}$				accept	$\boxed{3}$	$\boxed{1}$
1		3	$\boxed{7}$	3			
2		5	5	5			
3		$\boxed{4}$		$\boxed{5}$			
4	$\boxed{2}$						$\boxed{6}$
5				1			
6		2	$\boxed{7}$	2			
7	$\boxed{8}$						
8		4	4	4			

Figure 6.24: SLR(1) parse table for the grammar in Figure 6.21.

Tegenvoorbeeld

Genereert
a,ab,ac,xac

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A B
3		|	a c
4		|	x A c
5	A	$\rightarrow$	a
6	B	$\rightarrow$	b
7		|	λ

State 0	Goto
Start $\rightarrow$ • S \$	4
S $\rightarrow$ • A B	2
S $\rightarrow$ • a c	3
S $\rightarrow$ • x A c	1
A $\rightarrow$ • a	3

Figure 6.25: A grammar that is not SLR(k).

State 3	Goto
S $\rightarrow$ a • c	6
A $\rightarrow$ a •	

c is in Follow(A), c wijst op “reduce”, dus nog altijd geen oplossing

2 versies van A?
Niet erg leesbaar!

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A_1 B
3		|	a c
4		|	x A_2 c
5	A_1	$\rightarrow$	a
6	A_2	$\rightarrow$	a
7	B	$\rightarrow$	b
8		|	λ

State 0	Goto
Start $\rightarrow \bullet$ S \$	3
S $\rightarrow \bullet$ A_1 B	4
S $\rightarrow \bullet$ a c	2
S $\rightarrow \bullet$ x A_2 c	1
A_1 $\rightarrow \bullet$ a	2

State 2	Goto
S $\rightarrow$ a $\bullet$ c	8
A_1 $\rightarrow$ a $\bullet$	

Figure 6.26: An SLR(1) grammar for the language defined in Figure 6.25.

Een sterkere methode: LALR(1)

De SLR methode gebruikt de sets Follow(A), waar A de LHS is van het reduce item. Nochtans bevat dat item meer informatie dan enkel die LHS.
De LALR methode gebruikt lookahead informatie ($ItemFollow$) per item; deze wordt berekend door lookahead informatie te propageren met behulp van de zgn. propagation graph.

Gebruik van *ItemFollow*, voor koppels (state,item)

procedure TryRuleInState(s, r)
  if $\text{LHS}(r) \rightarrow \text{RHS}(r) \bullet \; \in s$
  then
    foreach $\mathcal{X} \in \Sigma$ **do**
      if $\mathcal{X} \in \mathit{ItemFollow}((s, \text{LHS}(r) \rightarrow \text{RHS}(r) \bullet))$
      then **call** AssertEntry($s, \mathcal{X}$, reduce r)
end

Figure 6.27: LALR(1) version of TryRuleInState.

Constructie van de *ItemFollow* sets en de propagation graph

De knopen zijn de items, een pijl van v naar w geeft aan dat de *ItemFollow* set van w die van v moet bevatten. De volgende gevallen doen zich voor:

- Voor elk item $A \rightarrow \alpha \bullet B\gamma$ in state s, moeten de symbolen van First(γ) toegevoegd worden aan de *ItemFollow* sets van de closure items $B \rightarrow \bullet\delta$ in state s (closure items: toegevoegd door de Closure operatie)

- De lookahead symbolen moeten ook gepropageerd worden naar de items die ontstaan door het verschuiven van the marker door δ, tot we $B \rightarrow \delta\bullet$ bereiken

- Voor elk item $A \rightarrow \alpha \bullet B\gamma$ in een state s zo dat uit γ het lege woord afgeleid kan worden, moet elk symbool dat op A kan volgen in state s ook op de closure items $B \rightarrow \bullet\delta$ in state s kunnen volgen

```
procedure ComputeLookahead( )
    call BuildItemPropGraph( )
    call EvalItemPropGraph( )
end
```

Build propagation graph nodes = items

```
procedure BuildItemPropGraph( )
    foreach s ∈ States do
        foreach item ∈ state do
            v ← Graph.AddVertex((s, item))                                  (24)
            ItemFollow(v) ← ∅
    foreach p ∈ ProductionsFor(Start) do
        ItemFollow((StartState, Start→ •RHS(p))) ← {$}                      (25)
    foreach s ∈ States do
        foreach A→α•Bγ ∈ s do                                               (26)
            v ← Graph.FindVertex((s, A→α•Bγ))
            call Graph.AddEdge(v, (Table[s][B], A→αB•γ))                   (27)
            foreach (w ← (s, B→ •δ)) ∈ Graph.Vertices do
                ItemFollow(w) ← ItemFollow(w) ∪ First(γ)                    (28)
                if AllDeriveEmpty(γ)                                        (29)
                then call Graph.AddEdge(v, w)
end
```

Compute sets ItemFollow by propagating lookahead info

```
procedure EvalItemPropGraph( )
    repeat                                                                  (30)
        changed ← false
        foreach (v, w) ∈ Graph.Edges do
            old ← ItemFollow(w)
            ItemFollow(w) ← ItemFollow(w) ∪ ItemFollow(v)
            if ItemFollow(w) ≠ old
            then changed ← true
    until not changed
end
```

Figure 6.28: LALR(1) version of ComputeLookahead.

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	A B
3		|	a c
4		|	x A c
5	A	$\rightarrow$	a
6	B	$\rightarrow$	b
7		|	λ

State		LR(0) Item	Goto State	Prop Edges Placed by Step 27	Prop Edges Placed by Step 29	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* 28
0	1	Start→ • S \$	4	13		\$	2,3,4
	2	S→ • A B	2	8	5	b	5
	3	S→ • a c	3	11			
	4	S→ • x A c	1	6			
	5	A→ • a	3	12			
1	6	S→x • A c	9	18		c	7
	7	A→ • a	10	19			
2	8	S→A • B	8	17	9,10		
	9	B→ • b	7	16			
	10	B→ •					
3	11	S→a • c	6	15			
	12	A→a •					
4	13	Start→S • \$	5	14			
5	14	Start→S \$ •					
6	15	S→a c •					
7	16	B→b •					
8	17	S→A B •					
9	18	S→x A • c	11	20			
10	19	A→a •					
11	20	S→x A c •					

Figure 6.29: LALR(1) analysis of the grammar in Figure 6.25.

Item	Prop To	Initial	Pass 1
1	13	$	
2	5,8	$	
3	11	$	
4	6	$	
5	12	b	$
6	18		$
7	19	c	
8	9,10,17		$
9	16		$
10			$
11	15		$
12			b $
13	14		$
14			$
15			$
16			$
17			$
18	20		$
19			c
20			$

Figure 6.30: Iterations for LALR(1) follow sets.

1	Start	$\rightarrow$	S \$
2	S	$\rightarrow$	x C1 y1 Cn yn
3		|	A1
4	A1	$\rightarrow$	b1 C1
5		|	a1
6	An	$\rightarrow$	bn Cn
7		|	an
8	C1	$\rightarrow$	An
9	Cn	$\rightarrow$	A1

Figure 6.31: LALR(1) analysis: grammar.

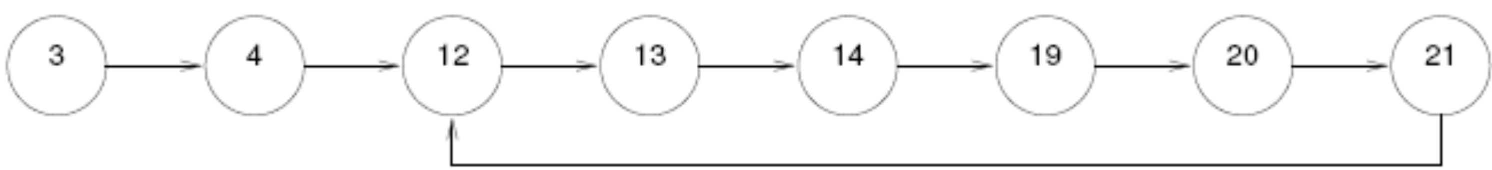

Figure 6.32: Embedded propagation subgraph.

State		LR(0) Item	Goto State	Prop Edges Placed by Step 27	Prop Edges Placed by Step 29	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* 28
0	1	Start→ • S $	3	11		$	2,3
	2	S→ • x C1 y1 Cn yn	1	6			
	3	S→ • A1	5	16	4,5		
	4	A1→ • b1 C1	4	12			
	5	A1→ • a1	2	10			
1	6	S→x • C1 y1 Cn yn	13	27		y1	7
	7	C1→ • An	7	18	8,9		
	8	An→ • bn Cn	8	19			
	9	An→ • an	9	23			
2	10	A1→a1 •					
3	11	Start→S • $	12	26			
4	12	A1→b1 • C1	6	17	13		
	13	C1→ • An	7	18	14,15		
	14	An→ • bn Cn	8	19			
	15	An→ • an	9	23			
5	16	S→A1 •					
6	17	A1→b1 C1 •					
7	18	C1→An •					
8	19	An→bn • Cn	10	24	20		
	20	Cn→ • A1	11	25	21,22		
	21	A1→ • b1 C1	4	12			
	22	A1→ • a1	2	10			
9	23	An→an •					
10	24	An→bn Cn •					
11	25	Cn→A1 •					
12	26	Start→S $ •					
13	27	S→x C1 • y1 Cn yn	14	28			
14	28	S→x C1 y1 • Cn yn	15	32		yn	29
	29	Cn→ • A1	11	25	30,31		
	30	A1→ • b1 C1	4	12			
	31	A1→ • a1	2	10			
15	32	S→x C1 y1 Cn • yn	16	33			
16	33	S→x C1 y1 Cn yn •					

Figure 6.33: LALR(1) analysis for the grammar in Figure 6.31.

Item	Prop To	Initial	Pass 1	Pass 2
1	11	$		
2	6	$		
3	4,5,16	$		
4	12		$	
5	10		$	
6	27		$	
7	8,9,18	y1		
8	19		y1	
9	23		y1	
10			$ y1 yn	
11	26		$	
12	13,17		$ y1 yn	
13	14,15,18		$	y1 yn
14	19		$	y1 yn
15	23		$	y1 yn
16			$	
17			$	y1 yn
18			y1 $	yn
19	20,24		y1 $	yn
20	21,22,25		y1 $	yn
21	12		y1 $	yn
22	10		y1 $	yn
23			y1 $	yn
24			y1 $	yn
25			y1 $ yn	
26			$	
27	28		$	
28	32		$	
29	25,30,31	yn		
30	12		yn	
31	10		yn	
32	33		$	
33			$	

Figure 6.34: Iterations for LALR(1) follow sets.

Nog sterkere methode:
LR(k)

Nu heeft elk item een lookahead (meer states!)

1	Start	$\rightarrow$	S $
2	S	$\rightarrow$	lp M rp
3		|	lb M rb
4		|	lp U rb
5		|	lb U rp
6	M	$\rightarrow$	expr
7	U	$\rightarrow$	expr

Figure 6.35: A grammar that is not LALR(k).

State 0	Goto
Start $\to \bullet$ S \$	1
S $\to \bullet$ lp M rp	2
S $\to \bullet$ lb M rb	3
S $\to \bullet$ lp U rb	2
S $\to \bullet$ lb U rp	3

State 1	Goto
Start $\to$ S $\bullet$ \$	13

State 2	Goto
S $\to$ lp $\bullet$ M rp	10
S $\to$ lp $\bullet$ U rb	9
M $\to$ $\bullet$ expr	6
U $\to$ $\bullet$ expr	6

State 3	Goto
S $\to$ lb $\bullet$ M rb	5
S $\to$ lb $\bullet$ U rp	4
M $\to$ $\bullet$ expr	6
U $\to$ $\bullet$ expr	6

State 4	Goto
S $\to$ lb U $\bullet$ rp	8

State 5	Goto
S $\to$ lb M $\bullet$ rb	7

State 6	Goto
M $\to$ expr $\bullet$	
U $\to$ expr $\bullet$	

State 7	Goto
S $\to$ lb M rb $\bullet$	

State 8	Goto
S $\to$ lb U rp $\bullet$	

State 9	Goto
S $\to$ lp U $\bullet$ rb	12

State 10	Goto
S $\to$ lp M $\bullet$ rp	11

State 11	Goto
S $\to$ lp M rp $\bullet$	

State 12	Goto
S $\to$ lp U rb $\bullet$	

State 13	Goto
Start $\to$ S \$ $\bullet$	

Figure 6.36: LR(0) construction.

State		LR(0) Item	Goto State	Prop Edges Placed by Step 27	Prop Edges Placed by Step 29	Initialize *ItemFollow* First(γ)	Initialize *ItemFollow* 28
0	1	Start→ • S $	1	??		$	2,3,4,5
	2	S→ • lp M rp	2	6			
	3	S→ • lb M rb	3	10			
	4	S→ • lp U rb	2	7			
	5	S→ • lb U rp	3	11			
2	6	S→lp • M rp	10	??		rp	8
	7	S→lp • U rb	9	??		rb	9
	8	M→ • expr	6	14			
	9	U→ • expr	6	15			
3	10	S→lb • M rb	5	??		rb	12
	11	S→lb • U rp	4	??		rp	13
	12	M→ • expr	6	14			
	13	U→ • expr	6	15			
6	14	M→expr •					
	15	U→expr •					

Figure 6.37: Partial LALR(1) analysis. Notice the propagation of rp and rb to Item 14 from Items 8 and 12, respectively. Item 15 suffers a similar fate from Items 13 and 9. This leads to a reduce/reduce conflict between M→expr and U→expr on rp and rb in State 6.

Marker 7: We initialize *StartItems* by including LR(1) items that have \$ as the follow symbol:

$StartItems \leftarrow \{[\,\mathsf{Start} \rightarrow \bullet\, \mathrm{RHS}(p)\,,\$\,] \mid p \in \mathrm{P} \qquad \mathrm{F} \quad (\mathsf{Start})\}$

Marker 13: We augment the LR(0) item so that A D returns the appropriate LR(1) items:

return $(\{[\,\mathsf{A} \rightarrow \alpha \mathcal{X} \bullet \beta\,,\mathrm{a}\,] \mid [\,\mathsf{A} \rightarrow \alpha \bullet \mathcal{X}\beta\,,\mathrm{a}\,] \in state\})$

Marker 15: This entire loop is replaced by the following:

foreach $[\,\mathsf{A} \rightarrow \alpha \bullet \mathsf{B}\gamma\,,\mathsf{a}\,] \in ans$ **do**
  foreach $p \in \mathrm{P} \qquad \mathrm{F} \quad (B)$ **do**
    foreach $b \in \mathsf{First}(\gamma \mathsf{a})$ **do** 31
      $ans \leftarrow ans \cup \{[\,\mathsf{B} \rightarrow \bullet\, \mathrm{RHS}(p)\,,b\,]\}$

Figure 6.38: Modifications to Figures 6.9 and 6.10 to obtain an LR(1) parser

procedure TryRuleInState(s, r)
  if $[\,\mathrm{LHS}(r) \rightarrow \mathrm{RHS}(r) \bullet\ ,w\,] \in s$
  then call AssertEntry$(s, w,$ reduce r$)$
end

Figure 6.39: LR(1) version of TryRuleInState.

State 0	Goto
[Start → • S $, $]	1
[S → • lp M rp , $]	2
[S → • lb M rb , $]	3
[S → • lp U rb , $]	2
[S → • lb U rp , $]	3

State 1	Goto
[Start → S • $, $]	13

State 2	Goto
[S → lp • M rp , $]	10
[S → lp • U rb , $]	9
[M → • expr , rp]	6
[U → • expr , rb]	6

State 3	Goto
[S → lb • M rb , $]	5
[S → lb • U rp , $]	4
[M → • expr , rb]	14
[U → • expr , rp]	14

State 4	Goto
[S → lb U • rp , $]	8

State 5	Goto
[S → lb M • rb , $]	7

State 6	Goto
[M → expr • , rp]	
[U → expr • , rb]	

State 7	Goto
[S → lb M rb • , $]	

State 8	Goto
[S → lb U rp • , $]	

State 9	Goto
[S → lp U • rb , $]	12

State 10	Goto
[S → lp M • rp , $]	11

State 11	Goto
[S → lp M rp • , $]	

State 12	Goto
[S → lp U rb • , $]	

State 13	Goto
[Start → S $ • , $]	

State 14	Goto
[M → expr • , rb]	
[U → expr • , rp]	

Figure 6.40: LR(1) construction.

Lookahead as part of the items